Use R!

Deborah Nolan
Duncan Temple Lang

XML and Web Technologies for Data Sciences with R

Springer

Use R!

Series Editors:
Robert Gentleman Kurt Hornik Giovanni Parmigiani

For further volumes:
http://www.springer.com/series/6991

Deborah Nolan • Duncan Temple Lang

XML and Web Technologies for Data Sciences with R

Springer

Deborah Nolan
Department of Statistics
University of California
Berkeley, CA, USA

Duncan Temple Lang
Department of Statistics
University of California
Davis, CA, USA

ISSN 2197-5736 ISSN 2197-5744 (electronic)
ISBN 978-1-4614-7899-7 ISBN 978-1-4614-7900-0 (eBook)
DOI 10.1007/978-1-4614-7900-0
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2013954669

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Doris and Harriet,
and my teacher Winifred Asprey.

— Deborah

To Zoë and Suzana,
and my family farther away.

— Duncan

Preface

There has been a major change over the last decade in many aspects related to working with data and doing scientific work. As the bloggers at simplystatistics.org put it, this is a "new era where data are abundant and statisticians are scientists." The growth of the Web and the numerous data technologies that it has fostered have changed what we can do, and how we do it. It has helped to broaden the focus of statistics from mostly the modeling stage to all stages of data science: finding relevant data, accessing data, reading and transforming data, visualizing the data in rich ways, modeling, and presenting the results and conclusions with compelling, interactive displays. In this book, we describe many technologies and approaches related to all of these additional stages in the data scientist's work flow. We focus on important and fundamental technologies that are likely to stand the test of time, explain their roles, and compare them with other technologies and approaches. Importantly, we also illustrate how to work with them within the *R* programming environment through numerous comprehensive examples and case studies.

Our focus is on technologies related to the different stages of the data analysis work flow. We can now access so much data from many different sources, in different formats and via many different techniques. We can use formal Web services and application programming interfaces (APIs) or simply scrape data from human-readable Web pages. The data may come in some dialect of *XML*, *HTML* or as a *JSON* document or some other self-describing format. We will explore how we both make Web requests—both simple and sophisticated—and transform the content into data in *R*.

While we can use *R*'s rich graphical functionality, we can also visualize data in new ways with a collection of interactive plots and text in a Web browser, or use applications such as Google Earth to display spatio-temporal data and models. For these, we can use *R* to create these "external" displays as *JavaScript*, *SVG*, or *KML* documents. We can export data from *R* as *JSON* for use in *JavaScript* code to connect the data analysis with the visualization of the results.

Technologies such as Web services and requests, *XML*, and *JSON* are widely used in contexts other than the Web. All of the desktop office suites that provide spreadsheets, word processors, and presentation applications use *XML* to represent the contents of those documents. We also interact with online office suites such as GoogleDocs via authenticated Web requests using *OAuth* and then digest the *XML* content, and we also communicate with newer *NoSQL* databases and other applications using Web service technologies locally. Therefore, the technologies we discuss in this book are, in many ways, fundamental infrastructure for modern computing with data.

One of the important concepts motivating this book is that data analysts and modern statisticians increasingly are working on multi-disciplinary projects or posing their own questions. They need to access data and find auxiliary data, and wrangle them into a usable form. They expect to create rich and informative graphical displays in Web browsers, dashboards, or dynamic reports, and should

be familiar with how to do this. They should also be able to understand the core technologies and know when and how to leverage them. We like to think that we foresaw this evolution when we first started developing the *R* interfaces for these technologies, back in December of 1999—a different century! We are very glad to see the rise of data science as the broadening of statistics and also see the technologies we cover in this book growing in importance in the practice of data analysis. This book is aimed at this new breed of scientist who thinks of statistics as spanning all the stages of the data work flow, and who wants to be involved in all of them.

The book describes a mixture of general infrastructure tools for *R* programmers, e.g., parse an *XML* document or make a Web request, and end-user tools *R* users can employ directly to perform a high-level task, e.g., read an *HTML* table as an *R* data frame or access the contents of a document from Google Docs. The aim in each chapter is to introduce to the reader an important technology and illustrate how it is relevant for modern statisticians. We introduce each of these technologies, provide a succinct overview of the essential concepts and elements, and describe *R*-based tools for working with the particular technology, including numerous packages developed as part of the Omegahat project (www.omegahat.org). We illustrate each technology with high-level, reasonably simple examples. We also provide more in-depth examples that put the different pieces in context and compare possible approaches. Combining these technologies into a book allows us to incrementally build on the descriptions of the fundamentals.

We have organized this book into three parts. The first part introduces the reader to *XML* and *JSON*. It assumes no prior knowledge of these data formats. It includes discussions of the tools in *R* for reading *XML* and *JSON* files and extracting data from them. It describes various different approaches and computational models for reading *XML* and explains why they are all useful in different circumstances, dealing with documents ranging from small to enormous in size. While we may spend most of our time processing content that was generated by other researchers and Web sites, etc., we often want to create our own *XML* and *JSON* documents, e.g., for displaying plots in Web browsers or Google Earth or upload to Google Docs. We also address this topic in the first part of the book.

Our focus in the second part is on how to obtain data over the Web from remote sites, services, and APIs. These methods include accessing data directly from *HTML* on static Web pages and also from dynamic *HTML* documents via *HTML* forms. We also explore using more structured access via Web services such as *SOAP* (Simple Object Access Protocol), *REST* (Representational State Transfer), and XML-RPC (*XML* Remote Procedure Call). Additionally, we may need to obtain data locally from an application running as a server, such as a *NoSQL* database. We will introduce and discuss the common technologies used for these methods. We will also explore the common approaches to authorization and authentication when accessing data from a remote site. Some of these are simple, e.g., passwords sent in a request, and others are more industrial-strength methods using technologies such as *OAuth*.

Accessing data from a remote host involves the client (*R*) communicating with that host. *R* already has several functions to do this. These are sufficient for many common tasks, but as we try to use *R* for more sophisticated network communications and interactions, we need richer tools and a richer, more flexible interface to use those tools. We will discuss the RCurl package that gives us these more general facilities. This infrastructure allows us now, and in the future, to harness new technologies built on *HTTP* and other protocols.

The final part of this book presents four in-depth examples that cover: 1) *XML* Schema, a grammar for describing the rules of an *XML* grammar; 2) *SpreadsheetML*, an Office Open *XML* vocabulary for spreadsheets and report writing; 3) Scalable Vector Graphics (*SVG*) for creating interactive graphical displays; and 4) Keyhole Markup Language (*KML*) for displaying geographic data on Google Earth and Maps. In each chapter, we describe the underlying computational model we have developed to work with these *XML* formats to, e.g., programmatically define *R* data structures, create reports, and

design new graphics formats—several important aspects of data science. We hope that these examples serve as case studies for how someone might create an *R* interface to other *XML* vocabularies.

Throughout the book, we present several dozen *R* packages for data scientists to work with Web-related data technologies. Some of these packages are essentially complete and can be useful immediately to *R* users. Others are more infrastructural and/or speculative to illustrate what might be useful and how it might be done. We hope these packages will excite readers to think about how these technologies might be used to do new things and that some readers will want to extend the software to realize new possibilities. For those readers, we have provided ideas for enhancements at the end of many of the chapters and we welcome your contributions and suggestions. If you are interested in extending our work and developing new applications, we encourage you to read about writing *R* packages [145] and useful, slightly advanced aspects of the *R* language in texts such as [30, 56, 108].

We make no claim that *R* is the best tool for working with *XML*, *JSON*, and other Web technologies, or that people should use it in preference to other languages such as *Python*, *Java*, *PERL*, and *JavaScript*. However, we are suggesting that statisticians already using *R* should be able to work with these new technologies without an entire shift in their programming language and environment.

Code from the examples in this book and additional materials are available via the Web page for the book – `http://rxmlwebtech.org`.

Typographic Conventions

In this book, we discuss numerous different programming languages, and also how we use the languages in combination with one another. We illustrate code in these different languages both inline in the text and as separate code blocks. While the context should clearly indicate the language used, we use color to distinguish the languages and different fonts to distinguish the elements of a language, e.g., to differentiate a function name from a package name in *R*.

The following is a collection of example formats for these languages.

R

A block of *R* code appears as

```
doc = as(filename, "XMLInternalDocument")
xpathSApply(doc, "//section/title/", xmlValue)
```

Output from *R* commands appears as

```
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
```

And, *R* expressions appear inline as `getNodeSet(x, "//js:*", "js")`.

References to *R* function names appear in the form xmlParse(), and names of function parameters look like *filename*. Names of *R* variables appear as `variable`, names of *S4* classes in *R* appear as `XSLStyleSheet`, and *S3* class names as `XMLInternalDocument`. Named elements in an *R* list appear as `name` and slots in an *S4* object as *slot*. Options that we can query and set in *R* via the options() function appear as *warning.length*. Also, special values in *R* appear as `TRUE`, `FALSE`, `NULL`, `NA`, `NaN`, etc. Formulas are shown as `y ~ a + b | time`. Keywords in the language appear as `while`. We display regular *R* package names as `lattice`, Omegahat packages (i.e., packages distributed from `http://www.omegahat.org`) as RCurl, and **Bioconductor** packages as, e.g., graph.

File Names and Directories
We render file names as `filename`, file extensions as **xlsx**, and directories with a trailing / to differentiate them from file names, e.g., `/home/frank/`.

XML
The content for any of the various *XML* vocabularies (e.g., *XHTML*, *KML*, *SVG*, *SpreadsheetML*) is displayed as

```
<?xml version="1.0" encoding="UTF-8"?>
<snapshot>
 <header>
  <total>576803</total>
  <page>1</page>
  <date>2010-01-29T20:00:23Z</date>
  <page_size>1000</page_size>
 </header>
 ...
</snapshot>
```

Inline *XML* content is displayed as `<xs:element name="ARIMA" />`. Element names are shown as `<r:plot>`, while namespace prefixes are rendered as `bioc`. Attributes for an *XML* node are displayed as *href*. *XML* entities appear as *<* or *&*.

XPath
The code blocks for *XPath* or non-inlined expressions are displayed as

```
/Envelope/Cube/Cube
```

We see an expression inline as `//a[@href]` and names of *XPath* functions appear as `starts-with()`. An *XPath* axis is rendered as **ancestor**, and we show node test expressions as **comment**. We display the literal logical values in *XPath* as `true`.

JSON
Generic *JSON* content appears as

```
{
 "values": [ 1, 2, 3, 4, 5 ],
 "names": [ "ABC", "DEF", "GHI", "JKL", "MNO" ]
}
```

Inline *JSON* content is shown as arrays with *[1, 2, null, 4]*. Fields in *JSON* content appear as `copyright`. The *JSON* literal logical values are displayed as *true* and *false*. Similarly, the null value appears as *null*. *JSON* number and string values are shown as *3.1415* and *string*.

JavaScript
a *JavaScript* code block appears as

```
function highlightEdges(evt, row, color)
{
  var labels = edgeTable[row];
  var reset = false;

  if(typeof color == 'undefined') {
    color = "black";
```

```
    reset = true;
  }
}
```

References to *JavaScript* function names appear in the form *getElementById()*, and names of variables appear as *neighbors*. Fields of an object are shown as `myVar` while methods of an object appear as *getValue*. We see inline *JavaScript* expressions as `x = slider.getValue()`.

RCurl and `libcurl` Options
The names of curl options appear as, for example, *verbose*.

HTTP
We refer to different aspects of *HTTP* in various chapters. We represent *HTTP* operations, e.g., **GET** and **DELETE**. We also refer to elements/fields of the *HTTP* header with *ContentType*. We show some or all of the header information from a request or response with

```
GET /folder/docName HTTP/1.1
```

Shell
A block of shell (sh, bash, csh, tcsh, etc.) code appears as

```
xmllint myDoc.xml
```

Output from shell commands is displayed as

```
149 book.xml
426 springerLatex.xsl
575 total
```

When we refer to a shell command/executable, we see it as `xmllint`. Shell variables are displayed as *XML_CATALOG_FILES*. Options or flags for shell commands are rendered as *–noout*.

Acknowledgements

Many have contributed to this project by reading chapters, suggesting interesting ideas to pursue, and expressing support for our work. We thank them all, including those who participated in our NSF-sponsored Computing in Statistics Workshops. (The material in this book is based upon work supported by the National Science Foundation under Grant No. 0618865.) We especially mention: Gabe Becker, Carl Boettiger, Vince Carey, John Chambers, Robert Gentleman, Jeff Gentry, Tammy Greasby, Kurt Hornik, Ross Ihaka, Cari Kaufman, Bitao Liu, Paul Murrel, Balasubramanian Narasimhan, Karthik Ram, Steve Sein, and Phil Spector. We are also grateful to Daniel Veillard for his *XML* parser and *XSL* toolkit, `libxml2` and `liblibxslt`, and Jonathan Wallace for his *JSON* parser.

We wish to acknowledge the users of our software who have provided useful examples, asked questions that turned into examples, and submitted bug reports. Their contributions have served as inspiration for many of the examples in this book. Open source software improves because of the community of contributors. In that vein, we express our appreciation to the *R* Core for maintaining a valuable infrastructure in which we can explore these technologies.

We thank the students who have taken our computing classes over the past eight years (STAT 133 at Berkeley and STAT 141 and 242 at Davis). They have given us valuable feedback as they explored ideas and used our software when working on their projects and assignments. We also thank the

computing support staff in our departments: Larry Tai at Davis; and Rick Kawin, Ryan Lovett, and the rest of the Statistical Computing Facility staff at Berkeley.

We are grateful to our original editor John Kimmel for his continual encouragement and the new folks at Springer—Jon Gurstelle, Marc Strauss and Hannah Bracken—who supported this project to completion.

Finally, we give special thanks to our families for their patience and encouragement throughout this project.

Contents

List of Examples

Part I
Data Formats: *XML* and *JSON*

Overview

The initial topic in this book is a brief introduction to both *XML* and *JSON*. We start with a practical hands-on approach by introducing some of the very high-level functions that we commonly use to read data from *HTML*, *XML*, and *JSON* documents. If you have tasks of this nature, you can hopefully read the first chapter and solve that problem. The remainder of this part of the book explains the details of both *XML* and *JSON* and how to work with these formats in *R*.

While the first chapter is a very high-level, detail-free introduction to *R* functionality, we follow it with a comprehensive introduction to *XML*. For readers who are not familiar with *XML*, this explains all of the concepts and elements of *XML*. For readers who already know the structure of *XML*, the chapter also explains some of the less common aspects such as namespaces, schema, and *DTD*s (Document Type Definitions). We also explain some of the potential and motivation for using *XML* and illustrate these with some examples of *XML* in action.

The next three chapters deal with how we extract data from *XML* documents in *R*. In Chapter 3 we start by introducing the core *R* functionality for parsing documents and working with *XML* trees and nodes. While we can process any *XML* document with these alone, the *XPath* language is a very powerful mechanism for locating particular nodes within a tree and so simplifies extracting data. We discuss *XPath* in Chapter 4. In Chapter 5, we go beyond the details of different functions and the "how-to's" of using them and discuss different strategies and approaches for extracting data from *XML* documents. In practice, we combine *XPath* and the functions for working with nodes when parsing a document. However, there are different techniques even within this hybrid approach. In this chapter, we also introduce the *SAX* approach for dealing with very large or streaming *XML* documents.

Having discussed how to read *XML* content in *R*, we turn to creating *XML* content in *R* so that we can create and use the documents in other applications such as Google Earth, Web service requests, spreadsheet and word processing software. Chapter 6 introduces the approaches and functions for creating *XML* from data in *R*.

The final chapter in this part of the book introduces the *JSON* format and the functions for both reading and writing *JSON* within *R*. *JSON* is much simpler than *XML* and we discuss the relative advantages and disadvantages of the two formats. *JSON* is not extensible in the same way that *XML* is, and it also has fewer concepts. As a result, we can cover all aspects of reading and creating *JSON* content in *R* in a single chapter, along with several examples.

Chapter 1
Getting Started with *XML* and *JSON*

Abstract The goal of this chapter is to provide a rapid introduction to a few high-level functions available to *R* users for parsing *XML* and *JSON* content. In many cases, these functions (readHTMLTable(), xmlToList(), xmlToDataFrame(), and fromJSON()) are all that you will need to read *XML*- or *JSON*- formatted data directly into an *R* *list* or `dataframe`. One of the purposes of this chapter is to introduce many of the functions you need for common applications for scraping data from Web pages, reading data from files, and working with *XML* and *JSON* data from Web services. We also want to give you a sense of the possibilities and entice you to learn more about these data formats.

1.1 Introduction

The eXtensible Markup Language (*XML*) [10] and *JavaScript* Object Notation (*JSON*) [3] are widely used on the Web to create Web pages and interactive graphics, display geographical data on, e.g., Google Earth, and transfer data between applications in an application-independent format. Being able to work with these data formats allows us to quickly and easily access and gather data from many different sources and present them in extraordinary new ways. It is exciting and relatively easy to get started gathering data in *R* [5] from Web pages and Web services and local *JSON* or *XML* files. Rather than begin by discussing details of the *XML* and *JSON* formats, we delegate this to the next chapters, and instead, jump in and learn about a few high-level functions that are available in the XML package [8] to work with *XML* content and the single function needed to import *JSON* found in the RJSONIO package [7]. These are often all we need, especially for working with *JSON* content. We hope to introduce readers to the tools needed to get them started on common tasks.

1.2 Reading Data from *HTML* Tables

We start with *HTML*, an *XML*-like vocabulary. It is quite common to find data that are available on a Web page, typically displayed in a table or a list. For example, Wikipedia [9] has a page giving the population counts for each country in the world available at `http://en.wikipedia.org/wiki/Country_population`. A screenshot of the table is shown in Figure 1.1. Tables such as this one are typically rendered to make it easy for humans to view. However, it is not necessarily easy to read such a table into a data analysis environment such as *R*. We can sometimes cut-and-paste the

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_1,

data into a spreadsheet and then export it from there. This approach is limited, awkward, and neither reproducible nor verifiable. Instead, we want to be able to read the data directly into the *R* environment in the same way we use the functions read.csv() and read.table(). At times we are lucky and the page is suitably formatted so we can use the readHTMLTable() function in the XML package [8] to do exactly this. The next example demonstrates this approach.

Figure 1.1: Wikipedia Table of Country Populations. This *HTML* table of country populations is one of five tables embedded in the Web page. With readHTMLTable(), the country populations can be extracted easily from the table of interest. This screenshot of the Wikipedia Web page `http://en.wikipedia.org/wiki/Country_population` was captured in September, 2011.

Example 1-1 Extracting Country Populations from a Wikipedia HTML Table

At its simplest, we pass readHTMLTable() either the *URL* or name of a local *HTML* file and it reads the data in the table within the document into a data frame. For the Wikipedia page, we use

```
u = "http://en.wikipedia.org/wiki/Country_population"
tbls = readHTMLTable(u)
```

This *HTML* document, like many documents on the Web, contains several tables that are used to format ads and other parts of the Web page. This can make it difficult to find the data we want. Fortunately, readHTMLTable(), by default, returns all the tables in the document so we can examine them to identify the one we want. For example, we can look at the number of rows in each table with

```
sapply(tbls, nrow)
```

```
 toc NULL NULL NULL NULL
   1  226    1    1   12
```

We see the one large table in position 2 and now know that is where we will find the population data. If we had known we wanted the second table before calling readHTMLTable(), we could have retrieved just that table by passing readHTMLTable() this information as follows:

```
pop = readHTMLTable(u, which = 2)
```

The function's argument *colClasses* also allows us to specify the classes/types for the columns and a function will convert the content to numbers, percentages, factors, etc. This is similar to read.table(), but is slightly more general, allowing transformation of data such as formatted number, currency and percentage strings, i.e., containing a comma to separate the digits, preceded by a $, or having a % suffix.

How does readHTMLTable() do its job? It first parses the *HTML* document using the htmlParse() function in the XML package. Then, it finds all of the `<table>` nodes in the document. Below is a simplified version of the *HTML* table for the country populations found in Wikipedia,

```
<table>
 <tbody>
  <tr>
   <th>Rank</th> <th>Country / Territory</th>
   <th>Population</th> <th>Date of estimate</th>
   <th> % of World population</th> <th>Source</th>
  </tr>
  <tr>
   <td>-</td> <td align="left"><b>World</b></td>
   <td>6,965,300,000</td> <td>September 30, 2011</td>
   <td>100%</td>
   <td>US Census Bureau's World Population Clock</td>
  </tr>
  <tr>
   <td>1</td>
   <td><img src="22px-Flag_of_China.png"> 
     China, People's Republic of</td>
   <td>1,339,724,852</td>
   <td>November 1, 2010</td>
   <td>19.23%</td> <td>2010 China Census</td>
  </tr>
  <tr>
   <td>2</td>
   <td>
     <img src="22px-Flag_of_India.png">  India
   </td>
   <td>1,210,193,422</td> <td>March 1, 2011</td>
   <td>17.37%</td>
   <td>Provisional 2011 Indian Census result</td>
 </tr>
...
 </tbody>
</table>
```

For each <table> node, readHTMLTable() loops over the row nodes (<tr>) and processes each cell in the row. These cells are either <th> or <td> elements for header or regular data, respectively. The data are typically in <td> elements, and we simply extract the contents of each cell as a string. Later, we examine the values for each column of the table to determine if we can coerce them to a common type other than strings, e.g., numbers. If the caller supplied a value for the *colClasses* parameter, we use that. The <th> elements in a table typically indicate column headers, and we use those for variable names when appropriate. That's the basic sequence of steps. This is all accomplished with *XPath* [6] queries via calls to xpathApply(), which is the topic of Chapter 4, and then regular *R* functions to process the string values.

1.3 Reading Data from *XML*-formatted Documents

We next turn to reading data from *XML* documents. *XML* documents are very general and can be used to describe very complex data structures. However, many *XML* documents are quite simple. As an example, we consider the lender information provided by Kiva [4], a nonprofit organization that provides microloans to individuals in developing countries by connecting them with people around the world who want to loan money for such activities. Kiva makes information about lenders and loans available via a Web service and also provides a "dump" of the entire database in both *XML* and *JSON* formats. These data include characteristics of the lenders and also of the loans, e.g., the amount and purpose of the loan, which were paid back and which were not, and when. The data are available from http://build.kiva.org.

Looking at one of the documents describing lenders below, we see the basic structure. In addition, the graphical representation of the document in Figure 1.2 makes clear the hierarchical structure of the document. We are interested in the information about each lender. A <lender> has a <lender_id>, a <name>, information about a picture of the lender, and a location given by the <whereabouts> node and <country_code> nodes. We also have information about the number of loans the individual has provided (<loan_count>), the short label for his or her stated occupation and a more comprehensive description of it (in <occupation> and <occupation_info>), and why the lender participates in Kiva (<loan_because>).

```
<?xml version="1.0" encoding="UTF-8"?>
<snapshot>
 <header>
  <total>576803</total>
  <page>1</page>
  <date>2010-01-29T20:00:23Z</date>
  <page_size>1000</page_size>
 </header>
 <lenders type="list">
  <lender>
   <lender_id>matt</lender_id>
   <name>Matt</name>
   <image>
    <id>12829</id>
    <template_id>1</template_id>
   </image>
   <whereabouts>San Francisco CA</whereabouts>
```

```
   <country_code>US</country_code>
   <uid>matt</uid>
   <member_since>2006-01-01T09:01:01Z</member_since>
   <personal_url>
     www.socialedge.org/blogs/kiva-chronicles
   </personal_url>
   <occupation>Entrepreneur</occupation>
   <loan_because>I love the stories. </loan_because>
   <occupational_info>I co-founded a startup
     nonprofit (this one!) and I work with an amazing
     group of people dreaming up ways to
     alleviate poverty through personal lending.
   </occupational_info>
   <loan_count>89</loan_count>
   <invitee_count>23</invitee_count>
  </lender>
  <lender>
   <lender_id>jessica</lender_id>
   <name>Jessica</name>
   <image>
    <id>197292</id>
    <template_id>1</template_id>
   </image>
   <whereabouts>San Francisco CA</whereabouts>
   <country_code>US</country_code>
   <uid>jessica</uid>
   <member_since>2006-01-01T09:01:01Z</member_since>
   <personal_url>www.kiva.org</personal_url>
   <occupation>Kiva cofounder</occupation>
   <loan_because>
     Life is about connecting with each other.
   </loan_because>
   <occupational_info/>
   <loan_count>54</loan_count>
   <invitee_count>26</invitee_count>
  </lender>
     ....
 </lenders>
</snapshot>
```

The sequence of <*lender*> nodes in the Kiva data naturally maps to a list in *R* with an element for each <*lender*> node. Similarly, each <*lender*> can be represented as a list. That is, each child node of a <*lender*> node can be mapped to either a string containing the text content of the child or in the case of <*image*>, a list or vector with two elements: <*id*> and <*template_id*>. Basically, it is natural to map an *XML* node to a list with an element for each child node using the name of the

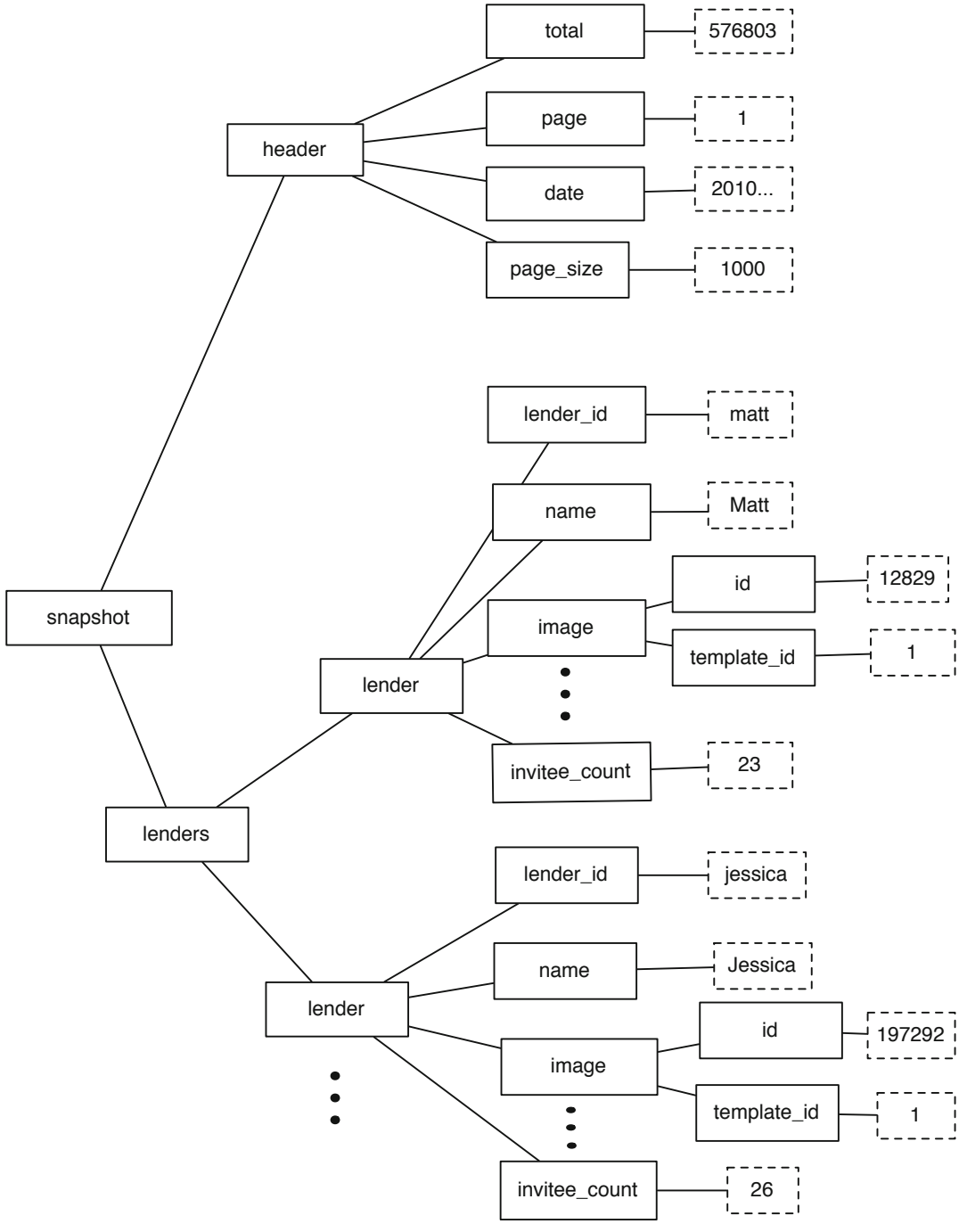


Figure 1.2: Tree Diagram of a Kiva Lender *XML* Document. This graphical representation of a Kiva lender document shows the basic structure of the *XML* document. Notice the hierarchical format of the *XML* where we have a single root node called <snapshot>, its two children <header> and <lenders>, and so on. In the figure, a rectangle with dashed lines denotes text content.

child as the name for the list element. In the next example, we use the function xmlToList() to help us do this.

Example 1-2 Converting XML-formatted Kiva Data to an R List or Data Frame

We begin by parsing the *XML* document with

```
doc = xmlParse("kiva_lender.xml")
```

We then pass `doc` to the xmlToList() function, and it will return an *R* list with an element for each of its top-level child nodes, mapping each of these children to an *R* list and so on, in the same recursive way:

```
kivaList = xmlToList(doc, addAttributes = FALSE)
```

The result is a list with 1000 elements, one for each <*lender*> node. (The `addAttributes = FALSE` ensures that any *XML* attributes are not included in the result, e.g., the `type="list"` in the <*lenders*> node.) The first lender element in the list is

```
$lender_id
[1] "matt"

$name
[1] "Matt"

$image
$image$id
[1] "12829"

$image$template_id
[1] "1"


$whereabouts
[1] "San Francisco CA"

$country_code
[1] "US"

$uid
[1] "matt"

$member_since
[1] "2006-01-01T09:01:01Z"

$personal_url
[1] "www.socialedge.org/blogs/kiva-chronicles"

$occupation
[1] "Entrepreneur"

$loan_because
```

```
[1] "I love the stories. "

$occupational_info
[1] "I co-founded a startup nonprofit (this one!)
and I work with an amazing group of people dreaming up
ways to alleviate poverty through personal lending. "

$loan_count
[1] "89"

$invitee_count
[1] "23"
```

When appropriate, xmlToList() makes converting *XML* content to *R* quite easy.

On the other hand, if the *XML* data have a simple structure, we can read it into a data frame with the xmlToDataFrame() function. We might arrange the Kiva lenders data as a data frame, with an observation for each lender and a column/variable for each node within <*lender*>, e.g., <*lender_id*>, <*name*>, <*country_code*>, <*loan_count*>. In our situation, the <*lender*> nodes are two levels below the root node so we need to access them to pass to xmlToDataFrame(). We get the top-level/root node and then its <*lenders*> node as follows:

```
lendersNode = xmlRoot(doc)[["lenders"]]
```

The xmlRoot() function gives us the top-level node of our document, i.e., <*snapshot*>. To fetch the <*lenders*> subnode, we treat the root node as if it were a list in *R* and use the expression `node[["lenders"]]` to extract the (first) child node whose element name is <*lenders*>. This is a convenient way to access child nodes. We can also index by position, e.g.,

```
xmlRoot(doc)[[2]]
```

as we know the second element is the <*lenders*> node. We want the *list* of <*lender*> nodes. The function xmlChildren() is the means for getting the list of all child nodes of a given node, e.g., the <*lender*> nodes under <*lenders*>. We then pass this list of the individual <*lender*> nodes to xmlToDataFrame() to create a data frame with

```
lenders = xmlToDataFrame(xmlChildren(lendersNode))
```

This function returns a 1000 by 13 data frame. The variables in the data frame correspond to the top-level *XML* elements in each of the <*lender*> nodes, i.e.,

```
names(lenders)
```

```
 [1] "lender_id"      "name"               "image"
 [4] "whereabouts"    "country_code"       "uid"
 [7] "member_since"   "personal_url"       "occupation"
[10] "loan_because"   "occupational_info" "loan_count"
[13] "invitee_count"
```

Note that this approach collapses the `image` column to just the value of the first child node in <*image*>. This may not be what we want. In Chapter 3, we continue with this example and demonstrate how to include the children of <*image*> in our data frame.

The previous example introduced two high-level functions, xmlToList() and xmlToDataFrame(), for extracting *XML* content into *R* lists and data frames, respectively. We also got a glimpse of other functions available in the XML package, such as xmlParse(), xmlRoot(), xmlChildren() and `[[` for accessing child nodes within a node. These and other functions provide much greater control over data extractions. The example in the next section gives a preview of the possibilities, particularly for working with attributes on *XML* nodes. Chapter 3 provides a more in-depth introduction to these parsing functions.

1.3.1 Extracting Data from XML Attributes

When we examine the *XML* for the Kiva lenders, we see only one attribute being used. This is `type = "list"` within the <*lenders*> node. In this case, the attribute conveys metadata about the content of the <*lenders*> node. In other *XML* documents, the attributes often contain data. For example, the following is a segment describing activities related to a bill in the US Congress [1] (available at `http://www.govtrack.us/developers`):

```
<bill session="111" type="h" number="1"
      updated="2011-01-29T15:03:08-05:00">
 <state datetime="2009-02-17">ENACTED:SIGNED</state>
 <status><enacted datetime="2009-02-17" /></status> ...
 <relatedbills>
  <bill relation="rule" session="111" type="hr" number="88"/>
  <bill relation="rule" session="111" type="hr" number="92"/>
  <bill relation="rule" session="111" type="hr" number="168"/>
  <bill relation="unknown" session="111" type="h" number="290"/>
  <bill relation="unknown" session="111" type="h" number="598"/>
  <bill relation="unknown" session="111" type="s" number="350"/>
    ....
 </relatedbills> ...
</bill>
```

We see that the date the bill was enacted is given in the attribute *datetime* in the <*enacted*> element, and the information about each related bill is provided via the attributes *relation*, *session*, *type*, and *number* in a <*bill*> node within <*relatedbills*>. Hence we need a tool that extracts attributes from *XML* nodes to access this information. There are two functions for this in the XML package: xmlAttrs() and xmlGetAttr(). The xmlAttrs() function returns a named character vector of all the attributes for a given node, from which we can extract individual elements as strings. The xmlGetAttr() function is used to retrieve a single attribute, rather than returning the entire collection of attributes. xmlGetAttr() also allows us to provide a default value if the attribute is not present in the node and to coerce the value if it is present. We demonstrate both functions in the next example.

Example 1-3 Retrieving Attribute Values from XML-formatted Bills in the US Congress

Consider the <*bill*> nodes within <*relatedbills*> for a particular bill. We have a list of these <*bill*> nodes available in the `rBills` variable; the attributes on the first related <*bill*> node are

```
xmlAttrs(rBills[[1]])
```

```
relation  session     type   number
  "rule"    "111"     "hr"     "88"
```

We can combine these attributes across <bill> nodes into a data frame with

```
do.call(rbind, lapply(rBills, xmlAttrs))
```

With the xmlGetAttr() function we can retrieve a single attribute and specify a default value that is returned if the attribute is not present. Furthermore, we can coerce the attribute's string value to a particular type. For example, if we want the *number* attribute back as an integer or NA, if not present, then we can use

```
xmlGetAttr(rBills[[1]], "number", NA, as.integer)
```

We can collect all the *number* values across the <bill> nodes with

```
as.integer(sapply(rBills, xmlGetAttr, name = "number"))
```

```
 [1]   88  92 168 290 291 598 629 679 679 861
[11]  336 336 350 350
```

Here, it is better to convert the vector of attribute values together into an *integer* rather than each one individually.

1.4 Reading Data from *JSON*-formatted Documents

Kiva also provides data in *JSON* format. This looks something like the following (after manual formatting)

```
{"header": {"total": "576803", "page": 1,
            "date":"2010-01-29T20:00:23Z",
            "page_size":1000},
 "lenders":[ {"lender_id": "matt", "name": "Matt",
              "image": {"id":12829,"template_id":1},
              "whereabouts" : "San Francisco CA",
              "country_code": "US", "uid":"matt",
              "member_since": "2006-01-01T09:01:01Z",
              "personal_url":
                  "www.socialedge.org/blogs/kiva-chronicles",
              "occupation":"Entrepreneur",
              "loan_because":"I love the stories. ",
              "occupational_info":"I co-founded a
               startup nonprofit (this one!) and I work
               with an amazing group of people dreaming
               up ways to alleviate poverty through
               personal lending. ",
              "loan_count":89, "invitee_count":23
             },
             {
              "lender_id": "jessica", "name":"Jessica",
              "image": {"id":197292,"template_id":1},
```

```
            "whereabouts":"San Francisco CA",
            "country_code":"US", "uid":"jessica",
            "member_since":"2006-01-01T09:01:01Z",
            "personal_url":"www.kiva.org",
            "occupation":"Kiva cofounder",
            "loan_because": "Life is about
             connecting with each other.",
            "occupational_info":"", "loan_count":54,
            "invitee_count":26
           }, ....
      ...
       ]}
```

We will describe the details of the format in Chapter 7. For now, hopefully you can see that this contains exactly the same information as the *XML* content and in the same order. We have the meta data in the header and a list of lender objects in the lenders field. The [ ] identifies an ordered collection such as a *vector* or *list* in *R* and the {}introduces a collection of named elements. Many people claim *JSON* content is easier to read and more compact than *XML*. This depends on how it is formatted, but is often the case. However, *JSON* is much less general than *XML* and there are times when we want the power and versatility of *XML*. Most importantly, we often do not have a choice in the format we are given and we need to be agile with both.

Example 1-4 Converting JSON-formatted Kiva Data into an R List

We can read the *JSON*-formatted Kiva content into *R* using the fromJSON() function from either of the RJSONIO or `rjson` [2] packages. The fromJSON() function reads the entire *JSON* content and converts it to the corresponding *R* type. For example,

```
kiva = fromJSON("lenders/1.json")
```

gives us a named list with two elements:

```
names(kiva)
```

```
[1] "header"  "lenders"
```

Let us examine the first lender and compare it to the return value from xmlToList():

```
kiva$lenders[[1]]
```

```
$lender_id
[1] "matt"

$name
[1] "Matt"

$image
         id template_id
      12829           1

$whereabouts
[1] "San Francisco CA"
```

```
$country_code
[1] "US"

$uid
[1] "matt"

$member_since
[1] "2006-01-01T09:01:01Z"

$personal_url
[1] "www.socialedge.org/blogs/kiva-chronicles"

$occupation
[1] "Entrepreneur"

$loan_because
[1] "I love the stories. "

$occupational_info
[1] "I co-founded a startup nonprofit (this one!)
and I work with an amazing group of people dreaming
up ways to alleviate poverty through personal lending."

$loan_count
[1] 89

$invitee_count
[1] 23
```

We see that it is virtually the same as the output from processing the *XML* content with xmlToList(). However, one difference is that fromJSON() has turned strings that have numeric values into numbers, and collapsed collections of these into numeric vectors. In contrast, the *XML* code leaves these values for the caller to interpret. We will see that there are also ways for us to automate this conversion for *XML* using schema (Chapter 14) which not only converts numbers and logical values, but also defines classes for higher-level data types.

JSON is a limited, non-extensible vocabulary that is very convenient and useful. It unfortunately does not allow one to represent `NaN` (not a number) or infinity and certainly not the notion of a missing value. We will see later in Chapter 7 that the caller of fromJSON() can try to control how to map *JSON* values to these symbolic values.

1.5 Summary of Functions to Read *HTML*, *XML*, and *JSON* into *R* Data Frames and Lists

Below are brief descriptions of the high-level functions in the XML package for parsing an *XML* document into an *R* list or data frame. Also included are a few other functions that we used in this chapter, which are available in the XML package. These other functions are more formally introduced in Chapters 3 and 5.

readHTMLTable() Return all the tables in the document as data frames. The *which* parameter allows us to specify those tables we want to extract from the document. The function's *colClasses* parameter allows us to specify the classes/types for the columns and a function will convert the content to numbers, percentages, factors, etc.

xmlToList() Loop over the child nodes of an *XML* node and create a list, where each child node is mapped to either a string containing the value of the node or to a list if the node contains child nodes, and so on.

xmlToDataFrame() Loop over the child nodes of an *XML* node and create a data frame, where there is one row in the data frame for each child node and the variables correspond to the value of top-level *XML* nodes in each child. If the top-level nodes in each child contain *XML* nodes, then the value of the first node is retrieved.

getHTMLLinks() Return all of the hyperlink targets (*href* attribute values) in the *HTML* document, optionally ignoring within-document links and making relative links absolute.

xmlParse() Parse an *XML* document. This function can read a local file, a remote *URL*, *XML* content that is already in *R* as a string, or a connection.

htmlParse() Parse an *HTML* document. This function is a less restrictive version of the *XML* parser that tolerates *HTML* content that is typically not well-formed in the *XML* sense.

xmlRoot() Retrieve the root node of an *XML* document. This is usually given the parsed document, but we can also pass it any *XML* node in a document to get the root node.

xmlChildren() Get a list of all the child nodes of a given *XML* node.

xmlAttrs() Retrieve a named character vector of all the attributes of a given node.

xmlGetAttr() Retrieve the value of a single attribute of a given node, optionally converting it from a string to a different type. This function also allows provision of a default value to use if the attribute is not present in the node.

The following lists the main functions in the RJSONIO package for parsing *JSON* content. The *JSON* format (and the RJSONIO package) is described in more depth in Chapter 7.

fromJSON() Parse *JSON* content into a list, and convert the content into the corresponding *R* type.

toJSON() Convert an *R* object to a *JSON* string.

1.6 Further Reading

The goal of this chapter is to entice you to read further in Part 1 of this book to see how to parse and write *XML* and *JSON* from within *R*.

References

[1] Civic Impulse, LLC. GovTrack.us developer documentation. http://www.govtrack.us/developers, 2012.

[2] Alex Couture-Beil. rjson: Converts *R* object into *JSON* and vice-versa. http://cran.r-project.org/web/packages/rjson/, 2011. *R* package version 0.2.6.

[3] *JSON* Advocate Group. Introducing *JSON*: A lightweight data-interchange format. http://www.json.org/, 2006.

[4] Kiva Organization. Kiva: Loans that change lives. http://www.kiva.org/, 2011.

[5] *R* Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. http://www.r-project.org.

[6] John Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[7] Duncan Temple Lang. RJSONIO: Serialize *R* objects to *JSON* (*JavaScript* Object Notation). http://www.omegahat.org/RJSONIO, 2011. *R* package version 0.95.

[8] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[9] Wikimedia Foundation. Wikipedia: The free encyclopedia. http://en.wikipedia.org/wiki/Main_Page, 2011.

[10] Worldwide Web Consortium. Extensible Markup Language (*XML*) 1.0. http://www.w3.org/TR/REC-xml/, 2008.

Chapter 2
An Introduction to *XML*

Abstract This chapter aims to give a reasonably comprehensive definition and motivation for the various aspects of the generic *XML* language and also to illustrate these aspects with some existing *XML* dialects or vocabularies. We describe elements, attributes, child elements, and the hierarchical structure of *XML*. We talk about "well-formedness" of an *XML* document and how to identify errors in a document's structure. We discuss the use of namespaces and end with a brief discussion of validating documents with respect to *DTD*s and *XML* Schema. Readers already familiar with all aspects of *XML* can skip this chapter and read about the functions used to work with *XML* in *R*, which are the subject of each of Chapters 3, 4, 5, and 6.

2.1 Overview

The eXtensible Markup Language (*XML*) [43] provides a general approach for representing all types of information such as data sets containing numerical and categorical variables; spreadsheets; visual graphical displays such as *SVG*; descriptions of user interfaces; social network structures; text documents such as word processing documents and slide displays; *RSS* (Rich Site Summary or alternatively Real Simple Syndication) feeds; data sent to and from Web services; settings and preferences on computers; *XML* databases; and the list goes on. *XML* is so generic, it can be used to represent any data. Over the last fifteen years, *XML* has grown from a proposed simplification of *SGML* to a widely adopted and used technology in a multitude of areas, and today claims a plethora of many powerful real-world applications related to the management, organization, dissemination, and display of a broad array of data. As data scientists, we encounter *XML* on a daily basis in most aspects of data technologies.

This ubiquity and broad set of applications makes a compelling case for why anyone working with data needs to have some familiarity with *XML*. Although not rocket science, *XML* is much more than just the syntax or general format that is the *XML* specification. *XML* also includes the collection of inter-related specifications and technologies, such as *XInclude*, *XLink*, *XPointer*, *XPath*, Schema, *XSL*, and *XQuery* that make *XML* so useful and powerful. This chapter gives an overview of *XML*'s syntax and structure. We will also briefly discuss *DTD*s and schema, which are used to describe the structure of classes of *XML* documents. In other words, this chapter looks at the essential structure and components of *XML* to provide a foundation for all of the other topics. *XPath*, *XInclude*, and *XPointer* are covered in Chapter 4.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_2,

XML is not itself a language for representing data. Rather, it is a very general structure with which we can define any number of new formats to represent arbitrary data. *XML* provides the basic, common, and quite simple structure and syntax for all of these "dialects" or vocabularies. For example, the reader who has read or composed *HTML* (HyperText Markup Language) will recognize the format of *XML* because *HTML* is a particular vocabulary of *XML*. We provide a sample document below, which succinctly introduces the main components and concepts of an *XML* document. This *XML* is a simple *DocBook* file. *DocBook* [26, 42] is a vocabulary designed for authoring technical documents, e.g., books, articles, help files, and tutorials. It is much like LaTeX but uses *XML* to represent the content in a way that separates content from presentation and also allows us to programmatically query and modify documents. The example also includes an extension of *DocBook* that we have created in order to mark up *R* code, functions, expressions, and values [23, 24]. (This book was written using *DocBook* and these *R*-related extensions.) This sample *XML* also demonstrates how we can dynamically include parts of another *XML* document using *XInclude*. These two documents illustrate how we can use the generic markup to represent quite different data. You may find that reviewing these documents and the brief descriptions of their various components offers enough information about *XML* to skip the remainder of the chapter.

```
<?xml version="1.0" encoding="UTF-8"?>  1
<!-- This comment can't be first in the document -  2
  the <?xml ...> has to be first unless that is
  not present. We can have multiple lines in a comment
  and special XML characters such as < > and &. -->
                                                    3
<article id="DataAnalysis"  4, 5
            xmlns="http://docbook.org/ns/docbook"  6
            xmlns:r="http://www.r-project.org">  7
  <title>Article Title</title>
  <para>
   This is text in a paragraph that includes  8
   &lt; and a comment    9
   <!-- comment--> and a processing instruction:
   <?R sum( 1, 3, 5) ?>  10
   and some <r/> code with <r:func name="table"/>  11
   <r:code r:width='50' ><![CDATA[   12, 13
   x <- (y > 1 & z < 0)  14
   table(x)
  ]]>
  <r:output>
FALSE  TRUE
   13    29
</r:output>
  </r:code>
  </para>
  <para>
  We also include a subset of data from
  another document using XInclude:
  <dataFrame>
  <xi:include href="fuelData.xml"  15
```

```
    xpointer=
     "xpointer(/dataFrame/*[@name = 'Cmb.MPG'])"/>  16
   </dataFrame>
   </para>
</article>
```

1 The *XML* declaration along with the character encoding that should start each *XML* document. The character encoding removes any ambiguity about how to interpret the characters from the author's language.
2 A comment *before* the root node (see 4). Note that the comment contains text that is ordinarily treated specially by an *XML* parser, i.e., < and >. These need not be escaped within the comment as an *XML* processor is looking for the literal sequence `-->` to end the comment and treats all other characters within the comment as regular characters and not as *XML*.
3 This is whitespace before the root node. It can also occur anywhere within the document and counts as a text node.
4 The topmost or *root* node is `<article>`.
5 An attribute *id* with a value of `"DataAnalysis"`.
6 The default namespace identified by `xmlns="a uri"`. Note that there is no prefix here. This namespace applies to all subnodes without an explicit namespace prefix.
7 The *R* namespace that is identified by the r-project URI (Uniform Resource Identifier). It uses `r` as a prefix to identify nodes that belong to this grammar.
8 A text node.
9 An entity *<* within text that expands to the character <. Another common entity is *&* which expands to &. Entities are used when the character would be interpreted by an *XML* parser as part of an *XML* construct, e.g., < as the start of an *XML* element.
10 A processing instruction that calls *R* to sum 1, 3, and 5.
11 Two elements that have no content. The first denotes the *R* language and the second denotes the name of an *R* function, table().
12 An element and an attribute that use a defined namespace.
13 This is escaped character data (CDATA). An *XML* parser reads the content without processing it as *XML* in any way and so treats it as verbatim text. We can use this *<CDATA>* construct to escape entire blocks of text that contain special characters such as < and &, rather than using entities to escape each instance individually.
14 This is the content that is escaped from the *XML* parser.
15 The parser will (by default) substitute the `<xi:include>` instruction with the contents from the specified file `fuelData.xml`.
16 This *xpointer* attribute identifies which nodes within the target document to include. This allows us to select a subset of the target document, in this case just three variables in the data frame. Only the nodes that match the *XPath* expression given in the *xpointer* attribute will be included.

Below is the `fuelData.xml` document included in the *DocBook* document. It represents a subset of the fuel efficiency data available from http://www.fueleconomy.gov/feg/download.shtml. Portions of this document are included in the *DocBook* article displayed above. We show only ten observations and four variables. The number of observations in the data set is given via the *numObs* attribute on the root *<dataFrame>* element. Variables appear in elements that are named according to the type of the variable, e.g., *<categorical>*, *<integer>*, *<real>*, and the names of variables are specified via *name* attributes on these elements. Categorical variables (or

factors) are given by their set of categories and then the indices into this set to give the actual values. The noncategorical variables are stored within individual values inside <*value*> elements.

```
<?xml version="1.0"?>
<dataFrame numObs="10" year='2012'>
 <source>
Energy Efficiency &amp;amp; Renewable Energy,
U.S. Environmental Protection Agency (EPA).
 <ulink url="http://www.fueleconomy.gov/...alpha_12.txt"/>
 </source>
 <categorical name="Model">
  <categories numCategories="10">
   <category>BMW Alpina B7 SWB</category>
   <category>CHEVROLET Malibu</category>
   <category>CHEVROLET Silverado 15</category>
   <category>DODGE Charger</category>
   <category>GMC Yukon XL 1500</category>
   <category>HYUNDAI Genesis</category>
   <category>KIA Optima</category>
   <category>KIA Sportage</category>
   <category>PORSCHE Cayenne Turbo</category>
   <category>VOLKSWAGEN CC 4Motion</category>
  </categories>
  <values>6,5,3,7,8,1,2,10,9,4</values>
 </categorical>
 <real name="Displ">
  <values>
   <value>4.6</value> <value>6.2</value>
   <value>5.3</value> <value>2.4</value>
   <value>2.4</value> <value>4.4</value>
   <value>2.4</value> <value>3.6</value>
   <value>4.8</value> <value>3.6</value>
  </values>
 </real>
 <integer name="Cyl">
  <values>
   <value>8</value> <value>8</value>
   <value>8</value> <value>4</value>
   <value>4</value> <value>8</value>
   <value>4</value> <value>6</value>
   <value>8</value> <value>6</value>
  </values>
 </integer>
 <categorical name="Cmb.MPG">
  <categories numCategories="8">
   <category>10/14</category> <category>13/17</category>
   <category>17</category> <category>17/23</category>
   <category>20</category> <category>24</category>
```

```
   <category>26</category> <category>28</category>
  </categories>
  <values>5,1,2,8,6,3,7,5,3,4</values>
 </categorical>
</dataFrame>
```

These two documents provide examples of the various parts of an *XML* document. The remainder of this chapter fills in the details. Section 2.2 provides the syntax rules that an *XML* document needs to abide, Section 2.3 introduces several *XML* grammars in order to give the reader a flavor for the features of *XML*, Section 2.4 examines the hierarchical structure of the *XML* document, Section 2.5 explains the remaining markup introduced above (e.g., processing instructions, `<CDATA>`, and the *XML* declaration), and Section 2.7 describes the *XML* grammar, called *XML* Schema, that is used to define allowable tags and structures in an *XML* grammar.

2.2 Essentials of *XML*

The basic unit in *XML* is the *element*, which we also refer to as a *node* when we talk about the hierarchical or treelike structure of the *XML* document. An element has a name, and may have attributes and child elements, each of which we describe below.

Element Names
Each element begins with the *start-tag*, e.g., `<title>` or `<article>`, and must close with a corresponding *end-tag*, e.g., `</title>` or `</article>`, respectively. That is, tags are paired and they delimit an element and its contents. In general, the start tag has the format `<tagname>` and the matching end-tag is identical except for the addition of the forward slash between the < and the tag name. In many respects, pairs of opening and ending tags are like parentheses, but with names that make it easier to identify the pairs when the elements are nested hierarchically.

Child Elements and Recursive Structure
XML elements can have content made up of other *XML* elements that are treated as child elements. It is this nested/recursive structure that allows us to represent different, complex data structures using *XML*. The *XML* content below (adapted from http://www.w3schools.com/XML/plant_catalog.xml)

```
<plant>
 <name>
  <common>Bloodroot</common>
  <botanical>Sanguinaria canadensis</botanical>
 </name>
 <zone>4</zone>
 <light>Mostly Shady</light>
 <price>$2.44</price>
</plant>
```

illustrates how the `<plant>` element contains four child elements: `<name>`, `<zone>`, `<light>`, and `<price>`. The `<zone>`, `<light>`, and `<price>` elements have content in the form of text elements, e.g., `Mostly Shady`. The `<name>` element also has child elements: `<common>` and `<botanical>`. Child elements are typically either regular *XML* elements, with a start and end tag, or simple text content. Text elements are simple *XML* elements that can have no children. We can

also have other types of *XML* elements such as comments, processing instructions, etc. However, the data in an *XML* document are mostly in regular elements, text nodes, and attributes.

The `<para>` element in the first document in Section 2.1 illustrates how we can mix regular elements and text as child nodes.

```
<para>
This is text ...
and some <r/> code with <r:func name="table"/>
 <r:code r:width="50"><![CDATA[
 x <- (y > 1 & z < 0)
 table(x)
]]>
<r:output>
FALSE  TRUE
   13    29
</r:output>
</r:code>
<para>
```

The text nodes, e.g., `"This is text ... and some"` and `" code with"`, are interspersed with `<r>`, `<r:func>`, and `<r:code>` elements. The `<r:code>` element contains both text (within what is called a `<CDATA>` or "character data" section) and another element `<r:output>`.

This recursive structure of *XML* elements gives rise to trees, or hierarchies, of elements. Each *XML* document has a single root, or top-level, element. In our two example documents from Section 2.1, these are `<article>` and `<dataFrame>`.

Attributes

A regular *XML* element can have zero or more attributes, which are `name="value"` pairs. Attributes are specified in the start tag of an *XML* element. The attribute value must be contained within quotes, either a pair of single or double quotes. For example, the `<dataFrame>` tag from Section 2.1

```
<dataFrame numObs="10" year='2012'>
```

contains two attributes named *numObs* and *year*. There is much debate over whether to use attributes or put the values as children within an element. For example, we could have represented the *dataFrame* element in an equivalent form as

```
<dataFrame>
 <numObs>10</numObs>
 <year>2012</year>
 <source>
   ...
</dataFrame>
```

We typically use an *XML* attribute when describing metadata about the data within the element. For example, an attribute might specify the number of observations in the data frame; a printing option such as the number of digits; or whether or not to evaluate the code in an `<r:code>` node. The attributes keep this information separate from the content, e.g., whether or not to evaluate code is separate from the actual code within an `<r:code>` element. Note that attribute names must be unique within an element, i.e., we cannot have two attributes with the same name.[1] When the need to have duplicates arises, it often suggests using child elements rather than attributes.

[1] We can have two attributes with the same name if they belong to different *XML* namespaces.

There are some attributes that are special in *XML*. One of them is the *id* attribute. For example, the `<article>` element has an *id* attribute with value `"DataAnalysis"`. We use the *id* attribute to assign a unique identifier to an element (and its subtree). These unique identifiers or elements are very useful when extracting portions of a document or creating cross references in a document. The value *must* be unique across all *id* attribute values within the entire document and an *XML* parser will typically insist on this or produce a warning or error. The attribute *xml:lang* (or often simply *lang*) is another special attribute. This is used to identify the spoken or natural language in which the content is written, such as English or French.

Merging Start and End Tags for an Element

Basically, each *XML* element has a start tag and matching end tag. However, the end tag is unnecessary when the element has no child elements. In these cases, we can use the short version `<tagname/>`, i.e., ending the start tag with a `/` character. The `<r/>` element is an example of this. The start tag and end tag have been collapsed or contracted into one tag. There are several reasons why we may have an element with no content. In the case of `<r/>`, the tag identifies a particular concept—the *R* language—without the need for any additional qualification or parameterized content. Another reason is that the content may be specified in the tag's *attributes*. For example, `<r:func name="table"/>` provides the name of the *R* function table() via the *name* attribute on the tag. Also note that an empty element may be specified in its long form, e.g.,

```
<r:func name="runif"></r:func>
```

Collapsing the empty element into one tag is optional. Different pieces of software can produce the *XML* either way. It does not make any difference when we parse the *XML* content as the parser will create an *XML* element for us and we never need to look at the start or end tag.

Well-formed *XML*

XML is a specification of a general structure for describing content. It defines the basic and general syntax for specifying elements, attributes, and hierarchical content by nesting elements as children of other elements. *XML* also provides constructs such as entities and `<CDATA>` sections, comments, and processing instructions, but it does not define any element or attribute names. Instead, it is just an overarching framework or scaffolding that allows anyone to define an *XML* vocabulary made up of element and attribute names and to give meaning to those elements and the relationships between them. *XHTML* is one *XML* vocabulary, introducing elements such as `<html>`, `<body>`, `<table>`, `<a>`, and `<img>` for describing Web pages. *SVG* is another vocabulary introducing elements such as `<circle>`, `<line>`, `<text>`, `<path>`, `<g>` (for group), `<animate>`, etc., for describing two-dimensional, interactive, and animated graphical displays. *KML* is a markup language for describing three-dimensional geographical information and displays. Our markup of the `<dataFrame>` in Section 2.1 is another vocabulary, albeit less standard. Each of these uses the same *XML* syntax and structure, but defines its own elements and attributes and their meanings. This is somewhat analogous to words, sentences, paragraphs, sections, and chapters. These hierarchical structures are common to many different languages, yet each of these languages uses different words. Furthermore, with the same basic structure, we can produce very different types of documents in any one of these languages.

When a document obeys the basic syntax rules of *XML*, it is said to be *well-formed*. The criteria are very general in nature, and do not pertain to a specific grammar, i.e., a specific set of allowable element and attribute names. The following few simple rules are an important subset of these syntactic requirements for a well-formed *XML* document; they cover the vast majority of cases that we encounter when we work with *XML* to access data. *XML* documents that are not well-formed produce potentially fatal errors or warnings when read.

Properties of Well-formed *XML*

Well-formed *XML* adheres to the following rules:

- One root element completely contains all other elements within the document, excluding the *XML* declaration and optional comments or processing instructions that may appear before the root node.
- Elements must nest properly, i.e., be opened and closed in the same order.
- Element names or tags are case sensitive.
- Tags must close, e.g., `<title>My article</title>`, or be self-closing, e.g., `<img src="pic.jpg"/>`.
- Attributes appear in start-tags of elements, and never in end-tags.
- Attribute values have a `name="value"` format and the value must be quoted either with matching single or double quotes, but not mixed.
- Attribute names cannot be repeated within a given element (except if they are within different namespaces).
- No blank space is allowed between the < character and the tag name. Extra space is allowed before the ending > in the opening and closing tag. The blank space after the element name is to separate the tag from the first attribute, if it is present.
- Element and attribute names must begin with an alphabetic character or an underscore _; subsequent characters may include digits, hyphens, and periods. No space, colon, or the triple `"xml"` may appear in a tag name.
- The colon character is used for namespace prefixes. The namespace must be defined in the node or one of its ancestors.
- Within attribute values, the special characters & and < must be specified as entities, e.g., `company="AT&amp;T"`.

There are additional rules that must be adhered to in order to be well-formed, but those found here cover the vast majority of the cases.

We explain the rules for well-formed *XML* in more detail.

- *An XML document must have one root element* that completely contains all other elements. The document can have special nodes such as the *XML* declaration, processing instructions, or comments before the root node. However, there must be exactly one regular *XML* element at the top-level of the document. We have seen some documents be created by concatenating different *XML* documents together, or writing several elements to a file. These are not valid *XML* documents since there are multiple top-level elements. Documents that have two or more top-level nodes can be "fixed" by enclosing the elements within a higher-level parent element.
- *XML tags are case sensitive* so start and end tag names must match exactly. For example,

  ```
  <date>1/1/2012</Date>
  ```

  is not well-formed because the start-tag begins with a lowercase `"d"` which does not match the capital `"D"` in the end-tag.
- *All start tags must have a closing tag, or be self-closing.* If we open a tag, we must close it with a corresponding `</tagname>` entry. A self-closing tag (i.e., one that is used as both the start and end tag of an element) has no enclosed or nested content and so can be contracted to a single tag of the form `<tagname />`. For example,

  ```
  <img src="myPic.png"></img>
  ```

is an empty <*img*> element, i.e., it has no child nodes, (it does have a *src* attribute) and can be represented equivalently as

```
<img src="myPic.png"/>
```

HTML documents often violate this rule with some elements having no closing tag.

- *Elements must nest properly.* Elements must open and close in the same order, i.e., if an element contains another element, then both the start and end tags of the inner element must be within the start and end tags of the containing element. For example, in the following *XML*,

```
<affiliation>
 <orgname>University of California</orgname>
 <orgdiv>Department of Statistics</orgdiv>
</affiliation>
```

the <*orgname*> and <*orgdiv*> elements are nested correctly within the <*affiliation*> element. The following *XML* is not properly nested because the end-tag for <*orgdiv*> appears after the closing tag of <*affiliation*>,

```
<affiliation>
 <orgname>University of California</orgname>
 <orgdiv>Department of Statistics
</affiliation>
</orgdiv>
```

Note the use of indentation is optional. Most people find that it is easier to see whether elements are properly nested when they are indented. However, we do note that the actual *XML* document will have text elements corresponding to the white space used for the indentation, which may be unwanted.

- *Attributes can only appear in the start-tag*; they may not appear in end-tags.
- *Attribute values must appear in quotes in a* `name="value"` *format.* Either single or double quotation marks are allowed. For example, the <*Cube*> element below has a value of `2008-04-21` for its *time* attribute.

```
<Cube time='2008-04-21'>
```

XHTML documents are well-formed, but *HTML* documents are frequently not. Attributes in *HTML* are frequently not quoted, and as aforementioned, elements often do not have an explicit closing tag.

- *Attribute names are case sensitive and follow the same rules as tag names.*
- *No more than one occurrence of an attribute name in a tag is allowed* (except if they are within different namespaces).
- *Attribute values cannot include* <, > or &. For example, we cannot have `attr="x < 10"`. Instead, we have to use the corresponding entity in the text of the value, `attr="x < 10"`. An *XML* parser will typically replace the entity with the actual text so we do not have to deal with the entity directly.
- *No blank space is allowed between the* < *and the tag name.* On the other hand, extra space is allowed before the >. For example, `< foo>` is not allowed, but `<foo >` is. Similarly, the closing tag `</foo >` is allowed, but not `</ foo>` or `< /foo>`.
- *An element must have a name*, i.e., <> is not allowed.
- *Tag names must begin with an alpha character or an underscore* `'_'`, *and subsequent characters may also include digits, hyphens, and periods*. Alphabetic characters can be upper or lowercase.

No space, colon, or the triple `"xml"` may appear in a tag name. The `"xml"` may be accepted by many *XML* parsers, but officially, it is reserved for future use. The colon is used to separate a namespace prefix from the name of the *XML* element, e.g., `<r:code>`. This is why it is not allowed as a character in the element's name.

Most of the remaining syntax rules cover circumstances that many of us are not very likely to encounter. They will not be discussed here. For more details on the rules for well-formed *XML* see [17] and [39].

An *XML* document may be well-formed but "not valid". Well-formed means only that the document conforms with the basic *XML* structure. It implies nothing about the validity of the content or meaning of the document. It may contain elements or attributes that make no sense or have no definition within the vocabulary. Similarly, it may have elements that are in the wrong location or incorrect order. *XML* schema and DTDs (Document Type Definitions) are used to specify the validity rules for a particular *XML* vocabulary. We will briefly discuss these later in this chapter.

2.2.1 Syntax Checkers

Most *XML* documents are created programmatically so they are typically well-formed, unless the software used to create them is broken. However, if a document is authored manually, it is quite common to introduce simple structural errors within the content. To avoid this, we use good *XML* authoring tools such as the open-source nXML mode [4] for Emacs [29], or dedicated commercial *XML* editors such as `<oXygen/>` (`http://www.oxygenxml.com/`), XMLmind (`http://www.xmlmind.com/`), and EditiX (`http://www.editix.com/`).

When we attempt to parse the document, the parser will typically inform us of syntax errors. Therefore, we can use our favorite *XML* parser to test for well-formedness of an *XML* document. Also, there are command-line tools we can use such as *xmllint* to test for well-formedness, e.g., we can invoke *xmllint* with

```
xmllint --noout doc.xml
```

This will show all the errors, if there are any. *xmllint* has many command-line options to control how it operates. There are also numerous other command line tools for testing well-formedness, e.g., *xmlwf* which uses the Expat *XML* parsing library. There is also a Web site RUWF (`http://www.xml.com/pub/a/tools/ruwf/check.html`) to which we can upload a document or point it to a *URL* to test.

The XML package [34] for *R* installs readily on most platforms. It provides an *R* function xmllint() to test for well-formedness of an *XML* document. This is perhaps the simplest, platform-neutral mechanism for testing a document as it does not involve command-line programs and shells. It also returns the errors as an *R* list with the information decomposed into *R* objects containing a description of each error, the name of the file and the line and column numbers at which it occurred. Furthermore, the processing of the errors can be customized.

Before we leave the topic of well-formedness, we should mention that sometimes we can work around an *XML* document not being well-formed. While it should not have been created as a malformed document, we often have to make do with the situation and attempt to extract the data. We can tell the parser in *R* to, e.g., attempt to recover from missing end-tags. We do this with the function xmlParse() and specifying the `RECOVER` option for the parsing, i.e.,

```
xmlParse(filename, options = RECOVER)
```

The parser has to guess what was meant and so does not always do sensible things with the input. For instance, it considers unquoted attributes within a start-tag as text outside of the start-tag. However, it can be useful in some circumstances.

When working with *HTML* documents, we can also try to correct these using the *HTML* Tidy facility. This can be used via a Web browser, or by sending *HTML* content to the Web interface, or directly within *R* using the RTidyHTML package [33].

2.3 Examples of *XML* Grammars

Many government agencies, commercial entities, and scientific research areas make their data available in *XML* formats. In the commercial arena, Microsoft Office [27], Libre Office [21], iWork [1], and Google Docs [14] use *XML* in their office suite tools, including spreadsheets and word processing documents. Google and the Open Geospatial Consortium (OGC) developed the Keyhole Markup Language (*KML*) [11] as a language for describing geo-spatial information that can be rendered interactively using Google Earth [12], Google Maps [13], and Google Sky [15]. The eXtensible HyperText Markup Language (*XHTML*) [19] is a collection of *XML* tags for representing Web pages, similar to *HTML* but using well-formed and structured *XML*.

Scientists have developed specific grammars suitable for data in their fields. As an example, the Systems Biology Markup Language (SBML, `sbml.org`) offers a common format for describing biological systems, e.g., metabolic pathways, biochemical reactions, and gene regulation. With SBML, researchers can share the essential aspects of their models independently of the software environment. CellML is another biological markup language that has evolved into a general format to store and exchange computer-based mathematical models of biological processes. The Geography Markup Language (GML) was defined by the Open Geospatial Consortium (OGC) [25] to express geographical features. The Materials Markup Language (MatML) is an *XML* standard for the exchange of a material's property information and has been used in various applications, e.g., for contaminant emissions data. The Sloan Digital Sky survey provides results from Internet database queries in an *XML* format [22, 32].

The International Monetary Fund, World Bank, the Statistical Office of the European Communities, and other organizations have sponsored and developed SDMX, the Statistical Data and Metadata eXchange [30]. This is an initiative to foster *XML* standards for the exchange of statistical information. Foreign exchange rate data from the US Federal Reserve Bank and the European Central Bank conform to SDMX standards, and the United Nations Statistical Commission has recognized SDMX as the preferred standard for the exchange and sharing of data and metadata [36]. As another example, the US Food and Drug Administration (USFDA) and the European Agency for the Evaluation of Medical Products (EMEA) are working on a series of initiatives to develop *XML*-based standards for data exchange, which includes Structured Product Labeling (SPL) for USFDA-regulated products [37]. Also, the US Census Bureau uses *XML* to facilitate the layout and assembly of economic census forms of US businesses. Similarly, a large number of data sets are available from `http://www.data.gov` in a variety of formats, including *XML*.

XML is widely used, but it is by no means the only format. *JSON* is popular for various reasons, most notably for its simplicity and brevity relative to *XML*. In many cases, *JSON* is simpler, but *XML* is more robust and better suited to precise description of the data structure via schema. This difference is analogous to the comparison between dynamic, untyped languages and strongly typed, compiled languages. The former is often better for one-off tasks or interactive exploration, while the latter often provide meta-tools for working with the data and also better long-term development and

reproducability needs. Some of the features of *XML* are shown in the box below. As consumers of data, we work with whatever format the data are made available to us.

Features of *XML*

- *XML* is *self-describing* in that it can contain the format and structural information needed to properly read and interpret the content. For example, an *XML* document typically specifies its character encoding in the *XML* declaration. It can contain a *DTD* or schema that describes the structure of all documents within that *XML* vocabulary. For traditional data sets, it can include the missing value identifier, description of its provenance, etc. Different data sets can be clearly identified within a document. Compare each of these with a CSV file.
- *XML* typically *separates* information (content and structure) from the appearance of the information. This is generally considered important in all aspects of software.
- The highly extensible format allows *XML* content and data to be easily *merged* into higher-level container documents and to be easily *exchanged* with other applications.
- The content of an *XML* document is *human-readable* using any simple plain-text viewer. Although human-readable, *XML* also supports binary data and arbitrary character sets.
- Since *XML* is highly structured, it is easily *machine generated* and read.
- Many communities are actively using *XML* and providing *extensive collections of tools* for working with *XML*, and these tools have been, or can be, incorporated into other environments and programming languages relatively transparently.

We provide a few examples in this section to highlight the important features of *XML*, e.g., that it is self-describing, separates content from form, and is easily machine generated.

Example 2-1 A DocBook Document

As mentioned at the start of this chapter, *DocBook* is a vocabulary designed for authoring technical material. It *leverages the extensive collection of XML tools*, such as *XSL* (eXtensible StyleSheet Language), *XPath*, *XInclude*, *HTML*, and *FO* (Formatting Objects) (and also LaTeX) for transforming structured *XML* documents and generating rendered versions that can be displayed on computer screens or printed on paper. For example, this book was written using *DocBook*.

Below is a snippet of author information in the *DocBook* format.

```
<author>
  <personname>
    <firstname>Jane</firstname>
    <surname>Smith Doe</surname>
  </personname>
  <email>janesd@gmail.com</email>
  <affiliation>
    <orgname>University of California</orgname>
    <orgdiv>Department of Statistics</orgdiv>
  </affiliation>
</author>
```

From this sample, one can see why *XML* is called *self-describing*. Without the mark up, the content reduces to:

```
Jane Smith Doe
janesd@gmail.com
```

```
University of California
Department of Statistics
```

Someone may have trouble distinguishing between Doe or Smith Doe as the author's surname. The element identifiers <*personname*>, <*firstname*>, and <*surname*> provide metadata about the data, i.e., the meaning/role of each of the strings. We see from these element names that it is a person's name (as opposed to a corporate name), that the person's first name is `Jane`, and her surname is `Smith Doe`.

Of course, there are many other self-describing formats. For example, name:value pairs also provide metadata, e.g.,

```
firstname:Jane
surname:Smith Doe
email:janesd@gmail.com
orgname:University of California
orgdiv:Department of Statistics
```

However, this approach is not as rich as *DocBook*'s structure, which allows complex nesting of elements.

We could achieve this with *JSON*. However, nobody would ever suggest authoring a book in *JSON*, primarily because text needs to be within quotes. Also, *JSON* does not have a way to specify the character encoding, but assumes/demands Unicode. Similarly, it does not support schema for validating and describing documents.

Note also, the meta information in *DocBook* does not contain instructions for formatting the data, e.g., that the author's name should appear in italicized font. In general, *XML separates the content and structure from the way the information is rendered.* We process the semantic information with separate information (e.g., *XSL* style sheets [31]) for how to render it for different audiences. This is much the same way we have learned to use *CSS* [2] for controlling the appearance of *HTML* and reducing or eliminating formatting information in the *HTML*.

While we have not formally introduced the *R* functions for working with *XML* content, it is useful to note that the rich structure and formal grammar of *XML* makes it easy to work with *XML* documents. For example, we can find all <*email*> elements, or all <*r:func*> or <*r:package*> nodes. We can even locate the <*section*> node in a book which is, e.g., a) within a chapter whose title contains the phrase `"social network"` and b) which has a paragraph with <*r:code*> that contains a call to load the graph package [10]. These are significantly harder to do robustly with markup languages such as LaTeX or *Markdown* [16] since they do not have formal grammars. Typically, people use line-oriented regular expressions for querying such documents and so cannot use the hierarchical context to locate nodes. This also makes it much harder to programmatically update content.

Our <*dataFrame*> example from earlier in the chapter also suggests how *XML* can be self-contained and self-describing. Rather than requiring the consumer of the data to both know and specify information about the structure of the data, software can determine this information by reading it from the *XML* document itself. We can read the number of observations and the types of the variables, obtain the names of the variables, determine the missing value symbol, and identify each separate data set within the document from *XML* markup. Also, we automatically determine the character encoding when parsing the *XML* document. Contrast this with CSV files and calls to read.table() and read.csv(). The extensibility of *XML* means that we can easily add new metadata to an *XML* representation and allow clients to query this as they make sense of the content.

The previous example illustrated *DocBook*'s choice of simple words for tag names that suggests its meaning or purpose. We saw in Section 2.1 that some names appeared to contain a colon, e.g., `<r:code>`. In fact, the name `<r:code>` is made up of two terms—`code` and the `r` prefix—separated via the `:` character. The `r` prefix identifies a namespace. *XML* namespaces have a role similar to package namespaces in *R*. They allow us to avoid (potential) conflicts from using the same name in different contexts or with different meaning, specifically when we mix element names from two or more different *XML* vocabularies. In *R*, we can refer to a function in either of two packages with `pkg1::aFunc` or `pkg2::aFunc`. We have a similar problem if we use an element named `<code>` in *XML*. Does this mean *R* code, *C* code, shell code, or code in any other language? We could use `<rcode>` for the element, but *Ruby* programmers might use that also to refer to *Ruby* code. There is still a conflict. We could use a *URL* uniquely identifying the project, e.g., `<r-project.org_code>`. This would remove the conflict, but be very verbose, tedious, and error-prone. Instead, *XML* namespaces provide a more robust, flexible, and richer approach to disambiguate conflicts. Within the *XML* document, we define an *XML* namespace as a pair consisting of a prefix and the uniquely identifying URI, e.g., `r` and `www.r-project.org`. Then, we can use the prefix to qualify the element name, e.g., `<r:code>`. We can define the prefix-URI mapping in an element using the form `xmlns:prefix=URI`, e.g.,

```
<dataFrame xmlns:r="http://www.r-project.org">...
```

This looks like an attribute, but is technically different. We can use the prefix in the node in which it is defined or in any of its descendant nodes, i.e., its child nodes, their children, and so on. In addition to making the node names shorter, the mapping of the URI to a prefix allows us, the authors of the *XML* document, to chose the prefix or to select which is the default namespace. The next example, provides another illustration of namespaces.

Example 2-2 A Climate Science Modelling Language (CSML) Document

The Climate Science Modelling Language (CSML `http://ndg.nerc.ac.uk/csml`) was developed by the British Atmospheric Data Centre and British Oceanographic Data Centre through the UK's Natural Environment Research Council's (NERC) DataGrid project [20]. Rather than starting from scratch to create their vocabulary for climate science modeling, NERC built on an existing grammar, Geographic Markup Language (GML), which already had many of the needed *XML* elements and features for CSML. This is an example of the *extensibility* represented by the "X" in *XML*.

The following snippet of CSML data contains daily rainfall measurements at a specified location for each day in the month of January. These measurements (5 3 10 1 2 ...) are the text content of the `<gml:QuantityList>` element, which is an element within `<PointSeriesFeature>`.[2]

```
<gml:featureMember>
 <PointSeriesFeature gml:id="feat02">
  <gml:description>
   January timeseries of raingauge measurements
  </gml:description>
  <PointSeriesDomain>
   <domainReference>
    <Trajectory srsName="urn:EPSG:geographicCRS:4979">
     <locations>0.1 1.5 25</locations>
     <times frame="#RefSys01"> 1 2 3 4 5 6 7 8 9 10 11
             12 13 14 15 16 17 18 19 20 21 22 23
```

[2] The values can be marked up individually, e.g., within `<value>` or `<double>` elements. However, since the values do not contain white space, we can separate them by spaces and recover them faithfully.

```
              24 2 26 27 28 29 30 31
      </times>
     </Trajectory>
    </domainReference>
   </PointSeriesDomain>
   <gml:rangeSet>
    <gml:QuantityList uom="udunits.xml#mm">
          5 3 10 1 2 8 10 2 5 10 20 21 12 3 5 19 12
          23 32 10 8 8 2 0 0 1 5 6 10 17 20
    </gml:QuantityList>
   </gml:rangeSet>
   <parameter xlink:href="#rainfall"></parameter>
  </PointSeriesFeature>
</gml:featureMember>
```

Notice that the tag name, <*gml:QuantityList*>, begins with the prefix `gml:` while <*PointSeriesFeature*> has no prefix. The prefix distinguishes GML element identifiers from CSML element identifiers. In creating CSML, NERC began with the element names and structure of GML and added to it tags needed for climate science. We can simply add new element and attribute names to an *XML* vocabulary without using a namespace. However, if we use the same name for a different concept, we have a conflict. We use a namespace to avoid this conflict.

The creators of CSML decided to clearly separate the additional CSML elements by using a separate namespace for them. The sample *XML* suggests that this situation is reversed, i.e., that the GML elements have a namespace prefix and the CSML elements do not. In fact, both sets of elements have a namespace, but the CSML elements use the default namespace (defined earlier in the document). The majority of the elements come from the CSML vocabulary so we use that as the default namespace to reduce the number of prefixes. We can equivalently use explicit namespace prefixes for both sets of elements, or use the GML namespace as the default and so qualify <*PointSeriesFeature*>, <*PointSeriesDomain*>, etc. We can use any prefix, and are not required to use the name of the vocabulary, e.g., `"gml"`. The namespace is defined by specifying a URI and a document-local prefix.

We leave this example with one final observation. The CSML snippet shows several layers of nesting. We use indentation to make the nesting of the elements clear and emphasize the tree structure. This *hierarchical structure gives great flexibility in describing complex data structures*. We can represent linear, tree, and even graph structures easily with *XML*. This generality, flexibility, and expressiveness make *XML* useful.

Example 2-3 A Statistical Data and Metadata Exchange (SDMX) Exchange Rate Document

The European Central Bank (ECB) provides daily foreign exchange rates between the euro and the most common currencies [7]. These are provided in several file formats, including an *HTML* format for the iPhone and two *XML* formats. Both *XML* formats were developed in accordance with the Statistical Data and Metadata Exchange initiative [6]. These foreign exchange reference rates (eurofxref, for short) use both the SDMX-EDI (GESMES/TS, GEneric Statistical MESsage for Time Series) format (`http://sdmx.org/`) and the ECB's extension vocabulary. The ECB uses this format to exchange data with its partners in the European System of Central Banks. According to them, this format "was a key element in the statistical preparations for Monetary Union and has proved both efficient and effective in meeting the ESCB's rapidly evolving statistical requirements" [8].

An *XML* snippet of exchange rates for four currencies on two days is shown below.[3] Notice the extensive use of attributes in this vocabulary and little use of text elements for representing data. The attributes *time*, *currency*, and *rate* contain, respectively, the date, name of the currency, and exchange rate to buy one euro in this currency.

```
<Envelope>
 <subject>Reference rates</subject>
 <Sender>
  <name>European Central Bank</name>
 </Sender>
 <Cube>
  <Cube time="2008-04-21">
   <Cube currency="USD" rate="1.5898"/>
   <Cube currency="JPY" rate="164.43"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="25.091"/>
  </Cube>
  <Cube time="2008-04-17">
   <Cube currency="USD" rate="1.5872"/>
   <Cube currency="JPY" rate="162.74"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="24.975"/>
  </Cube>
 </Cube>
</Envelope>
```

This document appears quite different from the *XML* in Example 2-2 (page 32). Here the snippet shows three levels of tags with the same name, i.e., <*Cube*>, and each has no text content. All of the relevant information is contained in the attribute values of the <*Cube*> elements. There is one parent <*Cube*> that holds all of the others. The next layer of <*Cube*> elements pertain to the date; there is one <*Cube*> element for each day, where the *time* attribute identifies the specific date, e.g., "2008-04-17". Within each of these "time" cubes are four <*Cube*> elements, one for each currency (US dollar, Japanese yen, Bulgarian lev, and Czech koruna). These innermost <*Cube*>s provide the name of the currency in *currency* and the exchange rate for that currency in *rate*. The exchange rate is for the date found in the parent <*Cube*> in which the element is nested. The <*Cube*> element corresponds to a multidimensional data cube, which represent the dimensions in a generic way. The values are grouped by time, space/geography, and other variables as <*Cube*>s, with arbitrary metadata associated with each dimension.

The eurofxref grammar uses the SDMX cube model for data. That is, the data are viewed as an n-dimensional object where the value of each dimension is derived from a hierarchy. According to SDMX [30]: "The utility of such cube systems is that it is possible to 'roll up' or 'drill down' each of the hierarchy levels for each of the dimensions to specify the level of granularity required to give a 'view' of the data."

This grammar offers an example where the attributes contain information used to differentiate one tag with the same name from another. The regular structure of the document is clear and can be

[3] The gesmes prefix on the <*Envelope*>, <*subject*>, <*Sender*>, and <*name*> nodes has been omitted to focus on the structure of the document. These tags are part of the SDMX-EDI (also known as GESMES) grammar. The <*Cube*> tags belong to the eurofxref extension vocabulary.

easily exchanged with other applications. For comparison, an alternative format would be to use, e.g., <date> and <rate> tags where the <rate> has a currency attribute as follows:

```
<Cube>
 <date>
   2008-04-21
   <rate currency="USD">1.5898</rate>
   <rate currency="JPY">164.43</rate>
 </date>
</Cube>
```

Of course, there are many other possibilities for structuring this information. Provided it follows the syntactical requirements of *XML*, you can develop any vocabulary you want. However, the consumers must be able to understand and make sense of the content.

2.3.1 A Discussion of XML Features

The eXtensible Markup Language is a standard for the semantic, hierarchical representation of data. It provides a general framework for supplying meta-information to representations of data that enables greater meaning. *XML*'s regular structure and strict parsing rules allow programmers to reuse the same generic and standard tools and computational models for working with very different *XML* vocabularies for different applications and domains. This means that we can focus on the meaning of the data and much less on its structure and how to process it. The applications that are centered around *XML* are very exciting and of immense potential significance to data analysts. Although using *XML* makes implementing software easier in many cases, it is the possibilities that it provides for end-users that make it more than just a software development technology. *XML* allows data analysts to think about data in new ways *because of the metadata and the structure for complex data.*

So, what are the drawbacks to *XML*? Like all pieces of software, it is not a silver bullet that solves all problems and introduces no new ones. In fact, the strengths of *XML* are also the sources of its problems. Although *XML* is structured to make it easy for a computer to read, this structure leads to relatively verbose content. This can be overcome by compressing the *XML* content. Since *XML* is simple text and human-readable, we are tempted to edit it manually. However, many people feel that *XML* is awkward and cumbersome to work with directly. This means tools to visualize and manipulate *XML* content are highly desirable, i.e., software that reads and writes the *XML* and insulates us from the details. This is not a simple problem because *XML* is extremely general (another one of its features!). However, many communities have been developing tools for viewing their own types of data, and often these tools can be shared across communities and applications. Fortunately, some general tools for editing and reading *XML* have emerged and can be easily customized to various needs. Furthermore, because of the plethora of general *XML* tools and technologies, we can build customized tools to manipulate *XML* content. This allows us to avoid working with the *XML* directly, and also makes the process more robust and reproducible.

Some people contrast *XML* and *JSON* as formats for data exchange. There are facilities in most languages for reading *JSON* directly into data structures in that language. The *JSON* format only has the standard and common data structures such as logicals/booleans, real numbers, strings, and ordered and associative arrays. There is a natural and simple mapping from *JSON* to native data

structures that needs no human interpretation. We can rearrange the data after reading them into these basic structures. In contrast, while most languages allow us to readily parse an *XML* document, there is another step in interpreting the *XML* data and mapping it to appropriate data structures in the language. Many *XML* documents are quite simple and map directly to native data structures, e.g., lists and data frames. However, since we can define complex data structures in *XML*, it is natural that we have to explicitly map these to more complex native data structures in different languages. This is not really different from post-processing *JSON* data into different and more complex data structures. For *XML* however, we can programmatically read *XML* schemas, which describe classes of *XML* documents. These tell us about the data types of different *XML* elements, using the much richer set of data types available in *XML* such as different types of numbers, times, dates, tokens, and identifiers. We can also use the schema to generate code and data type definitions that can read and convert *XML* documents to complex data types in our language.

2.4 Hierarchical Structure

The conceptual model of an *XML* document as a tree can be very helpful when processing and navigating the document. In fact, one of the two standards for parsing *XML* documents is the Document Object Model, or DOM for short, which reads the *XML* content and returns a data structure that represents the document in a hierarchical form (see Chapter 3).

The *XML* Tree Structure and Terminology

The hierarchical nature of *XML* can be represented as a tree. In the tree, the lines connecting elements are called branches, and the elements are referred to as nodes. The names used to describe relationships between nodes are modeled after family relations.

root The sole node at the "top" of the tree.

branch A directed edge that links one node to another. Branches emanate from the root "downward".

parent, *child* The branch between two nodes links a parent to a child node. The parent is one step higher in the hierarchy, i.e., closer to the root node. There are no branches between siblings in the tree.

ancestor, *descendant* Any node on the path up the tree from a node to the root is an ancestor of that node, and that node is a descendant of its parent and its parent's ancestors.

sibling Nodes with the same parent.

leaf-node Nodes with no children, also known as *terminal* nodes. Text nodes do not have children and so are leaf nodes. Regular *XML* elements may not have children and so are also leaf nodes.

The *depth* of a node is the number of branches to traverse from the root to that node. The root node is at depth zero. The set of all nodes at a given depth is called a level of the tree.

The nesting syntax rules for *XML* (i.e., opening and closing tags must be balanced) means that elements in a document must be hierarchically structured. Each element can be thought of as a node in the hierarchy where branches emanate from the node to those elements that it immediately contains. Figure 2.1 shows the tree representation of the *XML* document of Example 2-2 (page 32).

The *root* of this tree, also referred to as the *document node*, is `<gml:featureMember>`. As mentioned earlier, there is only one root per document. In this case, the document node has only one child, `<PointSeriesFeature>`, which has four children (grandchildren of the document

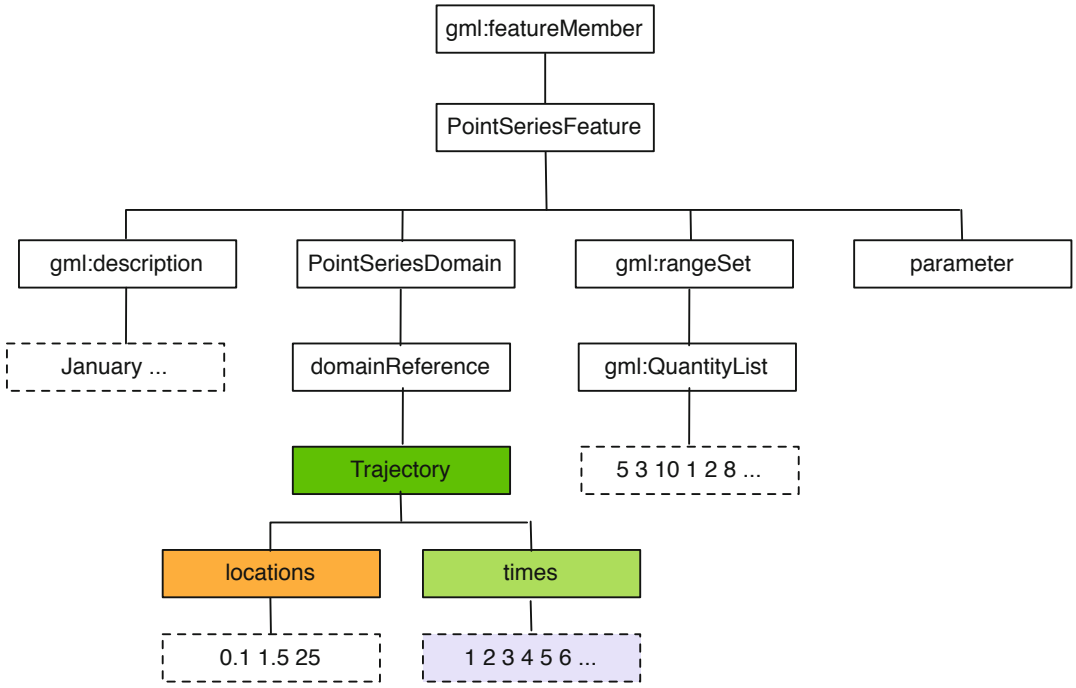


Figure 2.1: Example of a CSML Document Tree. This tree provides a conceptual model for the organization of data found in the CSML document in Example 2-2 (page 32). Each element is represented by a node in the tree and the branches from one node to another show the connections between the nodes. The <times> element (shown in light green) is a child of the <Trajectory> element (dark green). That is, <times> is nested directly under <Trajectory>, and the branch between these nodes in the graph indicates this connection. Additionally, the <locations> node (in orange) is a sibling to <times>. Text nodes are part of the hierarchy and are displayed via nodes with dashed borders, e.g., the purple node is a text child of <times>.

node)—one each of <gml:description>, <PointSeriesDomain>, <gml:rangeSet>, and <parameter>.

We use family tree terminology to describe the relative position of nodes in the tree. For example, consider the <times> node shown in green in the figure. Its *parent* is <Trajectory> (shown in dark green), and reciprocally, <times> is a *child* of <Trajectory>. The <locations> node (shown in orange) is a *sibling* to <times>. Also, <Trajectory>, <domainReference>, <PointSeriesDomain>, <PointSeriesFeature>, and <gml:featureMember> are all *ancestors* of <times>.

With all trees, the document node is the ancestor of all other nodes, and all other nodes are said to be *descendants* of this node. Additionally, the character content of an element is placed in a "text" node in the tree. In our example tree, the text 1 2 3 4 5 6 ... shown in purple is a child node of <times>. The *terminal* nodes in a tree at the end of the branches, which have no children, are known as *leaf nodes*. By design, any text content will always be a leaf node because text cannot contain an element. Of course, an element with no content will be a terminal or leaf node, e.g., <r/> or <Cube currency="USD" rate="1.5872"/>.

The examples that we examined in Section 2.3 take quite different approaches to organizing data. In the CSML example, the numeric data values are provided in "batches" in the text content (i.e., as a single text node with the values separated by white space, e.g., `"5 3 10"`), and metadata, such as the units of measurement, the names of the variables, etc., are provided through additional tags and attributes. Alternatively, the exchange rate data values are provided through element attribute values, with one datum per attribute. (See Figure 2.2.) The `<Cube>` element is used for multiple purposes, and it is through the attribute names that the type of information, or dimension of the data cube being provided, is ascertained. Note also that the eurofxref document is four nodes deep, in comparison to the CSML tree which is seven nodes deep. This is partly due to the use of attributes rather than child nodes to represent the data and to the additional `<PointSeriesFeature>` element under the root of the tree.

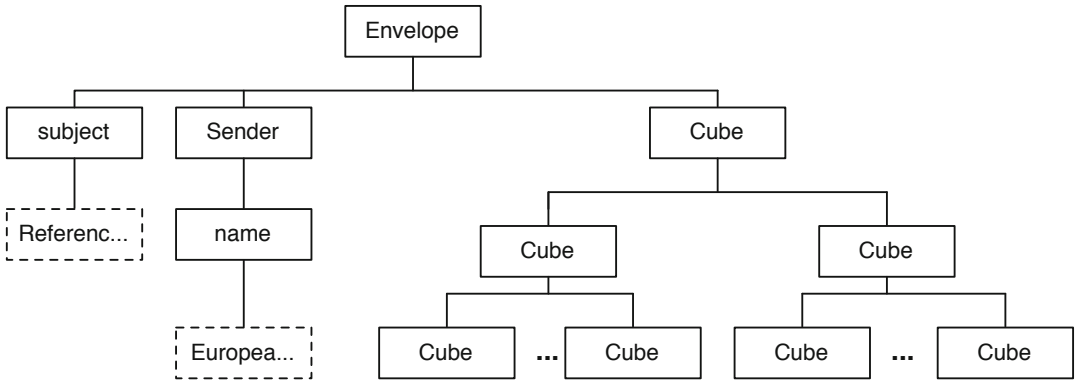


Figure 2.2: Example of an SDMX Document Tree. The tree shown here is a representation of the hierarchy of the SDMX document in Example 2-3 (page 33). Each element is a node in the tree, with text content represented by a dashed border on the node. The exchange rates, dates, and type of currency are found in the attribute values of the nodes, which are not shown in the diagram. These omissions are made merely for simplicity in the presentation of the tree.

Whether to use attributes or child nodes to represent data values is a matter of style. In general, attributes are often used to provide information about the data. There are no required rules about when to use attributes and when to use elements. However, if we have multiple data values, we cannot use the same attribute for each one separately. Instead, to use an attribute, we have to combine them into a single string, separated by some delimiter, e.g., `"5,3,10"`. Here it is often easier to separate child nodes, e.g.,

```
<value>5</value>
<value>3</value>
<value>10</value>
```

Similarly, if the data values are not simple primitive values, but instead have some complex structure, we cannot easily use attributes. A simple rule-of-thumb is that it is best to use attributes for information about the data, i.e., metadata, rather than the actual data values. For example, when we refer to an *R* function in our extended *DocBook* vocabulary, we add the name of the package as an attribute, e.g., `<r:func pkg="XML">xmlParse</r:func>`. Of course, the distinction between metadata and data is not always clear.

2.5 Additional *XML* Elements

While the element or node is the primary component in *XML*, the language also includes several other, somewhat less common, but useful constructs. These are the *XML* declaration, processing instructions, comments, <*CDATA*> section delimiters, entity references, and document-type declarations. In this section, we briefly describe each of these.

Overview of Additional *XML* Markup

CDATA Character data that is treated by the *XML* parser as literal text and not *XML* content. This is used to "escape" content that happens to contain special *XML* characters such as < and &, and treat it as verbatim text. Markup between the delimiters `<![CDATA[` and `]]>` is not processed, e.g.,

```
<![CDATA[ x <- y > 10 & z < 20 ]]>
```

Comment A block of text that is not considered part of the data itself, but informal information about the document. A comment appears between the delimiters `<!--` and `-->`, e.g.,

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

XML Declaration Optional statement at the beginning of a document that identifies the content as an *XML* document and provides additional information about the character encoding and the version of *XML*, e.g.,

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

Processing Instruction Optional hint or instructions to an application that might parse the *XML* document. It is a way for us to include directives to a target application so that it will do something as it processes the *XML* nodes, such as use a particular style sheet to render the document. A processing instruction has two parts—the name of the target application and the instruction. The target application is a simple string, while the instruction can be any sequence of text and is entirely application-specific. For example, we can set the width option in *R* with

```
<?R options(width = 140) ?>
```

Other applications reading the *XML* will ignore this. One can include the same information within regular *XML* elements, but processing instructions are a powerful and convenient way to specify application-specific information without adding to the *XML* vocabulary used for the data.

Document-type Declaration A declaration at the start of the document (before the root node) that identifies the "type" of the document, e.g.,

```
<!DOCTYPE html>
```

The type can either be one of the known types such as `html` or `xhtml`, or can also specify the location of an external *DTD* (Document Type Definition) document that describes the structure of a valid document of this type. Some or all of the *DTD* can also be inlined within the <*DOCTYPE*> node, and extensions and character entities can be added.

CDATA and Entities

When working to get data from *XML* documents, we often come across two other *XML* constructs—

entities and <*CDATA*> (the <*CDATA*> construct stands for character data). The good news is that the *XML* parser typically allows us to transparently work with these as regular text. However, it is good to be able to recognize and understand them and why they are necessary. Since the character < is used to indicate the start of an *XML* tag name, how can we write an inequality such as `x < y`? We need a way to "escape" or protect the < from the *XML* parser so that it does not think the < starts a new *XML* element. For this, we use the entity *<*, that is, an ampersand (&), the shorthand `lt` for "less than", and we end the name of the entity with a semicolon. Any entity is introduced with an ampersand and ended with a semicolon. *XML* defines several standard entities; the most commonly used are *<* for < and *&* for &. We need an entity for & itself since we cannot use the literal & character as *XML* thinks that starts an entity! We can use entities generally in *XML* as macros or text substitutions and also for specifying nonstandard characters, e.g., accents such as the cedilla under the character c (ç) with *ç*, or characters in other alphabets such as the Greek letter alpha with *α*, or for special symbols such as the copyright symbol (©) with *©*. These can also be inserted directly into the *XML* without entities. The *XML* parser can keep entities as special objects or just replace them with the corresponding text.

Entities are especially important when we have code within the *XML* document because many programming languages such as *R* and *C* make common use of the < and & characters. Rather than using entities for each instance of < and & within a complex piece of code, we can "escape" an entire block of text from the *XML* parser using a <*CDATA*> section. We start such a section with `<![CDATA[`, and we end it with `]]>`. For example,

```
<r:code>
<![CDATA[
if(all(x) > 0 && max(y) < 10)
   z = log(x) > 1 & y > median(y)
]]>
</r:code>
```

When the *XML* parser encounters the start of the <*CDATA*> section, it reads the characters up to the end of the <*CDATA*> section as regular text. As a result, we see this text not within a special *XML* construct, but as a simple text node.

Comments

XML allows comments to be included within a document. These are meant to provide information about (part of) the document. These are not necessarily structured in any particular way, although one could use some convention to include the information. However, it is much more sensible to include that information as regular *XML* if it is to be interpreted. Accordingly, comments are usually just free-form descriptions or notes about the content. A parser can ignore comments or keep them in the parsed *XML* tree.

A comment is introduced with `<!--` and ends with `-->`. Comments can contain the symbols < and > because all text between the delimiters is ignored by the *XML* processor and so is neither rendered nor read. For example, the following *KML* contains a comment in the <*Point*> element.

```
<Placemark id="2">
 <name>2</name>
 <Point>
    <!-- This comment is a reminder to
         check that -154 > minimum. -->
  <coordinates>-154.577,56.961,0</coordinates>
```

```
  </Point>
</Placemark>
```

Comments can appear anywhere after the *XML* declaration. They can even appear outside the root element of the document immediately following the *XML* declaration.

XML Declaration

Typically, an *XML* document starts with an *XML* declaration that identifies it as an *XML* document and provides, most importantly, the information defining the character encoding, e.g., UTF-8. The declaration must also specify the version of *XML*, which currently is always 1.0. For example, the following declaration,

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

indicates that version 1.0 of *XML* is being used and that the character encoding is UTF-8. The most common encodings for *XML* are UTF-8, UTF-16, and ISO-8559-1 (a.k.a. Latin1), which all *XML* processors support. The declaration appears first in the document, followed by the root element, i.e., the document appears as

```
<?xml version = "1.0" encoding="UTF-8" ?>
<root>
  ...
</root>
```

Although this declaration is not required, it is a good way to clearly identify a document as *XML*. One should always specify the character encoding to help consumers of the *XML* determine how to interpret the sequence of bytes as strings. It is essential when one uses non-ASCII characters.

Processing Instruction

A processing instruction (PI) is a directive or instruction within the *XML* content to a particular target application that might be parsing the document. If another application is processing the document, it will ignore PIs not meant for it. The idea of a PI is to be able to give an application-specific command to the parser to change its state or have it do something.

A PI identifies the target application and is followed by the command. This information is enclosed within an opening `<?` and a closing `?>`, e.g.,

```
<?R options(width = 140)?>
<?xml-stylesheet type="text/xsl" href="XSL/Todo.xsl" ?>
```

(This is similar to the form of the *XML* declaration above, but that is technically not a processing instruction.) In these examples, we have an instruction for *R* and another for an application named xml-stylesheet. The latter is intended for a Web browser. The information after the target application gives two attribute-like settings. The type identifies an *XSL* document which can be used to transform the *XML* document into *HTML*. The *href* "attribute" identifies the location of the *XSL*. A capable Web browser will then use this to transform the *XML* document and show the resulting *HTML* document in its place.

An xml-stylesheet processing instruction must be placed between the *XML* declaration and the root element of the document. Other processing instructions may also be placed there or at other locations in the document. The name–value pair format of the xml-stylesheet, e.g., `type="text/css"`, imitates the syntax for attributes. In general, the content can be any text values expected by the application, i.e., different applications support different formats for the processing instructions.

Each application will chose which target applications it will recognize and it will operate only on `<PI>`s for those applications and ignore the rest. The `<PI>`s are only hints as a parser may ignore them entirely.

Document-type Declaration

We have so far described the requirements for a document to be well formed, i.e., it meets the generic syntax requirements of *XML*. In addition to being well-formed, a class of documents may only make sense if there is a specific relationship between the elements. When a document meets these application-specific requirements it is said to be *valid*. With *HTML5*, this information is specified via the document-type declaration (*DTD*), as follows

```
<!DOCTYPE html>
```

Another approach to defining an application-specific vocabulary and rules is with *XML* Schema. Details on *DTD*s and *XML* Schema appear Section 2.7.

2.6 *XML* Namespaces

Applications often build on or extend other *XML* grammars, adding their own tag names to those of one or more well-established standards. For example, *RDocBook* extends the *DocBook* vocabulary with element names for *R* expressions, function and package names, code, etc. As another example, Example 2-2 (page 32) showed an *XML* document with GML and CSML vocabularies. This flexibility is one of the strengths of *XML*. Occasionally, two vocabularies use the same name to mean different things, and when this happens, *namespaces* [3] provide a mechanism to avoid conflicts. A namespace prefix qualifies an element name and connects it to a particular vocabulary (i.e., namespace). Example 2-3 (page 33) included element names from two vocabularies, the GESMES vocabulary and the eurofxref namespace. In that example, we ignored these namespaces for simplicity. However, the actual *XML* appears as:

```
<gesmes:Envelope
 xmlns:gesmes="http://www.gesmes.org/xml/2002-08-01"
 xmlns="http://www.ecb.int/vocabulary/2002-08-01/eurofxref">
 <gesmes:subject>Reference rates</gesmes:subject>
 <gesmes:Sender>
   <gesmes:name>European Central Bank</gesmes:name>
 </gesmes:Sender>
 <Cube>
  <Cube time="2008-04-21">
   <Cube currency="USD" rate="1.5898"/>
   <Cube currency="JPY" rate="164.43"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="25.091"/>
  </Cube>
  <Cube time="2008-04-17">
   <Cube currency="USD" rate="1.5872"/>
   <Cube currency="JPY" rate="162.74"/>
   <Cube currency="BGN" rate="1.9558"/>
   <Cube currency="CZK" rate="24.975"/>
  </Cube>
 </Cube>
</gesmes:Envelope>
```

A namespace is associated with a prefix via a namespace definition, which appears in the *xmlns* "attribute" of an element. (Technically, *xmlns* is not an attribute, but the syntax for specifying it is the same as for attributes.) The *Envelope* element defines the two namespaces. The prefix for the GESMES namespace is `gesmes` and the `<Envelope>` element uses this prefix to indicate it belongs to that namespace, i.e., `<gesmes:Envelope>`. Additionally, any descendant of `<gesmes:Envelope>` that has a `gesmes` prefix is associated with the GESMES namespace. The eurofxref grammar is also defined in `<gesmes:Envelope>`, but a prefix is not provided. In this case, the eurofxref namespace is treated as the default. That is, any child element of `<gesmes:Envelope>` that does not have a prefix belongs to the euroxref namespace. This convention can be very convenient because it saves us from having to add prefixes to node names.

***XML* Namespaces**

Namespaces provide a mechanism to differentiate element names from multiple vocabularies in a document.

A prefix is associated with a namespace via a namespace definition, and this prefix is prepended to the element name for those elements belonging to the associated grammar. The namespace definition can be provided in the start tag of any element and is available to that node and all of its descendants.

When a prefix is not used in a namespace definition, that namespace is taken as the default. In the following example,

```
<article xmlns="http://docbook.org/ns/docbook"
         xmlns:r="http://www.r-project.org">
...
<para>The expression <r:expr>runif(3)</r:expr> ...
```

two namespaces are defined; the first has no prefix and so is the default. Both `<article>` and `<para>` are associated with this namespace. The second namespace is associated with the "r-project" URI. It uses the prefix `r`, and the `<expr>` element is associated with it.

In general, the syntax for adding a namespace definition to an element is:

```
xmlns:prefix="URI"
```

The URI in the namespace definition is used to identify the vocabulary; it is not used to look up information about the vocabulary. However in practice, companies often use it as a pointer to a Web page containing information about the grammar.

It is often convenient to define all namespaces in the root element of the document so that they are available to all descendants, but namespace definitions can appear on any element. Also, if one document includes another, then it is not always possible to follow this simple rule as the included document may have its own namespace definitions. When this happens, we need rules for multiply defined namespaces, e.g., one namespace given two different prefixes or one prefix used for two different namespaces. Below are some rules for handling these situations:

- When a namespace prefix is reused in a namespace definition, then that element and all of its descendants will associate the prefix with the namespace in the new definition.
- A namespace can be redefined with a new prefix. When this happens, to associate the namespace with the element or any of its descendants, the new prefix must be used. This can occur when one document fragment is inserted into another document and the two use different prefixes for the same namespace.

- If a default namespace is not defined, then an element without a prefix will have no namespace associated it.

The *XML* below provides examples of these cases.

```
<article   xmlns="http://docbook.org/ns/docbook"  1
         xmlns:r="http://www.r-project.org">  2
<example>
<db:para  xmlns:db="http://docbook.org/ns/docbook"  3
       xmlns="http://www.gesmes.org/xml/2002-08-01">
The <db:emphasis>code</db:emphasis> below -
<r:code>  4
 x = runif(1)
</r:code>
is for the random uniform, and the following:
<s:code xmlns:s="http://www.r-project.org">  5
 z = rnorm(3)
<s:output>
[1] -1.0651854 -0.9383899 -0.4249840
</s:output>
</s:code>
is a normal random number generator.
</db:para>
<para>A short paragraph.</para>  6
<example>
</article>
```

1 The default namespace is defined as *DocBook* on <*article*>, and since the <*article*> element has no prefix, it belongs to this namespace.
2 A second namespace with a prefix of r is also defined on <*article*>. It is associated with the www.r-project.org namespace.
3 The default namespace has been changed and *DocBook*, no longer the default, has been given the db prefix. This prefix must be used with this node and all its descendants, if they are to be associated with *DocBook*.
4 The www.r-project.org namespace is associated with this <*code*> element.
5 The www.r-project.org namespace has been associated with a new prefix, s, and so this <*code*> element and its child <*output*> must use the prefix s if they are to be associated with the www.r-project.org namespace.
6 This <*para*> is not a descendant of <*db:para*> where the default namespace was changed so *DocBook* is the default namespace and we do not need to use the prefix db.

Attributes can also use prefixes to associate the attribute with a namespace. Attributes are usually associated with the element to which they belong, but depending on the application they may not have a namespace associated with them. For example, the following elements,

```
<r:func name="runif"/>
<r:func r:name="runif"/>
```

are not necessarily equivalent. The former does not explicitly associate the “r” namespace with the attribute *name*, while the latter does. One application may treat the two as equivalent, while another may not.

2.7 Describing the Structure of Classes of *XML* Documents: Schema and *DTD*s

When creating a grammar, we need a mechanism to tell people (and machines) what the format permits. Obviously, we can show them some examples. However, if we want to illustrate the format in its full generality, we will need a multitude of examples for all combinations of possibilities. Instead, we provide a set of constraints or definitions that limit the element and attribute names, the data types for attribute values and element content, and the allowable containment hierarchies of the elements. There are two common approaches for doing this; one uses *XML Schema* [9] and the other a *Document Type Definition* (*DTD*) [40]. An *XML* document that complies with a particular schema or *DTD*, in addition to being well-formed, is said to be *valid*.

We describe both *XML* Schema and *DTD*s in this section. Chapter 14 discusses the functionality to read and process *XML* schema that is available in the XMLSchema package [35].

2.7.1 The DTD

The oldest schema format for *XML* is the Document Type Definition (*DTD*). While *DTD* support is ubiquitous due to its inclusion in the *XML* 1.0 standard, it is seen as limited. For example, it uses a non-*XML* syntax, inherited from *SGML*, to describe the vocabulary. As a result, it is not as easily extensible, does not support namespaces, and lacks the expressiveness of the *XML* Schema Definition (XSD), such as the rich data typing and complex logical structure. However, the *DTD* is still used in many applications because it is easy to read and write, and for these reasons, we provide a brief example with a small sample of *DTD*.

Example 2-4 A DTD for XHTML

A simple example of *XML* is the eXtensible HyperText Markup Language (*XHTML*). Readers familiar with HyperText Markup Language (*HTML*), the language developed for presenting material on the Web, know that *HTML* need not be well-formed to be properly rendered in a browser. For example, there is no need to close a <*p*> tag and attribute values need not be enclosed in quotation marks. *XHTML* was developed to bring *HTML* in line with *XML* standards.

A *DTD* for *XHTML* is available at http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd. The strict standards for *HTML* require all element and attribute names to be lowercase. Another rule is that the root node must be <*html*> and it must contain a namespace definition that points to the *XHTML* URI: http://www.w3.org/1999/xhtml. The start of an *XHTML* document must include an *XML* declaration, a DOCTYPE declaration, and an <*html*> root with namespace declaration as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
```

Below is a sample of the *XHTML DTD*. It pertains to the ordered list element <*ol*> and its child, the <*li*> element.

```
<!-- Ordered (numbered) list -->
<!ELEMENT ol (li)+>
<!ATTLIST ol
```

```
  %attrs;
  >
<!-- list item -->
<!ELEMENT li %Flow;>
<!ATTLIST li
  %attrs;
  >
```

Notice the format of the *DTD*. The element names are defined via `<!ELEMENT` markup. The `"(li)+"` in `<!ELEMENT ol (li)+>` indicates that the `<ol>` tag can have only `<li>` children and it must have one or more of them.

We will not describe further the syntax of *DTD* as it is not our focus.

2.7.2 Schema

An *XML* parser has the job of reading the *XML*, checking it for errors, and passing it on to the intended application. If no schema (or *DTD*) is provided, the parser simply checks that the *XML* is well-formed. If a schema is provided, then the parser also determines whether the *XML* is valid before passing it on to the application. Models for parsing *XML* are described in greater detail in Chapters 3 and 5. The *XML Schema Definition* (XSD) is a schema for a particular grammar of *XML*. It defines the building blocks of an *XML* document: what elements can appear in the document; which attributes can appear in an element; which elements can be children of an element; the order and number of child elements; and which data types are allowed for element content and attribute values. With schema, the format of an *XML* document can be specified at a fairly high level; general parsing tools can be used to validate documents; and other applications can easily reuse, extend, and even combine schemas to cover specialized cases.

The schema is an *XML* document itself with its own grammar. The root element of an *XML* Schema Definition is the `<schema>` element, and following the root are elements that describe the allowable elements for the *XML* grammar being defined. These elements are defined via `<element>` tags. The value in the *name* attribute of `<element>` is the tag name in the grammar. The schema specifies whether, for example, the element is simple or complex. Recall, a simple element is one that contains only text content. The type of text content can be specified in the schema as a boolean, string, integer, date, etc., or it can be a custom type that is also defined in the schema.

Attributes for tags in the grammar being defined are specified through `<attribute>` tags. Again, the *name* attribute in the `<attribute>` tag supplies the name of the attribute in the element being defined. Other attributes of `<attribute>` are used to specify its data type, default value, and whether or not it is required.

A complex element is one that contains other elements. In the schema definition of complex elements, child nodes describe constraints on the hierarchy of the complex element. For example, the `<all>` node indicates that children can appear in any order and that each child element must occur only once; the `<sequence>` tag specifies that the children have to appear in a specified order; and the `<group>` tag is used to define related sets of elements. The `<any>` element indicates that any tag may be a child of the element being defined. The `<any>` tag makes it easy to extend an *XML* document with elements not defined specifically in the schema.

There are many more details to setting up a schema. We refer the interested reader to [38, 41]. Here we use an example to give only an idea of what is possible.

Example 2-5 Examining Schema for the Predictive Model Markup Language

The Predictive Model Markup Language (*PMML*) [5] is a language for representing statistical models in an application-independent way. Several applications support (*PMML*), including *ADAPA*, *CART*, *Clementine*, *Enterprise Miner*, *DB2 Intelligent Miner*, *R*, *Teradata Warehouse Miner*, and *Zementis*.

A PMML document must contain three main chunks: a header, a data dictionary, and a model as follows:

- The header provides general information about the model used in the application, such as the copyright and a non-application-specific description of the model.
- The data dictionary defines the variables used in the application of the model. It includes specifications of data types and value ranges. These data dictionaries can be shared across models.
- The model chunk consists of a mining scheme and a model specification. The mining schema lists fields that must be provided to use the model; this list can be a proper subset of the fields in the data dictionary. The model also contains information specific to the type of model; that is, the model specification is dependent on the type of model fitted. As an example, the tree model used for classification and prediction contains `<Node>` elements that hold the logical predicate expressions that define the rules for branching.

Below is the PMML document (with a shortened header) that results from fitting a classification tree to the `kyphosis` dataset using the rpart() function in *R*. The dependent variable in the fit is `Kyphosis`, which is a categorical variable with levels `"absent"` and `"present"`, and the independent variables are `Age`, `Number`, and `Start`. The *R* code to fit and output the fit as a PMML document are provided here:

```
fit = rpart(Kyphosis ~ Age + Number + Start, data = kyphosis)
saveXML(pmml(fit), file = "KyphosisRpart.pmml")
```

The pmml() function is in the `pmml` package. The saveXML() function is in the XML package, and is covered in greater detail in Chapter 6. The document, `KyphosisRpart.pmml`, follows.

```
<PMML version="3.1"
   xmlns="http://www.dmg.org/PMML-3_1"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Header copyright="Copyright..."
   description="RPart decision tree model">
   ...
 </Header>
 <DataDictionary numberOfFields="4">
  <DataField name="Kyphosis" optype="categorical"
      dataType="string">
   <Value value="absent"/>
   <Value value="present"/>
  </DataField>
  <DataField name="Age" optype="continuous" dataType="double"/>
  <DataField name="Number" optype="continuous" dataType="double"/>
  <DataField name="Start" optype="continuous" dataType="double"/>
 </DataDictionary>
 <TreeModel modelName="RPart_Model"
```

```
      functionName="classification" algorithmName="rpart"
      splitCharacteristic="binarySplit">
  <MiningSchema>
   <MiningField name="Kyphosis" usageType="predicted"/>
   <MiningField name="Age" usageType="active"/>
   <MiningField name="Number" usageType="active"/>
   <MiningField name="Start" usageType="active"/>
  </MiningSchema>
  <Node score="absent" recordCount="81">
   <True/>
   <Node score="absent" recordCount="62">
    <SimplePredicate field="Start"
      operator="greaterOrEqual" value="8.5"/>
    <Node score="absent" recordCount="29">
     <SimplePredicate field="Start"
        operator="greaterOrEqual" value="14.5"/>
    </Node>
    <Node score="absent" recordCount="33">
     <SimplePredicate field="Start"
         operator="lessThan" value="14.5"/>
     <Node score="absent" recordCount="12">
      <SimplePredicate field="Age"
         operator="lessThan" value="55"/>
     </Node>
     <Node score="absent" recordCount="21">
      <SimplePredicate field="Age"
          operator="greaterOrEqual" value="55"/>
      <Node score="absent" recordCount="14">
       <SimplePredicate field="Age"
          operator="greaterOrEqual" value="111"/>
      </Node>
      <Node score="present" recordCount="7">
       <SimplePredicate field="Age"
          operator="lessThan"  value="111"/>
      </Node>
     </Node>
    </Node>
   </Node>
   <Node score="present" recordCount="19">
    <SimplePredicate field="Start"
       operator="lessThan" value="8.5"/>
   </Node>
  </Node>
 </TreeModel>
</PMML>
```

The schema for the <*TreeModel*> element appears below. It gives a sense of how schema are used to provide the rules for a *valid* PMML document.

```
<xs:element name="TreeModel">  1
 <xs:complexType>  2
  <xs:sequence>  3
   <xs:element ref="Extension" minOccurs="0"  4
        maxOccurs="unbounded"/>
   <xs:element ref="MiningSchema"/>  5
   <xs:element ref="Output" minOccurs="0" />
   <xs:element ref="ModelStats" minOccurs="0"/>
   <xs:element ref="Targets" minOccurs="0" />
   <xs:element ref="LocalTransformations" minOccurs="0"/>
   <xs:element ref="Node"/>
   <xs:element ref="ModelVerification" minOccurs="0"/>
   <xs:element ref="Extension" minOccurs="0"
        maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="modelName" type="xs:string" />  6
  <xs:attribute name="functionName"
          type="MINING-FUNCTION" use="required"/>
  <xs:attribute name="algorithmName" type="xs:string" />
  <xs:attribute name="splitCharacteristic"
          default="multiSplit">  7
   <xs:simpleType>
    <xs:restriction base="xs:string">  8
     <xs:enumeration value="binarySplit"/>
     <xs:enumeration value="multiSplit"/>
    </xs:restriction>
   </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
</xs:element>
```

1 This <*element*> tag defines the <*TreeModel*> element and provides rules for its content.
2 The <*complexType*> child indicates that <*TreeModel*> has complex content.
3 According to the <*sequence*> child of <*complexType*>, the children of <*TreeModel*> must appear in the specified order: <*Extension*>, <*MiningSchema*>, <*Output*>,...
4 The *minOccurs* attribute has a value of "0" in the definition of <*Extension*>. This indicates that <*Extension*> is optional. Also, since *maxOccurs* is "unbounded" there may be arbitrarily many <*Extension*> elements in <*TreeModel*>.
5 *minOccurs* is not specified for <*MiningSchema*>, so the default of 1 is used.
6 The allowable attributes for <*TreeModel*> are provided via <*attribute*> elements. Here, the data type for the attribute *modelName* is any character data.
7 The attribute *splitCharacteristic* has a default value of "multiSplit".
8 In addition, <*restriction*> indicates that this attribute (*splitCharacteristic*) has only two possible values: "multiSplit" and "binarySplit".

The goal of this section was to introduce the basic concepts in *XML* Schema. The topic is revisited in Chapter 14 in the context of an application (the XMLSchema package [35]).

2.8 History of *XML*

XML originated as a simplification of the more comprehensive Structured General Markup Language (*SGML*) framework, developed by Goldfarb, Mosher, and Lorie of IBM in the 1970s and officially standardized in 1986 [17]. Like *XML*, *SGML* is not a markup language. It is a syntax or format for defining markup languages. When the Web was being designed, its creators developed *HTML* based on *SGML*. *HTML* is focused on display and presentation and has a fixed set of markup elements (e.g., `<h1>`, `<a>`, `<ol>`, `<li>`). In contrast, *SGML* allows us to define new elements and entire vocabularies. However, *SGML* is too flexible and complex for widespread use, and most applications of *SGML* only leverage a small part of its flexibility. *XML* is essentially the simple subset of *SGML* that has proved to be widely used and necessary. Terence Parr said of *XML* that "being an expert in *XML* is like being an expert in CSV" (comma-separated values) [28].

After the creation of *XML*, it was standardized and removed from the control of any one dominant commercial interest. Yet, the original companies, including Microsoft, IBM, and Sun, have adopted *XML* enthusiastically. This backing by these major companies, along with Oracle, SAS, Google, etc., and many smaller institutions, entities, and governments has been important to *XML*'s success. In addition, the availability of tools to work with *XML*, the spate of additional related technologies, and the subsequent derived technologies have been very important for its continued health.

2.9 Further Reading

There are several comprehensive books on *XML* and tools for working with *XML*, including [17, 18]. Also, [39] offers an online tutorial.

References

[1] Apple, Inc. Numbers for iOS: Supported file formats. http://support.apple.com/kb/HT4642, 2011.

[2] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. Cascading style sheets, level 2, revision 1 (CSS 2.1) specification. Worldwide Web Consortium, 2011. http://www.w3.org/TR/CSS2/.

[3] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry Thompson. Namespaces in *XML* 1.0. Worldwide Web Consortium, 2009. http://www.w3.org/TR/REC-xml-names/.

[4] James Clark. nXML mode: An addon for GNU Emacs. http://www.thaiopensource.com/nxml-mode/, 2004.

[5] Data Mining Group. Predictive Model Markup Language. http://www.dmg.org/pmml-v3-2.html, 2011.

[6] Economic Commission for Europe. Common open standards for the exchange and sharing of socio-economic data and metadata: The SDMX initiative. http://sdmx.org/docs/2002/wp11.pdf, 2002.

[7] European Central Bank. Euro foreign exchange reference rates. http://www.ecb.int/stats/exchange/eurofxref/html/index.en.html, 2011.

[8] European Central Bank. SDMX-ML and SDMX-EDI (GESMES/TS): The ECB statistical representation standards. http://www.ecb.int/stats/services/sdmx/html/index.en.html, 2011.

[9] David Fallside and Priscilla Walmsley. *XML* schema, Part 0: Primer. Worldwide Web Consortium, 2004. http://www.w3.org/TR/xmlschema-0/.

[10] R. Gentleman, Elizabeth Whalen, W. Huber, and S. Falcon. graph: A package to handle graph data structures. http://cran.r-project.org/package=graph, 2011. *R* package version 1.33.0.

[11] Google, Inc. Keyhole markup language (*KML*) reference. https://developers.google.com/kml/documentation/kmlreference, 2010.

[12] Google, Inc. Google Earth: A 3D virtual earth browser, version 6. http://www.google.com/earth/, 2011.

[13] Google, Inc. Google Maps: A Web mapping service application. http://maps.google.com/, 2011.

[14] Google, Inc. Google documents list API: Allows developers to create, retrieve, update, and delete Google Docs. http://code.google.com/apis/documents/, 2012.

[15] Google, Inc. Google Sky: An online outer-space viewer. http://www.google.com/sky/, 2012.

[16] John Gruber. Markdown: A text-to-*HTML* conversion tool for Web writers. http://daringfireball.net/projects/markdown/, 2004.

[17] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[18] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.

[19] Bill Kennedy and Chuck Musciano. *HTML and XHTML: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[20] B. N. Lawrence, R. Lowry, P. Miller, H. Snaith, and A. Woolf. Information in environmental data grids. *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences*, 367:1003–1014, 2009.

[21] LibreOffice; The Document Foundation. Calc: The LibreOffice spreadsheet program. http://www.libreoffice.org/features/calc/, 2011.

[22] R.G. Mann, R.M. Baxter, R. Carroll, Q. Wen, O.P. Buneman, B. Choi, W. Fan, R.W.O. Hutchison, and S.D. Viglas. *XML* Data in the virtual observatory. *Astronomical Data Analysis Software and Systems XIV*, 347:223, 2005.

[23] Deborah Nolan, Roger Peng, and Duncan Temple Lang. Enhanced dynamic documents for reproducible research. In M.F. Ochs, J.T. Casagrande, and R.V. Davuluri, editors, *Biomedical Informatics for Cancer Research*, pages 335–346. Springer-Verlag, New York, 2009.

[24] Deborah Nolan and Duncan Temple Lang. Learning from the statistician's lab notebook. In *Data and Context in Statistics Education: Towards an Evidence-based Society. Proceedings of the Eighth International Conference on Teaching Statistics (ICOTS8, July, 2010), Ljubljana, Slovenia.* Voorburg, 2010.

[25] Open Geospatial Consortium, Inc. OGC *KML* standards. http://www.opengeospatial.org/standards/kml/, 2010.

[26] Eric Raymond. *DocBook* demystification HOWTO, revision v1.3. The Linux Documentation Project, 2004. http://en.tldp.org/HOWTO/DocBook-Demystification-HOWTO/.

[27] Frank Rice. Introducing the Office (2007) Open *XML* file formats. http://msdn.microsoft.com/en-us/library/aa338205(v=office.12).aspx, 2006.

[28] Yakov Shafranovich. Common format and MIME type for comma-separated values (CSV) files. http://tools.ietf.org/html/rfc4180, 2011.

[29] Richard Stallman. GNU Emacs: An extensible, customizable text editor. http://www.gnu.org/software/emacs/, 2008.

[30] Statistical Data and Metadata Exchange Initiative. SDMX information model: UML conceptual design (version 2.0). http://www.sdmx.org/docs/2_0/SDMX_2_0SECTION_02_InformationModel.pdf, 2005.

[31] Bob Stayton. *DocBook XSL: The Complete Guide*. Sagehill Enterprises, Santa Cruz, CA, fourth edition, 2007.

[32] Alex Szalay, Jim Gray, Ani Thakar, Bill Boroski, Roy Gai, Nolan Li, Peter Kunszt, Tanu Malik, Wil O'Mullane, Maria Nieto-Santisteban, Jordan Raddick, Chris Stoughton, and Jan van den Berg. The SDSS DR1 SkyServer: Public access to a terabyte of astronomical data. http://cas.sdss.org/dr6/en/skyserver/paper/, 2002.

[33] Duncan Temple Lang. RTidyHTML: Tidy *HTML* documents. http://www.omegahat.org/RTidyHTML, 2011. *R* package version 0.2-1.

[34] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[35] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.

[36] United Nations Statistical Commission. Report on the thirty-ninth session. (Supplement No. 4, E/2008/24). http://unstats.un.org/unsd/statcom/doc08/DraftReport-English.pdf, 2008.

[37] US Food and Drug Administration. Structured product labeling resources. http://www.fda.gov/ForIndustry/DataStandards/StructuredProductLabeling/default.htm, 2012.

[38] Eric van der Vlist. *XML Schema*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[39] W3Schools, Inc. *XML* tutorial. http://www.w3schools.com/xml/default.asp, 2011.

[40] W3Schools, Inc. *DTD* tutorial. http://www.w3schools.com/dtd/default.asp, 2012.

[41] Priscilla Walmsley. *Definitive XML Schema*. Prentice Hall PTR, Upper Saddle River, NJ, 2001.

[42] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, first edition, 1999. http://www.docbook.org/tdg5/.

[43] Worldwide Web Consortium. Extensible Markup Language (*XML*) 1.0. http://www.w3.org/TR/REC-xml/, 2008.

Chapter 3
Parsing *XML* Content

Abstract In this chapter, we explore approaches to parsing *XML* content within *R* and extracting content from the various types of elements in the *XML* document. The primary approach is to parse an *XML* document into a hierarchical tree object. We show how the tree representation of an *XML* document (described in Chapter 2) can be treated as a list in *R*, which makes it easy to navigate nodes and branches in the *XML* document. In addition, we demonstrate how to use functions in the XML package that are designed to work with different elements of the tree, e.g., functions for accessing node names, text content, attribute values, namespaces, etc. Subsequent chapters introduce *XPath* (Chapter 4), a powerful *XML* technology for locating content in an *XML* document, and describe more complex strategies for extracting *XML* content (Chapter 5).

3.1 Introduction to Reading *XML* in *R*

In Chapter 1, we introduced some high-level functions that can read parts of an *HTML* or *XML* document directly into data frames and lists in *R*, e.g., readHTMLTable() and xmlToDataFrame(). These functions work well for many standard *HTML* and *XML* documents because we have specific understanding of the meaning of the *XML* elements, e.g., `<table>`, `<row>`, and `<td>`. However, while these high-level functions work well in specific contexts, they do not when we work with more general *XML* or *HTML* documents. More typically, we work with a less ubiquitous dialect of *XML* than *HTML* and it is up to us to extract a specific subset of the information from the *XML* tree, such as the values of particular attributes or the content of certain elements. When this is the case, it is simplest to explicitly parse the *XML* document in a general fashion and then use other tools in *R* to operate on the content of the parsed document. These tools are what we used to build the high-level functions such as readHTMLTable() and xmlToDataFrame().

In Chapter 1, we demonstrated how relatively easy it is to read *XML*-formatted Kiva data into an *R* list or data frame. However, suppose we only want the occupations of the lenders? That is, we want to extract just the `<occupation>` node from each `<lender>` without having to populate an *R* list/data frame with all of the additional information in the Kiva document. We demonstrate how to do this in the next example, which we use to briefly introduce the basic functions in the XML package [4] for extracting content from *XML* documents. Later sections of this chapter provide additional examples and a more comprehensive introduction to these functions.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_3,

Example 3-1 Subsetting a Kiva Document to Extract Lender's Occupation

Figure 1.2 provides a display of the Kiva document's hierarchical structure. This figure shows us that each lender's occupation appears in an <occupation> node within the lender's <lender> node. To access these data, we begin by parsing the Kiva document as we did in Example 1-2 (page 11) with

```
doc = xmlParse("kiva_lender.xml")
```

Rather than convert the entire parsed document into an *R* list via xmlToList(), as we did in our earlier example, we want only to retrieve the contents of the <occupation> nodes.

We first access the top-level/root node in the document and then its child node, <lenders> with

```
lendersNode = xmlRoot(doc)[["lenders"]]
```

We now have access to the <lenders> node and want to iterate over all of its children (the <lender> nodes) and extract the value of each of their <occupation> nodes. The typical *R* approach is to get a list of the child nodes and pass this to sapply(). The function xmlChildren() is the means for getting the list of all child nodes, i.e., the individual <lender> nodes, which we loop over with

```
sapply(xmlChildren(lendersNode), function(node) ....)
```

All that remains is to write the function we want to apply to each individual <lender> node.

This function needs to get the content of the <occupation> node, i.e., the text between that child node's start and end tags. For this, we can use

```
xmlValue(node[["occupation"]])
```

because the `node` variable will be a <lender> node and it has an <occupation> node as one of its children. Here, we are using subsetting by name to access the <occupation> node in <lender>, and the function xmlValue() extracts the text content from this node. Putting this all together we have

```
occ = sapply(xmlChildren(lendersNode),
             function(node) xmlValue(node[["occupation"]]))
```

We now have the information we want.

In this example, we used the xmlRoot(), xmlChildren(), and xmlValue() functions to access nodes and their content. We also used the `[[` subsetting operator to access child nodes by name. These and other functions for working with an *XML* hierarchy are the subject of this chapter.

3.2 The Document Object Model (*DOM*)

The XML package provides several approaches for extracting data from *XML* documents. In this chapter, we focus on one approach called Document Object Model (*DOM*) parsing [1, 2]. This is perhaps the most natural approach for many users as the document is represented as a tree (see Section 2.4 in Chapter 2). We demonstrate how to extract information from a document by using the tree hierarchy to access the content of interest. We introduce this type of parsing first because it is the simplest and the cornerstone for several other parsing models. Other models are described in later chapters. These include: *XPath*, a powerful language to navigate through nodes and attributes in the *DOM*; event-driven parsing with *SAX*, Simple API for *XML*; and hybrid parsing models that use handler functions for customized parsing of specific nodes. The functions introduced in this chapter

for working with the *DOM* are still necessary for the *XPath* and hybrid approaches to parsing *XML* documents.

Recall from Chapter 2 that with the tree, each node corresponds to an *XML* element and the hierarchy represents the relationships between elements. Our *DOM* parser returns a copy of the *XML* document in this hierarchical form, which makes it reasonably easy to operate on elements in the document to convert their contents or extract relevant information. The *DOM* also enables us to use *XPath* to locate nodes and attributes, and we will see in Chapter 5 how the functions introduced in this chapter, along with *XPath* queries, are used in various approaches to parsing *XML* documents.

Let's get started with parsing the *XML* document into a tree-structure and demonstrating the correspondence between the various *XML* elements and their representation in *R*. The xmlParse() function in the XML package is a *DOM* parser, meaning it reads a document into a structure that represents the hierarchical structure of the document. While xmlParse() has 19 arguments, the only one that is required is *file*, which we can use for specifying the location of the *XML* document, be it a local file, *URL*, or a string containing the document itself. Simple calls of the form

```
doc = xmlParse("FargoDailyWeather.xml")
```

will yield a tree object in *R* for accessing the contents of the *XML* document. When we show/print such objects in *R*, they appear in the *R* console as the existing *XML* document, suitably indented, etc.

To demonstrate how to parse an *XML* document, we provide a simple example from the US Geological Survey (USGS) catalog of earthquakes that occurred in a one-week period [7]. The USGS provides real-time, worldwide earthquake data in a variety of formats, including two *XML* vocabularies, which are available at http://earthquake.usgs.gov/eqcenter/catalogs/. Below is a snippet of our *XML* file that shows the structure of the information for one event/quake.

```
<?xml version="1.0" encoding="UTF-8"?>
<merge>
 <event id="00068404" network-code="ak"
       time-stamp="2008/09/16_22:17:31 " version="2">
  <param name="year" value="2008"/>
  <param name="month" value="09"/>
  <param name="day" value="14"/>
  <param name="hour" value="00"/>
  <param name="minute" value="59"/>
  <param name="second" value="04.0"/>
  <param name="latitude" value="51.8106"/>
  <param name="longitude" value="-175.9250"/>
  <param name="depth" value="146.0"/>
  <param name="magnitude" value="3.8"/>
  <param name="num-stations" value="10"/>
  <param name="num-phases" value="15"/>
  <param name="dist-first-station" value="126.1"/>
  <param name="azimuthal-gap" value="53"/>
  <param name="magnitude-type" value="L"/>
  <param name="magnitude-type-ext"
         value="Ml = local magnitude (synthetic Wood-Anderson)"/>
  <param name="location-method" value="a"/>
  <param name="location-method-ext"
         value="Auryn (Confirmed by human review)"/>
 </event> ...
```

Notice that the root node of the document is <merge>, and information for each earthquake/episode is contained in an <event> child of <merge>. The details for an earthquake are in <param> nodes in the earthquake's <event> element. Each <param> has a *name* and *value* attribute, creating a structure that is parallel to the name–value pair type of format used in many plain text files.

We can read this *XML* document into *R* with

```
doc = xmlParse("merged_catalog.xml.gz")
```

The xmlParse() function uses the *XML* parsing library `libxml2` [8] to build an *XML* tree as a native/*C*-level data structure. This data structure uses *C* pointers (similar to references) to connect nodes as children, parents, and siblings. As such, it is easy to navigate from a node to its children, or to its parent or ancestor, even to the top-level node in the document. It is harder to do this with the *R* data structures. For example, traversal up the tree is not possible with the list of lists returned from xmlToList(). By leaving the document in a *C* data structure, we are able to use *XPath* to traverse the tree in any direction.

The xmlParse() function returns an object of class, `XMLInternalDocument`; it has a field for the name of the file containing the *XML*, which we access with docName(), and a pointer to the root node of the document, which we access with xmlRoot(), e.g.,

```
root = xmlRoot(doc)
```

The xmlRoot() function returns an object of class `XMLInternalElementNode`. This is a regular *XML* node and not specific to the root node, i.e., all *XML* nodes will appear in *R* with this class or a more specific class. An object of class `XMLInternalElementNode` has four fields: name, attributes, children and value, which we access with the methods xmlName(), xmlAttrs(), xmlChildren(), and xmlValue(), respectively. For example, we can confirm that the root element is named "merge" with

```
xmlName(root)
```

```
[1] "merge"
```

Additionally, we can calculate the number of children of <merge> by passing `root` to xmlSize():

```
xmlSize(root)
```

```
[1] 1047
```

This shows that there are 1047 child nodes, which matches the length of the list returned from

```
length(xmlChildren(root))
```

The length() function is not overloaded to give xmlSize(); that is, `length(root)` returns 1.

In the next section, we demonstrate how to use these and other methods to navigate the tree and access specific content.

3.3 Accessing Nodes in the *DOM*

The XML package provides many generic functions that can extract element names, attribute values, child nodes, and text content from nodes in the tree. In addition, the `[[` and `[` operators allows us

to treat an *XML* node as a list of its child nodes. Similar to subsetting lists, we can use `[` and `[[` to access child nodes by positions, names, a logical vector, or exclusion. When we subset by "name", we use the node's element name. For example, we can extract the first <*event*> node from <*merge*> in our USGS document with

```
event1 = root[["event"]]
event1
```

```
<event id="00068404" network-code="ak"
       time-stamp="2008/09/16_22:17:31 " version="2">
  <param name="year" value="2008"/>
  <param name="month" value="09"/>
  <param name="day" value="14"/>
...
</event>
```

We can retrieve, say, the tenth child of the first <*event*> with

```
event1[[10]]
```

```
<param name="magnitude" value="3.8"/>
```

We see that this child holds the magnitude of the quake. As another example, we can get the first seven <*event*> nodes with

```
root[1:7]
```

Similarly, we can get all but the first seven children of the root node with

```
root[ -(1:7) ]
```

Finally, we note that when we subset by name using the single square bracket, e.g.,

```
evs = root["event"]
```

then we extract all <*event*> nodes from the <*merge*> node. A call to length() confirms this:

```
length(evs)
```

```
[1] 1047
```

This is different from regular *R* lists but more convenient than

```
root[ names(root) == "event" ]
```

The `evs` object is of class `XMLInternalNodeList`, which is essentially a list of `XMLInternalElementNode` objects. This means that we can apply the methods xmlName(), xmlValue(), etc. to the elements in `evs`, e.g., we find that the first <*event*> node has 18 children with

```
xmlSize(evs[[1]])
```

```
[1] 18
```

We consider the node as a named list where the names are the node names of the child elements of that node. To see the names of the children of `event1`, i.e., the first <*event*> node, we use

```
names(event1)
```

```
  param   param   param   param   param   param   ...
"param" "param" "param" "param" "param" "param"
```

We obtain the name of the node itself with xmlName(), i.e.,

```
xmlName(event1)
```

```
[1] "event"
```

In the following example, we return to the Kiva data of Example 3-1 (page 54) and demonstrate how to use `[` to extract the <*template_id*> from the images in the <*lender*> nodes.

Example 3-2 Retrieving Content from Subnodes in a Kiva Document

In Example 3-1 (page 54), we created `lendersNode` as a "list" of 1000 lenders with

```
doc = xmlParse("kiva_lender.xml")
lendersNode = xmlRoot(doc)[["lenders"]]
```

We can confirm that there are 1000 children in <*lenders*> with

```
xmlSize(lendersNode)
```

```
[1] 1000
```

Before getting the <*template_id*> from the <*image*> node in each <*lender*> node, we demonstrate some additional approaches to subsetting. Within a <*lender*> node, we can retrieve by name the <*name*>, <*occupation*>, and <*image*> nodes of the first lender with

```
lendersNode[[1]] [ c("name", "occupation", "image") ]
```

```
$name
<name>Matt</name>

$occupation
<occupation>Entrepreneur</occupation>

$image
<image>
  <id>12829</id>
  <template_id>1</template_id>
</image>

attr(,"class")
[1] "XMLInternalNodeList" "XMLNodeList"
```

Again, `lendersNode[[1]]` gives the first <*lender*> node and

```
lendersNode[[1]] [ c("name", "occupation", "image") ]
```

returns a list of the three nodes named name, occupation, and image. If there are, e.g., multiple <*name*> nodes in that <*lender*> element, then all of them are returned.

Suppose we want the children of `lendersNode[[1]]` that have more than one child. In this example, that is only the <*image*> node. We can determine which child nodes satisfy this constraint with

```
w = sapply(xmlChildren(lendersNode[[1]]), xmlSize) > 1
```

We can use this logical vector to get the subset

```
lendersNode[[1]] [ w ]

$image
<image>
  <id>12829</id>
  <template_id>1</template_id>
</image>

attr(,"class")
[1] "XMLInternalNodeList" "XMLNodeList"
```

Finally, we retrieve the content of the <*template_id*> subnodes within <*image*> with

```
template_id = sapply(xmlChildren(lendersNode),
                     function(x)
                        xmlValue(x[["image"]][["template_id"]]))
```

Looping over *all* child nodes of an *XML* node is very common and so we have provided a slightly simpler idiom than

```
sapply(xmlChildren(parentNode), function(node) ...)
```

The functions xmlApply() and xmlSApply() take the node and the function and do the looping for us. (The "sapply" version attempts to simplify the result; xmlApply() does not.) We can, for example, obtain the number of children in each of the child nodes of the first <*lender*> node [from Example 3-2 (page 58)] with

```
xmlSApply(lendersNode[[1]], xmlSize)
```

That is, xmlSApply() loops over the children of the node provided and applies the supplied function to each child node, i.e.,

```
xmlSApply(parentNode, function(childNode) ...)
```

This is marginally simpler. If you want to iterate over a subset of the child nodes then you have to use xmlChildren() and subsetting, and then use sapply() or lapply() to iterate over the list of nodes.

In the following example, we will demonstrate how to use xmlSApply() to extract the time-stamp and magnitude for each earthquake in the USGS document introduced in Section 3.2.

Example 3-3 Retrieving Attribute Values from a USGS Earthquake Document

In Section 3.2, we saw that the data for each earthquake appears in an <event> node. The time of the quake is available in the *time-stamp* attribute on the <event> element. The magnitude appears in a <param> child of <event>. The <event> has several <param> children, and these are differentiated by their *name* attributes. That is, the magnitude of the quake is found in the <param> node that has a value of "magnitude" for its *name* attribute. The actual measurement for the magnitude is in the *value* attribute for this <param> node.

We first retrieve the time-stamp for each <event> node. We can use the xmlGetAttr() function to do this. This function takes as input a node and the name of an attribute, and it returns the value of the attribute on that node. We use xmlSApply() to apply xmlGetAttr() to each child of the root node, i.e., to each <event> node, with

```
timests = xmlSApply(root, xmlGetAttr, "time-stamp")
timests[1047]

                  event
"2008/09/21_00:46:57 "
```

This apply function iterates over the input node's children, invoking xmlGetAttr() and returning the vector of time-stamps. We can pass additional arguments for each function call via the ... argument of xmlSApply() as we can for lapply() and sapply(). Here we passed "time-stamp" for the value of the *name* argument to xmlGetAttr().

Now that we have obtained the time-stamps, we turn to the task of getting the magnitudes of the quakes. We provide a few alternative approaches that increase in sophistication and in generality. To start, we notice that the magnitude of the first <event> appears in its tenth child :

```
root[[1]][[10]]

<param name="magnitude" value="3.8"/>
```

For our first approach, we extract the tenth element of each <event>. Then from each of these <param> nodes, we extract the content of its *value* attribute:

```
child10 = xmlSApply(root, "[[",  10)
mags = as.numeric(sapply(child10, xmlGetAttr, "value"))
head(mags)

event event event event event event
  3.8   1.9   1.1   1.2   0.6   1.3
```

The xmlGetAttr() function accepts a *converter* parameter, which is useful for coercing individual values. However, in this case, it is more efficient to get the attribute values of all the nodes and then convert this character vector to numeric.

An alternative approach is to combine the extraction of the children and the application of the xmlGetAttr() function in one step with

```
mags = as.numeric(xmlSApply(root,
                            function(node)
                                xmlGetAttr(node[[10]], "value")))
```

Of course, we have assumed in both of these approaches that the tenth child of each <event> node will always hold the magnitude. When we check the number of children in the first few events,

```
xmlSApply(root, xmlSize)[1:4]
```

```
event event event event
   18    18    18    22
```

we see that these nodes have different numbers of children. Relying on the tenth child of each <*event*> to contain the magnitude may be problematic. A more general approach that does not depend on a particular ordering of the <*param*> nodes is preferable.

A more robust approach is to get the value of the *name* attribute on each of the <*param*> nodes in an <*event*> node and determine which is the pertinent <*param*>. Then, we know that we are getting the magnitude value, rather than whatever might be in the tenth <*param*>. We can accomplish this using nested calls to xmlSApply() with

```
xmlSApply(root, function(evNode) {
                  parNames = xmlSApply(evNode, xmlGetAttr, "name")
                  i = which(parNames == 'magnitude')
                  xmlGetAttr(evNode[[i]], "value")
                })
```

In Chapter 4, we introduce *XPath* which we can use to locate the magnitudes more simply. Essentially, with an *XPath* expression, we can specify that we want the information in the *value* attribute of those <*param*> tags that have a *name* of "magnitude". For completeness and to demonstrate the potential of *XPath*, we provide the code for this approach. We use the getNodeSet() function, which takes a node and an *XPath* expression and uses this expression to navigate the DOM. We do this with

```
getNodeSet(root, "/merge/event/
                    param[@name='magnitude']/@value")
```

The return value is a list of the desired attribute values, which we can unlist and convert to numeric.

This more elegant approach demonstrates the power of *XPath* expressions to locate nodes, attributes, and content. *XPath* provides a convenient fast way to find nodes with particular characteristics anywhere in the tree. Often all that is needed is to identify a standard characteristic, such as all nodes with a particular name or all nodes with a particular attribute value. Such types of criteria can be combined into rich expressions that identify subsets of the nodes. These expressions are available in the *XPath* language and are discussed in greater detail in Chapter 4. Often, we use *XPath* to obtain the nodes and then operate on them using the *R* functions we have explored in this chapter.

In the previous example, we saw how to access nodes in a *DOM* object with functions for going down the tree hierarchy, i.e., [, [[and xmlChildren(). In addition, we also have functions for traversing up the tree and across siblings. The getSibling() function retrieves a sibling of a node, either the sibling to the left (i.e., above) or to the right (below) of the node. Whereas, xmlParent() gives us access to the parent of a node. For example, suppose `firstMag` holds the <*param*> element with the magnitude information for the first quake in the catalog, i.e.,

```
firstMag = root[[1]][[10]]
```

Then, we can access that <*param*> node's sibling to the left with

```
getSibling(firstMag, after = FALSE)
```

```
<param name="depth" value="146.0"/>
```

The default value for the *after* parameter in getSibling() is TRUE, which returns the sibling node that follows the provided node. The xmlParent() function gives us access to the <*event*> node in which this <*param*> is located. For example, we retrieve the attributes on this parent with

```
xmlAttrs(xmlParent(firstMag))
```

```
        id       network-code             time-stamp     version
"00068404"               "ak" "2008/09/16_22:17:31"          "2"
```

Notice that the XMLInternalElementNode in firstMag has not lost access to the parent even though we assigned this node to a new variable. This is because the assignment has not made a copy of the node. As mentioned earlier, the parsed document consists of pointers to a *C*-level data structure. When we assigned evs[[1]][[10]] to firstMag, we did not get a copy of the <*param*> node. Instead, we have a reference to that point in the tree. Any changes made to firstMag will be reflected in the parsed document.

The xmlChildren(), xmlParent(), getSibling() and xmlAncestors() functions use these pointers to navigate to any node in the tree, e.g., we can traverse from firstMag up the tree to the root with

```
xmlName(xmlAncestors(firstMag)[[1]])
```

```
[1] "merge"
```

The xmlAncestors() function walks the chain of parents to the top of the document and returns a list of those ancestor nodes. The first will be the root node. Also, from a parent or sibling of a parent we can navigate down the tree, e.g.,

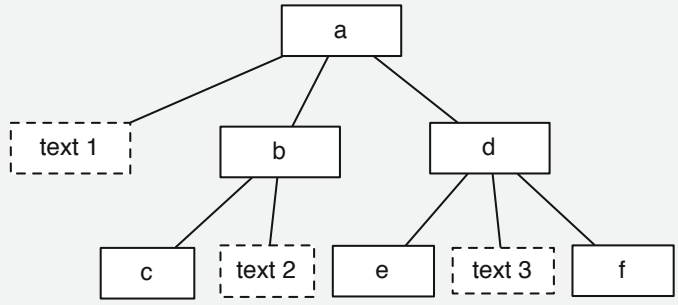
```
getSibling(xmlParent(firstMag))[[10]]
```

```
<param name="magnitude" value="1.9"/>
```

We have retrieved a "cousin" of firstMag. With the XMLInternalDocument object, it is easy to traverse from a node to its children, its parent, the top-level document containing the node, any ancestor, sibling, and so on.

The *DOM* Parser in *R*

The xmlParse() function in *R* by default returns a tree structure which can be treated conceptually as a list of lists, i.e., each node in the tree can be treated as a list of its children. Each node also has a name and attributes. The following functions in the XML package are available for accessing nodes and their content. We exemplify how to use them with the simple document shown below, where dashed boxes correspond to text content.



- Read the *XML* document into *R* with

```
doc = xmlParse("simpleEx.xml")
```

- Access the root node via

  ```
  root = xmlRoot(doc)
  ```

- Operate on a node as if it is a list of its children, i.e., use `[` and `[[` to access elements in the tree. For example, we access the <*e*> node in the document with either of the following:

  ```
  node3_1 = root[[3]][[1]]
  node3_1 = root[["d"]][["e"]]
  ```

- The XML package provides functions for determining information about a node. These include xmlName(), xmlSize(), xmlAttrs(), xmlGetAttr(), xmlValue(), xmlNamespace(), and getDefaultNamespace(), which provide, in order, the node's name, number of children, attributes, a specific attribute, text content of the node and its descendants, namespace, and default namespace. For example,

  ```
  xmlValue(root[["b"]])

  "text 2"
  ```

  and `xmlSize(node3_1)` returns 0.
- In addition to `[` and `[[`, other functions in XML enable us to work with a node's siblings, children, parent, and ancestors. These are getSibling(), xmlChildren(), xmlParent(), and xmlAncestors(), respectively. For example, we retrieve the parent of `node3_1` with

  ```
  xmlParent(node3_1)
  ```

  and the sibling following `node3_1` with

  ```
  getSibling(node3_1)
  ```

  With these functions, we can traverse the tree from one node to another anywhere in the tree. For example, from `node3_1` we access the first child of <*a*> with either of these expressions:

  ```
  xmlParent(xmlParent(node3_1))[[1]]
  getSibling(getSibling(xmlParent(node3_1), after = FALSE),
             after = FALSE)
  ```

- The tree object behaves differently from regular *R* objects. When we make the assignment, `node3_1 = root[[3]][[1]]`, we now have a reference to that point in the tree. Any operations on `node3_1` will be made to the tree as well. For example, `xmlParent(xmlParent(node3_1))` references the root of the parsed document, i.e., `root`.

3.4 Parsing Other *XML* Element Types

The xmlParse() function parses the entire document, including comments, text content, processing instructions, etc. These nodes are not all "regular" *XML* nodes. Indeed a text node does not have a name or any attributes. The various types of nodes in the document each have their own class in *R*. A comment has class `XMLInternalCommentNode`, a processing instruction has class `XMLInternalPINode`, text content has class `XMLInternalTextNode`, and <*CDATA*>

has class `XMLInternalCDATANode` (by default). These are all extensions of the base class `XMLInternalNode`. (See Section 2.5 for a description of the various elements of an *XML* document.) As xmlParse() encounters a node, it knows what type it is and maps it to the corresponding *R* object. To illustrate, let us work with the following simple *DocBook* document, which we have annotated to highlight the different kinds of elements it contains:

```
<?xml version="1.0" encoding="UTF-8"?>  1
<article xmlns="http://docbook.org/ns/docbook"  2
            xmlns:r="http://www.r-project.org">

<title>A Title</title>
<-- A comment -->  3

<para>
This paragraph includes text and a comment  4
<-- a comment in a paragraph -->
and a processing instruction <?R sum( 1, 3, 5) ?>  5
The paragraph includes code in a CDATA node
<r:code><![CDATA[  6
 x <- (y > 1 & z < 0)
]]></r:code>  7
</para>
</article>
```

1 The *XML* Declaration along with the encoding information.
2 The topmost root node with two namespace definitions.
3 A comment node.
4 Text content.
5 A processing instruction.
6 Escaped character data (*<CDATA>*).
7 There is text content after the *<r:code>* element which consists of a space and a new-line character.

This document contains comments, processing instructions, *<CDATA>*, and elements from the *RDocBook* extension of *DocBook* (see Example 2-1 (page 30)). The extension elements begin with the namespace prefix `r`. Those without a prefix are element names from the default namespace, *DocBook*. The two namespaces are both declared on the root node, *<article>*.

We read this document into *R* with a call to xmlParse() and access the root node with xmlRoot() as shown here :

```
rdbRoot = xmlRoot(xmlParse("simpleDoc.xml"))
```

We can look at the individual nodes in the usual manner using `[[` and, e.g., confirm that the second child is a comment:

```
rdbRoot[[2]]

<!-- A comment -->
```

Additionally, we can access the text content within the comment, with the xmlValue() function. That is, we can extract the information between the comment delimiters as follows:

```
xmlValue(rdbRoot[[2]])
```

```
[1] " A comment "
```

The xmlValue() function is generic so it works on different types of nodes. For nodes that are mixtures of text content and other nodes, xmlValue() returns a character string that concatenates the text content of all the node's descendants.

Next, we explore the contents of the <*para*> node in `rdbRoot`,

```
names(rdbRoot[["para"]])
```

```
   text   comment    text    R    text    code    text
 "text" "comment" "text" "R" "text" "code" "text"
```

These seven elements consist of text prior to the comment, the comment, text between the comment and processing instruction, the processing instruction, text immediately following the processing instruction, code, and the final text at the close of the paragraph. We check their classes with

```
sapply(xmlChildren(rdbRoot[["para"]]), class)
```

```
                    text                  comment
  "XMLInternalTextNode" "XMLInternalCommentNode"
                    text                        R
  "XMLInternalTextNode"      "XMLInternalPINode"
                    text                     code
  "XMLInternalTextNode" "XMLInternalElementNode"
                    text
  "XMLInternalTextNode"
```

The seventh node may seem unexpected because it does not appear that there is any text in the <*para*> node that follows the <*code*> node. With xmlValue(), we see that there is indeed a blank space followed by a new-line character:

```
xmlSApply(rdbRoot[["para"]], xmlValue)
```

```
                                                text
       "\nThis is text including < and a comment\n"
                                             comment
                    " a comment in a paragraph "
                                                text
                                   "\nand a PI "
                                                   R
                                 "sum( 1, 3, 5) "
                                                text
"\nThis paragraph includes code in a CDATA node \n"
                                                code
                        "\nx <- (y > 1 & z < 0) \n"
                                                text
                                              " \n"
```

We also have access to the namespace on a node with xmlNamespace(). For example, we can find the namespace on the <*r:code*> node in a *DocBook* document with, e.g.,

```
xmlNamespace(rdbRoot[[3]][[6]])

                          r
"http://www.r-project.org"
attr(,"class")
[1] "XMLNamespace"
```

This returns the URI associated with the `r` prefix on the node, which we see is `http://www.r-project.org`. Additionally, the getDefaultNamespace() function retrieves the default namespace declared on the top-level node in the document. We can pass it any node, e.g.,

```
getDefaultNamespace(rdbRoot[[3]][[6]])

"http://docbook.org/ns/docbook"
```

and the function retrieves the default namespace for the root node.

3.5 Parsing *HTML* Documents

We often want to scrape data from *HTML* documents. For example, we want to obtain the values in a table, or find all the links, or just the text in a particular paragraph. We have seen that we can use readHTMLTable() and other functions to do this. *HTML* is a particular *XML* vocabulary so these functions use many of the tree and node manipulation functions we have described up to this point in this chapter. Unfortunately, *HTML* is not a strict *XML* vocabulary. Instead, *HTML* documents are often not well-formed. Not all *HTML* elements will have a corresponding closing element, and attributes are often not within quotes. As a result, if we use the xmlParse() function to read an *HTML* document, we typically get an error and the document is not parsed. To parse an *HTML* document, we need a more forgiving parser such as those used in Web browsers which try to, for example, guess where a node should end if there is no closing tag. The `libxml2` *C* code provides a forgiving parser, and we can use this in *R* via the htmlParse() function. This is basically the same as the xmlParse() function, but works on possibly non-well-formed *HTML* documents. The htmlParse() function returns an object of class `HTMLInternalDocument` which extends the `XMLInternalDocument` class. The nodes in the tree are regular `XMLInternalNode` objects and we can use all of the "xml" functions we have discussed so far in the same way as for *XML* documents and nodes.

As an alternative to using a separate function to parse *HTML* documents, we can call xmlParse(), but use the *isHTML* parameter to indicate the content is *HTML*. That is, htmlParse() is the same as

```
xmlParse(source, isHTML = TRUE)
```

The *C* code underlying the htmlParse() function uses certain heuristics to parse the *HTML* document as a tree. It takes *HTML* content such as

```
<ol>
<li>First
<li>Second
<li>Third
</ol>
```

and correctly infers that there should be a closing `</li>` element before the next `<li>` starts and before the closing `</ol>` tag. The parser will also add missing elements such as `<body>` and even the root `<html>` node so parsing this content yields

```
htmlParse("<ol><li>First<li>Second<li>Third</ol>")
```

```
<!DOCTYPE html PUBLIC
   "-//W3C//DTD HTML 4.0 Transitional//EN"
   "http://www.w3.org/TR/REC-html40/loose.dtd">
<html><body><ol>
<li>First</li>
<li>Second</li>
<li>Third</li>
</ol></body></html>
```

On occasion, the heuristics of the *HTML* parser will result in a tree that is not appropriate. There are several things we can do in these situations. One of them is to modify the resulting tree, moving the different nodes and subtrees appropriately. We will explore how to do this in Chapter 6 using functions similar to what we have introduced here, but for modifying nodes.

Another approach is to try to use a different parser with different heuristics. Rather than using a different *XML*/*HTML* parser in *R*, we can try to filter the document to make the document well-formed. We can then pass this document with the correct closing tags and quoted attributes to htmlParse(), or even xmlParse() since it will be well-formed. There are Web sites and services that can convert an *HTML* document to *XHTML*, which is a strict form of *HTML* that requires the documents to be well-formed. There are also programs that we can install to do this locally. The *R* package RTidyHTML [3] and its function tidyHTML() is one approach that we can use directly in *R*. It uses a *C* library named `libtidy` to perform the transformation. We call the function with the name of the local file or the actual *HTML* content as a string, and it returns the resulting "corrected" *HTML* document as a single string, e.g.,

```
content = tidyHTML(filename)
```

We can then pass this to htmlParse()

```
doc = htmlParse(content, asText = TRUE)
```

3.6 Reading *XML* from Different Input Sources

The parser available through the XML package supports reading local files, in-memory *XML* strings, URLs, and "local" compressed files (i.e., via either local or networked file systems). As we have seen in many cases already, when the *XML* content that we want to parse is in a local file, we just pass the name of this file to xmlParse() via the *file* argument, e.g.,

```
xmlParse("FargoDailyWeather.xml")
```

As *XML* can be verbose, it can consume a lot of disk space when stored or bandwidth when sent across a network. However, there are a lot of repeated patterns in *XML*, and this tends to make *XML* content highly responsive to compression. Typically ratios of compressed form to original range between 10% and 30%. The `libxml2` parser, which we use in the XML package, can directly and transparently read the *XML* content from compressed files without us needing to explicitly uncompress the file first. For this reason, xmlParse() can parse a gzipped (**gz**) file as simply as

```
xmlParse("merged_catalog.xml.gz")
```

While this is slightly slower than reading the uncompressed file (since it has to uncompress the content as it sees it as well as do the usual parsing), it is convenient and allows us to avoid explicitly uncompressing the file and recompressing it.

Since *XML* is used to exchange data with different, potentially remote, applications, it is often conveniently found on Web sites. Rather than having the user explicitly download the file, we allow the user to pass the *URL* to the parser and have it parse the content as it reads the file directly from the server, e.g.,

```
xmlParse("http://www.omegahat.org/RSXML/index.html")
```

The `libxml2` parser does this for *FTP* and *HTTP* URLs. It transparently recognizes URLs and talks to the server. Instead of a two-step approach of downloading the file and then parsing it, the *XML* parser streams the data incrementally from the server and processes it on the fly.

Unfortunately, the *URL* option in the parser performs only basic requests. For example, it does not handle *HTTPS*. To process documents that require a more sophisticated Web request, one can either first download the document locally using a Web browser or a command-line tool and then pass it to the *XML* parser, or use a more flexible Web request facility such as RCurl [6] (described in Chapter 8). Indeed, with the functionality provided in RCurl, it is possible to parse from a streaming data source using xmlEventParse() (see Chapter 5). We can also use RCurl to retrieve a compressed *XML* document, uncompress or extract the content in memory (with functions in base *R* or the Rcompression package [5]) and then parse the content as an *XML* string.

In other contexts, we will have the *XML* document as a string. This can occur when we download it from a Web site. Or, we may read a file and extract only a portion of it to get the *XML* content of interest. Similarly, we may create the *XML* content as a string. Rather than write the *XML* string to a temporary file and pass that to the *XML* parser, we can pass the string directly to the parser and have it work on that, e.g.,

```
xmlParse(myXMLString, asText = TRUE)
```

The *asText* parameter indicates that the first argument should be treated as the *XML* content to parse and is not the name of a file. Alternatively, we can identify the string as *XML* content (and not the name of a file or URL) by using the I() function to make the string of class `AsIs`, e.g., `I(myXMLString)`.

3.7 Character Encoding

Each *XML* document should start with the standard *XML* declaration. This should also include the character encoding for the document. The declaration would look something like

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

The *encoding* component in this declaration tells the parser what encoding to use for the characters. It is important that *XML* documents be self-describing in this way. Unfortunately, not all *XML* documents have this important information. In these cases, the parser assumes the content is Unicode and uses UTF-8. If this is incorrect, we have a problem. In these situations where we happen to know the actual character encoding of the document, but it is either not in the *XML* declaration or is wrong, we can specify the correct encoding in the call to xmlParse(). We do this with, e.g.,

```
xmlParse(docSource, encoding = "ISO-8859-1")
```

Generally, we should not need to do this.

We can also query a parsed document to find the encoding that was used by the parser. We use the getEncoding() function to access this information.

3.8 Greater Control over the Parser

The functions xmlParse() and htmlParse() have parameters that allow us to control certain aspects of how the parser behaves. For example, we can specify whether entities are to be replaced by their text form or left as entities. Similarly, we can control whether *XInclude* directives are processed or left as-is in the tree. We can also provide our own error handler function, which the parser calls when it encounters malformed *XML* content. In addition, the `libxml2` parser has several other options that control aspects of how it processes documents. We can specify these in calls to xmlParse() and htmlParse() via the *options* parameter.

These additional options control aspects of the parser such as whether to access URLs via the network or not, or whether errors or warnings should be announced or suppressed and whether to attempt to recover from errors in the *XML* content and keep parsing. The options also indicate how the parser represents certain types of nodes in the *DOM*. For example, we can have content enclosed within <*CDATA*> markup explicitly represented by <*CDATA*> nodes in the *DOM* or left as simple text nodes. Similarly, we can control whether to enclose *XInclude*'d content within an invisible node where the type identifies it as being the result of an *XInclude* directive (see Section 4.9). We can also have the parser discard empty text nodes which are typically just used for indenting and formatting *XML* content. We can also have the parser remove redundant namespace definitions while it is parsing and so simplify the tree. Additional options control how the parser handles older *XML* content, while others allow us to fine-tune the internal workings of the parser. For example, the `HUGE` option disables any hard-coded limits the parser might impose as safety checks. The names and descriptions of the full set of supported options are given in Table 3.1.

Each of these parser settings that we can specify via the *options* parameter are simple binary options, i.e., either on or off. The parser has a default value for each. If we want to override this default setting in a call to xmlParse() or htmlParse(), we add the relevant option variable in *R* (e.g., `NONET`) to the vector we pass as the value of the `options` vector. For example, suppose we want to avoid any network access of URLs (e.g., *XInclude*'d documents or *DTD*s), discard nodes made up entirely of blank space, and also avoid creating explicit <*CDATA*> nodes. We can do all of this with the call

```
xmlParse(document, options = c(NONET, NOBLANKS, NOCDATA))
```

Similarly, if we want to suppress all errors and warnings and tell the parser to keep going when it encounters an error in the *XML* content, we can use

```
xmlParse(document, options = c(RECOVER, NOERROR, NOWARNING))
```

Note that the two parameters *xinclude* and *replaceEntities* in xmlParse() and htmlParse() essentially control the same feature as `XINCLUDE` and `NOENT`. The *xinclude* and *replaceEntities* parameters take `TRUE` or `FALSE`, but they are essentially the same as toggling the default settings. These *R* parameters came before the more general *options* parameter for the two functions.

Table 3.1: Options for Controlling Aspects of the *XML* Parser

Option	Description
RECOVER	Continue even if the parser encounters an error in the *XML* content and try to make sense of the document, regardless of the error.
NOERROR	Do not issue/announce messages about errors.This can be used in conjunction with RECOVER to both ignore and continue from errors.
NOWARNING	Do not issue/announce warning messages.
PEDANTIC	Make the parser very pedantic as it encounters content that is not quite correct but not an error.
NOBLANKS	Remove text nodes that are made entirely of white space between nodes. This can be used to remove formatting content that is used for indenting nodes.
NOCDATA	Do not create nodes of type <*CDATA*> to enclose text that needs to be escaped.
COMPACT	Compact short contiguous text nodes into a single text node. This can simplify working with the parsed tree in which we may have multiple adjacent text nodes that we think about as being a single text node. This should only be used if the resulting tree is to be considered read-only, i.e., that we will not modify any of these text nodes.
XINCLUDE	Perform the *XInclude* directives to substitute the <*xi:include*> nodes with the corresponding content from the *href* and *xpointer* attributes in the *XInclude* node. Without this, the *XInclude* nodes are left as is in the resulting document. This corresponds to a value of TRUE for the parameter *xinclude* in xmlParse().
NOXINCNODE	When the parser processes an *XInclude* node and replaces it with the actual content included, it typically creates an invisible parent node that allows us to identify the material as being the result of an *XInclude* directive. This option controls whether this invisible node is added. We use this in functions such as getNodeLocation() and getNodePosition() so this can be useful.
NOBASEFIX	Do any manipulation of the value of the URIs that is set in the *xml:base* attribute of an *XInclude* node. This attribute allows us to recover information about the origin of the nodes.
NONET	Do not follow any references to content that must be accessed via the network. This can be useful to avoid network access and to verify that all the referenced files are locally available. We can use *XML* catalogs to map remote documents to local files.
NOENT	Control whether the parser replaces entities (e.g., <) with the corresponding text (e.g., <). If this is not specified, entities are left in their entity form rather than being substituted with the text.
SAX1	Use the SAX1 rather than SAX2 interface. This is a low-level detail about how the *C* routines are invoked in response to different *SAX* events.
OLDSAX	Use SAX2, but the interface that precedes version 2.7.0 of libxml2.
NSCLEAN	Eliminate redundant namespace definitions and so create a more streamlined and less verbose *XML* document.
OLD10	Use an older version of the restrictions on what constitutes an element name in *XML*.

`HUGE`	Disables any hard-coded limits on the parser. This option can be used for enormous documents.
`DTDLOAD`	Load any external *DTD* content.
`DTDATTR`	Control whether default values for attributes defined in a *DTD* are used.
`DTDVALID`	Validate the *XML* content using the *DTD* in the document.
`NODICT`	Do not allow reusing of a shared dictionary. This is a low-level option controlling how the parser is implemented.

One can combine the options described in this table to control the XML parser (via xmlParse() or htmlParse()) when it creates a tree from an XML document. More information about the options can be found in the `libxml2` documentation at `http://www.xmlsoft.org/html/libxml-parser.html`

3.9 Three Representations of the *DOM* Tree in *R*

Although we treat the parsed *XML* document and each of the elements as a list object in *R*, the return value from xmlParse() is a reference to a native *C*-level data structure. In addition, the functions such as xmlRoot() and xmlChildren() return references/pointers to the nodes in this *C*-level tree structure. The consequence of this is that when we pass an *XML* element/node as an argument in a call to an *R* function, it is not copied. Any changes made to that node by a function are made in the original node. This is an important distinction from the usual way *R* passes arguments to a function by-value (i.e., by-copy) and local changes in functions are not reflected outside of that function call. This is very similar to *R*'s reference class objects, environments, and all external pointer objects. For *XML* documents this means we can extract a node from a document, modify it, and not only has that node changed, but also the changes are reflected in the original parsed document and tree/*DOM*. This can be useful, but it is an uncommon semantic behavior in *R*. To avoid this, in the case of *XML* nodes, we can explicitly make a copy of a node and its subnodes using the function xmlClone(). We can then modify this copy of the node without changing it in the parsed document.

Working with references in *R* is problematic in other ways. Perhaps more problematically, if the code changes the *R*-level attributes (not *XML* attributes) such as the class of the node object, then that will also be reflected in all subsequent uses of the object. As a result, one has to be very careful when using this *C*-level internal node type of object in general *R* programming. However, we have found this approach to be most convenient because an object can be modified in place without having to reassign it back to its parent, for which the *R* language is not designed. Also, with this representation, we can use the powerful *XPath* query language that we will discuss in Chapter 4 to identify nodes within a tree.

Another issue with using references to external *C* data structures to represent the tree and its nodes is saving the *R* object using save() and serialize(). When we use load() to restore the resulting object, the document will simply be `NULL`. This is not a significant problem. We can write the *XML* document to a regular text file using saveXML() and bypass save() and load(). Then this document/tree can be restored directly by parsing it again. Alternatively, we can add hooks to serialize() and unserialize() to write the document or node as a string and then restore it by parsing it.

The default return value from xmlParse() is an object of (*S3*) class `XMLInternalDocument`, which consists of the aforementioned external pointers to the *C*-level data structures. The XML package also supports two other *DOM*-type representations. When we specify the *useInternalNodes* parameter in xmlParse() as `FALSE`, the function converts the parsed *XML* tree into *R* data objects so

that we can work with it entirely within *R*. The resulting tree is implemented as a hierarchy of lists with each node being a list of named elements pertaining to the node (e.g., name, namespace, and attributes) and a list of child nodes. The nodes are regular *R*-level objects and so can be easily and directly serialized and restored in a different *R* session. This object is an instance of the `XMLDocument` class, in contrast to the `XMLInternalDocument` class.

The xmlTreeParse() function is a shorthand equivalent to

```
xmlParse(url, useInternalNodes = FALSE)
```

xmlParse() and xmlTreeParse() are identical except that the default value of *useInternalNodes* has the opposite value (`TRUE` and `FALSE`, respectively). Similarly, htmlParse() and htmlTreeParse() are the same, and we can use htmlTreeParse() to return an object of class `HTMLDocument`, which is a data structure entirely within *R*. These "Tree" versions of the parsing functions are merely convenience functions or syntactic sugar.

The third representation of a *DOM* is the *S3* class, `XMLHashTree`. It provides a "flat" representation of the tree in *R*. We call it "flat" because all the nodes are kept in a single collection and auxiliary information that describes the hierarchical structure of the tree, i.e., the child and parent relationships, is stored along side these nodes, but separately from each node. Three *R* environments are used to store the nodes, children, and parent information, and each environment uses a hash table for direct lookup of nodes by their unique identifiers. The hash tree can be created by converting the `XMLInternalDocument` to an `XMLHashTree` with

```
as(xmlParse(file), "XMLHashTree")
```

We can also implicitly convert an `XMLInternalNode` to a `XMLHashNode` using the as() coercion function.

The order of the nodes in the `XMLHashTree` collection is essentially arbitrary, but the child and parent relationships allow you to navigate the nodes in the order of the tree. Because the parent relationships are available, we can determine the parent of a given node and so traverse the tree in either order. Since we use environments to store the details of the nodes and the tree, we have a mutable object in *R*. Again, this means that we do not copy the tree when we pass it to a function and changes made by one function to a tree are reflected in the original tree object. This has advantages and disadvantages in *R*.

For all three of these classes of trees (internal *C*-level elements, *R list* objects as elements, and the flat hash tree), once the *XML* document has been parsed, we can navigate through the nodes and children recursively to pull out the different pieces we want. With the `XMLDocument` and `XMLNode` classes, we cannot access the ancestors of a node. The parent contains the child nodes, which are lists so they do not know who contains them. This is because *R* does not have the notion of a reference. Although the `XMLDocument` gives us a very natural way of processing an *XML* file and then accessing the bits we need from the tree, unfortunately, this post-processing of the tree in *R* typically involves several passes over the entire tree or a general-purpose single-pass. As a result, this can be slow. As seen in Section 3.3, with `XMLInternalDocument`s we can easily traverse a tree from a node to any ancestor or descendant. In addition, much of working with *XML* involves finding sets of nodes with a particular characteristic and the *XPath* query language is very fast for this purpose. In the XML package, *XPath* queries operate only on the *C*-level data structures. As a result, *XPath* will work only on `XMLInternalDocument` and `XMLInternalNode` objects. For these reasons, we very rarely use these other two representations.

3.10 Summary of Functions for Parsing and Operating on the *XML* Hierarchy

This chapter introduced many of the functions available in the XML package for working with the *DOM* version of the *XML* document. These are summarized in the table below. Later chapters will introduce more powerful ways of working with the *DOM* (see Chapter 4) and other approaches to parsing *XML* (see Chapter 5).

xmlParse() Parse an *XML* document. This function can read a local file, a remote *URL*, *XML* content that is already in *R* as a string (when *asText* is `TRUE`), or a connection. The default return value is an internal C-level structure. Set *useInternalNodes* to `FALSE` to create the tree in *R*. If the source is *HTML* that may not be well-formed, then set *isHTML* to `TRUE`.

xmlRoot() Retrieve the root node of an *XML* document. This is usually given the parsed document, but we can also pass it any *XML* node in a document to get the root node.

xmlChildren() Get a list of all the child nodes of a given *XML* node. The list is named, where the names correspond to the element identifiers of the nodes.

xmlAttrs() Retrieve a named character vector of *all* the attributes of a given node.

xmlGetAttr() Retrieve the value of a single attribute of a given node, optionally converting it from a string to a different type (*converter*). This also allows provision of a default value (*default*) to use if the attribute is not present in the node.

xmlName() Return the name of an *XML* element/node.

xmlSize() Get the number of child nodes of an *XML* node, including text node children.

xmlValue() Return the content of an *XML* node as a string, ignoring attributes and recursively including the content in child nodes, ignoring their names, and collapsing the different text nodes at all levels together in order, but with no separator. To get the string version of a node, with node names and attributes, use saveXML(). xmlValue() is typically used to get the text from an actual text node, or from a regular *XML* node with one text child node.

names() Get the vector of element names of the child nodes, including text nodes.

`[[` and `[` Subset child nodes as if the parent *XML* node were a list.

xmlApply(), xmlSApply() Loop over the child nodes of an *XML* node and apply a function to each node, returning a list or (with xmlSApply()) simplify it if possible to a vector, matrix, or array. Each takes a node as its primary argument, and the second argument is a function that takes a node as its argument.

xmlParent(), xmlAncestors() Return the parent node/ancestors of the *XML* element provided.

getSibling() Access the sibling node to the right (default action) or left (when *after* is `FALSE`) of the provided node.

xmlNamespace() Retrieve the namespace for the node provided.

xmlNamespaceDefinitions() Retrieve the namespace definitions declared in the node or root node of an *XML* document. The *recursive* argument indicates whether or not to get the namespaces definitions for all child nodes.

getDefaultNamespace() Retrieve the default namespace for the top-level node in the document.

getChildrenStrings() Retrieve the text for the individual child nodes of an *XML* node, keeping the strings separate, unlike xmlValue(). This function is a fast version of `xmlSApply(node, xmlValue)`.

htmlParse() Parse an *HTML* document using a less restrictive version of the *XML* parser that tolerates *HTML* content that is typically not well-formed. This function is the same as xmlParse() with *isHTML* set to `TRUE`.

xmlTreeParse(), htmlTreeParse() These functions are the same as xmlParse() and htmlParse(), respectively, with the *useInternalNodes* parameter set to `TRUE`. They both return the *DOM* as an *R* data structure rather than a pointer to an internal *C*-level structure.

xmlClone() Create a duplicate of a node and its subnodes. Modifications to the cloned node will not affect the original node.

saveXML() Write the *XML* tree to a string or file, allowing control of formatting and indentation.

getEncoding() Retrieve the encoding used by the parser for the parsed document.

3.11 Further Reading

The Document Object Model is described in Chapter 19 of [1] and in Chapter 11 of [2].

References

[1] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[2] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.

[3] Duncan Temple Lang. RTidyHTML: Tidy *HTML* documents. http://www.omegahat.org/RTidyHTML, 2011. *R* package version 0.2-1.

[4] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[5] Duncan Temple Lang. Rcompression: In-memory decompression for GNU **zip** and bzip2 formats. http://www.omegahat.org/Rcompression, 2012. *R* package version 0.94-0.

[6] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.

[7] USGS Earthquakes Hazards Program. Latest earthquakes: feeds and data. http://earthquake.usgs.gov/earthquakes/catalogs/, 2010.

[8] Daniel Veillard. The *XML C* parser and toolkit of Gnome. http://www.xmlsoft.org, 2011.

Chapter 4
XPath, *XPointer*, and *XInclude*

Abstract In this chapter, we focus on *XPath*, a domain-specific language that we can use from within *R* (amongst others) to query sets of nodes in an *XML* tree by patterns within nodes. *XPath* is quite simple but very powerful. Similar to a file hierarchy, it allows us to identify nodes of interest by specifying paths through the tree, based on node names, node content, and a node's relationship to other nodes in the hierarchy. We typically use *XPath* to locate nodes in a tree and then use *R* functions to extract data from those nodes and bring the data into *R*. The combination of *R* and *XPath* gives us very powerful and flexible facilities for working with *XML*, and anyone working with *XML* on a regular basis should learn the details of *XPath*. *XPath* is the primary tool for working with *XML* content, either from scraping data from Web pages, services, or processing local *XML* documents.

4.1 Getting Started with *XPath*

When we work with *XML* documents, we typically want to extract data from them and bring these data into *R* structures such as vectors or data frames. (We will discuss creating *XML* documents within *R* in Chapter 6.) For example, we may want to extract: *R* code from the examples in this book, daily exchange rates for the yen in an *HTML* document, lender information from a Kiva response to a Web service request, or articles published by a particular author in the Journal of Statistical Software. (JSS provides its bibliographic data in *XML*.) In the previous chapter, we saw how we can manipulate a node in *R* to get its name, attributes, namespaces, text content, and children. This small collection of functions (xmlName(), xmlAttrs(), xmlNamespace(), xmlValue() and xmlChildren()) allows us to process an entire tree as we can recursively traverse the hierarchy by processing the root node and then its children and their children and so on. In theory, we have all the functionality we need to extract data from any *XML* tree using recursive functions. With the xmlParent() function, we can even go back up the tree rather than only working downwards. However, many people find recursive functions difficult to understand and write. Also, it is slightly challenging to collect the results across function calls as we descend the tree. Closures and lexical scoping in *R* can help here, but again not all *R* users are familiar or comfortable with these concepts. Fortunately, there is a technology associated with *XML* named *XPath* [9, 14] that frees us from having to recursively traverse the tree with *R* code.

Suppose we want to process an *HTML* document and extract the URLs for all of the links it contains. We are not interested in where these nodes appear in the document; we are just interested in the value of each *href* attribute of the `<a>` nodes. Consider how involved it would be to write a function to loop over all nodes in the tree and extract the value of the *href* attribute if and only if

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_4,

the node name is "a". We would have to start at the root node of the tree, check its name, and then recursively do the same thing for each of the node's children. In contrast, if we have a list of all of the <a> nodes that have an *href* attribute, we can loop over these in *R* to get the value of the *href* attribute with

```
links = sapply(listOfANodes, xmlGetAttr, "href")
```

We can use *XPath* to get this list of <a> nodes using, e.g.,

```
doc = htmlParse("http://www.omegahat.org")
listOfANodes = getNodeSet(doc, "//a[@href]")
```

The *XPath* expression `//a[@href]` is very succinct and means essentially: find all nodes named "a" throughout the tree that have an attribute (@) named "href". This expression uses some shorthand in the *XPath* language for common operations which we will explain later in this chapter, but it should be clear that *XPath* is very succinct and powerful. We do not need to worry about where the nodes are in the tree. Also, we can add some constraints on the <a> nodes. For example, we can require them to be within a <table> node, in a <table> with a *class* attribute value "data", or within an ordered list node (i.e., <ol>) which has at least three list items. This is the power of *XPath*.

Much the same as with regular expressions, *XPath* is a separate language from *R* and *XML*, and typically consists of short strings that express a query. We can form these in *R* and then use the *R* function getNodeSet() to evaluate the *XPath* query and return the matching elements of the tree to which we applied the query. Note that once we retrieved the list of <a> nodes above, we used the function xmlGetAttr(), which we saw in earlier chapters, to retrieve the *URL* for the link. If we want the text displayed for the link in the *HTML* page, we can apply the xmlValue() function to each of the <a> nodes to get the text content, or we can process the children with xmlChildren(). Generally, we use *XPath* to find nodes and then we process them in *R*. In this particular example, we can combine finding the nodes and getting the *href* attribute in either of two ways:

```
xpathSApply(doc, "//a[@href]", xmlGetAttr, "href")
```

or, entirely within *XPath* with

```
getNodeSet(doc, "//a/@href")
```

The latter shows that we can actually return attributes and not just nodes from an *XPath* query. The xpathSApply() function (and similarly xpathApply()) is a generalized version of getNodeSet() that allow us to find elements of a tree and apply a function to each of them in a single *R* command. The following example demonstrates the usefulness of *XPath* when working with a large data set.

Example 4-1 Efficient Extractions from a Michigan Molecular Interactions (MiMI) Document

The Michigan Molecular Interactions (MiMI) [8] is part of the National Institute of Health's National Center for Integrative Biomedical Informatics http://www.ncibi.org, and is available at http://mimi.ncibi.org/MimiWeb/. MiMI provides access to data from several curated protein interaction databases for people studying systems biology and gene pathways and their interactions. The data are available via a Web service, but it is also available an *XML* file. This is reasonably large, with over 25,000 top-level <molecule> nodes. The file is 6 megabytes when compressed, and 70 megabytes as raw text. We can parse the document without uncompressing it via xmlParse() and the call

```
system.time(mi1 <- xmlParse("~/XML/mi1.txt.gz"))
```

Depending on the machine and amount of memory, this takes between 2 and 4 seconds to parse the entire 70 MB. We have not converted any of the content into *R* objects, but merely have a reference

to the *C*-level tree. There are almost 3 million nodes in this tree, and so traversing the entire hierarchy with *R* functions would be extremely time-consuming.

The basic structure is a collection of <*molecule*> nodes that look something like

```
<molecule>
 <prov><im><imid>30</imid></im></prov>
 <moleculeID>116226</moleculeID>
 <moleculeType>protein
  <prov><im><imid>30</imid></im></prov>
 </moleculeType>
 <organismID>9606
  <prov><im><imid>30</imid></im></prov>
 </organismID>
 <id><prov><im><imid>30</imid></im></prov>
  <idType>HGNC</idType><idValue>9859</idValue></id>
 <name>RAP1GDS1 <prov><im><imid>30</imid></im></prov> </name>
 <name>GDS1 <prov><im><imid>30</imid></im></prov> </name>
 <name>MGC118859 <prov><im><imid>30</imid></im></prov> </name>
 <name>MGC118861 <prov><im><imid>30</imid></im></prov> </name>
 <variant>
  <prov><im><imid>30</imid></im></prov> <variantID>0</variantID>
 </variant>
 <interaction><interactionRef>93569</interactionRef>
  <moleculeRef>116280</moleculeRef>
  <moleculeName>RAC1</moleculeName>
  <selfVariantRef>0</selfVariantRef>
  <partnerVariantRef>0</partnerVariantRef>
 </interaction>
 <interaction><interactionRef>104132</interactionRef>
  <moleculeRef>103040</moleculeRef>
  <moleculeName>RHOA</moleculeName>
  <selfVariantRef>0</selfVariantRef>
  <partnerVariantRef>0</partnerVariantRef>
 </interaction>
 <interaction><interactionRef>121818</interactionRef>
  <moleculeRef>74726</moleculeRef>
  <moleculeName>MBIP</moleculeName>
  <selfVariantRef>0</selfVariantRef>
  <partnerVariantRef>0</partnerVariantRef>
 </interaction>
</molecule>
```

A task we were asked to do was to find the content of the <*moleculeName*> nodes within the <*molecule*> nodes for only those <*molecule*> nodes that have a <*name*> node containing the string `'frm-1'`. For example, the following <*name*> node satisfies the requirement on the <*name*> node's value:

```
<name>frm-1<prov><im><imid>30</imid></im></prov></name>
```

We can get the list of <*molecule*> nodes that match this criterion using *XPath* with the command

```
mol = getNodeSet(mi1, "/*/molecule[.//name/text() = 'frm-1']")
```

This searches the entire tree of 25,452 molecules and returns the two matching nodes in about four-tenths of a second.

Now that we have these nodes, we can loop over them in *R* and fetch the text in the <*moleculeName*> node within each <*interaction*> node. Without *XPath* we can do this with

```
lapply(mol, function(node)
               sapply(node[names(node) == "interaction"],
                      function(x) xmlValue(x[["moleculeName"]])))
```

This is not very complicated as the structure of these <*molecule*> nodes is quite simple. We demonstrate how we can use the *XPath* expression

```
.//interaction/moleculeName/text()
```

to find the same information in a given <*molecule*> node. This *XPath* expression translates to: starting at this current node in the tree, look at all its descendants for nodes named <*interaction*>; for each of these find its child nodes named <*moleculeName*>; and for these <*moleculeName*> nodes, extract the child nodes that are just text. We use this *XPath* expression on each node from *R* with

```
xpexpr = ".//interaction/moleculeName/text()"
lapply(mol, function(node) xpathSApply(node, xpexpr, xmlValue))
```

yielding

```
[[1]]
[1] "alecting"

[[2]]
[1] "09H1.6W"
```

in about two-thousandths of a second!

If we just want the names of the molecules in the <*interaction*> nodes matching `'frm-1'` and do not care about the molecule with which they interacted, we can make the entire query within a single *XPath* expression by combining the two steps, i.e.,

```
xpexpr = "/*/molecule[.//name/text() = 'frm-1']//
            interaction/moleculeName"
int = xpathSApply(mi1, xpexpr, xmlValue)
```

This is very fast, about half a second. If there is a lot of matching <*molecule*> nodes, this can be faster than looping over them in *R*. The key point is that we can use either or both languages to find the "best" (fastest or easiest) solution.

At this point, we have seen some examples of the power and efficiency of *XPath*. In the next section, we give an informal introduction to *XPath* and describe how to think about it heuristically. In Section 4.3, we describe the full syntax and computational model that underlies *XPath*. Then in Section 4.4 we discuss some of the *XPath* functions we can use within queries which make the language more powerful, and in Sections 4.5 and 4.6, we address how to create compound expressions. In Section 4.7, we see how to use *XPath* in the context of some short examples, and we provide a case study that demonstrates more complex expressions. Section 4.8 covers how to work with namespaces. We

aim to provide the reader with a reasonably complete understanding of using *XPath* in *R*. Additional information can be gained from books dedicated to *XPath* such as [9].

XPath is also used in other *XML* technologies. For example, *XPath* is an important part of *XInclude* and *XPointer* (Section 4.9). *XInclude* is a mechanism that allows one *XML* document to include part, or all, of another *XML* document. This is a general merge mechanism that allows us to maintain *XML* content in different documents and act as if it is one single document. It is like, but more powerful than, LaTeX's include or input commands. For example, *XInclude* allows us to include only a part of the document. To specify which nodes to include, we use *XPath* as part of the *XPointer* language to identify the nodes we want.

We also mention here the eXtensible Stylesheet Language (*XSL*) [1, 12, 13], which is an *XML*-based language for transforming *XML* documents into different forms, be they other *XML*/*HTML* documents or text. We often use *XSL* to generate *HTML* or *PDF* from our articles written in *DocBook*. We can also use *XSL* to convert the data in a tree to a different format, e.g., CSV, where appropriate. *XSL* templates, or rules, for processing and transforming a node are written using *XPath*.

These *XPath*-based technologies illustrate that *XPath* is a general technology and very useful to know for many purposes, not just extracting data from *XML* content.

4.2 *XPath* and the *XML* Tree

XPath is a language for querying and locating elements in an *XML* document. It operates on the hierarchy of a well-formed *XML* document to specify the desired chunks to obtain. *XPath* is *not* an *XML* vocabulary; it has a syntax that is similar to but more powerful than the way files are located in a hierarchy of directories in a computer file system. For example, on Windows, the C: drive acts as a root node, and within this, a file is located in a hierarchy of folders (directories) by an expression such as `C:\MyDocuments\Memo_Aug2.docx` . On UNIX, the root node of all file systems is represented by a forward slash /, and within this there are files and sub-directories, e.g., `/Users/nolan/Documents/Memo_Aug2.docx`. Anyone familiar with navigating file-system trees, either with command line utilities in *UNIX* such as `ls` for listing directories and `cd` for changing directory, or with a graphical user interface such as Mac OS X's Finder or Microsoft's Explorer, will find similarities to *XPath* expressions. *XPath*, however, is much more succinct and expressive.

XPath has many similarities to regular expressions. In both cases, we are identifying patterns to match data or content. There is a trade-off between matching too liberally/permissively and being overly specific. We often mix the pattern matching with subsequent *R* computations on the resulting matches. Like regular expressions, experience helps compose correct *XPath* expressions. However, *XPath* is a simpler language and has a simpler computational model to understand than regular expressions.

Let's consider the *XML* document from Example 2-3 (page 33) that contains the currency exchange rates relative to the euro. The basic structure of the document (with the namespaces removed for simplicity) is shown below.

```
<Envelope>
 <subject>Reference rates</subject>
 <Sender>
  <name>European Central Bank</name>
 </Sender>
 <Cube>
  <Cube time="2008-04-21">
```

```
    <Cube currency="USD" rate="1.5898"/>
    <Cube currency="JPY" rate="164.43"/>
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="25.091"/>
   </Cube>
   <Cube time="2008-04-17">
    <Cube currency="USD" rate="1.5872"/>
    <Cube currency="JPY" rate="162.74"/>
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="24.975"/>
   </Cube>
  </Cube>
</Envelope>
```

This snippet of an *XML* document is in the SDMX format. See Example 2-3 (page 33) for more details about this particular *XML* vocabulary. The exchange rates for each currency are in <*Cube*> nodes with the currency and that day's exchange rate given as attributes. The currencies are grouped together in a parent node for each day. This parent is also named <*Cube*> and it has a *time* attribute. To further confuse matters, the collection of daily data are organized as elements of yet another <*Cube*> node. The <*Cube*> is a general way to represent multidimensional data. Here we have three dimensions. There is the overarching <*Cube*> to indicate what it is being measured (exchange rates) and within this the different days and within day the different currency values.

The following *XPath* expression,

```
/Envelope/Sender/name
```

locates the <*name*> element near the top of the document. The document hierarchy shown in Figure 4.1 shows the realization of this *XPath* expression, i.e., the nodes that have been identified by the query. We can evaluate the query in *R* with

```
nm = getNodeSet(doc, "/Envelope/Sender/name")
```

An *XPath* expression defines a *location path* consisting of one or more *location steps*, each separated by a forward slash. In this expression, we start at the root (/) and look for a child element named <*Envelope*>. Having found that, we continue with the next step in the search, and from this position we look for a child node named <*Sender*>. Finally, we start from this <*Sender*> node and search for a child called <*name*>. Here the steps are very specific, but we will see that they can be much more general, e.g., any descendant at any level.

The *XPath* computational model is designed to identify *node-sets*, which are collections of nodes in the target tree that meet the criteria in the *XPath* expression. The result of our query in *R* is of class `XMLNodeSet` and is a list of references to those nodes which the *XPath* query matched. This is a set in the sense that there are no repeated elements, i.e., each node in the result is unique within this result. In this case, it contains a single <*name*> element from the document, but in general the expression can match more than one node. Indeed, in our example, there was only one matching node *at each location step*. However, there might be many, and *XPath* follows all matching nodes at each step. In this way, it is vectorized in its searching.

For an example that matches multiple nodes, consider the *XPath* expression

```
/Envelope/Cube/Cube
```

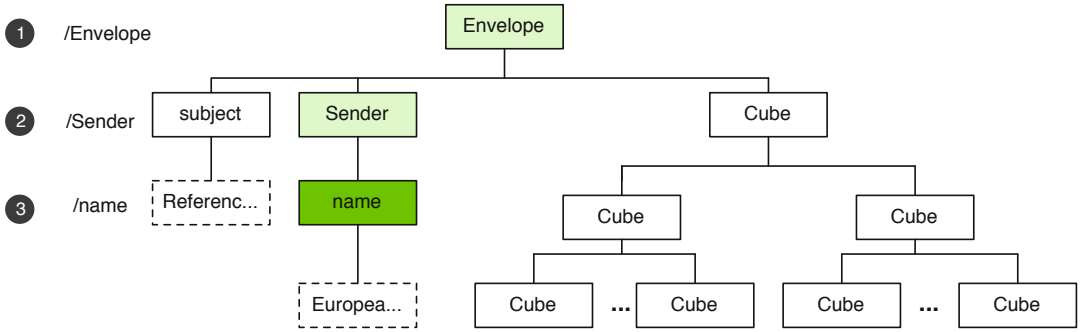


Figure 4.1: Simple *XPath* Expression Applied to a Tree. The shading in this diagram shows how the *XPath* expression, `/Envelope/Sender/name` locates the <*name*> node. The shaded nodes are location steps in the path to the matching node, which are progressively more brightly shaded as we move from one location step to the next and get more specific in the query.

1. The first location step identifies the root node, <*Envelope*>.
2. The next step locates the <*Sender*> child of <*Envelope*>.
3. The third step identifies <*Sender*>'s child called <*name*>.

This expression matches two nodes as shown in Figure 4.2, corresponding to the two days of data in our document. These matches are two sibling <*Cube*> nodes that are grandchildren of <*Envelope*>, and children of the topmost <*Cube*> node.

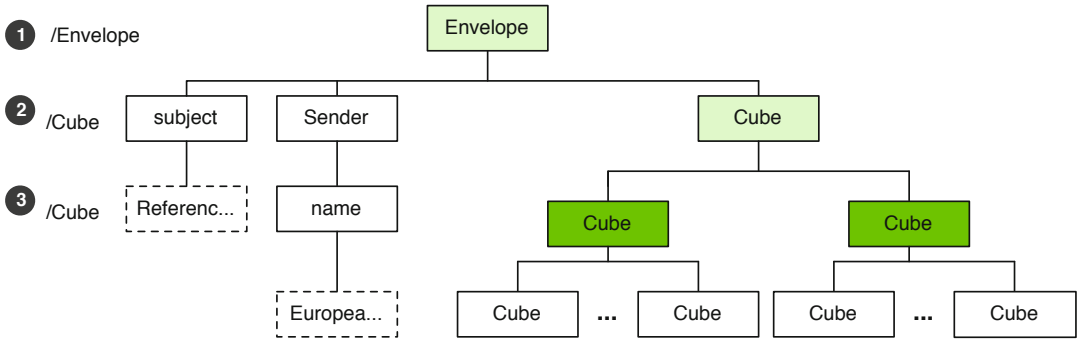


Figure 4.2: *XPath* Expression Locating Multiple Nodes in a Tree. The shading in the diagram shows how the *XPath* expression `/Envelope/Cube/Cube` locates two sibling <*Cube*> nodes. The lightly shaded nodes denote steps in the path toward the match of the two nodes that are brightly shaded in the diagram.

1. The first location step identifies the root node, <*Envelope*>.
2. The next step locates the <*Cube*> child of <*Envelope*>.
3. The third step identifies the two <*Cube*> children of the second-level <*Cube*> node matched from the second step.

The notion of a node-set, i.e., where a node can occur just once in the set, may seem problematic. A node may match multiple conditions in a composite *XPath* expression, i.e., where we specify two or more node tests in the *XPath* query. In contrast, when we subset the same element multiple times in an *R* vector, we explicitly obtain multiple copies of the element. But the concept of a node-set with each element occurring at most once is precisely what makes *XPath* useful. We can find all nodes in a tree that match a particular query and then work with just those. We do not have to worry whether we have already processed that node earlier in the node-set since we know it is unique. We are also guaranteed that the nodes will appear in the node-set in the same order that they occur in the document, i.e., in document order (see page 96). If we did two separate *XPath* queries, we would end up with two node-sets and they might contain some of the same nodes. If we were to process these two node sets, we might end up "double-counting" a node. Also, we would not know if we were processing the nodes in an appropriate order. The node-set is precisely what we want. For example, to extract all exchange rates for the Japanese yen from the SDMX data, we might look for all the <*Cube*> elements with a *currency* attribute value of JPY. The *XPath* expression

```
//Cube[@currency = "JPY"]
```

does just that. The expression `//Cube` is a shortcut for specifying the location step that indicates the <*Cube*> element may appear at any level in the document. It means: match all of the descendants, including the current node, by name. Having obtained the resultant matches, we apply a predicate to restrict the set. The square brackets are analogous to subsetting in *R* and provide a test on the nodes that have matched. That is, the expression within `[]` provides a condition that must be met by these matching <*Cube*> nodes, if they are to remain in the node-set. The expression `//Cube` matches all <*Cube*> nodes, but the condition `[@currency = "JPY"]` filters out those nodes that do not have a *currency* attribute or whose *currency* attribute does not have the value `"JPY"`. The @ symbol is shorthand for attribute in *XPath*. Two nodes match our expression; these are the two nodes with an exchange rate for the yen, as shown in Figure 4.3.

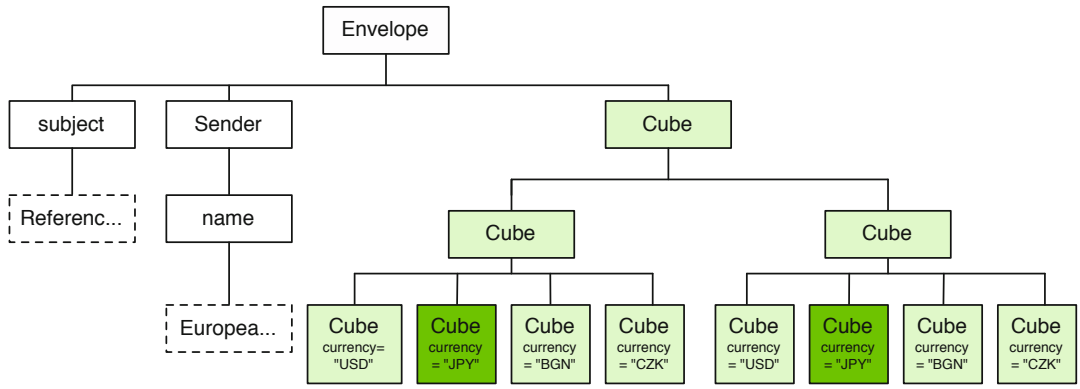


Figure 4.3: *XPath* Predicate Filters a Nodeset. The *XPath* expression, `//Cube`, matches all Cube nodes anywhere in the document. The // is shorthand in *XPath* for all descendant nodes from this point down, including this "self" node. The addition of the predicate `[@currency="JPY"]` filters the set of matches to those elements that have a *currency* attribute with a value of "JPY". In this case, two nodes meet this condition. The shaded nodes highlight all the <*Cube*> elements in the document, and the two darkest satisfy the predicate.

We can also use *XPath* to find nodes based on their currency value. For example, we might want to locate the rates for the Czech koruna when the exchange rate was less than 25. We can find this with the *XPath* query

```
//Cube[@currency = 'CZK' and @rate < 25]
```

This combines two tests (currency value and rate) on each <*Cube*> node, filtering out those that do not match.

These four examples demonstrate the power of the *XPath* language for specifying content in an *XML* document. In addition to the matching capability in these simple expressions, *XPath* allows the specification of complex relationships in a document's hierarchy. The notion of the parent, ancestor and sibling to a node can be part of the expression, which means that the steps in the location path can travel up and to the side in the hierarchy, i.e., not just down the tree. For example, if we want to find the dates when the Czeck koruna dropped below 25, then we need the *time* attribute on the parent <*Cube*> nodes that we have located. We can get this by backing up one level in the hierarchy with the following *XPath* expression

```
//Cube[@currency = 'CZK' and @rate < 25]/../@time
```

XPath also includes functions that we can use to perform computations on a node. For example, the `starts-with()` function allows us to compare the beginning of a node's content or name with a string, and `last()` determines if the node is the last matching element in the current node-set. We will explore these functions after we examine the formal details of the *XPath* syntax and computational model.

4.3 *XPath* Syntax

The examples of the previous sections have informally introduced *XPath* expressions, but to put the full power of the language to use we need a more formal description of its syntax and computational model. An *XPath* expression is made up of one or more *location step*s, separated by a / character. An individual location step has three distinct parts, the *axis*, *node-test*, and an optional *predicate*, which we specify in the following format:

```
axis::node-test[predicate]
```

(The shorthand forms we have used in earlier examples in this chapter deviate from this syntax, but all *XPath* steps can be written in this form.) The axis is separated from the node-test by two colons, (e.g., `::`). This allows us to use a namespace prefix in the node-test, e.g., `descendant::r:code`. The predicate, if present, is specified within square brackets, e.g., `child::Cube[@currency = 'JPY']`.

We specify an entire location path by separating the location steps with the forward slash (/), e.g.,

```
/child::Envelope/child::Cube
```

The *XPath* engine breaks the path into location steps, ignoring extraneous white space, and then evaluates each step in turn. At the end of each step, there is a set of matching nodes. Each of these are used as the current location when evaluating the next location step. The axis indicates which direction to search and how. It might be to look at the child nodes, or alternatively all descendants, or the siblings to the right, all relative to the current starting point for this step. Given the direction to search, *XPath* applies the node-test. This typically matches the name of a node to the specified name, e.g.,

Cube. However, the node-test can also test the type of a node such as text(), comment(), or the generic node(). The (optional) conditions in the predicate further restrict the matching nodes.

The examples from the beginning of this chapter appear to have a slightly different syntax than that just described because those expressions had no :: separating the axis from the node-test. This is because we used the shortcuts that *XPath* provides to specify some common axes. Instead of `/Envelope`, we can write `/child::Envelope` to mean the child is the axis. Similarly, `//Cube` is equivalent to

```
/descendant-or-self::Cube
```

Instead of `@currency = "JPY"`, we can write

```
attribute::currency = "JPY"
```

Clearly, the shortcuts are more convenient. In particular, the default axis (child::) can be dropped entirely from the location step. While these shortcuts are very handy, it is important to think in terms of axes to fully understand what an *XPath* expression means and how it will be evaluated.

XPath is very simple, and a good understanding of axes, node-tests, and predicates allows us to do very powerful processing both efficiently and succinctly. Next, we describe these three parts of the location step in more detail.

4.3.1 The Axis

The axis provides both the direction to look for nodes (from the current context) and also how to look along that direction. Directions are expressed using the family relationship terminology from the document hierarchy, e.g., child, parent, sibling, ancestor, descendant. The **child** axis looks down the tree one level (from the current location or context) to the immediate child nodes. The **parent** and **ancestor** axes look up the tree from the current location. The **parent** axis looks up one level to the immediate parent node while the **ancestor** looks at the parent node, the parent's parent, and so on up to the root node. The **ancestor-or-self** axis considers all the ancestors as well as the current node (the current nodes is the **self** axis). Similarly, **descendant** and **descendant-or-self** work on descendant elements in the tree. The axes **preceding-sibling** and **following-sibling** look along the same level of the current context at all of the preceding siblings (to the left) and the following siblings (to the right), respectively.

Axis Shortcuts and Abbreviations

Some axes are very common so there are shortcuts for these that make the *XPath* expression more succinct and clearer to read. These shortcuts can be used in a location step within the location path. The most common axis is **child**. This is the default axis and can be omitted from any *XPath* step.

Example 4-2 Simplifying XPath Axes to Locate SDMX Nodes

For example, the following expression:

```
/child::Envelope/child::Cube/child::Cube/child::Cube
```

consists of four location steps, each one proceeding down one level to the children of each of the nodes resulting from the context of the previous step. Written in this form, the *XPath* expression is very explicit. However, it can also be written as

```
/Envelope/Cube/Cube/Cube
```

This is more convenient to write and, when one is familiar with the *XPath* syntax, easier to comprehend. It also reminds us of navigation a file system of directories. How does this *XPath* expression work? It begins at the top of the document (denoted by the first /). The first step looks in the direction of the children of the current context—the root of the document—and searches for a node named Envelope, which it finds. The current node-set is then this single <*Envelope*> node. In the second step, we look from the current context of the <*Envelope*> element in the direction of its children to locate all nodes named Cube. Again, there is only one <*Cube*> child of <*Envelope*> and this in turn is the context or node-set for the third step. At this step, since the topmost <*Cube*> has two children, both are found. and multiple nodes satisfy this node-test. Then, in the final location step, the context is evaluated for each of the nodes from the previous step. For each of these two nodes, we look in the direction of the node's children and find many child <*Cube*> elements. The resulting node-set will be all eight great-grandchild <*Cube*> elements of <*Envelope*>, i.e., those with the *currency* attributes. This is the union of the two sets of four <*Cube*> elements coming from evaluating the final step on each of the two nodes that were located in the third location step. (See Figure 4.1 for a diagram of the hierarchy).

Another commonly used abbreviation is the double forward slash **//** which is shorthand for the axis **descendant-or-self**. For example, the expression `//Cube` starts at the root node and matches eleven nodes—the topmost <*Cube*> that is a child of the <*Envelope*> node, each of the <*Cube*> nodes with a *time* attribute, and then all eight of the <*Cube*> nodes with a *currency* attribute. The first of these nodes can be matched with an absolute path `/Envelope/Cube`. The advantage of the **descendant-or-self** axis is that we need not specify the exact and complete steps in the path so many nodes at different levels in the hierarchy can match the expression. In this case, nodes match at three different levels of the tree, e.g., eight of the matches are great-grandchildren of the root element. A disadvantage is that `//` requires *XPath* to traverse every descendant node. If the document is large, an exact path can be more efficient and also more specific. However, less specific and more inclusive expressions insulate us from knowing the exact details of the structure of the *XML*. This can be good if the documents change slightly across documents, but have the same basic content and structure. This trade-off between specificity and generality is quite similar to that when working with regular expressions.

Since **//** is an abbreviation for an axis, when we include it in a location step, we should still have an additional / character to separate the step from the other steps. This would mean we would have three /s in a row. *XPath* allows us to omit the separator / and abbreviate this to simply //, e.g., `/Envelope//Cube`.

Example 4-3 Using Parent and Attribute Axes to Locate Dates in an SDMX Document

Earlier we found the dates when the exchange rate for the Czeck koruna dropped below 25, by filtering the exchange rate nodes to locate those for the koruna that had rates below 25, and then we moved from each of these nodes up to its parent to access the date. Our *XPath* expression was

```
//Cube[@currency = 'CZK' and @rate < 25]/../@time
```

This expression uses several shortcuts—**descendant-or-self**, **attribute**, and **parent**. The fully qualified *XPath* expression would be

```
/descendant-or-self::Cube[attribute::currency = 'CZK'
                          and attribute::rate < 25]/
     parent::node()/attribute::time
```

The first location step locates all <Cube> nodes in the document and filters them to those that have an exchange rate for the koruna that is below 25. The second location step reverses up the tree to the parent of each node located in the first step. Then, the third location step looks along the attribute direction for the *time* attribute.

A list of axes and their abbreviations is provided in Table 4.1.

Table 4.1: *XPath* Axes

Axis	Description
child	Child elements of the context node. Since it is the default axis, it does not have to be
default axis	specified, e.g., `child::Sender` is equivalent to `Sender`.
attribute	Locates attributes of the context node.
Abbreviation: @	
parent	The parent node of the context node. There is no need to specify the name of the parent
Abbreviation: **..**	node because there is only one parent, e.g., `../` is a shortcut for `parent::node()`.
self	The context node itself, e.g., `./text()` or `./`. The self axis is used in functions in
Abbreviation: **.**	predicates, e.g., the `name(.) = 'Send'` expression can be used to test the name of the context element.
descendant	Any child, child of a child, and so on of the context node.
descendant-or-self	Any child, child of a child, and so on of the context node, along with the node itself. While not technically a shortcut for an axis, // can be used as a shortcut for `/descendant-or-self::node()/`. Note also that when locating all descendants or self of the root, we also use the shortcut `//` rather than `///`.
Abbreviation: **//**	
ancestor	Any parent, parent of a parent, and so on of the context node.
ancestor-or-self	Same as **ancestor** but also includes the current node itself.
following-sibling	All elements that follow the context node and share the same parent.
preceding-sibling	All elements that precede the context node and share the same parent
namespace	Locates the namespace nodes.
following	Matches all nodes after the current node in terms of the document order.
preceding	Matches all nodes before the current node in terms of the document order.

This table lists the different XPath axes we can use in an XPath location step. These specify the direction to search in the tree and how to perform that search, e.g., look at the child nodes, or at all descendant nodes.

4.3.2 The Node Test

The node-test component in a location step identifies the name or the type of node to be matched. This is often just simply the name of the nodes in which we are interested, e.g., Cube, Sender, or molecule. As mentioned earlier, there may be times when we are using multiple vocabularies and the same name is used in two different contexts. When this occurs, we use namespaces in our *XML* document to clarify the provenance and meaning of a the node name, e.g., <r:class> and <py:class>. When a document has multiple namespaces, *XPath* requires that we specify the namespace in our node-test. For example, to find *R* <code> nodes in an *XML* document, we might need to distinguish these <code> nodes from other vocabularies that use <code> as a node name. We can use a namespace prefix in the node-test to find all *R* <code> nodes in <example> nodes as follows:

```
//example//r:code
```

or, more explicitly,

```
/descendant-or-self::example/descendant-or-self::r:code
```

Of course, we need to associate the namespace prefix `"r"` with the appropriate URI, i.e., `http://www.r-project.org`. We will discuss this issue further in Section 4.8, where we will see how to specify the namespace definitions when using *XPath* in *R*.

At times we want to match a node with any name, and not a specific fixed name. In this case we can use the asterisk (`*`) as a wildcard that matches all named elements. For example, in Example 4-1 we used the *XPath* expression

```
/*/molecule[.//name/text() = 'frm-1']
```

to locate <*molecule*> nodes one-level down without having to know the name of the top node. The `*` symbol is really shorthand for the `node()` function that matches any regular node, i.e., it does not match text, comments, attributes, and processing instructions.

There are other circumstances where we want to match elements in the tree that are not regular nodes and do not have a tag name, e.g., a text node or a processing instruction node. There are *XPath* functions for specifying these. If we want to match text nodes we use the node-test **text()**. Similarly, we use **comment()** to match comments and **processing-instruction()** to match any processing instruction. If we want to match only processing instructions with a particular target, we pass the target name as a string in the call to `processing-instruction()`, e.g.,

```
//processing-instruction('R')
```

We will see later in Section 4.4 how we can use compound expressions to match two or more node names.

4.3.3 The Predicate

Predicates allow us to further restrict the node-set, but they are not always needed and so can be omitted. A predicate tests each of the candidate nodes matched by the node-test part of the location step. For each of these nodes, the expression in the predicate is evaluated in the context of that node. If the result of the comparison is true, then the node remains in the node-set; if not, it is discarded. That is, the predicate filters the node-set. This is similar to subsetting in *R* with a logical vector, i.e.,

```
nodeset[ sapply(nodeset, predicate) ]
```

It is very important to understand that the predicate is evaluated in *XPath* in the context of the node or *XML* element we are testing in the node-test, not in the parent node's context.

The syntax of the predicate is: `[ logical-condition ]`. These conditions can be unary or binary operations. For example, `Cube[@currency]` and `Cube[not(@currency)]` are examples of unary operations which test for the presence and absence of a *currency* attribute, respectively.

As another example, we can match <*section*> nodes that contain a <*figure*> element with

```
//section[ .//figure ]
```

This expression matches and returns the <*section*> nodes, not the <*figure*> nodes, and keeps only those <*section*> nodes that have a <*figure*> element as a descendant. Note that we use the shorthand **.** for the current node or **self**.

Binary operations are very common and have the form

```
[ expr1 booleanOperator expr2]
```

where `expr1` and `expr2` are compared via *XPath* boolean operators (i.e., `=`, `!=`, `>`, `>=`,`<`, and `<=`). We saw two examples of this earlier:

```
Cube[ @currency = 'JPY' ]
Cube[ @rate < 25 ]
```

In both examples, we test on the value of an attribute. This automatically fails if the node has no attribute with that name. In the case of comparing the *rate*, *XPath* coerces the value of the attribute from a string to a number for us. There are functions in *XPath* (e.g., `number()`) to do this explicitly if we need more control over how the conversion is done. Note that the test for equality is a single =, not two (==) as in *R* and other languages. This is because there is no assignment in *XPath*.

Predicates can appear in any location step, and indeed there can be multiple predicates in a single step. This is the topic of the next section. In that section, we will also explore several *XPath* functions for working with text and nodes.

The *XPath* Location-step

XPath locates sets of nodes in *XML* documents. An *XPath* expression is a *location path* that is made up of one or more location steps. Each step has two required parts—the axis and node-test—and an optional predicate as the third part. The location step follows the syntax:
`axis::node-test[predicate]`

The location step can be thought of as directions from one location (or context) to another. The direction is relative to each node in the the current set of nodes, as computed by the previous step. The step indicates which direction to look (axis), the node(s) name or type to locate (node-test), and the filter or subset condition to apply to the qualifying nodes (predicate). *XPath* ignores white space (blank spaces and new lines) within a location step and between steps which allows us to format the expressions freely.

axis:: Orients the search and is expressed in the vocabulary of a tree hierarchy, e.g., `child` looks at the children of the current context, `parent` looks at the parent, and `descendant-or-self` looks at the current context and all nodes that descend from it. These axes are often abbreviated. For example, the `child` axis is the default and can be omitted entirely from the location step. Also, `descendant-or-self::foo` can be abbreviated to `//foo` and `attribute::name` can be expressed as `@name`. We can refer to the current node and the parent with `.` and `..`, rather than `self::node()` and `parent::node()`, respectively.

node-test Provides the element name or an element type to locate in the location step. For elements with no names, e.g., text and comments, we use functions, such as `text()` and `comment()` to locate elements by type. We can use the generic `node()` function to identify any regular/named node (i.e., not text, comment or processing-instruction). The wildcard shortcut `*` also matches any regular node. Keep in mind that `*` is not a regular expression or glob, but matches any node name in *XPath*.

[predicate] This part of the location step operates on the nodes matching the `axis::node-test` and filters the qualifying nodes to those that meet the conditions of the predicate. The expression in the predicate is applied to each node in the node-test and if the result is `true`, the node remains in the node-set. *XPath* provides many functions to use in the predicate. For example, `[position() = last()]` keeps only

the last node in the node-set; this can be abbreviated to `[last()]`. Similarly, we can subset by position, e.g., `//section[2]` yields the second <*section*> node. There are functions to compute the name or value of a node and also to perform string manipulation and comparisons. Simple expressions can be combined using the `and` & `or` operators, e.g.,

```
//Cube[@currency = "USD" or @currency > 1.5]
```

Location steps are concatenated together with `/` to form a location path. At each step in the location path, the step's expression is evaluated within the current context, i.e., context for each node that matched the previous step. For example, with

```
//Cube[@time]/Cube[@rate < 25]
```

or

```
/descendant-or-self::Cube[@time]/child::Cube[attribute::rate<25]
```

the first step matches the two <*Cube*> nodes that have a *time* attribute. For each of these, we evaluate the next step which searches for a child <*Cube*> node with a *rate* attribute with value less than 25. The result is the union of the matching nodes from each of the two separate searches.

More than one element can be located by an *XPath* expression. The located nodes are called the *node-set*. Each matching node appears in the node-set just once. This is useful and especially important to remember when we work with compound *XPath* queries, i.e., using multiple *XPath* expressions together to search for this OR that.

4.4 *XPath* Functions and Logical Operators

XPath provides logical operators for combining predicates. Predicates can be combined together into a compound predicate using one of the binary operators `and` or `or`. For instance, to match <*Cube*> nodes for the US dollar or Japanese yen, we can use

```
//Cube[@currency = "JPY" or @currency = "USD"]
```

Other boolean operators in *XPath* include: `not()`, `true()`, and `false()`. The `not()` operator is used to compute the opposite or negation of a condition. It is analogous to the `!` operator in *R*, and we can use it in an *XPath* expression such as

```
//graphic[ not( contains(@fileref, '.jpg') )]
```

to find all <*graphic*> nodes that do not have a *fileref* attribute with the extension **jpg**. We should note, as an aside, this expression is not precise enough for two reasons. Firstly, it matches <*graphic*> elements which have no *fileref* attribute. We can remedy this by testing for the presence of the *fileref* attribute before the test for the extension, i.e.,

```
//graphic[ @fileref and not( contains(@fileref, '.jpg') )]
```

The second problem is that this does not test for the string `".jpg"` at the end of the file name. We would like to use a function such as `ends-with()`, similar to `starts-with()`. Unfortunately, *XPath* 1.0 does not provide such a function. However, we can get the same effect with `substring()` and `string-length()` via

```
//graphic[ @fileref and
          not( substring(@fileref, string-length(@fileref) - 3, 4)
                 = '.jpg' ) ]
```

This expression illustrates that *XPath* has no direct equivalent of *R*'s `!=` operator. *XPath* 1.0 was designed to be minimal, where you can build all of the functionality with a few primitive functions.

Predicates can look at both the content and location of the node they are testing to see if these match the condition(s). For this, we need to be able to perform computations on the node and its contents. *XPath* provides many functions that are helpful in constructing predicates, and queries generally. Some of these functions access the context of a node. For example, `position()` returns a numeric value giving the position of the node in the current node-set associated with the predicate. This function can be quite useful in extracting a specific node according to its position. As an example,

```
//section[position() = 2]/r:code[position() = 1]
```

locates the first <*r:code*> node in the second section of the document. This can be abbreviated to

```
//section[2]/r:code[1]
```

That is, *XPath* treats `[2]` as `[position() = 2]`. Unlike in *R*, the expression `[2]` is actually an implicit logical predicate. We do not have to use the logical form, but it is good to know how it is being evaluated.

As another example, suppose we wish to extract the last node in a node-set, and we do not know the number, just that it's the last node in the node-set. The *XPath* function `last()` combined with `position()` in the predicate below returns true when the position of the node in the node-set matches the position of the last node in the node-set (i.e., `last()` is equivalent to the size of the node-set).

```
/Envelope/Cube/Cube[position() = last()]
```

As before, this predicate can be abbreviated to `[last()]` because when the result of a predicate expression is a number, then *XPath* treats it as a logical condition that compares this number to the context node's position and, if they match, returns `true`.

In addition to `position()` and `last()`, there are many functions available in *XPath* for use in predicates. Some provide access to a node's properties, i.e., to the node's name (both local and qualified by its namespace), its position within the node-set, family relationship with other nodes in the tree, and string-value. In addition to functions that operate on a node, *XPath* provides functions that operate on strings. The most commonly used of these functions are summarized in Table 4.2.

As an example, we find <*r:code*> nodes that contain the word `'library'` with

```
//r:code[ contains(., 'library') ]
```

There are also several functions in *XPath* that deal with numbers. The function `number()` converts its (string) argument to a number, e.g., `number(@rate) > 1`. As we mentioned, *XPath* typically does the implicit conversion for us. The functions `floor()`, `ceiling()`, and `round()` perform the corresponding tasks as the functions in *R*. For example, we can find all magnitude 6 earthquakes with

```
//event[floor(number(./mag)) = 6]
```

where each earthquake is of the form

```
<event>
 <mag>number</mag>
....
</event>
```

Table 4.2: *XPath* Functions

Function	Input	Return Value
`last()`	node	Number of elements in the context node-set.
`position()`	node	Position of the context node within the node-set.
`count()`	node	Number of elements in the node-set.
`id()`	node	Element with *id* matching the input string.
`name()`	node	Name of the first node in the node-set.
`namespace-uri()`	node	Namespace of the context node or the first node in the node-set.
`concat()`	string	One string that concatenates the strings provided as input arguments.
`starts-with()`	string	`true` if the first string passed to the function begins with the second string. For example, `starts-with(@fileref, 'Images/')`
`contains()`	string	`true` if the first string contains the second.
`substring()`	string	Portion of the string starting at the first value for a length of the second value.
`substring-after()`	string	Portion of the first string that appears after the second string.
`substring-before()`	string	Portion of the first string that appears before the second string.
`string-length()`	string	Number of characters in the string.
`normalize-space()`	string	String with leading and trailing whitespace stripped and reduced if the second string starts with the first string.
`translate()`	string	Original string with the portion of the string starting at the first value for a length of the second value. For example, to change a, c, g and t to A, C, G and T respectively, we use `translate('acgt', 'ACGT', string(.))`

This table describes some of the important XPath (1.0) functions that we can use within XPath expressions.

Additionally, *XPath* supports the usual arithmetic operators `+`, `-`, `*`, `/`, and mod (`%`).

We should note that these functions are useful not only in predicates. *XPath* is used in the related technology *XSL* for creating transformations of *XML* documents. In these cases, we can output the results of computations into text or nodes in other *XML* documents. For instance, we can convert references to JPEG file names to PNG by replacing the extension using `substring-before()`. Similarly, we can do calculations across nodes in a node-set to compute aggregates, e.g., using `sum()`. Unfortunately, the set of numeric functions in *XPath* is quite limited, not even including the log function. As a result, we often do computations in *R*. We can even use *R* functions within *XSL* templates by integrating *R* and the *XSL* transformation engine (*XSLT*) [3]. The Sxslt package [11] makes this possible, embedding *XSLT* in *R* and also *R* in *XSLT*.

It is important to note that getNodeSet() and related functions in *R* use *XPath* 1.0 via the `libxml2` *C*-level library. *XPath* 2.0 provides additional and richer functions than are available in *XPath* 1.0, but we cannot use those within getNodeSet(), etc. While it would be convenient to use these additional functions, we do not actually need them. Instead, we can perform simpler *XPath* queries and then apply our own predicates or transformations to the nodes that getNodeSet() returns. We have a much richer language and set of functions in *R* than is available in *XPath* 2.0. Therefore, we can combine *XPath* and *R* to perform the computations we need. Readers interested in more powerful facilities than *XPath* 1.0 might explore the *XQuery* language [2, 15]. The RXQuery package [10] is a prototype of integrating *R* and Zorba [5], an Open Source implementation of *XQuery*.

4.5 Multiple Predicates in a Node Test

Predicates can be nested within one another and they can be stacked one after the other. For example, to find <section> nodes that contain a <table> node where the <table> node has a <title> containing the string "XPath Functions", we can use the expression

```
//section[ .//table[ title = "XPath Functions"] ]
```

Notice the use of the **self** axis in the predicate.

We call predicates stacked if they are specified in sequence, just as we do in *R*. For example,

```
getNodeSet(root, "//Cube[@rate > 25][2]")

[[1]]
<Cube currency="CZK" rate="25.091"/>
```

returns the second node that has an exchange rate above 25. We should be cautious in stacking predicates. We might think that this stacked predicate retrieves the same set of nodes as the compound predicate here

```
getNodeSet(root, "//Cube[@rate > 25 and position() = 2]")

[[1]]
<Cube currency="JPY" rate="164.43"/>

[[2]]
<Cube currency="JPY" rate="162.74"/>
```

They clearly do not. The compound predicate

```
[@rate > 25 and position() = 2]
```

is evaluated in the context of the <Cube> nodes that pass the node-test. Here, each <Cube> node for the yen is the second child of its parent and the <Cube> node for the Czech koruna is in the fourth position. Since the two nodes for the yen have rates above 25, they remain in the node-set. The value of the `position()` function is 2 for more than one node. In fact, it has this value for three nodes; that is,

```
getNodeSet(root, "//Cube[position() = 2]")
```

yields

```
[[1]]
<Cube currency="JPY" rate="164.43"/>

[[2]]
<Cube time="2008-04-17">
  <Cube currency="USD" rate="1.5872"/>
  <Cube currency="JPY" rate="162.74"/>
  <Cube currency="BGN" rate="1.9558"/>
  <Cube currency="CZK" rate="24.975"/>
</Cube>

[[3]]
<Cube currency="JPY" rate="162.74"/>
```

We can see from Figure 4.1 that of our 11 <*Cube*> nodes, one at the second level and two at the third level will be in the second position when the predicate is applied.

On the other hand, the predicate `[2]` in the stacked expression is evaluated in the context of the nodes that have been filtered by `[@rate > 25]`. In this case, the `CZK` nodes both have a value of 2 for `position()`, and for the two `JPY` nodes the `position()` function evaluates to 1.

Expressions with Multiple Predicates

Predicates can be simple or compound and they can be nested, i.e., a predicate within a predicate. Additionally, predicates can appear on multiple location steps within an *XPath* expression or path, and a single location step can have multiple predicates, one after the other.

Simple A simple logical condition such as

```
//Cube[@currency > 1.5]
//section[.//table]
```

to identify a <*Cube*> node that has a *currency* attribute value greater than 1.5 or a <*section*> node that contains a <*table*> element.

Compound Simple expressions combined using the operators `and` & `or`, e.g.,

```
//Cube[@currency = "USD" or @currency > 1.5]
```

and

```
starts-with(., 'abc') and contains('xyz')
```

Nested Two or more predicates nested within each other. For example, the expression:

```
//div[.//table[ contains(string(./th), 'Price')]]
```

finds an *HTML* <*div*> node that contains a table that contains the word `Price` in at least one of its column headers.

Multiple location steps Predicates can appear on more than one location step in an *XPath* expression. Both the section and paragraph location steps in the expression

```
/book/chapter/section[1]/para[3]
```

contain predicates. As a result, this location path finds the third paragraph in the first section of each chapter in the book.

Stacked One predicate can follow another in a location step, which is called a stacked predicate (this term was coined by [9]). When this happens, the second predicate is evaluated in the context of the first, i.e., it is applied to those nodes that satisfy the first predicate. For example, the first predicate in the expression

```
//section/para[code][2]
```

locates all paragraphs that contain code nodes and the second predicate selects the second of these paragraphs. In contrast, the following expression,

```
//section/para[2][code]
```

first selects all paragraphs that are the second paragraph in a section; then the second predicate keeps those second paragraphs that contain code.

Examples of the versatility and power of these complex predicates appear in Example 4-9 (page 98).

4.6 Combining *XPath* Location Paths in a Single Query

There is one final and important aspect of using *XPath* to match nodes, which we alluded to earlier: multiple criteria in a single query. For example, we might want to find all references to external files in an *HTML* document including images in <*img*> nodes, *JavaScript* [4] code in <*script*> nodes in the <*head*> node of the document, and Cascading Style Sheet (*CSS*) files also referenced in the <*head*> node. Alternatively, when working with an article or a book, we might want the <*title*> nodes within a <*table*> or a <*figure*>. We can typically do these sorts of queries with multiple criteria in multiple passes with separate calls to getNodeSet() for each individual criterion or query. For example, in our second example, we can use

```
tbl = getNodeSet(doc, "//section//table/title")
fig = getNodeSet(doc, "//section//figure/title")
```

However, while not true in this case, performing multiple queries can mean that we end up with the same node present in each query. This destroys the simplicity of the set characteristic of a node-set where we know each node is unique. Also, this involves traversing the tree multiple times which can be expensive for very large trees.

XPath (and hence getNodeSet(), xpathApply(), etc.) allows us to combine multiple queries into a single query. We separate the individual queries using the `|`, with or without surrounding spaces.

```
ti = getNodeSet(doc,
                "//section//table/title | //section//figure/title")
```

Importantly, by combining the two *XPath* queries, the nodes will be returned in the correct *document order*, i.e., the order in which the nodes appear in the document. This can be important if we need to process them in this order and would not be easily feasible if we had to perform multiple separate queries to get the nodes.

Since these two queries are so similar in structure, there is a natural tendency to avoid repetition and combine the two into a shortened query such as

```
//section//(table|figure)/title
```

or

```
//section(//table|//figure)/title
```

These may seem sensible, but, simply put, they are invalid *XPath* expressions and will cause an error. Each query separated by a `|` must be a complete and valid *XPath* location path in its own right.

4.6.1 Programmatically Generating XPath Queries in R

Since we are working within *R*, we can create queries programmatically to help us. For example, rather than paste the queries together ourselves, we can pass getNodeSet() and xpathApply() a character vector of individual queries. These functions will combine them into a single string, separating the queries with the `|` character. This allows us to keep related queries in a vector and to subset them to perform specific subqueries. To locate the <*title*> nodes within <*table*> or <*figure*> nodes in <*section*> nodes, we can pass getNodeSet() a vector of *XPath* expressions as

```
xpQueries = c(section = "//section/title",
              table = "//section//table/title",
```

```
                figure = "//section//figure/title")
ti = getNodeSet(doc, xpQueries[c("table", "figure")])
```

We can also change individual elements or add new ones without having to manipulate the single string containing all the queries, e.g.,

```
all.titles = getNodeSet(doc, xpQueries)
```

The three *XPath* queries in `xpQueries` have exactly the same structure but differ only in the element we look in for the <*title*>. We can create this more readily with

```
xpQueries = sprintf("//section%s/title",
                    c("", "//table", "//figure"))
```

This illustrates that we can create programmatically *XPath* queries in *R* using string manipulation and substituting values for *R* variables. We provide additional examples of this programmatic approach.

Example 4-4 Creating Multiple XPath Queries for Exchange Rates

We can create a query to get the exchange rates for different currencies with something like

```
currencies = c("USD", "JPY", "BGN")
q = sprintf("//Cube[@currency='%s']/@rate", currencies)
```

This gives us three separate queries and we can evaluate them separately to get the exchange rates for the different currencies with

```
exRates = lapply(q, function(q) as.numeric(getNodeSet(doc, q)))
```

and we can put them into a data frame with

```
as.data.frame(structure(exRates, names = currencies))
```

Should we want to, we can even compute values in one query and put them into another query. We demonstrate how in the next example.

Example 4-5 Using XPath Functions to Retrieve Loan Information for a Large Number of Kiva Loans

Each node in the collection of <*loan*> elements of the Kiva data set gives many details about the loan. These include the name of the person given the loan, their geo-location, the purpose of the loan, the amount, the history of payments, etc. We may be interested in all loans above the 90th percentile for the amount loaned. We could read each loan into *R* and then subset these based on the loan amount. However, this would involve processing the entire node for 90 percent of nodes that we do not want. Instead, we can get the value of the loan from the <*funded_amount*> node with

```
xpx = "//loan/funded_amount"
loanAmounts = as.numeric(xpathSApply(kiva, xpx , xmlValue))
```

where `kiva` contains the parsed *XML* document. Next, we find the quantile of interest with `quantile(loanAmounts, .9)`, and use it to get the nodes that exceed this funded amount:

```
q = sprintf("//loan[string(funded_amount) > %.2f]",
            quantile(loanAmounts, .9))
bigLoans = getNodeSet(kiva, q)
```

Combining Queries and Location Paths

Location paths can be combined using the | operator, where the expression on each side must be a valid location path, e.g.,

```
/book/chapter/section[1]/table | /book/chapter/section[1]/figure
```

locates all tables and figures in the first section of each chapter. We can combine any number of queries together, not just two.

Complicated compound expressions may have nodes that match more than one subexpression. However, the node-set will contain unique nodes and these will be in document order, e.g.,

```
xpQ = c("//section[.//table]", "//section[.//figure]")
getNodeSet(book, xpQ)
```

locates all sections that have either a table or figure in them in the parsed book object `book`. Sections that have both will appear once in the node-set and the sections will be in the order that they appear in the document. This is very different from

```
sapply(xpQ, getNodeSet, doc = book)
```

This returns two lists of nodes (i.e., two node-sets), where sections that have both tables and figures appear in both and the document order is of the union of these two node-sets is lost.

Document Order

XPath adds the matching nodes to a node-set in what is called "document order." This defines which elements of an *XML* document are considered to be "before" other elements. The root element is the first node in the document. Next come any namespace definitions on that node. The attributes are next in order. Next come the child nodes and the order is defined in the same manner for each of those. The order within a set of namespaces on a node is implementation-dependent. The same is true for attributes within a node. Consider the following *XML* document:

```
<section xmlns:r="http://www.r-project.org" id="sec:DocOrder">
<para>
The following figure displays a hierarchy.
<figure>
 <graphic width="6in" format="SVG"
          fileref="images/SDMXEnvelope-Sender.svg"/>
</figure>
The elements of the <xml/> are ordered.
</para>
</section>
```

We can get all of the elements (except the namespace definition) with an *XPath* query

```
getNodeSet(doc, "//text() | //* | //attribute::*")
```

and see the order of the elements. The <*section*> node comes first, then the *id* attribute. Next comes the <*para*> node and then its child nodes. The first of these is the text `"The following ..."`. Next is the <*figure*> node and since it has no attributes, its children are next. This means the <*graphic*> element is next, then its attributes. After this is the next text child of the <*para*> node. This is, `"The elements of the"`. After this is the node <*xml*> and finally we have the text node containing `" are ordered"`.

4.7 Examples of Accessing Data with *XPath*

In this section, we revisit earlier examples from Chapters 1 and 3 and explore how those data extractions can be performed with xpathSApply(), getNodeSet() and *XPath*. In addition, we present a more lengthy example using bibliographic data. The richness of the bibliographic data gives a sense of how we can use complex *XPath* expressions to query and extract data from an *XML* document.

Example 4-6 Retrieving Attribute Values with XPath for Bills in the US Congress

We saw in Example 1-3 (page 13) that `http://www.govtrack.us/` provides data about bills in the US Congress in the following *XML* format:

```
<bill session="111" type="h" number="1"
      updated="2011-01-29T15:03:08-05:00">
 <state datetime="2009-02-17">ENACTED:SIGNED</state>
 <status><enacted datetime="2009-02-17" /></status>
...
 <relatedbills>
  <bill relation="rule" session="111" type="hr" number="88"/>
  <bill relation="rule" session="111" type="hr" number="92"/>
  <bill relation="rule" session="111" type="hr" number="168"/>
  <bill relation="unknown" session="111" type="h" number="290"/>
  ....
 </relatedbills> ...
</bill>
```

In that example, we extracted the bill numbers for a bill's related bills. We can locate the related bills' <*bill*> nodes with

```
xpx = "//relatedbills/bill"
```

and then retrieve the numbers from these nodes with

```
as.integer(xpathSApply(root, xpx, xmlGetAttr,
                       name = "number", default = NA))
```

Example 4-7 Retrieving Magnitude and Time with XPath for Earthquakes in a USGS Catalog

Example 3-3 (page 60) foreshadowed how we might use *XPath* to extract the magnitude of a quake from its <*param*> node. Specifically, we located the <*param*> node that had an attribute *name* with a value of `"magnitude"` with

```
xpx =  "/merge/event/param[@name='magnitude']"
```

As in the previous example, we can combine the location of the elements using xmlGetAttr() with the xpathSApply() function as follows:

```
magValues = xpathSApply(root, xpx, xmlGetAttr, name = "value")
```

We now have the magnitudes of the quakes.

We can also locate the *time-stamp* attribute for each quake with, e.g.,

```
xpx = "/merge/event/@time-stamp"
```

The following call to getNodeSet() locates the time-stamp values and returns them in a list that we then convert to a vector:

```
timestamps = unlist(getNodeSet(root, "/merge/event/@time-stamp"))
```

An alternative approach is to use *XPath* to obtain the <event> nodes (or the <event> nodes that have a *time-stamp* attribute) and then loop over this collection of nodes in *R* to retrieve the value of the *time-stamp* attribute from each. We can do this with xpathSApply() and xmlGetAttr() as

```
timestamps = xpathSApply(root, "/merge/event",
                         xmlGetAttr, "time-stamp")
```

Example 4-8 Extracting Text Content from a Kiva Document

In Example 3-1 (page 54) our goal was to extract the text content from the <occupation> nodes for each of the lenders in our Kiva document. There we used [[and xmlChildren() to access this information. As an alternative, we can locate the desired nodes with *XPath* and then apply xmlValue() to them. We do this with

```
occ = xpathSApply(root, "//lenders/lender/occupation", xmlValue)
```

We now have the information acquired earlier. However, unlike in the earlier example, we do not need to know the depth of the <lenders> node in the document.

Example 4-9 Locating Content in Metadata Object Description Schema (MODS) Entries

The Metadata Object Description Schema (MODS) [7] provides a rich vocabulary for bibliographies. MODS is maintained by the Library of Congress, and it supports standards for a variety of genres, or types of works, including music, images, maps, and software as well as text. In the past, the Journal of Statistical Software (JSS) www.jstatsoft.org has made available bibliographic data in MODS format for its published articles.

In MODS, a list of bibliographic entries are located within the root document node <modsCollection>, which must only contain <mods> elements. Each <mods> element holds detailed bibliographic data about an entry in the bibliography. It supports a vast array of meta-information, e.g., roles for people associated with the work (including author, editor, and translator). Below is one <mods> element in the JSS MODS database. It represents information for a single article published in JSS.

```
<mods ID="Lumley:2004:ACS">
 <titleInfo>
  <title>Analysis of Complex Survey Samples</title>
 </titleInfo>
 <name type="personal">
  <namePart type="given">Thomas</namePart>
  <namePart type="family">Lumley</namePart>
  <role>
   <roleTerm authority="marcrelator" type="text">author</roleTerm>
  </role>
 </name>
 <originInfo> <dateIssued>2004</dateIssued> </originInfo>
```

```
 <typeOfResource>text</typeOfResource>
 <relatedItem type="host">
  <titleInfo>
   <title>Journal of Statistical Software</title>
  </titleInfo>
  <originInfo> <issuance>continuing</issuance> </originInfo>
  <genre authority="marc">periodical</genre>
  <genre>academic journal</genre>
  <identifier type="issn">1548-7660</identifier>
 </relatedItem>
 <identifier type="citekey">Lumley:2004:ACS
 </identifier>
 <location>
  <url>http://www.jstatsoft.org/counter....</url>
 </location>
 <part>
  <date>2004</date>
  <detail type="volume"><number>9</number></detail>
  <detail type="number"><number>8</number></detail>
  <extent unit="page"> <start>1</start> <end>19</end> </extent>
 </part>
</mods>
```

For simplicity, we will work with a pared down version of the <*mods*> element and just four articles. This MODS collection is still rich enough to demonstrate the power of *XPath* for locating elements in a document. Figure 4.4 provides a diagram of the document's hierarchy.

We consider three tasks for matching nodes in this collection of bibliographic entries. We wish to locate the:

1. last name of the first author of all the bibliographic entries,
2. last name of all co-authors of a particular author, say, Narasimhan.
3. identifying *citekey* attribute for all articles by a particular author, say, Lumley.

These exercises demonstrate a variety of *XPath* techniques, including using nested predicates, compound expressions, traveling up and across the tree, and issues of efficiency, specificity, and generality.

1. Locate the surname of the first author of all bibliographic entries

The surname of an author is in the text content of a <*namePart*> that has for its *type* attribute, `type="family"`. The following *XPath* expression identifies the text node containing the family name for each author in the collection. Since text nodes have no name, the `text()` function is used to locate them:

```
/modsCollection/mods/name/namePart[@type='family']/text()
```

There are five family names in our sample document since one of the four articles has two authors. Below is a much simpler expression that yields the same node-set:

```
//namePart[@type='family']/text()
```

If there had been other <*namePart*> nodes elsewhere in the document (with *type* values of `'family'`) this might not have given us what we wanted, as it would have included additional nodes in a different context. However, for this document, the two expressions yield the same node-set.

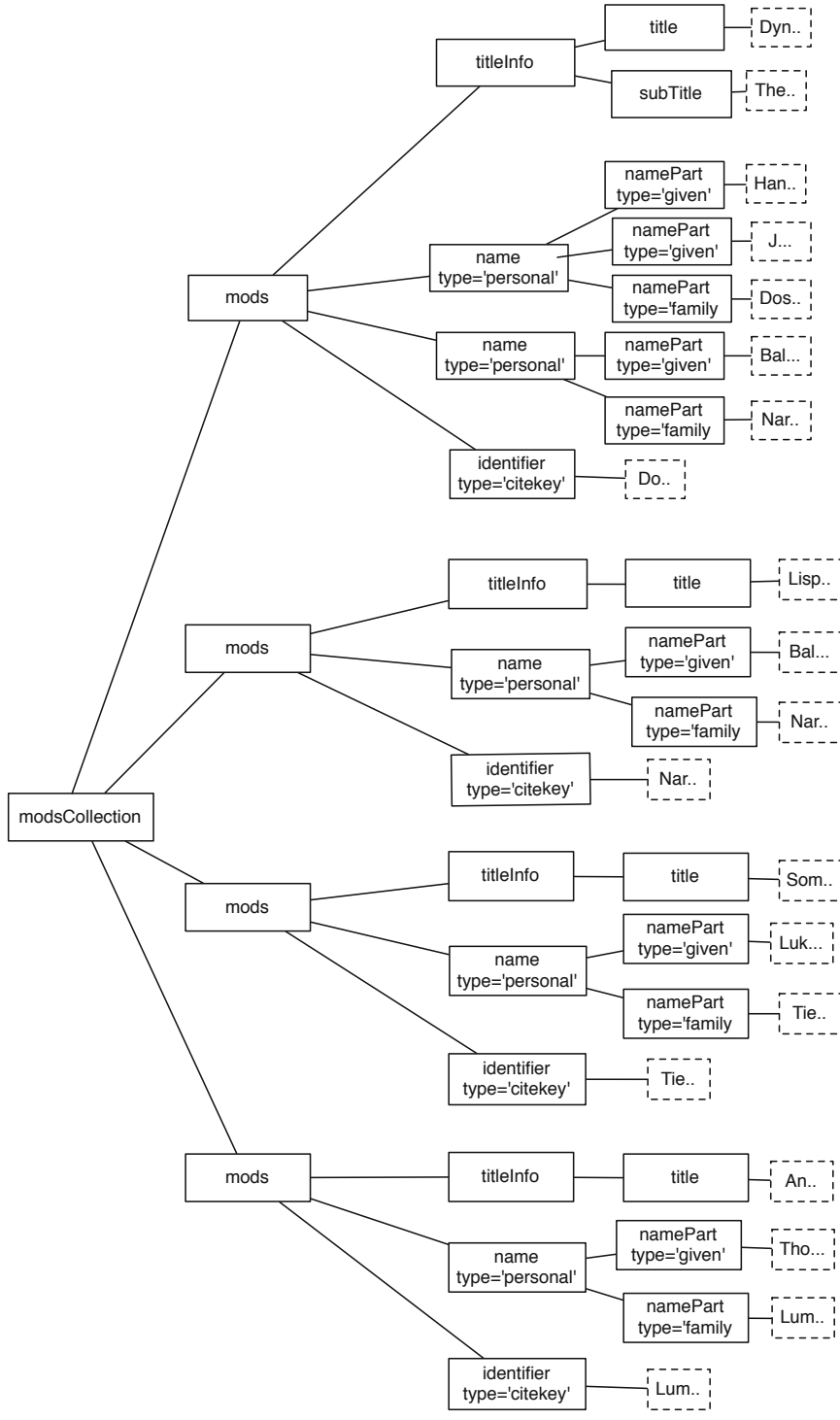


Figure 4.4: Example MODS Document Tree. This tree provides a diagram of a simple MODS document consisting of four articles. Each <mods> element contains information about an article's title, author(s), and identifier. The attribute names and values are also shown in the element nodes. Note the first article (the first and highest <mods> node) has two authors and the lead author of that article has two given names (first name and middle initial). Also, the first two articles have an author in common—Balasubramanian Narasimhan.

We do not quite have the information that we are after because we have all authors, not just first authors. By simply inserting another predicate in the location path, we can narrow the focus to the authors we want:

```
//mods/name[1]/namePart[@type='family']/text()
```

We placed a predicate on the second location step, i.e., `name[1]`. This filters all <*name*> nodes so that we locate only one (the first one) in each <*mods*> element.

We conclude this location exercise with an expression that demonstrates an incorrect way to perform the match:

```
/modsCollection/mods/name/namePart[@type='family' and
                                   position() = 1]
```

The reason this location path does not work as desired/expected has to do with the context in which the predicate is evaluated. The predicate is part of the location step that identifies <*namePart*> nodes. This means that both the attribute (*type*) and the `position()` function in the predicate are evaluated in the context of each <*namePart*>. The result is an empty node-set because the family name appears in the last <*namePart*> node within <*name*>, not the first. The logical conditions, `position() = 1` and `@type = 'family'` are never true simultaneously.

2. Identify the last name of all co-authors of the author with the family name Narasimhan
Our strategy for finding Narasimhan's co-authors is to first find all papers for which he is an author. We do this by finding the <*namePart*> node that has `Narasimhan` in the text content of the family name. We then go up the hierarchy from each of these nodes to find the <*mods*> node for that paper. Then we can descend from there to find the other <*name*> nodes and find the family names of all these authors.

From the previous exercise, we know how to locate the nodes containing the surname or family name of authors. To find a particular author, say, Narasimhan, we can use the following *XPath* expression:

```
//namePart[@type='family' and . = 'Narasimhan']
```

The single period, is shorthand for the **self** axis; in this case it is the <*namePart*> node.

We actually want the cousin nodes of the one just located, i.e., the co-authors' family names. To find them, we can add location steps that change direction and reverse up the tree to locate the sibling <*name*> elements of the author just identified and then get their family names. We show two approaches to do this. The first specifies a location step that reverses up to Narasimhan's <*mods*> ancestor node, and then steps back down to all family <*nameParts*> nodes within that <*mods*>:

```
//namePart[@type='family' and string(.) = 'Narasimhan']/
    ancestor::mods/name/namePart[@type = 'family']/text()
```

The **ancestor** axis retrieves the grandparent <*mods*> element. From there, the location step turns around and goes forward to collect the last names of all authors in the appropriate <*mods*> elements. Note that we have combined all of these navigational steps into a single *XPath* query.

Of course, this approach gives us Narasimhan's last name too. If we want to exclude it, we can add another condition to the predicate, as follows:

```
//namePart[@type='family' and
           string(.) = 'Narasimhan']/ancestor::mods/name/
   namePart[@type = 'family' and
               not(./text() = 'Narasimhan')]/text()
```

The single period for `self` is used again, this time to ask for the current node's child text node. The predicate also uses the `not()` function to exclude the element with Narasimhan's name.

An alternative approach is to reverse direction for only one level to the parent element (this would be a <*name*> element), then proceed to the parent's siblings, and then down one level to these siblings' <*namePart*> children. There are two sibling axes, `preceding-sibling` and `following-sibling` because one travels in the forward direction (`following-sibling`) to locate the siblings that come after the context element and the other locates sibling elements in the reverse direction that are found earlier in the hierarchy. Unfortunately, we cannot simply ask for all the siblings because that involves two directions. The following compound *XPath* expression looks in both directions:

```
//namePart[@type='family' and ./text()='Narasimhan']/
  parent::name/preceding-sibling::name/
  namePart[@type='family']/text()
  |
//namePart[@type='family' and ./text()='Narasimhan']/
  parent::name/following-sibling::name/
  namePart[@type='family']/text()
```

Notice that the two *XPath* expressions are combined via the logical operator, `|`. They are also very similar, with the difference merely being that one uses the `preceding-sibling` axis and the other uses `following-sibling`. This is another case where we can generate the queries with string manipulation in *R* such as

```
q = sprintf("//namePart[@type='family' and ./text()='%s']/
             parent::name/%s-sibling::name/
               namePart[@type='family']/text()", "Narasimhan",
            c("preceding", "following"))
```

We even added the family name as a parameter so that we can use the same code to generate queries for other authors. We can then pass this to getNodeSet() as `getNodeSet(doc, q)` to get back

```
[[1]]
Doss

[[2]]
Boik

attr(,"class")
[1] "XMLNodeSet"
```

Each of the two nodes in the result is actually an internal text node. We probably want just the text rather than the nodes themselves, unless we wanted to modify them. This is an example of where we should use xpathSApply() to extract the values of the text node with

```
xpathSApply(doc, q, xmlValue)

[1] "Doss" "Boik"
```

Rather than navigating down and then back up the hierarchy, it might be simplest to use a predicate to identify the different <*mods*> nodes of interest.

```
//mods[.//namePart[@type = 'family' and . = 'Narasimhan']]
```

Now that we have these nodes, we can fetch the `namePart[@type = 'family']` nodes again. We can do this in a separate step

```
mods = getNodeSet(doc, "//mods[.//namePart[@type = 'family'
                                and . = 'Narasimhan']]")
sapply(mods, function(x)
                xpathSApply(x, ".//namePart[@type='family']",
                            xmlValue))
```

Alternatively, we can do this in one step:

```
q = "//mods[.//namePart[@type = 'family' and
              . = 'Narasimhan']]//namePart[@type='family']"
xpathSApply(doc, q, xmlValue)
```

This loses the association between the authors by paper and just groups the authors together into a flat collection. These results do not exclude Narasimhan, but we can do that in *R* if we want.

3. Find the identifier key for all articles by Lumley
One approach uses nested predicates to select only those <*mods*> nodes that Lumley authored:

```
//mods[name/namePart[@type = 'family' and . = 'Lumley']]
     /identifier[@type = 'citekey']
```

We have placed predicates on two location steps. The predicate on the <*mods*> location step selects those <*mods*> elements that have a grandchild <*namePart*> element with Lumley as the text content. We can evaluate it and get the value of the <*citekey*> node with

```
xpathSApply(doc, "//mods[name/namePart[@type = 'family'
                                       and . = 'Lumley']
                         ]/identifier[@type = 'citekey']",
            xmlValue)
```

```
[1] "Lumley:2004:ACS"
```

Note that we can directly compare `.` (the **self** node) to the string `'Lumley'`. This is equivalent to calling `string(.)`, and since there is only one text child node of <*namePart*>, it is also equivalent to `string(text())` and `text()`.

We can also use a similar approach as we did in the other exercises of traversing back up the tree. To find the key for all articles written by Lumley, we can travel down the tree to the <*namePart*> node to find `Lumley`, reverse direction to go up to the <*mods*> ancestor, and reverse direction again to go to the child <*identifier*> that contains the citation key. The following *XPath* expression does exactly that:

```
//mods//namePart[text()='Lumley']/ancestor::mods/
  identifier[@type = 'citekey']
```

There are several alternatives to this *XPath* expression. One such alternative sets up location steps based on the number of levels to traverse rather than using a node-test to match the name <*mods*>. That is, we only go back up the tree two steps (from the <*namePart*> back to the <*mods*> via the <*name*> node), as follows:

```
//mods//namePart[text()='Lumley']/../../
  identifier[@type = 'citekey']
```

Here, we use the **parent** shortcut .. twice to go back up the tree. This is the same as `parent::*` but more convenient, especially given familiarity with navigating file systems. (Note that we use the asterisk to avoid having to specify an exact name for the parent axis.) We can use the location step `parent::name` and `parent::mods`, instead. Again, this is more specific and fails (giving an empty node-set) if the structure is not as we expect. This can be beneficial, but often we want the most expedient matching expression for a particular document we are working on immediately.

Finally, a note of caution about the `text()` function and whitespace. We have seen in Chapter 3 that whitespace can be a concern in matching text content. To remedy this, *XPath* provides the function `normalize-string()` to remove leading and trailing spaces and replace multiple consecutive occurrences of whitespace with a single space. We did not need to use this function in our example because the document had a consistent format with no extra whitespace in the text. However, a more general *XPath* expression might want to add this as a precaution in case the content is not what we expect.

4.8 Namespaces and *XPath* Queries

When we want to use *XPath* to query an *XML* document that uses namespaces, we need to do slightly more work. That is, we need to ensure that the namespaces in the *XPath* query and the document match. Recall that we use namespaces in *XML* documents to identify the vocabulary of a node or attribute name. These allow us to disambiguate when we use the same name for a node or attribute in a different sense within the same document. A namespace has a URI that uniquely identifies it, and a prefix, which is what we work with locally within our document. For example, we can use the node <*code*> to refer to *R* code. However, if we also want to refer to *C* code, we cannot use <*code*> for that too. Rather than, a priori, knowing about the conflict and using different names such as <*rcode*> and <*ccode*>, we use namespaces to differentiate between the two uses. For example, consider the following simple document:

```
<article xmlns="http://docbook.org/ns/docbook"
         xmlns:r="http://www.r-project.org"
         xmlns:c="http://www.C.org">
...
<section> <title>Random Numbers</title>
<para>...
<r:code><![CDATA[
 x <- rnorm(1000)
]]></r:code>
<c:code>
 for(int i = 0; i < n; i++)
    total += x[i];
</c:code>
</para>
</section>
</article>
```

Here we have defined two namespaces with their own URIs and local document-specific prefixes (`r` and `c`). These prefixes are used to qualify the `<code>` nodes, e.g., `<r:code>` and `<c:code>`. Rather than having to qualify the many *DocBook* nodes with a prefix, we made the *DocBook* namespace the default that applies to all unqualified nodes. That is, the unqualified nodes are in the default namespace associated with the URI `http://docbook.org/ns/docbook`.

XPath expressions also use namespaces to qualify node names. When we need to refer to a namespace in an *XPath* query, we need to define the namespace as a URI and prefix pair and then use that prefix locally within the *XPath* expression. The *XPath* query effectively replaces the prefix with the URI and only considers a match if the fully qualified name matches. In *R*, we specify the namespace mappings via the *namespaces* parameter for each of the *XPath* functions, i.e., getNodeSet(), xpathSApply() and xpathApply(). We give this a named character vector where the names are our own choice of prefix for the namespace, and the values are the URIs. These URIs must match the corresponding namespace URIs in the target document, if we are to match effectively. The prefixes, however, do not have to match at all as these are local to the *XPath* expression, just as the prefixes are local to the document. For example, to retrieve the `<r:code>` nodes in the above document we use

```
getNodeSet(doc, "//r:code",
           namespaces = c(r = "http://www.r-project.org"))
```

We did not have to use the same prefix—`r`—as that used in the target document. We can choose any prefix, but we do have to use that same prefix in both the *XPath* query and the *namespaces* argument to getNodeSet(). For example,

```
getNodeSet(doc, "//s:code",
           namespaces = c(s = "http://www.r-project.org"))
```

is equivalent to the previous command, but uses the prefix `s` rather than `r`.

We can, and sometimes must, use multiple namespaces within an *XPath* query. For example, suppose we wanted to get the `<code>` nodes for both *R* and *C* languages in our document. We can do this using a vector with multiple `prefix = URI` elements as follows:

```
getNodeSet(doc, "//s:code | //c:code",
           namespaces = c(c = "http://www.C.org",
                          s = "http://www.r-project.org"))
```

Perhaps the most common confusion arises when querying a document that has a default namespace (as opposed to no namespace). We have to tell *XPath* about that default namespace. Recall that the choice of default namespace and prefix (or lack thereof) is the choice of the document's author, i.e., local to the document. Similarly, the choice for the namespace prefix in our *XPath* query is local to us and not connected to the target document. Indeed, we want to be able to use the same *XPath* query across different instances of the same class of documents where some documents may have a default namespace and some may not but they will all use the same namespace definitions (i.e., URIs). Therefore, even if the document has a default namespace (i.e., with no explicit prefix), we have to explicitly identify and use that namespace in our *XPath* expression. For example, to query our document and find the `<r:code>` nodes in the first `<section>` node, we use the query

```
Namespaces = c(x = 'http://docbook.org/ns/docbook',
               r = 'http://www.r-project.org')
getNodeSet(doc, "//x:section[1]//r:code", namespaces = Namespaces)
```

(Note that it is a good idea to define namespaces in a character vector and reference this in calls to getNodeSet(). This avoids repeating them in different calls and having to change them in more than one place or making an error in typing them in more than one place.)

In our *XPath* query, we have to map the document's default namespace to an actual prefix, e.g., `x`, and then use that in our query to qualify the <*section*> node. If we did not introduce this explicit mapping to the default namespace, but instead used a query such as `//section//r:code`, we would get no matching nodes. This is because *XPath* is looking for a node named <*section*> with no namespace. It does not recognize the default namespace. Forgetting to deal with a default namespace is a common mistake that people make when using *XPath* initially.

Basically, to deal with namespaces, we really need to know about the document before we try to query it. However, we often want to query it to find out about its contents. The getNodeSet() function and related functions try to help us in these situations. Firstly, if the document has namespaces defined on the root node, then we can find these via the xmlNamespaceDefinitions() function. This returns the namespace definitions on a particular node. However, the *XPath* functions such as getNodeSet() query the namespace definitions on the target document's root node. We can get the individual namespace definitions in full form or in simplified form. The simplified form is sufficient for our needs here:

```
ns = xmlNamespaces(doc, simplify = TRUE)

"http://docbook.org/ns/docbook"
                              "r"
     "http://www.r-project.org"
                              "c"
             "http://www.C.org"

attr(,"class")
[1] "SimplifiedXMLNamespaceDefinitions"
[2] "XMLNamespaceDefinitions"
```

Here we see three namespace definitions. The prefixes are given in the names of the `ns` object's elements so we can find the default with

```
names(ns) == ""
```

Alternatively, we can use the function getDefaultNamespace() to find the default namespace, if there is one:

```
getDefaultNamespace(xmlRoot(doc))

"http://docbook.org/ns/docbook"
```

If there is no default namespace on our document's root node, we get an empty *vector*.

If there is a default namespace, we have to somehow express this in our *XPath* queries. We can have getNodeSet() (and the other *XPath* functions) assist us in this. If we do not specify an explicit definition for namespaces, but just a simple prefix, getNodeSet() will try to match that prefix to the target document's set of namespace prefixes. If it does not match, but there is a default namespace, getNodeSet() maps the prefix we specified to the URI of the default namespace. This allows us to use an arbitrary prefix in our *XPath* queries to identify the default namespace without having to know or specify its URI. For example, to find the <*para*> nodes in the first <*section*>, we can use the command

```
getNodeSet(doc, "//x:section[1]//x:para", namespaces = "x")
```

Here we just specify the prefix `"x"` rather than the `prefix = URI` pairing. Then getNodeSet() (and the other *XPath* *R* functions) maps this to the default namespace at the root node of the document. The result is equivalent to specifying the *namespaces* as `c(x = "http://docbook.org/ns/docbook")`. Note that this shortcut only works when the default namespace is defined on the root node of the document.

We can use the same shortcut for specifying the namespaces by their prefix alone and having getNodeSet() match these to the namespaces defined in the root node of the document. For example, we can specify the *R* namespace with the same prefix as used in the document as

```
getNodeSet(doc, "//r:code", "r")
```

Again, getNodeSet() looks up the namespaces on the root node and matches the string `"r"` to the namespace there and uses that. Since the `"r"` prefix is not the default namespace but is defined on the root node, we can actually reduce this to

```
getNodeSet(doc, "//r:code")
```

as getNodeSet() uses the namespace definitions from the root node if none are specified via the *namespaces* parameter.

If we want to use the namespace prefixes from the document but also need to specify the default namespace, we can do this by using a prefix that does not appear on the root node. We can use the same `"x"` as we did before and also use the `"r"` prefix as in

```
getNodeSet(doc, "//x:section[1]//r:code", c("x", "r"))
```

This is equivalent to specifying the *namespaces* argument as

```
c(x = 'http://docbook.org/ns/docbook',
  r = 'http://www.r-project.org')
```

When we just specify the prefixes as `c("x", "r")`, getNodeSet() retrieves the namespace definitions from the root node of the document and matches the `r` prefix. It finds no match for the `x` prefix but determines that there is a default namespace on the root node. It then maps the prefix `x` to this default namespace URI and so the two approaches are equivalent.

The implicit matching of namespace prefixes given to getNodeSet() (or xpathApply() and xpathSApply()) is convenient. It is useful for interactive queries that we do at the *R* prompt. However, as we can see, there is a lot underlying the matching and we may end up with surprises. We may end up with no matches or, worse, false matches because the implicit matching is not what we expect. Accordingly, when writing code that calls getNodeSet() and related functions, we strongly recommend that you explicitly specify the namespace prefix-URI mappings. Furthermore, we recommend that these mappings be defined in a centralized, nonlocal variable, e.g., once within a package, and that variable be used in the different *R* expressions that use *XPath* queries.

4.9 *XInclude* and *XPointer*

XInclude (http://www.w3.org/TR/xinclude/) and *XPointer* [9] are two additional technologies and standards related to *XML* documents. These provide a rich way to insert the contents of part of a document within other documents. As a result, they facilitate modularity, reduce redundancy, and

enable creation of complex descriptions and data structures within *XML*. We give examples of how *XInclude* and *XPointer* support these features.

Example 4-10 Building a Data Set from Fragments with XInclude

Let us say we are interested in the exchange rate for a particular currency, say the Australian dollar, relative to the euro. Each month, we extract the daily rates from the European Central Bank's Website and store them in files named, e.g., `euroAUD2008-04.xml` (for April 2008) in the following SDMX-ML format.

```
<DataSet>
 <Group CURRENCY="AUD" CURRENCY_DENOM="EUR"
   EXR_TYPE="SP00" EXR_SUFFIX="A" DECIMALS="4"
   UNIT="AUD" UNIT_MULT="0"
   TITLE_COMPL="ECB reference exchange rate,
     Australian dollar/Euro, 2:15 pm (C.E.T.)" />
 <Series FREQ="D" CURRENCY="AUD" CURRENCY_DENOM="EUR"
   EXR_TYPE="SP00" EXR_SUFFIX="A" TIME_FORMAT="P1D">
  <Obs TIME_PERIOD="2008-04-11" OBS_VALUE="1.7024" />
  <Obs TIME_PERIOD="2008-04-14" OBS_VALUE="1.7158" />
  <Obs TIME_PERIOD="2008-04-15" OBS_VALUE="1.7121" />
  <Obs TIME_PERIOD="2008-04-16" OBS_VALUE="1.7069" />
  <Obs TIME_PERIOD="2008-04-17" OBS_VALUE="1.6935" />
  <Obs TIME_PERIOD="2008-04-18" OBS_VALUE="1.6882" />
  <Obs TIME_PERIOD="2008-04-21" OBS_VALUE="1.6873" />
  <Obs TIME_PERIOD="2008-04-22" OBS_VALUE="1.6851" />
  <Obs TIME_PERIOD="2008-04-23" OBS_VALUE="1.6755" />
 </Series>
</DataSet>
```

This *XML* vocabulary is used by the European Central Bank (ECB) for disseminating time-series views for a currency on the Web. See http://www.ecb.int/stats/exchange/eurofxref/html/eurofxref-graph-aud.en.html for interactive time series plots and http://www.ecb.int/stats/exchange/eurofxref/html/aud.xml for the *XML*.

Briefly, the <*DataSet*> element contains the <*Group*> element, which contains information about exchange rates. This information includes which currencies are being compared via the attributes *CURRENCY* and *CURRENCY_DENOM*, respectively. Within <*DataSet*>, the <*Series*> element also acts as a grouping element. In addition to currency identifiers, it contains information about the frequency of observation. In this example, the value `D` in the *FREQ* attribute indicates "daily" measurements. The <*Series*> element contains observations for the days, each in an <*Obs*> element. The <*Obs*> element provides the time period and the observed rate for that time period via its attributes.

Storing the data in monthly files allows flexibility in how the data can be combined for analysis and reports. For example, to combine the data for April and May, we can create a new *XML* file that includes the <*Obs*> elements from both files. Rather than make copies of these data and updating them should the original data ever change, our document contains instructions to merge the <*Obs*> fragments from the monthly files into our document. Below is the *XML* document shell with the instructions to do this dynamic inclusion:

```
<DataSet xmlns:xi="http://www.w3.org/2001/XInclude">
 <Group CURRENCY="AUD" CURRENCY_DENOM="EUR"
```

```
    EXR_TYPE="SP00" EXR_SUFFIX="A" DECIMALS="4"
    UNIT="AUD" UNIT_MULT="0"
    TITLE_COMPL="ECB reference exchange rate,
      Australian dollar/Euro, 2:15 pm (C.E.T.)" />
  <Series FREQ="D" CURRENCY="AUD" CURRENCY_DENOM="EUR"
    EXR_TYPE="SP00" EXR_SUFFIX="A" TIME_FORMAT="P1D">
   <xi:include href="euroAUD04.xml"
     xpointer="xpointer(/DataSet/Series/Obs)"/>
   <xi:include href="euroAUD05.xml#xpointer(//Obs)"/>
  </Series>
</DataSet>
```

These two <*xi:include*> tags locate the <*Obs*> fragments. Each has an *href* attribute to locate the external resource, e.g., `euroAUD2008-04.xml`, and an `xpointer()` function to identify the specific fragments within the external document. The *XPointer* mechanism uses *XPath* expressions to identify node sets to include/copy into the document.

The two <*xi:include*> tags differ in how they identify their document fragments. The first uses a separate *xpointer* attribute to supply the *XPath* expression to identify the <*Obs*> nodes we want. In the second <*xi:include*> node, the *XPointer* expression appears as part of the *href* attribute, separated from the file name with a hash mark. The *XPath* expressions are intentionally slightly different, but for this document, the two expressions result in the same node-set.

When we parse this document, xmlParse() can process the *XInclude* directives and we will only see the resulting <*Obs*> nodes in the document. Alternatively, we can have the parser leave the *XInclude* directives, but this is much less common. With this modularity, one month's data can be used in several documents and any updates or changes would be made to the single source `euroAUD2008-04.xml` and these changes would be reflected in the other documents.

***XInclude* Structure**

XInclude is a technology for merging *XML* fragments into a document. The *XInclude* namespace contains two elements: <*include*> and <*fallback*>. The <*fallback*> element is a child of <*include*> and provides a mechanism for recovering from the inclusion of a missing resource and providing content to use when the content to be included is not available. This might occur when the link is incorrect or if the associated Web server is temporarily unavailable.

The <*include*> element has the following attributes for specifying the location and type of the resource to be included.

href Provides a URI reference for the location of the resource to include. When *href* is missing, the reference is to the same document and the *xpointer* attribute must be present and identify an element within the same document.

xpointer Identifies a portion of the resource to include. This attribute is optional; when omitted, the entire resource is included. The attribute value is an *XPointer* expression that contains an *XPath* expression to identify the elements of interest in the target document. This *XPointer* expression can define namespaces, as needed in the *XPath* expression.

parse Indicates whether to include the content/node-set as parsed *XML* (`parse="xml"`) or as text (`parse="text"`) directly and verbatim with no interpretation by the *XML* parser. The default value of this optional attribute is `"xml"`.

encoding When `parse="text"`, the *encoding* attribute specifies the encoding of the resource.

XPointer offers several formats for identifying fragments in a file. In the first example, we used the function, `xpointer()` to provide an *XPath* expression to locate the fragments. This function can be provided as part of the *href* attribute value,

```
href = "fileURI#xpointer(xpathExpression)"
```

or via the separate *xpointer* attribute as

```
href = "fileURI"  xpointer = "xpointer(xpathExpression)"
```

If the *XPath* expression involves namespaces, then these need to be defined in the `xmlns()` function within the *XPointer* expression, e.g.,

```
xpointer = "xmlns(x = namespaceURI)
            xpointer(xpath_expressionWithNamespaces)"
```

The namespaces are not read from the host document as they refer to the external/target document. We show how to use namespaces in the next example, but we first mention a few alternative ways to specify which fragments to include.

If a fragment/node has a unique identifier specified in its *id* attribute, we can include it with

```
href = "fileURI#identifier"
```

This is shorthand for

```
href = "fileURI#xpointer(id('identifier'))"
```

An element can also be specified by position in the document. For example, `element(/1/2)` locates the second child of the root element, and

```
element(/1/identifier/3)
```

locates the third child of the node with the *id* attribute given by the identifier within the root node. Of course, these specifications are not robust if the document is modified. If we insert an additional node in the hierarchy, using the position of a node can produce unexpected results.

We mentioned previously that *XML* can be used to represent arbitrary data structures. For example, with *XML* we can represent a graph of vertices and edges such as a computer network or a social network, where one *XML* element refers to other elements within the document. In one section of the document, we can define the vertices of the graph, giving each *XML* element a unique identifier via an *id* attribute. Then the network structure of the graph can be described by specifying the start and end vertex for each edge (via the vertices unique identifiers). We can do this by defining an appropriate *XML* vocabulary or via *XInclude* and *XPointer*.

Similarly, if we wanted to represent a complex data structure in C that had sub-components with shared access to instances of other data structures, then we would use the same approach of identifying the shared data structures with unique identifiers and specifying the higher-level data structures with fields that identified whichever lower-level shared data structure was being used. A concrete example will help clarify these concepts.

Example 4-11 Using XInclude to Create R Data Structures with Shared Sub-components

Suppose we have an *R* vector object that is assigned to the variable `x`. After that, it is assigned to the variable `y`. *R* does not make an explicit copy of the vector to store in `y`. Rather, it creates a small "header" object for `y` which points to the data part shared with `x`. Only when either of `x` or `y` is changed is a separate copy made for each variable. If we have code such as

```
x = 1:5
y = x
```

and then want to store both `x` and `y` as *XML*, we can do this as

```
<r:integer id="ID01" length="5">1 2 3 4 5</r:integer>
<r:var name="x" ref="ID01"/>
<r:var name="y" ref="ID01"/>
```

If we deserialize this *XML* content back into *R*, the effect that `y` is merely a virtual or pending copy of `x` can be maintained. Unlike the previous example, one part of this document has been referenced in multiple other places in the document, not actually included, and we have used the name attribute in a dialect-specific manner. Alternatively, we can use *XPointer* expressions to do this more explicitly, i.e., to actually insert a copy of a node from one part of the document into another part. For example, we can write the *XML* serialization of our `x` and `y` example as

```
<r:data xmlns:r="http://www.r-project.org"
        xmlns:xi="http://www.w3.org/2001/XInclude">
<r:var name="x">
 <r:integer id="ID01" length="5">
  1 2 3 4 5
 </r:integer>
</r:var>
<r:var name="y">
 <xi:include
  xpointer="xmlns(r=http://www.r-project.org)
    xpointer(//r:var[@name="x"]/r:integer"/>
</r:var>
</r:data>
```

In this *XPointer* expression, we use `xmlns()` to specify the namespace of the *XPath* expression. As in *XPath*, the namespace prefix used in `xpointer()` need not match the namespace prefix in the document. However, using the same prefix can help clarify which nodes are being referenced.

In summary, *XInclude* and *XPointer* provide instructions to the processing tools (e.g., the *XML* parser) to explicitly make a copy of nodes in the same or another document and insert them into the current document. This allows us to avoid repetition, and to have a single, central source that can be updated while all other documents dynamically contain this modified content when they are parsed and processed. The *XML* parser handles the inclusion for us and we can disable this when we want to directly with the *XInclude* nodes.

4.10 Summary of Functions for Applying *XPath* Expressions to *XML* Documents

The following *XPath* functions in *R* work on "internal" documents and nodes, i.e. those parsed and returned by xmlParse() and htmlParse() and nodes and documents created with newXMLNode() and newXMLDoc().

getNodeSet() Evaluate an *XPath* query on a given document or subtree. It returns an *XPath* node-set as a list of class `XMLNodeSet`. Nodes are returned as references to the nodes in the tree. This

includes text nodes. This allows one to then modify them. Attributes are returned by value, i.e., named character vectors.

xpathApply(), xpathSApply() These functions are generalized versions of getNodeSet() that allows us to specify a function to apply to each of the matching nodes. The following are equivalent:

```
xpathApply(doc, xpathQuery, fun, ...)
lapply(getNodeSet(doc, xpathQuery), fun, ...)
```

The xpathSApply() function attempts to simplify the return value to a vector, analogous to sapply() and lapply(). If the nodes involved in the *XPath* query have *XML* namespaces, we must identify the namespace in the call to getNodeSet() or xpathApply() via the *namespaces* parameter, e.g.,

```
getNodeSet(doc, '//db:section[1]//r:code',
           namespaces = c(r = 'http://www.r-project.org',
                          db = 'http://www.docbook.org'))
```

We use the prefixes in the *XPath* query to specify which namespace the elements must match. The URIs need to match those in the document; the prefixes can be arbitrary names although we often use the same prefixes as in the document. If we do, then we need not supply the URI because the functions can match these for us and make specifying the namespaces simpler. We must use a prefix and namespace definition even if there is a default namespace defined for the document and nodes.

getDefaultNamespace() Retrieve the default namespace for the top-level node in the document.

xmlNamespaceDefinitions() Retrieve the namespace definitions declared in the node or root node of an *XML* document. The *recursive* argument indicates whether or not to get the namespaces definitions for all child nodes.

4.11 Further Reading

There are entire books and many chapters in other books that cover *XPath*. For readers needing more details or an alternative explanation, we suggest reading the first four chapters of [9]. In addition, see Chapters 10–12 of [6] for more on *XPointer* and *XInclude*, respectively.

References

[1] Anders Berglund. Extensible Stylesheet Language (*XSL*) Version 1.1. Worldwide Web Consortium, 2006. http://www.w3.org/TR/xsl.

[2] Michael Brundage. *XQuery: The XML Query Language*. Addison Wesley, Boston, MA, 2004.

[3] James Clark. *XSL* transformations (*XSLT*). Worldwide Web Consortium, 1999. http://www.w3.org/TR/xslt.

[4] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[5] FLOWR Foundation. *Zorba*: The *XQuery* processor. http://www.zorba-xquery.com, 2012.

[6] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[7] Library of Congress. MODS: Metadata Object Description Schema. http://www.loc.gov/standards/mods/mods.xsd, 2010.
[8] National Center for Integrative Biomedical Informatics. Michigan molecular interactions. http:/mimi.ncibi.org, 2010.
[9] John Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly Media, Inc., Sebastopol, CA, 2002.
[10] Duncan Temple Lang. RXQuery: Bi-directional interface to an *XQuery* engine. http://www.omegahat.org/RXQuery, 2011. *R* package version 0.3-0.
[11] Duncan Temple Lang. Sxslt: *R* extension for `liblibxslt`. http://www.omegahat.org/Sxslt, 2011. *R* package version 0.91-1.
[12] Jenni Tennison. *XSLT and XPath On the Edge*. M & T Books, New York, NY, 2001.
[13] Doug Tidwell. *XSLT*. O'Reilly Media, Inc., Sebastopol, CA, 2008.
[14] W3Schools, Inc. *XPath* tutorial. http://www.w3schools.com/XPath/default.asp, 2011.
[15] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., Sebastopol, CA, 2007.

Chapter 5
Strategies for Extracting Data from *HTML* and *XML* Content

Abstract In this chapter, we compare different approaches to parsing *XML* and *HTML* documents and extracting data from these documents into *R*. We illustrate these with comprehensive, real-world examples that illustrate *XPath* and *R* functions for processing *XML* documents. We also introduce event-driven parsing where we use a collection of *R* functions to respond to events in the *XML* parser. These work for both tree-based (*DOM*) parsing and *SAX* parsing where we avoid building the tree. At the end of the chapter, the reader should have a good understanding of the various different strategies that can be used in *R* to parse *XML* documents and extract content.

5.1 Introduction

In Chapter 2, we saw the essential structure, grammar, and elements of *XML*. In Chapter 3 we presented a collection of *R* functions that allow us to work with *XML* documents, trees, and individual nodes. In Chapter 4, we saw the *XPath* query language and additional *R* functions to search for nodes within an *XML* document. Both sets of *R* functions worked on the *XML* document as a tree. This is called the Document Object Model (*DOM*). These *DOM*-based approaches assume we can read the entire *XML* document into memory and then process its contents. For very large documents, this is not practical. Instead, we have to process the *XML* content as we read it and convert those contents into *R* objects "on the fly." This involves a different, low-level processing model named *SAX*—Simple API for *XML*—that uses handler functions to respond to events in the *XML* stream. We discuss this and also a hybrid approach that involves converting parts of a stream of low-level *XML* tokens into nodes and then processing these using *DOM*-based approaches.

In this chapter, we aim to explore different strategies for working with *XML* documents and to give advice about how to tackle problems involving reading data from *XML* into *R*. We start with *DOM*-based approaches and end the chapter by presenting the *SAX* model. We explore these different approaches via case studies that illustrate and contrast different strategies for accessing the data. When discussing *DOM* parsing, we will introduce an approach that traverses the tree and extracts data using handler functions. This leads naturally to the lower-level *SAX* model.

We will show how to scrape data from *HTML* documents. Sometimes this is as simple as calling readHTMLTable(), readHTMLList(), or getHTMLLinks(). However, frequently the data in the *HTML* document are less regular or structured and we have to develop patterns to identify the relevant nodes within the *HTML* document. This often involves developing *XPath* expressions that exploit both the hierarchy of *HTML* elements and also *CSS* attributes such as *class* and *style*.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_5,

We make heavy use of *XPath* when working with *XML* documents. This typically involves finding the nodes of interest with getNodeSet() and then using the *R* functions to manipulate the resulting collection of nodes. These are functions such as xmlName(), xmlGetAttr(), xmlValue(), and xmlSApply(), subsetting the node to access child nodes, and so on. To work with *XML* in *R*, or any language, it is important to master these two skills—*XPath* and node manipulation. One of the purposes of this chapter is to take somewhat complex examples and show how we combine these two sets of tools to extract data. We aim to explore common general strategies and illustrate some of their complexities such as extracting variables individually from a document rather than collectively from each node. We also suggest that in some cases, it is convenient to modify the *DOM* (not the original *XML* document) in memory to facilitate data extraction. Changing the tree (e.g., adding new nodes or attributes, changing the hierarchy, and the relationships between the nodes) and the *R* functions to do this are covered much more extensively in Chapter 6.

The *SAX* approach is quite different from the *DOM* approach as it does not try to build the *XML* tree. Instead, it passes small pieces of the document to "handler" functions that the caller provides. These functions are responsible for determining how to combine the pieces of data from the *XML* stream into *R* objects. These functions are specific to the structure of the *XML* so they are application-specific. *SAX* parsing is necessary for very large *XML* documents that would not easily fit in memory. The approach is more low-level and involves a different mindset and strategy from *DOM* parsing. Branches allow us to combine *SAX* and *DOM* approaches for small parts of the entire hierarchy.

5.2 Using High-level Functions to Read *XML* Content

In Chapter 1, we saw some high-level functions for reading *XML* content into *R* objects. These were the functions readHTMLTable(), xmlToDataFrame(), and xmlToList(). We introduced these to show how to do common things easily in *R* with a single call to a function. We illustrated how they can be used in simple cases, but we did not discuss some of their more advanced features. In this section, we explore these functions in a little more detail and show some examples of how they can be used in different ways. Even when these functions are not suitable to read specific *XML* documents, it may help to understand both how they work and their design.

We also introduce the *HTML* focused functions: readHTMLList(), getHTMLLinks() and getHTMLExternalFiles(). Like readHTMLTable(), these are useful when all we want is very specific subsets of the *HTML* content or the hyperlinks in the document.

We also introduce some additional high-level functions for reading documents from specific *XML* vocabularies. One of these is readKeyValueDB() which can read (nonbinary) property list (or plist) documents that are used, e.g., in the Mac operating system OS X. The other is readSolrDoc() which can read Solr files that are used in the Lucene text search system. Both vocabularies represent data generally as scalars of different types, arrays of scalars, and named-element structures.

5.2.1 Simple HTML Access

If we are accessing *HTML* content, we may be lucky and the data of interest are either in a list or a table, or we may want the hyperlinks or the referenced files such as images, *JavaScript* or *CSS* files in the document. In these cases, we can use existing high-level functions in the XML package [4] to access this content.

All of these high-level functions allow the caller to specify the *HTML* document as

- the name of a local file,
- a *URL*,
- the *HTML* content itself as a string, or
- as an already parsed *HTML* document, i.e., an *R* object of class `HTMLInternalDocument`.

When we develop new *R* functions to process *XML* or *HTML* documents, we should emulate this to give the greatest flexibility. The last two of these are important. If, for example, the *URL* uses *HTTPS*, i.e., secure *HTTP*, then the htmlParse() function cannot retrieve the document. We would need a different approach than specifying the name of the *URL*, such as using getURLContent() from the RCurl package [5] to get the content as a string. At other times, we may want to pass the parsed document to the function. This arises for several reasons. We may want to parse it once and use the parsed document in several different computations without having to parse it again. We also may want to modify the document hierarchy and content before passing it to our function. Another situation is that we may have actually created the *HTML* document in *R* as an in-memory tree/*DOM* and do not want to serialize it unnecessarily.

We can often accept the document in any one of these four formats using the simple idiom.

```
function(doc, ...) {

  if(is.character(doc))
     doc = htmlParse(doc)

  ...
}
```

This will handle the case where the caller provides the document in any of the first three forms in the list above. The caller can identify the string as *HTML* content and not the name of a file or *URL* by passing the string as `I(htmlContent)`, i.e., using the I() function to make the string of class `AsIs`.

Sometimes our function will need to know the *URL* or file name of the original document, e.g., to expand *URL*s found within the document that are relative to the base document. When we reassign the parsed document to our variable `doc`, we appear to lose the original string. We can, however, retrieve it using `docName(doc)`. If the original value of *doc* was the *HTML* content, then there would be no name and docName() would return `NA`.

S4 Methods

Another good design style for our functions is to use *S4* methods to define methods for the different classes of inputs. We can, and do, define a generic function for our function and then define methods for each of the four different types listed above. We go further to define a method where the "document" is actually an *XML*/*HTML* node. For example, we have a method for readHTMLTable() that processes a <*table*> node. All of the other methods are written so that they end up calling this method.

getHTMLLinks()

We can find all of the links in an *HTML* document with the function getHTMLLinks(). This does a simple *XPath* query to find all *href* attributes on <*a*> nodes in the document. We also can specify our own *XPath* expression and limit the matches by, for example, applying a filter to the *href* attributes. For example, we can find all the external links to `omegahat.org` with

```
getHTMLLinks(doc,
             xpQuery = "//a/@href[contains(., 'www.omegahat.org')]")
```

The *xpQuery* parameter also allows us to retarget the function for non-*HTML* documents such as *DocBook* documents that use different *XML* elements for specifying links.

The targets of the links include remote *URL*s, local files with an absolute path, or local files with a relative path. It is often desirable to expand the local files to be *URL*s relative to the document we are reading. We can do this using `relative = TRUE`. If for some reason we want to specify a different base *URL* for the document or the function cannot determine it for itself (e.g., we pass it *HTML* content), then we can specify a *URL* via the *baseURL* parameter.

getHTMLExternalFiles()

When we are publishing an *HTML* document on a remote Web site, we have to upload not only the *HTML* document but also the external files that it references. These are (non-inlined) images, *JavaScript* code, and *CSS* files. We need to know the names of these so that we transfer them all. Similarly, when we download an *HTML* document, we may also want to download those related files. We are also interested in understanding Web sites and the network or graph defined by which documents reference other documents. We can define this network for hyperlinks or for file inclusion. Regardless of our intended use, we can use the getHTMLExternalFiles() function to get the names of these. This takes the document (in any of the forms above) and returns a character vector giving the referenced external files. For example, if the value of `u` is `http://www.omegahat.org/SVGAnnotation/tests/examples.html`, then

```
getHTMLExternalFiles(u)
```

```
 [1] "http://www.omegahat.org/OmegaTech.css"
 [2] "../tests/pairs_link.svg"
 [3] "../tests/axes_tips.svg"
 [4] "../tests/axes_hrefs.svg"
 ...
```

Like getHTMLLinks(), we can use the *relative* parameter to expand the local file references in the document relative to the base *URL* of the document, or specify our own. Using `relative = TRUE` in the example above, we get

```
[1] "http://www.omegahat.org/OmegaTech.css"
[2] "http://www.omegahat.org/.../tests/pairs_link.svg"
...
```

We can also change the *XPath* query that the function uses to find the external document references via the function's *xpQuery* parameter.

In some circumstances, we may want to not get the names or *URL*s of the external files, but instead change them in the document. For example, we may want to change relative paths or point to a different server. To do this, we want the *XML* nodes, not the value of the *href* or *src* attributes. We can then change the values of the attributes within the node and write the document back to a file. We can do this using the *asNodes* parameter of getHTMLExternalFiles(). This is another common idiom for functions that extract information from *XML* documents. Specifically, while we often get the text values, we sometimes want the corresponding nodes so that we can change them. It is good to write functions that allow the caller to do either when the content is from individual nodes rather than a collection that defines a unit, such as a list or a table.

5.2.2 Extracting Data from HTML Tables

The first function we looked at in Chapter 1 was readHTMLTable(). This is highly specific to *HTML* content. It recognizes *HTML* elements such as `<table>`, `<tr>`, `<th>` and `<td>`. There is an obvious mapping from an *HTML* table to an *R* matrix or data frame. The readHTMLTable() function extracts the values in the cells and arranges them into rows and columns of a two dimensional data structure in *R*. The function is intended to resemble read.table() and read.csv() and to make acquiring data simple. As with read.table(), there are many arguments we can specify to help control how the values are interpreted.

One of the important differences, or extensions, from read.table() is that readHTMLTable() can read multiple separate tables in a single document and return them as separate data frames. Often we are interested in a single table in a page. We can specify which table to process via the *which* parameter of readHTMLTable(). If we want more than one table, we can specify their positions or indices with a vector, e.g.,

```
readHTMLTable(doc, which = c(3, 5, 9))
```

Of course, we have to know which table(s) we want. We can do this by retrieving them all, examining the results, and then determining which we want in future calls. This can be useful as we can then pass additional arguments to control how that single table is processed, e.g., indicating if the first row contains the column headers or not, providing types/classes for the different columns, and so on.

Instead of specifying which table(s) we want by their index, we can identify them based on the *HTML* content. For example, we may know that the `<caption>` node with the table contains the string `OECD`, or we may know the first cell in table contains the text `Year`. We can find the corresponding `<table>` node in the document using an *XPath* expression. We can then pass the resulting `<table>` node directly to readHTMLTable(). We can find the nodes for our two examples with

```
getNodeSet(doc, "//table[caption[contains(., 'OECD')]]")
getNodeSet(doc, "//table[tr[1]/th[1][contains(., 'Year')]]")
```

By writing the function as a collection of methods, we can allow flexible entry points for the callers.

Unlike other tabular data such as CSV and TSV files, *HTML* tables are intended to be displayed to humans on a Web page with different colors, fonts, images, etc. The presentation information makes it hard to find the actual data in the *HTML* source, but it also often changes the actual text of the data itself. Specifically, numbers that are percentages are often displayed as 90%, rather than as the number .9. Similarly, currency values are often displayed as $978,000, i.e., with the currency symbol ($) and with a separator for thousands, millions, etc. We can postprocess these columns in our data frame, removing the %, $ and , in the text, converting to numbers and dividing percentages by 100. However, we can also do this directly in readHTMLTable() using the *colClasses* parameter.

Example 5-1 Extracting and Formatting Information from a Wikipedia Table on US Public Debt

The table at http://en.wikipedia.org/wiki/History_of_the_United_States_public_debt provides the US national debt. The first three rows are shown in Figure 5.1. This is the third table in the page. We can see in the table that several of the columns have $ signs and commas in the numbers, and two columns are percentages with a trailing %. We can read this table into *R* and convert the columns at the same time with

```
d = readHTMLTable(theURL, which = 3,
                  colClasses = c("integer", "Currency", "Currency",
                                 "Percent", "Currency", "Currency",
```

```
                                "Percent", "Currency", "Currency",
                                "Percent", "numeric"))
```

We are specifying the classes `Currency` and `Percent` for the potentially troublesome columns. The `readHTMLTable` function attempts to convert the vectors to those types using *S4* coercion. The XML package defines these `Currency` and `Percent` classes. It also defines `FormattedInteger` and `FormattedNumeric` which are for the different numbers that just have comma separators, i.e., no currency. We can use these existing classes in the *colClasses* vector. We can also define new *S4* classes and coercion methods (via setAs()) from a *character* vector to that class. We can then use these new class names in *colClasses* and readHTMLTable() will convert the column to that target type.

Fiscal Year	Federal Spending			Federal Debt			Gross Domestic Product			Inflation Adjustor[51]
	Billions[52]	Adjusted[53]	Increase	Billions[54]	Adjusted[55]	Percentage Increase	Billions[56]	Adjusted[57]	Increase	
1977	$409	$1,040		$706	$1,795		$1,974	$5,019		0.39
1978	$459	$1,093	5.1%	$776	$1,850	3.1%	$2,217	$5,285	5.3%	0.42
1979	$504	$1,107	1.3%	$829	$1,821	-1.5%	$2,501	$5,494	4.0%	0.46

Figure 5.1: Wikipedia Table of US National Debt. These are the first three rows of a national debt table at `http://en.wikipedia.org/wiki/History_of_the_United_States_public_debt`. Of interest to us are the columns that are formatted as US dollars and the percentages that have a trailing % character. We can read and convert these with the readHTMLTable() function using the *colClasses* argument and specifying target types of `Currency` and `Percent`.

5.2.2.1 Extracting Other Information from *HTML* Table Cells

The readHTMLTable() function allows us to extract different information from cells in the table by specifying our own function to process the cell node.

*Example 5-2 Extracting Hyperlinks to **KMZ** Files from Attributes in HTML Table Cells*

In this example, we have an *HTML* document that presents a table of earthquake-related data. The cells in the table in the page `http://earthquake.usgs.gov/earthquakes/eqarchives/epic/kml/` contain hyperlinks to data about earthquakes for that year (row) and magnitude (column). A cell looks something like

```
<td>
 <a href="2011_Earthquakes_Mag9.kmz">9.0-9.9</a>
</td>
```

In this case, the data of interest is the value of the *href* attribute. We want to collect these URLs so we can download all the data. To do this, we can again use readHTMLTable(). However, we do not want the text of the cell, but the link in the attribute. To get these, we can provide a function as the value of *elFun* in the call to readHTMLTable(). This will be called with the <*td*> (table data) or <*th*> (table header) node. Our function can check if the node is named `td` and if it has an <*a*> child node. If so,

we return the value of the *href* attribute. If not, we return an empty character vector. We can define our function as

```
getCellLink = function(node)
{
   if(xmlName(node) == "td" && !is.null(node[["a"]]))
     xmlGetAttr(node[["a"]], "href", character())
   else
     character()
}
```

Note that we test the name is `td` by using the xmlName() function. This returns the name of the node, as we might expect, so we can pass this to readHTMLTable() as

```
links = readHTMLTable(kmlUrl, which = 1, elFun = getCellLink,
                      stringsAsFactors = FALSE)
```

where `kmlUrl` contains the `eqarchives` *URL* shown above. Now, `links` is a data frame containing all the links so we can just unlist them to get a character vector with the names of the 391 data files:

```
unique(unlist(links))
```

The purpose of this example is to show how we can extract general information from each cell, and not just its displayed value. We can also use getHTMLLinks() to get the links in the entire document, or use *XPath* to find the `<table>` node and then get the links within that, e.g.,

```
doc = htmlParse(kmlUrl)
tb = getNodeSet(doc, "//table[1]")
getHTMLLinks(tb[[1]])
```

5.2.3 XML Property List Documents

Property list files are used on Mac OS X to store data. On a Mac, there may be upwards of 25,000 property list files ranging in purpose from storing the current state of windows in an application, to the history list for a Web browser, to the preferences for a user's account, to printer configurations, to a description of the applications and bundles, even for *R* itself. While property lists are used mainly on the Mac, there is nothing about these documents that makes them specific to Mac. Indeed, Major League Baseball (MLB) also uses plist documents to describe aspects of each game; see http://gdx.mlb.com/components/copyright.txt.

Some of these files are stored as *XML* and we can read them directly. More recently, property list files are stored in a binary format. We can convert these to *XML* using the *plutil* tool, e.g.,

```
plutil -convert xml1 -o theFile.xml theFile.plist
```

We can even convert them to the *JSON* format.

A property list in *XML* format has a root node `<plist>` which has any number of children that are `<string>`, `<real>`, `<integer>`, `<true>`, `<false>`, `<date>`, `<array>`, `<dict>`, or `<data>`. The first six of these are used to represent scalar values of the corresponding type. An `<array>` is like a vector or a list in *R* without names, while a `<dict>` element corresponds to a

vector or list that has names for the elements. A <dict> element is made up of <key> and value nodes, e.g.,

```
<plist>
 <dict>
  <key>CFBundleTypeExtensions</key>
  <array>
   <string>Rd</string>
   <string>rd</string>
  </array>
  <key>CFBundleTypeName</key>
  <string>Rd Documentation File</string>
 </dict>
</plist>
```

There are two keys here, each followed by one of the possible value node elements we just listed. In this case, these are <array> and <string>. The <key> node contains the name of the element that follows.

If we want to read a property list into *R*, we can parse the *XML* document and then process the nodes. We convert each of the different scalar nodes into the corresponding *R* value, i.e., a vector of length 1. We convert an <array> node into a *list* as the elements may be of different types. Once we determine the class of all of the elements, we can collapse them to a vector if they are of compatible types. Similarly, we convert a <dict> node into a named vector or list in the same way. The only difference for an <array> and a <dict> is that we use the names from the <key> nodes.

The function readKeyValueDB() reads an *XML* document in this form. This is a generic function that has methods for the various different classes of inputs, i.e., name of a file or *URL*, the content itself, a parsed *XML* document, or an *XML* node. We can call this function as, for example,

```
readKeyValueDB(content)
```

The result is

```
$CFBundleTypeExtensions
string string
  "Rd"   "rd"

$CFBundleTypeName
[1] "Rd Documentation File"
```

The methods for readKeyValueDB() give us a great deal of control over how we provide the property list content to the function. We can use the method for an individual node or subtree when we extract the property list from within a larger document. We can also use it recursively when processing the subnodes within a plist tree. Indeed, this is the most sensible approach to converting the *XML* to *R*. We start at the root node of the plist tree and convert it to *R* by processing its subnodes, if there are any. The method that takes an XMLInternalNode object handles the nodes for scalars by creating the corresponding *R* value, e.g., <string> yields a *character* vector, <true> yields TRUE, and so on. When we process an <array> node, we process each of its child elements using this same method, i.e., calling it recursively. We do this with xmlSApply(). This simplifies the result to a vector when it can and otherwise returns a list.

A <dict> node is processed by processing each of the key elements with

```
kids = xmlChildren(node)
keyPos = seq(1, by = 2, length = xmlSize(node)/2)
structure(sapply(kids[ keyPos + 1], readKeyValueDB),
          names = sapply(kids[ keyPos ], xmlValue))
```

There is no point in using *XPath* to process the content as we have to process each node relative to its parent. We need to do this by traversing the tree, which we can do with the node manipulation functions. (We will see another approach we can use in Section 5.7.)

There are two purposes to this example. One is to show that if we want to process property lists, there exists a function to do it. The second is to show how we process a document of this nature and structure the code.

The concept underlying a property list is very common and general—representing scalars, arrays, and named/associative arrays. These occur in many contexts and are covered well by *XML* schema, *JSON*, and so on. For whatever reason, there are various different formats with the same basic ideas. One of these is a Solr document which is used in the Open Source Lucene text search engine. Below is a sample Solr document.

```
<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">90</int>
</lst>
<lst name="index">
 <!-- ANN: Provides info about the state of the index -->
 <int name="numDocs">17</int>
 <int name="maxDoc">17</int>
 <int name="numTerms">1044</int>
 <long name="version">1297337332283</long>
 <bool name="optimized">true</bool>
 <bool name="current">true</bool>
 <bool name="hasDeletions">false</bool>
 <str name="directory">
 <!-- ANN: The choice of Directory can sometimes effect
      performance.  Lucene tries to automatically pick the
      correct one, but ... -->
 org.apache.lucene.store.NIOFSDirectory:...@[{PATH}...
 lockFactory=org.apache.lucene.store....
 </str>
 <date name="lastModified">2011-02-10T11:29:03Z</date>
</lst>
```

This document is conceptually similar to a property list document. The <*bool*> nodes correspond to <*true*> and <*false*> but have the actual value as text within the node. The <*int*> and <*long*> nodes correspond to numbers; <*str*> correspond to a string or character vector with one element; <*lst*> is an <*array*>. Names can appear on any element, not via a separate <*key*> element, but via a *name* attribute. The function readSolrDoc() can read documents of this form. It too uses a similar approach of recursively processing nodes.

5.2.4 Helper Functions for Converting Nodes

We have briefly seen the functions xmlToList() and xmlToDataFrame(). These functions can be useful for converting simple *XML* documents into *R* objects. They work best when the documents are very shallow, i.e., have only two or three levels of nodes. For more complex documents with descendants of the root node being three or more generations/levels, the functions may not map to an appropriate *R* representation. These functions are, however, useful as tools that we can use when converting subnodes within a tree. As such, we can sometimes use them as part of a larger strategy for processing an entire *XML* document. They are helper functions. They are additional examples of where we want the functions to be flexible in allowing us to specify a document by name, as a parsed document, or as a collection of nodes, e.g., a simple list from a call such as `node[ "player" ]` or from an *XPath* query and so of class `XMLNodeList`.

In addition to these two functions, there are two other high-level helper functions. One is named xmlAttrsToDataFrame() which processes a collection of nodes and takes their attributes rather than the subnodes and turns them into a data frame. This can be useful when all of the content is in the attributes of the nodes of interest. Alternatively, we can use xmlAttrsToDataFrame() to process the attributes and use xmlToDataFrame() to process the subnodes and then combine the results. We will look at an example of using this function.

Example 5-3 Reading Baseball Player Information from Attributes into an R Data Frame

Some people (Americans mostly) love to explore statistics about the game of baseball. We compute batting averages, earned run averages (ERA), hits in different ball parks, percentages of hits against left-handed pitchers, and percentages of hits against right-handed pitchers named Ernest or Joe, pitching on a Tuesday—0 for 1! While the number of observations is small, the inference may be useful so it may be valuable to analyze games. The Major League Baseball (MLB) site http://gd2.mlb.com/components/game/mlb/ provides detailed information about each game played for many years.[1]

Each baseball game has information about the players involved in the game. The document looks something like

```
<game venue="Busch Stadium" date="October 28, 2011">
 <team type="away" id="TEX" name="Texas Rangers">
  <player id="119984" first="Darren" last="Oliver"
    num="28" boxname="Oliver" rl="L" position="P"
    status="A" avg=".000" hr="0" rbi="0" wins="0"... />
  <player id="134181" first="Adrian" last="Beltre"
     num="29" boxname="Beltre, A" rl="R" position="3B"
     status="A" bat_order="5" game_position="3B" .../>
  <coach position="manager" first="Ron"
     last="Washington" id="123965" num="38"/>.....
 </team>
 <team>
  <player..../>
  <player..../>
  <coach .../>
 </team>
 <umpires>
```

[1] Use of these data is governed by the license at http://gdx.mlb.com/components/copyright.txt.

```
  <umpire....>
    ...
 </umpires>
</game>
```

Each of the <player> nodes has various attributes giving the first and last name, jersey number, batting and fielding position, average, rbi (runs batted in), wins and so on.

We want to create a data frame with a row for each player and columns corresponding to the attributes. Not all attributes are present in each <player> node. This means we have to decide if we want just those that are present in all, or if we want to use all of the available attributes and have missing values for those nodes in which an attribute is not present. Indeed, we might be interested in just a subset of the attributes, e.g., *id*, *first*, *last*. In other circumstances, we might want to ignore particular attributes and include the rest. The xmlAttrsToDataFrame() allows us to chose which approach to follow.

The first step is to read the *XML* document, i.e.,

```
doc = xmlParse("players2.xml")
```

We do not want to process the <team> and <umpires> nodes. Instead, we want only the <player> nodes, ignoring the <coach> nodes within the <team> elements. To do this, we will use *XPath* to retrieve the <player> nodes:

```
playerNodes = getNodeSet(doc, "//player")
```

We can now pass this list of nodes to xmlAttrsToDataFrame()

```
players = xmlAttrsToDataFrame(playerNodes, stringsAsFactors = TRUE)
```

This results in a data frame with 50 rows and 16 variables. The columns include all of the attributes in any of the <player> nodes:

```
names(players)
```

```
 [1] "id"            "first"         "last"          "num"
 [5] "boxname"       "rl"            "position"      "status"
 [9] "bat_order"     "game_position" "avg"           "hr"
[13] "rbi"           "wins"          "losses"        "era"
```

We can request xmlAttrsToDataFrame() to use the names of the attributes that are common to all of the nodes. We can do this by either computing and specifying the names of those attributes ourselves, or using the function XML:::inAllRecords():

```
players = xmlAttrsToDataFrame(playerNodes,
                              attrs = XML:::inAllRecords,
                              stringsAsFactors = TRUE)
```

This gives a data frame with only 11 columns:

```
names(players)
```

```
[1] "id"       "first"    "last"     "num"      "boxname"  "rl"
[7] "position" "status"   "avg"      "hr"       "rbi"
```

Suppose we just want the id, first, and last variables. We can, of course, subset the data frame after we have created it. We can also specify that we want just those variables in the call to xmlAttrsToDataFrame() with

```
players = xmlAttrsToDataFrame(playerNodes, stringsAsFactors = TRUE,
                              c("id", "last", "first"))
```

We can either specify the variables as a character vector or by providing a function that dynamically processes the nodes and returns the collection of desired names. This allows us to determine which variables we want based on the contents of the nodes and attributes.

We can also use the *omit* parameter to discard some attributes.

There is also the function xmlToS4(). It is similar to xmlToList() in that it decomposes the contents of a node and puts them individually into an *R* list. The xmlToS4() function takes an *XML* node and the name of an *S4* class as arguments. It then attempts to match the slot names in the *S4* class to node and attribute names in the *XML* node, and it converts the matching *XML* elements to the type of the corresponding slot. This allows us to transform string values to numbers, logical values, and so on. This function also works recursively so subnodes with children can be transformed to other *R* classes using the class of the target slot. This can be useful when we define *S4* classes either manually or programmatically from an *XML* schema. (See Chapter 14.) We will use the same *XML* document from Example 5-3 (page 124)) to illustrate xmlToS4().

Example 5-4 Converting Player Information into S4 Objects

Suppose instead of a data frame, we wanted to represent the player information as a list with an element for each player. We want these elements to be *S4* objects of class `Player`. We define this class as

```
setClass("Player",
         representation(id = "character", first = "character",
                        last = "character", position = "character",
                        avg = "numeric", num = "integer"))
```

We can also add a prototype to provide default values for the slots.

We can now loop over the <*player*> nodes and convert each to this class:

```
players = lapply(playerNodes, xmlToS4, "Player")
```

The first element is

```
An object of class "Player"
Slot "id":
      id
"119984"

Slot "first":
   first
"Darren"

Slot "last":
    last
"Oliver"

Slot "position":
position
     "P"
```

```
Slot "avg":
[1] 0

Slot "num":
[1] 28
```

If we do not specify the name of the target class, then xmlToS4() uses the name of the *XML* node, e.g., player in this case. Therefore, if we had named our class `player`, rather than with a capital P, we would not have needed to specify the name of the class in our calls.

Instead of passing the name of the target class, we can pass an actual instance of the object. This is useful when we are working with subclasses and want to fill in the slots of the parent or base class(es).

5.3 Examples of Scraping Content from *HTML* Pages

As we mentioned, we commonly find Web pages with interesting information that we would like to treat as data. We can use the functions readHTMLTable(), readHTMLList(), or getHTMLLinks() to get data that are in tables, lists or the hyperlinks of a page. Often, however, the data of interest are not in these *HTML* elements. Instead, they may be within `<div>` nodes used to format and present the content in a particular way. Similarly, even if the data are in, say, a table, we may want to extract certain parts of it in ways that readHTMLTable() does not facilitate. To scrape data from these *HTML* pages, we typically look at the *HTML* source for the page. We find examples of the data we want and then look for the pattern(s) in the *HTML* code that identifies them. As with regular expressions, we need the pattern to be general enough to match the data we want, but specific enough to not include data we do not want. We will look at a reasonably straightforward but involved example and consider different approaches.

Relative to other *XML* document types, *HTML* trees tend to be reasonably deep due to all the formatting information. As a result, using *R* commands to traverse the tree and extract specific nodes using subsetting, xmlChildren(), etc., tend not to be very efficient. Instead, we use *XPath* to identify the elements we want. We can either extract the content or the specific nodes directly with *XPath* or we can get the basic node and then use *R* operations to access the subelements that are children or subchildren, but not distant descendants.

Example 5-5 Extracting Headlines from The New York Times' Web Pages

In this example, we will see how we can obtain the current headlines on the "front page" of The *New York Times*' Web site. We might do this to see how rapidly the headlines change. We might also do this for many newspapers to compare which stories they consider to be most important. This example shows how we find the patterns in the *HTML* that allow us to identify the content we want using an *XPath* expression.

We can look at The *New York Times*' main page (`http://nytimes.com`) and see the headline stories in various sections of the page. We want to look at the *HTML* source of the page. We can do this in our Web browser using the `View source` menu item. Alternatively, we can save the *HTML* document to a local file and then explore it in a text editor. In either case, we can search for text in a headline and then will be brought to the relevant *HTML* code that is used to display that headline. Another approach is to use the `Developer Tools` facilities in a Web browser and explore the Web

page and the source in the same window by moving the mouse over different elements in the *HTML* source and seeing which elements of the page are then highlighted.

When we look at the *HTML* source for a headline, we see something like the following:

```
<div class="wideB noBackground opening module"
      id="ledePackageRegion">
<div class="aColumn">
<div id="aLedePackageRegion">
 <div class="columnGroup first">
  <div class="story">
   <h2>
    <a href="http://www.nytimes.com/....">
  Gaza Violence Is Unabating as ... Push for Truce
    </a>
   </h2>
   <h6 class="byline">
     By JODI RUDOREN and FARES AKRAM
     <span class="timestamp"></span>
   </h6>
   <p class="summary">
   Israel's onslaught against the Gaza Strip
   </p>
</div></div></div></div></div>
```

We have to look at several of these headlines and see if we can find a pattern that identifies all of them and no other parts of the page.

In many Web pages, the layout is achieved using class attributes on *HTML* elements and *CSS*, along with `<div>` elements. In the *HTML* code above, we see classes `story`, `columnGroup first`, `byline`, and so on. The `story` class suggests that this is a way to identify headlines.

Let's find all `<div>` elements in the page that have a *class* attribute with a value `story`. We do this with

```
doc = htmlParse("http://nytimes.com")
storyDivs = getNodeSet(doc, "//div[@class = 'story']")
```

We can examine all of these nodes and see if a) they all have the same pattern and identify a headline, and b) we are missing any stories on the page that this expression does not capture.

From our sample *HTML* code, we expect the `<div>` node to have a header node named `<h2>` as the first child. Let's see if this is the case. Again, we can look in the *HTML* source for a story that is not the main headline to see if this is true. Alternatively, we can look at the name of the first child with

```
table(sapply(storyDivs, function(x) xmlName(x[[2]])))
```

```
comment     div      h2      h3      h5      h6      ul
      2       2       1       3       6      21       2
```

(We use the second child because the first one is the simple text node containing a new line.) Does this suggest our proposed pattern is incorrect? `<h2>` indicates a second-level heading. The New York Times may use different headings to identify different types of stories.

Rather than looking at the name of the first node in each of our `<div>` nodes, we look at its class:

```
table(sapply(storyDivs, function(x) xmlGetAttr(x, "class")))

story
   37
```

This is trivially the case since we used *XPath* to get the <*div*> nodes with a class attribute of `story`.

We might look at the names of the child nodes of our story <*div*>s and see if they have the same basic pattern.

```
table(unlist(lapply(storyDivs, names)))

comment     div      h2      h3      h5      h6
      2      19       1       7      13      46
      p    span    text      ul
     21       3     149       8
```

This suggests a lot of variation. If we continue to explore, we see story nodes that have a photograph or image such as (abridged)

```
<div class="story" id="ledePhotoStory">
 <div class="ledePhoto" id="ledePhoto">
  <div class="image">
   <a href="http://www.nytimes.com/...">
    <img width="337" height="225" alt="An Israeli ...."
      src="http://i1.nyt.com/images/.....jpg"/>
   </a>
  </div>
  <h6 class="credit">Pavel Wolberg for The New ...</h6>
 </div>
</div>
```

This is a photo story. We have to chose if we want to include these or not.

After a little exploration and comparison, we can see that the top-level stories we want are within a <*div*> node with a *class* attribute with the value `story` and also have a header node with a hyperlink node <*a*>. The header node can be named <*h2*>, <*h3*>, . . . , <*h6*>. We can find these with the compound *XPath* expression

```
//div[@class = 'story']/h2/a | //div[@class = 'story']/h3/a |
       //div[@class = 'story']/h4/a
```

and so on, up to `h6`. We can write this string manually in *R*, but we can also create it programmatically with

```
xp = paste(sprintf("//div[@class = 'story']/h%d/a", 2:6),
           collapse = " | ")
```

Now we can evaluate this expression with

```
stories = getNodeSet(doc, xp)
```

This returns 44 nodes.

Our *XPath* expression also included some additional "headlines" that we may not consider actual headlines. These include stories within different sections such as Art & Design, Sunday Review, Books, and so on. An example of the *HTML* source for this is

```
<div class="story">
 <h6 class="kicker">
  <a href="http://www.nytimes.com/...ion/index.html">
    Sunday Review</a>
 </h6>
 <h3><a href="http://www.nytimes.com/...html">
     A Phony Hero for a Phony War</a>
 </h3>
 <p class="summary">Generals g....</p>
</div>
```

The presence of the `kicker` value for the *<class>* attribute of the *<h6>* element seems to identify it. To eliminate these, we want to change the header criterion we specified to ignore those header elements with `kicker` as the class. We can go back and remove this with

```
xp = paste(sprintf("//div[@class = 'story']/
                    h%d[not(@class = 'kicker')]/a", 2:6),
           collapse = " | ")
```

When we evaluate this with getNodeSet(), we do indeed remove the items we did not want.

If we carefully go through the stories on the Web page and the titles we retrieved in our *XPath* query, we see we are missing several elements listed under `More News` on the Web page. In the *HTML* source, these appear as

```
<div class="columnGroup last">
 <h6 class="kicker">More News</h6>
 <div class="story">
  <ul class="headlinesOnly">
   <li>
    <h5>
     <a href="http://www.nytimes.com/...">
       Syria Assails Support for Opposition</a>
     <span class="timestamp">1:31 PM ET</span>
    </h5>
   </li>
   <li>
    <h5>
     <a href="http://www.nytimes.com/...">
       Gas Boom County Strives for Economic ...</a>
     <span class="timestamp"></span>
    </h5>
   </li>
  </ul></div></div>
```

We want to include these but have to add to the *XPath* query. Here we are looking for a pattern that is a * * with a story class attribute with an unordered list (*<ul>*) node with a list item (*<li>*) node and a hyperlink within a header element (*<h5>*). We can specify this with

```
//div[@class='story']/ul//h5/a
```

We might want to make this specific to the `More News` section by specifying the nodes must be associated with the <*h6*> element that contains the text `More News`. The <*h6*> node is a sibling of the story <*div*> so we can specify this with

```
//h6[text()[1] = 'More News']/
        following-sibling::div[@class='story']/ul//h5/a
```

We can also write this by looking for the <*div*> node that has an <*h6*> sibling before it that contains the string `'More News'`. The following *XPath* expression captures this:

```
//div[@class='story' and
       preceding-sibling::h6[. = 'More News']]/ul//h5/a
```

Before we proceed, we should note that we have assumed a header node named <*h5*>. We also should probably generalize this, although this is so connected to the `More News` group, this may not be necessary. Rather than making this expression more specific, however, we should also look at any other headlines we have missed. A physical newspaper is typically folded in half. People talk about stories above and below the fold. The online version also has a similar divide and even uses the word fold to describe the divide. If we scroll down the page and look at the headlines in the different sections such as World, Business Day, Technology, etc., we find that we are missing the headlines in those groups. The *HTML* source for some of these headlines looks like the following:

```
<h6 class="moduleHeaderLg">
 <a href="http://www.nytimes.com/...x.html">
   Technology </a>
</h6>
<ul class="headlinesOnly">
 <li class="firstItem wrap">
  <h6><a  href="http://www.nytimes.com/...?hpw">
    The iEconomy: As Boom ....</a>
  </h6>
 </li>
 <li class="">
  <h6><a href="http://www.nytimes.com/...">
     You for Sale: Your Online Attention....</a>
  </h6>
 </li>
</ul>
```

This pattern is quite similar to what we saw for the `More News` group, namely an <*h6*> element followed by a <*ul*> node with a class of `headlinesOnly`. We should try to capture both types of groups with a general *XPath* expression. The key here is that we are looking for the unordered list (<*ul*>) with the correct class and immediately preceded by an <*h6*> element. We can match these with the *XPath* expression

```
//ul[@class = 'headlinesOnly' and preceding-sibling::h6]
```

We add to this to get the hyperlinks for the headlines:

```
//ul[@class = 'headlinesOnly' and preceding-sibling::h6]/li/*/a
```

We use the `*` here to match either <*h5*> or <*h6*>, but it will match any node.

After we run this *XPath* query and check the actual headlines with what we matched, we note that we are missing one. This is under the title `Akela Flats Journal`. The *HTML* for this is

```
<h6 class="kicker">Akela Flats Journal</h6>
<h5><a href="http://www.nytimes.com/.hp">
Exiled Tribe's Casino Plans Lead to Conflict</a>
</h5>
```

Indeed, this does not match our pattern. We can either omit it or add a pattern to identify it. We have to chose how specific to be, e.g., do we tie it to the title `Akela Flats Journal` or just an *<h6>* node with a *class* `kicker` followed by an *<h5>* node? To determine how general to be, we need to look at other front pages for The *New York Times*. For now, we will just omit this headline.

Since we do not care about the order of the headlines, we can evaluate the *XPath* queries that match the headlines we want separately and then combine the results. The only danger in this is that we can end up with duplicate headlines, but we can remove these in *R*. However, combining the *XPath* expressions into one ensures that we get each matching headline node only once and so avoid this problem.

If we evaluate the *XPath* expressions separately, we have to ensure that none of them introduce spurious matches. Let's put all of these together to find the headlines. We had the main headlines that were in the * * nodes with a story *class* attribute, and we had the `headlinesOnly` entries. We can create a vector of these separate *XPath* expressions and then combine them into a compound expression with

```
xpTerms = c("//ul[@class = 'headlinesOnly'
                   and preceding-sibling::h6]/li/*/a",
            sprintf("//div[@class = 'story']/h%d/a", 2:6))
xp = paste(xpTerms, collapse = " | ")
```

We want the text of the headlines so we use xpathApply() to find the nodes and apply xmlValue() to each with

```
headlines = xpathApply(doc, xp, xmlValue)
```

We end up with some duplicates, but these are not duplicate nodes. Instead, it is because the same story appears in different groups/sections within the page, e.g., in Arts and also in Movies.

Often, things will be simpler than our *New York Times* example. We used it to illustrate many of the different practicalities of scraping from an *HTML* page. This example shows how brittle this data extraction approach is. A slight change in the *HTML* will cause our code to break, missing headlines or matching too many. These changes are entirely due to layout issues and not changes in the data themselves.

It is clear that it is much simpler if we can acquire the data directly in *XML* or *JSON* rather than in *HTML* intended for displaying in a Web browser. Indeed, we can get the headlines dynamically via an *RSS* (Rich Site Summary) feed. For example, see `http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml`. This feed even contains more metadata about the entries, including categories and publication dates. We can get the headlines from the *RSS* feed with

```
doc = xmlParse(nyTimesRSSURL)
headLines = xpathSApply(doc, "//item/title", xmlValue)
```

This only yields the recent primary headlines. However, there are many other *RSS* feeds for the different subsections, e.g., World News, Technology, Arts, Science, and so on.

We will look at another example to illustrate additional aspects common in scraping content from multiple, related *HTML* pages.

Example 5-6 Scraping Job Postings from Kaggle Web Pages

Kaggle (`http://www.kaggle.com`) is a Web site that, among other things, runs predictive modeling competitions. Kaggle also allows businesses, institutions, and individuals to post jobs related to data analysis and data mining. We are interested in collecting these job postings and analyzing them to get an understanding of the general patterns such as trends in desired skills, salary ranges, locations of jobs, and so on. We can also scrape similar data from more general job sites such as `monster.com`, `careerbuilder.com` and several others.

The *URL* for the Kaggle jobs board is `http://www.kaggle.com/forums/f/145/data-science-jobs` and Figure 5.2 shows a screen shot of the page. This is the first of several pages. We will want to process all of the pages, but we focus first on finding the job postings on this page.

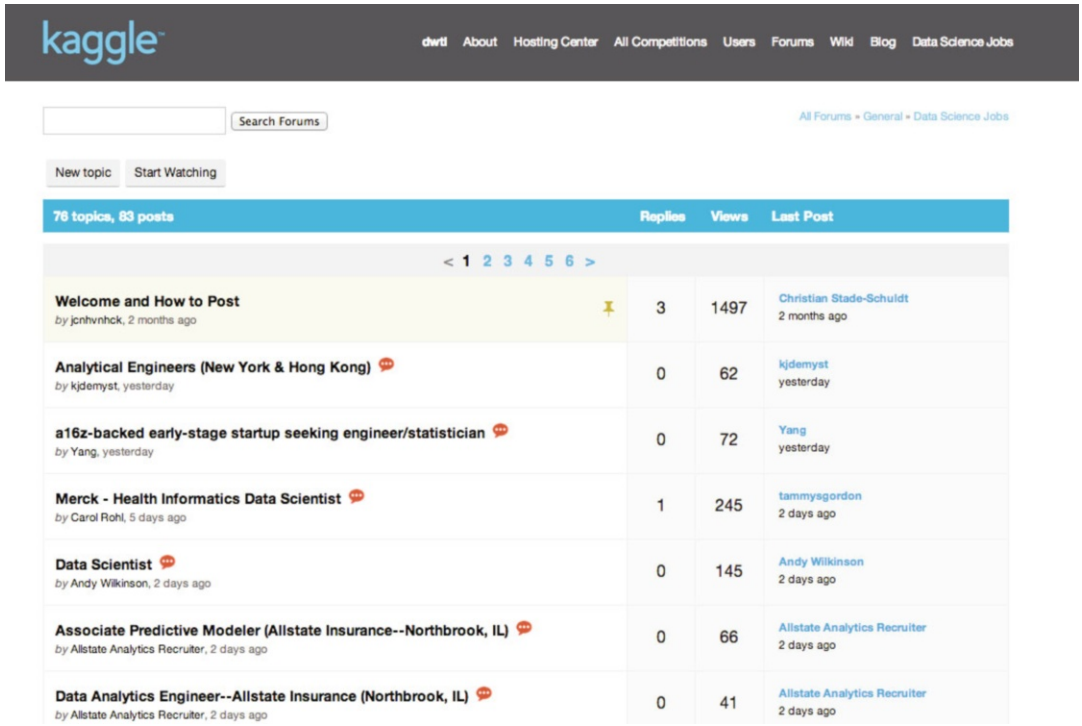


Figure 5.2: Example Kaggle Jobs Page. This is a screen shot of the Kaggle jobs posting page. Each job has a separate line and a link to the actual job posting. At the bottom of the page (not shown) is a row of links to subsequent pages.

Just looking at the Web page, we have no information about the link to each job posting. One immediate approach we can use in this situation is to merely look at the available links with the getHTMLLinks() function, e.g.,

```
links = getHTMLLinks(kaggleURL)
```

We can then look at these and see if there is a pattern, comparing them to the actual links we see when we mouse over the links on the page. One of the links is `http://www.kaggle.com/forums/t/3118/data-scientist`. Another is `http://www.kaggle.com/forums/t/3057/senior-scientist-playlists-oakland`. We also see links such as `/forums/f/145/data-science-jobs?page=2`. These are the links to the subsequent pages of job posts. Other links are of the form `/users/69144/carol-rohl` which are links to a Kaggle member's profile, and links to other part of Kaggle's Web page. It appears that the links we want are those that contain the path `/forums/t/`. We can use a regular expression and grep() to find these in the vector `links`, i.e.,

```
jobLinks = grep("/forums/t/", links, value = TRUE)
```

When we look at these *URL*s, we see two posts with very similar *URL*s, e.g.,

```
/forums/t/3125/analytical-engineers-new-york-hong-kong
/forums/t/3125/analytical-...-hong-kong/16954#post16954
```

It turns out that the first one is the link to the actual job announcement, and the second one is a link to the most recent posting about that announcement. We can ignore/discard those links in `jobLinks` that has a within-document anchor, i.e., #, in the *URL*. Again, we can do this with a regular expression. However, it is useful to see a) how we found out what these second links were for, and b) how we might go about finding only the links to the jobs more exactly.

We can look at the *HTML* source for the page. Like all *HTML* pages, the root node of the document is named <*html*>, and that has a <*head*> and <*body*> child node. The <*body*> is made up of many nodes. There is the navigation banner across the top with links to other parts of the Kaggle site, a search form, the footer, and so on. The list of job postings is in a <*div*> node, and this is inside another <*div*> node and the individual job entries are nested below these. Indeed, when we finally find a particular job listing, we can see that the actual link is the child of a level 3 header (<*h3*>) which is nested seven levels deep in <*div*> elements.

We found one of the job postings by searching for the text we see in the Web page associated with that. We can do this in the Web browser when viewing the source, or in a text editor viewing the saved *HTML* document. We can also do it in *R* with an *XPath* query, e.g.,

```
doc = htmlParse(kaggleURL)
getNodeSet(doc, "//a[contains(.,
            'Analytical Engineers (New York &amp; Hong Kong)')]")
```

This looks for the <*a*> node whose text child contains the string "Analytical ...". Note that we had to replace the & character with the *XML* equivalent of *&*.

Now that we have identified the particular node, we can look at it and its ancestors to try to identify a pattern that will allow us to find them all. The *HTML* around this link is

```
<div class="topiclist-topic">
 <div class="topiclist-topic-name">
  <h3>
   <a href="http:....-engineers-new-york-hong-kong"
    title="Go to topic '... Hong Kong) '">
  Analytical Engineers (New York &amp; Hong Kong)
    <img src='...b415d51cbe0b/shared/img/forum-new.png'
    class="new-icon" title="Unread message(s) in topic"
      width="16" height="15" />
   </a>
```

```
  </h3>
  <h4>
   <em>by</em>
   <a class="profilelink" href="/users/69760/kjdemyst"
      title="View kjdemyst&#39;s profile">kjdemyst</a>,
    21 hours ago
  </h4>
 </div>
 <div class="topiclist-topic-replies"> 0 </div>
 <div class="topiclist-topic-views"> 56 </div>
 <div class="topiclist-topic-lastpost">
  <a class="lastpost-link"
     href="http....-new-york-hong-kong/16954#post16954"
     title="Last post by kjdemyst">kjdemyst</a>
  <br />
    21 hours ago
 </div>
</div>
```

This is the *HTML* for a single job posting.

The *class* attribute on the `<div>` parent elements of the actual links (`<a>`) give us information about the purpose or role of these links. The outermost `<div>` has a class named `topiclist-topic`. The class for the `<div>` for the actual link is `topiclist-topic-name`. The link with the anchor is within a `<div>` that has class `topiclist-topic-lastpost`. We also see that we have information about the number of replies and the number of views of the job post, again identified via a class on the relevant `<div>` node.

We can look at other `<div>` nodes with class `topiclist-topic-name`. Again, this is probably easiest in a text editor or in the Web browser's view of the *HTML* source. We will see that these all contain the link to the job posting so we can use this to get the links directly with an *XPath* query in *R*. That is, we want to get the link within the `<div>` with a class attribute that has a value of `topiclist-topic-name`. There are actually two links under this `<div>` — the one we want and the one with a `profilelink` class. We notice that the link of interest to us is within an `<h3>` element. Therefore, we can compose our *XPath* expression as

```
"//div[@class='topiclist-topic-name']/h3/a/@href"
```

We can then evaluate this query with

```
doc = htmlParse(kaggleURL)
links = getNodeSet(doc,
              "//div[@class= 'topiclist-topic-name']/h3/a/@href")
```

The result is a list of the links.

While we were able to achieve the same result using a combination of getHTMLLinks() and grep() or a single *XPath* expression, the latter approach is much more common. Working just on the links is a special case that we were able to do here. It is simpler, but we typically have to explore the structure of the *HTML* and find *class* values that help us to identify the content we want.

We started Example 5-6 (page 133) by wanting the content of the actual job posting. We have only retrieved the links to them, and this is only for the first page. We will continue this example by getting the actual text of each of the job announcements.

Example 5-7 Getting the Content of Kaggle Job Posts Across Web Pages

If we click on any of the job postings in Figure 5.2, we see a page similar to that in Figure 5.3. There are several areas in the page, such as the header and footer, the title of the job post with a Reply and Start Watching button, information about the poster and the date of the post, a Quick Reply form and, of course, the posting itself in the center of the display. We want the text in that region.

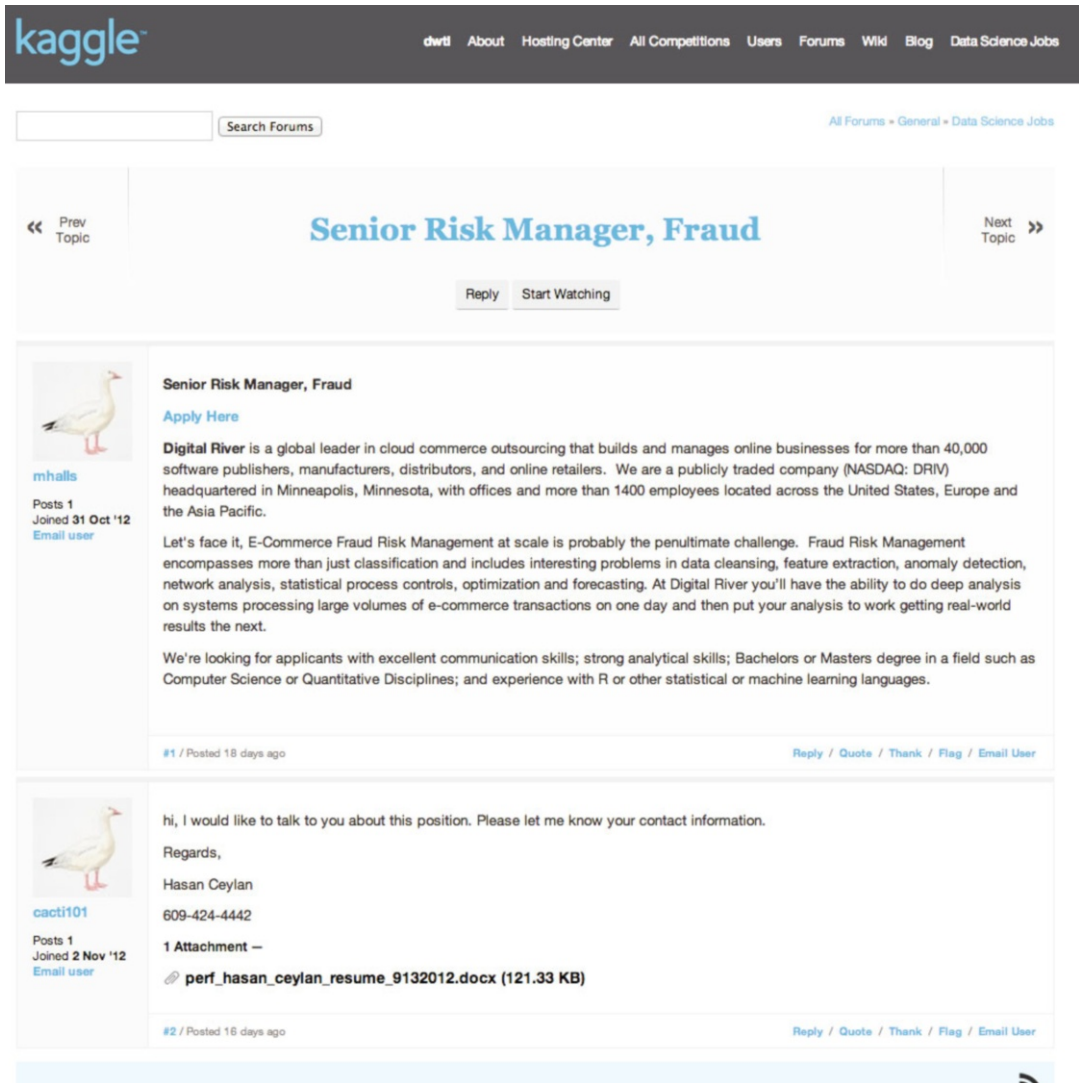


Figure 5.3: Example Kaggle Job Description Page. This screenshot shows the format of the page displaying a Kaggle job description. There are two posts on this page. The first is the job description and the second is a reply. The posts are arranged in a vertical table with a single column. Each post is itself a table with a column identifying the poster and the date of the post and a column for the post.

To find the *HTML* element containing the actual job description, we, again, have to look at the *HTML* source. By searching in a text editor for some text we see in the description, we are quickly led to *HTML* of the form

```
<div class="x_x_gs">
 <div id="x_x_:1yv"
     class="x_x_ii x_x_gt x_x_adP x_x_adO">
  <div id="x_x_:1zt">
   <a href="https://www.slice-data.com/" rel="nofollow"
    target="_blank">Slice Data</a>
is an early-stage startup based in Palo Alto,
comprised of folks from MIT, Google, Microsoft...
</div></div></div>
```

This is clearly not human-generated, but created by the editor used to enter the job description. That explains the *class* attribute values. We might guess that the *id* on the final <*div*> node might identify the description. However, will this be the same for other postings? In other words, is this a pattern we can use across postings? We have to look at a second job description.

It turns out that if we look at another page, we do not see the same <*id*> attribute or even the same sequence of <*div*> elements so the pattern we hoped might be present is not. We have to look for a different pattern. If we look at a page where somebody else has responded to the post, we see that the posts are aligned vertically and appear in a one-column table format. This does not mean that they are actually in a <*table*> node, but we can check.

When we use the "Developer Tools" in the Web browser to examine the *HTML* content and see the corresponding parts of the Web page highlighted as we mouse over parts of the *HTML* source, then we quickly see that each post on the job description page is in its own <*table*> node and these are arranged in a <*div*> of the form

```
<div id="topicview" class="forum">...</div>
```

Each post's <*table*> node has a *class* attribute with a value `post` so we can find the first post, which is the job description, with the simple *XPath* expression

```
//table[@class='post'][1]
```

The first column of each post displays information about the poster and the date. We want the second column. This is a <*td*> node and it actually has a *class* value of `postbox`. We can fetch the job description content in one *XPath* query with

```
//table[@class='post'][1]//td[@class = 'postbox']
```

We need to test this across different sample postings, but this turns out to be the case. We now also have a way to find content from responses to a job postingon this page.

The final step is to extract the actual text in the job description. The post is actually marked-up with *HTML* for formatting and providing links, etc. We want just the text. We can call the function xmlValue() with the <*td*> node, but this would collapse text across children without spaces. Instead, we can find all the text nodes and then get their content, e.g.,

```
xpathSApply(postNode, ".//text()", xmlValue)
```

where `postNode` is the *R* variable holding the node we obtained from the *XPath* query above.

Now we have the post and we can process it with some text mining functions to get the individual terms and words.

The final step in scraping all of the job posts on Kaggle is to process the remaining pages. To do this, we need to find the links to these other pages. One approach is to get links to all of the pages from the first page and then process these. An alternative is to step through the pages one at a time and get a link to the next page. For some sites, we have to use this second approach as there may not be links to all of the other pages displayed on the first or any of the individual pages. For example, Google displays links to the first 10 pages of search results. When we click on the link to the 10th page, that page shows links to additional pages, i.e., the 11th, 12th, etc., pages of the results.

How do we find the link(s) to the next page(s)? We can use getHTMLLinks() and grep() or find the actual pattern in the *HTML* source that identifies these links. Again, using regular expressions on the links works in this case. However, we will use this example to illustrate the more commonly needed approach.

Each of the links to a page appears in the Web page as a number, e.g., 2, 3, 4, ... Searching for these strings is likely to be frustrating. Instead, we might hover over one of the links when viewing the actual page in the Web browser and observe the target *URL* in the status bar. We can then search for this in the *HTML* source. We can search for `/forums/f/145/data-science-jobs?page=2` or some part of this. We find the *HTML* content is (formatted for easier reading)

```
<div class="forum-pages">
 <span class="disabled">&lt;</span>
 <span class="current">1</span>
 <a href="/forums/f/145/data-science-jobs?page=2">2</a>
 <a href="/forums/f/145/data-science-jobs?page=3">3</a>
 <a href="/forums/f/145/data-science-jobs?page=4">4</a>
 <a href="/forums/f/145/data-science-jobs?page=5">5</a>
 <a href="/forums/f/145/data-science-jobs?page=6">6</a>
 <a href="/forums/f/145/data-science-jobs?page=2">&gt;</a>
</div>
```

We want the value of the *href* attributes in the <*a*> elements that are children of the <*div*> element with class `forum-pages`. This is easy to get with *XPath* with

```
getNodeSet(doc, "//div[@class='forum-pages']/a/@href")
```

This expression identifies duplicate links, for two reasons. Firstly, there are actually two sets of links on the pages and so two <*div*> elements with class `forum-pages`. We want just one of these so we can use

```
// div[@class='forum-pages'][1] /a / @href
```

Here we have added `[1]` after the *class* predicate in order to get just the first of these <*div*> nodes.

The second reason for duplicates is because, in this case, the second page is repeated for the link displayed as a >. We can remove duplicates in *R* with a call to unique(). However, we can also do it in *XPath* by not matching <*a*> elements whose text is >. We would do this with

```
//div[@class='forum-pages'][1]/a[position() < last()]/@href
```

The links are local paths. We need to put them together with the base *URL*, i.e., `kaggle.com`. We can do this generally with

```
pageLinks = getNodeSet(doc, "//div[@class='forum-pages'][1]/
                                a[position() < last()]/@href")
getRelativeURL(as.character(pageLinks), docName(doc))
```

Here we compute the *URL*s relative to the document in which they are found. We get this via a call to docName().

If we want to just get the link for the next page, we can use

```
getNodeSet(doc, "//div[@class='forum-pages'][1]/a[1]/@href")
```

and this gives `/forums/f/145/data-science-jobs?page=2`. However, for the next page, this would give the wrong result: `/forums/f/145/data-science-jobs`, which sends us back to the first page. In fact, now we want the <*a*> node whose text value is >, which we excluded before. Alternatively, we want the <*a*> element which is the sibling following the <*a*> node that follows the <*span*> with the *class* attribute `current`. The former approach is easier, but we can use *XPath* to do the latter with

```
//div[@class='forum-pages'][1]/
    span[@class='current']/following-sibling::a[1]
```

Here we use a node test on the **following-sibling** to limit to only the <*a*> nodes. We only want the first of these.

We can now put all of the pieces together into code to get all the current job descriptions on Kaggle. We will create several short functions to do this.

The top-level function we will use is named getAllKaggleJobDescriptions(). This loops over the different pages of job postings and calls a function getKaggleJobPageDescriptions() to get the individual job descriptions. We define getAllKaggleJobDescriptions() as

```
getAllKaggleJobDescriptions =
function(doc = KaggleJobURL, pageLinks=getKaggleJobPageLinks(doc))
{
  if(is.character(doc))
    doc = htmlParse(doc)

  lapply(pageLinks, getKaggleJobPageDescriptions)
}
```

The caller can pass the document in many ways. The caller can use the default *URL* or pass an alternative, should there be one. We also allow the caller to pass the already parsed document. We can add more error handling code to ensure that `doc` is an `HTMLInternalDocument` before the call to lapply(). Some callers may want to use this function to loop over a subset of the pages. They specify this subset of *URL*s via the *pageLinks* parameter. This defaults to all pages as computed by getKaggleJobPageLinks().

We define this function getKaggleJobPageLinks() as

```
getKaggleJobPageLinks =
function(doc = htmlParse(KaggleJobURL))
{
  pageLinks = getNodeSet(doc,
                         "//div[@class='forum-pages'][1]/
                                a[position() < last()]/@href")
  union(docName(doc), # add the first one too.
        getRelativeURL(as.character(pageLinks), docName(doc)))
}
```

This uses the *XPath* expression we developed above to get the links we want. We have to take care to add the main job page which we omitted in our *XPath* expression.

Our top-level function getAllKaggleJobDescriptions() calls the getKaggleJobPageDescriptions() function so we define this next. Its job is to loop over the links in the page for the individual job descriptions and retrieve the text for each of those job postings. We define this with

```
getKaggleJobPageDescriptions =
function(doc, links = getNodeSet(doc,
                "//div[@class='topiclist-topic-name']/h3/a/@href"))
{
  if(is.character(doc))
    doc = htmlParse(doc)

  lapply(links, getKaggleJobDescription)
}
```

The final function is getKaggleJobDescription() and this corresponds to retrieve the text of the job posting. The code for this is

```
getKaggleJobDescription =
function(url, collapse = "\n")
{
  doc = htmlParse(url)
  postNode = getNodeSet(doc,
                        "//table[@class='post'][1]//
                              td[@class = 'postbox']")[[1]]

  txt = xpathSApply(postNode, ".//text()", xmlValue)

  if(length(collapse) && !is.na(collapse))
     paste(txt, collapse = collapse)
  else
     txt
}
```

This allows the caller to collapse the individual strings into a single string which makes it easier to process when we are looking for information about the jobs.

If we put this code into an *R* package named, say, `KaggleJobs`, we would probably omit the `KaggleJob` term in each of the function names.

Note that while Kaggle has divided the job postings into separate pages, some sites have a link that allows us to see all the postings, or to select the number we see, on one page. It is a good idea to use this to try to consume as many links as possible.

We have used htmlParse() to both retrieve and parse the *HTML* document in a single *R* command. When scraping information on different pages of a Web site, we often make many requests in rapid succession to that Web server. This can cause the site to limit your access. Additionally, we are making each request entirely separately. The call to htmlParse() fetches the page but starts a new conversation with the server each time. Since we are retrieving many pages, it may be more efficient to establish a connection to the Kaggle Web server and make the requests using that connection. For multiple requests to the same server, we can use the RCurl package to make the requests and use the same connection for each request. We would create a single curl handle with getCurlHandle() to pass to getForm() and getURLContent(). Then, we pass each result to htmlParse(). This can improve the

speed of the network part of the task. Indeed, we can even make the requests operate in parallel to potentially improve the speed further. See Chapter 8 for more details.

Extracting data from *XML* is typically easier and more robust than working with *HTML* content. We will look at a reasonably comprehensive example that illustrates the combination of *XPath* and the node manipulation *R* functions. It illustrates not just how we extract the data, but also how we find and verify the patterns in the *XML* structure as we explore it initially.

Example 5-8 Extracting Data About Pitches in a Baseball Game

We revisit the data of Example 5-3 (page 124) and look at the first game of the 2011 World Series These data are available in different files and directories under the *URL* `http://gd2.mlb.com/components/game/mlb/year_2011/month_10/day_19/`. Ball-by-ball information is available for each inning in the directory `gid_2011_10_19_texmlb_slnmlb_1/inning/`. There is a separate file for each inning, but also a single file containing all of the innings. We will read that, i.e., `http://gd2.mlb.com/components/game/mlb/year_2011/month_10/day_19/gid_2011_10_19_texmlb_slnmlb_1/inning/inning_all.xml`, with

```
doc = xmlParse(u)
```

We can explore the file in a text editor or by querying its structure in *R*. We can look at the root node and find the names of its child nodes:

```
r = xmlRoot(doc)
table(names(r))

inning
     9
```

Let's see if all of the innings have the same basic structure:

```
table(unlist(xmlApply(r, names)))

bottom    top
     8      9
```

All but the last inning has a <*top*> and <*bottom*> node. The final inning has no <*bottom*> node since this game was already won at that point and the home team did not need to bat.

Next we find the names of the child nodes in each inning:

```
table(unlist(xpathApply(doc, "//inning/*", names)))

action  atbat
    26     68
```

The <*atbat*> nodes are the primary source of information. The <*action*> nodes describe an event that happened during the at-bat, e.g., a coach coming to talk to he pitcher, or a wild pitch. These often give us more information about the most recent at-bat or pitch.

We can find the children of the first <*atbat*> with

```
names(r[[1]][["top"]][[1]])

   pitch    pitch    pitch   runner
 "pitch"  "pitch"  "pitch" "runner"
```

There are three pitches and then information about the runner on base, i.e., the node

```
<runner id="435079" start="" end="1B" event="Single"/>
```

shows the lead-off runner on first base. Each at-bat is made up of a number of pitches and we will focus on these. The <*atbat*> node itself provides information about the batter and pitcher, the time and summary of the at-bat (balls and strikes), and the state of the inning (outs). We can see this in the abridged first <*atbat*> node:

```
<atbat num="1" b="2" s="0" o="0" start_tfs="200524"
       start_tfs_zulu="2011-10-20T00:05:24Z"
       batter="435079" stand="R" b_height="6-0"
       pitcher="112020" p_throws="R"
       des="Ian Kinsler singles on a ground ball to
           shortstop Rafael Furcal, deflected by
           third baseman David Freese. "
       event="Single">
 <pitch ..... />
 <pitch ..... />
 <pitch ..... />
 <runner id="435079" start="" end="1B" event="Single"/>
</atbat>
```

The *num* attribute identifies the at-bat. Are these numbers consecutive within innings and restart with each new inning? or for each team? or are they consecutive numbers across the entire game? We can determine this with a simple *XPath* query:

```
as.integer(getNodeSet(doc, "//atbat/@num"))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
[18] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
[35] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
[52] 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
```

The batter and pitcher are given as unique identifiers which we have to look up in a different document. We see that the batter and runner are the same person from the <*runner*> node, as we would expect.

Each <*pitch*> node is similar to the following example *XML*:

```
<pitch des="Ball" id="3" type="B" tfs="200633"
       tfs_zulu="2011-10-20T00:06:33Z"
       x="58.37" y="132.97" sv_id="111019_190630"
       start_speed="91.5" end_speed="83.9"
       sz_top="3.25" sz_bot="1.47"
       pfx_x="-8.07" pfx_z="6.61"
       px="1.309" pz="3.069"
       x0="-1.469" y0="50.0" z0="6.1"
       vx0="10.089" vy0="-133.818" vz0="-4.169"
       ax="-14.57" ay="30.045" az="-20.166"
       break_y="23.8" break_angle="30.4"
       break_length="5.5" pitch_type="SI"
       type_confidence=".909"
       zone="12" nasty="45"
```

```
spin_dir="230.505" spin_rate="2046.932"
cc="" mt=""/>
```

Again all of the information is in attributes. We have a description of whether the pitch was a Ball or a Strike, in both *type* and *des*. We also have lots of information about the trajectory of the ball. One explanation of these can be found at http://webusers.npl.illinois.edu/~a-nathan/pob/tracking.htm.

Now that we know the structure and contents of the document, we can think about how to bring this into *R*, and more specifically how to represent it in *R*. There are certainly several different ways we can store these data in *R*. We can use the same hierarchical structure as in the document, i.e., a list with an element for each inning, each of which has two elements—one each for the top and bottom of the inning. Within that we can have the at-bats and this is a list with an element for each pitch. We can store each pitch as a character vector of the attributes. However, we should probably convert some of these values to numbers, e.g., *x0*, *y0*, and so on.

A hierarchical structure does not make it easy to perform common calculations in *R*, e.g., plotting the coordinates that show where the pitch crossed the plate. Instead, a data frame makes many operations simpler; however, it really depends on what we want to do with the data. A data frame is a good general representation so we will extract the *XML* data into a data frame. Each row in the data frame will correspond to a <pitch> node. We will ignore the <action> nodes, but we can add these as a column with relatively few nonmissing values.

The obvious columns in the data frame correspond to the attributes in the <pitch> nodes. We can select the attributes we want. Alternatively, we can take all of them. If we do the latter, we should verify that each <pitch> node has the same attributes. There are various ways to check this, but counting the number of occurrences of each attribute name should suffice.

```
atNames = xpathApply(doc, "//pitch", function(node)
                                        names(xmlAttrs(node)))
tt = table(unlist(atNames))
```

We can now see if these are all the same and equal to the number of <pitch> nodes.

```
table(tt)
```

```
 27  41  68 258
  1   1   1  36
```

This tells us that there are some attributes that occur in only some of the <pitch> nodes. We can find the names of these with

```
names(tt)[ tt != max(tt) ]
```

```
[1] "on_1b" "on_2b" "on_3b"
```

We will ignore these.

We will identify the at-bat, batter, pitcher, etc., for each pitch. This comes from the parent <atbat> node for each <pitch> node. This will involve repeating this information for each pitch within an at-bat. This is the nature of storing items in single data frame. We can use two different data frames in the same way as a relational database and use an index to avoid the redundancy. For simplicity, we will use a single data frame.

We have two reasonably obvious approaches to creating the core columns from the <pitch> nodes. One is to process each of these nodes, get the attributes, discard the on_1b, on_2b and on_3b entries and then stack the resulting vectors. We can do this with

```
ats = xpathSApply(doc, "//pitch",
                  function(x) {
                    a = xmlAttrs(x)
                    a[ !grepl("^on_[1-3]b", names(a)) ]
                  })
```

This gives us a matrix with rows corresponding to the attributes and columns corresponding to the <*pitch*> nodes. We have assumed that the attributes are in the same order for each <*pitch*>.

We can transpose this matrix and coerce it to a data frame:

```
pitches = as.data.frame(t(ats), stringsAsFactors = FALSE)
```

We have yet to convert the numeric variables.

The second approach is to extract each attribute separately using *XPath* to find that attribute in each <*pitch*>. We start by getting the names of the attributes we want

```
atNames = grep("^on_[1-3]b", names(tt), invert = TRUE, value =TRUE)
```

We can create the individual *XPath* queries in a single call

```
xp = sprintf("//pitch/@%s", atNames)
names(xp) = atNames
```

Now we can sequentially apply these to the document and create a list and then a data frame:

```
tmp = lapply(xp, function(q) as.character(getNodeSet(doc, q)))
pitches = as.data.frame(tmp, stringsAsFactors = FALSE)
```

The next step is to add the information from the <*atbat*> nodes. One approach to this is to find all of the <*pitch*> nodes and then get the attributes of each pitch's parent <*atbat*> node. This is the simplest approach. It assumes that all of the <*atbat*> nodes have the same attribute names. Again, we need to check this.

```
tt = table(unlist(xpathSApply(doc, "//atbat",
                              function(x)
                                 names(xmlAttrs(x)))))
sort(tt)
```

```
away_team_runs home_team_runs          score              b
             3              3              3             68
      b_height         batter            des          event
            68             68             68             68
           num              o       p_throws        pitcher
            68             68             68             68
             s          stand      start_tfs start_tfs_zulu
            68             68             68             68
```

We either want to discard the three attributes that are not in each <*atbat*> node, or add an `NA` for those nodes that do not contain an attribute. We will illustrate the second option with

```
extraAttrs = c("away_team_runs", "home_team_runs", "score")
atbats = xpathApply(doc, "//pitch",
                    function(node) {
                       a = xmlAttrs(xmlParent(node))
```

```
    i = match(extraAttrs, names(a))
    if(any(is.na(i)))
       a[ extraAttrs[ is.na(i) ] ] = NA
    a
})
```

Here we used xpathApply() rather than xpathSApply() so we end up with a *list*, not a matrix. We can combine the elements into a data frame with

```
ab = as.data.frame(do.call(rbind, atbats), stringsAsFactors =FALSE)
```

Both approaches end up with the same data frame. Using xpathApply() can be useful if we create an actual data frame for each of the <atbat> nodes. We will see this below.

We now have two data frames—one from the <pitch> nodes and the other from the <atbat> nodes, but with rows repeated to correspond to the <pitch> nodes. We can combine them with cbind():

```
pitches = cbind(pitches, ab)
```

To get the at-bat information, we had to repeat the computations for each <pitch> node. This involved adding the missing values for each <atbat> multiple times. Instead, we can work with each *atbat* node and create a data frame for that at-bat with the attribute values repeated as many times as there are <pitch> nodes within that <atbat> node. For a given <atbat> node, we can create this data frame with the following function:

```
makeAtBatDF =
function(node)
{
  n = sum(names(node) == "pitch")
  a = xmlAttrs(node)
  i = match(extraAttrs, names(a))
  if(any(is.na(i))) a[ extraAttrs[ is.na(i) ] ] = NA

  as.data.frame(lapply(a, rep, n), stringsAsFactors = FALSE)
}
```

We can then use this function in a call to xpathApply()

```
abs = xpathApply(doc, "//atbat", makeAtBatDF)
```

Each element of `abs` is a data frame with a different number of rows, but with the same columns. We can now use `do.call(rbind, abs)` to combine these into a data frame with rows corresponding to the `pitches` data frame. We can then cbind() the two data frames together.

The last step in creating our data frame is to convert the columns to *numeric* and *factor* vectors, as appropriate. We can use type.convert() with

```
pitches[] = lapply(pitches, type.convert)
```

The `pitcher` and `batter` elements look like numbers, but are really unique identifiers. We can convert those to a *factor*:

```
pitches[c("batter", "pitcher")] =
                  lapply(pitches[c("batter", "pitcher")], factor)
```

In fact, we can look up the names in another *XML* file. Within the top-level directory for each game, there is a file named `players.xml`. The structure of the document is that we have a root node `<game>` with two `<team>` child nodes. Each `<team>` node has many `<player>` nodes which appear similar to

```
<player id="119984" first="Darren" last="Oliver" num="28"
     boxname="Oliver" rl="L" position="P" status="A" avg=".000"
     hr="0" rbi="0" wins="0" losses="0" era="0.00"/>
```

These give us the identifier, the first and last name, the player's number, position, and performance in the series. There are also `<coach>` nodes that provide the details for the coaches within each `<team>` node, and the umpires for the game are listed under the `<game>` node as children of an `<umpires>` node.

We saw in Example 5-3 (page 124) how to read all of the information for the players into a data frame in *R*. Here, we only want the player's identifier and first and last name. We can extract these into an *R* data frame with

```
pdoc = xmlParse("players.xml")
info = xpathSApply(pdoc, "//player",
                   function(node)
                     xmlAttrs(node)[c("id", "first", "last")])
playerInfo = as.data.frame(t(info), stringsAsFactors = FALSE)
```

Now we can add the first and last name of the pitcher to our `pitches` data frame:

```
i = match(as.character(pitches$pitcher), playerInfo$id)
pitches$pitcher = factor(playerInfo$last[i])
```

Now that we have the data, we can, e.g., look at the locations of the pitches, identifying them as balls or strikes and by the different pitchers.

5.4 Extracting Multiple Variables From *XML* Content

It is very natural to use different *XPath* queries to retrieve all of the values in an *XML* document for a given variable. We can then combine the resulting vectors into a data frame in *R* and then we have the data we want from the *XML* document. Unfortunately, this approach sometimes runs into a particular problem. To illustrate the issue, we consider a very simple made-up example of an *XML* document displayed below. We have two observations—people—and three variables recording each person's first name, middle initial, and surname. For one of the records, we do not have the middle initial.

```
<data>
 <obs id="1">
   <firstname>Deborah</firstname>
   <surname>Nolan</surname>
 </obs>
 <obs id="2">
   <firstname>Duncan</firstname>
```

```
   <initial>W</initial>
   <surname>Temple Lang</surname>
 </obs>
</data>
```

Suppose we want to create a data frame with the values of <*firstname*> and <*surname*> as two separate columns. We can do this quite simply with

```
doc = xmlParse("MissingObsNode.xml")
data = data.frame(firstname = xpathSApply(doc, "//obs/firstname",
                                          xmlValue),
                  surname = xpathSApply(doc, "//obs/surname",
                                          xmlValue))
```

The problem arises if we also want to have a column for the initial. We can get this from the observations in the same way with

```
data$initial = xpathSApply(doc, "/data/obs/initial", xmlValue)
```

However, the xpathSApply() call returns a vector with only one element since only one node matches the *XPath* expression. When we add this to the data frame, the values are recycled and we end up with an initial `W` for both observations. The problem is that the <*initial*> node is optional and not present in all of the observations. When we extract the value for this optional variable via *XPath*, we lose the association between the values and the observation. The single value we got was for the second observation, not the first.

This fabricated example illustrates the problem of having optional nodes/variables within *XML* nodes that we treat as observations. When we use *XPath* to extract the values of individual variables one at a time, we may be processing different numbers of observations and the results may not be aligned correctly across the observations. We need to ensure that the values of the separate variables are associated with the correct observations. There are a few ways to do this.

This is not an abstract problem. It arises in many situations since *XML* allows us to have different child nodes within nodes of the same name. An example of this is information about lenders in the Kiva data that we introduced in Section 1.3 and read in Example 3-1 (page 54). The data describe different individuals (or groups) that provide micro loans. Each lender has a <*lender_id*>, a <*name*>, an <*occupation*> and so on. However, most but not all of the <*lender*> nodes also have a <*loan_count*> node giving the number of loans that lender has made since joining Kiva. Even fewer have a <*country_code*>, and very few have an <*invitee_count*>. We can find this information using some simple *R* commands on the <*lender*> nodes within the document. We calculate the names of the child nodes within each <*lender*> node and then count the total number of occurrences of each node name across all the lenders, i.e.,

```
lendersDoc = xmlParse("lenders1.xml")
lenderNodes = xmlRoot(lendersDoc)[["lenders"]]
varNames = xmlSApply(lenderNodes, names)
table(unlist(varNames))
```

```
     country_code             image     invitee_count         lender_id
              957               686               334              1000
     loan_because        loan_count      member_since              name
             1000               985              1000              1000
       occupation occupational_info      personal_url               uid
```

```
            1000              1000              1000          1000
      whereabouts
              948
```

We start by creating the basic data frame with variables that are present in all of the <*lender*> nodes, e.g., `lender_id`, `loan_because`, `name`, `occupation`. We can get these with

```
varNames = c("lender_id", "loan_because", "name", "occupation")
vars = lapply(varNames,
              function(var) {
                xp = sprintf("//lender/%s", var)
                xpathSApply(lendersDoc, xp, xmlValue)
              })
names(vars) = varNames
```

Now if we try to do the same operation to get the values of the <*loan_count*> nodes, we will end up with a vector of 985 values, and not 1000. We have lost the association between the values and the observation/row in our would-be data frame. In this case, the observational unit is the lender. We consider three approaches to address this problem. We describe the first approach, which uses unique identifiers in the nodes, in the following example.

Example 5-9 Extracting Loan Counts from Kiva Using Unique Identifiers on Nodes

If we can find a unique identifier for each observation, then we can use this identifier to associate the `loan_count` values with the appropriate observation. That is, we can then use the identifier to set the values for specific observations for variables that are optional in the *XML* nodes.

With these data, we have at least two potential candidate variables—`lender_id` and `uid`. The `name` variable is not actually unique across lenders so we cannot use this. We can create a data frame from the variables we have computed above and then use `lender_id` as the row names. We do this with

```
lenders = as.data.frame(vars, stringsAsFactors = FALSE,
                        row.names = vars$lender_id)
```

We can now create `loan_count` from the corresponding nodes in the *XML* document. However, we also need to get the `lender_id` at the same time so that we know how to associate the values with the correct row of our data frame. We can do this in two ways: a) a single *XPath* query to get the <*lender*> nodes that have a <*loan_count*> child and then extract both the value of the `loan_count` and the `loan_id`, or b) make two separate *XPath* queries to identify the same <*lender*> nodes, but get the specific children separately. Both approaches will require that we create the `loan_count` variable in our data frame with suitable default values for each observation, e.g.,

```
lenders$loan_count = rep(NA, nrow(lenders))
```

In both approaches, we will assign values to a subset of the observations using the `lender_id` values.

We can implement the first approach with

```
tmp = getNodeSet(lendersDoc, "//lender[loan_count]")
ids = sapply(tmp, function(node) xmlValue(node[["loan_id"]]))
lenders$loan_count[ids] =
        as.integer(sapply(tmp, function(node)
                                  xmlValue(node[["loan_count"]])))
```

Here `tmp` is a list of the 985 `lender` nodes that have a `loan_count` child. We loop over these to get the values of `lender_id` and `loan_count` and then use these to assign the values.

The second approach is quite similar. However, we get the values of `lender_id` and `loan_count` in two separate but similar and related *XPath* queries:

```
ids = xpathSApply(lendersDoc, "//lender[loan_count]/lender_id",
                  xmlValue)
lenders$loan_count[ids] =
    as.integer(xpathSApply(lendersDoc,
                           "//lender[loan_count]/loan_count",
                           xmlValue))
```

In our example, we had two unique identifiers for each observation—`lender_id` and `uid`. We can check they are unique with

```
length(unique(lenders$id)) == nrow(lenders)
```

However, what happens if we do not have a variable or node within each observation that uniquely identifies it? Well, we can always create a unique identifier for the nodes corresponding to the observational unit, i.e., <*lender*> in our example. The simplest identifier to use is the index of the node, i.e., 1, 2, 3, ... We can also compute a checksum from the text representation of the each node, e.g.,

```
sapply(lenderNodes,
       function(node)
         digest(saveXML(node), "md5", serialize = FALSE))
```

or we might use *XSL*'s `generate-id()` function to generate the identifiers. Regardless of how we create the identifiers, we have to somehow associate them with the *XML* nodes. There is no value having the identifiers in *R* as we cannot match the nodes resulting from an *XPath* expression (unless we compute the checksum for each of those nodes). Instead, we want to put the identifiers into the observation nodes in the *XML* document. We might add each identifier as an attribute named *id*. Alternatively, we can use a different attribute name (if any of the identifiers are already being used in the *XML* document for any node, not just the observation node). Instead of using an attribute, we can add the identifiers as child nodes, similar to <*lender_id*> in our example. In other words, we will modify the *XML* document to add our unique identifiers. Creating and modifying nodes is the topic of Chapter 6.

The unique identifiers we create are only for our own purpose. They have no other role in the *XML* document other than allowing us to associate a node with a row in our data frame or element of a vector. So why are we changing the *XML* document, and won't this have repercussions? In short, this is fine. We are not changing the original *XML* document on disk. Instead, we are changing a copy of it in memory. When we are finished with the modified document, it will be garbage collected and the modifications will disappear. If we want to reuse the *XML* document later in the *R* session and do not want our modifications to be present, we can make a copy of the *XML* tree before we make the changes. We can do this with the xmlClone() function, e.g.,

```
tempDoc = xmlClone(lendersDoc)
```

and then work with this instead of the original document, which will remain unaltered.

5.4.1 Extracting an Entire Observation: A Different Approach

The problem we had above was that we were extracting variables for our data frame individually and needed to associate the values with the correct observations. An alternative approach is to process an entire observation node in one step and create all of the different variables. We can then combine the different observations into a data frame knowing that we have values for all of the variables for each observation. To do this, we operate on the observation nodes, i.e., in <lender> nodes in our example. We can do this with code like

```
obs = lapply(lenderNodes,
             function(node) {
               ans = structure(sapply(node[varNames], xmlValue),
                               names = varNames)
               ans["loan_count"] =
                    if(!is.null(a <- node[["loan_count"]]))
                      xmlValue(a)
                    else
                      NA
               ans
               })
```

The key idea here is that we check if the optional elements (node or attribute) are present in the *XML* node for each observation and ensure that there is a value in the *R* object representing the observation.

The result of our code above is a vector (or a list or a data frame) for each observation. We can combine these with

```
lenders = do.call(rbind, obs)
```

We can then convert the columns to the appropriate types, e.g., *integer*, *logical*, *factor*, and so on.

5.4.2 Modifying the Tree Before Extracting Variables: A Final Approach

We end this section by looking at a different approach whereby we can still use separate *XPath* queries over the <lender> nodes to get different pieces of information. The problem we encountered was due to the *XML* having optional elements. If every <lender> node has a <loan_count> child, we can create the `loan_count` variable in our data frame in the same way as we did for the other nodes. There is no issue in associating the values with the correct observations. This suggests that we can modify our *XML* document by adding the missing elements to any of the relevant observation nodes. In our example, we can find the <lender> nodes that do *not* have a <loan_count> child. For each of these, we insert an empty <loan_count> node. We can perform these steps with

```
lendersDoc = xmlParse("lenders1.xml")
nodes = getNodeSet(lendersDoc, "//lender[not(loan_count)]")
sapply(nodes, function(node) node[["loan_count"]] = "")
```

The command `node[["loan_count"]] = ""` inserts (or replaces) the *XML* node named <loan_count> with no child nodes.

Now every <lender> node in our document has a <loan_count> node. When we extract the value for each <lender> node with

```
xpathSApply(lendersDoc, "//lender/loan_count", xmlValue)
```

we end up with a value for each observation. Converting this to an *integer* vector will result in `NA` values for each of the observations for which we had no `<loan_count>` node in the *XML* document.

Up to this point, we have parsed the entire document, including other *XML* documents or parts of them. Next we turn to examining how to control the inclusion of these parts when parsing a document.

5.5 Integrating Parts of Documents with *XInclude*

In Section 4.9, we introduced the *XInclude* mechanism for *XML* documents. This allows us to directly include specific segments or subparts of one or more *XML* or text documents within another *XML* document. This is a very powerful facility and typically we want the xmlParse() function to simply replace the `<xi:include>` nodes in our top-level *XML* document with the actual content being included. This is what xmlParse() does by default. There are, however, some situations when we want the parser to treat the `<xi:include>` nodes as they are and not to perform the *XInclude* operations for us. We can achieve this using xmlParse()'s *xinclude* parameter. Specifying this as `FALSE` causes the parser not to perform the *XInclude* and to leave the verbatim `<xi:include>` nodes in the resulting parsed document.

When would we not want to process the *XInclude* nodes, but treat them as regular *XML* nodes? One common situation is when we simply want to find the names of all the files that are *XInclude*d, either directly or indirectly, within the top-level document. For example, this book was written in *DocBook*, and we have a single top-level *XML* file, say `book.xml`, that uses *XInclude* to bring in the chapters, which are in separate files and directories. Each of these chapters, in turn, uses *XInclude* to bring in sections and so on. We use *make* to create the *PDF* version of the book (as well as *HTML*, *FO*, and LaTeX versions). To do this, we need the complete list of all of the input files so that *make* can determine if it needs to update the LaTeX and/or *PDF* document. We can obtain this list of included files by recursively parsing the input files and examining their `<xi:include>` nodes. If the parser actually performed the *XInclude* steps, we would not be able to determine which content came from which file. The function getXIncludes() finds the list of dependencies for us using the *xinclude* argument to xmlParse() and provides some additional features such as not recursively processing non-*XML* documents.

Another situation in which we do not want the *XInclude* content actually included in the document is where we want to parse a document, modify it and then write the individual updated parts to their original separate files. For example, suppose we are writing a document in *DocBook* and we have different sections in different files like this book or an article, etc.. We would use *XInclude* to incorporate the different pieces to create the overall document. Now, if we want to programmatically modify the document to, say, update all citations or change the URLs of hyperlinks to a particular site, we would not want to process and modify the fully parsed document. Instead, we want to read each of the individual files and modify them separately. For this, we want to avoid performing the *XInclude* operations when we read each file. If we did not, we would have the entire updated document, and when we wrote that back to a file, all of the content would be in a single file. We would have broken the separation into the different pieces and have the unmodified content in the original separate files and the updated content in a single file. When we do not want to do this, the *xinclude* parameter can be useful.

We should note that when xmlParse() does perform the *XInclude* operations to create the entire document, it actually inserts special nodes within the document that identify the *XInclude*d parts. Given a node in a subtree that was *XInclude*d, we can use the function findXInclude() to find the

special node and determine from which file it actually came. This feature of the parsed tree allows us to implement the useful getNodeLocation() and getNodePosition() functions, which returns the name of the file and the line number for a given node.

Up to this point, we have focused on manipulating the tree and its nodes. We have manually identified the data we want and the structure of the *XML* documents and written commands to retrieve them. Before we turn our attention to a quite different approach to parsing *XML* content, we digress briefly to mention how we might use *XML* schema (Section 2.7) to programmatically generate *R* class definitions and code corresponding to the data types in a schema. These can then be used to parse an *XML* document that is formatted according to this schema into appropriate *R* data structures.

5.6 Reading *XML* Data into *R* Using Schema

The theme of this chapter is to discuss different approaches to convert *XML* content into *R* objects. While we have explored different ways to do this, they all involve us writing code to interpret the *XML* content, be it a tree or streaming events. We typically write code based on one or a few sample documents. Our hope is that these generalize to a collection of similar documents. We try to identify patterns across a class of related documents and have the code read any instance of that class. A quite different approach is to use an actual general description of the content of the class of *XML* documents. We can programmatically generate *R* code that will read instances of this class of *XML* documents and convert the content to *R* objects.

XML schema are used to describe the structure of a class of related *XML* documents. Schema are themselves written in *XML*. Therefore, we can read a schema into *R* and make sense of its descriptions of the other *XML* documents they describe. We map these descriptions to *R* objects and then have *R* code that understands these objects. This code creates specialized *R* functions and data structures to read *XML* documents that are compatible with the schema.

Schema can identify optional elements or attributes in an *XML* tree. They can also specify default values for optional elements that we can use to fill in elements, cells or slots in *R* objects.

The point of this approach is to take out the guess work and human effort to identify the patterns across documents. Instead, we use explicit descriptions of these patterns. We go one step further than working manually with these descriptions. We programmatically process them and create the code to read many *XML* documents. This is what the XMLSchema package [6] does. When we have a schema, this can create more robust code without any human effort. Even if we do not have a schema, we can create one ourselves. We discuss this approach in Chapter 14 and use it in, for example, the implementation of *SOAP* clients in *R* to access rich Web services in Chapter 12.

We now return to processing *XML* content, but this time looking at event-driven parsing.

5.7 Element Handler Functions

Whether we use *XPath* or recursively traverse the tree directly with *R* code, the key concept is that we are traversing a tree. *XPath* queries do this in *C* code and so tend to be much more rapid. However, we frequently have to iterate over the resulting nodes in *R* and also make multiple *XPath* queries and so traverse the tree multiple times. This is typically significantly faster than iterating recursively over *all* of the nodes in the tree with *R* functions such as xmlSApply(). The fact that we are traversing the tree suggests another possible approach. We can iterate over the entire tree in *C* (rather than with

R code) and collect the information we want from the nodes as we process each node in the tree. The xmlParse() function allows us to do this. Instead of just asking for the parsed tree, we can pass a list of *R* functions via the *handlers* parameter to xmlParse(). If we provide a value for this *handlers* parameter, xmlParse() uses its elements to process the individual nodes as it (xmlParse()) makes a single pass over all of the nodes in the tree. The functions in our *handler*'s list can extract information from each of the relevant nodes and combine and store the information in a "central" location. When xmlParse() has finished traversing all of the nodes in the tree, we can pick up this information as an *R* object constructed by the *handler* functions.

The *handlers* list must contain named elements that either match a node name or correspond to generic node names such as `.startElement`, `.comment`, `.text` and so on. This allows us to provide specific functions for processing particular nodes based on their name, while also providing catch-all functions that can process all nodes of a particular type, e.g., generic nodes, text nodes, comments.

If the *handlers* argument is specified, the *XML* parser consults it each time it attempts to traverse a new node in the tree:

- It looks at the name of the *XML* element and searches for an element in the *handlers* list with this name. If it finds an entry, it calls that function and passes it the *XML* node as its primary argument. Then, it takes the return value from this call and, if it is non-`NULL`, it adds that value to the tree. If it is `NULL`, it drops that node from the tree.
- If there is no matching element in the *handlers* list, then the parser looks for a general function in the *handlers* list for handling the particular type of *XML* node and uses that, if it exists. For example, for a text node it will look for a function named text() in the list of handler functions. The association between node type and function name in the *handlers* list is given in Table 5.1.
- Finally, for those nodes that do not have a matching handler (of either of the two types described above), the *DOM* parser proceeds as usual.

While we will use the *handler* functions to collect the content from the tree, the functions can return arbitrary objects which are then combined into a hierarchical structure that mimics the tree of nodes. This allows us to create our own tree rather than the *C*-level tree or the simple *R* tree of nodes. Next we provide an example for the earthquake data from the USGS introduced in Section 3.2.

Table 5.1: General *DOM* Handler Names

Node Type	**Example**	**Function name**
XML element	`<node>`	startElement
Text node	`Simple text inside a node`	text
Comment node	`<!-- a comment -->`	comment
`<CDATA>` node	`<[CDATA[ literal text ]]>`	cdata
Processing instruction	`<?R library(XML)?>`	processingInstruction
XML namespace	`xmlns:r="http://www.r-proj..."`	namespace
Entity reference	`&gt;`	entity

This table describes the different elements we can specify in the list of functions passed to xmlTreeParse() or htmlTreeParse() via the handlers parameter. These functions respond to the different types of nodes the parser encounters in the tree/DOM as it traverses the tree after parsing it. The startElement() function will be called when processing a generic `<xml>` node. However, we can specify a more specific function to handle all nodes with a particular name by adding an entry to the list of handler functions with the name of the target nodes.

Example 5-10 Reading Earthquake Data with Handler Functions

Recall that the data are organized as a collection of <event> nodes within a root node named <merge>. Each <event> node is of the form

```
<event id="71880980" network-code="NC"
     time-stamp="2012/11/12_16:25:49 " version="0">
  <param name="latitude" value="38.7953"/>
  <param name="longitude" value="-122.7520"/>
  <param name="depth" value="1.7"/>
  <param name="magnitude" value="0.9"/>
  <param name="magnitude-type-ext"
         value="Mcd = coda duration magnitude"/>
  <param name="num-stations-mag" value="4"/>
  <param name="stand-mag-error" value="0.0"/>
    ...
</event>
```

We have attributes giving the time the earthquake was recorded, the network on which it was recorded and a unique identifier for the event. Details describing the event, such as where it occurred, its depth, magnitude and metadata about the details are all provided in <param> nodes with a *name* and *value* attribute.

Suppose we want to create a data frame with variables corresponding to the parameter names latitude, longitude, depth, magnitude, and rms-error. We can do this quite easily with *XPath*. For each of the variable names, we find all the <param> nodes with a *name* attribute that matches that variable name. We then extract the value of the corresponding *value* attributes. We can do all of this with

```
doc = xmlParse("merged_catalog.xml")
varNames = c("latitude", "longitude", "depth",
             "magnitude", "rms-error")
values = lapply(varNames,
                function(var) {
                   xp = sprintf("//param[@name = '%s']", var)
                   xpathSApply(doc, xp, xmlGetAttr, "value")
                })
names(values) = varNames
```

If some <event> nodes do not have a <param> for a given variable, we will end up with the value for the different variables misaligned across the events/observations. See Section 5.4 for a discussion of this issue.

Let's consider how we can do this with our *handlers* parameter for xmlParse(). We want to specify a function that will process each of the <param> nodes. That function will get the value of the *name* attribute of that node. If it is one of the variables we want to collect, it will add the value of the *value* attribute to an *R* vector for that variable. We can write our function as

```
paramFun = function(node)
{
  var = xmlGetAttr(node, "name")
  if(var %in% varNames)
     values[[var]] <<- c(values[[var]], xmlGetAttr(node, "value"))
}
```

These handler functions typically use our familiar node manipulation functions such as xmlGetAttr() to process the internal node object they are passed. Note that we use nonlocal assignment to update a variable across calls to this function. This is the variable `values`. We have to create it before we call this function. We can create a list with an empty vector for each variable:

```
values = structure(replicate(length(varNames),
                             character(), simplify = FALSE),
                   names = varNames)
```

Note also that we concatenate the new value to the existing vector, which will be very inefficient, but we will return to these issues.

Now that we have initialized the `values` object in which we will store the result and defined our sole handler function, we can parse the document and process the nodes with

```
xmlParse("merged_catalog.xml", handlers = list(param = paramFun))
```

When this returns, we can examine `values` with

```
sapply(values, length)
```

```
 latitude longitude     depth magnitude rms-error
     1291      1291      1291      1291      1277
```

We see that this now contains an observation for each event in the *XML* document. The `rms-error` element has fewer observations. This means that not all <*event*> nodes have a <*param*> node for this detail. This also means our *XPath* approach needs to be modified as we suggested in Section 5.4. We can however adjust our strategy here relatively easily to a) handle the missing values in some <*event*> nodes, and b) also make our code more efficient.

Firstly, we want to avoid concatenating the current value from the <*param*> node being processed to the end of a vector. This causes *R* to create a copy of the old vector with one extra element and then to populate that. This is a very expensive idiom in *R* generally. Instead, we would like to pre-allocate a vector of the correct length, or at least a guess and enlarge or shrink it as necessary. Unfortunately, we do not know the number of <*event*> nodes in our document. We can parse the document and then query this with `xmlSize(xmlRoot(doc))`. However, we want to parse and traverse the tree in one step. We can still use this code, but rather than parsing the document first, we can specify a handler function for our root node: <*merge*>. We can create the pre-allocated version of `values` in this function. We define it as

```
mergeFun = function(node)
{
  num = xmlSize(node)
  values <<- structure(replicate(length(varNames),
                                 rep(NA_character_, num),
                                 simplify = FALSE),
                       names = varNames)
  counter <<- 0
}
```

This assumes all of the children of <*merge*> are <*event*> nodes, but that is merely a detail we can easily fix. Our handler function then assigns our list of template vectors to a nonlocal variable `values`. We need to create this before we run the code so that mergeFun() can assign to it.

We also create another variable `counter` which we will use to identify to which row/position we are currently adding. Each time we process an <event> node, we will increment this. When we insert the value for any of the <param> nodes, we will use this counter to specify the position in our vector. We can define the handler function for <event> as

```
eventFun = function(node)
              counter <<- counter + 1L
```

We do not need to look at the node itself. If we wanted to collect the time-stamp or network information, we can store those also at this point. Again, we need to create this global variable `counter` before eventFun() or mergeFun() is called and tries to assign to it.

We want to change our function paramFun() so that instead of concatenating the result, it uses `counter` to insert the value at the appropriate position. This is easily done with

```
paramFun = function(node)
{
  var = xmlGetAttr(node, "name")
  if(var %in% varNames)
     values[[var]][counter] <<- xmlGetAttr(node, "value")
}
```

With these three handler functions defined, we can now pass them to xmlParse() via the *handlers* argument. There is one thing we have to specify, however. By default, xmlParse() processes the children of a node before it processes the node itself. Therefore, the handler for the merge() node will not be called until all of the <event> nodes have been processed and this involves processing all of the <param> nodes in each <event> node first. We need to change the order of evaluation so that the node is processed by the handler functions before the children. We indicate this via the *parentFirst* parameter:

```
xmlParse("merged_catalog.xml", parentFirst = TRUE,
         handlers = list(param = paramFun, merge = mergeFun,
                         event = eventFun))
```

We can now examine the contents of `values`. We can turn this into a data frame and convert the variables from strings to numbers and factors as appropriate, e.g.,

```
values = data.frame(lapply(values, as.numeric))
summary(values)
```

```
    latitude        longitude          depth
 Min.   :-59.35   Min.   :-178.4   Min.   :  0.00
 1st Qu.: 34.34   1st Qu.:-142.1   1st Qu.:  3.10
 Median : 38.79   Median :-121.7   Median :  8.10
 Mean   : 39.63   Mean   :-115.2   Mean   : 19.26
 3rd Qu.: 48.22   3rd Qu.:-116.8   3rd Qu.: 14.80
 Max.   : 67.12   Max.   : 179.8   Max.   :635.10

   magnitude        rms.error
 Min.   :-0.500   Min.   :0.0000
 1st Qu.: 0.900   1st Qu.:0.0000
 Median : 1.400   Median :0.1550
 Mean   : 1.678   Mean   :0.2901
```

```
 3rd Qu.: 2.100   3rd Qu.:0.4600
 Max.   : 6.800   Max.   :3.5200
                  NA's   :11
```

We see the 11 NA values and all of the variables have values that appear to be sensible.

One final issue we have to deal with is avoiding the global variables values and counter. We show how to do this in the next example.

Example 5-11 Extracting Earthquake Information Using Handler Functions with Closures

A much better approach to Example 5-10 (page 154) is to make the values and counter variables nonglobal, but shared across the three functions: mergeFun(), eventFun() and paramFun(). We can do this by defining a generator function that both defines these three handler functions and defines the shared variables within its body, i.e., with

```
quakeHandlers =
function()
{
  counter = 0
  values = NULL
  paramFun = function(node) {
    var = xmlGetAttr(node, "name")
    if(var %in% varNames)
       values[[var]][counter] <<- xmlGetAttr(node, "value")
  }

  eventFun = function(node) counter <<- counter + 1L

  mergeFun = function(node) {
    num = xmlSize(node)
    values <<- structure(replicate(length(varNames),
                                   rep(NA_character_, num),
                                   simplify = FALSE),
                         names = varNames)
    counter <<- 0
  }

  list(event = eventFun, merge = mergeFun, param = paramFun,
       .result = function()
                   data.frame(lapply(values, as.numeric)))
}
```

When we call this function, we obtain a list of functions. We can pass these directly to xmlParse(), as in

```
h = quakeHandlers()
xmlParse("merged_catalog.xml", handlers = h)
```

We can then get the data frame with h$.result(). If we call quakeHandlers() again, we get a different list of functions. They behave the same way, but have their own variables values and

`counter`. The two lists of functions operate independently of each other and each other's shared variables. Hence, we have removed the troublesome global variables and at the same time made creating and passing the event handlers easier.

This approach of using *handlers* functions is not likely to be as efficient as even multiple *XPath* queries over the entire tree. Indeed, comparing the two approaches for this earthquake data with only 1291 nodes, the handler function approach is about 10 times slower than the *XPath* approach. One reason for this is that we are processing individual nodes and making calls to functions such as xmlName(), xmlChildren(), xmlGetAttr(). These are not vectorized and involve a lot of interpreted *R* code. If calling *R* functions were significantly faster and the code itself were faster, e.g., compiled to native machine code or byte-code, then this approach might be faster than or competitive with an *XPath* approach as we can avoid repeated traversal of the entire tree. (Temple Lang is working on using LLVM, the Low Level Virtual Machine, to compile *R* code to machine code and, in some simple cases, has seen dramatic speedup. Similarly, the byte-code compiler already in *R* can often improve the performance of code by a factor three or four.) However, if we can compile *R* code generally, the speed of traversing a *DOM* tree in *R* code may also be significantly faster and competitive with the *XPath* or *handlers* functions approach.

If the function handler mechanism is slow, why do we include a description of it? As we mentioned, it may get more competitive with compilation of *R* code. However, it is conceptually a different and important approach and it serves as a good introduction to a mechanism for parsing very large *XML* documents for which we cannot keep the entire *DOM* in memory.

5.8 *SAX*: Simple API for *XML*

We have now seen how to parse an *XML* document using the *DOM* approach and to provide functions that are called asynchronously, or, as needed, to convert nodes. In this section, we will use the same idea of providing handler functions, but in a slightly different way. The Simple API for *XML*, commonly known as *SAX*, is an alternative parsing model that differs from *DOM* parsing. Unlike the *DOM* approach, the *SAX* parser never creates a tree or even an *XML* node. As the *SAX* parser reads content, it converts the bytes into tokens such as the start of an *XML* node (e.g., `<lender id="14232">`), the close of a node (`</lender>`), an entire processing instruction (`<?xsl-stylesheet html.xsl?>`, a comment (`<!-- -->`), or an entity (`&lt;`). As it encounters these different low-level tokens in the *XML* document, it generates events and invokes our handler/callback functions. These can then construct *R* objects that contain the data of interest. They can create a tree or any data structure for that matter, but critically, the parser does not create the tree.

SAX works in a linear manner on the *XML* stream, reading tokens from that stream until it finds enough to constitute an event. This leads to one of the biggest differences between the *DOM* and *SAX* models which is that it works top-down. In the *DOM* model, the parser collects all of the child nodes and their children and so on and then processes each node. When our handler function is called, it has access to the node and its descendants. The handler functions can then manipulate the entire node and its subcontents as a self-contained, meaningful unit. In the *SAX* model, however, we get information about the start of the parent node before we see anything about its subnodes. Our handler function is told of the start of a node, but it cannot process the child nodes. It must leave processing these children to calls to other handler functions. This means that we cannot transform a node into an *R* object in a single handler function since we do not have access to all the node's

descendants. Instead, we can often use the start of the parent node to create an empty or default object and then fill it in as we encounter the children nodes later in the *XML* stream, in separate calls to other handler functions. Only when we see the event that announces the closing of the parent node can we finalize the construction of the object corresponding to the complete node. In this way, the *SAX* model encourages a very incremental construction approach and typically one that involves sharing state across the callback functions to remember what object is currently being constructed. This is somewhat similar to the handler example in Example 5-11 (page 157). There we used one handler function to create the empty data frame, another to update a counter and a handler for the `<param>` nodes to populate cells/elements of the data frame.

The primary advantage of *SAX* is memory efficiency. The *SAX* parser does not incur the penalty of having both a tree and the target data structure in memory simultaneously. However, this typically comes at the expense of more complexity in the callbacks than one would have in the *DOM* processing. We will see in Section 5.10 that we can reduce this complexity by combining *SAX* with local *DOM* parsing and building nodes that we can manipulate as entire units.

We use the *R* function xmlEventParse() to implement *SAX* parsing. This function handles the *XML* input source in the same way that xmlParse() does, by assuming it is either a file name (either compressed or not), a remote *URL*, or a string containing the *XML*. As one might expect, the main difference between the functions is that, to be useful, you must supply callbacks to handle the different *SAX* events. As in *DOM* parsing, these functions are provided as a named list of functions via the *handlers* argument. The names of the list's elements correspond to the *SAX* event types, which are listed in Table 5.2.

Table 5.2: Event Handlers Available for *SAX* Parsing

Example	**Function name**
`<node att1="value" att2="...">`	startElement
`</node>`	endElement
`some text`	text
`<!-- a comment .. -->`	comment
`<?R library(XML) ?>`	processingInstruction
`%lt;`	externalEntity
`<!ENTITY % lt '<'>`	entityDeclaration

This table lists the different events that can arise in the SAX parser along with the names of the elements of the handlers list that are invoked to respond to such an event.

We will take a look at a simple and familiar example to show the basics of *SAX* parsing.

Example 5-12 Extracting Exchange Rates via SAX Parsing

We return to the daily euro exchange rates from the European Central Bank's *XML* files that we saw in Example 2-3 (page 33). A snippet is shown below. Recall that the data are provided in the attribute values of nested `<Cube>` elements. One outer `<Cube>` contains the entire set of exchange rates. This element has a separate `<Cube>` element for each day in the dataset, representing the time/date dimension. These `<Cube>` nodes have a *time* attribute that gives the date. Within each of these "time" elements, there is another set of `<Cube>` nodes, one for each currency. These innermost `<Cube>` nodes have two attributes: *currency*, which has a three-letter abbreviation for the particular

currency, and *rate*, which contains the exchange rate for that currency relative to the euro. The data look like

```
<Cube>
 <Cube time="2006-10-06">
  <Cube currency="USD" rate="1.2664"/>
   ...
 </Cube>
 <Cube time="2006-10-05">
  <Cube currency="USD" rate="1.2721"/>
   ...
 </Cube>
  ...
</Cube>
```

Our goal is to end up with a vector of dates (the *time* attributes) and the exchange-rate values for a subset of one or more of the currencies. We want the code to allow us to indicate which currencies to collect so that we can skip those that are not of interest. Suppose we want the rates for the US and New Zealand dollars (USD and NZD, respectively). We want to end up with a list with two *numeric* vectors, each of which contains the exchange rates for that currency. We also want a vector of the dates corresponding to those exchange rates.

As we encounter each <*Cube*> node with a *time* attribute, we will append that value to the end of the `time` vector. When we encounter a <*Cube*> node for one of the currencies, we will append the value of the *rate* attribute to the appropriate vector. Before we start, we need to first create *R* variables to store the values. We create a `rates` list with *numeric* vectors for each currency, and a `times` vector to store the date as it is encountered in the attributes of the <*Cube*> subelements.

Obviously we need a place so that all of the different callback/handler functions can access and update these variables. Essentially, we need to be able to make these shared objects that are available to the different callback functions and have any changes these functions make to the objects be available to subsequent calls. We can do this with closures as we did in Example 5-11 (page 157). We define a function that defines and returns a list of handler functions and that defines variables that these handler functions share and can update. We do this with

```
saxHandlers =
function(currencies = c("USD","NZD"))
{
  rates = vector("list", length(currencies))
  names(rates) = currencies
  times = numeric()
  day = 0

  startElement = function(name, attrs) {
      ...  # to be defined
  }

  list(startElement = startElement,
       rateData = function()
                    list(times = times, rates = rates))
}
```

Here the `day` variable is used as a counter to keep track of where to add the next *time* value and exchange rate values. We use this when appending the values to the `times` vector and the individual elements of `rates`.

We will use the rateData() function (in the list we return) to access the results, i.e., the `times` and `rates` variables.

Note that the currencies to collect are specified by the caller of the saxHandlers() function. We have specified defaults, but we can collect different currencies with different collections of handler functions created with different calls to this function. We will develop the code for the startElement() function next.

The important part of our saxHandlers() function is the startElement() function. This function is used to process any node in the *XML* document since it is named `startElement`. We can return this function in the list with the name `Cube` to apply only to <*Cube*> nodes. However, since there are only <*Cube*> nodes in our *XML* document, it will see all of them. Moreover, this function needs to handle the different <*Cube*> nodes differently. The startElement() function is called with both the name of the node and the vector of attributes for the *XML* node. It can ignore the top-level <*Cube*> node, i.e., those that have no attributes. It can also ignore any node not named <*Cube*>, should they be in the document. When the <*Cube*> node has a *time* attribute, we want to increment `day` and append the value of the *time* attribute to the `times` vector. When the attributes contain a *currency* element, we will add the exchange rate to the appropriate element of `rates`. We can implement all of this with

```
startElement = function(name, attrs) {
  if (name != "Cube")
     return(NULL)

  if ("time" %in% names(attrs)) {
    day <<- day + 1
    times[day] <<- attrs["time"]
    return(TRUE)
  }

  if ("currency" %in% names(attrs) &&
        attrs["currency"] %in% currencies)
    rates[[attrs["currency"]]][day] <<- attrs["rate"]

  TRUE
}
```

Since we define startElement() within the body of the saxHandlers() function above, rather than as a regular top-level function, it will have access to the variables `currencies`, `day`, `times` and `rates`. Note that we use the global assignment operator (`<<-`) to make changes in these variables.

Now we can use these handlers to extract the data.

```
h = saxHandlers()
xmlEventParse("../Data/eurofxref-hist.xml", handlers = h)
exchange.rate = h$rateData()
```

We can then transform the results from strings to numbers and dates and, for example, plot the values

```
rates = as.data.frame(lapply(exchange.rate$rates, as.numeric))
rates = cbind(rates,
              date = as.Date(exchange.rate$times, "%Y-%m-%d"))
matplot(rates$date, rates[, -ncol(rates)])
```

Since we do not need the handler functions after we have parsed the document and extracted the results by calling their rateData() element, we can extract the exchange rates in a single call

```
exchange.rate = xmlEventParse("../Data/eurofxref-hist.xml",
                              handlers = saxHandlers())$rateData()
```

In some cases, we do want to reuse the exact same handler functions and shared variables across different documents. In other cases, we may want to use the same functions, but re-initialize the shared variables. We do this by adding a `reset` element to the list of handler functions.

Another approach for implementing our startElement() handler function would be to keep a count of the depth of the nodes, i.e., how many start and end events we have processed for <*Cube*> nodes. When we see the first one, we would have depth 1 and ignore the node. For a second <*Cube*> node, we would collect the *time* attribute and increment the depth to 2. When we encounter another <*Cube*> node, we recognize that this must be a currency rate node since the depth is 2. We can increment depth for each node and then decrement it for each closing event for each <*Cube*> node. In this way, the depth tells us how to process the node without looking at the attributes. To implement this, we need an `endElement` in our list of handler functions which decrements a `depth` variable. We would implement this something like

```
saxHandlers =
function(currencies = c("USD","NZD"))
{
  rates = vector("list", length(currencies))
  names(rates) = currencies
  times = numeric()
  day = 0
  depth = 0

  startElement = function(name, attrs) {
    depth <<- depth + 1
    if (depth == 2) {
      day <<- day + 1
      times[day] <<- attrs["time"]
    } else if (depth == 3 &&
               attrs["currency"] %in% currencies)
      rates[[attrs["currency"]]][day] <<- attrs["rate"]
  }

  list(startElement = startElement,
       endElement = function(node) depth <<- depth - 1,
       rateData = function()
                    list(times = times, rates = rates))
}
```

For comparison, we provide an *XPath* approach to the problem. We pass over the tree several times—once for each currency for which we want the exchange rate, and again to get the date. The

XPath expression in the second case extracts those <*Cube*> nodes that have a parent <*Cube*> and an attribute *time*. Note that we do not check the value for the attribute, only that it exists.

```
doc = xmlParse("eurofxref-hist.xml")
dates = xpathApply(doc, "//x:Cube/x:Cube[@time]",
                   xmlGetAttr, "time", namespaces = "x")
currency = c("USD", "NZD")

currencies =
  as.data.frame(
     lapply(currency,
            function(cr) {
              xp = sprintf("//x:Cube/x:Cube[@currency = '%s']",
                           cr)
              as.numeric(xpathApply(doc, xp, xmlGetAttr, "rate",
                                    namespaces = "x"))
            }))

names(currencies) = currency
currencies = cbind(currencies,
                   dates = as.POSIXct(strptime(dates, "%Y-%m-%d")))

summary(currencies)
```

```
      USD              NZD             dates
 Min.   :0.8252   Min.   :1.641   Min.   :1999-01-04
 1st Qu.:0.9664   1st Qu.:1.868   1st Qu.:2001-04-02
 Median :1.1466   Median :1.955   Median :2003-07-12
 Mean   :1.1214   Mean   :1.954   Mean   :2003-07-08
 3rd Qu.:1.2614   3rd Qu.:2.044   3rd Qu.:2005-10-09
 Max.   :1.4895   Max.   :2.292   Max.   :2008-01-15
```

The *SAX* approach is less direct and more complicated than the equivalent process for the *DOM* model. This is because when we use the *DOM* approach we have the entire node and its subnodes within each callback and we can process all the information together. The *SAX* model requires us to construct the necessary information ourselves and store it so that we can process it when we have enough to make sense of it, i.e., in our examples the `day` and `depth` variables. Not only are we responsible for making sense of the information, we have the additional task of building the information from the low-level pieces the *XML* parser hands us—across calls to different functions. We definitely have more work to do when using a *SAX* parser. However, what this gives us is the control over what intermediate information is created. When we have to be concerned with the potential for a *DOM* parser creating an excessive amount of the tree that we will never use, we can assume control and use *SAX*. Of course, if we are reading small data sets, it is easier to program using the *DOM* approach. If we need to read a very large data set that will exceed the capacity of the *DOM* parser, we will need a *SAX* parser. Hence, we have this difficult trade-off of whether we implement both approaches and use them on different inputs depending on their (expected) size, or do we just implement a single, complicated parser? Unfortunately, there is no good, general answer to this problem. It will depend on

the circumstances in which you find yourself. Issues such as the cost of developing the code compared with maintaining and (re)testing it will be important. What we can say is that *SAX* parsing is not as complex as it may appear.

5.9 Managing State Across Handler Functions

When we discussed handler functions for both *SAX* and *DOM*, we used closures or lexical scoping to manage nonglobal variables that are shared across calls to the handler functions. These functions can query and modify these nonglobal variables and this is how they collect information from the *XML* document and make it available when the processing of all of the nodes is complete. Closures are not very complicated, but are extremely powerful and general and worth learning about when using *R*.

An alternative to *R*'s closure mechanism is *R*'s reference class mechanism. These classes are implemented using closures and so are a higher-level interface to the same concept. Reference classes have both fields and methods. The methods can access and update the values of the fields. We can use any of these methods as handler functions. We will use reference class methods in a very simple example of *SAX* parsing.

Example 5-13 Creating a Table of Counts of Nodes with a SAX Parser Using Reference Classes

We will implement a simple *SAX* parser that counts the number of occurrences of each node name in an *XML* document. To do this, we use an integer vector to store the counts. Each time we see the start of a node, we add one to the count in this vector for that node name. We can define a function for the `startElement` handler to process each start of node event.

```
openNode = function(name, attrs) {
              counts[name] <<- if(name %in% names(counts))
                                 counts[name] + 1L
                               else
                                 1L
           }
```

Note that this function refers to and updates the nonlocal variable `counts`. We will define this in our reference class along with methods for the start of a node (`open`) and for processing text nodes (`text`). We define the class with

```
mySAXParser = setRefClass("SAXParser",
                          fields = list( counts = "integer"),
                          methods = list(open = openNode,
                                         text = function(...)
                                                  open(".text")))
```

The variable `mySAXParser` is a generator object that we can use to create instances of this class:

```
h = mySAXParser$new()
```

We can now use the `open` and `text` methods in a call to xmlEventParse() with

```
xmlEventParse("merged_catalog.xml",
              handlers = list(startElement = h$open,
                              text = h$text))
```

When this finishes parsing the *XML* document, we can access the `counts` vector with

```
h$counts
```

```
    merge      .text      event      param duplicate
        1      30948       1267      28366        24
```

We can easily use closures to do the same thing. Reference classes do have some advantages however, e.g., direct and structured access to the fields and type specification for the fields.

5.9.1 Using State Objects

In the case of xmlEventParse(), we can use a very different approach to managing and updating state across calls to the handler functions. We can use xmlEventParse()'s *state* parameter to supply an *R* object that is passed in each call to any of the handler functions. These handler functions can access the contents of this object, but can also modify it and the return the updated state object. The xmlEventParse() function then stores this returned object and passes it as the new value of *state* when calling other handler functions. The *state* object is then updated in the handler functions and the final version is returned by xmlEventParse(). This is much less general than closures and reference classes, but can be useful, especially in a language such as S-Plus which does not have closures or reference classes.

Example 5-14 Creating a Table of Counts of Nodes with a SAX Parser Using a State Object

We will look at how we can implement our simple node name counter using a state object. We define our openNode() function in much the same way as for use in a reference class

```
openNode = function(name, attrs, .state) {
             .state[name] = if(name %in% names(.state))
                               .state[name] + 1L
                            else
                               1L
             .state
           }
```

The differences are: we have an additional parameter (*.state*); we use local assignment rather than <<-; and we return the updated state object. We can use this as follows:

```
xmlEventParse("merged_catalog.xml",
              handlers =
                list(startElement = openNode,
                     text = function(x, .state)
                              openNode(".text", , .state)),
              state = integer())
```

The result is the same as before

```
    merge      .text      event      param duplicate
        1      30948       1267      28366        24
```

The state parameter in each of the handler functions must be named *.state*. This is an unnecessary restriction but allows the functions to have different numbers of additional optional arguments.

5.10 Higher-level SAX: Branches

SAX avoids the creation of the entire *XML* tree and having all of the tree's nodes in memory simultaneously. However, the resulting computational model is more complex because it is much easier to work with nodes and their children and descendants rather than start and end events, where we must programmatically maintain the state between function calls to store the different aspects of the entire node, e.g., its name, attributes and all of its descendants. It would be much more convenient to be able to instruct the *SAX* parser to report start and end element events, etc. as usual, but to build entire nodes for those in which we are specifically interested. In other words, as the *SAX* parser is processing *XML* content, it would call the functions supplied for regular *SAX*-events, but when it encountered the start of any of the nodes that you want to deal with as a regular complete node with children and their children, it would continue reading the *XML* content and build the entire node and then call our handler function with the fully constructed subtree. This would give the best of both worlds: a fast, efficient parser that works with complete nodes and not low-level events.[2] These nodes are called "branches" in the *SAX* parser in the XML package. The idea is also present in *XML* parsers for other languages, e.g., twigs in the *PERL* module XML::Twig by Michel Rodriguez [3].

It is easy to use branches via the xmlEventParse() function in the XML package. We specify handlers for *SAX* events in the usual way via the *handlers* parameter. However, we can also specify a named list of multiple functions to process nodes corresponding to the different named elements in the list passed via the *branches* parameter. When the *SAX* parser encounters the start of a node, it compares the name of that *XML* element to the names of the elements in the list of branch functions. If it finds a match, the parser then continues reading the *SAX* content until it finds the end of that node and creates the native/*C*-level node and children. Then it invokes the matching branch function, passing it the `XMLInternalNode`. The branch function can do whatever it wants with the node, extracting information from it using node accessor functions (e.g., xmlName(), xmlAttrs(), xmlChildren(), xmlApply()) or *XPath* queries via xpathApply() or getNodeSet(). When the function has returned, the node is typically garbage collected and disappears unless it has been assigned to some variable.

As an example of using branches, consider the problem of analyzing information from Wikipedia pages. The entire collection of pages with revision information can be downloaded from `http://download.wikimedia.org/enwiki/latest/enwiki-latest-pages-meta-current.xml.bz2`. This is a very large document and so parsing with *SAX* makes sense. Note that this is a **bz2** compressed file and xmlEventParse() cannot read this directly so we have to uncompress it before parsing the document. We will see an approach to avoid this in Section 5.12.

Example 5-15 Extracting Revision History from Wikipedia Pages Using SAX Branches

The structure of the Wikipedia *XML* document is quite simple and consists of a node that provides some details about the particular dump and then a sequence of pages. Each page has one or more `<revision>` nodes, each of which has an identifier (`<id>`), time stamp (`<timestamp>`), author

[2] This is an idea that Seth Falcon raised during a visit of DTL to Robert Gentleman's lab at the Fred Hutchinson Cancer Research Center in Seattle, WA.

(<*contributor*> with an <*ip*> address or <*username*>), and a <*comment*>. The following gives an idea of the structure.

```
<mediawiki ... xml:lang="en">
 <siteinfo> ... </siteinfo>
 <page>
  <title>To his simplicity</title>
  <id>13826</id>
  <revision>
   <id>47116</id>
   <timestamp>2005-07-19T21:15:31Z</timestamp>
   <contributor>
    <ip>Khaldei</ip>
   </contributor>
   <comment>Created This Entry.</comment>
   <text xml:space="preserve"> ......  </text>
  </revision>
  <revision>
   <id>287657</id>
   <timestamp>2006-11-30T16:11:13Z</timestamp>
   <contributor>
    <username>TalBot</username>
    <id>5664</id>
   </contributor>
   <minor/>
   <comment>Robot: Removing from Category:
            Emily Dickinson Poems - T</comment>
   <text xml:space="preserve"> ....  </text>
  ....
  </revision>
 </page>
 <page> ....  </page>
</mediawiki>
```

Suppose we are interested in the revision history of the documents. That is, we want to explore the distribution of revisions, examine the time history of changes, and see when different contributors were active and on what topics. We can write *SAX* handler functions to process the start and end of each <*page*> and <*revision*> node and gather the information within each of their subnodes into an *R* object.

However, we can also use a branch function. We can write a branch function for the <*page*> nodes. This function would extract the page title and identifier and then iterate over the <*revision*> nodes to extract their information. The entries can then be put into rows of a data frame, repeating the page id and title for each revision. First, we define the structure of the data frame, leaving it with 0 rows.

```
N = 0
pages = data.frame(page = character(N), title = character(N),
                   rev.id = character(N), timestamp = numeric(N),
                   contributor.ip = character(N),
                   contributor.id = character(N),
```

```
                    contributor.name = character(N),
                    stringsAsFactors = FALSE)
```

The function to manipulate the <*page*> node is quite simple and defined as follows:

```
page = function(node) {
          page.id = xmlValue(node[["id"]])
          title = xmlValue(node[["title"]])
          lapply(node[names(node) == "revision"],
                 addRevision, page.id, title)
       }
```

The page() function calls the function addRevision() which does the actual updating to the data frame pages. The addRevision() function uses a "global" variable n to determine which row of the data frame pages is to be updated. It increments this variable each time it is called.

```
addRevision =
function(revNode, page.id, page.title)
{
  n <<- n + 1
  dflt = as.POSIXct(xmlValue(revNode[["timestamp"]]),
                    "%Y-%m-%dT%H:%M:%SZ")
  pages[n, 1:4] <<- xmlValue(page.id, page.title,
                             revNode[["id"]], dflt)

  who = revNode[["contributor"]]
  if(length(who[["username"]]))
     pages[n, c("contributor.name", "contributor.id")] <<-
           c(xmlValue(who[["username"]]), xmlValue(who[["id"]]))
  else
     pages[n, "contributor.ip"] <<- xmlValue(who)
}
```

The last thing to do is to associate the function addRevision() with the variables pages and n. We can do this with a closure in which we define addRevision() within a function that defines these variables. An alternative is to create an environment for the addRevision() function and create and initialize the variables pagesand n in that environment, i.e.,

```
env = new.env()
environment(addRevision) = env
env$pages = pages
env$n = 0
```

This is the manual approach to defining functions within functions. We have defined the pages data frame in terms of the number of observations N. When this is 0, we are appending and re-allocating vectors each time. If we know the number ahead of time, we can pre-allocate the vectors. Alternatively, we can guess a suitable number and expand and contract the vectors as necessary. For example, we can determine a suitable number by pre-processing the document to find the number of <*revision*> nodes with, e.g., the shell command *grep*.

We can use more efficient approaches such as allocating, say, one million rows of the data frame and filling in the next unfilled one (as recorded by n). Then addRevision() can double this when it

runs out of available records. At the end of the document processing, it can then discard the over-allocated rows. To implement this, we change the value of N, the initial number of rows allocated, and add an endDocument handler function to clean up the over allocation, e.g.,

```
endDocument =  function() {
    pages <<- as.data.frame(lapply(pages, function(x)
                                          length(x) = n))
}
environment(endDocument) = env
```

This merely sets the length of each of the vectors to n, the number of observed nodes.

5.10.1 Nested Nodes and Branches

The *SAX* parser builds the node for a branch by processing the content from the start of a branch to the end of that node. What happens when branch nodes are nested, i.e., when a branch node has a node with the same name? For example, consider the exchange rate data from the European Central Bank again:

```
<Cube>
 <Cube time="2006-10-06">
  <Cube currency="USD" rate="1.2664"/>
  ...
 </Cube>
 <Cube time="2006-10-05">
  <Cube currency="USD" rate="1.2721"/>
   ...
 </Cube>
 ...
</Cube>
```

Here, the <*Cube*> node with the exchange rate is nested inside another <*Cube*> which is itself nested inside a <*Cube*>. Specifying a branch function for the <*Cube*> element would be ambiguous. The branch mechanism, as implemented, behaves greedily and will process the entire topmost node, e.g., the first Cube, and call the branch function with that node. This is essentially the entire document, and not what we want. It will not invoke the branch function for a subnode within a node being built as part of another branch function. Similarly, if we had defined branch functions for both the <*page*> and <*revision*> nodes in the Wikipedia example above, only the <*page*> function would be invoked. This is because the <*revision*> node is a subnode/descendant of the <*page*> node. This behavior is different from the handler functions for xmlTreeParse() which *do* work recursively.

5.10.2 Deferred Node Creation

There is one additional feature of branches that can make them even more efficient in certain scenarios. When you specify a branch function, each and every node with the corresponding name will

be constructed and passed as an argument in a call to the corresponding branch function. If we are only interested in a subset of the particular nodes, then we can avoid the construction of those nodes that are not of interest. For example, we may want to sample a subset of the nodes or examine the attributes of the node to determine whether or not to process it. The key idea is that we dynamically determine whether the node is of interest, rather than just based on its name. If we do want to process a node, the *SAX* parser needs to be instructed to construct the entire node and, if not, to process it in the usual way (i.e., with regular *SAX* events and handler functions). We can accomplish this with a handler function—not a branch function—for the event that identifies the opening/start of the particular element, where that handler function determines whether it is interested in having the entire node. If we are interested in the entire node, the function returns another function of (*S3*) class `SAXBranchFunction`. If we do not want to process the node, the function returns any other value. The *SAX* parser recognizes this `SAXBranchFunction` class of function as special and immediately switches to building the entire subtree for this node. When it completes the building of the node, the parser invokes the `SAXBranchFunction` function that was returned by the call to the most recent handler function, passing it the entire node. This function can then extract the content it wants. We will look at an example of this functionality.

Example 5-16 Extracting a Random Sample of Revisions to Wikipedia Pages

Suppose we want to randomly sample <*page*> nodes for the Wikipedia corpus. We can relatively easily find (or guess) the number of pages and so generate the indices of the pages to be in our sample:

```
NumPages = system("wc -l '<page>'
                  enwikisource-pages-meta-history.xml")
which.pages = sample(n, 1:NumPages)
```

When we encounter the start of each <*page*> node, we can determine if this is one of the pages we want in our sample. If it is, we return a function that will process the resulting node; if not, we just ignore the page. We can implement this with a *handler* function for <*page*> as

```
page = function(name, attrs, ...)
{
  pageCounter <<- pageCounter + 1
  if(pageCounter %in% which.pages)
     processPageNode
}
```

Here we use a counter variable `pageCounter` to record how many <*page*> nodes we have seen so far. If the current page number is in our sample, we want to process it and return the function processPageNode(). That function is a regular branch function that manipulates the nodes, but is also identified as an object of class `SAXBranchFunction`:

```
class(processPageNode) = "SAXBranchFunction"
```

We can then use these functions together in a call to xmlEventParse().

```
xmlEventParse("wikipedia.xml", handlers = list(page = page))
```

Note that we are using the *handlers* argument here as we want our page() function to be called for the start of a <*page*> node. The processPageNode() is not listed in the *handlers* or *branches* arguments, but only enters the computations by being returned from the page() handler function.

We can, of course, use the attributes of the node in our start-element handler function to determine whether we want to build the entire node. For example, if Wikipedia <*page*> elements have the <*id*>

as an attribute rather than a child element, we can see if that is in a set of interest. Unfortunately, we cannot look at child nodes (such as the <id> node in the <page>) and go back to build the outer node. The streaming nature of the parser makes this very difficult and computationally expensive.

5.11 Accessing the Parser Context

Branch functions are, by default, invoked with just a single argument—the XMLInternalNode object. The different types of handler functions are called with different inputs, i.e., the name of the node and the attributes for start node events, a string for the text handlers, and so on. However, we can declare a branch or a handler function to be of class XMLParserContextFunction and then when it is called by the parser, it will be passed a reference to the *XML* parser context object as the first argument, with the other standard values provided as the second, third, ... arguments. We can make a function an XMLParserContextFunction object by explicitly setting its class or by calling the xmlParserContextFunction().

This parser context object is a native/C-level object and contains the state of the parser. This includes information about the low-level handlers, the document being parsed, whether entities are expanded or not, and other details that we might want to query or even dynamically modify.

One of the primary uses of this context object, at present, is to be able to terminate the parser from within the *R* handler or branch function. One can call the *R* function xmlStopParser() passing it this context object as the first argument and the *SAX* parser will terminate without processing any more of the *XML* content and gracefully return control to the the caller, e.g., the *R* prompt. This avoids processing content that we will just ignore. It is useful when we can dynamically/programmatically determine that we are no longer interested in the remainder of the document. This approach is different from raising an error via a call to stop() within our handler function. That would cause the call to xmlEventParse() to propagate the error. In contrast, calling xmlStopParser() allows xmlEventParse() to return without an error.

Let's consider our example in which we sample <page> nodes in the Wikipedia edits document—Example 5-16. We use a handler function to determine if we want to process a particular <page> node, and if we do, we returned a SAXBranchFunction. When we have processed all of the <page> elements in our sample, there is no point in continuing to process the remainder of the document. We would like to call xmlStopParser(). We can do this by changing our page() handler function from that example. We add a new parameter *context* to the definition. Before determining whether to process the <page> node, we check to see if we should terminate parsing the document. Our new function is

```
page =
function(context, name, attrs, ...)
{
  pageCounter <<- pageCounter + 1
  if(pageCounter > max(which.pages)) xmlStopParser(context)
  if(pageCounter %in% which.pages) processPageNode
}
page = xmlParserContextFunction(page)
```

When we use this page() function as a handler function, it will terminate parsing the document when we have processed the `NumPages` in our sample.

5.12 Parsing *XML* Content from *R* Connections

The *SAX* parsing model is used when the resulting parsed *XML* tree would be too large to readily fit in memory. We avoid building the tree by consuming the *XML* content as it is encountered, extracting what we want, and then throwing the *XML* away. This is similar to reading from any stream of data and may remind some readers of connections in *R*. Connections are an abstraction of how we read (or write) streams of bytes and they allow us to treat data coming from a file, a string/buffer in memory, a Web request, output from another application/process all in the same way—merely as a stream of bytes. Connections allow us to read a small segment of the input and then return in subsequent calls to read more, continuing to read from the end of the previous position read. We can use an *R* connection in a call to xmlEventParse(). When the *XML* parser needs more content, it pulls more bytes from the connection. This gives us more flexibility about where the *XML* content comes from. This can be useful when dealing with large documents, which is typically the case when working with *SAX* parsing.

Consider the example where we were reading the history of Wikipedia edits to different pages. The *XML* document was made available as a **bzip2** file: `enwiki-latest-pages-meta-current.xml.bz2`. We can uncompress this outside of *R*, but that takes time and disk space. We can also read it into *R* and pass the entire document as a string to xmlEventParse(). However, for such a large document (many gigabytes) this would defeat the purpose of using *SAX* as we do not want to have two copies of the data in memory at any one time. Instead, we can create a connection that can read small segments of the file incrementally. We do this via the bzfile() function in *R*:

```
con = bzfile("enwiki-latest-pages-meta-current.xml.bz2")
```

We can then pass this connection object directly to xmlEventParse() as the first argument to read the document without previously uncompressing it. We can also use other types of connections with functions such as xzfile(), gzfile() (although the *XML* parsers can read these files directly), pipe(), fifo(), and socketConnection(). We can even use the RCurl package to make a request to a Web server and arrange to read the response as a connection object in *R*. Then we can pass this to xmlEventParse() and process the *XML* directly from the Web server. See the documentation and examples for the RCurl package for an example of this.

5.13 Comparing *XML* Parsing Techniques in *R*

In this chapter, we have seen several ways to process records in an *XML* file into *R* objects: *XPath*, the hybrid *DOM* approach using a list of *handler* functions to traverse the tree, *SAX*, and *SAX* with branch functions. In this section, we compare these various approaches using an example *XML* document that is an iTunes database. This is not a particularly large file for most people, containing a few thousand tracks.[3] The file is a generic property list (plist) file, as we discussed in Section 5.2.3. Each track is a collection of key-value pairs represented by a `<key>` node and a sibling value identified by its type, e.g., `<integer>`, `<date>`, `<string>`. Each

[3] The word "record" becomes confusing in this context!

track is contained within a <dict> node and the collection of tracks are themselves stored in a <dict> node. Further, the top-level information (e.g., version details, location of the library) is in yet another <dict> node which is under the <plist> root node. The *DTD* is available at http://www.apple.com/DTDs/PropertyList-1.0.dtd and is very succinct. We will not use the readKeyValueDB() function in this example to read the plist document. Instead, we will explicitly write the code for the different approaches so that we can contrast their implementations.

The following is a small segment of the document to illustrate the structure of the contents:

```
<plist version="1.0">
 <dict>
  <key>Major Version</key>
   <integer>1</integer>
  <key>Tracks</key>
  <dict>
   <key>11340</key>
   <dict>
    <key>Track ID</key>
    <integer>11340</integer>
    <key>Name</key>
    <string>Fighting in a Sack</string>
    <key>Artist</key>
    <string>The Shins</string>
    <key>Album</key>
    <string>Chutes Too Narrow</string>
    <key>Genre</key>
    <string>Alternative</string>
    <key>Kind</key>
    <string>MPEG audio file</string>
    <key>Size</key>
    <integer>3582070</integer>
    <key>Total Time</key>
    <integer>148062</integer>
    <key>Track Number</key>
    <integer>6</integer>
    <key>Year</key>
    <integer>2003</integer> ...
   </dict>
...
```

The quick summary of the comparison of parsing techniques is mostly as we would expect. *SAX* is fastest, but probably most complex to implement; using the standard *DOM* in this context is faster than the hybrid *DOM* approach. One reason for this is that in this example there is very little traversal of a relatively flat tree. There is not much cost in finding the nodes of interest and converting them.

5.13.1 The Standard DOM Approach

The critical observation that allows us to easily pull out the tracks is that they correspond to <dict> nodes within a <dict> node which is itself within the top-level <dict> node of the <plist> root node. In other words, all the track nodes are accessible via the *R* expression

```
xmlRoot(doc)[["dict"]][["dict"]]["dict"]
```

Similarly, we can use an *XPath* expression to access these nodes, `/plist/dict/dict/dict`.

Once we have these nodes, processing them into *R* objects representing the tracks involves writing a function to process a given track node and then applying this function to these nodes via lapply() . This function needs to extract the key name and value pairs and turn them into a list. The child nodes of <dict> include not only the <key> and corresponding value nodes (e.g., <integer>), but also the text nodes containing the white space between these "real" nodes. When processing the child nodes, we need to ignore these `XMLInternalTextNodes` nodes. We can do this by subsetting based on the *R* class of the nodes, i.e.,

```
els = xmlChildren(dt)[!xmlSApply(x, inherits,
                                 "XMLInternalTextNode")]
```

Having discarded these text nodes, we will have just the key and value pairs and so an even number of nodes. Then lapply() can get the values for the different type of value nodes. This is done via the function getNodeValue() which takes the node and fetches both its name and its contents, e.g., integer, real, string, and performs the appropriate conversion of the content to an *R* value.

```
getNodeValue =
function(node)
{
  val = xmlValue(node)
  switch(xmlName(node),
   # some integers are too large for an R integer, so use numeric.
         integer = as.numeric(val),
         date = as.POSIXct(strptime(val, "%Y-%m-%dT%H:%M:%S")),
         string = val, true = TRUE, false = FALSE,
         real = as.numeric(val), default = val)
}
```

Getting the names of the keys is even easier as there is no type conversion to do so we use xmlValue(). All together, the function getTrack() that processes a single track appears below:

```
getTrack =
function(x)
{
  els = xmlChildren(x)[ !xmlSApply(x, inherits,
                                   "XMLInternalTextNode") ]
  idx = seq(1, to = length(els) - 1, by = 2)
  track = lapply(els[ idx + 1 ], getNodeValue)
  names(track) = sapply(els[idx], xmlValue)
  class(track) = "iTunesTrackInfo"

  track
}
```

With these functions defined, we can get all our tracks using the commands:

```
doc = xmlParse(fileName)
r = xmlRoot(doc)
tracks = lapply(r[["dict"]][["dict"]]["dict"], getTrack)
```

5.13.2 The DOM Approach with Handler Functions

Next we turn to the hybrid *DOM* method for parsing, where we still parse the tree but we use handler functions to convert the <*dict*> nodes into *R* objects. As the parser processes each of the *XML* nodes in the tree, it consults the list of user-provided functions that can customize the conversion of the nodes. We will provide a handler function for the <*dict*> nodes and have it combine the key-value pairs into a track object in *R*. This conversion function must ensure that it is dealing with a track <*dict*> node and not one of the higher-level <*dict*> nodes. It can do this by verifying that there are two non-NULL parent nodes and that those parents are also <*dict*> nodes. Remember that within this *DOM*-handler function approach, we have the entire tree and so can navigate to the parent nodes. This is very different from what happens in *SAX* handler or branch functions where there is no tree and no parent node.

While we can have our <*dict*> handler function convert the key value nodes, we can also do this with separate handler functions for <*integer*>, <*real*>, <*string*>, <*true*>, and <*false*> nodes. To implement this, we need to construct a list of handler functions, where the name of each element in the list matches the name of the node it is intended to handle. The following code does the job with getTrack() being essentially identical to the one defined earlier.

```
createHandlers =
function()
{
  integer = function(x) as.numeric(xmlValue(x))

  date = function(x)
           as.POSIXct(strptime(xmlValue(x), "%Y-%m-%dT%H:%M:%S"))

  dict =
  function(node)
  {
         # verify that this dict corresponds to a track
     p = xmlParent(node)
     if (is.null(p) || xmlName(p) != "dict" ||
         is.null(p <- xmlParent(p)) || xmlName(p) != "dict")
       return(node)
     getTrack(node)
  }

  tracks = list()

  getTrack =
  function(x)
```

```
  {
    els = xmlChildren(x)[!xmlSApply(x, inherits,
                                    "XMLInternalTextNode")]
    idx = seq(1, to = length(els) - 1, by = 2)
    track = lapply(els[ idx + 1 ], getNodeValue)
    names(track) = sapply(els[idx], xmlValue)
    class(track) = "iTunesTrackInfo"

    tracks[[ length(tracks) + 1 ]] <<- track
    NULL
  }

  list(integer = integer, date = date,
       string = function(x) xmlValue(x),
       dict = dict, tracks = function() tracks)
}
```

We can then use this code as

```
h = createHandlers()
doc = xmlParse(fileName, handlers = h)
h$tracks()
```

We should note that appending the new track to the end of the `tracks` list is potentially somewhat expensive. We would like to preallocate the list, instead, with the correct number of tracks. However, we do not know this number ahead of time. Having this information as an attribute of the top-level `<dict>` node would be valuable. In the absence of this, we could preallocate the list and double its length as needed. We would need an additional variable to store the index of the next "real" position at which we add the track objects. In the *DOM* approach, we have the entire tree and so can determine the number of `<dict>` nodes corresponding to individual tracks using xmlSize(). To do this, we need to process the `<dict>` node that contains each of the track nodes before we process those track nodes. We can do this using the *parentFirst* argument of the xmlParse() function as we saw in Example 5-10 (page 154).

5.13.3 SAX

The *SAX* approach requires us to work at the level of raw, low-level events such as the opening of an *XML* node, the closing of a node, available text (possibly a sub-part of a text node), and so on. The basic strategy we employ here for the iTunes property list file is quite simple relative to *SAX* parsers for other more complicated formats. Here, we maintain a list of the tracks and as we end each `<dict>` node that ends a track element, we add the *R* object for that track to this list. We have a handler function for `<dict>` nodes which is responsible for creating an empty track object. We have another handler function for `endElement` which recognizes the closing of a `<dict>` node and appends the completed track object. We have other handler functions which handle the `<key>`, `<integer>`, `<string>`, ... nodes, and also the text nodes within these nodes. The `key` handler function merely sets a flag that indicates that we are processing a `<key>` node. The `text` handler then cumulates the actual value of the `<key>` node. When we encounter the end of the `<key>` node (in the `endElement` handler function), we cannot use the value immediately as the name of an

element in the track object. This is because we have not yet read the value associated with that key in the track. Instead, we assign the key identifier/string in the variable `key`.

Processing the value node associated with a <*key*> node is slightly more complex than that of the key since we can have different types of values other than simple strings. We may have <*integer*>, <*real*>, <*true*>, <*false*> and <*string*> nodes. For a general property list, we may also have <*array*>s, nested <*dict*> nodes, and general data vectors/arrays. When we encounter the start of one of these nodes, we re-initialize the variable in which we cumulate text. When the `text` handler is subsequently called, it will add its contents to this string. We can store the name of the node (e.g., integer or real) when we process its start tag as we need the name when we interpret the text content of the node. However, we only need the name of the node when we process the closing event for that node. The name of the node being closed is passed in the call to the `endElement` handler function. We can use this to convert the text using the locally-defined convertValue() function to get the appropriate *R* object. We then add this value to the track object, using the previously stored value for the <*key*> node as the name of the element.

So far, our handler functions are quite simple. We must combine contiguous text nodes by concatenating one with the previously stored values since the parser may process a large block of text in separate chunks or as separate events. When we encounter the start of a node, we must reset the value of the encountered text to `character(0)`. Also, in the end-of-node action, we must convert the contents of the text to an *R* value, be it the name of the key or the value corresponding to a key. For non-key nodes, the end of the node action must assign the key name and value pair to the current track. Now, we turn our attention to this outer layer of the track container.

When we encounter the end of a <*dict*> element for a track, we need to append that track to the list of existing tracks and then re-initialize the variable holding the track information to the empty list. Again, this is very straightforward. It becomes slightly more complex because we have three levels of <*dict*> nodes. What is important for us is that the <*dict*> node corresponding to a track is at the third level. We create a variable to count the number of open <*dict*> nodes. We increment this in our start-element action and decrement when we exit our end-element action for the <*dict*> node. Within that end-element action, we only append the track to the list of tracks if `dictLevel` is 3.

All of this amounts to the following code defining the handler/action functions.

```
saxTrackHandlers =
function()
{
  tracks = list()
  dictLevel = 0L
  key = NA
  value = character()
  track = list()

  text = function(val) value <<- paste(value, val, sep = "")

  startElement =
  function(name, attrs)
  {
    if (name %in% c("integer", "string", "date", "key"))
      value <<- character()

    if (name == "dict")
```

```
      dictLevel <<- dictLevel + 1L
  }

  convertValue =
  function(value, textType)
  {
   switch(textType,
          integer = as.numeric(value),
          string = value,
          date = as.POSIXct(strptime(value, "%Y-%m-%dT%H:%M:%S")),
          default = value)
  }

  endElement =
  function(name)
  {
    if (name %in% c("integer", "string", "date"))
      track[[key]] <<- convertValue(value, name)
    else if (name == "key")
      key <<- value
    else if(name == "dict" && dictLevel == 3) {
      class(track) = "iTunesTrackInfo"
      tracks[[ length(tracks) + 1]] <<- track
      track <<- list()
      dictLevel <<- 2
    }
  }

  list(startElement = startElement,
       endElement = endElement,
       text = text, tracks = function() tracks)
}
```

The we get the tracks with *SAX* as follows:

```
h = saxTrackHandlers()
xmlEventParse(fileName, handlers = h, addContext = FALSE)
h$tracks()
```

5.13.4 Timings

We can time the three different approaches and compare their relative efficiencies. We first time the standard *DOM* approach.

```
std.dom = replicate(10, system.time({
            doc = xmlParse(fileName)
            r = xmlRoot(doc)
```

```
            x = lapply(r[["dict"]][["dict"]]["dict"], getTrack)
          }))
```

We replicate the hybrid *DOM* approach and measure its time with

```
hyb.dom = replicate(10, system.time({
                          h = createHandlers()
                          doc = xmlParse(fileName, handlers = h)
                          x = h$tracks()
                        }))
```

Last, we measure the *SAX*-based approach with

```
sax = replicate(10, system.time({
                      h = saxTrackHandlers()
                      xmlEventParse(fileName, handlers = h,
                                    addContext = FALSE)
                      h$tracks()
                    }))
```

Comparing these, we see that *SAX* is fastest, the simple *DOM* approach is about two times slower and the node conversion *DOM* approach is slightly less than three times slower than *SAX*.

```
           SAX     DOM   Converter DOM
Mean Time 6.17  11.70           16.69
SD        0.21   0.02            0.58
```

5.13.5 SAX Branches

While we are investigating the efficiency of different approaches, let's push forward in an effort to see if we can gain some insight into more general approaches to the parsing problem which will help us identify good approaches initially. The *DOM* approach of *XPath* queries and node manipulation is typically simpler for most programmers than the more efficient *SAX* mechanism. What we would like is a compromise that falls somewhere along the continuum of these two dimensions: a slightly more efficient approach that is easier than *SAX*. The "obvious" answer is that it would be beneficial to take advantage of the fact that *SAX* does not create the tree and is relatively lightweight but to allow handler functions at the "entire node" level rather than at the open and close nodes, text chunks, etc. The hybrid *SAX* model which uses branches is precisely this. We can specify handlers for entire nodes via the *branches* parameter of xmlEventParse() and these functions will be called with the entire node. This allows us to use our getTrack() function from the standard *DOM* approach to process a node without building the entire tree; just the <*dict*> nodes for the tracks.

To implement this we write a simple handler function that processes the node and appends the newly constructed *R* track object to a list of tracks.

```
branchHandlers =
function()
{
  tracks = list()
  dict = function(node)
```

```
            tracks[[ length(tracks) + 1 ]] <<- getTrack(node)

  list(dict = dict, tracks = function() tracks)
}
```

We would then pass this branch handler function in the call to xmlEventParse().

```
h = branchHandlers()
xmlEventParse(fileName, handlers = NULL, branches = h)
h$tracks()
```

Note that we specify the regular *SAX* handlers as `NULL` as otherwise, the parser will invoke the standard handlers and become very slow.

Unfortunately, this approach does not work in this case as desired. The problem is that the *SAX* parser sees the top-level `<dict>` object and converts it to a regular *XML* node and passes that to our handler. This is essentially the entire tree (the primary child node of the `<plist>` root node), which means we do not get to see the individual `<dict>` track nodes.

The problem lies with the fact that we cannot differentiate between the track `<dict>` nodes and the other nodes with that same name. For the moment, let's change the name of the other `<dict>` nodes in our file to `<xdict>` and `<ydict>`. If this turns out to yield performance gains, then we can look into a mechanism for dealing with recognizing and separately processing nested nodes.

```
hyb.sax = replicate(10, system.time({
                            h = branchHandlers()
                            xmlEventParse("~/iTunes-modified.xml",
                                          handlers = NULL,
                                          branches = h)
                            h$tracks()
          }))
```

What we find is timings around 11.58 seconds which is only very slightly faster than the standard *DOM* approach.

5.14 Summary of Functions for Parsing *XML*

Most of the functions that we demonstrated in this chapter have been introduced in earlier chapters. The functions to parse an *XML* document and access and manipulate nodes in the document were introduced in Chapter 3 and we refer the reader to Section 3.10 for a summary of these functions (xmlName(), xmlAttrs(), xmlValue(), xmlSize(), xmlRoot(), xmlChildren(), xmlApply(), and xmlSApply()). When describing alternative parsing strategies, we introduced some additional parameters to xmlParse() that allow greater control over the parsing of a document, and we introduced other functions for parsing *XML* documents (e.g., xmlEventParse()). We summarize these functions below and provide an augmented description of the xmlParse() function that addresses its capabilities for transforming nodes in the tree/*DOM*.

xmlParse() Parse an *XML* document from a local file, a remote *URL*, or *XML* content that is already in *R* as a string (*asText*). By default, the function returns the parsed document as a tree of internal/*C*-level nodes. If we specify *useInternalNodes* as `FALSE`, the function converts the tree to *R* representations of the nodes (lists of lists). We can also use the function to traverse the

entire tree and process each node to extract and accumulate information of interest. We do this by specifying a list of node processing functions via the *handlers* parameter. As the function traverses each node in the *DOM*, it calls the appropriate function from the *handlers* list, passing it the node. The *handlers* list must contain named elements that either match a node name or correspond to generic node names, such as `.startElement`, `.comment`, and `.text`. We can also specify options to xmlParse() which control how the parser behaves. These are described in Table 3.1 in Chapter 3

xmlTreeParse(), htmlTreeParse() These functions are very similar to xmlParse() and htmlParse(), respectively, differing only in the default value for the *useInternalNodes* parameter. By default, these functions return *R* structures rather than internal *C*-level structures. As a result, we cannot use *XPath* queries on the resulting tree.

xmlEventParse() Parse an *XML* document using the low-level *SAX* event parsing approach. This function can read a local file (compressed or not), a remote *URL*, *XML* content that is already in *R* as a string (*asText*), or a connection. The key concept underlying this function is that we specify *R* functions (or *C* routines) that are invoked by the parser when it encounters the corresponding event, e.g., the opening of an *XML* node, the end of a node, a comment. Callback functions to handle the different *SAX* events are provided via the *handlers* parameter. The value for *handlers* is a named list where the list's element names correspond to the names of *XML* nodes and/or the names of *SAX* event types, e.g., `startElement`, `endElement`, and `text`. These functions will be passed with information about the event, e.g., the name and attributes for the start of a node, and just the name for the end/close of a node. Nodes are not passed to the functions since *SAX* does not build the nodes or subtrees. However, we can have the parser build nodes and subtrees for particular nodes within the document and pass these to *R* functions that we have specified via the *branches* parameter as a named list of functions. The element names in the list should correspond to the names of the *XML* nodes that we want to process as subtrees.

In addition, to these more sophisticated parsing strategies, this chapter also described strategies that take advantage of high-level functions for reading *XML* and *HTML* into *R* data frames and lists. Several of these functions were first introduced in Chapter 1. These are readHTMLTable(), htmlParse(), xmlToDataFrame(), xmlToList(), and getHTMLLinks(), and we refer the reader to Section 1.5 for summaries of these functions. We also introduced in this chapter a few additional high-level functions, such as xmlToS4(), and we briefly describe them below.

xmlAttrsToDataFrame() Process a collection of *XML* nodes or the child nodes of a document and convert their attributes into a data frame. This is convenient when the data of interest in the *XML* document are stored as attributes rather than subnodes. Each node corresponds to a row in the resulting data frame and columns correspond to attributes. Use the *attrs* parameter to specify a subset of attributes to be included in the data frame, or use the *omit* parameter to specify which attributes to discard.

xmlToS4() Map the *XML* content directly to an *S4* object with a particular class. This decomposes the children and attributes of the *XML* node and maps each, by name, to a slot in the *S4* object identified via the second argument, either as a class name or object. Elements in the *XML* content that correspond to a slot are converted to the corresponding type of that slot. This function works recursively on the child nodes.

getHTMLExternalFiles() Return a vector of all of the files/URLs referenced within the *HTML* document. This includes the image files (via the `<img>` and `<embed>` nodes), *JavaScript* and *CSS* documents and the hyperlink targets (*href* attribute values). The purpose of this function is to allow one to easily identify and collect all of the local files in order to transfer them to, for example,

a Web site. The function can work recursively, processing *HTML* documents referenced in the local links.

readHTMLList() Read each *HTML* list element in a document and return it as an *R* object. The function identifies ordered, unordered, and definition list nodes (`<ol>`, `<ul>` and `<dl>`), reads each of its elements, and represents them in *R* as strings or lists. This function is intended to be analogous to readHTMLTable() and to allow easy retrieval of semi-structured data from an *HTML* document.

5.15 Further Reading

The *SAX* parsing model is described in greater detail in Chapter 12 of [2] and in Chapter 20 of [1].

References

[1] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[2] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.

[3] Michel Rodriguez. *XML*::Twig: A *PERL* module for processing huge *XML* documents in tree mode. http://search.cpan.org/dist/XML-Twig/, 2012.

[4] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[5] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.

[6] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.

Chapter 6
Generating *XML*

Abstract In this chapter, we explore approaches to creating *XML* content within *R*. The primary approach is to create nodes and trees that are identical in nature to those returned by the *XML* parser via the xmlParse() and htmlParse() functions. Rather than generating an entire document in one step, we use functions to build individual nodes, add child nodes and attributes, and set namespaces. We discuss different approaches to building these trees—from top down or bottom up, or a hybrid of both approaches. In some circumstances, where the subtree is large and the nodes have a very regular structure, we can create the *XML* content more speedily via vectorized string manipulation and then parsing the content into a subtree, thus combining a string-based approach with operating on nodes and trees.

6.1 Introduction: A Few Ideas on Building *XML* Documents

While statisticians may spend most of their time working with *XML* that was generated by other researchers and Web sites, there are times when they want to create their own *XML* documents or adapt existing *XML* documents. For example, we may want to: create an *R* plot as a *Scalable Vector Graphics (SVG)* document and then add interactivity and/or animation to the display; export geographic data to *KML* for viewing on Google Earth; convert a data frame into an *HTML* table to include in a Web page; create *XML* to be sent to a Web service as the body of a request; update the contents of a spreadsheet or word-processing document stored as a collection of related *XML* documents; or, simply export data as *XML* to share with other applications because the data to be shared are not in a simple rectangular form or because there is a desire to include metadata with the data.

The basic idea for creating *XML* or *HTML* content in *R* using the XML package [5] is to create individual nodes and combine them into fragments or subtrees, and then into larger trees and complete documents. Sometimes it is convenient to build a tree/subtree from the top down, i.e., build the parent node and then build its children, then their children, and so on. In other cases, we build the children first and then add them to the parent. Generally, we want to use good concepts and practices from software development and design functions that create different pieces of the tree. We can then combine the subtrees appropriately. We can also call a function to create a subtree and then query and modify that tree before inserting the result into the larger context. We can use the functions we explored in Chapter 3 such as getNodeSet(), xmlName(), xmlAttrs(), xmlChildren(), etc., to query the subtree. We also provide several analogous and similarly named functions for changing nodes and their contents, e.g., `xmlName<-()`, addChildren().

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_6,

An alternative approach to building individual nodes and combining them is to create the *XML* content by incrementally building a large string or block of text. This is the approach used in several *R* packages such as R2HTML [3]. Generally, this works, but is not as flexible and robust as working with nodes and trees directly. Firstly, these string-based functions need to explicitly specify closing tags for *XML* elements, deal with empty nodes, and escape text content containing special characters such as & and < in the content of nodes or attributes. Much more importantly, modifying the output of these functions can be more difficult. Suppose we have a function that creates *XML* for an *HTML* table, or a collection of objects in an *SVG* display. We may want to change the alignment of one of the columns in the *HTML* table or add an *id* attribute to each of the *SVG* elements. To do this, people often use regular expressions. Parsing and querying *XML* is more difficult using regular expressions than with an *XML* parser, and it is especially challenging for *HTML* due to its often irregular/malformed structure. With an *XML* parser, we can work with the tree and individual nodes. We can query and modify nodes of interest and adapt the tree. We can then serialize the result back to a string if we want, e.g., to write to a file. In other words, creating *XML* content via string manipulation works adequately, but we typically want to operate at a higher level with nodes and trees. By parsing the string content, we can continue to think in terms of working with tree and node objects.

It is easy to switch from string-based *XML* content to a tree and nodes to get the flexibility we need to modify or extend the content. We encourage users and especially programmers developing functions to work with and return nodes and trees rather than strings. This results in more flexible and adaptable software. The XML package provides the necessary functionality for creating and modifying trees and nodes to generate hierarchical *XML* documents. In this chapter, we provide several simple examples of building *XML* content, e.g., *HTML* tables, *KML* for display in Google Earth and other general *XML* content. Later chapters (15, 16, and 17) and many packages provide case studies in developing a general approach to create different types of *XML* content and interface with important *XML* dialects and services.

This chapter is organized as follows. In Section 6.2 we provide an example to introduce a common and simple approach to generating *XML*. Then in Section 6.3 we more formally introduce the content generation functions available in the XML package and demonstrate several approaches to create *XML* documents using nodes and tree/subtree objects, and we also introduce approaches to modify an existing *XML* document and individual nodes. After that we provide more extensive examples of how to use these tools in Section 6.4. There are, of course, occasions when using string manipulation to create *XML* content is useful. These are typically when we need to create many nodes that have the same basic structure but with different values in the nodes or attributes. We demonstrate this technique in Section 6.5 and show how generating nodes in this fashion can be significantly faster. Section 6.6 describes how to add and use namespaces when creating and modifying *XML* nodes. Finally, in Section 6.7 we describe *R* functions to generate *XML* using one of the alternative representations of an *XML* tree that was introduced in Section 3.9.

6.2 A Simple Top-down Approach to Generating *XML*

In this section, we consider an example of creating an *HTML* page to post information on the Web where the information is presented as a table of values. These values are in an *R* data frame, and the task is to format it as an *HTML* table. For a small set of data, we might be inclined to write the *HTML* by hand. However, we want to do this with reusable code so that we maintain one source for the data. This way, when the data change, we can rerun the code to update the *HTML* table. This also hides the details from us. The R2HTML package [3] provides facilities to do this that are string-based. Instead,

we demonstrate how we can use the *XML* tree structure to create our document. For the purpose of demonstration, we use a small data frame.

Example 6-1 Generating an HTML Table from a Data Frame

We start with the data frame `chips`:

```
chips[1:4, 1:3]
```

```
      Date Transistors Microns
8080  1974       6000    6.00
8088  1979      29000    3.00
80286 1982     134000    1.50
80386 1985     275000    1.50
```

From it, we create the *HTML* table in Figure 6.1. For now we focus only on the structure of the content, and ignore issues of alignment of the values in the columns or making the table "pretty" in any way. We will return to these issues in a later example. Below is a snippet of the *HTML* that we want to generate:

```
<table border = "1" cellspacing="2">
 <tr>
  <th/> <th>Date</th> <th>Transistors</th> <th>Microns</th>
  <th>ClockSpeed</th> <th>Data</th> <th>MIPS</th>
 </tr>
 <tr>
  <td>8080</td> <td>1974</td> <td>6000</td> <td>6</td>
  <td>2</td> <td>8</td> <td>0.64</td>
 </tr>
...
</table>
```

Figure 6.2 shows the structure of this *HTML* subtree. Each row in the table corresponds to a <*tr*> node; each column header is in a <*th*> (table header) element within the first row's <*tr*> node; and the data appear in <*td*> (table data) nodes in the remaining <*tr*>s.

We build this *HTML* table from the top down and left to right, adding child nodes successively to the parent nodes. That is, we first create the top-level <*table*> node; then add to it a <*tr*> element for the first row in the table; to this <*tr*>, we add the variable names in <*th*> children. Subsequently, we add each row in the data frame to the table, with each data value as text inside its own <*td*> node.

We start by creating the "root" node of this *HTML* fragment, i.e., <*table*>. We call it a fragment because it is not a complete *HTML* document with the required <*html*>, <*head*>, and <*body*> nodes. We create the <*table*> node with the following call to newXMLNode():

```
chipTable = newXMLNode("table",
                       attrs = c(border = "1", cellspacing = "2"))
```

The "value" of `chipTable` at this point is an empty *XML* <*table*> node. As with other *R* objects and parsed *XML* documents created with xmlParse(), etc., we can examine the contents of the node by printing it on the *R* console:

```
chipTable
```

```
<table border="1" cellspacing="2"/>
```

	Date	Transistors	Microns	ClockSpeed	Data	MIPS
8080	1974	6000	6	2	8	0.64
8088	1979	29000	3	5	16	0.33
80286	1982	134000	1.5	6	16	1
80386	1985	275000	1.5	16	32	5
80486	1989	1200000	1	25	32	20
Pentium	1993	3100000	0.8	60	32	100
PentiumII	1997	7500000	0.35	233	32	300
PentiumIII	1999	9500000	0.25	450	32	510
Pentium4	2000	42000000	0.18	1500	32	1700

Figure 6.1: *HTML* Table Generated from an *R* Data Frame. This *HTML* table was created from an *R* data frame using the newXMLNode() function. The first row contains the variable names; the first column holds the row names; and subsequent entries in the table's cells display the corresponding data values. These variables are a combination of factor and numeric types.

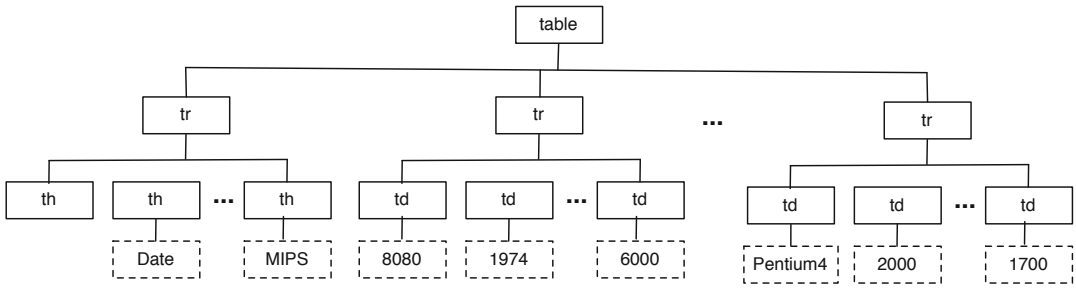


Figure 6.2: Tree Diagram of a Sample *HTML* Table. This tree provides a conceptual model for the structure of the *HTML* table in Figure 6.1. Each data cell in the *HTML* table is represented by a `<td>` node in the tree. Text nodes are displayed as boxes with dashed borders. In a later example, we will show how to add styles to the table entries.

This `<table>` node has two attributes, *border* and *cellspacing*, that have the values 1 and 2, respectively. These were specified via the *attrs* parameter in the call to newXMLNode(). The value of the *attrs* argument must be a named vector, where each name corresponds to an *XML* attribute name and the value corresponds to that attribute's value.

We can work with `chipTable` as a regular node, the same as those returned as part of a tree from a call to xmlParse() or htmlParse(). This means, we can use all of the functions we saw in Chapter 3 to query its contents, e.g., name, attributes, namespace, and children. We can also modify its contents to, for example, add children. We do this next.

The first child we add is the first `<tr>` node for the table's row of column headers. We again use newXMLNode() to create this node, but this time we specify the parent of the node we are creating (which is the value of `chipTable`). We provide the parent via the function's *parent* parameter with

```
headerRow = newXMLNode("tr", parent = chipTable)
chipTable
```

```
<table border="1" cellspacing="2">
  <tr/>
</table>
```

The <table> node is no longer empty; it now contains a <tr> element. We next want to add seven children to this header row, one for each variable name in the data frame plus a first blank <th> node that will appear above the column of row names. Each of these children are <th> nodes. We create the initial empty entry/cell by just omitting any content, again specifying the *parent* node, e.g.,

```
newXMLNode("th", parent = headerRow)
chipTable
```

```
<table border="1" cellspacing="2">
 <tr>
  <th/>
 </tr>
</table>
```

We did not assign the return value from the call to newXMLNode() to an *R* object because we do not need to refer to this node later as, e.g., a parent to another node. The node was added as a child of our <tr> node and so does not disappear (i.e., is not garbage collected), but remains a part of that tree.

Each subsequent child of `headerRow` contains text identifying the variable name for that column. We can create the first one as follows:

```
newXMLNode("th", names(chips)[1], parent = headerRow)
```

In this case, we are passing the string `"Date"` as the second argument to newXMLNode(). The newXMLNode() function collects this via its ... parameter and treats `"Date"` as a child for the new node. Since this child is a string, it becomes a text node. Rather than create each <th> node one at a time, we use sapply() to add all them in one call with

```
sapply(names(chips), function(x)
                        newXMLNode("th", x, parent = headerRow))
```

We check that indeed these <th> nodes and their text children are added to `headerRow`, i.e., the first <tr> node,

```
<table border="1" cellspacing="2">
 <tr>
  <th/> <th>Date</th> <th>Transistors</th> <th>Microns</th>
  <th>ClockSpeed</th> <th>Data</th> <th>MIPS</th>
 </tr>
</table>
```

We next turn to creating the rows of data in the table. We need to place the values of each observation in `chips` into a <td> child node of a <tr> node. In this case, it is convenient to pass newXMLNode() a list of all the child <td> nodes to add to a row. First, we define a helper function to create a single <td> as

```
makeCell = function(val) newXMLNode("td", format(val))
```

We can use this helper function and the *.children* parameter of newXMLNode() to create all the cells in the table with

```
sapply(row.names(chips),
       function(x) {
         newXMLNode("tr", parent = chipTable,
                    .children = c(newXMLNode("td", x),
                                  lapply(chips[x, ], makeCell)))
       })
```

This alternative form for specifying children as a list can be useful when the children are already in a list, e.g., when we create them with a call to a function such as lapply() or another that returns a list of nodes, such as getNodeSet().

At this point, we have added all of the cells to `chipTable`. The next step might be to add the table to an *HTML* document. We might have a template *HTML* document that we use when creating new *HTML* pages. We can read it into *R* with

```
doc = htmlParse("template.html")
```

We can then use the getNodeSet() function to find the <*body*> node and then add the table as a child of that node:

```
bNode = getNodeSet(doc, "//body")[[1]]
addChildren(bNode, chipTable)
```

We can, of course, add this to any other node in the template or even create a new node, e.g.,

```
newXMLNode('div', attrs = c(id = 'table:ChipSpeeds'),
           chipTable, parent = bNode)
```

This creates a new <*div*> node under the <*body*> node and adds `chipTable` as a child.

Now that we have the *HTML* document, we can write it to a file or generally serialize it in some manner. We can convert the document, or any node and its subtree, to a string with saveXML(). We can also use this function to write the tree directly to a file, e.g.,

```
saveXML(doc, file = "chipTable.html")
```

We do not have to use a template document to house the <*table*> node. Instead, we can create the document ourselves with newHTMLDoc(). We create the *HTML* document and, by default, it adds both a <*head*> and a <*body*> element. We can then add our <*table*> node as before. This is as simple as

```
doc = newHTMLDoc()
addChildren(xmlRoot(doc)[["body"]], chipTable)
```

Alternatively, we can use getNodeSet() to find the <*body*> node as we did above.

The code to create the table can easily be wrapped into a function and generalized to include other features for displaying the data, e.g., the number of digits displayed or the background color of the cells. However, our purpose here is only to introduce the basic approach to creating *XML* documents with the tools in the XML package. We will revisit this example later in this chapter to demonstrate how to insert nodes into a document, remove nodes, and add and change attributes on existing nodes. [See Example 6-3 (page 202).]

6.3 Overview of Essential Functions for Constructing and Modifying *XML*

In this section, we provide a brief introduction to the essential functions for creating *XML* nodes, subtrees, and documents. More extensive examples that cover the details of how to use these tools are provided in subsequent sections. As we emphasized in the introduction to this chapter, we strongly encourage people creating *XML* content in *R* to become familiar with and use node and tree objects to create hierarchical structures rather than representing the content in a flat string. That is, we create *XML* nodes and combine them into a document or tree by adding them to parent nodes. This can be accomplished in several ways and we demonstrate a variety of approaches in this section. The workhorse function is newXMLNode(). At its simplest, this creates a basic node with a name, e.g.,

```
vNode = newXMLNode("vector")
```

We can also give the node attributes via the *attrs* parameter, e.g.,

```
vNode = newXMLNode("vector",
                   attrs = c(type = "integer", length = 3))
```

Note that we have to name the elements of the *attrs* argument explicitly as these are the node's attribute names.

These nodes are identical in nature to those returned indirectly in the document created via calls to xmlParse() and htmlParse(). We can query the name of a node with xmlName() and get its attributes with xmlAttrs(). If the node had children, we can access those with xmlChildren() and `vNode[[1]]`, etc.

There are many different approaches to create child nodes and each has merit in different situations. We provide several short examples, and we summarize these approaches in page 192. When we create a node by calling newXMLNode(), we can specify an existing node to be the parent so the call to newXMLNode() is creating the child node and parenting it in the tree. For example, suppose we want to create an *SVG* group node <*g*> that contains a <*circle*> and a <*line*> child node. (We do not need to know all of the details of *SVG* to understand the concepts we are describing here, because the focus is on creating the nodes.) We might first create the <*g*> node with

```
grp = newXMLNode("g")
```

Then we can create the two child elements, passing `grp` as the *parent* argument with

```
newXMLNode("circle", attrs = c( cx = 30, cy = 50, r = 20),
           parent = grp)
newXMLNode("line", attrs = c(x1 = 3, y1 = 5, x2 = 7, y2 = 9),
           parent = grp)
```

This gives the *XML* fragment

```
<g>
  <circle cx="30" cy="50" r="20"/>
  <line x1="3" y1="5" x2="7" y2="9"/>
</g>
```

Here we are creating the nodes in a top-down order (parent, then children, etc.). If we want to add a child to <*line*>, we would assign that node to an *R* variable and then pass it as the value of *parent* in a call to newXMLNode() to create that child of <*line*>.

An alternative approach to creating <*g*> and its two children is to call newXMLNode() to create the <*g*> node and pass it the <*circle*> and <*line*> nodes to add as its children. We do this with

```
newXMLNode("g", newXMLNode("circle",
                           attrs = c(cx = 30, cy = 50, r = 20)),
                newXMLNode("line",
                           attrs = c(x1 = 3, y1 = 5, x2 = 7, y2 = 9))
)
```

We have specified the children as a loose collection via the . . . parameter. There is not a big distinction between this approach and the earlier one. They represent a slightly different way of organizing the computations. The former allows us to separate the creation of the two child nodes, and this can be useful. Here, newXMLNode() creates the <*g*> node and then adds the children to this new node.

Suppose instead of creating a <*circle*> and a <*line*>, we were looping over pairs of circle centers and creating many circles, e.g.,

```
circles = mapply(function(x, y)
                   newXMLNode("circle",
                              attrs = c(cx = x, cy = y, r = 20)),
                 coords[, 1], coords[, 2])
```

Given this *list* of <*circle*> nodes, how can we add them as children in a call to newXMLNode()? We can add each one by explicitly listing each element with

```
grp = newXMLNode("g", circles[[1]], circles[[2]], ...)
```

This is not a general or effective approach. (Although, we can use do.call().) Instead, we can provide a list of all the children via the *.children* parameter, e.g.,

```
grp = newXMLNode("g", .children = circles)
```

This achieves what we want, and is a pattern for other people's functions to follow.

Yet another approach to creating our original group containing a <*circle*> and <*line*> is to create all three nodes— <*g*>, <*circle*>, and <*line*> —separately:

```
grp = newXMLNode("g")
circ = newXMLNode("circle", attrs = c(cx = 30, cy = 50, r = 20))
line = newXMLNode("line", attrs =c(x1 = 3, y1 = 5, x2 = 7, y2 = 9))
```

Then we can explicitly add the two nodes as children of the <*g*> node. We can do this in one of two ways. We can use addChildren() with

```
addChildren(grp, circ, line)
```

or we can call xmlParent() to set the <*g*> node as the parent for each of the other two nodes:

```
xmlParent(circ) = grp
xmlParent(line) = grp
```

This allows us to set the two nodes as children separately. Of course, we can also make two separate calls to addChildren().

The resulting trees are identical for all of these approaches.

Suppose we want to add another shape node to our group. We can create it and add it as a child of the <*g*> element with any of

```
newXMLNode("rect", attrs = c(x = 20, y = 30,
                             width = 40, height = 30),
           parent = grp)
```

or

```
r = newXMLNode("rect", attrs = c(x = 20, y = 30,
                                 width = 40, height = 30))
addChildren(grp, r)
```

or

```
r = newXMLNode("rect", attrs = c(x = 20, y = 30,
                                 width = 40, height = 30))
xmlParent(r) = grp
```

Each of these will add <*rect*> as the third child of <*g*>.

Inserting Sibling Nodes at Other Locations

What if we wanted to add the rectangle in between the circle and the line, for some reason? We have a few ways to do this. The first is to use the newXMLNode() function to create the rectangle and simply use this function's *at* parameter to indicate the position in the collection of children of the *parent* node where we want the node inserted. In our case, we want to add the new node immediately *after* the first node so we specify 1 as the value of *at*, e.g.,

```
newXMLNode("rect", attrs = c(x = 20, y = 30,
                             width = 40, height = 30),
           parent = grp, at = 1)
```

Alternatively, we can use `at` in a call to addChildren().

Sometimes rather than knowing where in the collection of children to add a node, we know that the node should come immediately after or before an existing child node. We can use the function newXMLNode() to create the node and then use addSibling() to position the new node in the tree. We create the node with

```
r = newXMLNode("rect", attrs = c(x = 20, y = 30,
                                 width = 40, height = 30))
```

but do not specify the parent as we do not want to position the node at the end of the children. Instead, we use

```
addSibling(grp[["circle"]], r)
```

which places the rectangle immediately after the first <*circle*> node. The result is

```
<g>
  <circle cx="30" cy="50" r="20"/>
  <rect x="20" y="30" width="40" height="30"/>
  <line x1="30" y1="50" x2="50" y2="50"/>
</g>
```

What if we want to insert the rectangle before the <*circle*> node? The addSibling() function has a parameter called *after*, similar to the corresponding getSibling() function we saw in Chapter 3. This parameter allows us to control on which side of the target node to put the new node. We can place the <*rect*> node ahead of the <*circle*> with

```
addSibling(grp[["circle"]], r, after = FALSE)
```

The addSibling() function allows us to add more than one node at a time. We can specify them individually using the . . . mechanism or as a list via the *kids* parameter, similar to the *.children* parameter in newXMLNode().

We can also use `[[` to add children by position or by name. For example,

```
grp[[ 3, after = FALSE ]] = r
```

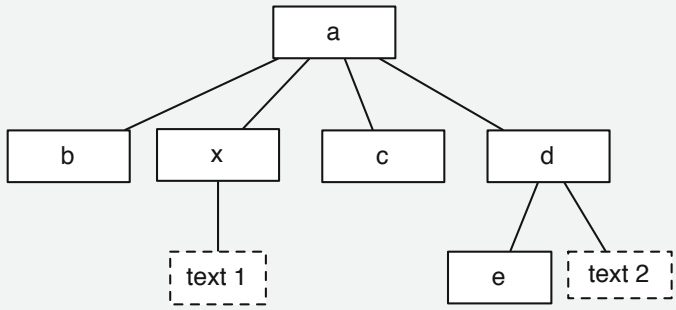
will add the <*rect*> node between the second and third children of `grp`.

How to Add Child Nodes to an *XML* Structure

We can create a simple node with newXMLNode(), passing the name of the element and a named character vector of attributes, e.g.,

```
aN = newXMLNode("a", attrs = c(id = "1"))
```

There are several ways to create child nodes within an *XML* element. We exemplify them by building the simple document shown below.



- If the parent node already exists, we can pass it as the value of the *parent* parameter in the call to newXMLNode() to create the child node, e.g.,

  ```
  newXMLNode("b", parent = aN)
  ```

  The <*b*> node being created is automatically added as a child of that parent (the <*a*> node).
- We can call newXMLNode() to create a node and pass it child nodes via the function's ... parameter. This involves creating the children before the parent or within the call to newXMLNode(), e.g.,

  ```
  subTree = newXMLNode("d", newXMLNode("e"), "text 2")
  ```

  The child nodes can be created with newXMLNode() or can be simple strings which are converted to *XML* text nodes and added.
- Rather than passing child nodes individually via the ... parameter in newXMLNode(), we can specify the entire collection of nodes as a *list* via the function's *.children* parameter. The nodes in this list can be *XML* nodes or text strings.
- We can separate the steps of creating a node and adding it as a child. An existing node can be given a parent with, e.g.,

  ```
  xmlParent(subTree) = aN
  ```

- Alternatively, we can add existing nodes as children to a parent node using addChildren(). For example, the previous call to xmlParent() yields the same result as

  ```
  addChildren(aN, SubTree)
  ```

  Multiple child nodes can also be specified via the ... parameter or in a list via the `kids` parameter.
- Finer control over where a node is added is possible with the `at` parameter of the newXMLNode() and addChildren() functions. For example,

```
addChildren(aN, newXMLNode("c"), at = 1)
```

adds <*c*> after the first child of <*a*> (which is the <*b*> node).

- If we know that we want to place the node next to another, and we know this other node's name but not its position, we can use addSibling(). We can also treat a node as if it were a *list* and add nodes to it by assigning a node as an "element". For the node,

```
xNode = newXMLNode("x", "text 1")
```

the following are equivalent:

```
addSibling(aN[["b"]], xNode)
addSibling(aN[["c"]], xNode, after = FALSE)
aN[["b", after = TRUE ]] = xNode
```

They each add the <*x*> node between the <*b*> and <*c*> nodes.

6.3.1 Changing a Node

We have seen how to add child nodes to an existing *XML* document or fragment/tree, but we may have an existing *XML* document and want to adapt it by changing an existing node's attributes or its name. For example, we may want to change an *SVG* document created by *R*'s graphics device to add interactivity, e.g., we might add *onmouseover* attributes to graphical elements to make the display respond to mouse events (this is the topic of Chapter 16). As with newXMLNode(), the functions offering these capabilities take advantage of the hierarchical structure of *XML*. These functions include addAttributes(), removeAttributes(), xmlAttrs(), and xmlName().

Adding and Changing Attributes

We can add attributes to a node after we have created it with the assignment version of the xmlAttrs() function, i.e., `xmlAttrs<-()`. For example, to add the attribute *na* with value `"NA"` to `vNode`, we can use

```
xmlAttrs(vNode) = c(na = "NA")
```

This looks as if it should replace all of the current attributes with the vector on the right-hand side. In most cases, however, we are either adding to the existing attributes or replacing a single attribute. For this reason, by default, xmlAttrs() merges the values on the right-hand side with the current attributes on the node, e.g., to yield

```
<vector type="integer" length="3" na="NA"/>
```

If we do want to replace all of the attributes with the new collection, we specify this with the *append* parameter, e.g.,

```
xmlAttrs(vNode, append = FALSE) = c(na = "NA")
```

to obtain

```
<vector na="NA"/>
```

Changing a Node's Name

If for some reason, we want to change the name of an existing node, we can do this with

`xmlName(node) = "newName"`. For example, to change the name of our <*vector*> node to `"integer"`, we use

```
xmlName(vNode) = "integer"
```

and we get

```
<integer na="NA"/>
```

One other operation we can perform on the node itself is to change its namespace. We will look at this later in Section 6.6.

6.3.2 Removing Nodes and Attributes

We have seen how to change the names and attributes of individual nodes and add new children throughout the tree. However, sometimes we need to remove nodes. As you might expect, the function removeNodes() does this for us. We call it with one or more nodes that we want to remove from their parents. The nodes can be from different parents. The removeNodes() function simply disconnects the nodes from the tree and cleans up after them, as appropriate.

We can remove a node and then insert a new one in its place. To do this, we might determine the sibling of the node to be replaced, use removeNodes() to discard the target node, and then call addSibling() to insert the replacement. Instead of this two-step process, the function replaceNodes() does both steps in one call.

A simpler, less flexible version of replaceNodes() is to assign a node as an "element" of a node's child list. We can use something like

```
grp[["circle"]] = newXMLNode("text", attrs = c(x = 10, y = 20),
                             "our own shape")
```

This is equivalent to

```
replaceNodes(grp[["circle"]],
             newXMLNode("text", attrs = c(x = 10, y = 20),
                        "our own shape"))
```

Both use the first child node named <*circle*>, but the first approach will create a new node named <*circle*> if it does not exist.

Updating an *XML* Document

To insert new elements into a tree, we can use addChildren() where we supply the parent node to which the children are added, or we can call addSibling() and supply the node that will be the next older (left) or younger (right) sibling. We can also specify the location of a new node using the *at* and *parent* arguments in newXMLNode().

A node can be replaced with another using replaceNodes(). Also, we can remove nodes by reference using removeNodes(), or by position in their parent using removeChildren(). With removeChildren(), we supply the parent of the nodes to be removed along with the position(s) of the nodes among the children, e.g.,

```
removeChildren(grp, "circle", 2)
```

removes the <*circle*> child and the second child of `grp`.

Additionally, we can add or replace attributes on a node with addAttributes() or `xmlAttrs<-()`, e.g.,

```
xmlAttrs(circ) = c(cx = "10", cy = "2")
addAttributes(circ, r = 17)
```

By default, the assignment of attributes using `xmlAttrs<-()` appends the attributes to the existing collection, overwriting existing attributes of the same name. To replace all of the attributes with a new set, we specify `FALSE` in the *append* argument, e.g.,

```
xmlAttrs(circ, append = FALSE) = c(cx = "1")
```

The function removeAttributes() allows us to explicitly remove one or more attributes by name.

We can create a copy of a tree of nodes with the function xmlClone() and then edit that independently of the original tree. We can add it to another node, even within another document.

We can change the name of a node with `xmlName<-()`, e.g.,

```
xmlName(circ) = "rect"
```

We can change or add text content to a node with

```
xmlValue(node) = "replacement text"
```

6.3.3 Generating Text Nodes

The functions we have described so far have illustrated both how to create a node and also how to change its name and attributes and how to introduce child nodes. These provide all the facilities one needs to manipulate and modify regular nodes. However, we often work with text content, and newXMLNode() provides a convenient way for us to add text nodes to a document. We saw in Example 6-1 (page 185) that we can add text to cells in an *HTML* table with, e.g.,

```
newXMLNode("th", "Date")
```

In general, a string or numeric passed via the ... argument of newXMLNode() (or as an element in its *.children* argument) is treated as text content. That is, it is turned into simple strings and then used as the value for a text node, which we can create explicitly with newXMLTextNode(). The previous command can be given as

```
newXMLNode("th", newXMLTextNode("Date"))
```

At times, we may need to change this text content. Suppose we had an *SVG* `<text>` node for displaying a segment of text in a display, e.g.,

```
<text x="30" y="50">a string</text>
```

We might create this with

```
tt = newXMLNode("text", attrs = c(x = 30, y = 50), "a string")
```

We have seen in Chapter 3 that we can access the text content with xmlValue(). Similarly, we can assign a value using

```
xmlValue(tt) = "a different string"
```

The result is

```
<text x="30" y="50">a different string</text>
```

6.3.4 Creating Other Kinds of XML Nodes

We have focused on creating regular *XML* elements consisting of a name and possibly attributes and children. We have seen how newXMLNode() creates `XMLInternalTextNode` objects when we give it a string as a child. It is typically more convenient to use this implicit mechanism, but we have seen that we can also explicitly create a text node with newXMLTextNode(), e.g.,

```
newXMLTextNode("some text", parent = node)
```

In addition to newXMLTextNode(), the XML package provides functions for creating other types of *XML* elements, such as comments, <*CDATA*> delimiters, processing instructions, and document-type declarations. These functions are newXMLCommentNode(), newXMLCDataNode(), newXMLPINode(), and newXMLDTDNode(), respectively. We can use these functions to create different types of nodes, and we can optionally specify the *parent* node and the position at which the new node should be added. Alternatively, we can create the node and add it to a tree using any of the functions we saw above, e.g., addChildren(), xmlParent(). In other words, these nodes and functions behave the same way as regular *XML* nodes and the newXMLNode() function.

6.3.5 Copying Nodes

Sometimes we want to create a copy of a node, i.e., subtree, and then modify that copy. For instance, in our *SVG* document, we might want to copy the <*g*> subtree that we created, and then add some new nodes and remove some others. Similarly, in a spreadsheet, we might want to copy a particular style and then change the color and font, but keep all of the other characteristics.[1] In these cases, we can use the xmlClone() function to create a duplicate of an existing node. Importantly, we can modify the new copy of the node without changing the original node.

Recall that we created three nodes, <*g*>, <*circle*> and <*line*> and assigned them to `grp`, `circ`, and `line`, respectively. That is,

```
grp = newXMLNode("g")
circ = newXMLNode("circle", attrs = c(cx = 30, cy = 50, r = 20))
line = newXMLNode("line", attrs = c(x1 =3, y1 =5, x2 =7, y2 =9))
```

We then made <*g*> the parent of the circle and line nodes with

```
addChildren(grp, circ, line)
```

Both `circ` and `grp[[1]]` reference the same <*circle*> node. If we clone this <*circle*> node, we get a new node. We can do this with

```
circClone = xmlClone(circ)
```

A change to `circ` will be reflected in `g`, but not in `circClone`. For example, we can modify the *cx* attribute in `circ` with

```
xmlAttrs(circ) = c("cx" = 100)
```

We see the new attribute value in `grp`:

[1] Actually, we should create a style derived from the original style in this situation.

```
<g>
  <circle cx="100" cy="50" r="20"/>
  <line x1="3" y1="5" x2="7" y2="9"/>
</g>
```

However `circClone`'s attribute remains `30`. We confirm this with

```
circClone
```

```
<circle cx="30" cy="50" r="20"/>
```

6.3.6 Creating an XML Document

Finally, we introduce the concept of a document. We have been focusing on creating nodes and subtrees of nodes. We can think of these as fragments of a document. However, when we have finished creating the individual nodes, we often want them to be part of a document. There are two reasons for this. Firstly, when we serialize a node or subtree, we see its hierarchical structure and contents. However, when we serialize a document object, we get a preamble that precedes the root node hierarchy. This preamble provides metadata about the document. For an *XML* document, the preamble appears as something like

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

This identifies the document as a regular *XML* document and tells the consumer about the character encoding, if necessary. For an *HTML* document, the preamble is something like

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
   "http://www.w3.org/TR/REC-html40/loose.dtd">
```

or simply

```
<!DOCTYPE html>
```

for an *HTML5* document.

The second benefit to using a document is that we need a tree to be within a containing document in order to perform *XPath* queries. This is a detail related to using `libxml2`. The getNodeSet() function and the xpathApply() functions take care of this for us by temporarily putting the tree into a document, if necessary. This can involve quite a lot of computation and so it is better to avoid this if possible by first creating a document and then adding nodes to it and its subnodes.

There are two functions for creating documents —one for regular *XML* documents and one for *HTML* documents. These are newXMLDoc() and newHTMLDoc(), respectively. We can use either to create an empty document, e.g.,

```
doc = newXMLDoc()
```

Then we can add children to them in various ways similar to the approaches above. For example, we can provide the document as the value of the *parent* argument in a call to newXMLNode(), e.g.,

```
newXMLNode("root", parent = doc)
```

Alternatively, we can do this with

```
addChildren(doc, newXMLNode("root"))
```

We can also pass the document object as the value for the *doc* argument in calls to newXMLNode(). This is different from using it as the parent node. Instead, this means that newXMLNode() should create the new node with a connection to the document and its namespace definitions and other metadata, but should not use it as the root node. This can be a good thing to do when the *parent* node is not specified but we know the node is to be part of a particular document.

The newHTMLDoc() function is similar to newXMLDoc(), but allows us to specify the type of *HTML* document. Specifically, we can identify the *DTD* declaration to use. There are various different forms of *HTML*4 and also the newer *HTML5*. We can specify which *DTD* to use with one of the strings `"strict"`, `"loose"`, `"frameset"`, or `"xhtml1-strict"`, or by specifying a full *URL*. We can also use the number (or string) 5 to indicate that we want an *HTML5* document. The newHTMLDoc() function, by default, also adds `<head>` and `<body>` elements within a root `<html>` node.

6.4 Combining Nodes to Construct an *XML* Document

Section 6.2 provided an example of using newXMLNode() in an orderly top-down approach to create an *HTML* table. We added nodes following the hierarchy of the desired tree, starting at the top and working our way down the tree moving left to right. In this section, we explore how to use newXMLNode() to generate *XML* documents when there is less repeated structure and the nodes are more heterogeneous. Again, the hierarchy of the document (and references to mutable nodes) allows us to do this, which is a distinct advantage over accumulating string content.

We provide two examples. In the first example, we create a simple *XML* file for display on Google Earth, as shown in Figure 6.4. For the second example, we modify the *HTML* table created in Example 6-1 (page 185) to add additional rows to the table and to add styles to the cells, as shown in Figure 6.5.

Instructions for drawing on Google Earth are given in the Keyhole Markup Language (*KML*), an *XML* grammar for displaying geographic data in an Earth browser. A *KML* tutorial is available at `http://code.google.com/apis/kml/documentation/kml_tut.html`, and the full API can be found at `http://code.google.com/apis/kml/documentation/kmlreference.html`. Also, Chapter 17 provides a case study for how to develop general functionality in *R* to create *KML* content for displaying data. (See also the RKML package [4].)

Example 6-2 Creating a Great Circle in the Keyhole Markup Language (KML)

In this example, we will demonstrate how to construct the following simple *KML* document from data in *R*.

```
<?xml version="1.0"?>
<kml xmlns="http://earth.google.com/kml/2.2">
 <Document>
  <name>Great Circle</name>
  <description>This segment of a great circle runs
  from (34N, 120W) to (46.8N, 145.5W).</description>
  <Placemark>
   <Style>
    <LineStyle>
     <color>ff0000ff</color>
     <width>2</width>
```

```
    </LineStyle>
   </Style>
   <LineString>
    <tesselate>1</tesselate>
    <altitudeMode>clampToGround</altitudeMode>
    <coordinates>-120.00,34.00,0 ... </coordinates>
   </LineString>
  </Placemark>
 </Document>
</kml>
```

This document contains, among other information, instructions for drawing a curve on Google Earth. The hierarchy of this document is displayed in Figure 6.3 (see Figure 6.4 for a screenshot of the curve on Google Earth).

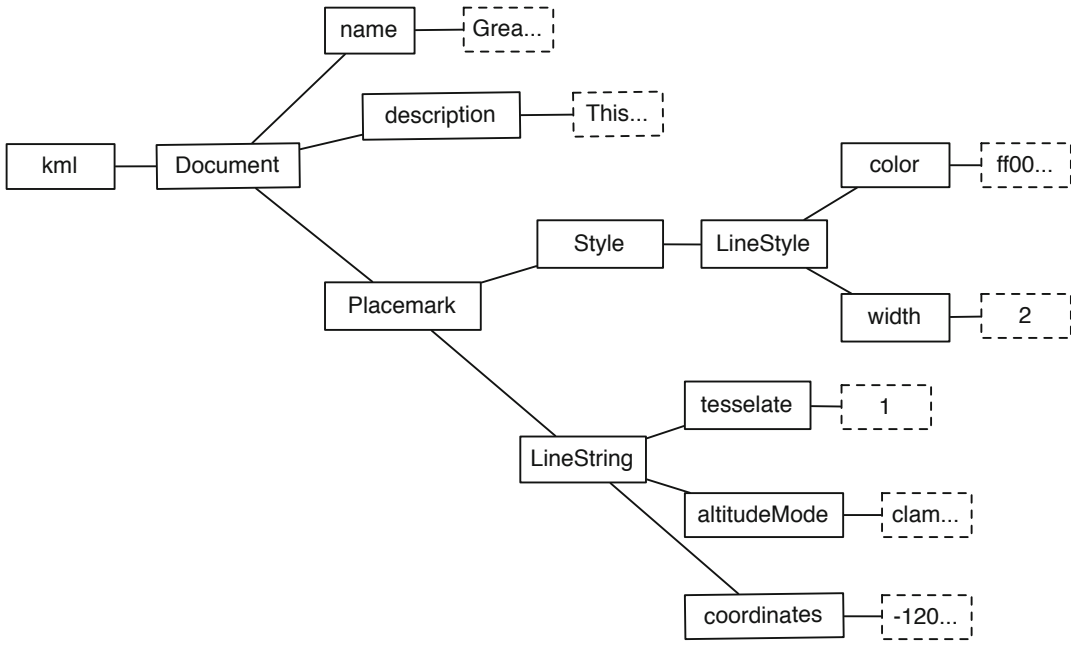


Figure 6.3: Tree Diagram of a *KML* Document. The tree shown here represents the hierarchy of a simple *KML* document that contains instructions for drawing a set of red connected line segments on Google Earth. The single text-node child of `<coordinates>` contains the sequence of longitude, latitude, altitude triples that make up the endpoints of these line segments.

The curve is drawn as a series of connected line segments, which are determined by the coordinates in the character vector `coordGC`.

```
head(coordGC)
```

```
[1] "-120.00,34.00,0 " "-121.11,34.80,0 "
[3] "-122.25,35.58,0 " "-123.40,36.36,0 "
[5] "-124.58,37.12,0 " "-125.78,37.88,0 "
```

Each element is a longitude, latitude, altitude triple. These will be added to the *KML* document as text content to the <*coordinates*> node.

To begin, we create the root node, <*kml*> and its sole child <*Document*>. We provide the default namespace for the root node via the *namespaceDefinitions* argument to newXMLNode(). The function expects a named character vector for the value of *namespaceDefinitions*, where the name of each element corresponds to a namespace prefix and the value is the URI identifying that namespace. If an element in this *R* vector has no name, i.e., the empty string `""`, then that element is used as the default namespace for this node and its descendants. We create the root node and its only child with

```
kml.ns = c("http://earth.google.com/kml/2.2")
kmlTree = newXMLNode("kml", newXMLNode("Document"),
                     namespaceDefinitions = kml.ns)
kmlTree

<kml xmlns="http://earth.google.com/kml/2.2">
  <Document/>
</kml>
```

We want to put these nodes within the context of a document. This will add the *XML* declaration when we serialize the nodes. It also facilitates using *XPath* queries on the document to find nodes. It is therefore a good idea to put the root node within a document. We create this document with the newXMLDoc() function, in this case, passing the <*kml*> node as the root as follows:

```
gcDoc = newXMLDoc(node = kmlTree)
gcDoc

<?xml version="1.0"?>
<kml xmlns="http://earth.google.com/kml/2.2">
  <Document/>
</kml>
```

Notice the addition of the *XML* declaration. `gcDoc` is an *XML* document object that contains both the *XML* content and information about the document, such as its *DTD* and the character encoding, if specified.

Figure 6.3 shows us that <*Document*> has three children and the first two, <*name*> and <*description*>, are simple in that they each contain only text content. We create these two children next as children of the <*Document*> node. Since the <*Document*> node was created in the call to make the <*kml*> element, it was not assigned to an *R* object. This means we will need an alternative approach to specify the parent of <*name*> and <*description*>. We access the <*Document*> element through its parent, which is in `kmlTree`, e.g., as `kmlTree[["Document"]]` or `kmlTree[[1]]`. We provide the necessary information to place the <*name*> and <*description*> elements at their correct positions in the hierarchy as follows:

```
newXMLNode("name", "Great Circle", parent = kmlTree[["Document"]])
newXMLNode("description",
           sprintf(
              "This segment of a great circle runs from\n
                (%.2f N, %.2f W) to (%.2f N, %.2f W).",
           latitude[1], -longitude[1],
           latitude[20], -longitude[20]),
           parent = kmlTree[["Document"]])
```

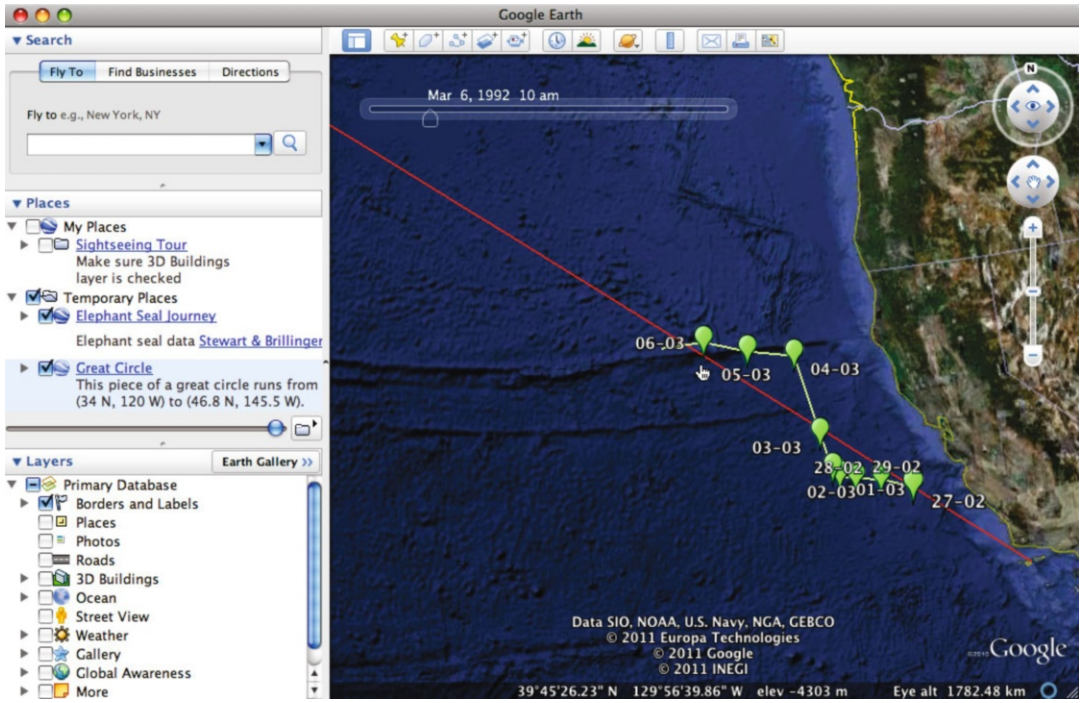


Figure 6.4: Portion of a Great Circle on Google Earth. This screenshot shows the movements of an elephant seal and a fragment of a great circle (in red). One endpoint of the circle segment is near the Channel Islands in southern California. In the left panel, the "Great Circle" link next to the Google Earth blue-and-white icon controls the display of the *KML* document on the earth viewer and provides additional information about the document.

This approach to treating a node as a list of its children and subsetting by index is described in more detail in Chapter 3.

For the third node, `<Placemark>`, we demonstrate yet another approach to building an *XML* node and children. We generate this node by creating its children first; that is, we create the node from the "bottom" up. The children of `<Placemark>` are `<Style>` and `<LineString>`. If we create them before their parent node, then they are not (yet) part of the `kmlTree` structure.

We begin with `<Style>`. It too has a child, `<LineStyle>`, which we create now as

```
lsty = newXMLNode("LineStyle", newXMLNode("color", "ff0000ff"),
                  newXMLNode("width", "2"))
lsty
```

```
<LineStyle>
  <color>ff0000ff</color>
  <width>2</width>
</LineStyle>
```

Again, the *XML* node assigned to `lsty` is not (yet) part of `kmlTree` so it has no parent. When we create `<Style>`, we pass `lsty` via ... in the following call to newXMLNode():

```
style = newXMLNode("Style", lsty)
style
```

```
<Style>
  <LineStyle>
    <color>ff0000ff</color>
    <width>2</width>
  </LineStyle>
</Style>
```

Now, the <*LineStyle*> node has <*Style*> as a parent, but these nodes do not yet belong to the *KML* document.

Similarly, we create the second child of <*Placemark*>, which is <*LineString*>, without specifying a parent because the parent does not yet exist:

```
lineStr = newXMLNode("LineString", newXMLNode("tesselate", "1"),
                     newXMLNode("altitudeMode", "clampToGround"),
                     newXMLNode("coordinates", coordGC))
```

Lastly, we create the <*Placemark*> element, using the <*Document*> node as its parent, and we specify its children as `style` and `lineStr`. The command is

```
pm = newXMLNode("Placemark", style, lineStr,
                parent = kmlTree[["Document"]])
```

The tree is now complete. We can write the document to a file using saveXML()

```
saveXML(gcDoc, file = "greatCircle.kml")
```

This can then be immediately viewed in Google Earth locally, or published on a Web page for others to view.

This example used a namespace that applied to all nodes and so was quite simple. In Section 6.6, we will discuss creating nodes with namespaces more generally.

Next we explore the functions in the XML package for modifying nodes in an *XML* tree. We saw these briefly in Section 6.3, and we will explore them here via a more comprehensive example to give a better sense of how they may be used in practice. We demonstrate how to add, remove, and replace nodes and attributes in an *HTML* table. In general, if we originally programmatically generated the *HTML*, then we can modify and rerun our code to create the new *HTML* that reflects the changes. On the other hand, if we did not create the original *XML* or if the change was small compared to the complexity of regenerating the entire document, then we can update the existing *XML* document rather than reproduce it from scratch.

Example 6-3 Modifying an Existing HTML Table

This example uses the simple *HTML* table created in Example 6-1 (page 185) to demonstrate how to modify *XML*. Rather than regenerate the table by changing the code from the earlier example, we demonstrate how to modify the existing table, which was saved as the text file `chipTable.html`. We will make the following modifications:

- include additional rows (<*tr*> and <*td*> elements) for other microprocessors;
- insert a caption/title (a <*caption*> element) into the table;
- modify the text content of an existing cell;

- add attributes containing style information to a subset of the rows;
- reassign the value of the *cellspacing* attribute for the table.

See Figure 6.5 for a screenshot of the revised table. Also, Figure 6.6 provides a diagram that indicates the changes that we will make to the hierarchy.

Intel Microprocessors

	Date	Transistors	Microns	ClockSpeed	Data	MIPS
8080	1974	6000	6	2	8	0.64
8088	1979	29000	3	5	16	0.33
80286	1982	134000	1.5	6	16	1
80386	1985	275000	1.5	16	32	5
80486	1989	1200000	1	25	32	20
Pentium	1993	3100000	0.8	60	32	100
PentiumII	1997	7500000	0.35	233	32	300
PentiumIII	1999	9500000	0.25	450	32	510
Pentium4	2000	42000000	0.18	1500	32	1700
Prescott	2004	1.25e+08	0.09	3600	32	7000
Conroe	2006	2.91e+08	0.065	2670	64	9533

Figure 6.5: Modification of the *HTML* Table from Figure 6.1. This screenshot shows an updated version of the *HTML* table from Example 6-1 (page 185), which is shown in Figure 6.1. The title "Intel Microprocessors" has been added, and new rows have been added to the bottom of the table. Additionally, alternate rows have been given a blue background, and the cell spacing has been reduced from 2 to 0. To change the background colors, we use a class attribute on alternate rows, and we use a *CSS* file to specify the background color for this class of rows.

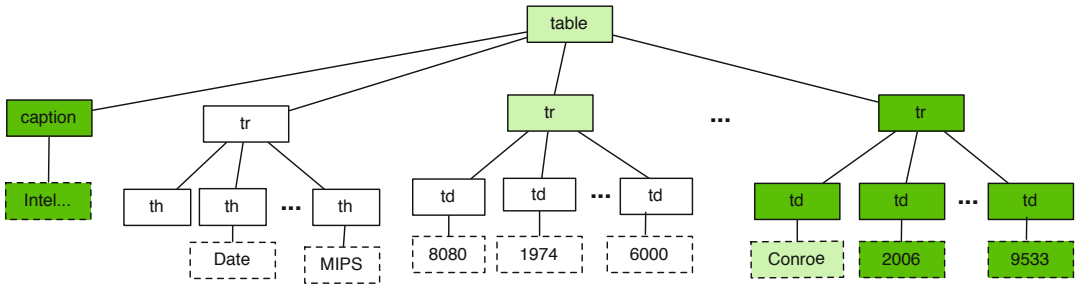


Figure 6.6: Tree Diagram of the Revised *HTML* Table. This diagram shows the updated hierarchy of the *HTML* table in Figure 6.5. The nodes shaded dark green are new nodes that have been inserted into the structure. The nodes that are lightly shaded were in the original structure, but have been modified, e.g., we changed the second `<tr>` node by adding `class="odd"` to it.

Our goal is to change the existing table node that we created earlier and assigned to the *R* variable `chipTable`. We can work directly with that node. Alternatively, should we want to preserve the original, we can make a copy of it with

```
Table = xmlClone(chipTable)
```

This means that if we need to start over, we have not corrupted the original table. Alternatively, if we had written the table to a file, we can parse that document with

```
doc = htmlParse("chipTable.html")
```

In this case, we want the <*table*> node and not the document so we get this with either of the commands

```
Table = xmlRoot(doc)[["table"]]
Table = getNodeSet(doc, "//table")[[1]]
```

(These functions, htmlParse() and xmlRoot(), are covered in detail in Chapter 3, and getNodeSet() is explained in Chapter 4).

Add New Rows to the Table

Our first task is to update the table with information for two new processors, i.e., to add new rows to the table. This information is available in the data frame `newrows`:

```
newrows[, 1:3]
```

```
                    Date Transistors Microns
Pentium Prescott    2004    1.25e+08   0.090
Intel Core 2 Conroe 2006    2.91e+08   0.065
```

We use the same approach as in Example 6-1 (page 185) to create a list of <*tr*> nodes. If we had wrapped the previous code into a function, called makeTableRow(), we would do this with

```
newTRNodes = lapply(row.names(newrows), makeTableRow)
```

Then we can call addChildren() to add this list of two nodes to the tree in the `Table`,

```
addChildren(Table, kids = newTRNodes)
```

Insert a Caption into the Table

Next we add a <*caption*> node to the table. We will insert it as the first child of the table. We can do this with addChildren() or alternatively, with addSibling().

The addChildren() function has a parameter called *at* that allows us to specify where among the node's existing children we want to add the new nodes. The new node is added after the specified node. We set *at* to 0 in the following call so that <*caption*> becomes the new first child:

```
cap = "Intel Microprocessors"
addChildren(Table, newXMLNode("caption", cap), at = 0)
```

This adds the new node and moves the others to the right. It does not replace the node at the position specified via the *at* argument. To do that we would use replaceNodes().

Note that more than one node can be supplied and inserted into the tree, and that each can be added at a different location if *at* is given as a numeric vector of positions, e.g.,

```
addChildren(parent, x, y, z, at = c(1, 3, 0))
```

Instead of adding the caption via addChildren(), we can use addSibling() and provide this function the first <tr> as the sibling next to which the new node is placed:

```
addSibling(Table[[1]], newXMLNode("caption", cap), after = FALSE)
```

With the *after* parameter we can specify whether we want the new node to appear before (to the left of) or after (to the right of) the specified sibling. FALSE means before, or not after.

Neither approach is better than the other. If we already have the sibling at which we want to insert the new node, then addSibling() is simplest. If we are working with the parent object and know the position, then calling addChildren() with the *at* parameter to index the node can be convenient.

Change a Cell's Text Content

The third task we have is to change the text value that gives the processor name in the cell in the last row of the table. Again we demonstrate a few ways to carry out this task. The first replaces the entire <td> node that contains the text content, and the second approach merely reassigns the text content of the existing <td> node only. We use the replaceNodes() function for the first approach:

```
n = xmlSize(Table)
replaceNodes(Table[[n]][[1]], newXMLNode("td","Conroe"))
```

This function takes two nodes as input. The first is the node we want to replace, and the second is the node that will take its place. We can instead change the content of the <td> element via the xmlValue<-() function. We do this with

```
xmlValue(Table[[n-1]][[1]]) = "Prescott"
```

This creates a text node from the string provided and replaces the content of Table[[n-1]][[1] with this text. In this example, the <td> node has simple content and using xmlValue() works well. In other circumstances, when the content is mixed (i.e., there are text and regular child nodes), we typically want to add or replace nodes.

Yet another approach uses removeNodes(). Here, we first remove the node that we wish to replace and then add the replacement node using [[<-() with

```
removeNodes(xmlChildren(Table[[n]]))
Table[[n]][[1]] = newXMLTextNode("Conroe")
```

Add Style Attributes to Every Other Row in the Table

The two final tasks involve making modifications to the attributes of nodes. The first is to change every other row in the table to have a blue background. We can do this by specifying the color via a *bgcolor* attribute on each of the relevant <tr> nodes. We use addAttributes() to do this with

```
sapply(Table[seq(3, n, 2)], addAttributes, bgcolor = "#9999FF")
```

Typically, we want to centralize style specifications using a Cascading Style Sheet (*CSS*). Suppose that the *HTML* file into which we are embedding the table already has a *CSS* style called "tr.odd" for this purpose.[2] If this is so, we simply need to add a *class* attribute with a value of "odd" to alternate <tr> nodes, e.g.,

```
sapply(Table[seq(3, n, 2)], addAttributes, class = "odd")
```

Note that addAttributes() can also be invoked as follows:

```
addAttributes(Table[[3]], .attrs = c(class = "odd"))
```

[2] This style sheet might be in an external or inlined *CSS* document, and the style specification might look something like: `tr.odd = { background-color = #9999FF; }`

This form allows us to compute the attributes and pass them as a single collection, similar to the *.children* parameter of newXMLNode().

Change the Cell Spacing Attribute in the Table

The final change to the table is to reset the *cellspacing* attribute to 0 on the top-level <table> node. We use the xmlAttrs() function to do this:

```
xmlAttrs(Table)["cellspacing"] = 0
```

or

```
xmlAttrs(Table) = c(cellspacing = 0)
```

Note that if the node did not already have a *cellspacing* attribute, then this attribute would be added to the node, i.e., `xmlAttrs<-()` can be used to add attributes as well as modify them.

An alternative approach would be to remove the attribute, and then add a new attribute with the desired value. We include the code to do this solely to demonstrate the removeAttributes() function. The following two function calls are equivalent:

```
removeAttributes(Table, "cellspacing")
removeAttributes(Table, .attrs = c("cellspacing"))
```

Having removed the *cellspacing* attribute, we call addAttributes() (or xmlAttrs()) to insert a new *cellspacing* attribute that has the desired value:

```
addAttributes(Table, .attrs = c(cellspacing = 0))
```

Finally, we save the updated table to a text file with

```
saveXML(doc, file="chipTableUpdated.html")
```

6.5 Vectorized Generation of *XML* Using Text Manipulation

We have indicated that creating node objects and combining them into trees using newXMLNode() and the other functions is a good and robust approach to create *XML* content, and that creating *XML* by pasting strings together is less robust and flexible. However, there are occasions when using string manipulation to create *XML*/*HTML* content is useful. These are typically when we need to create many nodes that have the same structure but with different values in the content or attributes. For example, consider our earlier exercise of creating an *HTML* table with a row for each observation in a data frame [Example 6-1 (page 185)]. There, we looped over the rows and created the corresponding <tr> nodes and, within these, we created each of the <td> elements for each cell in the row. The child nodes were all the same for each row. In this case, working with character strings can have an advantage of speed. The reason is because, in simple cases where all the cells and all the rows have the same structure, we can vectorize the creation of the <td> and <tr> nodes with calls to sprintf(). This can result in significant speedup relative to looping in *R* and creating the nodes with individual calls to newXMLNode(). This approach is not very general, but not uncommon, and the performance gain for large collections of similar nodes is often very significant. There are downsides to pasting together strings. This approach can be more complex when the nodes do not have the same structure (and in this situation it may also be inefficient). We are responsible for ensuring the *XML* is

well-formed, nodes have a closing tag, empty nodes are reduced to a self-closing form, and also for escaping all special characters such as < and &.

The XML package includes the function parseXMLAndAdd(), which takes *XML* content as a string, parses it, and returns the parsed tree or adds the parsed nodes to a specified parent. When we develop functions to generate *XML*, the ideal approach is to use a string-based approach when it is significantly faster and to use a node approach otherwise. Importantly, users should not know how we create the content within our functions. We can abstract the details of the approach and return a tree object regardless of whether the function uses strings to create it. For example, a function that creates an *HTML* table might use strings, but would return the tree object. We can return the tree object by parsing the string using parseXMLAndAdd() or xmlParse(). These strategies are outlined below, and in Example 6-4 (page 207), we demonstrate how to generate a *KML* document with this approach of building logical "chunks" of a tree as text and then converting these to nodes or subtrees to add to a tree. We then compare how fast this is relative to creating the nodes individually and illustrate the efficiency gains from the judicious use of strings.

Strategies for Creating Nodes via Text Manipulation

It can be more efficient to use strings when we create many nodes that have a similar structure. The reason for this is that we can use vectorized functions to create content for the different nodes in a single call rather than having to explicitly loop in *R* to create each node. For example, consider creating an *HTML* table to display a data frame. We can create the individual cells in a row in the table using sprintf(), and then we can then combine them into a single string for all of the cells in all of the rows. We do this with code like

```
cells = sapply(1:nrow(DF),
               function(i)
                   paste(sprintf("<td> %s </td>", DF[i, ]),
                         collapse = ""))
tableRows = paste("<tr>", cells, "</tr>", collapse = "")
```

Due to the vectorized functions (sprintf() and paste()), this is much faster than looping over each row and each cell and calling newXMLNode().

If there are many similarly structured nodes to create, we recommend a hybrid approach where the repetitive nodes are constructed by text manipulation/generation and then parsed into an *R* structure and added as children to an existing node, e.g.,

```
parseXMLAndAdd(tableRows, parentNode)
```

Here, `parentNode` in not a character string, but an *XML* node.

It is better to work with nodes rather than strings so that we can use higher-level robust operations instead of pattern matching via regular expressions to manipulate the resulting tree. However, creating the initial content as strings and then parsing that and adding it to an existing tree is just an implementation detail.

Example 6-4 Creating KML Using Text Manipulation

In this example, we add `<Placemark>` nodes to a *KML* document for each observation in the data frame `quakes`, a data frame containing a row for each of the 22,118 earthquakes recorded in 2011. This is available from the USGS/NEIC database available at `http://earthquake.usgs.gov/earthquakes/eqarchives/epic/epic_global.php`. We want each node to appear as

```
<Placemark>
 <Point>
  <coordinates>143.166,27.247,0</coordinates
 </Point>
</Placemark>
```

where 143.166 is the longitude and 27.247 is the latitude of the quake. The 0 indicates that we want the placemark to be at 0 elevation (which we want for all <*Placemark*>s). We can create each <*Placemark*> element with code such as

```
newXMLNode("Placemark",
           newXMLNode("Point",
                      newXMLNode("coordinates",
                                 paste(143.166, 27.247, 0,
                                       collapse = ","))))
```

Alternatively, we can create a series of <*Placemark*> nodes as a string in a vectorized manner as follows:

```
kmlTxt = sprintf("<Placemark><Point><coordinates>%.3f,%.3f,0
                  </coordinates></Point></Placemark>",
                 quakes$Longitude, quakes$Latitude)
```

This vectorizes the operation across all 22,118 quakes, which we cannot do with newXMLNode().

We want to add the <*Placemark*> nodes as children of the <*Folder*> node in the following *KML* document:

```
kmlQuakes

<kml xmlns="http://earth.google.com/kml/2.2">
 <Document>
  <name>2011 Quakes</name>
  <description>Locations of all earthquakes in 2011</description>
  <Folder>
   <name>Earthquakes in 2011</name>
  </Folder>
 </Document>
</kml>
```

We cannot do that directly using the text representation, but we can parse the *KML* text in `kmlTxt` and then add the <*Placemark*> nodes to the <*Folder*> node. We cannot parse the *XML* in `kmlTxt` as it is not well-formed since there is no root/top-level node. We need to add a root node to act as a parent for the collection of <*Placemark*> nodes. We can do all of this with

```
tmp = sprintf("<doc>%s</doc>", paste(kmlTxt, collapse=""))
doc = xmlParse(tmp, asText = TRUE)
addChildren(kmlQuakes[["Document"]][["Folder"]],
            xmlChildren(xmlRoot(doc)))
```

However, the parseXMLAndAdd() function does this for us; it adds a root node, parses our text, and re-parents the nodes in a single call. We call it with

```
parseXMLAndAdd(kmlTxt, kmlQuakes[["Document"]][["Folder"]])
```

Let's explore the performance differences between these two approaches—the one that constructs one node at a time and the one that creates all the `<Placemark>` elements via strings. The following function creates the nodes individually via newXMLNode()

```
addPlacemarks.slow =
function(lon, lat, parent)
{
  mapply(function(a, b)
          newXMLNode("Placemark",
                     newXMLNode("Point",
                                newXMLNode("coordinates",
                                           paste(a, b, 0,
                                                 collapse = ","))),
                     parent = parent),
         lon, lat)
}
```

We time how long this takes to create the 22,118 nodes with

```
system.time(invisible(
              addPlacemarks.slow(
                      quakes$Longitude, quakes$Latitude,
                      kmlQuakes[["Document"]][["Folder"]])))
```

```
   user  system elapsed
 33.211   0.197  33.836
```

The alternative string-based function is implemented as

```
addPlacemarks.fast =
function(lon, lat, parent)
{
  txt = sprintf("<Placemark><Point><coordinates>%.3f,%.3f,0
                 </coordinates></Point></Placemark>",
                lon, lat)
  parseXMLAndAdd( paste(txt, collapse = ""), parent)
}
```

We time it using the same data with

```
system.time(invisible(
              addPlacemarks.fast(
                      quakes$Longitude, quakes$Latitude,
                      kmlQuakes[["Document"]][["Folder"]])))
```

```
   user  system elapsed
  0.490   0.031   0.524
```

The improvement is more than a factor of 50! There are several reasons for this. One is that the call to sprintf() is implemented in *C* code and so is very fast. The mapply() loop is also implemented in *C*, but it calls our *R* function for each iteration and so is not vectorized at the *C*-level. Also, the functions are doing different things. parseXMLAndAdd() parses the *XML* text. It does this via *C*-level code.

While it tests that the *XML* is valid, it does not do much to process it other than converting it to a tree structure. In contrast, the newXMLNode() function uses *C*-code, but is primarily implemented in *R*. Moreover, newXMLNode() does a lot of work to try to make creating nodes easier for the *R* programmer. It takes care of resolving namespaces, checking the node name and attributes for a namespace prefix, creating and adding children at different positions in the parent, and so on. While we can create a stripped-down version of newXMLNode() that deals with simple nodes and that does not do the extra computations for less common situations, using stings and parseXMLAndAdd() is still the best approach for many common situations.

6.6 *XML* Namespaces

Recall from Section 2.6 in Chapter 2 that a single *XML* document may sometimes use multiple vocabularies, combining concepts and data from different domains. For example, we may have *R* and *C* code in the same document and need to distinguish `<code>` nodes in the two languages. Similarly, we may want to use `<model>` nodes in different ways, e.g., for a statistical model and a graphical model, or we may have `<vector>` nodes that denote a vector in the *R* sense and in the math sense, etc. In *XML*, we specify to which vocabulary a node belongs using a namespace. We define a namespace on a node as a pair: `prefix=URI`. The URI is a global identifier for the namespace and this should be the same across different documents. The URI need not exist or be anything more than a string because its purpose is only to provide a unique identity for the vocabulary. The prefix is used only on the document and is a local shorthand for identifying the namespace.

We define namespaces on a node similar to how we place attributes on a node, e.g., `xmlns:prefix="URI"`. For example, the node

```
<example xmlns="http://docbook.org/ns/docbook"
         xmlns:r="http://www.r-project.org" />
```

defines two namespaces. The first has no prefix and is therefore the default namespace for this node and its descendants. This namespace identifies the *DocBook* vocabulary for authoring technically oriented documents. The second namespace uses the URI `http://www.r-project.org` and the prefix `r`. We do not use it within this node, but it is defined here for use within the descendant nodes. For example, consider the tree given by

```
<example xmlns:r="http://www.r-project.org"
         xmlns="http://docbook.org/ns/docbook">
 <para>
The <emphasis>expression</emphasis>
  <r:expr>rep(1:3, 2)</r:expr> returns
  <r:output>
[1] 1 2 3 1 2 3
  </r:output>
 </para>
</example>
```

The `<expr>` and `<output>` nodes use the `r` prefix to identify that they are part of the *R* namespace. We find the definition for this prefix in the `<example>` ancestor node. The `<para>` node is part of the *DocBook* namespace since that is the default namespace in effect from its parent, `<example>`.

We can define a namespace on each node. However, these repeated definitions are not necessary because a node inherits the namespaces of its ancestors. The non-local definition is better to avoid redundancy, but of course requires synchronization between the code that creates the nodes. When we create a new node, it may not be complete without its parent, or ancestors generally, because it needs a namespace definition to make sense of the prefix. This is different from our previous experiences with newXMLNode() where we can define nodes independently of each other and add them to the tree in different ways and order.

In this section, we discuss several approaches for working with namespaces and definitions for nodes that we create in *R*. These approaches depend on the way the node is being constructed (these are summarized on page 214. For the most part, newXMLNode(), addChildren(), etc., do the hard work for us and attempt to make sense of namespaces when creating and inserting nodes into a tree. Basically, we can define *XML* namespaces when we create a new node with newXMLNode() using the *namespaceDefinitions* parameter. We can use that namespace within the name of the node being created by using its prefix, e.g.,

```
rc = newXMLNode("r:code", namespaceDefinitions =
                                  c(r = "http://www.r-project.org"))
```

When we do not define the namespace when creating the node, newXMLNode() needs a way to resolve the namespace prefix given on the node name and find the actual namespace definition. If we specify the parent node via the *parent* argument, newXMLNode() can walk the hierarchy of ancestors until it finds a namespace with the same prefix. Therefore, a call such as

```
newXMLNode("r:output", parent = rc)
```

adds this node as a child of the <*r:code*> node we created earlier and finds the namespace definition there. This produces the subtree

```
<r:code xmlns:r="http://www.r-project.org">
  <r:output/>
</r:code>
```

In general, when we build a document top down, we can define the namespaces at the appropriate common ancestor. Then when we create the child node with newXMLNode() and specify the parent node, we can resolve the namespace by looking up the ancestor hierarchy. For instance, we commonly define all of the namespaces in the root node so that they are available to all nodes in the tree. This has the advantage of defining all namespaces in the same place, but it has the disadvantage of being harder to connect the namespace definition to its use. It would seem that we also have to know which namespaces to define when we create the root node. In fact, this is not the case as we can navigate from one node in the tree to the root node and *add* a new namespace as needed at any time with the newXMLNamespace() function, e.g., with either of the commands

```
newXMLNamespace(node, "http://www.omegahat.org", "omg")
newXMLNamespace(node, c(omg = "http://www.omegahat.org"))
```

The potential problem involving namespaces arises when we create a node without specifying its parent. This occurs, for example, when we use individual functions to create subtrees and these functions do not know if the namespace they need to use has been defined in the larger tree as that may not have been created when it is called. Ideally, these functions should use the same namespace definitions and have them defined centrally. Another solution is for each piece of code to define its own namespaces, e.g., at the topmost node of the subtree it creates. Again, this may lead to redundant

namespace definitions, but the functions and the resulting trees are robust and this is a good practice. If necessary, we can remove redundant definitions with some additional computations, e.g., with xmlCleanNamespaces() or `NSCLEAN` as an option to xmlParse() (see Section 3.8).

When we are creating nodes and assembling them into a tree, it is convenient to be able to create the nodes separately and refer to namespaces, knowing that they *will* be defined in an ancestor node *when* we add the node to a tree. We might create several child nodes and then add them to their parent. For instance, we can create the `<r:output>` node with

```
rOut = newXMLNode("r:output", "[1]  1.4965313 -0.4305708")
```

Here, newXMLNode() cannot resolve the namespace prefix `r` and so creates a temporary definition for it, e.g.,

```
<r:output xmlns:r="<dummy>">
[1]  1.4965313 -0.4305708
</r:output>
```

Now, when we add this to our `<rc>` node such as

```
addChildren(rc, rOut)
```

addChildren() does the extra work to "repair" the nodes and resolve the namespaces. This takes care of both removing the "dummy" namespace definitions and resolving the prefix, and also putting the default namespace on the nodes if the parent or ancestors define a default namespace.

Repairing the namespaces is merely a convenience. It does incur some computational cost. As a result, it is sometimes best to avoid this. We can override this by telling addChildren() to not repair the namespaces. We do this by specifying `FALSE` for the *fixNamespaces* parameter. We can control whether to resolve the default namespace definitions independently from the "dummy" with

```
fixNS = c(dummy = TRUE, default = FALSE)
addChildren(rc, rOut, fixNamespaces = fixNS)
```

This avoids the overhead of attempting to find an ancestor with a nontrivial default namespace. The newXMLNode() function behaves similarly.

We can repair namespaces easily by serializing the *XML* tree as a string and then parsing it. The *XML* parser puts default namespaces on the subnodes for us. For example, suppose we create the tree

```
<article xmlns="http://docbook.org/ns/docbook">
 <section>
  <title>Namespaces in <xml/></title>
  <para>
Namespaces in <xml/> can appear complicated, but are quite simple.
  </para>
 </section>
</article>
```

All of these nodes should have the *DocBook* namespace as their effective namespace. If we were to create these nodes programmatically, and not set the default namespace (either with `fixNamespaces = FALSE` or not setting the namespace explicitly), the tree would technically be incorrect. For example, we may want to search for nodes such as `<title>` using *XPath* queries. For this to work, the nodes needs to have the proper namespaces, i.e.,

```
getNodeSet(tree, "//db:title",
           c(db = "http://docbook.org/ns/docbook"))
```

would not match our <title> node as it has no namespace. This is why we should ensure that the resulting tree has the correct namespace on each of its nodes.

"Round tripping" the tree by serializing and parsing it with

```
doc = xmlParse(saveXML(tree))
```

would yield the "correct" tree. For example, we can confirm that the <title> node has the appropriate namespace:

```
xmlNamespace(xmlRoot(doc)[["section"]][["title"]])
```

```
[1] "http://docbook.org/ns/docbook"
attr(,"class")
[1] "XMLNamespace"
```

If we just create the *XML* content and then serialize it, this is a reasonable approach.

Not only can we round-trip the tree to fix the default namespaces, we can also clean up the namespaces at the same time. The *NSCLEAN* option of xmlParse() allows us to specify that the parser should remove redundant namespaces. For example, the following document repeats the *DocBook* namespace for each node:

```
<article xmlns="http://docbook.org/ns/docbook">
<section xmlns="http://docbook.org/ns/docbook">
<title xmlns="http://docbook.org/ns/docbook">
  Namespaces in <xml/></title>
<para xmlns="http://docbook.org/ns/docbook">
Namespaces in <xml/> can appear complicated, but are quite simple.
</para>
</section>
</article>
```

When we parse it with

```
doc = xmlParse(txt, options = NSCLEAN)
```

the result is

```
<?xml version="1.0"?>
<article xmlns="http://docbook.org/ns/docbook">
<section>
<title>Namespaces in <xml/></title>
<para>
Namespaces in <xml/> can appear complicated, but are quite simple.
</para>
</section>
</article>
```

Before we look at some more involved examples of creating nodes, let us note that the <xml> node in the *DocBook* examples above illustrates a problem. We have used a default namespace (`http://docbook.org/ns/docbook`) for all of the other nodes. However, the <xml> node should not use this namespace. It is not possible when we serialize an *XML* document to have a default namespace and to have a subnode with no namespace. It is possible to do this when working with node and tree objects. We can explicitly set the namespace of the <xml> node to `NULL`, e.g.,

```
x = getNodeSet(doc, "//x:xml",
               c("x" = "http://docbook.org/ns/docbook"))
lapply(x, setXMLNamespace, NULL)
```

or

```
xmlNamespace(x[[1]]) = NULL
```

This is an example of how we have much greater control with nodes than with string representations.

Approaches to Specifying a Namespace When Creating Elements

- **Namespace definition on the element**
  When we create a node with newXMLNode(), we can use the *namespaceDefinitions* argument to define namespaces and the *namespace* argument to associate a particular namespace with the node being created.

  ```
  ns = c(db = "http://docbook.org/ns/docbook",
         r = "http://www.r-project.org")
  exRdb = newXMLNode("example", namespaceDefinitions = ns,
                     namespace = "db")
  ```

  The value of *namespaceDefinitions* should be a named character vector, with a name being the prefix and value the URI identifying the namespace. The default namespace has an empty name, i.e., `""`.
  We can also simply add the prefix to the element name to associate a namespace with the element:

  ```
  newXMLNode("db:example", namespaceDefinitions = ns)
  ```

  Here `db` in the node's name matches the second element of *namespaceDefinitions*.
  If an empty prefix is given for one of the elements in the value of *namespaceDefinitions*, that element is taken as the default namespace. This default namespace is associated with that node, if a prefix is not included on the element name or provided via the *namespace* parameter. For example,

  ```
  ns = c("http://docbook.org/ns/docbook",
         r = "http://www.r-project.org")
  exRdb = newXMLNode("example", namespaceDefinitions = ns)
  ```

  makes the docbook.org URI the default namespace for the new node.

- **Supplying a definition through reference to another node's namespace**
  We can specify the namespace for a node by supplying a reference to a namespace definition in another node.

  ```
  rNS = xmlNamespaceDefinitions(exRdb, TRUE)$r
  newXMLNode("code", namespace = rNS)
  ```

  This reuses the existing definition rather than creating a new namespace definition.

- **Implicit definition through parent specification**
  Once we have defined a namespace on a parent or an ancestor of a node, we can use that namespace by using its prefix, without having to provide a new definition. We supply the

parent node via newXMLNode()'s *parent* parameter so that the function can search the ancestors to resolve the namespace definition by the prefix we use.

```
p1 = newXMLNode("para", "Text", parent = exRdb)
newXMLNode("emphasis", "xxx", parent = p1, namespace = "")
newXMLNode("r:expr", "rep(1:3, 2)", parent = p1)
newXMLNode("output", parent = p1, namespace = "r")
```

The namespace assigned to the <*para*> child of exRdb is the default namespace from the <*example*> node above. This is the docbook.org namespace. The newXMLNode() function also gives the default namespace to the <*emphasis*> node as a grandchild of <*example*>. The <*expr*> and <*output*> elements are given the r-project.org namespace by newXMLNode() as it matches the prefix r in the <*example*> namespace definitions.

- **Delayed namespace resolution**
  When we create nodes with a namespace prefix but without specifying a parent, newXMLNode() cannot resolve the prefix. When the node is eventually added as a child to another node, newXMLNode() or addChildren() "repairs" these unresolved namespace prefixes and also sets default namespaces as appropriate. This extra computation does not occur when we use a namespace reference, even without a parent, e.g.,

  ```
  newXMLNode("code", namespace = rNS)
  ```

- **Removing redundant namespace definitions**
  We can always define namespaces on nodes that use them to ensure they are available and not rely on them being defined in an ancestor node. We can remove these using

  ```
  doc = xmlParseDoc(saveXML(doc), NSCLEAN)
  ```

  This will create entirely new nodes.

6.6.1 Adding Namespaces to Child Nodes

The following example illustrates various approaches for adding namespaces to elements when they are created in *R* and describes their advantages and limitations.

Example 6-5 Generating SDMX Exchange Rates with Namespaces

The *XML* snippet in Example 2-3 (page 33) is a mix of two grammars and so uses two namespaces. We reproduce it here, and we have modified the <*Cube*> names slightly for ease of reference:

```
<g:Envelope
  xmlns="http://www.ecb.int.vocabularly/
   2002-08-01/euroxref"
  xmlns:g="http://www.gesmes.org/xml/2002-08-01">
 <g:subject>Reference rates</g:subject>
 <g:Sender>
  <g:name>European Central Bank</g:name>
 </g:Sender>
 <Cube1>
```

```
  <Cube2 time="2008-04-21">
   <Cube3 currency="USD" rate="1.5898"/>
   <Cube3 currency="JPY" rate="164.43"/>
  </Cube2>
 </Cube1>
</g:Envelope>
```

The default namespace (which therefore needs no prefix) is the grammar developed by the European Central Bank, i.e., `http://www.ecb.int.vocabularly/2002-08-01/euroxref`. The other namespace is `http://www.gesmes.org/xml/2002-08-01` and is referred to by the prefix `g`.

We create this *XML* tree to demonstrate some of the approaches for associating a namespace with an element. To begin, we create the top node *<Envelope>* and define two namespaces and specify which to associate with *<Envelope>*. There are a few combinations of the *namespaceDefinitions* and *namespace* parameters that can both define namespaces and associate a namespace with the node being created. For example, in the following code, we define the namespaces via the *namespaceDefinitions* parameter and specify which to use for the element with the *namespace* parameter:

```
ns = c("http://www.ecb.int...euroxref",
       g = "http://www.gesmes.org/xml/2002-08-01")
exData = newXMLNode("Envelope", namespaceDefinitions = ns,
                    namespace = "g")
```

The result is

```
<g:Envelope
  xmlns="http://www.ecb.int.,,2002-08-01/euroxref"
  xmlns:g="http://www.gesmes.org/xml/2002-08-01">
</g:Envelope>
```

Here we have specified the default namespace as an unnamed element in the character vector `ns`, which we passed to newXMLNode() via its *namespaceDefinitions* argument. We have also identified the namespace prefix for the node by specifying `g` as the value for the *namespace* parameter. We can confirm that the namespace associated with *<Envelope>* is indeed `www.gesmes.org` with

```
xmlNamespace(exData)
```

```
                                    g
"http://www.gesmes.org/xml/2002-08-01"
attr(,"class")
[1] "XMLNamespace"
```

Similarly, we can query the default namespace with

```
getDefaultNamespace(exData)
```

```
"http://www.ecb.int.vocabularly/2002-08-01/euroxref"
```

Instead of specifying the namespace for the node via the *namespace* parameter, we can put a prefix in the name of the node, i.e., `g:Envelope`. Now, we call newXMLNode() as

```
exData = newXMLNode("g:Envelope", namespaceDefinitions = ns)
```

In some cases, we will already have the namespace prefix in the node name so it is easier to pass both via the node name. In other cases, we will have the node name and the namespace prefix separately, and so it is more convenient to use the *namespace* parameter. Which is better depends on the circumstances.

Next, we add the two children <*subject*> and <*Sender*> to the <*Envelope*> node. In each case, we specify the name space prefix and newXMLNode() will resolve it by looking for the corresponding namespace definition in the parent and its ancestors. Again, we can choose whether to specify the namespace prefix in the node name or via the *namespace* parameter. For the <*subject*> element, we specify the prefix via the node name, i.e., "g:subject":

```
newXMLNode("g:subject", "Reference rates", parent = exData)
```

For the other two nodes, we specify namespace prefix via *namespace* with

```
sender = newXMLNode("Sender", namespace = "g", parent = exData)
newXMLNode("name",   namespace = "g", parent = sender,
           "European Central Bank")
```

Notice that the <*name*> element has access to the g namespace definition from its grandparent's namespace definitions, even though we specify only the parent when the node is created. This is because we are working top down and have access to the grandparent's namespace when <*name*> is created. Again, we can confirm that <*name*>, which is the grandchild of <*Envelope*>, has the correct namespace associated with it, i.e.,

```
xmlNamespace(exData[["Sender"]][["name"]])
```

```
                                    g
"http://www.gesmes.org/xml/2002-08-01"
attr(,"class")
[1] "XMLNamespace"
```

Although namespace definitions look as if they are attributes on a node, they are not. That is why we specify them via the *namespaceDefinitions* parameter. However, since this is a common mistake, the newXMLNode() identifies such "errors," corrects them and issues a warning. When you encounter such a warning from your own code, we recommend fixing that code to use the *namespaceDefinitions* parameter.

When we create child nodes in nested *R* calls, we need to be aware of what is happening with the namespaces. For example, to create the <*Cube*> nodes in one call, we might consider doing the following:

```
newXMLNode("Cube1",
  newXMLNode("Cube2",
             newXMLNode("Cube3", attrs = c(currency = "USD",
                                           rate = "1.5898")),
             newXMLNode("Cube3", attrs = c(currency = "JPY",
                                           rate = "164.43")),
             attrs = c("time" = "2008-04-21")),
  parent = exData)
```

In the top-level call to newXMLNode(), we create the <*Cube1*> node. We specified the *parent* node and so we can find the default namespace. However, when we created its sole child node, <*Cube2*>, we did not specify the parent. This is because the <*Cube1*> node is to be its parent. We cannot pass

this top-level <Cube1> node as the parent because we have not created it yet! The same is true of the two <Cube3> nodes as we are creating their parent node in the nested calls to newXMLNode(). This means that these three inner/nested calls to newXMLNode() cannot resolve the namespaces or default namespace. When we add the innermost <Cube3> nodes to their parent (i.e., the intermediate <Cube2> node), newXMLNode() attempts to repair the namespaces. It cannot do this because it too does not have a parent at this point and so cannot look up the hierarchy for namespace definitions. However, when we add the <Cube1> node to the topmost node, we do have a parent—exData. Now we can repair all child nodes being added to this <Cube1> and all of their descendants, too.

Nested calls such as these mean newXMLNode() has to do more computations to repair the descendants. The simplest way around this is to create the parent node first and then create the children and add them to the parent, e.g.,

```
cube1 = newXMLNode("Cube1", parent = exData)
cube2 = newXMLNode("Cube2", parent = cube1,
                   attrs = c("time" = "2008-04-21"))
```

We have made the specification of the parent possible because we created it separately from the other nodes so it is available.

6.6.2 Namespaces on Attributes

A node's attributes can also have namespaces associated with them, and again, we identify these by a namespace prefix defined on a node or one of its ancestors. At the time that a node is created, we can use a namespace prefix in the attribute name. For example, suppose in Example 6-5 (page 215), we wanted to add the g prefix to the *rate* attribute in the <Cube3> element:

```
<Cube3 currency="USD" g:rate="1.5898"/>
```

We can do this in the call to newXMLNode() as

```
cube3USD = newXMLNode("Cube3", attrs = c(currency = "USD",
                                         "g:rate" = "1.5898"),
                      parent = cube2)
```

We simply separate the namespace prefix and the attribute name with a ":" as we do for the name of an *XML* node. Note that we have to quote the attribute name *g:rate* so that *R* can parse the name correctly. The gesmes namespace associated with the prefix g is defined in an ancestor of cube2. When we provide that as the parent, newXMLNode() can resolve the prefix g to find the namespace.

A node's namespace prefixes and definitions are available in the *R* attribute namespaces, e.g.,

```
xmlAttrs(cube3USD)

currency     rate
"USD"    "1.5898"
attr(,"namespaces")
     http://www.gesmes.org/xml/2002-08-01
""                                    "g"
```

When we subset this vector of attributes, the subset of the namespaces is attached.

6.6.3 Using Namespace Reference Objects

We have seen how we can refer to namespaces when creating a new node by using the appropriate prefix defined in one of the ancestor nodes. This requires newXMLNode() and related functions searching up the tree when a new node is created and added as a child in order to resolve the namespace. Of course, we can define the namespace for each node and end up with multiple redundant definitions and require newXMLNode() to create new namespace objects. We can remove these by parsing the tree with xmlCleanNamespaces().

An alternative approach to incurring the cost of resolving a namespace by prefix is to use a reference to the namespace definition in calls to newXMLNode(). If we use a top-down approach, or at least create the root or ancestor node first, we can explicitly create a namespace object for a node and then use that as the value for the *namespace* argument in calls to newXMLNode(). For example, we can create a node with

```
p = newXMLNode("para")
```

We can then define a namespace for that node using newXMLNamespace(). For example,

```
rNS = newXMLNamespace(p, rURI, "r")
```

creates a namespace with `http://www.r-project.org` as the value of `rURI` and `r` as the prefix. Importantly, newXMLNamespace() returns a reference to the newly created *C*-level namespace object. We can pass this to newXMLNode() as the value of the *namespace* parameter when creating new nodes. For example, we can create an <*r:code*> node with

```
newXMLNode("code", namespace = rNS, parent = p)
```

The resulting tree looks like

```
<para xmlns:r="http://www.r-project.org">
  <r:code/>
</para>
```

with the namespace definition on the <*para*> node and used in the <*code*> node. Here, the newXMLNode() function did not need to resolve the namespace. If we did not specify the parent, the namespace would still be defined.

We do not need to explicitly create the namespace definition with newXMLNamespace(). Instead, we can define the namespaces in a call to newXMLNode() via the *namespaceDefinitions* parameter, e.g.,

```
omgURI = 'http://www.omegahat.org'
p = newXMLNode("para",
               namespaceDefinitions = c(r = rURI, omg = omgURI))
```

We can then retrieve references to these namespace definitions with the function xmlNamespaceDefinitions(), using the *ref* parameter to get references rather than the values, e.g.,

```
ns = xmlNamespaceDefinitions(p, TRUE)
```

This returns a list with the two namespace references. We can access the elements using the namespace prefix, e.g., `ns$omg` and pass this to newXMLNode() as the *namespace* parameter value. Again, the purpose of using namespace references is to avoid searching the ancestor nodes to resolve the namespace prefix when we create a new node.

6.7 Working with Alternative Tree Representations to Generate *XML*

Throughout this chapter, we have worked with the tree in *R* as a *C*-level structure via functions such as newXMLNode(). However, we have not needed to be aware of how each element and subtree was represented. The *R* functions hide these details. This makes it easy for us to use alternative representations. Indeed, we described alternative data structures and representations for *XML* trees in Section 3.9 when discussing parsing *XML* documents. We can also use any of these representations when constructing an *XML* tree/*DOM* in *R*, and not just parsing an existing document. In this section, we describe the capabilities available in the XML package for creating and modifying these alternative representations of the *XML* tree, as they use a different computational model. Generally, there is little advantage to working directly with these other representations. We can use the "internal" *C*-level node approach via, e.g., newXMLNode() and convert the resulting tree to these other representations at any time. Importantly, the *C*-level representation allows us to make *XPath* queries on the (sub-) tree at any time. Furthermore, this approach is the most widely used and hence best tested.

6.7.1 Building an XML Tree Entirely with Regular R Objects

We mentioned in Chapter 3 that we can think of an *XML* node in *R* as a type of *list*, with some additional information such as the tag name, a character vector of attributes, and any namespace information. The elements of the *list* are the child elements of the *XML* node. While this was just a conceptual model, the XML package supports working with *XML* elements explicitly as *list* objects in this form, with each node being a list of named elements pertaining to the node (e.g., name, namespace, and attributes) and a list of child nodes. These nodes are regular *R* objects, not references to *C*-level structures and so have more typical *R* semantics, e.g., pass-by-copy, serializable via save(). In addition to parsing an *XML* document into a representation of this form, the constructor function xmlOutputDOM() provides the facility to build and modify this kind of tree and node. Given the nature of lists in *R*, this tree cannot be generated in as flexible a manner as the internal *C*-level representation. Specifically, we do not have references and this makes it impossible to traverse up the hierarchy from a node to is parent. Calling the xmlOutputDOM() function returns a list of functions that operate on the *XML* data in a shared environment. These are

```
tree = xmlOutputDOM()
names(tree)
```

```
[1] "value"     "addTag"     "addEndTag"  "closeTag"
[5] "reset"     "addNode"    "add"        "addComment"
[9] "addPI"     "addCData"   "current"
```

The two core functions needed to build elements in the tree are addTag() and closeTag(). The *R* structure necessitates the explicit closing of tags. With addTag(), we can add a new element to a tree, and when its *close* argument is `FALSE`, the node will remain open (i.e., active) to allow other elements to be nested as children in it. This function has a ... argument so that we can supply text content and additional child nodes. The closeTag() function closes the active node. After closeTag() is called, a subsequent call to addTag() will add a new node to the tree as a sibling after (to the right of) the node just closed.

With this type of construction, we have the notion of a cursor that marks the node to which new nodes will be added as children. This cursor points to the "active" node. The current() function returns

the location in the tree of the active node. This helps us keep track of where we are in building the tree. Also, the value() function retrieves a snapshot of the contents of the *XML* document. In the next example, we demonstrate how to build an *XML* document in this way.

Example 6-6 Generating an XHTML Table from an R Structure

We revisit Example 6-1 (page 185) and demonstrate how to construct an *HTML* table for an *R* data frame, but this time as a hierarchy of *R* list objects using xmlOutputDOM() and its related functions/methods. We begin by creating the "root" node of this subtree with a call to xmlOutputDOM():

```
chipHTML = xmlOutputDOM("table", attrs = c(border = "1",
                                           cellspacing = "2"))
chipHTML$value()

<table border="1" cellspacing="2"/>
```

When the value of `chipHTML` is printed to the terminal, it appears that the <table> node is closed. However, it is actually open, or active, and available for child nodes to be added to it. The "cursor" points to the <table> node, ready to add children via calls to addTag(). That is, we next fill in the tree with calls to addTag() to add the row and cell nodes, i.e., <tr>, <th>, <td>, and text nodes.

The following call adds the first <tr> child node to the table:

```
chipHTML$addTag("tr", close = FALSE)
```

The *close* argument is specified as `FALSE` because we need to add cells to this row (<tr>) element. We add the first <th> node, which has no text content, with

```
chipHTML$addTag("th")
chipHTML$value()

<table border="1" cellspacing="2">
 <tr>
  <th/>
 </tr>
</table>
```

This <th> node was created and closed (since *close* was not specified in the call), so the active cursor points to the <tr> element, which is still open. The next call to addTag() will add another child to that <tr> node, as a sibling to this empty <th> node. We confirm this by examining the value for current():

```
chipHTML$current()

[1] 1
```

The value of 1 indicates the first node in the first level of the tree is active and children will be added to it. Alternatively, a value of `c(2, 1)` for current() denotes the active element is the first child of the second child of the root node.

We add the remaining <th> nodes and their text children with

```
sapply(names(chips), function(x)
                        chipHTML$addTag("th", x))
```

Note that we are adding the text for each <th> element as a child via the ... parameter of addTag(). This is the same mechanism we use in newXMLNode() where we can add other elements or text nodes.

Now that we have added all <th> elements to the first <tr> node, the table header is complete, and we close the table row with a call to closeTag():

```
chipHTML$closeTag()
```

The table has one row:

```
chipHTML$value()
```

```
<table border="1" cellspacing="2">
 <tr>
  <th/>
  <th>Date</th>
  <th>Transistors</th>
  <th>Microns</th>
  <th>ClockSpeed</th>
  <th>Data</th>
  <th>MIPS</th>
 </tr>
</table>
```

Since the sole <tr> element is now closed, new nodes that we add via addTag() will be created as siblings of this node.

We complete the table by adding all of its data rows as siblings to the table's header row. We add the values in each row in the data frame to a new <tr> node with

```
sapply(row.names(chips),
       function(x) {
         chipHTML$addTag("tr", close = FALSE)
         chipHTML$addTag("td", x)
         sapply(chips[x, ], function(y)
                              chipHTML$addTag("td", y))
         chipHTML$closeTag()
        })
chipHTML$closeTag()
```

We have now completed building the table, and we can serialize it to an *HTML* file with

```
saveXML(chipHTML, file = "chipDom.html")
```

in the same we would a *C*-level representation. Indeed, the generic function saveXML() writes any of the representations of an *XML* tree (output *DOM*, hash tree, or internal *C* data structure) to a string or file.

In addition to the addTag() function, the functions addComment(), addPI(), and addCData() can add comments, processing instructions, and <*CDATA*> nodes, respectively, to the tree. Also, the xmlOutputBuffer() function has an interface for building a tree that is similar to xmlOutputDOM(). The main difference between these two functions is that the buffer version stores the *XML*

representation explicitly as a string, rather than as hierarchy of *XML* elements (represented as *R list* objects).

We can create elements in our *DOM* tree with namespaces in the same way as with newXMLNode() and newXMLDoc(), i.e., the *C*-level representations. We can define a collection of namespaces at the root of the document and reference those when creating individual elements by using the prefix identifying the global namespace. Alternatively, we can define a new namespace within a node in a call to addTag() when we create that node. It is difficult in this model to define a namespace on one node below the root, and to reference that in a descendant node other than an immediate child. This is due to the fact that we cannot traverse up the hierarchy of nodes. However, the functions returned by xmlOutputDOM() do manage the default namespace for subnodes. In other words, if we define a namespace on a node and make it the default namespace for that node, subsequent child nodes and descendant nodes will use that as their namespace, by default.

When creating nodes with newXMLNode(), we can create child nodes before their parent node. This is possible using xmlOutputDOM() by creating one subtree for the children and then merging it with a separate tree for the parent. In other words, we have to create two separate subtrees. When working with xmlOutputDOM() trees, we work top-down. This is much less flexible.

As we saw in Section 3.9, we have other ways to represent an *XML* tree entirely with *R* objects rather than using *C*-level data structures. For example, we can use a representation that manages the elements and their hierarchical relationships using *R* `environment` objects. This overcomes many of the difficulties of using the list-of-lists approach to representing the tree. We can use this representation to build the tree via the function xmlHashTree(). Like xmlOutputDOM(), this returns a function to add a node and attempts to manage to which parent element a new node will be added.

There are other approaches and functions in the XML package for creating trees. However, we reiterate that using newXMLDoc() and newXMLNode() and the *C*-level representation of elements is the most flexible and best-tested approach.

6.8 Summary of Functions to Create and Modify *XML*

In this chapter, we have introduced functions available in the XML package that we can use to create and modify *XML* nodes and documents. The functions below can create *XML* nodes of various types (regular nodes, as well as text, `<CDATA>`, etc.) and clone nodes.

newXMLNode() Create an *XML* node with the element name provided. Child nodes can be supplied either as separate arguments (via the ... parameter) or in a list via the *.children* parameter. The node is added as a child of the node in the *parent* parameter, if specified. Also, attributes, namespaces, and namespace definitions are specified via *attrs*, *namespace* and *namespaceDefinitions*, respectively.

newXML...Node() Create other kinds of *XML* nodes, such as text, `<CDATA>`, comment, processing instruction, and *DTD* with, respectively, newXMLTextNode(), newXMLCDataNode(), newXMLCommentNode(), newXMLPINode(), and newXMLDTDNode().

parseXMLAndAdd() Parse the specified string content into an `XMLInternalElementNode` object and add it to the node provided in *parent*. This function is helpful when we use string manipulation to create subtrees as blocks of text. It parses the text into nodes and adds them to a tree.

xmlClone() Create a copy of the *XML* node or document provided. The element will be cloned to create a new *C*-level structure. The *recursive* argument indicates whether all the child nodes will

be cloned as well, or only the top-level node. Cloning is not the same as assignment. When we clone, we make an explicit copy of the *C* data structure and return that copy (which may then be assigned to an *R* variable). Simply assigning an internal node to a variable does not make a copy of it, unlike most objects in *R*, but merely copies the `externalptr` object. As a result, a simple assignment means any subsequent changes to the node will appear in all references to it. In contrast, when we clone a node or document, the original and the copy are independent copies and changes to one are not reflected in the other.

newXMLNamespace() Adds a new namespace definition to the node specified via the *node* parameter. The namespace URI and prefix are provided in *namespace* and *prefix*, respectively.

There are many other functions available in the XML package that we can use to modify an existing *XML* document to, e.g., add and remove child nodes, replace nodes, add and remove attributes on nodes, and access and change nodes. These are summarized below.

addChildren() Add *XML* nodes to an existing node. The child nodes are provided either as separate arguments (via the ... parameter) or in a list via the *kids* parameter. A child can be a string, in which case a text node with that content is added, or any type of internal *XML* node, such as an element, comment, etc. The nodes are added after the parent's existing children, unless the *at* argument provides the index of the child after which the new children are to be added. Multiple values in the *at* argument can be used to place the new children at different locations.

addSibling() Add nodes next to, as a sibling of, a specified node. The first argument is the existing sibling node, and the new siblings are supplied either through ... or the *kids* parameter. Nodes are inserted either after or before the sibling, according to the *after* parameter value (`TRUE` or `FALSE`, respectively).

removeChildren(), removeNodes() Eliminate elements from the parent node. The child elements are specified either by index (recommended) or by element name. If there are multiple elements with the same name, only the first of these is removed. The removeNodes() function removes the nodes provided.

replaceNodes() Replace one node with another. The node to be replaced is supplied in *oldNode* (the first parameter) and the new node is supplied via *newNode* (the second parameter).

addAttributes() Add attributes to the given node. The attributes are provided either in ... as `attribute_name = "value"`, or via the *.attrs* parameter with a named character vector where the element name corresponds to the attribute name and value corresponds to the value of the attribute.

removeAttributes() Remove attributes from the node provided. The names of the attributes are provided either in ... or as a character vector of attribute names via *.attrs*.

xmlName(), xmlValue(). xmlParent(), xmlAttrs() The assignment version of each of these functions, e.g., `xmlAttrs<-()`, reassigns the node name, text value, parent, and attributes, respectively, for an existing node.

6.9 Further Reading

When generating *XML* content, a key idea is the Document Object Model (*DOM*). This concept is covered in Chapter 3. In addition, Chapter 11 of [2] and Chapter 19 of [1] describe the *DOM*.

References

[1] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[2] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.

[3] Eric Lecoutre. R2HTML: *HTML* exportation for *R* objects. http://cran.r-project.org/package=R2HTML, 2011. *R* package version 2.2.

[4] Deborah Nolan and Duncan Temple Lang. RKML: Simple tools for creating *KML* displays from *R*. http://www.omegahat.org/RKML/, 2011. *R* package version 0.7.

[5] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

Chapter 7
JavaScript Object Notation

Abstract In this chapter, we describe the *JSON* format and then the fromJSON() and toJSON() functions to both read and create *JSON* content. Because *JSON* is so simple and there are few supporting technologies for *JSON*, there are not many details that we need to examine before being able to work with *JSON* effectively. As a result, we spend a significant part of this chapter illustrating how we work with *JSON* in different contexts. These display other technologies that use *JSON* which are interesting in their own right. We look at reading *JSON* data from both Web services and from local files. We also show how to serialize data from *R* to *JSON* in order to add it to interactive *HTML* and *SVG* documents. Finally, we explore the *ElasticSearch* text search engine and how we use *JSON* to both insert documents into the engine and receive query results from the engine.

7.1 Introduction: Sample *JSON* Data

While the majority of the previous chapters have focused on *XML*, it is of course not the only format for representing data. There are many other formats, ranging from general-purpose to application-specific, open standards to proprietary, and both text and binary formats. Each of these has their merits and demerits, but we are interested in general-purpose formats defined by open standards or specifications. Why? Because these formats are used across a wide spectrum of different applications, so they are worth learning for general use. *JSON* [17], *JavaScript* Object Notation, is an important format that is commonly used for Web services (see Chapter 10), client-server applications such as (*NoSQL*) databases, and simply as a means of storing and exchanging data between different applications. *JSON* is a simple, lightweight format. Its origins stem from the *JavaScript* [11] language syntax for creating objects and so is familiar to *JavaScript* programmers. Furthermore, data given in *JSON* format can be used directly in *JavaScript* code such as in Web pages. All of these characteristics make *JSON* a valuable tool for working with data and especially Web-related data such as coming from Web services or being displayed in browsers.

The following shows data in *JSON* format, taken from *New York Times* Web service documentation [26] (`http://developer.nytimes.com/docs/campaign_finance_api/campaign_finance_api_examples#candidate-leaders-json`). These data illustrate the full set of data types and how they are represented in *JSON*. The data values and types are *null*, *true*, *false*, number, string, and two types of containers— *[]* and {}.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_7,

```
{     1
  "results": [  2
      {
        "city": "NEW YORK",  3
        "address": "34 WEST 38TH ST - FLR 5",
        "name": "AMERICANLP",
        "zip": 10018,  4
        "treasurer": "TJ WALKER",
        "super_pac": true,
        "relative_uri": "/committees/C00507244.json",
        "candidate": null,  5
        "id": "C00507244",
        "leadership": false,  6
        "sponsor_name": null,
        "party": "",
        "fec_uri": "http://query.nictusa.com/.../C00507244/",
        "state": "NY"  7
      },
...
      {
        "city": "LANSING",
        "address": "313 S WASHINGTON SQUARE",
        "name": "TEA PAC",
        "zip": 48933,
        "treasurer": "CHERYL SHERRELL",
        "super_pac": true,
        "relative_uri": "/committees/C00504464.json",
        "candidate": null,
        "id": "C00504464",
        "leadership": false,
        "sponsor_name": null,
        "party": "",
        "fec_uri": "http://query.nictusa.com/.../C00504464/",
        "state": "MI"
      }
  ],  8
  "base_uri": "http://api.nytimes.com/.../finances/2012/",
  "copyright": "Copyright (c) 2011 The New York Times Comapny...",
  "cycle": 2012,
  "status": "OK"
}
```

1 The } delimits the *JSON* object. We can think of the content of this object, in *R* terms, as a list with five named elements: *results*, `base_uri`, `copyright`, `cycle`, and `status`.

2 The first element, *results*, is an ordered collection of individual result objects. That is, the [indicates that *results* is a simple ordered array.

3 Each element of *results* is an object with many fields such as `city`, `address`, `name`, `zip`, etc.
4 The `zip` field is a number. Although in the original source, these were represented as strings.
5 The `candidate` and `sponsor_name` fields have the special value *null*.
6 The value of each of the `leadership` and `super_pac` fields is a logical value: *true* or *false*.
7 Many of the values are strings.
8 The last four elements of this object are simple strings named `base_uri`, `copyright`, `cycle`, and `status`.

The structure of *JSON* is quite straightforward. There are tools for almost all programming languages to convert *JSON* content to values in that language. Similarly, there are tools for converting values to *JSON* in most languages. The functions to do this in *R* are fromJSON() and toJSON(). These are available from either the `rjson` [7] or RJSONIO [24] packages. The latter provides some additional flexibility and functionality and builds on an open source *C++* parsing library.

There are many people who compare *JSON* and *XML* and make claims that one dominates the other. For data analysts, such debate is largely irrelevant because we are typically given the data in a particular format and have to work with that. We do not have a choice. When we do have a choice, which format we choose should depend on many criteria and not just on, say, the immediate/short-term simplicity of format. *JSON* is often simpler to work with for direct serialization and deserialization of data. *XML* is richer, more flexible, and structured. *XML* is accompanied by a broad collection of well-established features and technologies such as namespaces, schema, *XPath*, *XSL*, *XInclude*, and *XPointer*. There are some suggested tools analogous to *XPath*, e.g., JPath [10], JSONPath [13], and JSON Schema [25]. Unfortunately, these are not standards or widely used and implementations are not available for many languages.

We advocate considering the long-term value and benefit of structured, self-describing data that contains as much metadata as possible. This push towards the "semantic Web" can lead to a qualitatively improved means of finding and understanding meaningful data. While both *XML* and *JSON* are capable of supporting this extra dimension of data, the philosophy of *JSON* is quite different.

7.2 The *JSON* Format

The *JSON* format is very simple. As with most data-oriented computer languages, *JSON* has common primitive data types: boolean/logical (*true* and *false*), number, and string. Unlike in *R*, these are scalars in *JSON*. In *JSON* there is only one type of number—a real or floating-point value—which is a *numeric* in *R* terms. *JSON* does not have the notion of a missing value, infinity, or "not a number", i.e., *R*'s `NA`, `Inf`, `NaN`, respectively. It does have the notion of *null*, the empty "object."

These four types (boolean, number, string, and *null*) make up the entire collection of scalar values, and so there is a reasonably obvious mapping between *JSON* and *R* primitive data types. Specifically, *JSON* scalars map to *R* vectors with length 1 and vice versa. The *true* and *false* map to `TRUE` and `FALSE`, respectively; *null* maps to `NULL`; and numbers map to *numeric* vectors. A *JSON* string maps to a character vector of length 1.

In addition to the scalar types, *JSON* has two container data types for collections of zero or more values. In *R*, these correspond to unnamed and named vectors or lists. In other languages, these are often referred to as simple ordered arrays and associative arrays. In *JSON*, these are termed arrays and "objects," respectively. The simple, ordered, unnamed array is identified by the notation

```
[ value, value, value, ... ]
```

That is, an opening [and a closing]. Named arrays or "objects" use { and } and have the form

```
{ "name" : value, "name" : value, ...}
```

The quotes are necessary for the names or "keys." In both types of containers, elements are separated by a comma.

Containers can be nested, i.e., an element of a container can itself be a container (or a scalar). This allows nesting of values and gives *JSON* the flexibility to represent arbitrary data structures.

Unlike *R*, there is no distinction between a collection of homogoneous values and nonhomogeneous values, i.e., *vector* and *list* in *R*. In other words, *JSON* ignores the fact that a collection contains values of the same type and just has the notion of a container, either named or ordered. As a result, it is easy to map from *R* to *JSON*. An *R* vector maps to an array in *JSON* if it has no names, and it maps to an object/associative array if it has names. Additionally, a data frame maps easily to a *JSON* object consisting of a named list of arrays. That is, suppose we have observations for each of four variables for three earthquakes in a data frame. We might represent this in *JSON* as

```
{
 "mag":  [ 7.2,  7.1,  7 ],
 "long": [ 63.943, -73.349, -63.091 ],
 "lat":  [ 28.784, -38.372, -26.788 ],
 "location": [ "SOUTHWESTERN PAKISTAN", "CHILE", "ARGENTINA" ]
}
```

An important caveat is that toplevel *JSON* content cannot be a simple primitive value. Instead, it must be a container, either named or not. This means that we would never have *JSON* content of the form *1*, *true* or `"xy" : [1, 2]`. Instead, we would need to have, respectively,

```
[ 1 ]
```

or

```
[ true ]
```

or

```
{"xy" : [1, 2] }
```

Note the white-space between values in *JSON* content is ignored, unless of course it is within a string.

The mapping in the other direction, i.e., from *JSON* to *R* has some ambiguity. For example, an array in *JSON* can map to either a *list* or *vector*.

The two main operations when dealing with *JSON* in *R*—converting *JSON* content to *R* objects, and converting *R* objects to *JSON*—are handled by the functions named fromJSON() and toJSON(), respectively. In most situations, these functions do the correct thing and so are very simple to use. The fromJSON() function can read content from a file, *URL*, or directly from a string, i.e., in memory. It processes the bytes from this source and converts the values into *R* objects and returns a single *R* object containing the subelements. The toJSON() function takes an *R* data object (i.e., not a language object, such as calls and expressions) and generates the *JSON* representation as a single string. This can then be, for example, added to a file, sent as part of an *HTTP* request, or sent to another application. Examples of each of these are shown later in this chapter.

7.2.1 Converting from JSON to R

As we mentioned, we typically only need to pass fromJSON() the source of the *JSON* content and it will return the desired *R* object. Unlike processing an *XML* document where we frequently parse it and then use getNodeSet() to fetch and process the pieces of interest, fromJSON() parses and processes all of the subelements in the document. We then extract the subcomponents of interest. When we are extracting these, we often transform them in the same or related operation, e.g., setting specific values to `NA`, converting strings to date or date-time values, splitting a string into a collection of numbers, collapsing a list into a vector, setting the class of an object, or simply dropping specific elements. Some general processing can be controlled by fromJSON()'s optional parameters. We will discuss these inputs in this section.

We can pass *JSON* content directly to fromJSON() as a string, as opposed to the name of a file or *URL*. The fromJSON() function will endeavor to distinguish between the string being actual content or the name of a file/*URL*, and it will generally not have any difficulty. However, it is advisable to indicate that the string is the actual content. We do this using the *asText* parameter (with `asText = TRUE`), e.g.,

```
fromJSON('[ 1, 2, 3]', asText = TRUE)
```

Or, we can pass the string as an instance of the `AsIs` class. This can be done via a call to I(), e.g.,

```
fromJSON(I( '[ 1, 2, 3]' ))
```

The fromJSON() function is designed to give sensible results for mapping from *JSON* to *R* objects. As we indicated above, there is ambiguity in mapping from *JSON* containers to *R* *list*s and *vector*s. Also, the *null* value in *JSON* can represent different values such as `NULL`, `NA`, or `NaN` in *R*. Because of these ambiguities, the fromJSON() function allows the caller to control how the mapping is performed. Next, we describe the two primary parameters of fromJSON() to control these mappings.

simplify
Consider the *JSON* content

```
[ 1, 2, 3]
```

How would we map this to an *R* object? A numeric vector with three elements seems entirely natural. However, it can also be treated as a list with three elements. If we have *JSON* content

```
[ 1, "2", 3]
```

then this should map to a list as the elements are not of the same type. But how should

```
[ true, 3.1415, 4]
```

be converted to *R*? We can keep this as a list with three elements since the logical/boolean value is of a different type from the two numbers. But as *R* users know, we can also think of elevating the "true" value to the number 1. The same choices apply when working with associative arrays/objects as well as arrays.

The *simplify* argument allows the caller to control how lists are collapsed to vectors when possible. We can specify several different possible values for this argument. The value `FALSE` maps all containers to *R* *list* objects and does no collapsing of scalar elements into vectors. If *simplify* is `TRUE`, the elements of each container are combined in a manner equivalent to calling the c() function in *R*. This means that scalars are coerced to a common type. In our example *JSON* content above, `[ 1, "2", 3]` would result in a character vector `c("1", "2", "3")`. Similarly, the array `[ true,`

`3.1415, 4]` would result in a numeric vector `c(1, 3.1415, 4)`, i.e., the logical value would be coerced to a numeric value.

We can also specify one of `Strict`, `StrictLogical`, `StrictNumeric`, or `StrictCharacter` value for *simplify*. These enumerated constants control the types of scalar elements that can be combined into a vector, for containers with elements of the same type, i.e., containers of homogeneous element types. Using the symbolic name `StrictLogical` causes containers made up of only boolean values to be converted to an *R* *logical* vector. Similarly, `StrictNumeric` collapses containers of numbers to *numeric*, and `StrictCharacter` yields a *character* vector for containers consisting of only strings.

By specifying one of `StrictLogical`, `StrictNumeric` or `StrictCharacter`, containers of that type are collapsed to *R* vectors. If we want containers of either logical or numeric elements, we specify this by adding the constants `StrictLogical` and `StrictNumeric`, e.g.,

```
fromJSON(I('[ [true, false, true],  [1, 3, 4.15],
               ["a", "b", "def" ] ]'),
         simplify = StrictLogical + StrictNumeric)

[[1]]
[1]  TRUE FALSE  TRUE

[[2]]
[1] 1.00 3.00 4.15

[[3]]
[[3]][[1]]
[1] "a"

[[3]][[2]]
[1] "b"

[[3]][[3]]
[1] "def"
```

We see that the logical and numeric containers are collapsed to vectors and the container of strings is left as a list. We can combine any combination of `StrictLogical`, `StrictNumeric`, and `StrictCharacter` by adding them together. Indeed, `Strict`—the default value of `simplify`—is the sum of these three values.

Note there is (currently) no mechanism to allow collapsing of scalars of different types other than using `simplify = TRUE` and collapsing all scalars. That is, we cannot collapse a container consisting of just scalars and numbers, but exclude those containers that also contain strings.

simplifyWithNames

When discussing above how we can simplify *JSON* containers to *R* vectors, we mentioned that the logic applies to both simple and associative arrays. Clearly,

```
{ "a" : 1, "b" : 2, "c" : 3}
```

can be mapped to a named *R* numeric vector

```
c(a = 1, b = 2, c = 3)
```

However, there are occasions when we want to treat associative arrays as "objects" and leave them as *list*s in *R*. If *simplify* is enabled (i.e., not `FALSE`), we can prohibit collapsing associative arrays that would otherwise be collapsed by specifying `FALSE` for the *simplifyWithNames* parameter.

nullValue
It is natural to map the *JSON* *null* value to *R*'s `NULL` object. However, consider the array *[1, 2, null, 4]*. Here we might want *null* to mean `NA` or alternatively map it to some context-specific value. We can, of course, post-process the *R* object and its subelements and convert such values to whatever we want. However, when we can do this globally, this will be simpler and faster. The *nullValue* parameter takes an arbitrary *R* object. Wherever the *JSON* *null* value is encountered in the *JSON* content, that *R* object is inserted. In our example above, we can replace the *null* with an `NA` via

```
fromJSON(I( "[1, 2, null, 4]"), nullValue = NA)
```

```
[1]  1  2 NA  4
```

Note that the result is a *numeric* vector. If we had not mapped *null* to `NA`, the result would have been a *list* with four elements since the `NULL` value prohibits us from collapsing the container to a vector.

stringFun
The omissions of data types in the *JSON* format has lead to different conventions to overcome them. For example, people represent dates in several different ways. They use the format `"date"` and `"newDate(1335795539000)"` and `"/Date(1335795539000)/"`. The number is the number of milliseconds since midnight January 1st, 1970. (See `http://weblogs.asp.net/bleroy/archive/2008/01/18/dates-and-json.aspx`.) We can transform such a value after we have converted the entire *JSON* content to an *R* object, e.g.,

```
obj = fromJSON(txt)
convertDate(obj$loans[[4]]$date)
```

This involves traversing the entire *R* object a second time. A better solution is to convert the string as we encounter it in the *JSON* stream and before we insert it into the resulting *R* object. The *stringFun* parameter allows us to do exactly this.

The *stringFun* parameter can be either an *R* function or a compiled routine. If *simplify* is `TRUE` and *stringFun* is provided, the routine/function in *stringFun* gets processed before the simplification. For purposes of speed, the compiled routine is greatly preferred. However, it is much easier to deploy an *R* function. For example, the following function checks to see if the string it is passed is one of the forms of "Date" described above. If it is, it converts the string inside *Date* or *newDate* to a number and then to a `POSIXct` object. If the string is not a date, the function just returns the string unchanged. The function is defined as

```
convertJSONDate =
function(x)
{
  if (grepl("/?(new )?Date\\(", x)) {
    val = gsub(".*Date\\(([0-9]+)\\).*", "\\1", x)
    structure(as.numeric(val)/1000,
              class = c("POSIXct", "POSIXt"))
  } else x
}
```

Now we can use convertJSONDate() in converting *JSON* content. For instance,

```
txt = '{ "magnitude": 3.8,
         "longitude": -125.012,
         "latitude": 40.382,
         "date":  "new Date(1335515917000)",
         "when": "/Date(1335515917000)/",
         "country": "USA",
         "verified": true
       }'
fromJSON(txt, stringFun = convertJSONDate)

$magnitude
[1] 3.8

$longitude
[1] -125.012

$latitude
[1] 40.382

$date
[1] "2012-04-27 01:38:37 PDT"

$when
[1] "2012-04-27 01:38:37 PDT"

$country
[1] "USA"

$verified
[1] TRUE
```

We see from the output that the two dates were converted, but the string `"USA"` remains unaltered. Suppose we know that the creator of the *JSON* represented dates in this manner and also represented `NA` as `"NA"` and ∞ as `"Inf"`. We can perform all of these conversions with the following function:

```
fromJSON('[ 1, "NA", "Inf", "/Date(1335515917000)/", "XML"]',
         stringFun = function(x) {
                        if(x == "NA")
                           NA
                        else if(x == "Inf")
                           Inf
                        else
                           convertJSONDate(x)
                })

[[1]]
[1] 1

[[2]]
```

```
[1] NA

[[3]]
[1] Inf

[[4]]
[1] "2012-04-27 01:38:37 PDT"

[[5]]
[1] "XML"
```

This also illustrates that, of course, the function can be defined in the call.

The *stringFun* function will be invoked for each string value in the *JSON* content. This does not include the strings that are the names of fields in objects/associative arrays. These are assumed to be literal strings and so do not need to be processed in any special way. If the *JSON* content has many string values, the calls to the *R* function can become costly in terms of time. It will be significantly faster to use a compiled (*C/C++*) routine. The routine should take the value of the string from *JSON* (and the character encoding in effect) and return an *R* object. This can be any *R* object, including, of course, an *R* character vector consisting of the original string.

We can tell fromJSON() to use this routine in one of several ways. The basic approach is to pass the name of the routine. For example, the RJSONIO package provides a routine named `R_json_dateStringOp` in the compiled code. We can use it with the simple call

```
fromJSON('[ 1, "/Date(1335809312)/", "XML"]',
         stringFun = "R_json_dateStringOp")

[[1]]
[1] 1

[[2]]
[1] "2012-04-30 11:08:32 PDT"

[[3]]
[1] "XML"
```

We see that the date was converted to a `POSIXct` object and the other string, `"XML"`, remains unaltered. The `R_json_dateStringOp` routine looks for `"/Date("` or `"/new Date("` at the beginning of the string. If it finds either of these, it processes the content and returns a `POSIXct` object. If these prefixes are not found, the routine returns the string as an *R* character vector.

Using *C* Routines

When fromJSON() sees the name of a routine as the value of *stringFun*, it calls getNativeSymbolInfo() to find the routine by looking through the loaded dynamic shared objects (DSOs). There is a chance that we might find another routine with the same name earlier in this lookup. That would be very bad. Accordingly, it is often best to call getNativeSymbolInfo() ourselves and specify the name of the DSO as well. This removes any possible ambiguity of finding a routine in another DSO with the same name.

```
sym = getNativeSymbolInfo("R_json_dateStringOp", "RJSONIO")
fromJSON('[ 1, "/Date(1335809312)/", "XML"]', stringFun = sym)
```

How do we define our own routines? We can use *stringFun*, for example. The routine takes both the *C*-level string (i.e., a pointer to a `char`) and the encoding for strings as its two inputs. It returns a `SEXP` so its signature is

```
SEXP  routine(const char *value, cetype_t encoding);
```

It is probably simplest to copy the definition of `R_json_dateStringOp` from the RJSONIO source. To implement the routine, you should be familiar with the *R* API described in [22] or the *C/C++* API provided by the `Rcpp` package [9].

7.2.2 Creating JSON from R

As with the fromJSON() function, converting an *R* object to *JSON* is simply a matter of calling toJSON(). This takes an arbitrary *R* data object (but not a language object such as a function, expression or call) and returns a string giving the *JSON* representation of that data. The conversion from *R* to *JSON* is quite intuitive. Basically, named vectors or lists are mapped to associative arrays, and unnamed vectors are mapped to simple *JSON* arrays. For example,

```
toJSON( c(1, 4, -6))
```

yields

```
[      1,      4,     -6 ]
```

Similarly,

```
toJSON(list(val = 1:3, other = 3.1415,
            b = c(x = TRUE, y = FALSE, z = TRUE),
            states = state.abb[1:5]))
```

gives

```
{
 "val": [ 1, 2, 3 ],
 "other": 3.1415,
 "b": {
    "x": true,
    "y": false,
    "z": true
  },
 "states": [ "AL", "AK", "AZ", "AR", "CA" ]
}
```

In this second example, we see the names of the vectors and lists being used to create associative arrays and simple arrays created from vectors without names, e.g., the state abbreviations. We can also see that vectors of length one are serialized as scalars in *JSON* and not containers with a single element. However, if we serialize a scalar at the top level, we do get a container. For instance, consider the following two operations:

```
toJSON(1)
```

```
[      1 ]
```

```
toJSON(c(a = 1))
```

```
{
 "a":       1
}
```

Since *JSON* content must be given in a container, we generate an array and an associative array. It is at the secondary level (and all subsequent levels also) that scalars are represented as simple values and not containers.

Logical values are serialized to *JSON* as *true* and *false*. Numbers, either integer or numeric values, are serialized as numbers, but the number of digits in a real-valued number is controlled by the *digits* argument. For example, if we want 20 digits, we can serialize as

```
toJSON(pi, digits = 20)
```

```
[ 3.1415926535897931159 ]
```

Character vectors are serialized as containers or single values depending on the level. However, each element is contained within double quotes, e.g., `"a string"`. Single quotes are not valid. These rules for logicals, numbers, character vectors, and vectors and lists, either named or not, are all that we need for serializing any general *R* object.

How are matrices or data frames mapped to *JSON*? A matrix is really a vector in *R*, so if serialized directly it would be represented as a single container with `nrow() * ncol()` elements. This is often not what we want. Instead, in *JavaScript* and other consumers of *JSON*, we might want to represent this as a container with each row represented as its own container. For example, the 5 by 3 matrix consisting of the elements 1–15 is serialized as

```
toJSON(matrix(1:15, 5, 3))
```

```
[ [ 1, 6, 11 ],
  [ 2, 7, 12 ],
  [ 3, 8, 13 ],
  [ 4, 9, 14 ],
  [ 5, 10, 15 ] ]
```

This serialization is achieved by specifying a method for toJSON() for a `matrix` object, e.g.,

```
setMethod("toJSON", "matrix",
          function(x,
              container = .level == 1L || length(x) > 1 ||
                          length(names(x)) > 0,
              collapse = "\n", ..., .level = 1L,
              .withNames = length(x) >  0 &&
                           length(names(x)) > 0,
              .na = "null",
              .escapeEscapes = TRUE, pretty = FALSE) {
               ......................
          })
```

We can define methods for any *R* class we want using this approach. For example, the `googleVis` package [12] defines methods for the `POSIXct`, `POSIXlt` and `Date` classes. As we mentioned

earlier, *JSON* does not specify how to represent dates or date-times. As a result, both the producer and consumer of the *JSON* must agree on how to represent them. Several different suggestions have been proposed. One is to identify a date-time or date as a string that is surrounded by / at either end and has a *Date()* constructor call surrounding the value. For example, we might have a value such as `"/Date(1198908717056)/"`. Now to convert a POSIXct object to this form, we would define a method as

```
setOldClass(c("POSIXct", "POSIXt"))
setMethod("toJSON", "POSIXct",
          function(x,
              container = .level == 1L || length(x) > 1 ||
                            length(names(x)) > 0,
              collapse = "\n", ..., .level = 1L,
              .withNames = length(x) >  0 && length(names(x)) > 0,
              .na = "null",
              .escapeEscapes = TRUE, pretty = FALSE) {
          sprintf("\\/Date(%.0f)\\/", as.numeric(as.POSIXlt(x)))
  })
```

When *R* values are serialized to *JSON*, missing values, `NA`, cause problems since there is no corresponding entity in *JSON*. One approach is to map `NA` to *JSON*'s *null* value. This mapping loses the unambiguous meaning of the value, but is a reasonable compromise. This is the default, e.g.,

```
toJSON(c(1, NA, TRUE, NA))
```

```
[      1, null,      1, null ]
```

In other cases, we might want to use a specific value to identify a missing value. This is common in CSV and other formats where we use some unlikely value to identify a missing value, e.g., -9999. We can automatically replace all `NA` values with a particular *JSON* value via the *.na* parameter in toJSON(). For example, to map `NA` to -9999, we can use the command:

```
toJSON(c(1, NA, TRUE, NA), .na = -9999)
```

```
[      1, -9999,      1, -9999 ]
```

As with all regular calls to toJSON(), the *.na* argument is a global option that applies to all elements of all vectors processed recursively. If we need to handle different vectors at various levels differently, they should be pre-processed before calling toJSON(). Alternatively, the result of toJSON() can be post-processed, but this is quite difficult.

7.3 Validating *JSON*

There are times when it is useful to be able to verify whether *JSON* content is valid. This may be after we have generated *JSON* content ourselves and want to ensure it is correct. Alternatively, we may receive *JSON* content from a Web service or download a file from a Web site and may not know whether it is valid *JSON*. The function isValidJSON() checks *JSON* content and returns `TRUE` if the content is valid, and `FALSE` otherwise.

As we mentioned, converting to and from *JSON* is typically very simple. We will take the opportunity of showing how to work with *JSON* to also illustrate some additional technologies.

7.4 Examples

In this section we provide three examples of *JSON*. In the first example, we read *JSON* data from a Web service and a large local file. Next we show how to serialize data from *R* to *JSON* in order to add it to an *HTML* document. In the third example, we explore the *ElasticSearch* text search engine and how we use *JSON* to both insert documents into the engine and receive query results from the engine.

7.4.1 Reading JSON Content from Kiva Files

Kiva [18] is a nonprofit organization that "connect[s] people through lending to alleviate poverty." Essentially, it allows people like you and me to volunteer small amounts of money to loan to people around the world who use it for such things as starting a small business. It is interesting to explore the types of activities and people who receive loans and also the characteristics of those who give loans. Kiva makes the data about loans and lenders available via a Web service, but also dumps the entire database of loans and lenders in both *JSON* and *XML* format. We will explore the *JSON* format.

First, we retrieve information about the most recent loans that have been granted. The API for Kiva's Web services is described at `http://build.kiva.org/api`. We use this to find the URL for our request at `http://api.kivaws.org/v1/loans/newest.json`. The `'json'` at the end of the *URL* indicates that we want the result as *JSON* content, and not *XML*. We make the request with

```
u = "http://api.kivaws.org/v1/loans/newest.json"
loansJSON = getURLContent(u)
```

The `loansJSON` object is now a character vector with a single element that contains the *JSON*-formatted data. It looks something like

```
{
 "paging": {
      "page": 1,
      "total": 2987,
      "page_size": 20,
      "pages": 150
     },
 "loans": [
            {
             "id": 4.1610e+05,
             "name": "Gertrude",
             "description": {
                   "languages": "en"
                 },
             "status": "fundraising",
             "funded_amount": 0,
             "basket_amount": 0,
             "image": {
                "id": 1.0732e+06,
                "template_id": 1
              },
```

```
            "activity": "Fruits and Vegetables",
            "sector": "Food",
            "use": "To buy more boxes of fruits and vegetables ",
            "location": {
               "country_code": "UG",
               "country": "Uganda",
               "town": "Kampala",
               "geo": {
                  "level": "town",
                  "pairs": "0.315556 32.565556",
                  "type": "point"
               }
            },
            "partner_id": 112,
            "posted_date": "2012-04-22T16:50:03Z",
            "planned_expiration_date": "2012-05-22T16:50:02Z",
            "loan_amount": 50,
            "borrower_count": 1
         },
          ......................
      ]}
```

This is a *JSON* "object" with two fields, `paging` and `loans`. The `paging` field is itself an object with four fields of its own, all numbers. The `loans` field is an array of objects. Each element in the array describes a loan with fields for such things as the loan identifier, the name and location of the person looking for the loan, the more specific purpose of the loan, and the amount being sought (`loan_amount`).

Having completed the Web service request, we can convert the *JSON* text to *R* with

```
loans = fromJSON(loansJSON)
```

The result is a list with two elements: `paging` and `loans`. The `paging` element is a *numeric* vector with four elements:

```
loans[[1]]
```

```
     page     total page_size     pages
        1      2987        20       150
```

The fromJSON() function has collapsed this *JSON* object to a vector rather than keeping it as a list. It does this because the four fields have the same data type, i.e., a number. If we wanted to avoid collapsing the result to a vector and instead keep the object as a list, we can use *simplify = FALSE* in the call to fromJSON(). However, this will affect all objects that we can collapse into a vector. For instance, the image field within each loan is naturally transformed to a vector, e.g.,

```
loans[[2]][[1]]$image
```

```
         id template_id
    1073195           1
```

If we use *simplify = FALSE*, then this too will be returned as an *R list* as will the loan's `geo` element—a collection of three strings.

We now turn our attention to the lenders. We can get these with

```
lU = "http://api.kivaws.org/v1/lenders/newest.json"
lenders = fromJSON(getURLContent(lU))
```

For each lender, we can query his or her previous loans. We can create the *URL* for querying each lender and then retrieve and convert the *JSON* that describes each loan for that lender:

```
u = sprintf("http://api.kivaws.org/v1/lenders/%s/loans.json",
             sapply(lenders[[2]], `[[`, "lender_id"))
info = lapply(u, function(u) fromJSON(getURLContent(u)))
```

We can look at the distribution of the number of loans for these lenders with

```
table(sapply(info, function(x) length(x$loans)))
```

```
 0  1  2  3  4  5
12 28  6  1  2  1
```

We can compute information about the loan amounts with

```
summary(unlist(lapply(info, function(x)
                              sapply(x$loans, `[[`,
                                     "loan_amount"))))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  125.0   793.8  1200.0  2005.0  2606.0 10000.0
```

Let's turn our attention to the complete set of loans and lenders rather than just the most recent. We download the entire database as a **zip** file from Kiva. This is as a collection of *JSON* files. We compute the names of the files with

```
filenames = list.files("Kiva/JSON/lenders", full.names = TRUE)
```

We can then read each file and convert its contents into an *R* object with

```
lenders = lapply(filenames, fromJSON)
```

We then collapse the lenders across files into a single list with

```
lenders = unlist(lapply(lenders, function(x) x[[2]]),
                 recursive = FALSE)
```

Note that not all lender objects have the same fields. For example, some have a `country_code` element and others do not. The fields in the different loan descriptions are even more diverse.

7.4.2 Putting Data into JavaScript Documents

Since the *JSON* format is based on *JavaScript* and *JSON* content is valid *JavaScript*, we can easily write *R* objects into *JSON* content and insert them into *JavaScript*. This way we can include *R* objects (in *JSON* format) in *HTML*, *SVG*, and *Flash* documents in order to create interactive documents and graphics for display on the Web. For example, when we use the *JavaScript*-based visualization toolkit Google Visualization [15] to create a display, we explicitly need to include the data in the

JavaScript code. Also, when we add interactivity to an *SVG* or *Flash* document, the event handlers often need data to implement the interactivity, where these data are provided as *JSON* content within the document.

The common idea here is that we are using *R* to create a document to be displayed at a later time when *R* is no longer running. The document contains *JavaScript* code and that code needs access at viewing time to some of the data we had in *R* when we created the document. (Note that if *R* is embedded within the Web browser, then an *R* session is available at viewing time and we can keep the data in *R*, or dynamically exchange it with *JavaScript* at viewing time.)

Example 7-1 Creating HTML Tables Using Election Results Stored in JSON

Consider the case where we produce an *HTML* document that allows the viewer to look through the county-level election results for different states in the US presidential election. The *HTML* document provides a pulldown menu which the reader uses to select a state name. The *HTML* table for that state is then displayed beside the menu. (Example 16-5 (page 559) contains a slightly more complex example of this where we use an *SVG* map instead of the pulldown menu to allow the viewer to select the state of interest.)

In this case, the county-level results for each state are in *R* and we create in *R* the *HTML* table to be displayed for each state. The code to create the *HTML* tables would be something like:

```
htmlTables = mapply(makeCountyHTMLTable, stateResult, stateNames)
```

where makeCountyHTMLTable() formats the values in `stateResult` into an *HTML* table. The result is an *R* character vector with an element for each state. We do not use these *HTML* tables directly in *R*, but instead insert them as *JavaScript* objects in the *HTML* document so that the *JavaScript* code can retrieve and display them in the *HTML* document on demand. It makes sense for the *JavaScript* code to use the state name or abbreviation (e.g., CA) to index the collection of tables. These data are best represented as a simple *JavaScript* associative array of the form

```
var stateTables = {
  "AL" : "<table>....</table>",
  "AK" : "<table>...</table>", ...,
  "WY" : "<table>...</table>"
 }
```

To create this object, we first add the names to the *R* character vector

```
names(htmlTables) = state.abb
```

Then we can readily generate this *JavaScript* table from within *R* by serializing the named character vector with toJSON(). We can then insert this *JSON* content into our *HTML* document either manually or programmatically. We prefer the latter:

```
newXMLNode("script",
           c("var stateTables = ", toJSON(htmlTables)),
           parent = xmlRoot(htmlDoc)[["head"]])
```

Rather than writing the code explicitly, we can use the addECMAScripts() function in the SVGAnnotation package [20] (see Section 16.8). The addECMAScripts() function provides a flexible and robust way to both add code and define *JavaScript* variables serialized from *R*. In this case, the call is

```
addECMAScripts(htmlDoc, "jshandlers.js",
               .jsvars = list(stateTables = htmlTables))
```

This function call adds the contents of the *JavaScript* code file `jshandlers.js` to the *HTML*. It also implicitly calls toJSON() to serialize `htmlTables` and then add it to the *HTML*. Generally, the *R* objects provided as a list via the *.jsvars* argument are serialized and added to the document as *JavaScript* variables, where the element names in the list, e.g., *stateTables*, are used as the names of the *JavaScript* variables.

The function asJSVars() in RJSONIO can be used to convert *R* objects to *JSON* in the same manner. However, this does not insert the content into a file as addECMAScripts() does.

Note that each item in the *HTML* menu uses the correct state abbreviation. To ensure this, it is good to generate this menu *HTML* description directly within *R*. Again, doing things programmatically reduces errors and makes the process reproducible and updateable.

7.4.3 Searching Text Documents with ElasticSearch and JSON

In this section, we look at using a local text search engine named *ElasticSearch* [5], which is based on ApacheLucene [3], an Open Source information retrieval software library. We will see how we can query the contents of text files and get the results as *JSON*. We will also see how to specify more complex queries using *JSON* as input. Finally, we show how to populate the text engine with *JSON* documents, and how to do this from *R* using regular *R* objects.

Text is an increasingly important form of data. Text data range from books, articles, and Web pages to emails, tweets, etc. We can treat these text "documents" as simple collections of strings and search them with regular expressions. However, we can also treat many forms of text as having both text content and additional fields such as the title, author's name, time-stamp, and so on. We can use regular expressions to search the entire text or break the documents into fields and search the fields separately. Matching via regular expressions works well, but there are better solutions for searching large collections of semi-structured text. A text search engine such as Lucene can be useful in similar ways to using a database to manage tabular data. It can centralize the management of the documents, and importantly can index them in different ways to make searching on different fields fast. It also searches in richer ways than just pattern matching, stemming the words and discarding stop words. Generally it analyzes and makes sense of terms, even in different languages.

ElasticSearch is a text search application built on top of Lucene. It uses *JSON* to represent documents and also to express rich queries of the documents. We communicate with *ElasticSearch* via *HTTP*, sending and receiving *JSON* content. Since it is based on Lucene, *ElasticSearch* can be highly distributed so it can scale to handle large bodies of documents and also searches. *ElasticSearch* therefore provides a *REST*ful interface that we can quickly deploy from within *R*.

We will illustrate how we can interact with *ElasticSearch* from *R* by working with articles from Google News [14] and also a collection of email messages. These illustrate two different ways to get data into the text search engine, both involving *JSON*.

Before we get started with our examples, you should install *ElasticSearch*. This is quite straightforward, assuming you already have a suitable version of *Java* on your machine. Download *ElasticSearch* from `http://www.elasticsearch.org/download/`. Extract the files from this archive (i.e., **zip** or **tar.gz** file). This will create a directory named something like `elasticsearch-0.18.7/`. Within this directory, you will find the application `elasticsearch` within the `bin/` directory. You can just run this, e.g.,

```
./elasticsearch-0.18.7/bin/elasticsearch -f
```

The flag `"-f"` causes this to run in the foreground and write messages on the console. This allows us to watch what the *ElasticSearch* server is doing. That's all there is to get the *ElasticSearch* engine up and running.

You should also install the *RSS* river plug-in with the shell command

```
./elasticsearch-0.18.7/bin/plugin -install dadoonet/rssriver/0.0.5
```

Example 7-2 Using ElasticSearch to Search Google News

We start by looking at articles from Google News' *RSS* feed. This is an aggregation of articles from many of the world's newspapers. Each article in the feed has a title, the text summarizing the article (description), a publication date (publishedDate), and a link to the full article. The *RSS* feed is updated frequently with new articles. *ElasticSearch* uses the term "river" to describe this form of streaming data and there are plug-ins for different types of rivers. There is an *RSS* river plug-in that reads the *XML* content from the *RSS* feed at regular intervals and converts each `<item>` in the *XML* document into a separate document within the text search engine. We can use this for any *RSS* feed. All we need to do is create a new river within our *ElasticSearch* server. To do this, we need to create an "index" within *ElasticSearch* that corresponds to a collection of documents, and specify the *RSS* feed and how often to retrieve updates. We do this by sending information as *JSON* documents to the *ElasticSearch* server via *HTTP* requests. We use the name "googlenews" for our index. To create this, we use a **PUT** request and send the simple empty *JSON* object, i.e., the string {}. We do this with

```
library(RCurl)
httpPUT("http://localhost:9200/googlenews", '{}')
```

Here we have specified the simple *JSON* content {} directly. If we want, we can also create it as

```
toJSON(emptyNamedList)
```

Note that this is different from

```
toJSON(list())
```

which generates *[]*, i.e., an array rather an associative array.

Next we specify the fields in each document and what types they have. We can omit this since *ElasticSearch* will do it automatically as it processes each document. However, there are occasions where we need to explicitly control how fields in the documents are processed. For instance, if the title was in French, we would want to specify this via the analyzer setting. These mappings of the fields indicate to *ElasticSearch* how to tokenize and make sense of the content. In the case of the *RSS* feed, the `<pubDate>` node in each `<item>` corresponds to a date. *ElasticSearch* may not recognize this and leave it as a string so we can specify the mapping directly. That is, we want to specify that each document (or page) has a pubDate property whose type is date. We can create the map information with the following *R* code

```
map = list(page = list(properties =
                           list(pubDate = c(type = "date"))))
```

We convert this to *JSON* and then send it to *ElasticSearch* with

```
uMap = "http://localhost:9200/googlenews/page/_mapping"
httpPUT(uMap, toJSON(map))
```

We can leave *ElasticSearch* to handle the other fields as it encounters them.

We have created the index and the optional mapping. Now we need to create the river so that *ElasticSearch* will start collecting records/documents from the *RSS* feed. For this, we again need

to use an *HTTP* **PUT** request, this time to the URL `http://localhost:9200/_river/googlenews/_meta`, i.e., to provide the metadata for this river. We need to specify the type of river, `type = "rss"`, so that *ElasticSearch* can select the appropriate river plug-in. We also need to describe the *RSS* feeds. Since there can be more than one, we need an array of them in the *JSON* document. Each feed can have a name, the *URL* from which to retrieve the updates, and a rate at which to check for new items. We can describe this information in *R* with

```
uRSS = 'http://news.google.com/news?output=rss'
meta = list(type = "rss",
            rss = list(feeds =
                       list(list(name = 'googlenews',
                                 url = uRSS,
                                 update_rate = 900))))
```

Note the extra *list* to create a list of feeds. The *update_rate* value is in milliseconds. The name within the feed is used when identifying elements from that feed. We have used `googlenews` as the name of the river, index, and feed. If we were to have a river that combined two *RSS* feeds, such as those for r-help and r-devel mailing lists from Gmane [16] (e.g., `http://dir.gmane.org/gmane.comp.lang.r.devel`), we might use "rmailinglists" for the name of the river, and "rhelp" and "rdevel" for the names of the individual feeds. This would allow us to determine from which mailing list each message/document came.

Now that we have the metadata described in *R*, we enable the river by sending the metadata to *ElasticSearch* with

```
uMeta = 'http://localhost:9200/_river/googlenews/_meta'
httpPUT(uMeta, toJSON(meta))
```

Immediately, *ElasticSearch* will process the feed(s) and create a new document for each *<item>* in the *RSS* feed. We can now perform text queries on these documents via the *ElasticSearch* index (not the _river). We can perform simple queries by passing a query string as part of the URL, e.g.,

```
uS = 'http://localhost:9200/googlenews/_search'
tmp = getForm(uS, q = 'Obama')
```

The result is *JSON* and so we can convert it with

```
ans = fromJSON(tmp)
```

This yields a list with four elements:

```
[1] "took"      "timed_out" "_shards"   "hits"
```

The first three elements provide information about the request, i.e., how long it took, whether it failed because it timed out, and which are the "shards" or distributed nodes. The `hits` element is the one with the actual results.

The `hits` element also has information indicating how many documents matched. Since we did not say how many we wanted returned, *ElasticSearch* just returns the first ten, but reports, in `ans$hits$total`, how many documents matched. We can specify how many to return by specifying a number for the size parameter in the call to getForm(), e.g.,

```
tmp = getForm(uS, q = 'Obama', size = 50)
```

The results are in the list `ans$hits$hits`. Each element of the list is an actual result that has fields

```
names(ans$hits$hits[[1]])
```

```
[1] "_index"  "_type"   "_id"     "_score"  "_source"
```

The document itself is in the `_source` element and has the content from the *RSS* `<item>`, e.g., `title`, `description`, `link`, `publishedDate`, and `source`:

```
names(ans$hits$hits[[1]]$"_source")
```

```
[1] "feedname"  "title"          "author"  "description"
[5] "link"      "publishedDate" "source"  "river"
```

The `river` and `feedname` elements identify the *ElasticSearch* provenance of the document.

The search above was very simple—a single string `"Obama"`. *ElasticSearch*, being a text search engine, allows us to perform much more powerful searches including faceted searches. *ElasticSearch* has a domain-specific language (DSL) for specifying searches. This is a rich topic and there are many sources of information online, e.g., http://www.elasticsearch.org/tutorials/2011/08/28/query-dsl-explained.html. We only look at a simple search and focus on how we specify this in *R* and send it to *ElasticSearch* as *JSON*.

Let's search for the string `"Europe"` or `"USA"` in either the title or description field of each document. We do this by sending the *JSON* document

```
{
 "query": {
           "query_string": {
                "fields": [ "title", "description" ],
                "query": "Europe USA"
              }
    }
}
```

We can create this query in *R* as

```
q = list(query = list(query_string =
                        list(fields = c("title", "description"),
                             query = "Europe USA")))
```

We can then perform this query with

```
tmp = getForm(uS, size = 10000,
              .opts = list(postfields = toJSON(q)))
```

and get the results in *R* via

```
results = fromJSON(tmp)$hits$hits
```

Example 7-3 Inserting Email Messages into ElasticSearch

In the example above, we used a river so that *ElasticSearch* would automatically create documents in our index from the *RSS* feed. We now explore how we can insert our own documents into *ElasticSearch*. We work with email messages from the Spam Assassin corpus [4]. Suppose we have already

read the individual messages into *R* and represented each as a *list* with two or three elements—`header`, `body` and `attachments`. The `header` element is a character vector with the name: value pairs from the message's metadata. The `body` is a character vector containing the lines from the body of the message. If the message has attachments, these are organized as a list in the `attachments` element. Each attachment is made up of a header and a body.

The list of messages is in the *R* variable `msgs`. We want to create a new index in *ElasticSearch* named, say, `"spamassassin"`. We use `"email"` as the type for each message. Any identifier can be used for each message. We can use some cryptographic value computed from the message's content, e.g., md5 or sha. Instead, we use the sequence of numbers 1, 2, ..., `length(msgs)`. Loading a message into *ElasticSearch* is quite straightforward. We augment the message with an element that indicates whether it is spam or not. We can compute this for all messages based on their file names, e.g., `"spam_2/00488.e88c2c87a3b72ab47b6420b61279242e"` or `"hard_ham/0244.4a88cbb01e5c7f05db4e12180bc122db"`. Next, we collapse the body to a single string. Then we convert the message from *R* to *JSON* and use an *HTTP* **PUT** request to send this to *ElasticSearch*. We can define a function to do this as follows:

```
insertMsg =
function(msg, id, spam, curl = getCurlHandle())
{
  msg$body = paste(msg$body, collapse = "\n")
  msg$spam = spam
  url = sprintf("http://localhost:9200/spamassassin/email/%d", id)
  try(httpPUT(url, toJSON(msg), curl = curl))
}
```

We call insertMsg() with the first message

```
insertMsg(msgs[[1]], 1L, FALSE)

[1] "{\"ok\":true,\"_index\":\"spamassassin\",\"
    _type\":\"email\",\"_id\":\"1\",\"_version\":1}"
```

We can convert the return value to *R* and check the value of the `"ok"` element. For now, we just assume that the request is successful.

We can insert all of the messages as follows:

```
curl = getCurlHandle()
invisible(mapply(insertMsg, msgs, seq(along = msgs),
                 grepl("spam", names(msgs)),
                 MoreArgs = list(curl = curl)))
```

We have created a curl handle and used it across all of the requests. This should speed up the requests a little as the connection does not need to be reestablished for each insertion. An alternative is to use *ElasticSearch*'s bulk loading facility.

With the messages now in *ElasticSearch*, we can search them using its search facilities. For example, let's find the messages that have the word `"VIAGRA"` in the subject field in the header. We can do this with

```
q = list(query = list(query_string =
                          list(fields = "header.subject",
                               query = "VIAGRA")))
uSpam = "http://localhost:9200/spamassassin/_search"
```

```
tmp = getForm(uSpam, size = 4000L,
              .opts = list(postfields = toJSON(q)))
```

Again, we convert the result to *R* with fromJSON() and explore the matching documents.

Our examples for *ElasticSearch* have explicitly shown both the *JSON* and the *HTTP* requests. To work more conveniently with *ElasticSearch* from *R*, we would define classes and functions to make these aspects transparent to the users and allow them to work at a much higher level of abstraction. See http://github.com/duncantl/RElasticSearch for some sample code.

We should also note that *ElasticSearch* is a *Java* application and can be embedded within *R* or another application. We can use that interface rather than the *REST* approach.

7.5 Comparing *XML* and *JSON*

Here we look at the specialized task of extracting specific elements from a nontrivial data set rather than reading the entire data set into *R*. We contrast *JSON* and *XML* formats and how we can manipulate them. We work with data from Kiva.org describing microloans. These data are available from http://build.kiva.org in both *XML* and *JSON* format. We look at a single file: the 100^{th} loan file. We parse both the *XML* and *JSON* versions of the file and then extract the country field for each loan within the file. We start with the *XML* file and repeat the process 10 times to get an adequate measure of time:

```
library(XML)
xml.tm = system.time(replicate(10, {
                  doc = xmlParse("XML/loans/100.xml")
                  country = xpathSApply(doc, "//location/country",
                                        xmlValue)
                  }))
```

We then restart *R* to avoid any memory issues and perform the equivalent computation for the *JSON* formatted file:

```
library(RJSONIO)
json.tm = system.time(replicate(10, {
                   j = fromJSON("JSON/loans/100.json")
                   sapply(j[[2]], function(x)
                                    x[["location"]][["country"]])
                  }))
```

The comparison of the time yields

```
json.tm/xml.tm
```

```
   user  system elapsed
   5.94    1.05    5.57
```

So the *XML* processing is over five times faster. Note that we can also make the *XPath* expression more precise and slightly more efficient (about 12 percent) by specifying the complete path `"/snapshot/loans/loan/location/country"`.

Next, we compare the process of simply parsing the two documents and not extracting any of its contents. We merely construct the in-memory native *DOM* for each, not converting the content from *C* to *R* objects. We do this with

```
xml_parse.tm = system.time(replicate(10,
                                     xmlParse("XML/loans/100.xml")))
json_parse.tm = system.time(replicate(10,
                      .Call("R_json_parse",
                              readLines("JSON/loans/100.json"))))
```

The ratio of times yields

```
json_parse.tm/xml_parse.tm
```

```
   user  system elapsed
  2.423   0.424   2.051
```

This indicates that the *JSON* parser is approximately two times slower than the *XML* parser just for processing the documents. Again, this is not entirely surprising. Firstly, the *JSON* parser is examining the content and identifying the types of the nodes: however, it is not actually converting the values to *R* objects. Secondly, the *XML* parser is probably more optimized at this stage in its development. Furthermore, there may be faster *JSON* parsers, although the one used by the RJSONIO package *claims* to be very fast.

The original tasks we are performing on the *XML* and *JSON* documents above involve both reading the document and querying the relevant fields (e.g., country field in the loan) within the hierarchies. If we want to extract several different variables from the document, then reading the document need only be done once. Here the *XML* and *JSON* steps are doing quite different things. The xmlParse() function reads the document as a tree. The fromJSON() function reads the hierarchy and also converts it to *R* objects. When we query specific variables, we rely on *XPath* for the *XML* document. For the *JSON* document, we are working with existing native *R* objects. *XPath* performs the search for the nodes quite quickly. The *R* subsetting, which is quite simple in our example, can be quite expensive and becomes slower as the depth of the subsetting within the hierarchy increases.

We now compare times for what fromJSON() does and the equivalent steps for *XML*. That is, we parse and convert the *XML* hierarchy into an *R* object. The xmlTreeParse() function does this so we can compare xmlTreeParse() to fromJSON()

```
xml_treeparse.tm = system.time(replicate(10,
                                xmlTreeParse("XML/loans/100.xml")))

json_treeparse.tm = system.time(replicate(10,
                                  fromJSON("JSON/loans/100.json")))

xml_treeparse.tm/json_treeparse.tm
```

```
   user  system elapsed
   9.63    7.44    9.60
```

Now, the *JSON* parser is faster than xmlTreeParse() by a factor of 9.

The outcome of theses comparisons tell us several things. If we want to extract a small subset of the hierarchical data, the *XML* approach is more efficient, but requires using *XPath*. If we want all of the data in *R* directly, fromJSON() is simple and efficient. There is room for improving the

performance of both the *XML* and *JSON* functions. There is more improvement to be made to how xmlTreeParse() creates the *R* objects. Perhaps the most important consideration is that processing either *XML* and *JSON* is pretty efficient and if the content is not large, or the number of documents to process is small, the time differential is not necessarily very significant.

7.6 Related Work

Representing *R* Objects as *JSON*
The `opencpu.encode` package [21] provides a *JSON* format for encoding *R* objects. The idea is to make exchanging data between *R* and other applications and languages relatively straightforward via *JSON*. The format is *R*-centric, but since the format is *JSON*, we can develop tools in other languages for serializing to and deserializing from this format.

Each *R* object is serialized as an associative array with three elements. These elements are named `"encoding.mode"`, `"value"`, and `"attributes"`, and they identify the type of the object, the *R* data objects, and the collection of *R* attributes such as names, dimension, etc. The `Inf`, `NaN` and `NA` values are serialized and deserialized appropriately.

***R* in the Web Browser**
We can read and write *JSON* data from within *R* to exchange data with another application, either in real time or for processing later. When we create content in *R* for technologies such as *HTML*, *SVG*, and *Flash*, we can use *JSON* to put *R* data into those documents which can then be used by the *JavaScript* code the documents contain. Unfortunately, when viewing those documents, the *R* session is not available and the computations and data cannot be updated from *JavaScript* using *R* code (without access to another *R* server). In order to make *R* available when the documents are being viewed in a Web browser, the RBrowserPlugin package [6] embeds *R* within a browser. This means that *JavaScript* can call *R* functions and update computations in real time. Furthermore, *R* code can call *JavaScript* functions and methods and so manipulate these documents dynamically. We can include live *R* graphics devices within *HTML* pages and update them dynamically. We can use *JSON* to exchange data between *JavaScript* and *R* in this framework; however, we also do this more directly with *C* code that connects to the two languages. This requires a great deal more specialized code than just using *JSON*, but has benefits (e.g., references rather than copies of objects).

7.7 Possible Enhancements and Extensions

Interface to the *JSON* Tree
It may be useful to work with the hierarchical structure corresponding to *JSON* content. This would allow us to parse the *JSON* content but not convert it to *R* values as we go, and instead return the hierarchical tree of nodes. Then we can traverse these nodes in the tree and extract the relevant data. This involves interfacing to the *JSONNODE C* data structure in `libjson` [28] (`http://sourceforge.net/projects/libjson`). We want methods to query its type, name, value, children, parent, and siblings. We like to programmatically generate the code to do this sort of thing using, e.g., the RGCCTranslationUnit package [23] or facilities in the `rdyncall` package [2].

Connections
The fromJSON() function currently collects the entire *JSON* content into a string and passes that to the *C*-level parser. It would be useful to be able to pass an arbitrary connection object from *R*. This would allow the *JSON* parser to pull data as it is needed. Again, this is similar to facilities in the *XML* event parser xmlEventParse(). The `json_new_stream` and `json_stream_push` are the primary routines to investigate for this. One approach is to couple this with the event-driven parser style and have the function that processes each node pull from the *R* connection and push it to the *JSONSTREAM* object. Unfortunately, since the *C*-level interface to connections in *R* is not "public," one will need to retrieve new content for the parser by calling an *R* function. This would be more compelling if we could do this entirely in *C* code.

Error Classes
It would be valuable to have the fromJSON() function raise different classes of errors and warnings (generally conditions). This would allow the caller to provide structured handlers for different types of errors.

Event-Driven Parser
One can write a different *R* interface to the *C*-level parser that allows the *R* programmer to provide an *R* function (or *C* routine) that is called each time the *JSON* parser encounters a value or the start or end of a container. This would resemble the SAX parser for *XML* and facilitate processing very large *JSON* documents without holding the entire document in memory.

Package for *ElasticSearch*
The code at `http://github.com/duncantl/RElasticSearch` is a prototype for some functionality to interface *R* with *ElasticSearch*. There is a great deal of scope to extend and enhance it and develop a complete interface.

BJSON
There are variations of *JSON* that allow binary content. MongoDB [1] is an application that uses such a format. It would be good to explore how to add this feature to RJSONIO or the `rmongodb` package [19].

7.8 Summary of Functions to Read and Write *JSON* in *R*

The RJSONIO package has two main functions, one to read *JSON* into *R* and another to create *JSON* content from *R* structures. Additionally, a third function determines if *JSON* content is valid, without actually converting it to *R* objects. These three functions are described below.

fromJSON() Parse *JSON* content into an *R* object. When *simplify* is `TRUE`, the elements of each array are collapsed from a *list* to a homogeneous *vector* when possible. In addition, *simplify* takes one or more of the values `StrictLogical`, `StrictNumeric` and `StrictCharacter` to specify that containers of the corresponding type are to be collapsed into vectors. These values can be combined as a vector or by adding them together, and `Strict`, the sum of these three, is the default value for *simplify*. The parameter *simplifyWithNames* specifies how associative (or named) arrays are to be simplified from a *list* to a *vector*. We can also specify how to map the *JSONnull* value to *R* with the *nullValue* parameter. Also, *stringFun* provides a way to specify a function that is used to convert strings, e.g., to convert dates that are often represented as strings in *JSON*.

toJSON() Convert an *R* object to *JSON*, recursively. Primitive *R* types (vectors and lists) are mapped to the corresponding *JSON* forms, with vectors of length 1 mapped to *JSON* scalars by default. Named vectors are mapped to dictionaries or associative arrays in *JSON*. We can define methods for toJSON() to control how different *R* classes are represented as *JSON*. Matrices are serialized as a *JSON* array of *JSON* arrays (i.e., a vector for each row), and similarly data frames are serialized as an associative array with elements corresponding to the columns.

isValidJSON() Check the *JSON* content and return `TRUE` if the content is valid, and `FALSE` otherwise. This parses the *JSON* document, but avoids converting the contents to *R* objects.

7.9 Further Reading

Some online resources that are useful for describing the *JSON* format include [17] and [27]. In addition, books on *JavaScript* typically have one or more chapters dedicated to *JSON*. See for example, Chapter 7 in [11] and Appendix E in [8]

References

[1] 10gen, Inc. The MongoDB NoSQL database. http://www.mongodb.org, 2011.
[2] Daniel Adler. rdyncall: Improved foreign function interface (FFI) and dynamic bindings to *C* libraries (e.g., *OpenGL*). http://cran.r-project.org/package=rdyncall, 2012. *R* package version 0.7.5.
[3] Apache Software Foundation. Apache Lucene: Open-source search software. http://lucene.apache.org, 2011.
[4] Apache Software Foundation. Spam Assassin: A spam filter that can be used on a wide variety of email systems. http://spamassassin.apache.org, 2011.
[5] Shay Banon. Elasticsearch: An open source, distributed, *REST*ful search engine. http://www.elasticsearch.org, 2011.
[6] Gabriel Becker and Duncan Temple Lang. RBrowserPlugin: *R* in the Web browser. https://github.com/gmbecker/RFirefox, 2012. *R* package version 0.1-5.
[7] Alex Couture-Beil. rjson: Converts *R* object into *JSON* and vice-versa. http://cran.r-project.org/web/packages/rjson/, 2011. *R* package version 0.2.6.
[8] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Sebastopol, CA, 2008.
[9] Dirk Eddelbuettel and Romain Francois. Rcpp: Seamless *R* and *C++* integration. http://cran.r-project.org/package=Rcpp, 2011. *R* package version 0.9.15.
[10] Bryan English. JPath: A *JavaScript* class which provides an *XPath*-like querying ability to *JSON* objects. http://bluelinecity.com/software/jpath, 2011.
[11] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.
[12] Markus Gesmann and Diego de Castillo. googleVis: Interface between *R* and the Google Visualisation API. http://cran.r-project.org/package=googleVis, 2011. *R* package version 0.2.13.
[13] Stefan Goessner. *JSON* Path – *XPath* for *JSON*. http://goessner.net/articles/JsonPath/, 2011.
[14] Google, Inc. Google News: A news aggregator service. http://news.google.com, 2011.

[15] Google, Inc. Google Visualization API reference. https://developers.google.com/chart/interactive/docs/reference, 2012.
[16] Lars Magne Ingebrigtsen. Gmane: A public mailing list archive. http://gmane.org, 2011.
[17] *JSON* Advocate Group. Introducing *JSON*: A lightweight data-interchange format. http://www.json.org/, 2006.
[18] Kiva Organization. Kiva: Loans that change lives. http://www.kiva.org/, 2011.
[19] Gerald Lindsly. rmongodb: *R*-MongoDB driver. http://cran.r-project.org/package=rmongodb, 2011. *R* package version 1.0.3.
[20] Deborah Nolan and Duncan Temple Lang. SVGAnnotation: Tools for post-processing *SVG* plots created in *R*. http://www.omegahat.org/SVGAnnotation, 2011. *R* package version 0.9.
[21] Jeroen Ooms. opencpu.encode: Encodes *R* objects to a standardized *JSON* format. http://cran.r-project.org/web/packages/opencpu.encode/, 2012. *R* package version 0.22.
[22] *R* Core Team. *Writing R Extensions*. Vienna, Austria, 2012. http://cran.r-project.org/doc/manuals/r-release/R-exts.html.
[23] Duncan Temple Lang. RGCCTranslationUnit: *R* interface to GCC source code information. http://www.omegahat.org/RGCCTranslationUnit, 2009. *R* package version 0.4-0.
[24] Duncan Temple Lang. RJSONIO: Serialize *R* objects to *JSON* (*JavaScript* Object Notation). http://www.omegahat.org/RJSONIO, 2011. *R* package version 0.95.
[25] The *JSON* Schema Community. *JSON* schema: A *JSON*-based format for describing *JSON* data. http://json-schema.org/, 2011.
[26] The *New York Times* Company. The Times Developer Network: An API clearinghouse and community. http://developer.nytimes.com/docs/campaign_finance_api/campaign_finance_api_examples, 2012.
[27] W3Schools, Inc. *JSON* tutorial. http://www.w3schools.com/json/default.asp, 2012.
[28] Jonathan Wallace. The libjson project: A *JSON* reader and writer. http://sourceforge.net/projects/libjson/, 2012.

Part II
Web Technologies
Getting Data from the Web

Overview

Having discussed working with *XML* and *JSON*, we now change our focus to accessing data over the Web. Web services and APIs are becoming an exciting source of interesting data. In this part of the book, we aim to cover many aspects of programmatically accessing data from Web services and less structured *HTML* forms. We also see how these same concepts are used to communicate more "locally" with different applications such as *NoSQL* databases, text search engines, and graph visualization software.

One of two themes of this part of the book is how to make general requests over the Web (or locally to other applications). For this, we need richer tools than *R*'s download.file() function. We need to be able to use different and secure protocols (e.g., *FTP* and *HTTPS*), control how we send data in both the body and the header of the request, specify how we send authorization and authentication information, and also be able to make multiple requests to the same server as part of an on-going dialog (e.g., with cookies and authorization tokens). We discuss this general topic extensively in Chapter 8, where we present the functionality provided by the RCurl package.

The second theme of this part of the book is about how we interact with the general concept of Web services. We show that we can access Web and local services, APIs, and *HTML* forms as if they were regular functions in *R*, but implemented in a remote language or machine. We can pass *R* objects as arguments or inputs to these "functions" and then get a result back as an *R* object that we can use in subsequent computations and analysis. We can explicitly create *R* functions to mirror these remote functions and manage marshalling the requests and responses. In many cases, we can programmatically generate these *R* functions We look at various different technologies that make all of this possible.

In Chapter 9, we address the basic mechanism of *HTML* forms. This builds on our ability to extract data from *HTML* documents, but illustrates how we can do this by mimicking a Web browser in *R*, which is the equivalent of filling in form elements and submitting them to a server. Rather than getting back an *HTML* page to view, we can extract the data directly into *R* for use in other computations. An important aspect of working with *HTML* forms is that we can programmatically create local *R* functions that interface with the form's remote functionality and we will see this arise with some of the other technologies.

HTML forms are not commonly used to implement Web services. This is because *HTML* forms typically return *HTML* pages that are to be viewed by a human within a Web browser. As a result, the data we want in the page are often enclosed within all sorts of unnecessary markup and obscured by formatting information, advertisements, and so on. Web services are very different. They are designed for computers and programs to make requests and directly consume the results. The data returned from a request is much more structured and easy to process directly within languages such as *R*. This

makes Web services a much richer approach to programmatically accessing data over the Web and, fortunately, this is becoming increasingly common.

Web services are increasingly being implemented using the *REST* architecture, which is the topic of Chapter 10. At its very simplest, *REST* leverages URLs as identifiers of resources which we can also think of as functions operating on data. We can query the current state of the resource, but our function calls may also update or modify the resource's state. We can send inputs in the request in much the same way that we submit an *HTML* form. The result, however, is rarely *HTML*. Instead, it is often text, an *XML* or *JSON* document, or some other data representation.

Two other approaches to Web services that are still in use, but are being dominated by *REST*, are XML-RPC and *SOAP*. The *REST* architecture is a quite powerful way to think about delivering Web services. It affords a great deal of flexibility in how consumers interface with the functions, e.g., how inputs and outputs are formatted. In contrast, XML-RPC defines precisely how each request and result is represented using an *XML* format. This is quite general but uses a small set of simple primitive data types and a specific format for each message. We describe XML-RPC and the XMLRPC package in Chapter 11.

In Chapter 12, we describe the *SOAP* approach to Web services and how to work with such services in *R*. *SOAP*, in spite of the "S" standing for "simple", is a Web service framework that is much more flexible and extensible than XML-RPC. While *SOAP* describes the basic structure of requests and responses, it allows us to use the full richness of *XML* to describe the data being passed as inputs and outputs of the remote function calls. This flexibility results in each Web service being different, much the same as *REST*. Fortunately, *SOAP* is also highly structured and self-describing. It promotes the notion of programmatically generated interfaces to a Web service via processing of a Web Service Description Language (*WSDL*) document. This description details the service's available methods and the data structures for their inputs and outputs. We illustrate the functionality in the SSOAP package to be able to programmatically generate *R* classes and functions to interface to arbitrary *SOAP* Web services. We also discuss the essentials of how to customize the programmatic code generation and the .SOAP() function that lies at the heart of the SSOAP package.

Many Web services are public and provide anonymous access to their functions and data. Increasingly, Web services want to monitor who is accessing the data by using a registration token in each request, but still providing access to public data. However, there is an entirely different set of Web services that provide access to private data. For these, the typical scenario we outline is that the owner of the private data is attempting to access them from *R*. While a login and password would suffice here, we end up using a more general authentication and authorization framework named *OAuth*. This separates the owner of the data, the host storing the data, and the application accessing the data. We use this "three-legged" mechanism to authenticate both the application and the owner and then retrieve an access token that we use in all requests that access the private data. Chapter 13 discusses two versions of this mechanism and is important for accessing nonpublic data common on many e–commerce and social networking sites on the Web and in the cloud.

How is this part of the book related to the first part? Clearly when we access data via *HTML* forms, we are reading *HTML* documents as we discussed in the chapters of Part I. Similarly, *SOAP* and XML-RPC use *XML* as the format for both requests and responses in Web services. In practice, *REST* typically returns results from requests as either *JSON* or *XML* content. While many *REST* services only require simple inputs such as numbers and strings, when more complex inputs are needed (e.g., documents, matrices, data sets), we typically send these as *JSON* or *XML* documents. Not only does this involve reading *XML* and *JSON* content, but also the ability to construct *XML* and *JSON* from within *R* as discussed in Chapters 6 and 7.

Chapter 8
HTTP Requests

Abstract In this chapter, we focus on general, all-purpose infrastructure we can use in *R* for accessing networks and the Web. The RCurl package provides both high-level and intermediate-level functionality within *R* that allow us to make rich and flexible requests to a large variety of different servers, including Web servers and applications that speak different protocols. RCurl generalizes the functionality built into *R* for downloading documents, etc., by a) supporting more protocols, b) allowing us to control many more aspects of the requests. We use RCurl as the foundation in *R* for scraping data from static Web pages, submitting *HTML* forms, interacting with *REST*ful APIs (Application Programming Interfaces) on both Web servers and local applications, invoking methods provided by *SOAP* servers, and using XML-RPC for remote procedure calls. RCurl provides support for many protocols used on the Internet (e.g., *HTTP*, *HTTPS*, *FTP*) and a large collection of options for controlling the requests (e.g., cookies, login and passwords, content type, header fields). In this chapter, we will focus primarily on *HTTP* and illustrate how to use RCurl generally by developing interfaces to different Web APIs.

8.1 Introduction

There are so many sources of interesting data available to us on the Internet these days. We can download datasets directly from repositories such as the University of California, Irvine's Machine Learning Repository [3]. We can also get data from static Web pages, or by filling in forms on *HTML* pages and specifying subsets of the data that we want. Many companies and institutions are making data available via "Web services." These allow us to get data more directly than via *HTML* pages. Web services are commonly provided as *REST*ful (Chapter 10) or SOAP (Chapter 12) APIs to which we make straightforward Web requests as if they were local functions in *R*. New APIs for social networking and also for many scientific and social portals are becoming available on a monthly basis. Importantly for all of these sources, we can return at a different time and get updated data, so our analysis is dynamic. We can access shared data and even share our own. This leads to qualitatively different kinds of research that make use of dynamically updated, shared data and results, and results that are inputs, or data, for other analyses. We are in a very exciting and transformative "era of networked science" [7], marked by open data and often called e-science. The potential of the Semantic Web and metadata about the data makes this even more exciting.

The key component underlying all of these opportunities is the Internet and our ability to use it to find and retrieve data. Specifically, our ability to make requests over the Internet makes all of these

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_8,

opportunities to access data and engage in e-science possible, and, of course, new technologies will continue to emerge based on the Internet. Increasingly, these technologies use the HyperText Transfer Protocol (*HTTP*) [2, 6] as the primary mechanism for making, or more specifically communicating, requests. We are all familiar with *HTTP* when using our Web browser to access pages, or fill in Web forms. Some of us have also used the browser to explore Web services. However, to realize the full potential of these technologies, we need to be able to use them programmatically within our work flow. We need powerful and flexible tools within *R* that allow us to do common things easily, and make new and complex tasks possible. The RCurl package [19] provides these high-level functions and lower-level facilities for working not only with *HTTP*, but also with many other Internet protocols such as FTP, SSH/SCP, LDAP, IMAP, POP3, SMTP, and "secure" *SSL*-based versions of each of these.

The default *R* environment already provides functionality for accessing data from the Web. The function download.file() uses either C-level code (from the `libxml2` library that also underlies the XML package [17]) or it calls an external command-line program that makes the Web requests on *R*'s behalf. Two of these command-line programs are `wget`, standing for "Web get", or `curl`, standing for "Client for *URL*s". These give us reasonably high-level access to Web requests and have served the *R* community well. The built-in *C*-level facilities allow us to specify only a few details of the Web requests, and are limited to only one type of *HTTP* request (**GET** methods). This is perfectly adequate for many requests, but is a significant limitation when dealing with Web services or communicating with other applications. Furthermore, we cannot use secure *HTTP*, i.e., *HTTPS*, with this approach.

Using external programs such as `wget` or `curl` works well. These are very flexible tools which give us a great deal of control over Web requests, including any of the *HTTP* methods. They also support *HTTPS* and many other protocols. The only drawbacks to using these from within *R* are that they may not be available on, for example, a Windows machine; we have to know the syntax of the command line flags, and potentially for each of the possible tools; they require that we compose the Web request as a string to be passed to a shell; they write the results to a file that *R* then has to read, involving a lot of unnecessary interaction with the disk; and they separate or discard the information about the content, e.g., the character encoding or type of content.

An alternative to download.file() for making Web requests in *R* is to use connections. The function url() allows us to make *HTTP* requests, but this is again based on the C-level code (optionally) used by download.file(). Accordingly, it is still limited. The socketConnection() function give us a very low-level way to make requests, but unfortunately, we are responsible for all of the details of the request, including the basic protocol to use and its vocabulary.

Each of these issues above is not a serious problem. However, the RCurl package avoids all of them and provides us with a cleaner programming model. We can control many aspects of each Web request, maintain information (or state) across different requests, and work with data in-memory rather than using temporary files and unnecessary reading and writing of data. We only need to learn one programming model and collection of options. Furthermore, RCurl uses `libcurl` [12], the code used in the `curl` command line program so everything we can do with `curl`, we can do with RCurl and more. We can use *R* functions to customize how we read the response from a Web request, or to upload data as it is needed. We can even make multiple requests simultaneously and have them be processed in parallel, potentially making many requests faster. By utilizing a third-party library such as `libcurl`, we transparently get any improvements its developers and widespread community add to it.

In this chapter, we will describe the essentials of *HTTP*, the HyperText Transfer Protocol that forms the basis for the Web. We interleave these details with descriptions of the high-level *R* functions we use to make common types of Web requests. We discuss how we customize these requests, and also illustrate how we might process the results. We aim to provide a strong understanding in this chapter for those who want to access data quickly, and also for those who want to go further and develop

interfaces to different Web services and applications. For the latter, it can be useful to explore the numerous packages that use RCurl. One of us (DTL), has written several such packages, and these provide good illustrations of the "best practices." Both groups of readers will also benefit from reading the subsequent chapters in this book, which use RCurl in different ways to access data.

Basically, once you have a strong working proficiency with RCurl, you can do very powerful things with Web requests. This is a significant trend for the future, so there is a good incentive to master the topic.

8.2 Overview of *HTTP*

The HyperText Transfer Protocol (*HTTP*) is mostly quite simple. It may seem more complex because there are lots of details for different situations that most people do not encounter or need to know. However, an *HTTP* request consists of:

1. a method or operation such as **GET**, **POST**, **PUT**, **DELETE**, or **HEAD**;
2. a *URL* to which to send the request;
3. information to include in the header part of the request that provides auxiliary information about the request;
4. an optional body for the request which contains the data characterizing the request. This might be included in the *URL* itself, or in the body, e.g., for a **POST** or **PUT** method request.

The header starts with a line containing, in order, the method name, the path to the local document of the *URL*, and the protocol and its version, e.g.,

```
GET /folder/docName HTTP/1.1
```

After this, we can add different `name: value` pairs to the request header. These allow us to provide additional information to the Web server about our request such as the identity of our application, a login and password for the caller, what type of data we want back, and so on.

The server processes the request and returns its response. This again consists of two parts—a header and a body (optionally). The header starts with a line that tells us the status of the request. The remaining lines are again lines of `name: value` pairs, telling us about the content of the response, e.g., the character encoding, and the type of content it is (e.g., an image, an *HTML* document, etc.). The body of the response is typically what we want. This can be any sequence of bytes.

HTTP is very flexible and allows us to add our own vocabulary in the *HTTP* headers for different applications, and even use request types/methods that only some servers will understand. This makes *HTTP* open-ended and complex. However, most of what we need is quite simple. We look at the different *HTTP* methods and the ways to add data to the header and the body.

8.2.1 The Simple GET Method

The most common *HTTP* operation is a mechanism to request a document from a Web server. This corresponds to the **GET** method. In *R*, we use the function getURLContent() in RCurl for this. At its simplest, you provide the full name of the document on a Web server, such as `http://www.omegahat.org/RCurl/index.html`, with

```
u = "http://www.omegahat.org/RCurl/index.html"
txt = getURLContent(u)
```

This *URL* identifies

- the protocol for the request (*HTTP*), i.e., the language the client and server will use during the request
- the name of the Web server or host: `www.omegahat.org`
- and the fully qualified path name of the specific document (RCurl/index.html) relative to the the server's top-level directory/folder.

The client software (e.g., the browser or *R*) uses the information in the *URL* to communicate with the Web server. It establishes a connection to the server via a socket, typically connecting to the Web server machine's port 80, although a different port can be specified in the *URL*. Having established the basic communication channel, the client makes the request by sending (at least) the three lines shown below. Each line in the header terminates with a control-linefeed combination i.e., the characters `\r` and `\n`. The third, and final, line is blank, and that signifies the end of the header information for the *HTTP* request.

```
1 GET /RCurl/index.html HTTP/1.1
2 Host: www.omegahat.org
3
```

These lines are quite easy to understand. The first word is `GET` which identifies the nature of the request, or the *HTTP* method. This means that the client wants to retrieve a document from the Web server.

The next word in the request is the full path of the document being requested — `/RCurl/index.html`. The final term on the first line is *HTTP*/1.1, which identifies the dialect of the protocol the client software is speaking. This tells the Web server that the client is using *HTTP* and, in particular, version 1.1. There are two options for the *HTTP* version, 1.0 and 1.1. As you might guess, 1.1 is more recent, richer and more flexible.

The second line (`Host: www.omegahat.org`) seems redundant as this is the name of the host to which the client connected. However, it is necessary as it helps a Web server that acts as multiple different hosts, or virtual sites, on the same machine. The second line tells the server application the identity of the appropriate virtual host so that it processes the request relative to that host's collection of files rather than any of the other virtual host's root directory. Our call to getURLContent() hides all of the details of creating the header from us. We just pass it the *URL*. It creates the connection to the Web server and the header and sends it. The function then waits for the server to respond.

8.2.1.1 Adding Fields to the *HTTP* Header

As we mentioned earlier, each line in the header (after the first) is of the form `name: value`. The line `Host: www.omegahat.org` is an example of this format. There may be others such as a line identifying the application making the request. This can be of the form `User-Agent: R (2.15.0)`, or `User-Agent: twitteR (0.99.19), R (2.15.0)` Of course, this detail is application-specific and so we would have to add it in our call to getURLContent(). We can do this with something like

```
u = "http://www.omegahat.org/RCurl/index.html"
txt = getURLContent(u, useragent = R.version$version.string)
```

Here, *useragent* is treated specially and identifies a `libcurl` option and causes getURLContent() and its helper functions to use the string as the value for the field named *User-Agent* in the header.

Adding a *User-Agent* field is very common, so there is a special option for specifying it, i.e., *useragent* in getURLContent(). There are other common fields that have their own argument. For example, we may want to provide a user login and password if the *URL* is password-protected. We can do this with the *userpwd* option. Alternatively, we may want to tell the server that we are following a link and so provide that as the value of a "Referer" field, using the *referer* option. However, not all header entries have a built-in option. Indeed, we can specify new ones that a particular server expects or understands. We specify these via the general *httpheader* option to getURLContent() and pass it a named vector of values, e.g.,

```
u = "http://www.omegahat.org/RCurl/index.html"
txt = getURLContent(u, useragent = R.version$version.string,
                    httpheader = c(Authorization = "xxx",
                                   From = "login@mail.com",
                                   'X-Do-Not-Track' = "1"))
```

This results in the header

```
1 GET /RCurl/index.html HTTP/1.1
2 User-Agent: R version 2.15.0 (2012-03-30)
3 Host: www.omegahat.org
4 Accept: */*
5 Authorization: xxx
6 From: login@mail.com
7 X-Do-Not-Track: 1
8
```

We did not specify the *Host* or *Accept* entries, but `libcurl` added them for us. The header fields we added were included. Of course, the Web server has to process and understand these fields for them to have any effect. There are numerous commonly recognized fields, e.g., see http://en.wikipedia.org/wiki/List_of_HTTP_header_fields. In other cases, certain APIs and servers with which we are communicating will identify special headers they recognize.

8.2.1.2 Understanding the Server's Response

Given the complete header in the example above, the server then processes the details and returns its result as a collection of bytes. The response, like the request, starts with a header and is followed by the body of the response. In this example, the header looks like the following:

```
1 HTTP/1.1 200 OK
2 Date: Fri, 01 Jun 2012 22:56:46 GMT
3 Server: Apache/2.2.14 (Ubuntu)
4 Last-Modified: Wed, 01 Feb 2012 04:08:30 GMT
5 ETag: "3262089-10bf-4b7df3b75ab80"
6 Accept-Ranges: bytes
7 Content-Length: 4287
8 Vary: Accept-Encoding
9 Content-Type: text/html
10
```

Again, the header is terminated by the presence of a blank line (numbered 10). It is also made up of a first line describing the dialect of *HTTP* and its status, `200 OK`. This status consists of a number and a string. These numbers are grouped by 100s, e.g., 100, 200, 300, ..., and each group has a different meaning. See Table 8.1 for an explanation. The string (e.g., `OK`) is less useful than the number, but does gives us some idea of the status. Often, for errors, the body of the response will contain more information which we can then show to the user to help determine the potential cause of the problem.

Table 8.1: *HTTP* Status Codes and Explanations

Status	Category	Description
100	Informational Continue	A 100 status code typically indicates that the communication is continuing and that more input is expected either from the client or the server. It is also used to indicate a "chunked" response where the server sends the results back in different sections. As users of `libcurl` and RCurl, we rarely have to concern ourselves with these "intermediate" status codes.
200	Success	This indicates that the request was successful. A 200 indicates general success. Other values in this group provide different details. For example, 201 indicates that the new resource was created, while 202 means that the request was accepted, but the processing of it has not yet been completed. A 202 status code might occur in asynchronous requests.
300	Redirection or Conditional Action	This often indicates that the requested *URL* is actually located somewhere else. The server typically responds by pointing the caller to that new location. We can follow those redirections automatically with the curl option *followlocation* set to `TRUE`. If we have asked for a document conditional on it having been updated since a particular date and there have been no updates, we receive a status of 304.
400	Client Error	An error occurred. A common version of this is the 404 status code which indicates that the document was not found for the given *URL*.
500	Internal Server Error or Broken Request	An internal error occurred in the server or Web site's mechanics. When we generate Web service requests and send them from *R* or any client other than a Web browser, this status code may indicate an error in the call, not the Web server.

This table lists the general categories of HTTP status/response codes. Each category has subcodes/values with more specific meanings and information. See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html` *for more details.*

The `name: value` pairs in the header provide information from the server to the client describing details about the response. These contain information about the body of the response which is the content of the requested document and how to process it. From the *Content-type* field, the client knows the body is text containing *HTML*. The client also provides the number of bytes being sent (4287), and information about the Web server and what software it is running. There are many possible fields in the request and response headers and this gives *HTTP* its flexibility. The interested reader is referred to `http://www.w3.org/Protocols/rfc2616/rfc2616.html`. More important for us here than the details of the different parameters is the knowledge that *HTTP* supports a rich set of controls, and applications may need to provide and interpret these in different ways. In this respect, a flexible interface that obviates the need to know the details, but that still provides access to them is important for general use in many different contexts.

The getURLContent() function processes the header in the response and can then read the body. It knows (or guesses) whether the content is binary or text based on the *Content-Type*. Similarly, it knows the character encoding from that field also, e.g., `"text/html; charset=ISO-8859-1"` (informally known as latin1), in some responses. By default, getURLContent() just returns the body. If the content is considered binary (either because getURLContent() did not recognize the *Content-Type* as indicating text, or because the caller explicitly indicated that the content was binary via the

binary parameter), getURLContent() returns the body as a *raw* vector, i.e., the bytes. If the content is text, it is returned as a single string, with the character encoding already established.

The getURLContent() function also sets the `Content-Type` attribute on the result. This gives more information that the caller may need in order to interpret the content. For instance, is the text plain text, *XML*, *HTML*, or a comma-separated value document? Is the binary compressed, and if so with what mechanism? Or, is it an image and if so what format? Basically, the `Content-Type` tells us how to interpret the data in *R*.

8.2.1.2.1 Processing the Body in *R*

Let's see how we can process the body of the server's response in different circumstances. As an example, we will download the mail archives for the R-help mailing list for the month of May, 2012.

Example 8-1 Retrieving a Gzipped Mail Archive from a Secure Site

These mail archives are available as a gzipped document at `https://stat.ethz.ch/pipermail/r-help/2012-May.txt.gz`, which we have assigned to the `uRHelp` variable. We can retrieve this file with

```
gdata = getURLContent(uRHelp)
```

The `gdata` variable in *R* is a *raw* vector and has a `Content-Type` attribute with the value `application/x-gzip`. This tells us the nature and format of the content and how to process it. Unlike with a browser or download.file(), the returned document is not in a file outside of *R*, but in memory and assigned to the *R* variable `gdata`. We can uncompress the document directly with the gunzip() function in the Rcompression package [18] using

```
library(Rcompression)
txt = gunzip(gdata)
```

The result is a single character string which we can break into separate lines

```
lines = strsplit(txt, "\\n")[[1]]
```

and then process as a mail box. For instance, we can find the names of the senders with

```
from = grep("^From: ", lines, value = TRUE)
who = gsub("From: ([^ ]+) at ([^ ]+) .*", "\\1@\\2", from)
head(sort(table(who), decreasing = TRUE), 3)
```

```
michael.weylandt@gmail.com       ruipbarradas@sapo.pt
                       162                        123
    dwinsemius@comcast.net
                       122
```

As a second example, we will process data from the National Stock Exchange of India Limited, which makes available daily and monthly reports for stock prices of many different companies.

Example 8-2 Retrieving a CSV File from National Stock Exchange (NSE) India

NSE India provide a CSV file describing the volatility of each stock price for, e.g., June 7, 2012 at `http://www.nseindia.com/archives/nsccl/volt/CMVOLT_07062012.`

CSV, which we assign to the character string uIn. Unfortunately, if we try to import these data into *R* with read.csv(), this will fail

```
d = read.csv(url(uIn))

Error in open.connection(file, "rt") : cannot open...
In addition: Warning message:
In open.connection(file, "rt") :
  cannot open: HTTP status was '403 Forbidden'
```

The problem is that the nseindia.com Web site is quite picky about the requests it accepts. We *must* include both an *Accept* and a *User-Agent* field in the request's header. The *Accept* field should be something appropriate such as */* to accept any type of data in response to the request. If we omit either of these, the request fails. We can, however, separate importing the data into *R* into two steps: a) download the contents of the CSV file into *R*, and b) read the text as CSV content. We use getURL-Content() via

```
d = getURLContent(uIn, useragent = "R")
```

The variable d is now a string in *R* containing the entire CSV document. We can use read.csv() to process this in memory by using a textConnection() object, e.g.,

```
data = read.csv(textConnection(d))
```

In the actual problem, we want to download daily reports of stock prices. We can retrieve these via an *HTML* form on the page http://www.nseindia.com/products/content/equities/equities/archieve_eq.htm by interactively specifying the date. By looking at the pattern of the file names, we can see that we can fetch the data for a given date by constructing the appropriate *URL* with the day of the month, the month name, and the year inserted in different places. The *URL* for February 8, 2012 is http://www.nseindia.com/content/historical/EQUITIES/2012/FEB/cm08FEB2012bhav.csv.zip, which is assigned to the character string in u8Feb.

Even though the files are not very large, they are stored and returned as ZIP files containing a single CSV file. This means that even if the Web site would accept requests from read.csv() for regular CSV files, we would not be able to use the function for these zipped CSV files. But we can use getURLContent() and then process the resulting document:

```
z = getURLContent(u8Feb, useragent = "R")
```

The result in z is a *raw* vector. This is the zipped document. We can unzip it directly within memory using the zipArchive() function in the Rcompression package with

```
library(Rcompression)
txt = zipArchive(z)[[1]]
```

Finally, we can pass this to read.csv() as above

```
data = read.csv(textConnection(txt))
```

These examples illustrate that we sometimes need more control over the *HTTP* requests *R* makes with its built-in functions. We may have to use *HTTPS* which *R* does not support, or provide additional information in the request such as a login and password, or some form of authentication [e.g., a cookie or an *OAuth* token (see Chapter 13)], or an *Accept* field in the header of the request.

8.2.1.2.2 Manipulating the Header in *R*

Even though the return value from getURLContent() includes the *Content-Type* as an attribute on the body of the request, there are occasions when we want both the body and the header from the request, and not just the body. For instance, we may want to store the value of the *Last-Modified* field to implement a form of local caching, or get the ETag or any other header field. We can get the entire header in various ways, but the simplest way is to ask getURLContent() for it via the *header* parameter. If we specify a value of `TRUE` for this parameter, getURLContent() returns a list with two elements, the header and the body, using those names for the elements. The header is processed to break the header fields from `name: value` strings into a named character string, and also decompose the first line into the status (code) and status message, which are added to the header. This is done via a call to the function parseHTTPHeader().[1]

The getURLContent() function makes it simple to retrieve an existing document. As we have seen, we can add fields to the header. We will see later that we can control a great deal more about the **GET** request. However, next we look at providing additional information in the **GET** request, specifically in identifying the *URL*. *HTTP* can be used to request dynamic or conditional content. In the next section, we will examine *HTML* forms and how they allow us to use a Web browser or *R* to specify inputs to an *HTTP* query.

8.2.2 *GET* Requests with Input Parameters

We saw how the **GET** operation is used to retrieve a regular document identified by a *URL*. However, often we can, and need to, provide additional information in the request to specify more details about the content we want from that particular *URL*. This occurs commonly when submitting an *HTML* form or in a call to a method in a *REST* API. When submitting an *HTML* form through the browser, we select these input items from a menu, checkboxes, and/or radio buttons, and then submit the request via a button. At its simplest, the browser sends the *HTTP* query in exactly the same way as a regular request for a document; that is, it uses the **GET** action and specifies the name of the script associated with processing the form as the target document with the user-specific information from the form appended onto the file name. For example, instead of `"/RCurl/index.html"`, the request would include *name=value* pairs from the form in the URI (Uniform Resource Identifier) being requested. These *name=value* pairs are separated from each other by the & symbol, and separated from the file name by a `?`. To send a request to a script file named `/apps/myForm` with two variables named `first` and `last`, the browser would construct the query, e.g.,

```
GET /apps/myForm?first=Duncan&last=Temple+Lang
```

Note that the space in the value of the `last` field (Temple Lang) is "escaped." Spaces are converted to the character +, and non-alphanumeric characters are represented by their hexadecimal position in the character set. Additionally, the client should indicate to the server that this *HTTP* request is for a form by adding

```
Content-Encoding: application/x-www-form-urlencoded
```

to the request's header.

[1] If, for some reason, you want the unparsed header, use `header = I(TRUE)` in the call to getURLContent().

Example 8-3 Requesting Stock Prices via a Form on Yahoo

The Yahoo! finance site allows us to look at the history of stock prices for different companies. For example, the *URL* `http://finance.yahoo.com/q/hp?s=GOOG` shows a table of Google's stock price from August 19, 2004, to today's date. The Web page has a form that allows the reader to change the month, day, and year for the start and end dates. We will see in Chapter 9 how we can programmatically create an *R* function to emulate this form, but for now we focus on the lower-level mechanics of getting the data in *R*. When we click on the "Get Prices" button, the browser fetches the new page for those dates and we see a new *URL* in the browser's location field: `http://finance.yahoo.com/q/hp?s=GOOG&a=07&b=19&c=2004&d=05&e=3&f=2012&g=d`. The resource part of the *URL* is `http://finance.yahoo.com/q/hp`. Everything after the `?` is called the query string. It provides the input to the form to customize the request to get data back for Google's stock price for the specified period of time. We can think of these as arguments or inputs to the script that parameterize how it computes what data to return.

We can deal directly with the *HTML* page and get the data from it using something like readHTMLTable(). However, near the bottom of the updated page is a link to a CSV file. We can access that directly and read the contents into *R*. The *URL* is `http://ichart.finance.yahoo.com/table.csv?s=GOOG&a=07&b=19&c=2004&d=05&e=3&f=2012&g=d&ignore=.csv`. This is similar to the one for the *HTML* page but the base part is `http://ichart.finance.yahoo.com/table.csv`. The parameters for the form are the same. As with *R*, the parameters have names and here we have to specify values for each parameter using their name, e.g., `s = GOOG`, `c = 2004`. The names of the parameters are not very informative: `s` for stock and `a`, `b`, `c`, `d`, `e`, `f`, `g` for the dates! The names are not really important as the code in the *HTML* page hides all of these from the viewer. For programmatic access within *R*, however, we need to understand these names.

We can take the parameter names and values in *R* and create the *URL* as a single string and then use a regular **GET** operation, e.g., if `yahooU` holds the string `http://ichart.finance.yahoo.com/table.csv?s=GOOG&a=07&b=19&c=2004&d=05&e=3&f=2012&g=d` we have

```
txt = getURLContent(yahooU)
```

However, a better way to do this in *R* is to call getForm(), such as

```
txt = getForm("http://ichart.finance.yahoo.com/table.csv",
              s = "GOOG", a = 7, b = 19, c = 2004, d = 5, e =  3,
              f = 2012, g = "d", ignore = ".csv")
```

As with the earlier regular **GET** requests, `txt` has a `Content-Type` attribute and we see this is explicitly defined as a CSV document:

```
attr(txt, "Content-Type")
```

```
"text/csv"
```

We can then pass this text to read.csv() as a textConnection(), with

```
goog = read.csv(textConnection(txt))
head(goog)[ , 1:4]
```

```
        Date   Open   High    Low
1 2012-06-01 571.79 572.65 568.35
2 2012-05-31 588.72 590.00 579.00
```

```
3 2012-05-30 588.16 591.90 583.53
4 2012-05-29 595.81 599.13 588.32
5 2012-05-25 601.00 601.73 588.28
6 2012-05-24 609.16 611.92 598.87
```

Why is it better to use getForm() and specify the parameters as *R* arguments rather than just using the string `http://ichart.finance.yahoo.com/table.csv?s=GOOG&a=07&b=19&c=2004&d=05&e=3&f=2012&g=d`? There are several reasons.

- If we have the individual values of the parameters in *R*, it is easier to let getForm() create the string for the *URL* on our behalf. Calling paste() or sprintf() is more error-prone.
- It is easy to change a single parameter in the call rather than edit the string. For example, we can loop over different symbols such as

```
lapply(c("GOOG", "FB", "ORCL", "YHOO", "AAPL", "MSF"),
       function(sym)
          getForm("http://ichart.finance.yahoo.com/table.csv",
                  s = sym, a = 7, b = 19, c = 2004, d = 5,
                  e = 3, f = 2012, g = "d"))
```

  We have only changed the value passed for the *s* parameter.
- When we create the string form of the *URL* for a **GET** operation, we have to ensure that the parameters are "escaped" correctly. For example, when submitting the string `"R XML"` as a query string to Google, we have to write this as `"R+XML"` (or `"R%20XML"`). Similarly, an `&` in a string gets converted to `%26`, and the @ character is sent as `%40`. This is called percent encoding. RCurl provides the function curlPercentEncode() to perform the conversion for a string, but this is clearly more involved than simply leaving getForm() to do the work for us.
- As we will see in the next section, *HTML* forms often use **POST** rather than **GET** operations. This changes the format of the *URL* entirely and the parameters and their values are sent in the body of the request, not appended to the *URL*. Accordingly, we can change our request to use postForm() rather than getForm(), but leave the remainder of the *R* command unchanged, i.e.,

```
u = "http://ichart.finance.yahoo.com/table.csv"
txt = postForm(u, s = "GOOG", a = 7, b = 19, c = 2004, d = 5,
               e = 3, f = 2012, g = "d", ignore = ".csv")
```

  Using getForm() or postForm() yields a more robust piece of code. It keeps the different components of the call separate and clearly states the intent of the call, i.e., **GET** or **POST** for a form rather than **GET** a generic *URL*.

8.2.3 *POST'ing a Form*

Using the **GET** method to submit a form or pass arguments in a call to a *REST*ful method is quite simple to implement as the entire request is in the *URL* and the *HTTP* header. There is no body to deal with. Simplicity is good, but there are drawbacks. The most important problem relates to the length of the URL in the request. Suppose we want to use a remote method in some Web service to fit a model, or perform some numerical optimization. We may have to send it a data set. There are

issues about how we should represent it (e.g., CSV, *JSON*, *XML*, *R*'s binary data format (encoded in base-64), etc.), but regardless of this choice, we will send the resulting string as part of the *URL*. This can make that *URL* very long, and the Web server will probably reject it. A common limit in Web servers on the length of a *URL* is 2048 characters. This severely limits what we can send in a **GET** request. *HTTP*'s **POST** method overcomes this problem.

POST separates the *URL* and the additional information for the request by putting the latter into the body of the request. The body of a **POST** request can be anything, such as the contents of a file or an in-memory image, or the collection of parameter name and values for a form. We will look at forms first and return to arbitrary "uploads" later.

Many *HTML* forms use **POST** rather than **GET** requests to submit the form. Not only do these avoid the limits on the length of the *URL* query, but they also keep the query out of the browser's location field. This can be useful if the data should not be seen by others, e.g., a password. From the point of view of submitting a request to the Web server, the user interface controls are not important. Instead, an *HTML* form is simply a collection of name and value pairs. We need to submit this collection of pairs in the body of the *HTTP* request in the same way a browser would, and then the Web server or script can access the body and separate the parameter pairs. For this exchange to work, the client and server need to agree on how to represent the pairs in the body of the request. There are many different possible approaches, but two are in common use. These are termed `application/x-www-form-urlencoded` and `multipart/form-data`, and the former is the most common. For submitting form requests from *R*, we do not actually need to know how to format the body. Instead, we just need to tell the function postForm() which format to use. We do this via the *style* parameter and specify either `"post"` for `x-www-form-urlencoded` or `"httppost"` for `multipart/form-data`. Unfortunately, even though `x-www-form-urlencoded` is the most common format in use, the default value for the *style* parameter is `"httpost"`, and so the function uses `multipart/form-data` by default. Therefore, we often have to explicitly specify `style = "post"` in calls to postForm(). We look at a simple example.

Example 8-4 Posting a Form to Obtain Historical Consumer Price Index (CPI) Data

The Web page `http://www.rateinflation.com/consumer-price-index/usa-historical-cpi.php` provides a table giving the monthly CPI (consumer price index) for the last five years in an *HTML* table. It also provides an *HTML* form that allows us, the viewer, to specify a different period of time by selecting the starting and ending year, identified by the parameters named fromYear and toYear. Values for each of these parameters are given as years of the form 2010, i.e., four digits. There is also an additional "hidden" parameter, *_submit_check*, for which we must send a value of 1 or else the server will not return the requested data. We submit this request with `cpiUrl` the URL

```
txt = postForm(cpiUrl, fromYear = "1913", toYear = "2012",
               "_submit_check" = "1", style = "post")
```

The result is an *HTML* document as it is intended to be used within a Web browser. We can use readHTMLTable() to read the tables, and in this case the data we want are in the twelfth table. So we can extract the data from this table with

```
cpi = readHTMLTable(txt, which = 12, header = TRUE)
dim(cpi)
```

```
[1] 100  14
```

```
cpi[c(1:6, 100), c(1:5, 14)]
```

```
    Year     Jan     Feb     Mar     Apr  Annual
1   2012 226.665 227.663 229.392 230.085
2   2011 220.223 221.309 223.467 224.906 224.939
3   2010 216.687 216.741 217.631 218.009 218.055
4   2009 211.143 212.193 212.709  213.24 214.537
5   2008  211.08 211.693 213.528 214.823 215.303
6   2007 202.416 203.499 205.352 206.686 207.342
100 1913     9.8     9.8     9.8     9.8     9.9
```

8.2.3.1 Two POST Formats: `x-www-form-urlencoded` and `form-data`

While `libcurl` takes care of creating the body of the request from our collection of inputs, it is sometimes useful to know what the different formats entail. In this section, we briefly describe these two formats. The `x-www-form-urlencoded` format is basically the same as the format for appending the name–value pairs in the **GET** request. That is, we escape all of the names and the values and then combine them with the = character between each name and value, and separate the pairs with &. In our example, this would be

```
fromYear=1913&toYear=2012&_submit_check=1
```

and this would be sent as the body of the request.

For `multipart/form-data`, the body consists of a part for each parameter, separated by some unique string that does not occur in any of the parts. Within each part, there is a header that describes the nature of the part, e.g., its *Content-Disposition*, the name of the part, the *Content-Type*, etc. Following the header is the actual content of each part. If our example used `multipart/form-data`, the body would be sent with something like the following format:

```
------------------------------380d34374351
Content-Disposition: form-data; name="fromYear"

1913
------------------------------380d34374351
Content-Disposition: form-data; name="toYear"

2012
------------------------------380d34374351
Content-Disposition: form-data; name="_submit_check"

1
------------------------------380d34374351--
```

The string `"------------------------------380d34374351"` is used to separate the parts. `libcurl` generates this itself, ensuring that the string does not occur in any of the values being submitted. It adds this separator string to the header of the *HTTP* request as

```
Content-Type: multipart/form-data;
 boundary=------------------------------380d34374351
```

(we have reformatted it from one line to two for display purposes). This allows the receiver to identify the separating boundary between the parts and correctly decompose the body into what the sender originally had. Some readers may recognize this as the same format used for representing attachments in an email message.

8.2.3.2 Uploading the Contents of Files

We end this section by noting that we have not yet discussed how to upload files in a form. We will discuss this in more detail when describing the **PUT** method. However, it may occur to the alert reader that we can do it with what we have discussed so far. We can read the contents of the file into *R* using whatever function is appropriate, e.g., readLines() or readBin(). We can then collapse this content into a string and pass that as the value of the relevant form parameter in the call to postForm(). If the contents of the file are not text, we have to convert them to a string. To do this, we read them as a *raw* vector and then convert this to a base64 encoded string. For example, to send a PNG image as part of a form, we can use

```
filename = "ChromeNetworkTools.png"
dd = readBin(filename, raw(), file.info(filename)[1, "size"])
tt = base64(dd)
```

One feature that is missing in this approach is that the name of the file is not included in the information that is sent in the body of the **POST** request.

8.2.4 Specifying Request Options in getForm() and postForm()

Both getForm() and postForm() allow the caller to pass arbitrary arguments as parameters for the form. These are handled by the ... mechanism in *R*. This is different from getURLContent() which uses the ... parameter for curl options to `libcurl`. This means that we cannot specify curl options directly in calls to getForm() and postForm(). Instead, we pass them in a list via the *.opts* parameter. For example, if we want to set the *useragent* option and also turn on *verbose* display of the request being processed in a call to postForm(), we can do this as

```
txt = postForm(u, fromYear = "1913", toYear = "2012",
               "_submit_check" = "1", style = "post",
              .opts = list(verbose = TRUE, useragent = "R"))
```

Note also that while we can specify the parameters for the form via the ... parameter, we can also provide them as a single list via the *.params* argument. This can be useful if the parameters are already in a list from some earlier call, e.g., to lapply(). As a slightly contorted example, we might have a list for the parameters such as

```
args = list('_submit_check' = "1")
```

Then we can set the other two parameters with

```
args[c("fromYear", "toYear")] = range(values)
```

and submit the form with

```
txt = postForm(u, .params = args)
```

In this case, there is no gain. However, if there are many parameters that we can compute easily in *R* as a list using some vectorized operation, then this approach is convenient.

We have now seen the three high-level functions getURLContent(), getForm() and postForm(). The first is the primary way to make a simple **GET** request; the other two are for submitting "forms," and use the **GET** and **POST** methods. These are very common operations. Since forms are different from other types of **POST** operations, we have specialized functions to deal with them. However, *HTTP* provides other methods, and even for **GET** and general **POST** methods, we may want to use less-specialized functions. RCurl provides the functions httpGET(), httpPOST(), httpPUT(), httpHEAD(), httpDELETE(), and httpOPTIONS(), corresponding to each of the common *HTTP* methods. There are other possible *HTTP* methods, but these are much less commonly used and can be easily used by specifying the *HTTP* method name as the value for the curl option *customrequest*.

The function httpGET() is a very thin layer around getURLContent() and is only defined so that we have functions with a name of the form http*METHOD-NAME* for each method. In the remainder of this section, we examine these *HTTP* methods and the corresponding functions in *R*. Since httpGET() is a very simple call to getURLContent(), we will not discuss it.

8.2.5 The General *POST* Method for Data in the HTTP Request Body

In addition to using **POST** for sending form parameters, we can also use it to send arbitrary content as the body of the request. For example, we can upload the contents of a file such as a regular text file, or a binary image, or data in memory in the *R* session, either binary or text. We use this in both the XMLRPC [23] and SSOAP [21] packages to send *XML* documents that we create in memory to Web servers and applications. We do not need to know the details of how we create these documents to explore how we send its contents to the host with

```
httpPOST(targetURL, postfields = saveXML(doc),
         httpheader = list('Content-Type' = 'text/xml',
                           SOAPAction = "urn:soapinterop"))
```

If we look at the request being made (with *verbose* set to `TRUE`), we see

```
POST /interop.cgi HTTP/1.1
Host: services.soaplite.com
Accept: text/xml
Accept: multipart/*
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:soapinterop"
Content-Length: 579
```

The httpPOST() function used our value for *Content-Type* and also set the *Content-Length* field to be the number of characters in the string in *postfields*, i.e., the *XML* document as a string. The body of the request is simply the text representing the *XML* document.

We look at two ways to **POST** the contents of a file. One way is to read it into *R*, and the other is to have `libcurl` read directly from the file and bypass reading it into *R*.

At the end of Section 8.2.3, we saw how we can upload the contents of a binary file as a value of a form parameter. We can use a very similar technique to upload the contents of a file as the body on a request. As in that example, we again read the file contents into *R* with

```
fN = "RCurl/Screenshots/ChromeNetworkTools.png"
dd = readBin(fN, raw(), file.info(fN)[1, "size"])
```

We convert this to text using base-64 encoding via

```
tt = base64(dd)
```

To send this as the body of the request, we need to do two things: pass this to httpPOST() as the value of the *postfields* option, and also specify the type of data this represents using the *HTTP* header field *Content-Type*. We do this with

```
httpPOST(targetURL, postfields = tt,
         httpheader = list('Content-Type' = 'image/png'))
```

Again, httpPOST() sets the *Content-Length* field in the header to the number of elements in `tt` and uses our *Content-Type*.

We have seen how we can use httpPOST() to send data from an *R* variable, either as text or as binary. When the data to **POST** are in a file and not in memory, this approach requires us to read them into *R* before **POST**ing them. For large files, this can be an unnecessary and possibly prohibitive step. Instead, we would like to be able to tell `libcurl` to transfer the bytes directly from the file to the Web server. We can use the function CFILE() to create a *C*-level reference to a file. We can then pass this as the value for the *readdata* curl option. `libcurl` will then read the bytes from this file for us. Unfortunately, `libcurl` cannot determine the size of the file, so we have to specify this explicitly by setting the *infilesize* option (or *infilesize.large*). We can use file.info() to compute the number of bytes in the file and so can upload the file with the command

```
httpPOST(targetURL, readdata = CFILE(fN)@ref,
         infilesize = file.info(fN)[1, "size"],
         upload = TRUE,
         httpheader = c('Content-Type' = "text/plain"))
```

We can hide these details in a function, say upload(), that computes the file size and sets all of the options for the caller.

If we had not set the *Content-Type* field in our *HTTP* requests above, httpPOST() would have left it as `"application/x-www-form-urlencoded"`. This is not correct and can be misleading. Often, the script on the Web server will know what to expect and things will work. However, it is much better to always specify the correct MIME type for the content and make it as specific as possible. For example, if we are sending *XML* content, we specify this as `"text/xml"`, not just `"text/plain"`, even though the content is valid plain text. It is more structured than that and so we identify the more specific structure. We can look up MIME types at `http://svn.apache.org/viewvc/httpd/httpd/trunk/docs/conf/mime.types?view=markup` or `http://www.iana.org/assignments/media-types/index.html`. These are reasonably comprehensive and up-to-date. The RCurl package also has a vector of MIME types and commonly used extensions for those types. These are available via

```
data(mimeTypeExtensions)
```

We can use these to programmatically work with MIME types, e.g., to allow *R* users to specify a type such as `"doc"` and we would map it to `"application/msword"`. The function guessMIMEType() does this using `mimeTypeExtensions`.

Many packages on the Omegahat and the rOpenSci repositories make **POST** requests, typically when working with *REST* APIs. For instance, see the RGoogleDocs [20] and `rDrop` [9] packages to see examples of how **POST** is used. These two services and packages are also discussed in Chapter 10 and Chapter 13, respectively.

8.2.6 HTTP's *PUT* Method

The **PUT** method is quite similar to **POST**. Both place the data in the body of the request and we typically use the same curl options to specify the content of the body. So, is there a difference? The answer is yes, but it may not make a practical difference in most cases as we tend to use APIs that tell us whether to use a **PUT** or a **POST** for a particular request. The difference is important when thinking about *REST* APIs, especially in designing them.

The idea behind **PUT** is that we are sending the contents that are to be placed at the specified *URL* to which the request is sent. In contrast, a **POST** request is sending the body to be processed by the target *URL*. That may be a script that performs a computation on the data in the body of our request and then returns a page. In this case, the script at the *URL* is not changed. It may update some data on the server, maybe in another public *URL*, but the script has not been changed. We can think of the difference in heuristic terms, as a **PUT** request assigns the body to the *URL*. A **POST** request is like calling the *URL* as a function with the body as the argument. In *R* terms, this is analogous to

```
POST: URL(body)
PUT:  URL <- body
```

Another way to think about this is that for a **PUT** operation, the caller is in control of specifying the location of the response. A **PUT** request uses the *URL* to specify where the new content/resource will be created. For a **POST** operation, the server is responsible for determining if and where the response is created.

In a **PUT** request, the client is specifying the *URL* at which the new content should appear. In a **POST** request, the server is in control of where any new content should appear, if at all; it may just be returned.

While there may be an important difference in the philosophy of **POST** and **PUT** requests, the details for making a **PUT** request are very similar to the general **POST** request we saw earlier. Generally, we have a block of data that we are uploading as the body of our *HTTP* request. The data may be in a file outside of *R* or data in memory as either a character string or vector, or binary data as a *raw* vector. When dealing with the contents of a file, we can read it into *R* and treat it as *character* string or *raw* vector. Alternatively, we can use the CFILE() function and the *readdata* curl option to let `libcurl` read the contents directly as bytes.

When we work with data already in *R*, i.e., in memory, we can specify it as the body of the request by passing it as the value of the *postfields* option and setting the *upload* option to `TRUE`. As an example, we will look at how we rename a document in Google Docs.

*Example 8-5 Using **PUT** to Rename a Google Docs Document*

The function rename() in the RGoogleDocs package [20] takes an object that describes a Google document and also a new name for the document that will be used in the listing of our documents. We first create an authorized connection between our *R* session and the Google Docs API:

```
gdInfo = getOption("GoogleDocsPassword")
con = getGoogleDocsConnection(names(gdInfo), gdInfo)
```

Next, we get a list of all the documents in our account with

```
docs = getDocs(con)
```

There are several *HTTP* requests that underlie these two *R* commands.

Renaming a document via the Google Docs API requires several pieces of information. We send the *XML* document

```
<entry xmlns="http://www.w3.org/2005/Atom">
 <category
  scheme="http://schemas.google.com/g/2005#kind"
  term="http://schemas.google.com/docs/2007#document"/>
 <title>My new title</title>
</entry>
```

as the body of the **PUT** request. We do this via the *postfields* option and also set fields in the header to specify the *Content-Type* and our authorization information for the request. We do this with

```
httpPUT(url, postfields = saveXML(doc),
        httpheader = c('Content-Type' = 'application/atom+xml',
                       Authorization = 'GoogleLogin auth=xxxxx'))
```

We will see in Section 8.6.1.2 that we can also use the *readfunction* option to send data via the **POST** and **PUT** methods.

8.2.7 HTTP's *HEAD* Method

HTTP provides a **HEAD** method. This is a curious concept. It essentially allows us to make a full request, but the sever only returns the header in the response and no body. This request allows us to examine the header, without having to wait for the body to be received or waste any time or bandwidth processing it. If the body is a large document, this type of request can be useful when we just want to know about the resource, e.g., whether it exists, when it was last changed, how large it is.

We can determine if a *URL* actually exists with a **HEAD** request. We make the **HEAD** request and then examine the header and its status code to see whether it is in the 200 or either the 400 or 500 category. This is what the function url.exists() does. It is a useful function when testing code, much like file.exists(). We use it to first determine if the *URL* is available, and if it is, then we can make the regular request. For example,

```
u = "http://www.omegahat.org"
if(url.exists(u)) data = getURLContent(u)
```

The url.exists() function in RCurl is useful when writing tests for code and packages. As we will see later in this chapter (Section 8.4), it is beneficial to supply the same curl handle to both requests to make use of an existing connection to the server.

8.2.8 HTTP's *DELETE* Method

The **DELETE** method in *HTTP* does what its name suggests—delete a resource associated with the *URL*. This can be a very "strong" action, and one that most servers do not permit. However, if we can create a resource, then we can often delete it. This arises when dealing with *REST* applications such as *ElasticSearch* (a text search engine). This operation is intended to not just change the resource, but remove it entirely. For example, when working with *ElasticSearch*, we may see a request such as

```
httpDELETE("http://localhost:9200/googlenews/")
```

to remove the entire collection of documents under the googlenews index.

The RGoogleDocs package uses a **DELETE** request to remove a document. See the deleteDoc() methods in that package's source code for an example.

8.2.9 *customrequest* and Extended Methods

The *HTTP* specification allows for the addition of new methods. How can we specify these in a request? We use the curl option *customrequest* to specify the method name. For example, we implement httpDELETE() in RCurl simply as

```
getURLContent(url, customrequest = "DELETE", ..., curl = curl)
```

As another example, we can implement an **OPTIONS** request with

```
getURLContent("http://www.nytimes.com", customrequest = "OPTIONS",
              header = TRUE)$header["Allow"]
```

We can also specify any of our other existing *HTTP* methods (e.g., **POST**) using the *customrequest* approach too. This means we can specify methods or operations in our requests that are not officially part of the *HTTP* specification, but which are implemented for a particular server. The *customrequest* option also allows us to write quite general functions in *R* that allow the caller to specify the *HTTP* method. Rather than that function having to select either httpGET() or httpPOST() and so on, it can just set the *customrequest*. Indeed, getURLContent(), httpGET(), etc., are only a thin layer on top of the low-level curlPerform() function.

There are several options in listCurlOptions() that relate to the method used for a request and that *customrequest* handles more generally. For example, setting *httpget* to `TRUE` makes the request a **GET** operation. Similarly, we can use the options *httppost*, *put*, or *post* to set the method, but *customrequest* is more generic. We can even use it to specify non-*HTTP* operations that a particular server understands, e.g. **COPY**.

8.3 Character Encoding

When we make an *HTTP* request and receive text as the body of the response, functions such as getURLContent() and postForm() need to know the character encoding of the text so that they can make sense of the bytes in the response. Fortunately, in most cases, the Web server includes this information in the header of its response, typically as part of the *Content-Type* entry. These functions typically process the header of the response before they process the body.[2] They extract the character encoding and then use this when reading the body, creating the strings in *R* with the correct encoding.

Unfortunately, not all Web servers return the character encoding for the response, and some mistake the encoding. In this case, the RCurl functions can fail or set the wrong encoding for the body. However, the primary functions we use in RCurl each support a *.encoding* parameter. This allows us, the caller, to specify the encoding if we know it. We provide an example below.

[2] It is actually the dynCurlReader() function that is responsible for this step. The high-level functions all use it, by default, to process the response's header and body, to deal with character encoding and identify binary content.

Example 8-6 Specifying the Character Encoding for a US Census Bureau CSV File

The US Census Bureau makes population estimates for different metropolitan and micropolitan areas available in CSV format at http://www.census.gov/popest/data/metro/totals/2011/index.html. Let's read the file CBSA-EST2011-alldata.csv from this site into *R*. We can read it directly in *R* with

```
data = read.csv(uCBSA)
```

where uCBSA holds the string http://www.census.gov/popest/.../2011/files/CBSA-EST2011-alldata.csv. However, if we want to retrieve it with, e.g., getURLContent() we end up with an error:

```
txt = getURLContent(uCBSA)

Error in nchar(str) : invalid multibyte string 1
```

We can see the header of the server's response (with the *verbose* option) and note that it does not have any "charset" information about the character encoding. While the software may be able to handle this case as *R* does, we should explicitly specify the encoding if we know it. We do this with

```
txt = getURLContent(uCBSA, .encoding = "UTF-8")
```

8.4 Using a Connection Across Requests

In our discussion so far of making Web requests, we have focused on single requests. In many cases, we want to make several requests to the same Web server in reasonably rapid succession. For example, we probably want to retrieve the election results for each state rather than just California. Similarly, when we interact with the KEGG API as a SOAP Web Service (see Chapter 12), we typically use the results from one request as inputs to other methods. Similarly, we may query Zillow for information about a particular address and get the unique identifier for that house. We then use that identifier to find comparable houses in a second request. While becoming less common, in some situations, we need to first login to a Web site and then make the requests. Typically, the first request returns a cookie that we must use in subsequent requests to identify ourselves. Each of these involves a sequence of requests to a server. In each request, we need to set all of the options and establish a new connection between *R* and the server. Not only does this make for more code, it also is slow. Connecting to the Web server can take a significant amount of time relative to actually retrieving the document of interest. Indeed, one of the significant changes to *HTTP* in version 1.1 was to allow connections to be kept open between the client and server. This allows the client to make multiple requests using the same connection without having to re-establish it each time. This continued connection led to significant performance gains for Web browsers. When the browser retrieved a Web page, it also had to retrieve numerous images, *JavaScript*, and *CSS* files, and other supporting content that were needed to display that page. Establishing the connection once and reusing it is helpful. We can do the same with RCurl by re-using a connection object across calls and so avoid the overhead of re-establishing the connection each time, although the benefit in our common usage may be less than for Web browsers.

All requests made through RCurl functions use what we call a "curl handle." All of the RCurl functions create one of these as needed and then discard it when the request is complete. However,

these functions allow us to explicitly supply a curl handle in the call via the *curl*. We can explicitly create a handle using the function getCurlHandle(). We can call this function with no arguments or with any number of curl options, e.g.,

```
h = getCurlHandle()
h = getCurlHandle(
      useragent = R.version$version.string,
      followlocation = TRUE, cookiefile =  "",
      cainfo = system.file("CurlSSL", "cacert.pem",
                           package = "RCurl"))
```

We can then use the handle in any of our requests, as shown in the example below.

Example 8-7 Using the Same Connection to Retrieve Multiple Files in a Mail Archive

Earlier we demonstrated how to download the mail archives for the *R*-help mailing list for the month of May, 2012, with getURLContent(). We can make that same request and provide the curl handle with

```
u = "https://stat.ethz.ch/.../2012-May.txt.gz"
gz = getURLContent(u, curl = h)
```

Now if we want to get the mail archives for different months or years, we can reuse this same connection, such as

```
yrs = sprintf("https://stat.ethz.ch/.../%d-May.txt.gz", 2006:2012)
gz.archives = lapply(yrs, getURLContent, curl = h)
```

These calls to getURLContent() all reuse the connection `h` and should be a little faster than recreating the connection across the seven calls.

Any time we create a new curl handle, it merges any options we specify in the call to getCurlHandle() with another set of default "global" curl options. This set of global options is an *R* list of named curl options that can be stored for the *R* session in the options() list under the name *RCurlOptions*. We can use *RCurlOptions* to specify curl options that are applicable for *all* requests and that are specific to our system or *R* session. Useful options to set include

- *followlocation* and *maxredirs* to both follow redirections and also to limit the number to avoid infinite loops;
- *cainfo* to identify the file containing the digital signatures used in verifying *SSL* certificates, e.g.,

  ```
  system.file("CurlSSL", "cacert.pem", package = "RCurl")
  ```

  or an equivalent file on your system;
- *cookiefile* for a file containing cookies to use in requests (or simply to activate cookie management);
- time-out options, such as *timeout*;
- options for using netrc passwords, i.e., *netrcnetrc.file*;
- the full names of the files containing the public and private SSH keys if you are going to be making SSH requests via *ssh.public.keyfile* and *ssh.private.keyfile*;
- a default value for *useragent* set to something like `R.version$version.string`, although *R* packages that use RCurl should explicitly set this option in their requests to identify their package (and the version of *R*).

Note that most of these options are not transmitted in requests, but are used by `libcurl` locally to customize how the request is made. This makes them private and specific to each user. Therefore, these are useful to set globally, for use in any request.

Notice that by setting the options in the call to getCurlHandle(), we did not have to set them in the call(s) to getURLContent(). This can make code somewhat clearer. It also reduces the computations in each request to unnecessarily reset the curl options. We should note, however, that we can add or override any of the options in a curl handle in calls to RCurl functions. Suppose, for example, that we make one request with a connection and then want to make another, but add a particular cookie in that second request, and we want to see the exchange by setting the *verbose* option. We can do this with the same curl handle with something similar to

```
h = getCurlHandle()
getURLContent(url, curl = h)
getURLContent(url, curl = h, cookie = "RMID 8362050d3d9744a5",
              verbose = TRUE)
```

This second request sets the *cookie* and *verbose* options in the curl handle and then makes the request. Note that these functions do not unset or restore the previous values of the options to their values before the call. They remain in effect in the curl handle until they are set again.

While there are definite advantages to re-using a curl handle across requests to the same server, we can use the same handle in requests to different servers. Again, recall that all of the settings in one request will be used in the next request, unless we override them. This means we can set the options once and use them for multiple requests to different servers. This can be convenient. It can also be problematic if there is "private" information that should only be sent to a particular server. For instance, if we set the *username* and *password* (or *userpwd*) options in a curl handle and then use that handle in a request to a different server, that server will see that information as `libcurl` tries to authenticate the request. For example, code such as

```
h = getCurlHandle(cookiefile = "")
ans1 = getURLContent(url1, curl = h,
                     userpwd = "bob:welcome", httpauth = "Basic")
ans2 = getURLContent(url2, curl = h)
```

will lead to the user:password string being sent to `url2` (not in the header but during the authentication negotiation) as well as in the first request to `url1`. We might expect that the *userpwd* would only be in effect for that one request (to `url1`), but that is not how the curl handles work in *R*, for better or for worse.

We should add that this issue of private data being sent to different sites is a problem with passwords, but not with cookies managed by `libcurl` via, e.g., the *cookiefile* option. `libcurl` is smart enough to only send cookies in requests to the site that previously generated the cookie in an earlier response. `libcurl` manages the cookies on a site-specific basis.

We can circumvent the problem of sending private data in a curl handle. We can create a curl handle and set the options that make sense to share for different requests. Then, we can make a copy of that curl handle for each of the different servers. We can then use that in future requests. The benefit of this is that the new copy will contain the same settings for the options as the original curl handle. However, any changes we make to it will be independent of the original handle. The function to create the copy of the handle is named clone() (or also dupCurlHandle()). We can use it to simply copy a curl handle, or we can also override or set new options at the same time. For example,

```
h = getCurlHandle(verbose = TRUE)
doc = getURLContent("http://www.omegahat.org", curl = h)
```

```
h2 = clone(h)
h3 = clone(h, verbose = FALSE)
```

The clone() function gives us a solution to the issue above when setting options in a request and leaving them set in the curl handle we used in the request. Instead, we can use something like the following:

```
h = getCurlHandle(cookiefile = "")
ans1 = getURLContent(url1, curl = clone(h), userpwd="bob:welcome")
ans2 = getURLContent(url2, curl = clone(h),
                     cookie = "RMID 8362050d3d9744a51")
```

In these calls, we explicitly clone the original handle *before* setting any of the options. Since we do not assign the new handles to an *R* variable, they are discarded after the request is complete.

8.4.1 Setting Options in a curl Handle

As we saw, we can pass an existing curl handle in any call to (almost) any RCurl function, and any options specified in that call will be set, and remain set, in that curl handle. However, there are occasions when it is convenient to set options in an existing curl handle, but not actually make a request at the same time. The function curlSetOpt() is used for this case.

It is important to keep in mind that creating new curl handles is quite cheap in terms of computing time and memory. It is good to reuse the same connection to a single server, but even that may not yield major time improvements when working with nontrivial requests. Furthermore, unlike programming in *C* and the `libcurl` API, we can manage and manipulate options entirely as *R* objects and set them in new curl handles quite easily. For example,

```
opts = list(cookiefile = "", followlocation = TRUE,
            cainfo = system.file("CurlSSL", "cacert.pem",
                                 package = "RCurl"))
h1 = getCurlHandle(.opts = opts)
h2 = getCurlHandle(.opts = opts)
```

Being able to clone an existing handle in order to preserve its settings is not very important. Indeed, since a user can use and modify a curl handle in unexpected calls, when we are writing functions that use a curl handle, we often need to explicitly set options in case they are incorrect for our call, e.g., the type of request may have been set to a **PUT** in an earlier call.

For *R* programmers writing *R* functions that make Web requests using RCurl, we strongly encourage you to allow the caller to specify both the curl handle to use in the request and also one or more curl options for that request. For example, we may have a function as

```
fun =
function(arg1, arg2, arg3 = TRUE,
         curl = getCurlHandle(.opts = .opts), ...,
         .opts = list(...))
   getURLContent(url, curl = curl, .opts = .opts)
```

We can then call the function in any of the following ways:

```
fun(1, "abc")
h = getCurlHandle(verbose = TRUE, useragent = "MyApp")
fun(1, "abc", h)
fun(1, "abc", verbose = TRUE, useragent = "MyApp")
fun(1, "abc", .opts = list(verbose = TRUE, useragent = "MyApp"))
fun(1, "abc", getCurlHandle(verbose = TRUE, useragent = "MyApp"))
```

If the function needs to use . . . for its own purposes, the caller can provide a list of curl options via the *.opts* parameter. In those cases, the default value for *.opts* should be simply `list()`.

Setting Options in an RCurl Request

The ability to set options for the Web requests is what gives RCurl its richness. There are several ways to specify them. We can set options for an entire *R* session that are used in each request, or we can set the options for each call. In between, we can set options for a group of requests. In many common cases, we can set the options individually in the call to the *R* function, or specify a collection of them via the *.opts* parameter.

- Use the *RCurlOptions* option in *R* to specify a collection of general-purpose curl options that will be used as the default options in each call, e.g.,

  ```
  caI = path.expand("~/cacert.pm")
  options(RCurlOptions = list(followlocation = TRUE,
                              cainfo = caI))
  ```

  We can set these options for all *R* sessions in the `.Rprofile` file, and we can override any of them in a particular request.
- We can specify options for a particular request using the *.opts* parameter of any of the functions in RCurl (and other packages, too) that make an *HTTP* request using `libcurl`, e.g.,

  ```
  getURLContent(url, .opts = list(followlocation = TRUE,
                                  cainfo = caI))
  ```

- We can create a `CURLHandle` with getCurlHandle() and set options when we create it, e.g.,

  ```
  h = getCurlHandle(.opts = list(followlocation = TRUE,
                                 cainfo = caI))
  ```

- When we specify the curl handle in a call to an RCurl function, we can also override any of its current options in the call. For example, using the handle `h` above

  ```
  getURLContent(url, followlocation = FALSE)
  ```

  overrides the value for *followlocation* in `h` for this request.
- getForm() and postForm() require curl options be specified via the *.opts* parameter. The other functions such as getURLContent(), httpGET(), httpPOST() allow the options be specified *either* as a *list* via the *.opts* parameter or individually via the . . . mechanism, or both. For example,

  ```
  getURLContent(url, verbose = TRUE,
                .opts= list(followlocation = TRUE, cainfo = caI))
  ```

8.5 Multiple Requests and Handles

As we have described, we can create numerous separate curl handles to process different requests and reuse a particular handle for a sequence of several requests. This gives us a lot of flexibility in programming network requests in *R*. However, when we perform a query in *R* either directly using curlPerform() or a higher-level function, control is passed to `libcurl` and *R* must wait until that request has completed. If we have multiple documents to retrieve, we must do them sequentially if using the curl handles as described earlier. However, it is easy to see that if a Web server is responding slowly, or if resolving the Internet Protocol (IP) address of a server from the given name takes a long time because of a slow domain name server (DNS), then *R* and `libcurl` can potentially spend a lot of time idle, just waiting. If `libcurl` can be given several requests at once, then it can send them concurrently and check on each of them to see how they were progressing. A slow Web server would not inhibit the speed at which the other requests were being processed. This is a form of multitasking with which we humans are familiar. It does not necessarily involve a multithreaded programming model (although `libcurl` supports that), but rather an ability to manage multiple requests at the same time and be able to interleave the processing of these requests. RCurl provides an interface to `libcurl`'s "multi" interface for processing multiple requests concurrently.

The function getURIAsynchronous() is a reasonably simple-minded high-level function that we can use to access data for several days from the National Stock Exchange of India Limited. We demonstrate how to do this in the next example.

Example 8-8 Making Multiple Web Requests to NSE India

Instead of retrieving each day's data sequentially from the site as in Example 8-2 (page 265), we might request all of the files for each day's trading in, say, June 2012. For this, we first need to calculate which dates correspond to trading days in this month, i.e., weekdays. We can then use these dates to create the URLs we want to download:

```
june = seq.Date(as.Date("2012/06/1"),
                as.Date("2012/06/30"), by = 1)
days = june[!(weekdays(june) %in% c("Saturday", "Sunday"))]
uIn = paste0("http://www.nseindia.com/archives/",
             "nsccl/volt/CMVOLT_%s")
urls = sprintf(uIn, format(days, "%m%d%Y"))
```

Once we have this vector of URLs, we can retrieve them all in a single call to getURIAsynchronous(). Since each file is quite small and there are only 21 of them, we will repeat the retrieval 10 times to increase the stability of the timings:

```
async = system.time(replicate(10, getURIAsynchronous(urls)))
```

We can compare this to sequentially downloading the files, and we will even reuse the same curl handle across the downloads within each replication. We do this with

```
serial = system.time(
           replicate(10, {curl = getCurlHandle();
                          sapply(urls, getURI, curl = curl)
                          }))

serial/async
```

```
   user  system elapsed
   1.37    1.62    7.34
```

The times will vary, but the results illustrate a significant improvement in the total time to completion — a factor of seven. We should also note that all of the URLs are on the same server. The times for the asynchronous downloads should improve when the URLs are on different servers.

Let's also look at how the ratio of times behaves for larger, binary files. The USGS provides a *KML* file for each year in the period 1973 through to the present. Each file contains information about all of the earthquakes for that year. We assign the "base" *URL*, `http://earthquake.usgs.gov/earthquakes/eqarchives/epic/kml/`, to `uKML`, and then we can easily generate the names of the URLs for the different years with

```
u = paste0(uKML, "%d_Earthquakes_ALL.kmz")
kmz.urls = sprintf(u, 1973:2012)
```

We can then download them all using getURIAsynchronous() with

```
async = system.time({
          getURIAsynchronous(kmz.urls, write = NULL,
                             binary = rep(TRUE, length(kmz.urls))
                             )})
```

We can download them serially/sequentially using the same curl handle object for each request with

```
serial = system.time({curl = getCurlHandle()
                      lapply(kmz.urls, getURLContent, curl = curl)
                     })
```

If we repeat these requests, there is quite a bit of variation in the total/elapse time. While in some tests the serial approach is faster than the asynchronous approach, the asynchronous approach is faster on most occasions, by between 8 and 50 percent. Here we are making requests to the same Web site and this reduces the advantage of making multiple asynchronous requests. In contrast, the user and self times are always lower for the serial approach. The reason for this is partially explained by the fact that the `MultiCurlHandle` object is doing more work to manage the requests. It is constantly checking the different requests to see which is complete. In this case, the elapsed time is the important measure, but this is quite variable due to network latency and Web server behavior. As with most benchmarks, we should be skeptical of the numbers without more investigation. This is a situation where looking at the results from the output of getCurlInfo() for each of the individual `libcurl` handles would tell us more about the performance of each request.

8.5.1 The Multihandle Interface in R

While the getURIAsynchronous() function provides a high-level interface to perform multiple requests concurrently, we can develop other functions that take advantage of the *R* functions that make this possible. This section gives a brief description of these functions and the computational model they use.

This multirequest interface used by the getURIAsynchronous() function uses much of what we have discussed so far, i.e., we can create the same type of requests using the same curl options as for

a single curl handle. The first step is to create a curl handle describing (but not performing) each of the different concurrent requests. We can use either curlSetOpt() or getCurlHandle() to both create and set options, or we can create the handle (along with some options) and set other options later.

Once we have created and set all of the relevant options in each of the curl handles, we gather them all together into a collection of requests. We do this by creating a `MultiCURLHandle` object using the function getCurlMultiHandle(). This object will manage the dispatch of the requests. We can either call getCurlMultiHandle() and then add each of the individual `CURLHandle` objects, or we can pass these `CURLHandle` objects in the call to getCurlMultiHandle(). This is just a matter of the order in which we create and add the individual requests, either before we create the collection handler or afterwards. For example, we can create a collection of individual handles with

```
curlHandles = lapply(urls, function(u)
                               getCurlHandle(url = u))
```

and then pass them to getCurlMultiHandle() with

```
mh = getCurlMultiHandle(.handles = curlHandles)
```

Similarly, we can pass them as individual arguments with

```
rU = "http://www.r-project.org"
oU = "http://www.omegahat.org"
mh = getCurlMultiHandle(getCurlHandle(url = rU),
                        getCurlHandle(url = oU))
```

Note that each `CURLHandle` should have its own *headerfunction* and/or *writefunction* to collect the body of the request. Similarly, each can have its own set of curl options describing the request.

As an alternative to passing the curl handles in the call to getCurlMultiHandle(), we can create the `MultiCURLHandle` object and then add `CURLHandle` objects to it. We do this using the `push` function. For example, we can push each of the curl handles in sequence with

```
mh = getCurlMultiHandle()
mh = push(mh, getCurlHandle(url = rU))
mh = push(mh, getCurlHandle(url = oU))
```

The return value of push() is an updated `MultiCURLHandle` object and we need to assign it to an *R* variable, typically the same variable we pass as the first argument to push().

When we have populated the `MultiCURLHandle` object with all of the requests, we can tell `libcurl` to send all of the requests and process them as they complete, collecting the results. We do this with the function curlMultiPerform(), analogous to curlPerform() for a single `CURLHandle` object. This function then hands control to `libcurl` which processes the requests. When *any one* of these requests is complete, curlMultiPerform() returns control back to the *R* caller, telling us how many requests still remain to be completed. This gives us, the *R* programmer, an opportunity to do something in response to the completion of this one request. For example, we can schedule a new request, reusing the same curl handle.

While we can call `curlMultiPerform` so that each time any request completes it returns, we can also instruct it to return only when all the requests are complete. We do this by calling curlMultiPerform() with the *multiple* argument given as `TRUE`. Alternatively, we can just call the generic function complete(), passing it the `MultiCURLHandle` object. This will then only return when all the requests have been completed. It is important to keep in mind that at any point in time, each of the individual `CURLHandle` objects will process a single request, but *R* will be waiting on multiple concurrent requests.

Once a handle has completed a request, we can use it again for another request or discard it. Just as we can add regular curl handle objects to the multihandler manager with push(), we can also remove them at any time. We use the pop() function for this. To use this, we need to pass the curl handle to remove, e.g.,

```
mh = pop(mh, curl)
```

We need to have assigned the `CURLHandle` objects to some *R* variable so we can refer to them again.

As with push(), pop() returns an updated *R* object containing the managed curl handles. It is important to reassign that value to an *R* variable, typically the one containing the original value of the curl multihandle. Then, to reuse a handle, we first pop() that handle and then set the options for the new request and push() it onto the multihandler again.

This multiple request interface is very powerful. Some more advanced examples of its use are available in the package's documentation and on the package's Web site `http://www.omegahat.org/RCurl`.

8.6 Overview of `libcurl` Options

We have discussed the different high-level functions the RCurl package provides for making requests. We have also seen some of the options we can set to customize the request, both how it is performed (e.g., *followlocation*, *customrequest*) and what it contains (e.g., *useragent*, *httpheader*, *postfields*). In this section, we identify other options that may be useful to explore, and we describe the powerful "callback" options and options for authentication, proxies, and *SSL*. We do not describe the individual options in detail since a long list without context and practical examples is unlikely to make much sense. Instead, readers can consult the `libcurl` documentation for more information.

At the time of writing this chapter,[3] there were 174 options that could be specified for a curl request via the RCurl package. We can get the names of all the options by calling the *R* function listCurlOptions(). These options control a wide range of different aspects of how the request is submitted and how the response is processed as it is received. These correspond to the *C*-level options described in the `libcurl` documentation pages. The names in *R* are direct mappings from the *C*-level names using the following rule. In *R*, the CURLOPT_ prefix is removed, the word is converted to lowercase, and underscores (_) are replaced by periods (.). For example, CURLOPT_NETRC in *C* becomes *netrc* in *R*, and CURLOPT_NETRC_FILE in *C* becomes *netrc.file*.

Of course, each request needs to identify the target *URL*. Almost all of the high-level functions in RCurl take this *URL* as the first argument. The curlPerform() function does not. To use that function, we have to specify the *URL* via the *url* option. We can supply the port number for the request in the *URL* string or separately via the *port* option. The high–level functions set the *url* option for us, so there is no point in passing it as an option in calls to those functions.

The *url* option is the only required option in a request. The other options fall into several categories and are listed in Table 8.2.

Generally, there is a one-to-one mapping between the options available in `libcurl` and those specifiable in RCurl. There are certain ones that do not make as much sense for the *R* interface such as specifying the error buffer that `libcurl` will use for making human-readable error messages available in the event of an error. This is a C-level data structure, and while we can provide an interface to it from *R*, the RCurl package automatically handles reporting errors.

[3] Version 7.26.0 of `libcurl`.

Table 8.2: Primary Categories of curl Options

Category	curl Options
HTTP	parameters controlling the behavior of sessions using HTTP: *useragent*, *httpheader*, *followlocation*, *maxredirs*, *referer*, *autoreferercookie*, *cookiefile*, *cookiejar*, *cookiesession*, *upload*, *postfields*, *postfieldsize*, *postfieldsize.large*, *infilesize*, *put*, *post*, *httppost*, *httpget*, *http.version*, *encoding*, *postredir*, *http200aliases*.
Behavior	controls diagnostic output, whether process-level signals are handled by libcurl: *verbose*, *header*, *progressfunction*, *noprogress*, *nosignal*, *timeout*, *nobody*, *maxconnects*, *customrequest*, *filetime*, *upload*.
Network	target URI, proxy information, network interface and IP resolution parameters: *url*, *protocols*, *redir.protocols*, *proxy*, *proxyport*, *proxytype*, *httpproxytunnel*, *buffersize*, *dns.cache.timeout*, *port*, *tcp.nodelay*, *interface*.
Connections	controls for the socket connectivity between the client and the server: *url*, *range*, *resume.from*
Authentication	names and passwords: *userpwd*, *username*, *password*, *httpauth*, *netrc*, *netrc.file*, *unrestricted.auth*.
Proxies	*proxyuserpwd*, *proxyusername*, *proxypassword*, *proxyauth*, *noproxy*.
FTP	parameters controlling the behavior of sessions using HTTP: *ftpport*, *dirlistonly*, *quote*, *postquote*, *prequote*, *ftp.use.eprt*, *ftp.use.epsv*, *ftp.create.missing.dirs*, *ftp.response.timeout*, *ftp.ssl*, *ftpappend*, *ftplistonly*, *transfertext*.
SSL	facilities for controlling the use of *SSL* and digital certificates: *ssl.verifypeer*, *cainfo*, *capath*, *use.ssl*, *ssh.public.keyfile*, *ssh.private.keyfile*, *ssl.verifyhost*, *keypasswd*, *sslkey*, *sslkeytype*, *sslkeypasswd*.
Callbacks	event handler functions for dynamically interacting with libcurl. *writefunction*, *writeheader*, *writedata*, *readfunction*, *readdata*, *debugdata*, *debugfunction*, *ioctlfunction*, *ioctldata*, *seekfunction*, *seekdata*, *sockoptfunction*, *sockoptdata*, *headerfunction*

This table shows most of the available libcurl options that we can use when making requests with the different RCurl functions. They are grouped into broadly related categories. More information is available at http://curl.haxx.se/libcurl/c/curl_easy_setopt.html.

While there are many options, they are easily grouped into different categories such as *HTTP*, *FTP*, network, and authentication, and this is done in the libcurl documentation. Perhaps the group that needs most explanation in the context of *R* is the collection of callback options.

8.6.1 Asynchronous Callback Function Options

We have seen how we can customize some aspects of how libcurl makes requests such as telling it to follow redirections, supplying a login and password or pointing it to a file containing login and password information, or telling it how long to wait for a response. However, there is a qualitatively different set of options we can specify which we term "callbacks." At certain points when making a request, libcurl can, or needs to, hand control over to some other entity to do something. For example, when we make a request and libcurl starts to process the body, where does libcurl put those characters/bytes? If left to its own devices, it just writes them to the *R* console, and while we can see the contents of the body, we cannot do anything with them. Instead, we need to collect the contents into a string or a *raw* vector. Unfortunately, the very low-level libcurl utilities do not know where to put this content. Functions such as getURLContent() and getForm() take care of this for us by providing an *R* function that libcurl calls each time it receives a block of data from the server as part of the response. This function takes care of collecting the data, and libcurl continues to manage the interaction between the response and *R*. The *R* function is the callback and it acts like

an event handler. The event, in this case, is that `libcurl` has some information to provide. In other cases, the event might be that `libcurl` needs some information, such as being able to read data from a file as input to a request. In another case, the event might be `libcurl` reporting progress about how much of the response it has received. For each of these events, we can specify an *R* function or a *C* routine to call each time the relevant event occurs. The function can process the information in the event, e.g., a part of the response's body, in whatever way it wants and then return control back to `libcurl` which continues what it was doing. `libcurl` might invoke the callback multiple times for each request.

There are several curl callback options in RCurl that are commonly used. The main ones are *writefunction*, *headerfunction*, *readfunction*, and *progressfunction*. We discuss these in the remainder of this section. There are others, such as *debugfunction*, which we briefly explore in Section 8.11. We will not discuss others that are not widely used, such as *opensocketfunction*, *ssh.keyfunction* *ssl.ctx.function*.

In general, only the *headerfunction* and *writefunction* options are needed for a `libcurl` request. These take care of reading the response's header and body, respectively. As important as these options are, we never *have to* explicitly provide values for them when we use any of the high-level functions in the RCurl package, e.g., getURLContent(), getForm(), postForm(), httpGET(), httpPOST(), etc. These functions use the dynCurlReader() function, by default. They set a callback function for reading the header. When the header is complete, the function examines the content type and sets the *writefunction* for the curl handle processing the request to read the body either as a string or as a *raw* vector in *R*. Therefore the defaults just work for all high-level requests. Users who want to use the fundamental curlPerform() function directly or specify their own functions for the curl options *headerfunction* and *writefunction* may want to read the next section that discusses how to customize these options.

8.6.1.1 Customizing the *writefunction* and *headerfunction* Options

Consider a simple **GET** request. `libcurl` constructs the *HTTP* header for the request and sends it to the server. It then waits for the response. When this arrives, `libcurl` reads the header information. We can use curl's *headerfunction* option to provide a function that `libcurl` will invoke for each block/chunk of the header as it receives them. `libcurl` will call this function and pass it the current block of text as an *R* *character* vector with one element. That function can do whatever it likes with the data, and then it passes control back to `libcurl` to continue. A pretty standard approach for processing an *HTTP* header is to collect the chunks until there are no more, i.e., we find a blank line that marks the end of the header. We can then concatenate the chunks into a single string and interpret the header, examining its status code and any of the named fields, e.g., *Content-Type* or *Content-Length*. We can then use this information to determine an appropriate way to process the body. We can the specify a function for the *writefunction* that `libcurl` will call when it receives the next chunk of the body in the response. The RCurl package provides two functions that take care of processing the header and the body of different types of requests. Before we discuss those, however, we look at how we might implement our approach using the *headerfunction* and *writefunction* options in an *HTTP* request.

The function for the *headerfunction* option needs to accept the current chunk as its only argument. It must append this to a character vector of previous chunks in case the header is long and `libcurl` breaks it into chunks. Then, the function needs to see if the chunk contains the ending blank line, i.e., `"\r\n"`. If it does, it parses the entire block of text as an *HTTP* header and determines and sets the *R* function to use for reading the body. Since we are not looking at individual lines but at a string that

contains the line breaks, it is possible that the final part of the header may not be a single empty line. That is, we may have something like "Content-Length: 450\r\n\r\n" as the final chunk so we need to search for the pattern "\r\n\r\n".

When we have the complete header, we can use parseHTTPHeader() to break the text of the header into the different name–value pairs and get the status code, etc. from the first line. We can pass this to our function getBodyHandlerFunction() so it can determine the content type of the body, its length, the character encoding, and any other aspects. If we can recognize the content type as being text and not binary content, we can set the *writefunction* to accumulate the text using a function very similar to the function we use to process the header. If we decide that we need to treat the body as binary data, then we set the *writefunction* to use a different function. We will see how we can do that below using a *C* routine rather than an *R* function. For more specific content types such as *XML* or CSV, we might create a function that accepted a chunk of the body and processed that partial chunk. For example, we can have the *XML* event parser in the XML package read from a document that is being streamed to it in pieces.

Regardless of the nature of the function we use for reading the body, we can hand the function to libcurl for this request being processed via a call to the curlSetOpt() function. We specify the function as the value for the *writefunction* option. Importantly, we need to set this for the CURLHandle object being used in this request. We need to explicitly create it before the call to getURLContent() and not rely on that functions default value for the *curl* parameter. This is simply because we need it in the call to curlSetOpt().

We *might* be inclined to define our header function something like

```
chunks = character()
headerFunction =
function(chunk) {
   chunks <<- paste(c(chunks, chunk), collapse = "")
   if(any(grepl("\\r\\n\\r\\n$", chunk))) {
      hdr = parseHTTPHeader(chunks)
      bodyFun = getBodyHandlerFunction(hdr)
      curlSetOpt(writefunction = bodyFun,
                 curl = myCurlHandle)
   }
   TRUE
}
```

This is not a good programming approach as it uses global variables—chunks and myCurlHandle. Before we show how to improve this with closures, we discuss aspects of the function relating to libcurl and processing *HTTP* requests. Firstly, as we mentioned, this function is likely to be called multiple times while libcurl processes a single header. It may be called for each line of the header, but this is not guaranteed, and the chunks may come in arbitrary lengths. Also, the line breaks ("\r\n") will still be in the value of chunk, unlike, say, the output from readLines(), in *R*. Both of these reasons are why we simply concatenate the different chunks together with no additional white space.

We can fix the code above by using a function that has its own state which it can query and change across calls to that function, but which is not globally accessible in *R*. This is called a closure. Many people find closures somewhat confusing, but in fact, they are quite simple. We do not have the space here to delve into closures generally, so we refer readers to [4] for a more comprehensive discussion. However, the basic strategy for creating a closure is that we can define a function that

returns one or more functions. We can see this more concretely with an example. We can define a function createHeaderHandler() as

```
createHeaderHandler =
function(myCurlHandle) {
  chunks = character()
  header = NULL

  update = function(chunk) {
             chunks <<- paste(c(chunks, chunk), collapse = "")
             if(any(grepl("\\r\\n\\r\\n$", chunk))) {
               header <<- parseHTTPHeader(chunks)
               bodyFun = getBodyHandlerFunction(header)
               curlSetOpt(writefunction = bodyFun,
                          curl = myCurlHandle)
             }
             TRUE
  }

  list(update = update,
       value = function() list(chunks = chunks, header = header),
       reset = function() {
                 chunks <<- character()
                 header <<- NULL
               })
}
```

Let's create two handlers, each with its own `CURLHandle` object:

```
curl1 = getCurlHandle()
h1 = createHeaderHandler(curl1)
curl2 = getCurlHandle()
h2 = createHeaderHandler(curl2)
```

The key idea here in *R* is that each call to createHeaderHandler() returned a list of three functions. The update functions in `h1` and `h2` are very similar, but are importantly different. The code is the same, but where they find the variables `chunks` and `myCurlHandle` is different. This means that they actually have their own instances of these variables and they do not get confused by accessing a shared global variable, like the approach with which we started. We use one of these `update` functions in an *HTTP* request. Since we have not yet defined the function getBodyHandlerFunction(), we just have it return a dummy function `function(txt) TRUE` and we ignore the body of the request by just asking for the **HEAD** in the request.

```
httpHEAD(u, headerfunction = h1$update)
```

Now the `chunks` and the `header` variables in `h1` have been updated. We can retrieve their values with

```
h1$value()
```

However, the corresponding values for `h2` have not been set since `h2` has not been used in a request. When we do use it, the variables for `h1` will not be changed, as both `h1` and `h2` have their own private versions of these variables.

The ability to process the header dynamically before the body of the response is retrieved allows us to collect information from the header settings so that we can prepare for processing the body via the *writefunction* value. For example, we may be able to determine the size or type of the data structure for the body and pre-allocate the space in order to save memory when processing the body. Similarly, we can determine the type of content—simple text, *HTML*, *XML* or binary data—and communicate this to the *writefunction* callback function.

See Section 5.12 for how to use this to parse *XML* documents as streaming content using *SAX*.

8.6.1.2 The *readfunction* and *readdata* Options

We have seen the *writefunction* callback option which can be used to process bytes when libcurl receives them from a server and is looking to write them somewhere. There is a similar *readfunction* option that we can use when libcurl is seeking bytes to send as part of a request. This happens when we are uploading data. As we saw with general **POST** and **PUT** requests, we can often use the *postfields* option to specify the data to be included in the body of the request. This is useful if the data are already in memory in an *R* object. However, when the data are in a file, it is more convenient and efficient to have libcurl read the data directly from the file. We can do this using the CFILE() function and the *readdata* option. By specifying the address of the *C*-level *FILE* object as the value of *readdata*, libcurl will read the contents of the file for us. We should specify the size of the file as the field *Content-Length* in the header by specifying the size via the *infilesize* option. We should also specify the content type by setting the field in the header. For example, if we have permission to make a **PUT** request to a *URL*, we can upload the file with

```
filename = "path/to/file"
httpPUT(url,
        readdata = CFILE(filename)@ref, upload = TRUE,
        infilesize = file.info(filename)[1, "size"],
        httpheader = c('Content-Type' = 'text/xml'))
```

What if the data are not in a file and not in memory? The data may be in an *R* connection that we can read. Alternatively, the data may be in another data structure such as an external object that we can subset from within *R*. In these cases, the entire data are not available so that we can pass them to *postfields*. Similarly, we cannot read them from a file. Essentially, we have a stream of data that we have to pull from as we need more elements—bytes or characters. This is a situation in which we can use the *readfunction* option. We can pass an *R* function for the value of this option. When libcurl needs data, it calls this function with the number of bytes it needs. This function should return either a string or a *raw* vector with the requested length, or less if the remaining data is smaller than that being requested by libcurl.

Let's explore a simple example. Rather than introduce more complexity to the situation, we use an existing file as the source of the data. We can use the CFILE() and *readdata* approach to read this, but we pretend it is a dynamic connection such as a fifo(), pipe(), or url(). The key idea is that the data come from a dynamic connection. We start by creating the connection, e.g.,

```
con = file(filename, "rb")
```

We open this in binary mode since we will not try to interpret the data, but merely pass the bytes to libcurl. This allows us to use the same code for any type of file — text or binary.

Next, we create a function that reads a number of bytes from this file with

```
readDataFun = function(size) readBin(con, raw(), size)
```

This will continue to extract data from the connection, starting at the last point it previously read. We can pass this function as the value of the *readfunction* option in a call to httpPOST() or httpPUT() with the following command:

```
httpPUT(url, readdata = readDataFun, upload = TRUE,
        httpheader = c('Content-Type' = 'text/R'))
```

We do not specify the number of bytes being sent as we may not know this. `libcurl` will call our function, asking for data and will stop asking when we return a vector of length 0, i.e., with no bytes.

As with the advice for the *writefunction* option, we should not use global variables in our functions. Instead, we should use a closure that has access to its connection to from which it reads. We can do this with the function

```
function(con)
{
  function(size) readBin(con, raw(), size)
}
```

or

```
function(filename)
{
  con = open(filename, "rb")
  function(size) readBin(con, raw(), size)
}
```

The function uploadFunctionHandler() in the RCurl package provides help to do this.

8.6.1.3 The *progressfunction* Option

Some requests may take a considerable length of time to complete. This can be due to retrieving a large block of data or to large latency on the network connection, or both. It can be helpful if `libcurl` notifies us about the progress of the request. The *progressfunction* option allows us to pass an *R* function that `libcurl` will invoke while it is processing the request, passing it information about how much data has been sent and received and how much is expected. Our progress function may then display the percentage already completed, either on the console or in a GUI, e.g., using a progress meter. We may even estimate the time to completion.

When we provide a function for the *progressfunction* option, `libcurl` calls that function with two arguments, each a *numeric* vector with two elements. The first argument is for the data being downloaded, i.e., received from the server as the response. The second argument is for what we upload to the server as part of the request, i.e., the body. The first element in each vector gives the total number of bytes involved in the request, if known; the second is the number of bytes processed up to this point in this request, separately for uploading and downloading. As an example, we download a compressed CSV file from the *URL* `http://www.omegahat.org/Data/CBSA-EST2011-alldata.csv.gz`, which is in `uCBSA`. We define our progress function and pass it in the call to the getURLContent() function with

```
progress =
function(download, upload)
   cat(download[2]/download[1], "\n")
```

and

```
z = getURLContent(uCBSA, progressfunction = progress,
                  noprogress = 0L)
```

Note that when specifying a callback for the *progressfunction* option, we must also set the *noprogress* option to FALSE or 0, or else our function will not be called.

libcurl calls the progress function at regular intervals. It is not uncommon that the function will be called before libcurl has received the server's response. This may happen because of a slow connection or because the request requires uploading a large amount of data. These both cause the server's response to be received after libcurl calls the progress function. Similarly, some servers may not include the total length of the response's body in the response's header. In either case, libcurl does not know the total number of bytes in the response and will report it as 0. Any progress function should test for this if it wants to compute percentages.

8.6.1.4 Using *C* Routines as Callbacks

Typically, *R* users will provide an *R* function as the value of a callback option in RCurl, as we have seen above. These are relatively easy to write and modify, and using closures allows one to manage mutable state across successive calls to this function. However, *C* routines are similar in nature to *R* functions when thinking of callbacks. In fact, libcurl can readily call a *C* routine with the correct signature, but we need to explicitly write code for libcurl to call an *R* function. Therefore, it is often useful to be able to specify a *C* routine as a callback. The *C* routine is typically faster because it is compiled and not interpreted code. Since libcurl calls the *C* routine directly, we avoid converting data from *C* to *R* which we would need to do when using an *R* function as the callback. Additionally, it allows for reusing existing code in *C* libraries so there is less programming in *R*.

For any callback option in libcurl, we can specify a *C* routine in our request in *R*. We use the function getNativeSymbolInfo() to get a reference to the *C* routine in *R*. The getNativeSymbolInfo() function takes the name of the routine and the name of the DLL (dynamically loadable library) in which it is located, and returns information about that routine. This information includes its address in memory. This address can be passed to the curl handle as the value of the particular option. Consider our example above where we wrote our own function for the *headerfunction*. We discussed using a *C* routine for reading binary data in the body of the request. The RCurl package contains a *C* routine to do this named R_curl_write_binary_data. It is defined so it can be used as the value for the *writefunction* option and libcurl will be able to invoke it directly, passing it each chunk of the body as libcurl receives it. To set this as the value of *writefunction*, we use the command

```
cfun = getNativeSymbolInfo("R_curl_write_binary_data",
                           "myDLL")$address
getURLContent(url, writefunction = cfun)
```

or

```
curlSetOpt(writefunction = cfun, curl = theCurlHandle)
```

This may work, but not properly as we cannot access the results and the routine does not store it anywhere. The *C* routine R_curl_write_binary_data needs a place (memory) into which it will put the binary data as it processes them. The routine actually expects this to be passed to it as part of a *C*-level data structure that it can update across each call. To do this, we need to first create an instance of that *C* data structure and then tell libcurl to pass it to our *C* routine each time it calls it while processing this request's body. libcurl lets us set this with the *writedata* option. We create an

instance of this data structure with the *R* function binaryBuffer(). Then we can set the *writefunction* and *writedata* together with

```
buf = binaryBuffer(1000)
getURLContent(url, writefunction = cfun, writedata = buf@ref)
```

When the call to getURLContent() completes, we can get the contents of the buffer with

```
data = as(buf, "raw")
```

We should note that the argument to binaryBuffer() is the initial length of the buffer to allocate and we can guess a suitable value. If the data are actually bigger than this, `R_curl_write_binary_data` will enlarge the buffer as needed, but this will add some overhead to the processing. Therefore, it is best to create the binary buffer after processing the header of the *HTTP* response. We typically can determine the length of the body exactly from the header and so create a buffer with the correct length at that time.

There are several other cases in which we can use the approach of having a *C* routine for a callback and creating an instance of a *C* data structure to use as data that `libcurl` passes to it each time it calls the routine. There is a convenient special case of this *C*-level approach for callbacks. Sometimes, instead of reading the body of the response directly into an *R* session, we may want to write it to a file like download.file() does. We can do this by specifying a pointer to a *C*-level *FILE* data type as the value for the *writedata* option. `libcurl` assumes the value for *writedata* is a *FILE* when no value for the *writefunction* option is set, and uses a default callback routine that writes the data to the file.

We create the *FILE* instance in *R* with the function CFILE(). In this case, we want to be able to write to the file, so in addition to the name of the target file, we specify the mode as `"w"`. We can use this in the request

```
curlPerform("http://www.r-project.org",
            writedata = CFILE("/tmp/rproject.html", "w")@ref)
```

The full contents of the file will be written and the file closed when the object created by CFILE() is garbage collected. It is a good idea to explicitly call gc() after a call like this. Alternatively, we should have functions to explicitly close a `CFILE` object.

We can generalize from the *FILE* data type and develop *C* and *R* code for creating, populating, manipulating, and deleting other *C* data types. We can programmatically generate this code using tools such as RGCCTranslationUnit [13] or RCIndex [15], and we can also generate dynamic bindings using packages such as `rdyncall` [1] and Rffi [16]. We can then work with instances of these *C* data types and use them to parameterize other callback options and routines. This is a useful approach if we want to work with the full collection of callback options that `libcurl` provides.

8.6.2 Passwords for Web Pages

Some Web sites require a user name and password to access certain files. This is different from having to login to the Web site *before* accessing any of the files. It is also different from having to obtain a token to access resources on the Web server. Instead, we are talking about passwords on particular directories or files on the Web server.

We can specify a login and password in a request via the *userpwd* option for curl. This is a simple string that gives the user name and password in the form user:password. The Web site for the RCurl package provides a test for this at `http://www.omegahat.org/RCurl/testPassword/`

index.html, which we have assigned to the variable uTP. The user name is "bob" and the password is "welcome". We can request the page with the *R* command

```
getURLContent(uTP, userpwd = "bob:welcome")
```

We can also specify the login and password separately with the *username* and *password* options, e.g.,

```
getURLContent(uTP, username = "bob", password = "welcome")
```

However, we should not put private information directly in code. Items such as a user name or password should be referenced indirectly. One way to do this is to set the information as an *R* option, such as

```
options( OmegahatLogin = "bob:welcome" )
```

We can then put this information in a file such as our .Rprofile that *R* reads each time it starts. When we want to use the information, we use getOption(), e.g.,

```
getURLContent(uTP,
              userpwd = getOption(OmegahatLogin,
                                  stop("no OmegahatLogin option")))
```

It is important to ensure that the file containing the passwords can only be read by the owner and nobody else on the system.

Putting user names and passwords in a .Rprofile is, of course, *R*-specific. There are other simple, general formats for storing login information that many applications can read. One format is called "netrc". Some people like to keep passwords in a "netrc" file, often in their home directory in a file named .netrc so that applications know where to find it. A "netrc" file contains lines of the form:

```
machine www.omegahat.org  login bob  password welcome
```

We can have a separate line for numerous machine, login, and password triples. libcurl can read such files and use the passwords. We can tell it to use the default netrc file via the *netrc* option. We pass a value of TRUE which indicates that the file is optional. If it is present, then libcurl will read it to find the password. However, we can also specify the password via the *userpwd* option to override values in that file. Assuming we have the file .netrc in our home directory and it contains the above line, we can use the following *R* command to access the restricted page:

```
getURLContent(uTP, netrc = "optional")
```

Note that we used the symbolic value "optional" rather than the C-level value 1 for the option. We can use either, and the RCurl package coerces the value to the associated enumerated value.

If the netrc file is not in a standard location, or we want to use a different one, we can use the *netrc.file* option to specify its full path:

```
getURL(uTP, netrc = 1,
       netrc.file = path.expand("~/otherPasswordFile"))
```

libcurl supports several different authentication mechanisms such as Basic, Digest, GSS, NTLM. libcurl attempts to determine what authentication mechanism is used by the server. By default, libcurl starts with a regular request with no authentication and then tries each of the authentication mechanism in turn until one works. If we know the type of authentication being used, we should specify it so that libcurl can use that directly. We do this via the *httpauth* option. For instance, many Web sites use Basic authentication, and we can limit libcurl to use just the Basic scheme with

```
getURLContent(uTP, userpwd = "bob:welcome", httpauth = AUTH_BASIC)
```

If we want to allow either Basic or Digest, we can merge these together with `AUTH_BASIC | AUTH_DIGEST` or `c(AUTH_BASIC, AUTH_DIGEST)`.

8.6.3 Cookies

Cookies are used in an *HTTP* conversation, i.e., repeated requests to a server by the client and server, to provide state information across requests. Cookies allow a server to send information in the header of an *HTTP* response that the client will then send back in subsequent requests. The cookies therefore preserve state across calls and allow the server to use the data in the cookies to continue from where it last left off. Cookies typically contain small amounts of data such as the key or identifier that the server can use to look up more complete data locally.

Cookies are commonly used to identify the client as being the same individual across requests. For example, we may visit a Web site and be redirected to its login page. After that, the Web site remembers us, gives us access to protected pages and customizes the pages using information it has stored about us on its local machine. The key is that there is a connection between logging in and subsequent requests. This is done via cookies. When we enter our login and password for the Web site, the server returns not only the welcome page, but also a cookie in the response's header that uniquely identifies this session between us and the server. We send this cookie back in the header of each of the subsequent requests and the server recognizes us and can restore the state of our session.

Of course, a server does not necessarily want to maintain a session indefinitely. Accordingly, cookies can have an expiration time, after which the cookie is invalid. In our example of logging into a Web site, the user would have to login a second time.

For security and privacy reasons, we should not send a cookie from one server to another server. That second server might then use it to impersonate us and access our session.

Manually managing cookies can be nontrivial. Fortunately, `libcurl` can take care of all of the details for us. We can use a collection of curl options both to activate cookie management and also to take control and customize many of details for specific "conversations."

We start with a somewhat unusual but simple case involving a single request from *R*. Suppose we happen to know the value for one or more cookies that a server expects when we make a request. Then we can specify these in the request as the value of the *HTTP* header field named *Cookie*. We can specify this via the *httpheader* option, but it is easier to use the more direct *cookie* option, e.g.,

```
getURLContent(url, cookie = "known cookie value")
```

Here is an example.

Example 8-9 Using Cookies to Access the Caltrans Performance Measurement System (PeMS) Site

The Web site `pems.dot.ca.gov` provides an enormous amount of information about traffic on different highways in California. We can get historical data for a given location for each lane on a particular highway. We may be interested in the number of cars passing that location (flow) within a particular aggregation interval, and also the proportion of time there was a vehicle over the detector (occupancy) for that same aggregation interval. The data are available for different intervals such as one second, 5 or 15 minutes, and hourly.

The Web site allows us to navigate through a sequence of pages to select the highway and location/detector of interest. At the end of this, we come to a form and can select the start and end dates and the variables in which we are interested. Before we can do any of this, however, we need to login

to the Web site. We do this via their main page, having applied for a free account and received the login and password. We can find the cookies that the server sent back when we logged-in via our Web browser. Precisely how we do this depends on which Web browser we are using.

The PeMS Web site sets cookies named PHPSESSID, __utma, __utmb, __utmc, and __utmz. The values are not intended for humans. However, we can copy them from the browser and create a string in *R* of the form:

```
"PHPSESSID=xxxx;__utma=yyyy;__utmb=zzzz;........"
```

Note that each cookie is given in the form of `name=value` and the cookies are separated by a `';'`.

The traffic data are available through a **GET** form. We will discuss how we find the names and possible values of the parameters in Chapter 9, but for now we just put the values for our particular request into a *list* in *R* as

```
args = list(report_form = 1, dnode = "Controller",
            content = "detector_health", tab = "dh_raw",
            export = "text", controller_id = 404250,
            s_time_id = 1338508800, s_mm = 6, s_dd = 1,
            s_yy = 2012, s_hh = 0, s_mi = 0,
            e_time_id = 1339135200, e_mm = 6, e_dd = 8,
            e_yy = 2012, e_hh = 6, e_mi = 0,
            lanes = "402826-0", q = "flow", q2 = "occ", gn= "5min")
```

We can now use `args` and our cookie string to make the same request our Web browser can with

```
uP = "http://pems.dot.ca.gov/"
tt = getForm(uP, .params = args,
             .opts = list(followlocation = TRUE, cookie = cookies))
```

The result is a string that is actually a tab-separated document. We can read this into *R* with

```
flowOcc = read.table(textConnection(tt), sep = "\t", header = TRUE)
```

Retrieving the cookies manually from the Web browser is not ideal. Firstly, if we have to look them up manually by searching in the GUI, then cannot script the process and make it programmatic and reproducible. We can read the cookies for certain browsers from the local files in which they are stored. See [14] for some examples of approaches to do this. The second issue is that we have to login to the PeMS account via a Web browser. It would be better to do this entirely within *R* so that, again, we do not need to have a human involved. If we can find a way to perform the login step in *R* and get the cookies the server returns in that request, then we can use those in our request for the flow and occupancy data. We make one request to login and a second request for the data. Importantly, we use the same `CURLHandle` object to make the two requests and we will enable `libcurl`'s cookie management.

Logging into our PeMS account involves submitting a form via a **POST** request. The form parameters are named *username* and *password*. We do not send these via `libcurl`'s *userpwd* option, but instead submit them as actual values in the form,

```
pems = getOption("PEMSLogin")
curl = getCurlHandle()
doc = postForm(uP, username = names(pems),
               password = pems, curl = curl,
               .opts = list(cookiefile = "", verbose = TRUE))
```

The option *cookiefile* tells `libcurl` to enable cookie management and also to read any cookies in the specified file given by the value of the option. We use the empty string for the *cookiefile*, which clearly does not correspond to any file, but just activates the cookie manager.

With the *verbose* output from our postForm() request, we see the lines (which we have reformatted for display)

```
* Added cookie
  PHPSESSID="2df765926eeee73a585afbed10477db4" for
  domain pems.dot.ca.gov, path /, expire 0
< Set-Cookie: PHPSESSID=2df765926eeee73a585...; path=/
```

This tells us that `libcurl` found the PHPSESSID cookie.

Next, we can submit our request to get the tab-delimited document. We can use much the same call to getForm() as we did above, but this time we do not need to specify the cookie. Instead, we pass the same curl handle that we used when logging in. Our simpler call is

```
tt = getForm(uP, .params = args, curl = curl)
```

On occasion, it can be convenient to write the cookies that have been collected during a sequence of requests to a file. We can arrange for this by specifying a value for the *cookiejar* option in our `CURLHandle`. Unlike with *cookiefile*, `libcurl` does not try to read cookies from this file. Instead, it writes the cookies to this file when the curl handle is discarded, or when we explicitly force `libcurl` to write the cookies. For example, if we want to store the cookies for PeMS login, we can use

```
curl = getCurlHandle(cookiejar = "/tmp/cookies")
doc = postForm(uP, username = names(pems), password = pems,
               curl = curl, .opts = list(cookiefile = "",
                                         verbose = TRUE))
```

`libcurl` does not write the cookies to the file at this point. Instead, it waits until the curl handle is garbage collected. Alternatively, we can use the *cookielist* option and the special value `"FLUSH"` to write the current cookies to the file. For example,

```
curlSetOpt(cookielist = "FLUSH", curl = curl)
```

will also create the file `/tmp/cookies`. The file will then have something like the following contents

```
# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This file was generated by libcurl!
# Edit at your own risk.

pems.dot.ca.gov FALSE   /   FALSE   0
PHPSESSID        63d2eeeff0d277982ed811c2bb3040c1
```

We can use the cookie(s) it contains in another curl handle by specifying this file name as the value for *cookiefile*, e.g.,

```
curl = getCurlHandle(cookiefile = "/tmp/cookies.txt")
```

We can then use this in our call to getForm() to get the CSV document without having to login again.

The RGoogleTrends package [14] provides an example of manipulating cookies within *R* after a request that is more complex than those we have explored in this section.

8.6.4 Working with SSL and Certificates

The Secure Socket Layer (SSL) is a mechanism that encrypts data sent between a client and a server (on a socket). Basically, we create a connection and the client and server agree on keys they can use to encrypt the contents so that no third-party can intercept the contents and decrypt them. This allows us to send any private data such as passwords, financial data, and confidential data about individuals. Secure *HTTP*, i.e., *HTTPS*, is simply *HTTP* on a secure socket. One of the features of libcurl and RCurl is its support for *SSL* and *HTTPS*. This is an increasingly important facility, especially as we move towards using *OAuth* 2.0 for Web services which requires *HTTPS*, and generally for passing private data. Whether a particular installation of the RCurl package has support for *HTTPS* depends on the installation of the libcurl library. We can determine whether *HTTPS* is supported using the curlVersion() function and the command

```
"https" %in% curlVersion()$protocols
```

If this returns TRUE, then we can make *HTTPS* requests. Then, typically, we only need to use the *HTTPS* qualifier for a *URL* to use this encrypted connection and libcurl takes care of all of the *SSL*-specific details. However, there can be some issues.

When we use *SSL* to make a request, there is a lot more security involved. All of the exchange is protected so that nobody can snoop on the packets and see their contents. However, there is an additional important step. When we first connect to the server, we verify that it is the server we think it is, and not a server masquerading as our target. The way we verify this is basically that the server sends us a digitally-signed certificate. This is signed by a trusted entity (e.g., a centralized security company) who puts their signature on the certificate. This way, when we connect, we verify this third-party signature of the trusted entity. To do this, we need to recognize these trusted authorities and their signatures. Since there is a small number of these, we can have their public signatures against which we check. If we cannot verify the signature, then it is possible that the server is not who it claims to be. In this case, we should not be sending private data to it.

The first step in an *SSL* connection is to validate the server's certificate against a set of public signatures. For this, we need libcurl to find these public signatures. If it cannot, we get an error such as

```
gz = getURLContent(uRH)

Error in function (type, msg, asError = TRUE)  :
SSL certificate problem, verify that the CA cert is OK.
Details: error:14090086:SSL
routines:SSL3_GET_SERVER_CERTIFICATE:
certificate verify failed
```

(Here the *URL* in uRH is https://stat.ethz.ch/pipermail/r-help/2012-May.txt.gz.) On many systems, libcurl will use the default collection, and no error will occur. However, on other systems, we have to tell libcurl where to find these signatures, and this can be a file containing one or more signatures or a directory containing files with individual signatures. The RCurl package installs the "standard" set of public signatures and we can find this with

```
sigs = system.file("CurlSSL", "cacert.pem", package = "RCurl")
```

We can then pass this as the value of the *cainfo* option as in

```
gz = getURLContent(uRH, cainfo = sigs)
```

This verifies the server's signature and proceeds to establish the *SSL* connection.

We can also tell `libcurl` to use a directory of individual certificates via the *capath* option. This allows us to add new signatures easily.

There are occasions when a server does not have a digital certificate, but we know that it is who it claims to be (or are willing to take the chance). In this situation, we can avoid validating the server by using the *ssl.verifypeer* option and setting it to `FALSE`. For example,

```
gz = getURLContent(u, ssl.verifypeer = FALSE)
```

This is a potentially dangerous approach and you should be cautious.

There are numerous other options that control how the *SSL* layer connects to the server. Some of these are technical and require a good understanding of *SSL*, so are outside the scope of this chapter. Readers are referred to both the `libcurl` and openssl [8] documentation and `www.openssl.org` for more information.

8.6.5 Using a Proxy Server

People often access Web pages through a proxy server. The proxy acts as an intermediary for the request. The client asks the proxy to send the request, the proxy does this, and then the proxy returns the response. A proxy *can* be efficient when the proxy caches pages locally so it can return a local version to the client without having to make the request to the remote server. Of course, the extra intermediate steps can also lead to slower requests. A proxy also keeps the client machines anonymous. This is helpful for security and allows the machines to be behind a firewall. A proxy also allows people to browse the Web anonymously, i.e., without identifying their machine as making the requests.

We can use a proxy for requests using RCurl. At its simplest, we need only identify the proxy machine's address (either by name or an IP address) using the *proxy* option. In the next example, we demonstrate how to use a publicly available proxy server.

Example 8-10 Making a Web Request Through a Proxy Server

There is a list of publicly available proxy servers at `http://www.proxy4free.com/list/webproxy1.html`. We choose a machine in France with the IP address 80.14.30.132. We connect to it using port 3128. We can make our request through this proxy with

```
uPr = "http://www.iplocation.net/tools/ip-locator.php"
txt = getURLContent(uPr, proxy = "80.14.30.132:3128",
                    followlocation = TRUE)
```

The *URL* we have requested is a service that identifies the location of the machine making the request. By looking at its contents (either in a browser or in *R*) we see that it reports the request as coming from a machine in France and so it sees our proxy and not our machine.

In our request, we appended the port to the proxy's IP address. We can specify the two separately by specifying the port via the *proxyport* option, e.g.,

```
txt = getURLContent(uPr, proxy = "80.14.30.132",
                    proxyport = "3128", followlocation = TRUE)
```

For proxies that require a login and password, we use the options *proxyuserpwd* in the same way that we do *userpwd*. Similarly, we can specify the two values separately using the options *proxyusername* and *proxypassword*. We can also identify the authentication type with *proxyauth* in the same

manner as we can for *httpauth*. When using a proxy that requires authentication, it may be beneficial to make an initial request to authenticate with the proxy and reuse the same connection in subsequent calls. Also, enabling cookies (with, e.g., `cookiefile = ""`) may avoid re-authenticating on each request.

8.7 Getting Information About a Request

When a request has been processed, `libcurl` stores information about the processing of that request in the curl handle. This includes information such as the effective URI (after redirects, etc.), cookies sent back by the server, the status or "response code" of the request, the content type of the response, the size of the different parts of the request and response, the total time and the times for the different subtasks, etc. Given the curl handle used to make the request, this information can be retrieved by calling getCurlInfo(). For example,

```
h = getCurlHandle()
getURI("http://www.omegahat.org/RCurl/index.html", curl = h)
names(getCurlInfo(h))
```

```
 [1] "effective.url"           "response.code"
 [3] "total.time"              "namelookup.time"
 [5] "connect.time"            "pretransfer.time"
 [7] "size.upload"             "size.download"
 [9] "speed.download"          "speed.upload"
[11] "header.size"             "request.size"
[13] "ssl.verifyresult"        "filetime"
[15] "content.length.download" "content.length.upload"
[17] "starttransfer.time"      "content.type"
[19] "redirect.time"           "redirect.count"
[21] "private"                 "http.connectcode"
[23] "httpauth.avail"          "proxyauth.avail"
```

Of course, in order to be able to get this information, we need the `CURLHandle` used to make the request. Therefore, we need to create the curl handle separately rather than rely on a default handle being created in, e.g., getURLContent(), so that we can pass that same handle to getCurlInfo().

This information is useful in several ways. In simple cases, we can use it to check if the request was successful, i.e., by looking at the `response.code`. Similarly, we can find the content type and the number of bytes or characters the response contains. Generally, we cannot wait until the request is complete for this information and must take care of it with our own functions used as callbacks for the *headerfunction* and *writefunction* options.

The information from getCurlInfo() also gives us metadata about the request and its performance. For one, it allows us to change subsequent repeat queries to the actual URI if it is different from the nominal URI. Also, it allows us to measure the different characteristics of our communication and potentially dynamically understand the operating characteristics of our software that uses the *HTTP* requests and optimize it, if this is relevant.

8.8 Getting Information About `libcurl` and Its Capabilities

It is often useful to know which version of `libcurl` and its sublibraries we are using. We also want to know what functionality or capabilities our version of `libcurl` supports. We can use the function curlVersion() to query this kind of information for a particular installation of `libcurl` and RCurl. An example of the output is

```
curlVersion()

$age
[1] 3

$version
[1] "7.26.0"

$vesion_num
[1] 465408

$host
[1] "x86_64-apple-darwin11.3.0"

$features
     ipv6       ssl      libz      ntlm largefile
        1         4         8        16       512

$ssl_version
[1] "OpenSSL/0.9.8r"

$ssl_version_num
[1] 0

$libz_version
[1] "1.2.5"

$protocols
 [1] "dict"   "file"   "ftp"    "ftps"   "gopher" "http"   "https"
 [8] "imap"   "imaps"  "ldap"   "ldaps"  "pop3"   "pop3s"  "rtsp"
[15] "scp"    "sftp"   "smtp"   "smtps"  "telnet" "tftp"

$ares
[1] ""

$ares_num
[1] 0

$libidn
[1] ""
```

The result is a list with 12 named elements detailing the different information. This is a regular *R list* object. We have information about both the supported protocols and also some of the features that may have been enabled when installing `libcurl`. We can use these to determine if a particular protocol is available before making a request. For example, we can use *HTTPS* if it is supported, or fall back to regular *HTTP* if it is not. We can do this with

```
hasHTTPS = "https" %in% curlVersion()$protocols
```

The element `ares` refers to asynchronous host name resolution and whether this is enabled. A "libidn" is a library that provides facilities for working with internationalized domain names. The `libidn` field in the result of curlVersion() tells us if this is enabled.

8.9 Other Protocols

The focus in the chapter has primarily been on *HTTP*. However, `libcurl` has support for numerous other protocols. We can determine which of these were available when our version of `libcurl` was compiled using curlVersion():

```
curlVersion()$protocols
```

```
 [1] "dict"   "file"   "ftp"    "ftps"   "gopher" "http"   "https"
 [8] "imap"   "imaps"  "ldap"   "ldaps"  "pop3"   "pop3s"  "rtsp"
[15] "scp"    "sftp"   "smtp"   "smtps"  "telnet" "tftp"
```

There are potentially others (e.g., RTMP, Real Time Message Passing) that can be enabled if the relevant supporting libraries are available on your machine when compiling `libcurl`.

8.9.1 Secure Copy (scp)

We look at one of these protocols— scp, Secure Copy —to see how we can use it in a manner very similar to *HTTP* to transfer files from an account on one machine to an account on another machine. Basically, we can reuse getURLContent() or the low-level curlPerform() and specify the *URL* with the SCP protocol, e.g., `scp://duncan@host.machine.com/Users/duncan/catalog.xml`. (If the user name is the same on the current and target machine, we do not need to provide the login in the URL.) We can either use a password or rely on our public ssh key being registered with the remote machine's account (in the `authorized_keys` file). We can specify the password for the remote account using the *password* option:

```
uS = "scp://duncan@localhost/Users/duncan/catalog.xml"
o = getURLContent(uS, binary = FALSE, password = pwd)
```

Alternatively, to use the local ssh keys, we need to include their location. We can do this using

```
o = getURLContent(uS, binary = FALSE,
                  ssh.public.keyfile =
                    path.expand("~/.ssh/id_rsa.pub"),
                  ssh.private.keyfile =
                    path.expand("~/.ssh/id_rsa"))
```

Here we use the *ssh.public.keyfile* and *ssh.private.keyfile* options and give them the full paths to the relevant files. Since these are usually in the `.ssh/` directory within our home directory, we can use `~/.ssh`. The getURLContent() function and the other functions do not know that these are path names and so we have to call *R*'s path.expand() function to replace the `~` with the actual home directory.

If our ssh key requires a passphrase to access it (a good thing), we can specify this via the *keypasswd* option, e.g.,

```
o = getURLContent(uS, binary = FALSE, keypasswd = "myPassphrase",
                  ssh.public.keyfile =
                    path.expand("~/.ssh/id_rsa.pub"),
                  ssh.private.keyfile =
                    path.expand("~/.ssh/id_rsa"))
```

This illustrates that we use the same basic approach for different protocols. We specify the *URL* and any curl options and eventually call curlPerform(). We do this indirectly through getURLContent() as this establishes the appropriate *writefunction* and *headerfunction*. We automatically inherit the ability to specify whether the resulting content is to be treated as *binary* or not.

The RCurl package does provide an scp() function to make this slightly simpler to call, as it uses sensible default values for the file names for the public and private keys.

8.10 *HTTP* Errors and *R* Classes

Any Web request that we invoke from *R* can lead to an error. These can happen for several reasons, basically grouped into one of three categories:

1. We may just have a regular error in *R* code such as a typo in referring to a variable or function, or having an `NA` or empty vector in an `if` condition, for example.
2. We can also have an error in the *HTTP* request, such as a request to a non-existent *URL*, or an incorrect authentication/password.
3. Alternatively, we can have an error that comes, not from the Web server itself that is processing the *HTTP* request, but code on that Web site that is processing the request, e.g., a script processing a form submission or a method from an API.

The first source of errors gives rise to the usual *R* error. These typically lead to a call to stop() with an error message. The user deals with these using whatever debugging strategy normally used.

In the last of these cases, where a script on the Web server generates an error, that script may raise either an *HTTP* error (e.g., with a status code in the 400 or 500 groups), or return a regular *HTTP* result (i.e., status 200), but the body of this is a document that describes the error. The latter is what happens in *SOAP* and XML-RPC and many *REST* APIs. These types of errors are generally handled by specialized functions that are responsible for making the request in the first place, e.g., the .SOAP() or xml.rpc() functions. Both of these recognize that the document returned to it indicates an error so they process that and generate their own type of error. For instance, the xml.rpc() function generates an error of class `XMLRPCError` that contains the error message returned to us by the XML-RPC method. Similarly, the .SOAP() function creates and raises an error of class `SOAPError`.

The fact that these types of errors have their own classes makes it possible for users and programmers to differentiate between them based on their class, and not on the content of the error message. This gives us an opportunity to respond to different error messages in different ways. We can do this

using the tryCatch() function. Before we see how to do this, we consider the second category of errors where we get an error in the *HTTP* request. Since it is useful to have an error identify the nature of the error via a class, we have done this in the RCurl package. Basically, functions such as getURLContent() and curlPerform() examine the status of the response to the request. When they detect an error, they use the status code to generate an error with a class corresponding to that status value.

When the errors have distinguishable classes, we can use tryCatch() to provide customized handlers for the different classes of errors. This function takes an *R* command or expression to evaluate, and zero or more additional named arguments, each of which identify the name of the class of the error (or warning) and a function to handle that type of "condition" (a warning or error). For example, suppose we can send the same request to two different servers (or URLs) and get the same result. One is fast, but services a lot of requests; the other is slower and is used by a small number of users. We may want to make the request to the faster server, but switch over to the second server if we do not get a response in say, 3 seconds. We can do this with

```
o = tryCatch(getURLContent(url1, timeout = 3),
             OPERATION_TIMEDOUT =
               function(e) getURLContent(url2, timeout = 3),
             error = stop)
```

If the initial call to getURLContent() times out, we get an error of class `OPERATION_TIMEDOUT`. The tryCatch() function then calls our error handler for this class so we make the request to the second *URL*. For any other type of error, we call stop() and raise a regular *R* error.

There are over 80 different error classes that the RCurl package can raise, corresponding to the different error codes raised by `libcurl`. Rather than list them here, the package provides the function getCurlErrorClassNames() that returns a character vector of all the class names.

In some cases, the Web server may provide more information about the error as shown in the next example.

Example 8-11 Catching Errors in a Request to Open Street Map

The Open Street Map API allows us to retrieve a map for a given rectangular region. We can call this with

```
uOS = "http://api06.dev.openstreetmap.org/api/0.6/map"
xml = getForm(uOS, bbox = "-123,37,-122,38")
```

However, there is a limit on the width and height of the rectangular region. As a result, the Web server raises a `"Bad Request"` error with status code 400. The getForm() reports this as a generic `HTTPError` with the more specific `Bad_Request` class with a `message` string `"Bad Request"`. However, the Open Street Map server has returned a more informative error message in the Error field in the *HTTP* header. We can access this and use it as a better error message with

```
tryCatch(getForm(uOS, bbox = "-123,37,-122,38"),
         Bad_Request = function(e) {
            e$message = e$httpHeader["Error"]
            stop(e)
         })
```

A list of the status codes and their meanings is available at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

8.11 Debugging Web Requests

As with all programming, there will be errors, mistakes and bugs when making Web requests. The fact that our requests involve a client (our code) and a server (the Web server) makes debugging more complicated. We typically do not have access to the Web server and therefore we cannot see what it is doing, i.e., what it actually receives and what it does with this. We can see what we send and what the server returns. We can do this using `libcurl`'s *verbose* option and we can even use the *debugfunction* option to collect the information rather than just seeing the content on the console. We will explore this option later in this section. There are several other approaches that we also use in different circumstances. We can send our request to a different Web server where we can see what it receives in both the header and body. Fortunately, there is a publicly available server (`requestb.in`) that provides such as a service. Also, in many cases, we are mimicking what we do in the browser, e.g., with a *JavaScript*-based form and request. We can use the Web browser's developer tools to examine the details of the request and what is sent and received. A lower-level, but sometimes useful, approach is to use a network monitoring tool to examine the packets that are sent to and from the server as part of our requests and responses.

Diagnosing Web Request Errors

The `requestb.in` site provides a useful free service for helping to diagnose *HTTP* request issues. The idea is simply that we can create a unique *URL* on their server to which we can send our requests. They store each request we send, and we can then view their details in our Web browser. For each request, we can see the method, the header information, and the body or content, which consist of the parameters in the case of a **GET** request with data. To get started, we visit the `requestb.in` home page and click on the "Create a RequestBin". This displays a *URL* in a text field and we use this in our requests from *R*. For example,

```
httpPUT("http://requestb.in/ybrialyb",
        postfields = "This is a % sign & an ampersand",
        user = "bob", password = "welcome")
```

We can see this and other requests we made by visiting this URL in our Web browser and clicking on the bin of interest. This allows us to see what the *R* functions and `libcurl` actually sent to the server.

Browser Developer Tools

We can often use the Web browser to capture the information we need to use in *R* to make a request. When mimicking how a Web browser works in a particular situation, we can actually use it make the request and see what it does, examining the request and the response. Each browser has some facilities for doing this. The Google Chrome browser gives us access to lots of details via its Developer Tools user interface. We bring this up within the browser display via the "View" menu, its "Developer" item and selecting "Developer Tools". Within this, we switch to the Network tab. After we make the request in the regular browser page, we can look at the different documents it retrieved and find the one we want, e.g., the entry `"usa-historical-cpi.php"` in Figure 8.1. We can examine various aspects of the request, including the header for the request and the response, the query parameters, and form data that were sent. We can also look at the cookies. This is a useful approach even when using *HTTPS*, which we typically cannot view, as the Web browser can decrypt the content and show us the raw text.

Monitoring Network Packets

Another approach is to use a packet-sniffing tool such as Wireshark or `tcpdump`. These are tools that

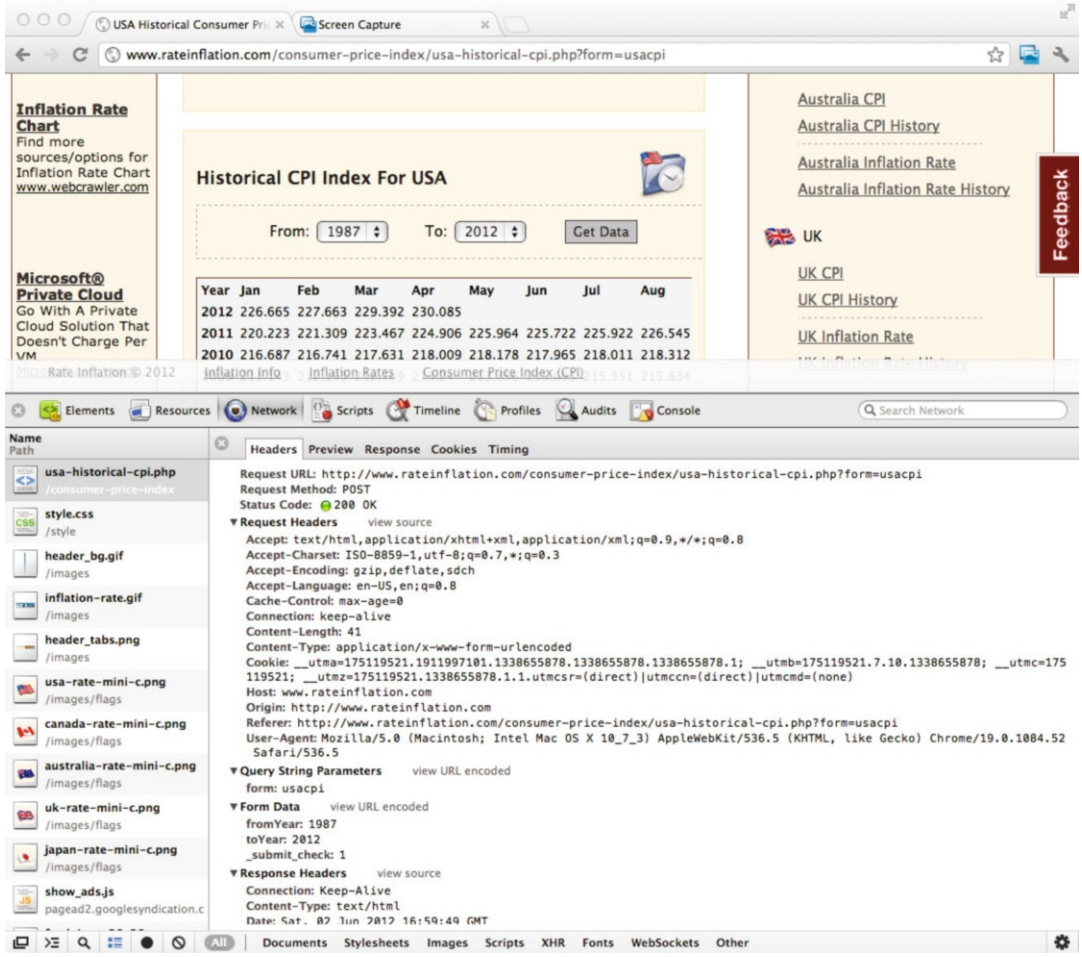


Figure 8.1: Using Chrome's Developer Tools to Explore a Web Request. We can use the Developer Tools user interface to examine the Web browser's view of a request. We changed the year in the "From" selection menu in the Web page and submit the form. In the Network panel of the Developer Tools, we select the document of interest (`use-historical-cpi.php`) and look at its Headers tab. This gives details of the browser's request for submitting the form. We see the form parameters and values in the "Form Data" segment. This is one mechanism to determine the names of the parameters in an *HTML* form.

allow us to collect all network packets that are sent or received on our computer. Assuming we have sufficient privileges, we start capturing packets using one of these tools and then make our request, and then stop the capturing. Then we can examine each of the relevant packets. Wireshark provides a graphical interface with which we can start and stop the collection of packets and also filter and examine the different packets of interest. This is an effective approach, but much lower level than using the browser's tools. It does allow us to export the actual contents of the packets exactly as they are sent, which is very useful. Unfortunately, when looking at *HTTPS* requests, we can see the information about the packets being sent, but not their contents as they are encrypted. This is where

the browser tools are significantly more convenient. They benefit from being the actual middle-party involved in the requests rather than a post-hoc third party.

debugfunction

Occasionally, requests will fail or return data that seems wrong or incomplete. We can, and should, enable `libcurl`'s diagnostic mode via the *verbose* option which causes it to output information to the console about what it is doing, showing the request and the response headers, and more. To get even more information, we can specify a callback function for the *debugfunction* option which `libcurl` will then use when it has information to share with us or report. `libcurl` calls the function we provided with three arguments: the message text; the "topic" of the notification which is an integer value with a name corresponding to one of `"TEXT"`, `"HEADER_IN"`, `"HEADER_OUT"`, `"DATA_IN"`, `"DATA_OUT"`, `"SSL_DATA_IN"` or `"SSL_DATA_OUT"`; and a reference to the curl object performing the request. The `"TEXT"` items are messages from `libcurl` that would usually appear on the console because of the *verbose* option. The `"HEADER_IN"` and `"DATA_IN"` are the contents that are received by `libcurl` from the server in the header and body, respectively. Note that for a given request, each of the header and/or the body may be broken into different segments or consecutive pieces, and these are passed to our debug function in separate but consecutive calls. In other words, we should not necessarily expect to get the entire contents of either the header or the body in a single call to our function, but may need to glue the pieces together across calls. The `"HEADER_OUT"` and `"DATA_OUT"` are what we send in our outgoing requests. We can therefore use the debug function to examine what is actually sent, e.g., in a **POST** or **PUT** request where the body would not ordinarily be visible.

As a reasonably simple example of how to use this feature, we can specify a basic function for the *debugfunction* option that just prints the topic and the number of bytes in the message. For example, we define the function as

```
dfun =
function(msg, topic, curl)
   cat(topic, ":", length(msg), "\n")
```

and use it in a call with

```
z = getURLContent(u, useragent = "R", verbose = TRUE,
                  debugfunction = dfun)
```

Note that we need to use `verbose = TRUE` to cause `libcurl` to call our debug function. Also, the usual output from `libcurl` due to the *verbose* option does not appear in the usual way when we specify a debug function. Instead, these messages are passed to our debug function as `"TEXT"` items. In this way, we can customize and collect the debugging information.

The RCurl package contains a function debugGatherer() that can be used as a callback function for the *debugfunction* option. This collects the message/content passed in each call to the debug function. At the end of the request, we can then examine these. Like dynCurlReader() and basicTextGatherer() which we discussed for the *writefunction* callback option, we call debugGatherer() and it returns a list containing three functions. We pass the `update` element as the value for the *debugfunction* option.

As an example of using the *debugfunction* option, we look at accessing a Twitter account from *R*. To be able to post tweets to a Twitter account from within *R*, we first have to use *OAuth* to get an access token. This is described in detail in Chapter 13, but the important part here is that, having specified private information in an `OAuth` object (`cred` in the code below), we call the handshake() function to obtain the access token. This actually makes a series of several requests to the server to obtain the access token. We can collect information about these with

```
dbg = debugGatherer()
cred = handshake(cred, debugfunction = dbg$update, verbose = TRUE)
info = dbg$value(ordered = TRUE)
```

The result in `info` is a *list* with 82 elements, and each element is the text passed to our `dbg$update` function. The names of the elements identify the topic or type of the message, i.e., TEXT, HEADER_IN, HEADER_OUT, etc. We can see that there were two requests (in the HEADER_OUT category)

```
table(names(info))
```

```
     DATA_IN     DATA_OUT    HEADER_IN   HEADER_OUT
           2            2           40            2
 SSL_DATA_IN SSL_DATA_OUT         TEXT
           5            4           27
```

The *debugfunction* option can be useful when we want to understand what `libcurl` is doing, and not just how we should be mimicking a request made by a browser or another application.

8.12 Curl Command Line Arguments and RCurl

We mentioned the command-line program `curl` at the beginning of this chapter. `curl` is based on `libcurl` so it provides ways to use many of `libcurl`'s facilities via command-line flags and arguments. We can set options such as the User-Agent or the *netrc.file* and specify files and other inputs for Web requests on the command line. Since RCurl and `curl` share the same underlying software (`libcurl`), they both can access many of the same features. There are many examples on the Web of using `curl` for making *HTTP* requests, so sometimes it can be useful to understand how to translate those to *R* commands using RCurl functions. In this section, we look at some examples and show the `curl` shell command and the equivalent *R* command.

Many of the `curl` command-line flags have two forms—the longer, but more suggestive name such as *–user-agent*, and the simpler to type *-A*. The longer forms correspond reasonably closely to the names of the curl options in RCurl. When translating a shell command invoking curl, it is good practice to consult the help page for curl and map the short forms of the command line options to their longer counterparts, at least mentally, and then map them to the corresponding RCurl option names.

For example, we can upload the contents of a file to a *URL* using `curl` with a command such as

```
curl -T filename -X PUT url
```

The *-X* identifies the type of *HTTP* request. The *-T* flag specifies the file that has the contents we want to send. We can do this in *R*, with

```
httpPUT(url, upload = TRUE, readdata = CFILE(filename)@ref,
        infilesize = file.info(filename)[1, "size"])
```

Example 8-12 Comparing Command Line and RCurl Requests for GlobalGiving

As another example, we send a request to a *REST* API provided by GlobalGiving [5] at `www.globalgiving.org`. It explicitly sets two *HTTP* header fields—*Accept* and *Content-Type*. We also set the *User-Agent* field. The command is

```
curl -H "Accept: application/xml"
   -H "Content-Type: application/xml"
   -A 'curl' -X GET
   url
```

where the `url` is `https://api.globalgiving.org/api/public/projectservice/all/projects/ids?api_key=YOUR_API_KEY` and you insert the appropriate key. With RCurl, we can use getForm() and pass the *URL* and parameter(s) as separate arguments rather than having to paste our key to the *URL*. We do this with `uGG` as `https://api.globalgiving.org/api/public/projectservice/all/projects/ids` and

```
getForm(uGG, api_key = key, useragent = "RCurl",
        httpheader = c(Accept = "application/xml",
                       "Content-Type" = "application/xml" ))
```

The *-A* in `curl` corresponds to the *useragent* option in RCurl and the two *-H* arguments are combined into a vector in *R* and set via the *httpheader* option.

Example 8-13 Posting a Tweet with a `curl` Command versus httpPOST()

We can post tweets programmatically via a **POST** request. Twitter [22] provides a Web interface to show you what the request would be as a `curl` command. (See `https://dev.twitter.com/apps` and the `OAuth` tab for your particular application.) For example, we post the tweet `"Here it is"` with

```
curl --request 'POST'
    'https://api.twitter.com/1/statuses/update'
    --data 'Here_it_is='
    --header 'Authorization: OAuth
                oauth_consumer_key=xx,
                oauth_nonce=45b55,
                oauth_signature_method=HMAC-SHA1,
                oauth_token=xxxx'
    --verbose
```

With RCurl, we can use

```
uT = 'https://api.twitter.com/1/statuses/update'
httpPOST(uT, postfields = "Here it is",
         httpheader =
             c(Authorization = 'OAuth oauth_consumer_key=xx,
               oauth_nonce=45b55,
               oauth_signature_method=HMAC-SHA1,
               oauth_token=xxxx' ),
          verbose = TRUE)
```

Note that RCurl encodes the text (`"Here it is"`) for us.

The *–request* flag is the same as the shorter *-X* flag for identifying the method of the request, e.g., **POST** or **PUT**.

One of the benefits of RCurl relative to `curl` is that we are working in *R* rather than the shell. This means we have a more flexible programming language and richer data structures. This makes it easier to do computations before and after the Web requests, i.e., we can can create the inputs from other computations, with data in *R*. We can also use the same curl handle across requests, which makes it easier to manage state and content (e.g., cookies) across the requests.

8.13 Summary of RCurl Functions

In this section, we provide brief descriptions of many of the functions in the RCurl package, including both high-level functions for making *HTTP* requests and intermediate-level functions for working with curl handles and options. Below are descriptions of the high-level functions.

getURLContent() Retrieve the content of a URL. This general function handles binary or text results.

getForm() Submit a form or invoke a *REST*ful API method using *HTTP*'s **GET** method.

postForm() Similar to getForm(), submit a form using **POST**. This function can be used for either `multipart/form-data` or `x-www-form-urlencoded` types of submissions.

getURLAsynchronous() Make many requests simultaneously. This can lead to a significant speedup relative to sequential requests and waiting for each request to complete before making the next.

httpGET(), httpPOST(), httpPUT(), httpDELETE(), httpHEAD() These functions correspond to the main *HTTP* methods or actions. They are equivalent to calling getURLContent() with the curl option *customrequest* set to a string giving the method name, e.g., `customrequest = "DELETE"`.

dynCurlReader(), basicTextGatherer() Create objects from which the response can be collected and read.

url.exists() Determine whether the specified *URL* exists and is accessible without an error. This is analogous to file.exists() in *R* and can be used to conditionally evaluate code based on whether the *URL* is available.

Note that we discourage people from using getURL() or getURI(). These do not handle binary data correctly; getURLContent() does.

Next we provide descriptions of functions in RCurl for creating and modifying curl handles and options.

getCurlHandle() Create a curl handle for use in one or more requests. It is beneficial to create a single handle to use in multiple requests with the same set of options, especially if the requests are to the same server.

getCurlMultiHandle() Create a curl handle that is capable of performing multiple asynchronous requests simultaneously.

dupCurlHandle() Make a copy of the given curl handle and its current settings so that it can be used independently of the original handle

curlSetOpt() Set one or more options in a curl handle without making a Web request. These settings will be used in subsequent requests using this handle.

curlPerform(), curlMultiPerform() The low-level functions for actually making one or more request.

listCurlOptions() Get the names of all the valid/recognized curl options.

getCurlInfo() Return information about the last request made on the given curl handle object. The information includes details such as the time the request took and the actual *URL* (after redirects).
curlVersion() Provide details about the version of `libcurl` we are using and the capabilities it has, such as the protocols it supports.

8.14 Further Reading

Although the topic of [10] is *REST*ful Web Services, the introductory chapters and appendices offer a good introduction to *HTTP*. Since `libcurl` is used in other programming languages, we can find many examples of using curl options in those languages.

The Web page that documents all of the `libcurl` options can be found at [11].

References

[1] Daniel Adler. `rdyncall`: Improved foreign function interface (FFI) and dynamic bindings to *C* libraries (e.g., *OpenGL*). `http://cran.r-project.org/package=rdyncall`, 2012. *R* package version 0.7.5.

[2] R. Fielding, J. Gettys, J. Mugul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol: *HTTP*/1.1. Worldwide Web Consortium, 1999. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[3] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.

[4] Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

[5] GlobalGiving Foundation. Globalgiving: Donate to projects around the world supporting disaster relief, education, health, women and children, and more. `http://www.globalgiving.org/`, 2012.

[6] David Gourley and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[7] Michael Nielsen. *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press, Princeton, NJ, 2012.

[8] Open *SSL* Project. Open *SSL*: Cryptography and *SSL*/TLS toolkit. `http://openssl.org`, 2011.

[9] Karthik Ram and Duncan Temple Lang. `rDrop`: Dropbox *R* interface. `https://github.com/karthikram/rDrop/`, 2012. *R* package version 0.3.

[10] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., Sebastopol, CA, 2007.

[11] Daniel Steinberg. curl_easy_setopt — Set options for a curl easy handle . `http://curl.haxx.se/libcurl/c/curl_easy_setopt.html`, 2012.

[12] Daniel Steinberg. libcurl: The multiprotocol file transfer library. `http://curl.haxx.se`, 2012.

[13] Duncan Temple Lang. RGCCTranslationUnit: *R* interface to `GCC` source code information. `http://www.omegahat.org/RGCCTranslationUnit`, 2009. *R* package version 0.4-0.

[14] Duncan Temple Lang. RGoogleTrends: Download Google Trends data. http://www.omegahat.org/RGoogleTrends, 2009. *R* package version 0.2-1.
[15] Duncan Temple Lang. RCIndex: *R* interface to the clang parser's *C* API. http://www.omegahat.org/RCIndex, 2010. *R* package version 0.2-0.
[16] Duncan Temple Lang. Rffi: Interface to libffi to dynamically invoke arbitrary compiled routines at run-time without compiled bindings. http://www.omegahat.org/Rffi, 2011. *R* package version 0.3-0.
[17] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.
[18] Duncan Temple Lang. Rcompression: In-memory decompression for GNU **zip** and bzip2 formats. http://www.omegahat.org/Rcompression, 2012. *R* package version 0.94-0.
[19] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.
[20] Duncan Temple Lang. RGoogleDocs: Primitive interface to Google Documents from *R*. http://www.omegahat.org/RGoogleDocs, 2012. *R* package version 0.7-0.
[21] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.
[22] Twitter, Inc. Twitter: A real-time information network. http://twitter.com/about, 2012.
[23] Dave Winer. XML-RPC specification. http://xmlrpc.scripting.com/spec.html, 1999.

Chapter 9
Scraping Data from *HTML* Forms

Abstract In Chapter 5, we saw how to scrape data from *HTML* pages. In this chapter, we focus on a variation of this where we get the Web page containing the data we want by submitting an *HTML* form. Rather than using a Web browser, we submit the form from *R*, providing inputs to parameterize the request from data in *R*. We use functionality in the RCurl package such as getForm() and postForm() to make the requests for the Web pages and then we scrape the data using the XML package and *XPath*. Since the forms are described in *HTML* documents, in many cases we can programmatically query the *HTML* form and learn about its parameters and their default and possible values. We can convert this information into an *R* function that acts as a proxy for the *HTML* form. The key ideas in this chapter are 1) to be able to get data programmatically via *HTML* forms as part of a reproducible workflow, and 2) to further automate the creation of the code that we use to get these data. These functions attempt to provide higher-level facilities and concepts relative to getForm() and postForm().

9.1 Introduction

There are a lot of interesting data that we can scrape directly from Web pages. There is a significant collection of data that we can also retrieve through Web pages that allow the reader to select the values of different parameters to query the data. These Web pages often use *HTML* forms to allow the user to select a setting from a menu, toggle a check box, select an option from a set of radio buttons, put free-form text into a text field, move a slider to set a value within an interval, or pick a date from a calendar widget. The new *HTML5* specification adds even more user interface (UI) controls that we can use, and there are even more provided by *JavaScript* libraries. These forms can be very convenient when, say, interactively purchasing an airline ticket or even retrieving data such as the Consumer Price Index (CPI) for a single time period. Unfortunately, when we want to access the data programmatically as part of an automated, reproducible, computational workflow, interacting with forms in a Web browser is less convenient. Increasingly, interesting data is being made available via *REST* APIs, which bypass the interactive *HTML* forms, either in addition to, or instead of, providing it via *HTML* forms. However, there are still many sources of data that we must access through Web forms. In this chapter, we present some reasonably simple and often effective techniques for getting data via Web forms. We discuss how to mimic a form submission from within *R* by knowing about the details of the *HTML* form. However, we will also discuss how we can use *R* to find these details and to actually generate a new *R* function that corresponds to submitting the form from a Web browser.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_9,

The RHTMLForms [2] package provides the basic functions for this, building on the foundations of the XML [7] and RCurl [8] packages.

To scrape data from an *HTML* page, we first parse the document using htmlParse(), and then find the nodes of interest with getNodeSet() and *XPath* expressions. We can then extract the data with node manipulation functions such as xmlName(), xmlValue(), xmlGetAttr(), xmlChildren(), and so on. The initial step is to get the document. For simple static documents, we can pass the *URL* to htmlParse(). If htmlParse() cannot download the document, e.g., if we need a password, cookie or to use *HTTPS*, then we may need to use functions in the RCurl package such as getURLContent(). If we need to submit a form to get the page with the data, we can use the functions getForm() and postForm() (or any of the intermediate-level functions in the RCurl package such as httpPOST() or httpPUT()). However, to submit a form, we need to know what data to submit and also how to submit the form. This involves knowing:

1. the names of the necessary parameters,
2. the set of possible and valid values for each of these parameters,
3. the *URL* to which we send the request, and
4. whether to **GET** or **POST** the request.

Once we have all of this information, we can send the request from *R* in the same way that a Web browser does. The server then returns a document. If it is a CSV document, for example, we can read the data with a combination of the functions read.csv() and textConnection(). If the content is *JSON*, we use the fromJSON() function. If the result is an *HTML* document, we can scrape its contents in the same way we did for a static page. If the content is *XML*, we can use fromXML(), xmlToList(), etc., or explicitly parse and process the document. We deal with other types such as images, **docx** or **xlsx** files, etc., in the appropriate manner.

The challenge in accessing the data is knowing the four aspects needed to submit the *HTML* form. How do we find these? For *HTML* forms, this information is very rarely documented explicitly on a Web page, unlike *REST* or SOAP methods, as the form is intended as the primary interface for users. Instead, the only source of this information is the *HTML* containing the form. There are three ways to get the information. For forms we access via a **GET** method, we can look at the location field in the browser after we submit the form. Alternatively, we can get more information using the "developer tools" available in most browsers. Finally, for forms that are not controlled dynamically by *JavaScript* code, we can get the information directly from the *HTML* which allows us to programmatically generate an *R* function to mimic the form. Even in cases where *JavaScript* is used, we can sometimes work directly from the *HTML* and adapt how we create the function. This allows us to minimize our manual involvement in creating the function and accessing the data underlying the form.

Example 9-1 The Google Search Form

Consider what appears to be the simplest and most well-known of *HTML* forms—Google's home page http://www.google.com. This is an *HTML* document and we can parse it and find the <*form*> element within it and its children. We can do this with htmlParse() and getNodeSet(), but getHTMLFormDescription() does this for us:

```
u = "http://www.google.com"
form = getHTMLFormDescription(u)[[1]]
form
```

```
HTML Form: http://www.google.com/search
q: [  ]
```

This suggests that there is a single parameter named "q" that we need to specify for our search. The "q" stands for query and this contains the query string. In fact, there are hidden parameters that identify the language and character encoding and some other information Google wants in order to process the query. We do not need to worry about these. We can convert this description of the form into an *R* function that will submit the query appropriately. We use createFunction() to do this, e.g.,

```
gsearch = createFunction(form)
```

We can now perform a search with

```
txt = gsearch("RCurl")
```

This returns the *HTML* page Google would return to our Web browser containing links to the top results for our search. We can then scrape this using tools, such as

```
getHTMLLinks(txt)
```

This general function returns all the links in the page. We can find the search result links by specifying a more specific *XPath* query

```
xpQ = "//a/@href[starts-with(., '/search?q=RCurl')]"
getHTMLLinks(txt, xpQuery = xpQ )

 [1] "/search?q=RCurl&hl=en&gbv=1&ie=UTF-8&
      prmd=ivns&source=lnms&tbm=isch&sa=X"
 [2] "/search?q=RCurl&hl=en&gbv=1&ie=UTF-8&
      prmd=ivns&source=lnms&tbm=vid&sa=X"
 ...
```

What does getHTMLFormDescription() do? It parses the *HTML* document and processes each <*form*> element. For each form, it determines the *URL* to which we should submit the form and how to submit it (i.e., **GET** or **POST**). It also processes each of the form's <*input*> elements that correspond to how a user sets a value. These can be simple free-form text fields, check boxes to toggle a setting, a collections of radio buttons to choose one of several settings/values, pull-down menus that also allow the user to chose one of several values, and now in *HTML5* we also have new <*input*> controls such as sliders, date-time/calendars, telephone numbers, color selector, etc. For each <*input*> element, getHTMLFormDescription() gets the name of the corresponding parameter in the form, the possible value for that element and whether it is the default value for that parameter. It then groups these together by parameter name so we have the set of all possible values for each parameter and the default value and the type of the parameter, e.g., single, exclusive value (radio button) or set of possible multiple values (check box), or a single string (text area). We can then use this HTMLFormDescription object to create an *R* function that

1. allows the caller to specify values for each of the (nonhidden) parameters;
2. uses the default values in the form as the default values for the *R* function's parameters;
3. checks the caller's values against the set of valid values before sending the request;
4. submits the request to the form's *URL* using the appropriate *HTTP* method (**GET** or **POST**);
5. returns the result from the Web server, optionally processing and converting it.

When matching the caller's values against the allowable values, we can also do partial matching, and we can match the user's input against both the set of possible values and the corresponding

human-readable values displayed in the form and mapped to the low-level values. This is much more information than we can get by observing the location field in the browser after making a single request from this form. For that, we only see one set of possible values for the parameters. However, getHTMLFormDescription() only reads the *HTML* content and so cannot determine what happens when the form is actually submitted. If the *HTML* page containing the <*form*> uses *JavaScript* code to change the form or override submitting the form, then the description from the *HTML* may not correspond to what is actually submitted or how it is submitted. We will see that we can still use getHTMLFormDescription() and createFunction() in some cases where *JavaScript* code changes the behavior of the form. Before we look at these and other examples of using these functions, we briefly describe other ways to determine the parameters and method for submitting a particular form so we can mimic it in *R*. Both involve using a Web browser to submit the form and see the details of that submission.

Programmatic *HTML* Form Submission

- To submit a form, we need to know: the names of the parameters, the set of possible values for these parameters, the *URL* to which we send the request, and whether the submission is a **GET** or **POST**. The getHTMLFormDescription() function obtains this information for us and places it in the returned `HTMLFormDescription` object.
- The createFunction() function uses the `HTMLFormDescription` object to generate an *R* function for form submission. The function it creates: allows the caller to specify values for each of the (nonhidden) parameters; uses the default values in the form as the default values for the *R* function's parameters; checks the caller's values against the set of valid values before sending the request; submits the request to the form's *URL* using the appropriate *HTTP* method (**GET** or **POST**); and returns the result from the Web server.
- In cases where the form uses *JavaScript* to override the form submission, the description of the form from the *HTML* may not correspond to what is actually submitted or how the form is submitted. In some of these cases, getHTMLFormDescription() and createFunction() can still be useful for automating the form submission, although the process may require additional tuning.

9.1.1 *GET* and *POST* Methods of Form Submission

If the <*form*> we submit uses the **GET** method, the entire request including the parameter names and values will appear in our Web browser's location bar/field. In the next example, we explore the *URL* resulting from a Google search made in different browsers.

When we visit the google.com home page in the Google Chrome browser and search for `"r scrape website"`, we see the following *URL* in the location bar (reformatted and shortened)

```
https://www.google.com/search?hl=en&biw=1343&bih=980
   &noj=1&gbv=2&sclient=psy-ab&q=r+scrape+website
   &oq=R+scrape+&gs_l=serp.3.0.0l2j0i30l2j0i1...
   136.1771.5j12.17.0...1c.BRq7FSGMSQg
```

We can see the main *URL* is `https://www.google.com/search` This is separated from the collection of name=value parameters by a `'?'` character. The parameter names are `hl`, `biw`, `bih`, `noj`, `gbv`, `sclient`, `q`, `oq`, and `gs_l`. It is more convenient to see these separately and we can do

this with the function getFormParams() in the RCurl package. This breaks a *URL* into its different parts and returns the **GET** parameters as a named character vector. For example,

```
names(getFormParams("https://www.google.com/search?hl=en&biw=1343&
                        ...1c.BRq7FSGMSQg"))
```

yields the parameter names

```
[1] "hl"      "biw"     "bih"     "noj"     "gbv"
[6] "sclient" "q"       "oq"      "gs_l"
```

If we compare these parameters to those we got via getHTMLFormDescription(), we see that they are quite different:

```
names(form$elements)
```

```
[1] "ie"     "hl"     "source" "q"      "gbv"
```

There are three in common — `q`, `hl` and `gbv`. This is related to the fact the Google Chrome page is using *JavaScript* code. If we temporarily disable this for the browser and then revisit the Google Web page and resubmit our search, the *URL* in the navigation bar is

```
https://www.google.com/search?hl=en&output=search&
    sclient=psy-ab&q=r+scrape+web&gbv=1&
    sei=5OcCUI22Le7q2wXgpY2DCw
```

If we do the same in the Safari browser, we get

```
http://www.google.com/search?q=r+scrape+web&
   btnG=Search&sclient=psy-ab&hl=en&site=&btnK=
```

The Web server (Google) can return different pages with different content for different browsers, especially if some use *JavaScript* and others do not. Again, bringing this into *R* with getFormParams() allows us to see the individual values more readily:

```
gparams = getFormParams(
 "https://www.google.com/search?hl=en&output=search&
  sclient=psy-ab&q=r+scrape+web&gbv=1&sei=5OcCUI22Le7q2wXgpY2DCw")
```

```
                    hl                output
                 "en"               "search"
              sclient                     q
            "psy-ab"       "r+scrape+web"
                gbv                   sei
                "1" "5OcCUI22Le7q2wXgpY2DCw"
```

We can use the result of getFormParams() to submit the form from within *R*. We can pass these inputs to getForm() via the *.params* parameter, having changed the query string in the element named `'q'`. For example, to search for *R* and *XML*, we can use

```
gparams["q"] = "R XML"
ans = getForm("https://www.google.com/search", .params = gparams)
hdoc = htmlParse(ans, asText = TRUE)
```

Again, we can scrape the results in whatever way we want. The key is that we have been able to capture the details of a particular call in a Web browser and adapt it and mimic the submission from within *R*.

Copying the full *URL* in a Web browser that results from submitting a form supplies the needed parameters when the form uses the **GET** method. For a **POST** submission, the name=value parameters are not appended to the *URL* and displayed in the browser's navigation bar. Instead, they are hidden in the body of the *HTTP* request. However, we can still discover them from within our Web browser. The details differ across browser, but most have "Developer tools" or extensions that allow us to examine many aspects of how the browser gets and displays a page. In the Google Chrome browser, we can display these tools within our tab for our page by selecting the View−>Developer−>Developer Tools−> menu cascade. This splits the tab into two regions as shown in Figure 9.1. When we then submit our search in the top panel of the tab, i.e., the regular Google home page, and details of the communication are shown in the Network tab of the developer tools panel. We can click on the element corresponding to the actual form submission in the Network tab and then see all of the information about the communication. This includes all of the details of the *HTTP* header including any cookies we sent to the server. We also get to see the parameters in any **GET** part of the query, and a second set of parameters for the **POST** submission. We can then copy these into *R* and use them as we did the `gparams` vector to mimic a request.

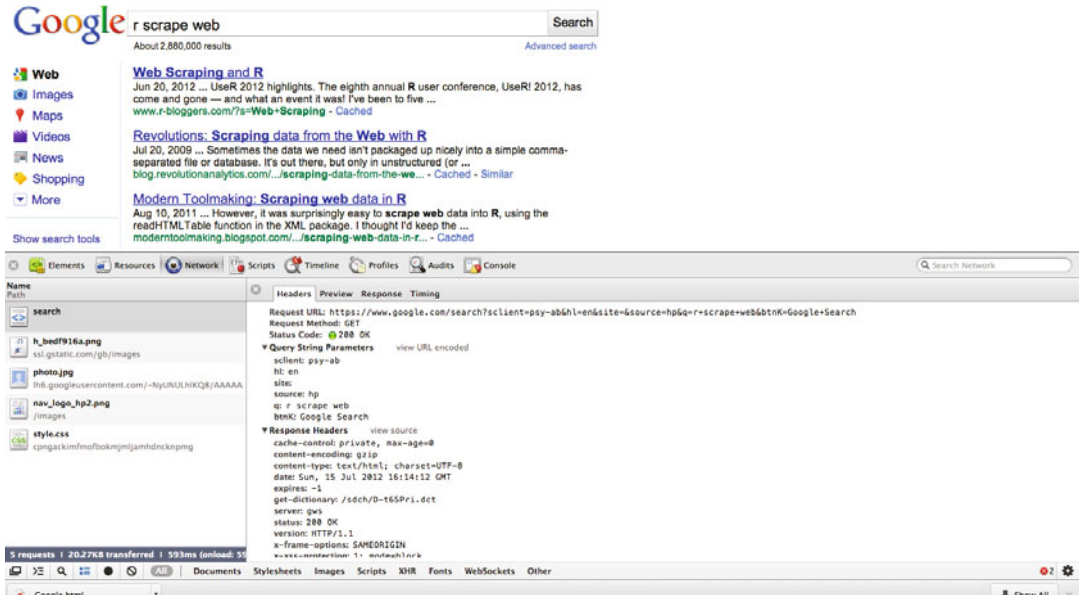


Figure 9.1: Developer Tools in a Web Browser. These are the tools in the Google Chrome Web browser for exploring all aspects of a page, from downloading the different parts of the document to the layout of the elements in the *HTML* page. From the Network tab, we can get all of the information about the *HTTP* request, including the client and server headers and the content of the submission. This allows us to get the parameters in a form submission, for either **GET** or **POST** submissions.

For the Firefox browser, LiveHTTPHeaders is a good way to explore the HTTP communication, except for with *HTTPS*.

Like the result of getHTMLFormDescription(), the result of getFormParams() is essentially a simple description of a form submission, limited to a particular submission rather than the class of all submissions. However, as we have seen, we can use it to make other requests. Getting the information from the browser and bringing this into *R* is a useful approach when trying to mimic a form when getHTMLFormDescription() is not adequate, e.g., when the form involves complex *JavaScript* code. Generally, if there is no *JavaScript* code, we can use getHTMLFormDescription() and createFunction() to mimic a function programmatically with no, or very little, manual intervention. If there is *JavaScript* code in the *HTML* form, often we can use a combination of getHTMLFormDescription(), the browser tools to examine a sample submission and getFormParams() (or directly) to merge additional parameters into the form description. For very complex forms, the browser tools and getFormParams() can get us a long way towards being able to emulate the form in *R*. In the remainder of this chapter, we explore some examples of these strategies.

9.2 Generating Customized Functions to Handle Form Submission

Example 9-2 Accessing Historical Consumer Price Index Data with a Form

In this example, we look at a simple **POST** form and how we can use getHTMLFormDescription() and createFunction() to create an *R* function to access data directly. The form is on the page `http://www.rateinflation.com/consumer-price-index/usa-historical-cpi.php` and appears as in Figure 9.2.

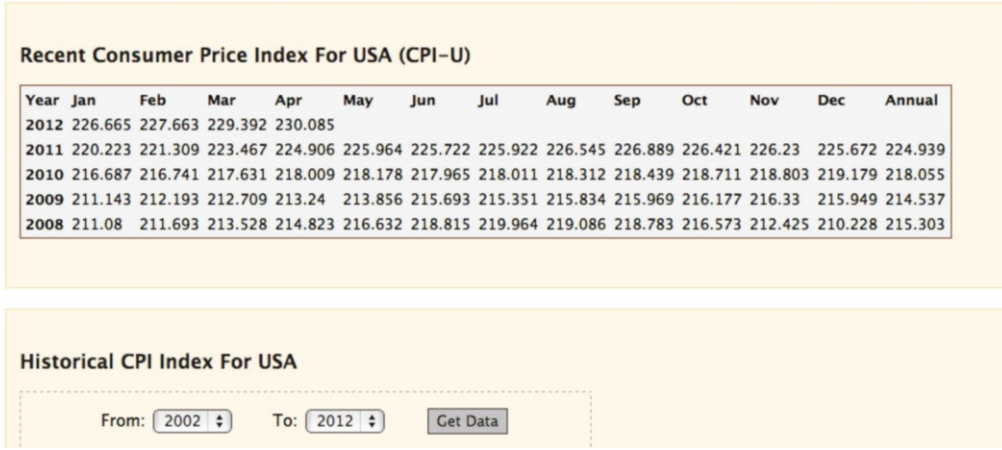


Figure 9.2: Form for Accessing Historical CPI Values. The form is below the table for the current period. The reader can select the starting and ending years for the data he/she wants to retrieve, and then click on the "Get Data" button. The Web server returns a new page containing an additional table with the CPI values for each month in that period. Retrieved from `http://www.rateinflation.com/consumer-price-index/usa-historical-cpi.php` in May 2012.

There are three forms on the page. The first two are for searching this particular site and for subscribing for updates. It is the third one that we want:

```
<form method="post"
 action="/consumer-price-index/...cpi.php?form=usacpi">
 <div align="center" style="border:1px dashed ...">
From:
  <select style="font-size:11px;color:#333333;..."
          name="fromYear">
   <option value='1913'>1913</option> ...
   <option value='2012'>2012</option>
  </select>
  <span style="padding-left:25px;">
To:
   <select style="font-size:11px;color:#333333;..."
          name="toYear">
    <option value='1913'>1913</option> ...
    <option value='2012' selected='selected'>2012</option>
   </select>
  </span>
  <span style="padding-left:25px;">
   <input style="font-size:11px;color:#333333;..."
          type="submit" value="Get Data"/>
  </span>
 </div>
 <input type="hidden" name="_submit_check" value="1"/>
</form>
```

This form has two select menus to choose the starting and ending year, a submit button, and a hidden input.

We can also get a description of the forms on the page with

```
forms = getHTMLFormDescription(uCPI)
```

where `uCPI` holds the *URL*, http://www.rateinflation.com/consumer-price-index/usa-historical-cpi.php. Since there are three forms, we can examine them in the *R* console to see which one we want by simply printing the `forms` object. The third one appears as:

```
forms[[3]]

HTML Form: http://www.rateinflation.com/consumer-price
-index/usa-historical-cpi.php?form=usacpi
fromYear: [ 2002 ]  1913, 1914, 1915, 1916, 1917, ...,
                    2007, 2008, 2009, 2010, 2011, 2012
toYear: [ 2012 ]  1913, 1914, 1915, 1916, 1917, ...,
                  2007, 2008, 2009, 2010, 2011, 2012
```

We see the two parameters corresponding to the From and To labels beside the option menus in the page. By default, getHTMLFormDescription() omits a button in a form. Typically these are not important in the request, but there are forms for which we need to explicitly include a value corresponding to the button variable in the *HTTP* request. For these cases, we can use

```
getHTMLFormDescription(u, dropButtons = FALSE)
```

In fact, there is a third and important parameter that is hidden. We can see this by displaying the entire collection of parameters with

```
print(forms[[3]], showHidden = TRUE)
```

which adds the extra line to the output

```
_submit_check: 1
```

We can also examine the `elements` of the form directly with

```
forms[[3]]$elements
```

The *_submit_check* parameter needs to be submitted in the form but the user/viewer does not need to, and cannot, set it or change its value. This is information that the Web server will use when processing the form request, but that is information shared from the initial form page and the Web server.

The output describing the form shows us the *URL* to which we submit the request. We can also find this and whether to **GET** or **POST** the request from the `formAttributes`, e.g.,

```
forms[[3]]$formAttributes
```

```
                                                method
                                                "post"
                                                action
"http://www.rateinflation.com/consumer-price-index/
           usa-historical-cpi.php?form=usacpi"
attr(,"class")
[1] "HTMLFormAttributes"
```

Since the form is so simple, we can compose the request ourselves or write a simple function to encapsulate it for reuse. However, we can generate the function with a simple call to createFunction(), such as

```
getCPI = createFunction(forms[[3]])
```

We can use getCPI() to get the data for entire time period that is available with

```
d = getCPI(fromYear = 1913, toYear = 2012)
```

A useful feature of the function we created is that it can identify incorrect inputs in *R* before sending the request to the Web server. For example, the call

```
getCPI(fromYear = 1900, toYear = 2012)
```

yields the error

```
Error in checkFormArgs.HTMLFormDescription(
           formDescription, args) :
 1  error(s) validating the form arguments
        list(message = "fromYear must take  a value
 in the set: '1913', '1914', ..., '2012'.  Not '1900'",
call = validateValue.HTMLSelectElement(desc[[i]], obj))
```

When we call the getCPI() function with acceptable values, the Web server returns an *HTML* document. This is intended to be displayed to the reader in a Web browser. However, we want to extract the data instead. With some exploration of a sample *HTML* document, we determine that the data are in the twelfth table in the document. We can use readHTMLTable() to get the values into a data frame with

```
cpi = readHTMLTable(htmlParse(d), which = 12, header = TRUE,
                    colClasses = rep("numeric", 14))
```

This gives us what we set out to fetch.

9.2.1 Adding a Function to Convert the Result

We may want to create a single function that submits the form and performs the conversion of the resulting page. It is quite straightforward to combine the two calls such as

```
getCPIData =
function(fromYear, toYear, ...)
{
  d = getCPI(fromYear = fromYear, toYear = toYear, ...)
  readHTMLTable(htmlParse(d), which = 12, header = TRUE,
                colClasses = rep("numeric", 14))
}
```

Note that we definitely want to allow the caller to specify additional arguments to the call to getCPIData() such as curl options. We add a ... for this. We also had to know the names of the parameters, and we did not specify default values, which we can get from the form or from our programmatically generated getCPI() function. While this is not difficult, it becomes tedious when the number of parameters is large.

We can use an alternative approach to create a wrapper function that combines the form request and converting the *HTML* document to a data frame. Instead of the function above, we first turn the conversion step into a function of its own, and then pass that as the default function to convert the document returned from the form submission. We define the converter function easily as

```
readCPI =
function(doc)
  readHTMLTable(htmlParse(doc), which = 12, header = TRUE,
                colClasses = rep("numeric", 14))
```

We can use this function in a particular call to our getCPI() function that we generated earlier via the *.reader* parameter, e.g.,

```
d = getCPI(1950, 2012, .reader = readCPI)
```

Alternatively, we can make this function the default value for the *.reader* parameter when we create the getCPI() function. To do this, we generate our getCPI() function with

```
getCPI = createFunction(forms[[3]], reader = readCPI)
```

The *reader* parameter is added as the default value for the *.reader* parameter in the generated function. (This parameter is called *.reader* to avoid possible conflicts with a form that has a parameter named *reader*.)

Our new getCPI() function now knows how to return the data frame and so

```
d = getCPI(fromYear = 1950, toYear = 2012)
```

yields a 63 by 14 data frame.

9.3 Supplying the curl Handle and Modifying the Form

The Freeway Performance Measurement System (PeMS) has a Web site that allows registered users to get information about California's freeways. It provides access to a vast amount of information, including data from each loop detector embedded in the road for each lane at different locations along highways in California. The data can be accessed at different time resolutions, even at 1-second intervals. In Example 8-9 (page 296), we saw how to access data programmatically for a given controller or station. We first have to login and obtain a cookie which we can then use to make subsequent requests. We can manually find the relevant information in the login form and then the second form (which yields the CSV data file). However, we can also use getHTMLFormDescription() to process each of the two forms. In this section, we demonstrate how to do this. Along the way, we show how to supply the curl handle when we make the request so that we can save state from the initial login form to the second form. Additionally, we modify the second form description to eliminate some of the parameters since they are used for intermediary computations and ignored by the Web server.

9.3.1 Saving State Across Submission of Different Forms

The login form is available on the PeMS home page `http://pems.dot.ca.gov`. We read the forms on that page with

```
library(RHTMLForms)
ff = getHTMLFormDescription("http://pems.dot.ca.gov")
```

There is only one form:

```
ff$ua


HTML Form: http://pems.dot.ca.gov/
username: [ Username ]
password:
```

We can create the *R* function that allows us to login using

```
pems.login = createFunction(ff[[1]])
```

We can now use that function to login.

The key point of logging into the PeMS Web site is to obtain the cookie we need in other requests. Since the cookie connects the login request with these later requests, we create a single `CURLHandle` object that we will use in each request. We create the handle and enable cookie management for that `CURLHandle` with

```
curl = getCurlHandle(cookiefile = "")
```

We use our pems.login() function by passing our login name and password for the site and the curl handle, e.g.,

```
invisible(pems.login(pemsInfo, names(pemsInfo), .curl = curl))
```

Here, we have put the password and login name in the variable `pemsInfo` in the form `c(login = "password")` so as to avoid exposing them directly in our code. We may also set the information in an *R* option via our `.Rprofile` script that is read at the start of each *R* session. (This should be

readable only by the owner and not by any other account on the system.) Note also that the order in which we specify the inputs is different from how they appear when displayed by the form summary in *R*. Specifically, the password is first and the user name is second. This is because the *username* has a default argument (`Username`) and when we created the *R* function it thought this default value might suffice.

Having logged in via the pems.login() function we created, the `CURLHandle` now contains the necessary cookie. We can then retrieve any of the pages from the PeMS site. By manually exploring the site, we know that the page containing the form from which we obtain the data of interest is `http://pems.dot.ca.gov/?dnode=Controller&content=detector_health&tab=dh_raw&controller_id=403159`. Here we have explicitly included the controller identifying the highway, location along it, and the direction of travel. We can change this to specify a different loop detector, but the important thing is that this is contained in the *URL*. The resulting *HTML* page and form looks something like what is displayed in Figure 9.3 Note that is not the page containing the data, but the page containing the form used to retrieve the data. We want to extract a description of the form on that page and create an *R* function to call it.

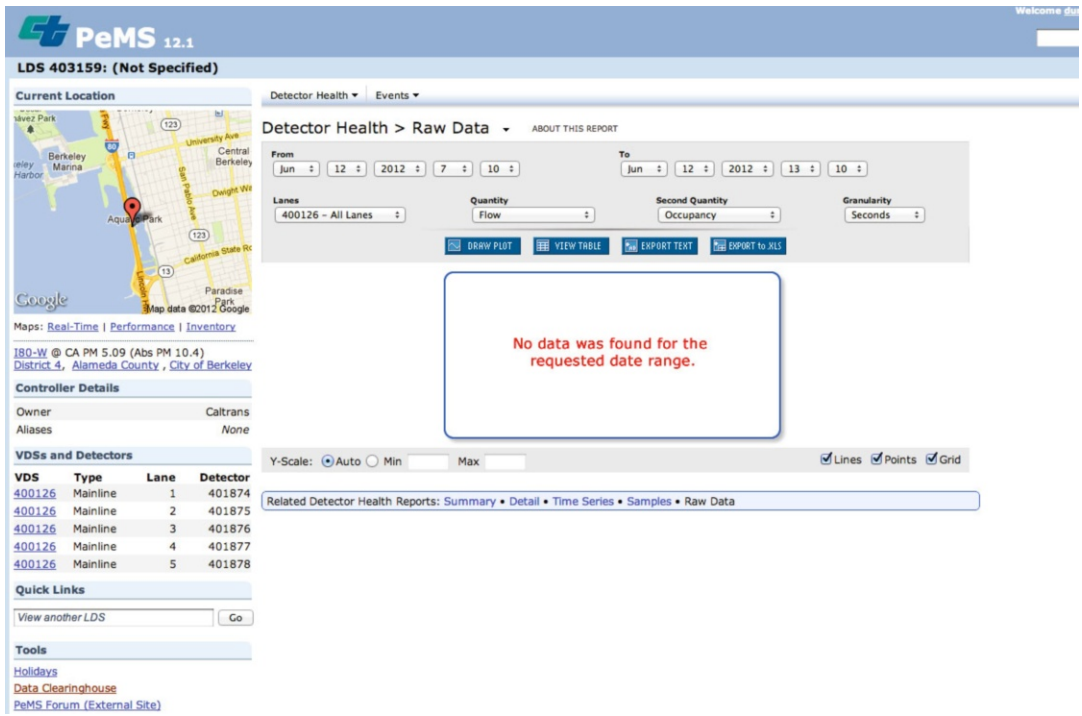


Figure 9.3: Form for Retrieving PeMS Traffic Data. The page on which we can access traffic data for a given period for the specified location on a specific highway. Retrieved from `http://pems.dot.ca.gov/?dnode=Controller&content=detector_health&tab=dh_raw&controller_id=403159` in June 2012.

In earlier examples of using getHTMLFormDescription(), we passed the *URL* that contains the form to the function. In this case, we cannot do that as the *URL* is not accessible unless we have logged in, and getHTMLFormDescription() does not know how to do this. Instead, we retrieve the

HTML document separately, and then pass the parsed document to getHTMLFormDescription(). We do this with

```
pems.doc = getForm("http://pems.dot.ca.gov/", dnode = "Controller",
                   content = "detector_health", tab = "dh_raw",
                   controller_id = 403159, curl = curl)
```

Note that we explicitly supply the `CURLHandle` object that we used to login so it contains the cookie we need to retrieve the document.

We parse this document and pass it to getHTMLFormDescription():

```
hdoc = htmlParse(pems.doc, asText = TRUE)
forms = getHTMLFormDescription(hdoc,
                               baseURL = "http://pems.dot.ca.gov/")
```

Since getHTMLFormDescription() is not reading the document directly from a *URL*, but must resolve relative file names in the forms, we supply the base *URL* for these relative paths. We can, instead, set the docName() on the parsed *HTML* document, e.g.,

```
docName(hdoc) = "http://pems.dot.ca.gov/"
forms = getHTMLFormDescription(hdoc)
```

The page has five forms; by looking at all of them, we can see that there is one named `"rpt_vars"` which is the one we want:

```
forms$rpt_vars
```

```
HTML Form: http://pems.dot.ca.gov/?dnode=Controller&
content=detector_health&tab=dh_raw&controller_id=403159
s_mm: [ 6 ]  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
             Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
             Sep, Oct, Nov, Dec
s_dd: [ 12 ] 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
            17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
s_yy: [ 2012 ]  2012, 2011, 2010, 2009, 2008, 2007, 2006, 2005,
                2004, 2003, 2002, 2001, 2000, 1999, 1998, 1997,
                1996, 1995, 1994, 1993
s_hh: [ 7 ]  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
            13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23
s_mi: [ 30 ]  0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55
e_mm: [ 6 ]  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
             Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
             Sep, Oct, Nov, Dec
e_dd: [ 12 ]  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
             13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
             23, 24, 25, 26, 27, 28, 29, 30
e_yy: [ 2012 ]  2012, 2011, 2010, 2009, 2008, 2007, 2006, 2005,
                2004, 2003, 2002, 2001, 2000, 1999, 1998, 1997,
                1996, 1995, 1994, 1993
e_hh: [ 13 ]  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
             13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23
e_mi: [ 30 ]  0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55
```

```
lanes: [ 400126-0 ]  400126-0, 400126-1, 400126-2,
                     400126-3, 400126-4, 400126-5,
                     400126 - All Lanes, 400126 - Lane 1,
                     400126 - Lane 2, 400126 - Lane 3,
                     400126 - Lane 4, 400126 - Lane 5
q: [ flow ]  flow, nflow, occ, gspeed, speed, Flow,
              Normalized Flow, Occupancy, G-Factor (22) Speed, Speed
q2: [  ]  , flow, nflow, occ, gspeed, speed, samples,
             --None--, Flow,  Normalized Flow,
             Occupancy, G-Factor (22) Speed, Speed, # Samples
gn: [ sec ]  sec, 5min, 15min, hour, Seconds,
              5 Minutes, 15 Minutes, Hour
```

There are also some hidden parameters that we can show with

```
print(forms$rpt_vars, showHidden = TRUE)
```

The additional output appears as

```
report_form: 1
dnode: Controller
content: detector_health
tab: dh_raw
export:
controller_id: 404250
s_time_id:
e_time_id:
```

In our request, we specify the from and to dates and times via the `s_mm`, `s_dd`, .. and `e_mm`, `e_dd`, ... parameters which represent the month, day, year, hour, minute of the start (s) and end (e) time periods.

We do want the data returned from our request to be a regular text, or a basic CSV file, rather than an **xls** file or an *HTML* document that displays the data as a table or a plot. To do this, we should change the (default) value for the hidden parameter *export* to the string `text`. We can change the form description with

```
forms$rpt_vars$elements[["export"]]$value = "text"
```

and now the export parameter will be sent in the form with a value of `text`. How do we know to do this and that `text` is the correct value? We can either examine the *HTML* page or we can look at a submitted request in our browser

In theory, we should be ready to call createFunction() to generate the function that makes the request to the PeMS site with values we provide. However, there is a problem. We can certainly create the function and make some calls, e.g.,

```
pems.getData = createFunction(forms$rpt_vars)
tt = pems.getData(s_mm = 6, s_dd = 1, s_hh = 0, s_mi = 0, e_mm = 6,
                  e_dd = 8, q = 'flow',  q2 = 'occ', gn = "5min",
                  .curl = curl, .url = "http://pems.dot.ca.gov/")
```

Unfortunately, this just returns an empty data set with no observations. We can use the *verbose* option for curl via the *.opts* arguments to examine the request and even the *debugfunction* to see the details of the exchange between *R* and the server.

The problem with the request is a little hard to diagnose, but one simple thing we can do is query whether it appears to use *JavaScript* code on any of the elements. The function usesJS() tells us this for a particular form:

```
usesJS(forms$rpt_vars)
```

This tells us the form does use *JavaScript*. What does this mean? Basically, the *JavaScript* code can modify anything in the form when the viewer changes any of the form elements, changing the values of other selections, updating *JavaScript* variables used when submitting the form, and so on. The *JavaScript* code can also take over submitting the request and do whatever it chooses in that step. So the behavior of the form is dynamic rather than a declarative description of what is to happen as specified in the contents of the *HTML*.

In this form, the *JavaScript* code updates the value of the hidden `s_time_id` and `e_time_id` form elements each time the user selects any of the corresponding time elements (e.g., month, day, year, hour, or minute). It does some computation that generates a number such as 1339495200 for the start time. This looks like a `POSIXct` value giving the number of seconds from 00:00, Jan 1, 1970 to represent a date and time. To emulate this form, we need to set the values of `s_time_id` and `e_time_id` based on the values in `s_yy`, `s_mm`, `s_dd`, `s_hh`, `s_mi`, and the corresponding elements for the end time. We need to do this within our generated function before the form is submitted. We can use the *.cleanArgs* parameter to do this. The purpose of this parameter is to modify the arguments to the form just before they are sent via getForm() or postForm(). This gives us an opportunity to change the values of the parameters, or add and/or remove parameters. As with the *reader* parameter to createFunction() and the corresponding *.reader* parameter in generated functions, we can specify a value for *cleanArgs* to use as the default function for all calls to our wrapper function, or we can specify a function via the *.cleanArgs* parameter in the generated function for a single, specific call. In other words, we can provide a default way to process the arguments and can also override it in individual calls.

In our form, we want to assemble the time information for each of the start and end times, and convert them to a number. We can write a function to do this which expects the year, month, etc., for that time. This creates a string and calls strptime() to create a `POSIXct` object and then returns its numeric value. We can make this quite general, but there is little value in this so we will define it to best suit our immediate needs. The function is

```
getTimeID =
function(vals)
  as.numeric(strptime(sprintf("%s/%s/%s %s:%s",
                              vals[["yy"]], vals[["mm"]],
                              vals[["dd"]], vals[["hh"]],
                              vals[["mi"]]),
                      "%Y/%m/%d %H:%M")) + 25200
```

Where does the 25200 come from? This seems to be the difference in the calculation in the number of seconds the *JavaScript* code makes from what we compute in *R*. We determined this by comparing different inputs.

For our form, we need to call the getTimeID() function for both the start and end times. So we can write a function to do this

```
pems.cleanArgs =
function(args, form)
{
  names = c("yy", "mm", "dd", "hh", "mi")
```

```
  args[["s_time_id"]] =
    getTimeID(structure(args[sprintf("s_%s", names)],
                                    names = names))
  args[["e_time_id"]] =
    getTimeID(structure(args[sprintf("e_%s", names)],
                                    names = names))
  args
}
```

This is the function that can modify the parameter values for the form submission. We can use this in a call to our existing pems.getData() function with

```
tt = pems.getData(s_mm = 6, s_dd=1, s_hh=0, s_mi = 0, e_mm = 6,
                  e_dd = 8, q = 'flow', q2 = 'occ', gn = "5min",
                  .curl = curl, .url = "http://pems.dot.ca.gov/",
                  .cleanArgs = pems.cleanArgs)
```

To make pems.cleanArgs() the default for *.cleanArgs* so that callers do not have to explicitly pass it as an input to pems.getData(), we can create pems.getData() with

```
pems.getData = createFunction(forms$rpt_vars,
                              cleanArgs = pems.cleanArgs)
```

Then we can call the function with

```
tt = pems.getData(s_mm = 6, s_dd=1, s_hh=0, s_mi = 0, e_mm = 6,
                  e_dd = 8, q = 'flow', q2 = 'occ', gn = "5min",
                  .curl = curl, .url = "http://pems.dot.ca.gov/")
```

9.3.2 Changing the Form Description

It turns out that the two sets of year, month, day, hour, and minute parameters, i.e., for the start and end times, are only used in the form to compute the time_id parameters. The Web server only looks at these `time_id` parameters when processing the request. We can omit these 10 other parameters entirely from our request and just use them to compute the values for the `s_time_id` and `e_time_id` form parameters. Therefore, we can create a different function that takes the times as `POSIXt` objects in *R* and submits those directly. We still need to support the other parameters in the form such as `q`, `q2`, `gn`, `lanes` and the hidden parameter `export`. We can, of course, write the function directly ourselves. Alternatively, we can modify the form description and use createFunction() to do this for us. For instance, we remove the 10 parameters for specifying the individual time components from the form description. These start with `s_` or `e_` and are followed by two characters such as `mm`, `yy`, `dd`, `mi`. So we can identify them with a regular expression and then remove them with

```
fm = forms$rpt_vars
i = grep("^[se]_[a-z][a-z]$", names(fm$elements))
fm$elements = fm$elements[- i ]
```

Then we make the `time_id` parameters visible, i.e., not hidden, so that they are added to the parameters of the function we generate. To do this, we just remove the `HTMLHiddenElement` class that is on each and leave them being of class `HTMLFormElement`. We do this with

```
ids = c("s_time_id", "e_time_id")
fm$elements[ids] = lapply(fm$elements[ids],
                          function(x)
                              structure(x, class = class(x)[-1]))
```

It is a good idea to see what the form description now contains:

```
print(fm, showHidden = TRUE)

HTML Form: http://pems.dot.ca.gov/?dnode=Controller&
content=detector_health&tab=dh_raw&controller_id=404250
report_form: 1
dnode: Controller
content: detector_health
tab: dh_raw
export: text
controller_id: 404250
s_time_id:
e_time_id:
lanes: [ 402826-0 ]  402826-0, 402826-1, 402826-2, 402826-3,
                402826-4, 402826-5, 402826 - All Lanes,
                402826 - Lane 1, 402826 - Lane 2, 402826 - Lane 3,
                402826 - Lane 4, 402826 - Lane 5
q: [ flow ] flow, nflow, occ, gspeed, speed, Flow, Normalized Flow,
             Occupancy, G-Factor (22) Speed, Speed
q2: [  ]  , flow, nflow, occ, gspeed, speed, samples,
           --None--, Flow, Normalized Flow, Occupancy,
           G-Factor (22) Speed,  Speed, # Samples
gn: [ sec ]  sec, 5min, 15min, hour, Seconds,
              5 Minutes, 15 Minutes, Hour
```

Now we can generate the function for this adapted form

```
pems1 = createFunction(fm)
```

We can then call this new function, giving it the start and end times as numbers, e.g.,

```
tt = pems1(s_time_id = as.numeric(as.POSIXct("2012/6/1 0:00")),
           e_time_id = as.numeric(as.POSIXct("2012/6/8 0:00")),
           q = 'flow', q2 = 'occ', gn = '5min',
           .url = "http://pems.dot.ca.gov/",
           .curl = curl, .opts = list(verbose = TRUE))
```

We should not have to convert the date–time values ourselves. Instead, we should have the function do this for us. There are many ways for us to add this to the new function. We can customize the way the code is generated or we can post-process the function. The latter seems complex to many, but is both powerful and reasonably simple once one understands the ideas. We can do this by changing the first expression in the function from

```
args = list(s_time_id = s_time_id, e_time_id = e_time_id,
            lanes = lanes, q = q, q2 = q2, gn = gn)
```

to

```
args = list(s_time_id = dateToNumber(s_time_id),
            e_time_id = dateToNumber(e_time_id),
            lanes = lanes, q = q, q2 = q2, gn = gn)
```

We have added a call to the function named dateToNumber() for each of the *time_id* parameters. We have to write this function, but can do this separately. To make these changes to the existing pems1() function, we use the commands

```
body(pems1)[[2]][[3]][["s_time_id"]] =
                                  quote(dateToNumber(s_time_id))
body(pems1)[[2]][[3]][["e_time_id"]] =
                                  quote(dateToNumber(e_time_id))
```

What does this do? First, `body(pems1)[[2]]` gives us the first expression in the body of the function pems1() (after the initial `'{'`). Therefore, `body(pems1)[[2]][[3]]` gives the expression

```
list(s_time_id = s_time_id, e_time_id = e_time_id,
     lanes = lanes, q = q, q2 = q2, gn = gn)
```

which is on the right-hand side of the assignment above. We then change the elements named `"s_time_id"` and `"e_time_id"`.

We can define dateToNumber() with

```
dateToNumber =
function(val)
{
   if(is.character(val) || is(val, "Date"))
     val = as.POSIXct(val)

   as.numeric(val)
}
```

We can now call our pems1() function more simply with

```
tt = pems1(s_time_id = "2012/6/1", e_time_id = "2012/6/8",
           q = 'flow', q2 = 'occ', gn = "5min", .curl = curl,
           .url = "http://pems.dot.ca.gov/",
           .opts = list(verbose = TRUE))
```

We can also specify the hour and minute in the time identifiers such as `"2012/6/1 15:31"`.

When we think about the steps and the exploration involved in extracting these data, this may seem like a lot of work, and we ask: Have getHTMLFormDescription() and createFunction() helped us? We think the answer is yes, because these functions are somewhat orthogonal to the exploration and discovery needed to access the data. There is no doubt at all that the entire process can be simpler if the data were available in a more direct manner. However, this is a good example of when an *HTML* form, mixed with a very small amount of *JavaScript* code, makes accessing data possible but nontrivial. Increasingly, organizations are making data available via *REST* APIs either in addition, or as an alternative, to *HTML* forms. However, we still have to deal with these more complex and obscure or arcane interfaces. Automated tools can help, if they allow customization at different points in the process. RHTMLForms allows us to change the description of a form before generating a function corresponding to a form. It also allows us to specify ways to process the parameters being sent to a form just before they are sent, either in a particular call or in all calls, by default. Similarly, the package allows us to customize how the results of the form submission are converted to an *R* object, both by default or in each call.

9.4 Forms and Elements that Use *JavaScript*

We finish this chapter by looking at another form that does not readily lend itself to getHTMLFormDescription(). The form is located at `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236` and allows us to get data about flight delays for every domestic flight within the United States. This information is provided by the Bureau of Transportation Statistics, specifically its Research and Innovative Technology Administration (RITA) [9] group. The Data Expo competition [1] in the 2009 Joint Statistical Meetings was centered around this data, organized by Hadley Wickham. It is an interesting data set, consisting of an entire population rather than a sample. We like to retrieve updates to the data and be able to download **zip** files for different months containing the different variables. Part of the Web page is shown in Figure 9.4.

We can use the "Prezipped File" check-box to retrieve a **zip** file of all the variables. Unfortunately, this is not always available for recent time periods so this request can fail. However, the data are available from the underlying database, just not as a prezipped file. To fetch the data in these cases, we would potentially have to select all 109 check-boxes identifying the variables of interest. That is a lot and it is easy to miss one. Instead, we want to be able to check all of them programmatically. We can do this within our Web browser using some *JavaScript* code to find the relevant check-boxes and toggle them. However, we want to do this in *R* rather than in a Web browser. One reason for this is because each query within the same browser session is downloaded to the same basic file name and contains a CSV file with the same name. Therefore, if we download the data for two or more months, we have to be careful not to overwrite the data. We want to loop over each year and month, download the data for that time period, extract the CSV file, read the data into *R* as a data frame, and serialize it.

As with the PeMS form that uses *JavaScript* to create the start and end time identifiers, the RITA form uses *JavaScript* to create important parts of the query before it submits it. Each variable we select (via a check-box) to include in the resulting data set is added to an *SQL* query. How do we know this? We examine a sample query using our Web browser. For example, using Google's Chrome Web browser, we can use the "Developer Tools" option to explore the Network communication tab. Within this, we see the parameters as part of the **GET** part of the *URL*

```
Query String Parameters
Table_ID:236
Has_Group:3
Is_Zipped:0
```

and the other important parameters sent as the **POST** part of the request:

```
Form Data
UserTableName:On_Time_Performance
DBShortName:
RawDataTable:T_ONTIME
sqlstr: SELECT YEAR, QUARTER FROM T_ONTIME
        WHERE Month =1 AND YEAR=2012
varlist:YEAR,QUARTER
grouplist:
suml:
sumRegion:
filter1:title=
filter2:title=
geo:(unable to decode value)
```

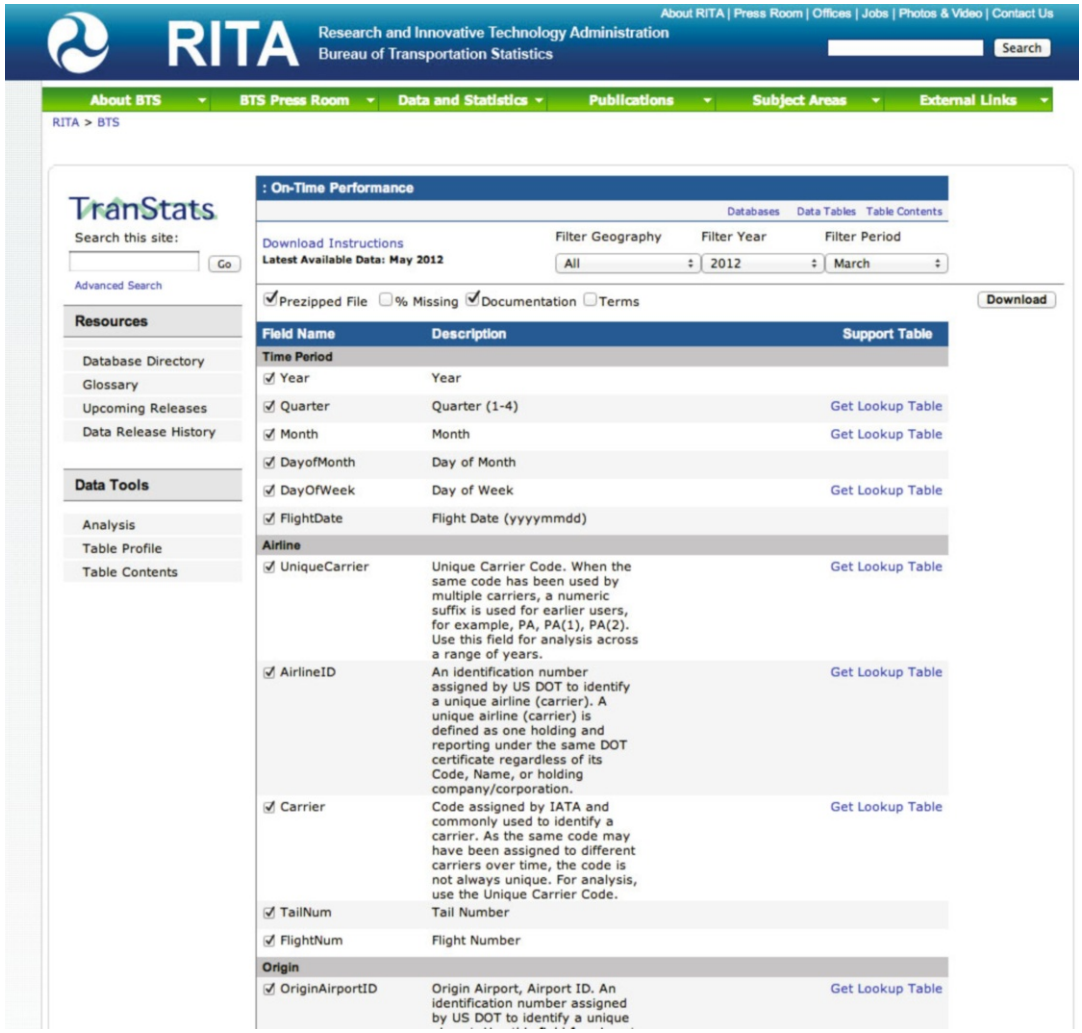


Figure 9.4: Form for Accessing RITA Airline Delay Data. This is the form that we can use to download data about each domestic airline flight within the United States and find out about its "on time" characteristics. We can select the "Prezipped File" option or select individual variables via the check boxes in the second part of the page. The values from these check-boxes are not actually part of the *HTTP* request when submitting the form. Instead, they are used locally within *JavaScript* code within the page to create an *SQL* query which is part of the *HTTP* request. Retrieved from `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236` in April 2012.

```
time:January
timename:Month
GEOGRAPHY:All
XYEAR:2012
FREQUENCY:1
VarName:YEAR
VarDesc:Year
VarType:Num
VarName:QUARTER
VarDesc:Quarter
VarType:Num
...
```

The key elements (that change) are `sqlstr` and `varlist`. We want to construct these strings based on the variables of interest. Here we have only selected the first two variables, `YEAR` and `QUARTER`. In addition to the initial parameters in the output, each variable is described in the **POST** submission as a triple of `VarName`, `VarDesc` and `VarType`.

Our strategy for being able to submit the query from *R* is to find the basic elements of the form and then to programmatically create and fill in the value of the `varlist` and `sqlstr` parameters in that query.

We start by reading the description of the form with

```
library(RHTMLForms)
fm = getHTMLFormDescription(uBTS)
```

There are three forms on the page and we want the one named `"form1"`, so we assign this to `fm = fm$form1`. We can see the elements within the form with

```
print(fm, showHidden = TRUE)

HTML Form: http://www.transtats.bts.gov//
   DownLoad_Table.asp?
   Table_ID=236&Has_Group=3&Is_Zipped=0
UserTableName: On_Time_Performance
DBShortName:
RawDataTable: T_ONTIME
sqlstr:
varlist:
grouplist:
suml:
sumRegion:
filter1: title=
filter2: title=
geo:
time:
timename: Month
GEOGRAPHY: [ All ]  All, Alabama, Alaska, Arizona, Arkansas,
     California, Colorado, Connecticut, Delaware, Florida, Georgia,
     Hawaii, Idaho, Illinois, Indiana, Iowa, Kansas, Kentucky,
     Louisiana, Maine, Maryland, Massachusetts, Michigan,
```

```
     Minnesota, Mississippi, Missouri, Montana, Nebraska, Nevada,
     New, North, Ohio, Oklahoma, Oregon, Pennsylvania, Puerto,
     Rhode, South, Tennessee, Texas, U.S., Utah, Vermont, Virginia,
     Washington, West, Wisconsin, Wyoming, All, New Hampshire,
     New Jersey, New Mexico, New York, North Carolina,
     North Dakota, Puerto Rico, Rhode Island, South Carolina,
     South Dakota, U.S. Pacific Trust Territories and Possessions,
     U.S. Virgin Islands, West Virginia
XYEAR: [ 2012 ]  1987, 1988, 1989, 1990, 1991, 1992, 1993, 1994,
       1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003,
       2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012
FREQUENCY: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    January, February,  March, April, May, June, July,
    August, September, October, November, December
```

We will change the default values for `varlist` and `sqlstr`.

The next step is to find the names of all the variables from the *HTML* document. Somewhat strangely, these are not elements within a <*form*> node. Instead, they are stand-alone <*input*> elements within a <*table*> such as

```
<input type="checkbox" name="VarName" id="VarName"
       title="Year" value="YEAR" onclick="Try4(this);" />
```

We can find these nodes and the *value* attributes to find the entire set of variable names. We can do this with

```
doc = htmlParse(uBTS)
varNames = xpathSApply(doc, "//input[@name='VarName']",
                           xmlGetAttr, "value")
```

We can construct the values for varlist and sqlstr with

```
vars = paste(varNames, collapse = ",")
sqlQuery = sprintf("SELECT %s FROM T_ONTIME
                     WHERE Month = %%d AND YEAR = %%d",
                   vars)
```

Note that we have left the value for the month and the year to be filled in later. These will come from the call to our function that we generate from the form. Next we add these back to the description of the form with

```
fm$elements[["varlist"]] =
  structure(list(name = "varlist", value = vars),
            class = c("HTMLHiddenElement", "HTMLFormElement"))
fm$elements[["sqlstr"]] =
  structure(list(name = "sqlstr", value = sqlQuery),
            class = c("HTMLHiddenElement", "HTMLFormElement"))
```

We can also use the simpler

```
fm$elements[["varlist"]]$value = vars
fm$elements[["sqlstr"]]$value = sqlQuery
```

since each element already existed in the form description. We used the longer form to illustrate how to introduce new hidden elements.

We are now ready to create the function corresponding to the form with createFunction(). We need to fill in the actual values for the year and month in the value of the *sqlstr* parameter. We can do this via the *cleanArgs* parameter for createFunction(). The following function does the job for us

```
function(args)
{
  args[["sqlstr"]] = sprintf(args[["sqlstr"]],
              as.integer(args$FREQUENCY), as.integer(args$XYEAR))
  args
}
```

We can create the actual function to retrieve the data with

```
rita =
 createFunction(fm, cleanArgs =
               function(args, formDescription) {
                 args[["sqlstr"]] = sprintf(args[["sqlstr"]],
                                            as.integer(args$FREQUENCY),
                                            as.integer(args$XYEAR))
                 args
               })
```

That gives us our function to mimic the form. However, we need to add more details when we call it which we can do via the curl options parameter *.opts*.

Firstly, we need to ensure that we follow redirects with the *followlocation* option. Secondly, we can write the result returned by the Web server to a file rather than work with it directly in memory. To do this, we will use the CFILE() function to create an *C*-level *FILE* reference. To have `libcurl` write to that file, we need to specify values for *writefunction* and *writedata*.

```
library(RCurl)
out = CFILE("rita.zip", "wb")
curlOpts = list(followlocation = TRUE, verbose = TRUE,
                writefunction = NULL, writedata = out@ref)
rita(XYEAR = 2012, FREQUENCY = 2, .opts = curlOpts)
close(out)
```

Note that it is essential to explicitly close the `CFILE` object `out` (or remove it and call the garbage collector to close it). This causes the contents to be flushed. Without this, the file is likely to be incomplete on disk.

We now have the **zip** file. We can then extract the CSV file and read it into *R*.

In this example, we have done a lot of work aside from the calls to getHTMLFormDescription() and createFunction() to extract the data. It is reasonable to ask whether these two functions have helped or whether we may have written the code from scratch more simply. When evaluating this, we do want to err in favor of reusing code across different problems. Such functions are more likely to be better tested and more robust. Any changes we make to them can also be reused in future tasks. This is especially important when dealing with Web pages as the structure, layout and details of the page and its forms may change over time merely for cosmetic/aesthetic effects. Using automated tools often insulates us from these minor changes. Accordingly, we think it is valuable to put the effort into reusing, customizing and enhancing the two general functions rather than pursuing one-off, page-specific solutions.

getHTMLFormDescription() Read an *HTML* document (given as a *URL*, local file, or an already parsed document) and extract a description of each `<form>` element in the document. Each description contains the *URL* to where the form should be submitted; the method for submitting the form (**GET** or **POST**); and a list of the parameters and their default values and acceptable values. Each parameter has a class that identifies the nature of the form element corresponding to that parameter.

createFunction() Create an *R* function from a description of a form. This function can be used to submit the corresponding form, passing it values from *R* that are checked by the function and sent as part of the form. The *reader* parameter of createFunction() and the *.reader* parameter in the created function allow us to control how the result is processed in *R*, e.g., converting the result to a data frame or to a list of links. The *cleanArgs* and *.cleanArgs* parameters allow us to process the form arguments before they are actually sent to the Web server, giving the opportunity to modify and add to these arguments.

getFormParams() Extract the collection of form parameters in the name=value format in a *URL*. We can cut and paste the *URL* into *R* and pass it to getFormParams() to get a description of the specific call.

9.5 Further Reading

The specifications for *HTML* forms can be found at [5] for *HTML*4 and at [3] for *HTML5*. In addition, Chapter 9 of [4] and Chapter 3 of [6] contain descriptions of form elements.

References

[1] ASA Sections on Statistical Computing and Graphics. Data Expo 09: Airline on-time performance. http://stat-computing.org/dataexpo/2009/, 2009.

[2] Sandrine Dudoit, Sunduz Keles, and Duncan Temple Lang. RHTMLForms: Programmatically create *R* functions corresponding to Web/*HTML* forms. http://www.omegahat.org/RHTMLForms, 2012. *R* package version 0.6-0.

[3] Ian Hickson. *HTML5*: A vocabulary and associated APIs for *HTML* and *XHTML*. Worldwide Web Consortium, 2011. http://www.w3.org/TR/html5/.

[4] Mark Pilgrim. *HTML5: Up and Running*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[5] David Raggett. *HTML* 4.01 specification. Worldwide Web Consortium, 1999. http://www.w3.org/TR/html401.

[6] Christopher Schmitt and Kyle Simpson. *HTML5 Cookbook*. O'Reilly Media, Inc., Sebastopol, CA, 2011.

[7] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[8] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.

[9] US Department of Transportation. Research and innovative technology administration. http://www.rita.dot.gov/, 2011.

Chapter 10
REST-based Web Services

Abstract Web services are an important development in making data available to clients in a programmatic manner. Rather than displaying results for humans to view on the Web, we can make requests to a Web service in order to get the resulting data directly and then consume it in our applications immediately. An increasingly common architecture for Web services is termed *REST* and exploits the concept of URLs as resources. We can make general *HTTP* requests to retrieve, update, modify, and even create resources. The results (and sometimes the inputs) are often formatted as *JSON* or *XML*. In this chapter, we look at how we can access *REST* services from within *R*, mapping the documentation for the services into requests, converting the results, and developing *R* functions to provide higher-level interfaces that hide the details of the *HTTP* and conversion. *REST* is becoming the dominant mechanism used for Web services which are also becoming very common. *REST* is not just used for Web services, but is also increasingly used to communicate with regular applications such as *NoSQL* databases and text search engines and we explore these also in this chapter.

10.1 Introduction

Scraping data from *HTML* pages, either directly or via *HTML* forms, can be very effective but is always somewhat ad hoc. For example, see Chapter 5. We need to navigate the large amount of formatting information that is used to display the data (and other content) in a Web browser, but which is entirely superfluous for our needs. If the layout of the page changes slightly, we may have to change our code to extract the data. We would much prefer to make requests and have just the data returned directly, without the extraneous formatting. Web services do precisely this. We make *HTTP* requests, passing any inputs, and the result is the data, typically in *XML*, *JSON*, or plain text (e.g., CSV) format or some binary format such as a PNG image. Web services bring the Web beyond links from one page to another, and beyond pages for humans to read. Instead, they allow us to exchange structured data across the Web and provide a coherent way for our applications to request data in rich, programmatic ways.

Web services provide access to a range of different data and functionality. Some services allow us to dynamically access public data that is updated regularly. Other services give us access to social network facilities such as Facebook, Twitter, LastFM, and so on. Others provide access to functionality in the cloud such as creating, editing, and viewing documents (e.g., Google Docs), data storage (e.g., Google Drive, Amazon S3), and creating virtual machines and running computationally intensive tasks on remote machines.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_10,

Web services are also referred to as APIs (Application Programming Interfaces) and are an increasingly important tool for data scientists. We will focus on how to use these as consumers, not as creators or providers. There are two primary technologies used in Web services: *REST* and *SOAP*. Both use *HTTP* to make requests and return data. *SOAP* came first, and *REST* is a response to make Web services simpler, and also provide a different perspective that makes them more in line with the philosophy of the Web and *HTTP*. The Web site `ProgrammableWeb.com` maintains a directory of Web service APIs. While it is not clear how representative of all Web services this collection of 7,600 APIs is, it gives a reasonable indication of the prevalence of different technologies. According to a blog post on the site [7], in March 2012, there were 195 "science" APIs. In this category, there were an almost equal number of *REST* and *SOAP* APIs. However, when we look at all categories of APIs, there is an increasing number of *REST* APIs, accounting for over 75% of all APIs. This trend will continue and *REST* is likely to become even more dominant.

While *SOAP* has some advantages, *REST* has one important but low-level advantage that motivated its design. *SOAP* focuses on calling methods. For *REST*, we think of resources. We can call a method in *REST*, but we are sending a request to a resource. The request may simply query the resource, but we may also update it, replace it, or even delete the resource. The particular operation in the *HTTP* request indicates which of these is happening. *SOAP* does not reveal this information. In some cases, this encapsulation offered by *SOAP* might be a good thing. However, since we are using *HTTP* operations, it can cause some issues. Can the results be cached, e.g., to avoid repeating the requests? *SOAP* methods can include information about this in the response, but typically do not. Can we "safely" repeat a call without corrupting the resource or inappropriately changing its state? For example, consider a request to transfer money from one bank account to another. If we repeat the same request, will this repeat the transaction or be recognized as a repeat of the same initial request. *SOAP* is not uniquely tied to *HTTP*, so this issue is not surprising. Again, the two approaches have different advantages and disadvantages, histories and (mis)perceptions. We discuss these below.

In short, while *REST* and *JSON* are simpler than *SOAP* and *XML* in some ways, this supposed simplicity is a little more complex than some people suggest, especially when one uses *XML* for many different purposes. One technology may be inherently more complex than another, but using it may be simpler for the end-user or some group of end-users. There is no doubt that *REST* is growing in popularity, and at a tremendous rate. However, while we can debate whether *REST* or *SOAP* is better, the discussion is largely academic for consumers of APIs. Often, we will want to use an API that is either *REST* or *SOAP*-based and we will have no choice. For this reason, we as consumers need to know about both approaches.

10.1.1 Key Concepts

What are the essential concepts underlying *REST*? The key notion in *REST* is a resource. The idea is that we can use a *URL* as the address of a given resource. A resource may be a static document such as an *HTML* page. More interestingly, the content may be dynamic so that when we request the resource, the response is potentially different each time, e.g., the current weather details for an observation station or sensor. The resource may be a data set in a file or a table within a database, to which new values are being added asynchronously. A resource may be an entire database or a collection of documents. Here we would access the individual elements of either with subpaths in the *URL* or parameters in the request. We may consider a Twitter or Facebook account as a resource. Twitter also has a *URL* corresponding to the resource of the list of the most recent tweets. Similarly,

Mendeley[1] has a *URL* for accessing the most referenced authors across all disciplines. A *URL* that returns stock quotes for a particular business may be a resource of its own, or we may have a single resource that provides stock quotes for many companies. When querying the latter, we would specify which stock we want.

Another important concept of *REST* is that it is stateless. This requires a little explanation. Each resource has a current state. In that sense, *REST* is definitely stateful. The statelessness refers to how information is carried across two or more requests to the resource by the same client. Many Web sites will use a cookie to identify the same session across requests. The Web server stores information about the user and each requests he or she makes. It then uses the cookie to retrieve this information for each request and customize the response. In this way, the server uses state to remember the user and what has happened in previous requests. For *REST*, all of the caller-specific information needed in each request must be specified in that particular request. The Web service does not store information across requests. In this regard, *REST* is stateless, having no state for the caller across requests.

REST stands for Representational State Transfer. We transfer representations of these resources between clients and the server. A resource is a more abstract entity than the actual representation which may be, for example, an *XML* or *JSON* document. Alternatively, it may be an *HTML* document, spreadsheet, or image, or any specific format. The actual representation provides the state of the resource.

REST is tightly coupled to *HTTP*, and indeed the creator of the *REST* architecture, Roy Fielding, was involved in the definition of *HTTP*. However, *REST* is not restricted only to the Web and Web servers. We can, and do, use *REST* to connect to applications such as *NoSQL* databases (e.g., CouchDB) and text search engines (e.g., *ElasticSearch*). The same *HTTP* operations such as **GET**, **POST**, **PUT**, and **DELETE** allow us to query and manipulate resources in applications.

If we were to create *REST* Web services, we should think seriously about how to structure and expose the resources, and what *HTTP* operations to permit so that the service follows the best practices for *REST*. However, when we are consumers of existing Web services, we do not necessarily need to think in terms of resources and the *REST* philosophy, although it can be helpful. Instead, the important skills and the basic approach involve

1. reading the API,
2. finding the important methods and understanding the inputs for the requests,
3. determining the format (e.g., *JSON* or *XML*) and structure of the results.

The *REST* paradigm is both simple and flexible. However, it affords creators of *REST*-based Web services a great deal of freedom in how they structure the services. One can represent resources in different ways, e.g., as subpaths within a base *URL* or via parameters in requests to a single *URL*. In many cases, the choice of expected *HTTP* operations (**GET**, **POST**, etc.) is clear, but in some scenarios, it can be confusing or ambiguous. This flexibility and somewhat vague connection between the paradigm and the implementation can make it more complex to easily understand a particular *REST* Web service. This is where it is beneficial a) to have experience in mapping the collection of resources and *HTTP* requests to *R* functions, and b) to be able to automate the generation of the interface (e.g., with *WADL*/*WSDL* documents) which we discuss later in the chapter. Because of the flexibility and variations across *REST* services, we explore consuming *REST* services in this chapter by looking at many different examples. For the most part, Web services are very simple to access from *R* using the RCurl [15], XML [13], and RJSONIO [11] packages.

[1] Mendeley is a Web site for managing citations, *PDF* documents, and notes on papers for academics/researchers. It also supports community sharing of these resources.

While there is a lot of variability in the details of *REST* services, there are some common concepts, approaches, and idioms. One important aspect is authentication and authorization. Some services allow direct requests without the caller having to provide identification. This is anonymous access. Most services, however, will require some form of authentication. This may be as simple as sending a unique key that identifies the caller. This is used when the data being accessed are public and not specific to the caller. Therefore the server does not need to restrict access to the data based on who the caller is. However, the owners of the service may want to monitor who is accessing the data and perhaps provide a differential service to some clients. Many sites use the key to put a limit on the number of requests by a caller in a given interval. For example, Google limits the number of requests by each caller to its *URL* shortener API to one million per day. A third form of authentication involves accessing private data. In some cases, this involves logging into the service at the beginning of each session and obtaining a session-specific token for that login. We then send this token in all subsequent requests to identify ourselves and our credentials to access the resources. Google Docs is an example of this (currently). Many services require even stronger authentication that involves digitally signing each request so that the server can verify that each one has come from the appropriate user. This typically also involves the user, the server, and the application accessing the data to participate in a three-way login to grant the application authorization to the user's private data. This typically involves *OAuth* (version 1.0 or 2.0). We present examples in this chapter for each of these categories: a) anonymous, b) public data with a caller authentication token, c) session login to access private data, and d) three-way *OAuth*-based authentication and access and signing of each request. We mention examples of *OAuth* in this chapter, but we provide a much more comprehensive description and set of examples in Chapter 13.

As we mentioned, *REST* builds on the concepts of *HTTP* that we saw in Chapter 8. The hope is that readers are reasonably familiar with how to use the RCurl package and functions such as getURLContent(), getForm() postForm(), httpPUT(), httpDELETE() and so on, and are able to specify different options for the *HTTP* requests. *REST* services typically use *JSON* or *XML* as the format for the results (and nontrivial inputs). So processing the results will build on the fromJSON() and xmlParse() and getNodeSet() functions.

10.1.2 A Brief Contrast of REST and SOAP

Before we examine either *REST* or *SOAP* in any detail, it is useful to make a very high-level comparison in terms of accessing Web services from with *R*. We will explore *SOAP* in Chapter 12. For now, it suffices to know that *SOAP* uses *XML* to format both the request and response and the data transferred in each, whereas *REST* typically uses *either XML* or *JSON*. Increasingly, Web services are using *REST* and *JSON*. However, many *REST* APIs also use or support *XML*.

While the 'S' in *SOAP* stands for "simple", the details of a *SOAP* request and response pair are anything but simple. If we had to build the request and extract the result ourselves, *SOAP* would be very complicated. However, most *SOAP*-based services provide a detailed, machine-readable description of the available methods and their expected inputs and outputs. This is called a *WSDL* (Web Service Description Language) document. The SSOAP package [17] can read this description of the Web service and generate *R* functions and classes that hide all of the details of how *SOAP* works from the *R* user.

In contrast, many *REST* APIs are quite simple to understand. We send an *HTTP* request, with any inputs included in the *URL* or placed in the body of the request, much as we do for an *HTML* form. The results are usually returned directly. We are already familiar with how to compose requests using

GET, **POST**, ... requests. Processing the *XML* or *JSON* results is also quite straightforward using xmlParse() and *XPath*, or fromJSON(). So each part of the low-level details of a *REST* call is quite straightforward. This is the great strength of *REST*: it leverages familiarity with existing technologies. *SOAP* also leverages *HTTP* and *XML*, but also involves various additional layers of complexity. If we were working at this level, *REST* would dominate *SOAP*. But because code to use *SOAP* is typically generated programmatically for us, *SOAP* is actually easier to use as an end-user. We do not need to concern ourselves with reading the documentation for the methods, the details of the *HTTP* requests, or transforming the result to an *R* object. Furthermore, it provides better type specification and type checking, which we can do on the client-side before making erroneous requests. Fortunately, there is a slowly emerging equivalent to *WSDL* documents for *REST*: *WSDL* 2.0 and *WADL* (Web Application Description Language) documents. We discuss the latter in Section 10.5.

10.2 Simple *REST*

At its very simplest, using a *REST* application can be no more complex than a call to getURLContent() or getForm(). We can send numeric and text values for the expected parameters, and the Web service responds with content that we then interpret. The following example, considers the *REST* API for the biology-related site of the European Bioinformatics Institute (EBI).

Example 10-1 Accessing the European Bioinformatics Institute Protein Databases via REST

The European Bioinformatics Institute (EBI) provides access to a lot of biological data via Web Services. Most of the services are anonymous in that they require no authentication. Many of these services can be accessed using either *SOAP* or *REST*. Documentation for the *REST* services is available from `http://www.ebi.ac.uk/Tools/webservices/about/rest`. One of the services offered at EBI is the Protein Identifier Cross-Reference (PICR) Service. This is a service that helps bioinformatics researchers to associate or connect protein sequences across different datasets. The problem is that each protein sequence does not have a unique identifier. Instead, a protein sequence may be assigned one identifier by one research group or central database curator, and be assigned a different identifier in another setting. When we try to combine data using different identifiers, we want to map the different identifiers for the same protein sequence to a common identifier. The PICR service allows us to do this by searching across many different databases to associate an identifier with the corresponding identifiers for the same sequence. It is important to recognize that this is a "live" service and the data within the service is continually updated. This is one of the advantages of a Web service over the alternative of using a local database that is not updated centrally.

One of the things that we want in such a service is to query what databases it can search for cross-referencing identifiers. This allows us to see if the databases used in our data sets are included. It also allows us to determine the exact names of the databases so that we can potentially use them in requests to limit the search to specific databases. In addition to listing the available databases, the important method we need is to lookup a protein sequence identifier we specify and return the identifiers used in other databases for the same sequence. We can find documentation for these methods at `http://www.ebi.ac.uk/Tools/picr/RESTDocumentation.do`. There are four different methods: getUPIForAccession, getUPIForBLAST, getUPIForSequence, and getMappedDatabaseNames. The documentation for each tells us the *URL* to which we send the request and the list of required and optional parameters and what they mean and control. For this API, the documentation also has a link to a sample result. This gives us a sense of what information will be returned and how. However, it is a sample and may not illustrate all of the possible elements. For that, we would need a

schema defining the potential structure. For many *REST* methods, we will also be told how to make the request, i.e., which *HTTP* operation to use. In many cases, this is **GET** and when we are not told, **GET** will be the default.

One of the first things we can do with this interface is get a list of all the available databases. We do this by querying the *URL* or resource `http://www.ebi.ac.uk/Tools/picr/rest/getMappedDatabaseNames`. The documentation indicates that this method has no parameters. Therefore, we can use getURLContent() to access the resource, i.e., if the *URL* is a string in `u`,

```
db = getURLContent(u)
```

This returns a very "flat" *XML* document which has a root node with many *<mappedDatabases>* children. It looks something like

```
<ns2:getMappedDatabaseNamesResponse
  xmlns="http://model.picr.ebi.ac.uk"
  xmlns:ns2="http://ww...picr/AccessionMappingService">
 <ns2:mappedDatabases>EG_BACTERIA</ns2:mappedDatabases>
 <ns2:mappedDatabases>EG_FUNGI</ns2:mappedDatabases>
  ...
</ns2:getMappedDatabaseNamesResponse>
```

Each *<mappedDatabases>* child element of the root contains only a text node giving the name of the database. We can convert this easily to an *R character* vector using xmlToList():

```
as.character(xmlToList(db))
```

```
 [1] "EMBL"              "EMBLWGS"           "EMBL_ANNCON"
 [4] "EMBL_TPA"          "EMBL_TPX"          "ENSEMBL"
 [7] "ENSEMBL_ARMADILLO" "ENSEMBL_BUSHBABY"  "ENSEMBL_CAT"
[10] "TROME_HS"          "TROME_MM"          "UNIMES"
[13] "USPTO"             "VEGA_DOG"          "VEGA_HUMAN"
[16] "VEGA_MOUSE"        "VEGA_ZEBRAFISH"    "WORMBASE"
```

The reason we might use the PICR service is to look up the protein identifiers for a value we have in our data to find the alternate identifiers used in different database for that same protein. These protein identifiers come from the UniProt archive and are termed UPIs. The PICR service provides three methods for matching UPIs in the collection of databases. These methods allow us to specify the input protein of interest in different formats—by an accession string, a BLAST fragment, or a protein sequence. These three methods are getUPIForAccession, getUPIForBlAST, and getUPIForSequence, respectively. The API provides documentation for the methods, including the *URL*, the parameters, whether or not they are required, and any default values. For example, the getUPIForAccession method has six parameters: accession, version, database, taxid, onlyactive, and includeattributes. Only two are required—accession and database—while the others are all described as being optional. We have to specify at least one database in which to search. However, it is possible to provide multiple databases, which has the effect of querying multiple databases in the request. We can use a vector of database names in the call to getForm() when making the request, and this takes care of repeating the `database` parameter for each value in the request.

We can make a simple request to this method by specifying the accession value from our data and the databases of interest in a call to getForm(), e.g., let `ebiURL` contain the string `http://www.ebi.ac.uk/Tools/picr/rest/getUPIForAccession`,

```
acc = getForm(ebiURL, accession = "P29375",
              database = c("SWISSPROT", "ENSEMBL"))
```

We now have the result as an *XML* document in `acc` which we can parse and explore:

```
xmlParse(acc, asText = TRUE)
```

```
<ns2:getUPIForAccessionResponse
 xmlns="http://model.picr.ebi.ac.uk"
 xmlns:ns2="http://www.ebi.ac.uk/picr/AccessionMappingService"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <ns2:getUPIForAccessionReturn
    containsActiveSPTRCrossReference="true">
  <CRC64>FCF6DC22DEF001DF</CRC64>
  <UPI>UPI0000DB2E73</UPI>
  <identicalCrossReferences>
   <accession>P29375</accession>
   <accessionVersion>3</accessionVersion>
   <databaseDescription>UniProtKB/Swiss-Prot</databaseDescription>
   <databaseName>SWISSPROT</databaseName>
   <dateAdded xsi:nil="true"/>
   <dateDeleted xsi:nil="true"/>
   <deleted>false</deleted>
   <gi xsi:nil="true"/>
   <taxonId  xsi:nil="true"/>
  </identicalCrossReferences>
  <logicalCrossReferences>
   <accession>ENSP00000382688</accession>
   <accessionVersion xsi:nil="true"/>
   <databaseDescription>ENSEMBL</databaseDescription>
   <databaseName>ENSEMBL</databaseName>
   <dateAdded  xsi:nil="true"/>
   <dateDeleted  xsi:nil="true"/>
   <deleted>false</deleted>
   <gi xsi:nil="true"/>
   <taxonId xsi:nil="true"/>
  </logicalCrossReferences>
     ...
  <logicalCrossReferences>
       ...
  </logicalCrossReferences>
  <sequence xsi:nil="true"/>
  <timestamp>2009-01-12T00:00:00.000Z</timestamp>
 </ns2:getUPIForAccessionReturn>
</ns2:getUPIForAccessionResponse>
```

The content and structure of this document are defined via a schema available at http://www.ebi.ac.uk/Tools/picr/service?wsdl. This helps us to understand the general structure and possible elements rather than basing our understanding on one or two empirical examples. We can even use the XMLSchema package [20] to help define *R* classes and functions

to read such documents. The data we want are in the <identicalCrossReferences> and <logicalCrossReferences> elements. We can get these with

```
nodes = getNodeSet(doc, c("//x:identicalCrossReferences",
                          "//x:logicalCrossReferences"), "x")
```

In fact, we may just want the values of the <accession> elements within the <logicalCrossReferences>. We may also want the <databaseName>. In other situations, we may want the date added and so on.

Our example shows a single call to the getUPIForAccession method. However, we are more likely to have a collection of accession identifiers and will want to lookup up the identifiers for each. The Web service does not provide a vectorized version of the method that allows us to pass many accession values. Instead, we have to make individual requests for each accession value. In this case, it is a good idea to create a curl connection object with getCurlHandle() and to use this same connection for all of the requests. This may speed up the overall time to make the requests by avoiding re-establishing the connection to the Web server for each request.

This basic example illustrates that it is up to us to process the response from the *REST* query. In this example, there is a schema available for understanding the structure and semantics of the resulting document. In other cases, we may get back data in a different format, such as *JSON*. The format will be described in the Web service's documentation and also identified in the header of the Web server's *HTTP* response. If we need to access this information in the response, we can do so using `header = TRUE` in the call to the lower-level RCurl functions and via the *.opts* in the getForm() call.

Next, we look at an example where we specify arguments in each request as part of the path in the URL. Also, the Web service returns the result as *JSON* (by default).

Example 10-2 Parameterizing REST Requests for Climate Data from the World Bank

Climate change is, of course, of immense interest. Many different research groups run simulations for their own global circulation/climate models (GCMs) under different scenarios and report the results. The two primary outcomes are temperature (at the surface) and precipitation, named `Tass` and `pr`, respectively. It is sometimes useful to be able to retrieve updated predictions for these variables for different years for some or all of the models and scenarios. The World Bank provides a *REST* service to query both model predictions and historical data used to calibrate the models. Results are available for each country and also for different river basins. We can read about the details of calling the *REST* methods at `http://data.worldbank.org/developers/climate-data-api`.

The basic mechanism to get data for a country is to make a **GET** request to a *URL* of the form `http://climatedataapi.worldbank.org/climateweb/rest/v1/country/type/var/start/end/ISO3[.ext]` We parameterize the request by specifying values for type, var, start, end, and ISO3 in the *URL* to which we send the request. The `type` parameter specifies the form of aggregation, e.g., monthly or yearly average, or monthly or annual change. The `var` part of the *URL* identifies which variable we want and should be either `tas` or `pr`. We cannot access both in a single request. The `start` and `end` parts specify the years of interest. These must be given in pairs spanning 20 years, e.g., 1920 and 1939, 1940 and 1959, and so on from there up to 2080 to 2099. The `ISO3` parameter identifies the country of interest, using the *ISO*'s official country name abbreviations.

The `[]` in the `[.ext]` component of the *URL* typically indicates that the parameter `.ext` is optional. In this API, this part of the *URL* allows us to specify the format of the result, e.g., `json` for *JSON* output, `xml` for *XML* content or `csv` for CSV format. The default is *JSON*.

Having understood this information, we can make requests such as

```
ans = getURLContent("http://../v1/country/mavg/tas/2020/2039/USA")
```

The (abbreviated) *JSON* result is something like

```
[{"scenario":"a2","gcm":"bccr_bcm2_0","variable":"tas",
  "monthVals":[-8.894878359,-7.6987..,-6.192306899],
  "fromYear":2020,"toYear":2039},
 {"scenario":"b1","gcm":"bccr_bcm2_0","variable":"tas",
   "monthVals":[-8.831965839,-6.985148936,-3.......],
    ....
 } ....
```

We can use fromJSON() to convert this and then access the details directly as *R* objects:

```
tas = fromJSON(ans, asText = TRUE)
```

The result includes an entry for each combination of global circulation model (GCM) and scenario under which the model was run and we requested. The `monthVals` field in each element is a vector giving the predicted value for each of the 20 years.

Note that as with the PICR example, we again have the result as a *JSON* or *XML* document in memory, i.e., a string, rather than in a file. This is why we use the *asText* argument to parse the result.

We can make requests for different years, variables, etc., by substituting the different parameters into the *URL* and calling getURLContent(). However, it makes sense to create a function to perform the requests and also convert the results. The function can provide useful defaults so that the caller does not need to remember or specify all the details. It can also check that the input values are correct before making the request. For example, we can verify that the name of the country is valid, or match it against the full name of the countries to get the abbreviation we need in the request. Similarly, we can align the year to the nearest start and end expected by the *REST* server. A basic version of the function might look something like

```
climateData =
function(country, start = 2020, variable = "tas", type = "mavg",
         format = ".json", curl = getCurlHandle(.opts = .opts),
         ..., .opts = list(...), baseURL = ClimateDataAPIURL)
{
  years = c(seq(1920, by = 20, length = 4),
            seq(2020, by = 20, length = 4))
  origin = years[which.min(abs(years - start))]
  u = sprintf("%s/%s/%s/%s/%s/%s%s", baseURL, type, variable,
              origin, origin + 19L, country, format)

   ans = getURLContent(u, curl = curl, .opts = .opts)
   tmp = fromJSON(ans)
   convertToDataFrame(tmp, type, variable)
}
```

where `ClimateDataAPIURL` holds the *URL* `http://climatedataapi.worldbank.org/climateweb/rest/v1/country`. The function takes the different inputs and constructs the *URL* for the request using these inputs. It then makes the request and converts the result, first parsing the *JSON* document and then converting it from a list arranged by GCM and scenario to a data frame. Note that we perform a lot of error checking on the inputs within the function before making the request. It is better to catch the errors early rather than wasting bandwidth and time waiting for

a response from the server that will end in an error. Our function also must check the result of the request and check if the server indicated an error. getURLContent() and the other functions used to make a request will catch an *HTTP* error, but we need to check the value if the request is successful but the Web server method is not.

Our function allows the caller to specify a `CURLHandle` object via the *curl* parameter. Again, this allows us to create a single connection to the Web server and reuse it in repeated calls for different data. The function also allows us to specify options for controlling how the request is performed via the *.opts* and ... parameters. It is useful to allow the options be specified either individually via ... or as a single list. The default value for *.opts* is the list of options specified via ... and we pass this to getURLContent() when making the request.

We can develop this function further and allow the caller to specify start and end years that span more than 20 years and have the the function make multiple requests for the different 20-year periods and merge the results. We may also choose to vectorize this function so that it gets both temperature and precipitation.

We can process the country the caller specifies by matching it to the complete table of abbreviations or full country names. We can get a table of the official country abbreviations and their name/form at `http://unstats.un.org/unsd/methods/m49/m49alpha.htm`, as specified in the documentation for the *REST* service. This is an *HTML* document that displays the country and abbreviation in a table. We can use readHTMLTable() to get this information into *R*.

The *REST* interface allows us to specify a particular global circulation model (GCM) and/or scenario. This limits the result to that particular model or scenario. These are not optional in the same way that the format or extension is added to the end of the *URL* for the request. Instead, we change the *URL* to include the name of the GCM or the scenario after the `/v1/country/type` part of the path, but before the name of the variable (`tas` or `pr`), separating the GCM or scenario with a /. For example, to get the monthly precipitation values from the Norwegian model `bccr_bcm2_0` for the period 2020 — 2039, we would use the URL `http://climatedataapi.worldbank.org/climateweb/rest/v1/country/mavg/bccr_bcm2_0/pr/2020/2039/USA`

We can change our function to allow the caller to specify a string identifying the GCM and another parameter for indicating the scenario. If either (or both) of these is specified, we create the *URL* differently. The rest of the function remains the same.

While *REST* is very closely connected with *HTTP* and Web servers, *REST* can also be used to communicate with applications that are not Web services. It is becoming common for applications such as databases, text search engines, or simply stand-alone applications to allow clients to communicate with them via *HTTP* requests. Clients can get information from the application via **GET** and **POST** requests, using specific URLs that the application makes available and understands. These URLs are similar to methods, but often correspond to explicit resources in the *REST* sense. Clients may be able to update the application and its views via **POST**, **PUT**, and **DELETE** requests. Again, this is quite simple using what we know of *HTTP*, *XML*, and/or *JSON* and the different *R* packages.

One of the key ideas here is that we can use *HTTP* to make these requests to the applications. This has a lot of immediate advantages. We can use the different authentication mechanisms *HTTP* provides, including passwords and *HTTPS* for secure connections. System and network administrators typically allow or can easily permit *HTTP* requests on their networks.

We look next at an example of using *REST* to connect to a *NoSQL* database. There is also an extensive example showing how to communicate with the ElasticSearch text search engine in Section 7.4.3.

10.2.1 Accessing the NoSQLDatabase CouchDB via REST

In this example, we will show how we can use *R* to communicate with the *NoSQL* database CouchDB [2]. The `R4CouchDB` package [3] provides high-level functions to do this for us, but here we will show the low-level *REST* operations performed directly in *R* to illustrate the *REST* concepts. CouchDB is a nonrelational database that uses *JSON* to store documents. We can insert, modify, and access documents using *HTTP* requests via a *REST*-style interface. We see how to use all of the usual *HTTP* operations and also a nonstandard operation, **COPY**. Each of the *HTTP* requests return a *JSON* document and we specify the inputs to some of our requests also as *JSON*. This example illustrates that *REST* is not constrained to be used only via the Web, but can exploit *HTTP* in other circumstances.

We start by running a CouchDB server. We do this by downloading and installing CouchDB and then running the *couchdb* executable as an administrator or root, e.g.,

```
sudo /usr/local/bin/couchdb
```

We now have a *REST* server that is listening for requests on port 5984 on our local machine. Depending on how CouchDB is configured, we can send requests from other machines as well as the local host on which the server is running.

We can start by "pinging" the server to see if it is running and listening. We do this by simply requesting a top-level resource:

```
fromJSON(getURLContent("http://localhost:5984"))
```

```
  couchdb   version
"Welcome"   "1.1.0"
```

The request returns the *JSON* document and we convert it to an *R* object.

Before we add any documents, we need to create a database. We do this by creating a resource in the server identifying the new database, and we specify it as a URI that gives a name, say `mydata`, to the database, e.g., `http://localhost:5984/mydata`. (Note that in CouchDB the database name must be lowercase. This is not a *REST* restriction.) We use a **PUT** request to create the database. This is because we are controlling the name of the resource, i.e., the URI by which we can access it, and we are not leaving it to CouchDB to create its own name or URI for the resource. So our request can be made as

```
httpPUT("http://localhost:5984/mydata")
```

Although we use a **PUT** request, there is no body in our request. Again, we convert the result from *JSON* and check the status field.

Now that we have our own database, we can populate it with content. We can store any value in the database by sending its *JSON* representation. In our example, we store *R* objects. We serialize them to *JSON* via toJSON() and treat that as a CouchDB document. We can then retrieve the values and deserialize with fromJSON(). We can also use the `opencpu.encode` [4] package to serialize and deserialize the *R* objects using *JSON* in order to preserve all of the *R*-specific attributes and handle missing values. Regardless of the particular mechanism, if we serialize the objects, we end up with a *JSON* string. We can then insert this into the database as a document by creating another new resource, again specifying the name in the URI. Since we are specifying the name of the resource, again we use a **PUT** operation. We do this with

```
jfit = toJSON(robject)
fromJSON(httpPUT("http://localhost:5984/mydata/fit", jfit))
```

Here, `robject` is any *R* object, e.g., a vector, data frame, fitted linear model, etc. Our call returns a *list* with the status, the identifier ("fit") and also the revision/version identifier for the document. This is generated by CouchDB.

We can insert documents into our database without specifying the name. In this case, CouchDB will generate its own name for the document. For this, we use a **POST** request, e.g.,

```
info = httpPOST("http://localhost:5984/mydata/",
                postfields = jfit,
                httpheader = c('Content-Type' = 'application/json'))
```

We convert the *JSON* content in `info` to get the status and also the identifier for the new document. Note that we have to specify the *Content-Type* in the header of the *HTTP* request for CouchDB to recognize the content.

Retrieving the contents of a document in the database is a simple **GET** request identifying the document by its URI. For example,

```
fromJSON(httpGET("http://localhost:5984/mydata/jfit"))
```

Similarly, we can get a list of all of the documents in a database with a **GET** operation using the special resource name in the database `"_all_docs"`:

```
fromJSON(httpGET("http://localhost:5984/mydata/_all_docs"))
```

CouchDB allows us to directly copy a document to another name. We can do this by first retrieving the document and then **PUT**'ing the document to the new name. However, this involves sending the document between the server and client twice, which is unnecessary. CouchDB allows us to use the nonstandard *HTTP* operation **COPY**. We use this by making a request to the existing resource—the URI of the document to be copied. We add a *Destination* field to the *HTTP* request header to specify the name of the new document to be created in the database. We can do this with

```
curlPerform(url = "http://localhost:5984/mydata/fit",
            httpheader = c(Destination = "modelFit"),
            customrequest = "COPY")
```

The *Destination* value is not part of the URL or something we pass as a form parameter in the *HTTP* request. It is part of the *HTTP* header. This illustrates yet another way we can provide information to the server.

We can use a **DELETE** request to remove a document or an entire database. For a database, we can use a simple **DELETE** request such as

```
httpDELETE("http://localhost:5984/mydata/")
```

To do this for a document, however, CouchDB requires us to specify the current version/revision number in the request. We obtain this value by retrieving the document and accessing its `_rev` field, or by listing all of the documents via the _all_docs resource for the database. We do the former with

```
rev = fromJSON(httpGET("http://localhost:5984/mydata/fit"))$"_rev"
```

but note that this involves retrieving the entire contents of the document. Once we have the revision number, we can specify it as the value of the rev parameter in a *HTTP* request to delete the document. We can create the *URL* with the `rev` parameter appended to it, and then use httpDELETE() to make the request. Alternatively, we can use getForm() to specify the URL and the *rev* argument separately. However, we have to change the *HTTP* operation. So either of the following commands work:

```
httpDELETE(sprintf("http://localhost:5984/mydata/fit?rev=%s",
                   rev))
getForm("http://localhost:5984/mydata/fit", rev = rev,
        .opts = list(customrequest = "DELETE"))
```

Note that we can rename a resource by a **COPY** operation followed by a **DELETE** request.

This description was initially motivated by Christopher Bare's blog post at http://digitheadslabnotebook.blogspot.com/2010/10/couchdb-and-r.html. More information about CouchDB's *HTTP* API is available on the Web, e.g., http://wiki.apache.org/couchdb/HTTP_Document_API. As we mentioned, the R4CouchDB package provides a higher-level interface, hiding the details of the requests.

Section 7.4.3 provides another example of *R* communicating with an application—*ElasticSearch*—via *REST*. The focus in that example is on the *JSON* content of the communication. However, the *REST* operations are reasonably clear and reinforce the concepts and techniques of using *REST* to communicate with an application.

10.3 Simple Authentication

We have seen that basic *REST* is no more complex than emulating the submission of an *HTML* form or other simple *HTTP* requests. We can query a resource and there is no need to identify ourselves or provide any authentication information to access the information. Many Web services, however, do require some form of user/caller identification. In this section, we focus on sites that provide access to public data, but which require a user identifier in each request. Even though the data being accessed are not specific to a particular user, the sites want to know who is accessing the data. They want to know so that they can give some callers preferred access. They also want to prohibit abuse of the service and possible denial-of-service attacks by limiting the number of requests each caller can make in a given period. We typically obtain this ID by registering with the service and obtaining a login and password. Having created an account, we get a unique identifier (different from the login and password) that we can include in each Web service request to identify ourselves as the caller. We demonstrate how to authenticate the request with a Web service ID with two examples. The first uses the Zillow API. Zillow is a real estate site that offers a *REST* service to retrieve home valuation estimates, valuations for comparable houses, market trends, information about recently sold homes, and mortgage rates. The second example accesses weather data from the National Oceanic and Atmospheric Administration (NOAA). Additional examples such as accessing data from last.FM and *New York Times* are discussed on the Web site for this book.

Example 10-3 Authenticating a Request to Zillow for Housing Prices

Zillow [21] (http://www.zillow.com) provides access to US housing prices and estimates through *HTML* forms as well as through *REST* Web services. The *REST* services allow registered users to retrieve the estimated market value, the low and high estimated market value range that indicates the level of accuracy of the estimate, and the change in the estimated value over the past 30 days. Additionally, in a separate query, we can retrieve information about properties that Zillow deems comparable to a specified property of interest. More information about these data can be found at http://www.zillow.com/howto/api/APIOverview.htm along with several other APIs, such as for mortgage rates.

Before we can use these services, we need to register with Zillow and create a login and password. We do this in our Web browser by visiting https://www.zillow.com/webservice/

Registration.htm and filling in the form. Once we have our account, we request a Zillow Web Services ID, or ZWSID for short. The ZWSID is what we use to authenticate our requests. We do not use the login and password we used to obtain the ZWSID.

In this example, we focus on two of the Web service methods which return the Zillow estimate and the comparables for a property. The *R* package Zillow [14] provides two relatively simple functions to interface to these two Zillow *REST URL*s. These functions are zestimate() and getComps(). They correspond to two of the methods described in the documentation for the API. The first gives the estimate for a single property and the second returns a data frame giving details of 30 houses that are comparable to the property. The zestimate() function requires two arguments that identify the property. The first is a street address such as 1234 Springfield Avenue. The second is either a ZIP code or, alternatively, a city and state separated by a comma, e.g., Chicago, IL. This function makes a **GET** call to the *URL* http://www.zillow.com/webservice/GetSearchResults.htm and passes the caller's Zillow id (ZWSID), the address, city, and state strings as form parameters. Assuming the property can be located with this information, Zillow returns an *XML* document containing the result, which zestimate() parses to extract the information. Again, we need to understand the format and content of the result in order to extract the relevant information. The function zestimate() calls the getForm() function to make the request and then processes the return value into a data frame. It is defined as

```
zestimate =
function(address, citystatezip,
         zillowId = getOption("ZillowId",
                              stop("No ZillowId set")),
         ..., .opts = list(...), curl = getCurlHandle())
{
  u = "http://www.zillow.com/webservice/GetSearchResults.htm"
  reply = getForm(u, `zws-id` = zillowId, address = address,
                  citystatezip = citystatezip,
                  curl = curl, .opts = .opts)
  doc = xmlParse(reply, asText = TRUE)
  checkStatus(doc)
  zpid = xmlValue(doc[["//result/zpid"]])
  est = doc[["//result/zestimate"]]
  data.frame(
   amount = as.numeric(xmlValue(est[["amount"]])),
   low = as.numeric(xmlValue(est[["valuationRange"]][["low"]])),
   high = as.numeric(xmlValue(est[["valuationRange"]][["high"]])),
   valueChange30Day = as.numeric(xmlValue(est[["valueChange"]])),
   row.names = zpid)
}
```

As an example, we get the estimate for a particular house:

```
zestimate("1280 Monterey Avenue", "94707", zillowId = zid)

         amount    low   high valueChange30Day
24842790 727500 640200 814800            -1500
```

We get the current estimate and a range for the value, and also how much the valuation has changed in the last 30 days. The row name on the data frame is the identifier for the property in Zillow's database. We can use this in requests to other methods.

Note that the function again allows the caller to specify curl options and also a curl handle object via the the ..., *.opts*, and *curl* parameters, respectively.

The function also allows the caller to specify the ZWSID to be sent in the request. However, we also provide a default value for this in the function. We definitely do not hard code our own private identifier or share it with other users. Instead, we use the idiom of looking for the identifier as an *R* option. We agree on a name for the option, *ZillowId*, and then users can set that option with their own identifier string. The default value for the argument in the function then tries to access this, and raises an error if it is not found. This approach means that we never have to explicitly have the private identifier in code. This means we can share code with others that will work but that does not reveal the identifier. We set the option in our `.Rprofile` that is read when *R* starts. We make certain that this is not readable by anybody but ourselves.

Note that the function checks the response from the server to see if the Web service request was successful. If there is an *HTTP* error, the getForm() function will raise that. However, if the *HTTP* request is successful, but the *REST* method returns an error, we need to identify this and in turn raise an *R* error. We do this by writing the function checkStatus(), which is specific to the Zillow API, and maybe even to the particular method. The documentation for each of the Zillow methods provides a table of error codes that might be returned within the response's *XML* document. We can use these to raise different types or classes of error in *R*. This is good practice as it allows callers of the function to trap and handle an error based on its class using the tryCatch() function. See the condition mechanism in *R* [5].

Once we have the Zillow property ID for a house (the row name of the data frame), we can use it in other requests. For example, we can get information about comparable houses near the property. The *URL* for this request is `http://www.zillow.com/webservice/GetComps.htm`. From the documentation, this has four parameters. The first two are `zws-id` for the ZWSID value and `zpid` for the property identifier for which we want the comparables. The third parameter is named `count` and specifies the number of comparable properties we want. This is limited to 25. The fourth parameter is optional. It is named `rentzestimate` and is expected to be `true` or `false`. This controls whether rent estimates are returned along with the comparables. The Zillow package provides a function named getComps() as a wrapper for this *REST* method. We can call it with

```
comps = getComps("24842790", zillowId = zid)
comps[1:5, 11:14]
```

```
          yearBuilt lotSizeSqFt finishedSqFt bathrooms
24842790       1925        6520         1397         1
24842214       1906        2500         1198         2
24844881       1921        5600         1262         1
24842788       1926        6960         2361         2
24837285       1922        4750         1674         2
```

Again, the function uses getForm() to make the request and converts the resulting *XML* document into a data frame. The row names on the data frame are the property identifiers, and we can use these to find their comparables, for example.

The API provides other methods, including accessing details about individual properties, e.g., number of bedrooms, when it was sold, and demographics about a neighborhood.

Next, we look at a *REST* service that uses an access token and a flexible *URL* template to make different data available.

Example 10-4 Using an Access Token to Obtain Historical Weather Data from NOAA

The National Oceanic and Atmospheric Administration (NOAA) makes data available via *REST* and *SOAP* Web services. (See `http://www.ncdc.noaa.gov/cdo-web/webservices`.) We can get information about what data sets or variables are available, what stations are available within a data set, where the stations are located, and of course actual values for the variables such as annual or hourly precipitation or temperature. Before we can make requests, we must request a (no-cost) access token with which we can make requests. The data are public, but NOAA wants to be able to track who is using the service. They might collect this to understand their users and the patterns of access. They might also use it to limit excessive use that would degrade the performance of the service. The access token is a 32-character key or string that identifies an individual and it must be used in all requests. This should be the value of the `token` parameter for each method. These requests all use **GET** operations. This is because a) we are not changing the resources, but merely querying them, and b) the inputs in our requests are quite simple and do not need to be **POST**ed in the body of the request.

By default, the service returns the result in *JSON* format. We can, however, request the result in *XML*. The content is the same, but just in a different format. We might prefer *XML*, for example, if we want only a small subset of the response. We might use an *XPath* expression to extract those values, rather than processing an entire *JSON* document and discarding most of the content. For some Web services, we specify the desired output format with another parameter in the request, e.g., `format=xml`. For some Web services, we can append the format to the request as if it were a file extension, e.g., set `u` to `http://www.ncdc.noaa.gov/cdo-services/services/datasets.xml` rather than `http://www.ncdc.noaa.gov/cdo-web/webservices`. For the NOAA service, we can also specify the desired format using the *HTTP* header entry *Accept*. For example,

```
xml = getForm(u, token = NOAAToken,
              .opts = list(httpheader = c("Accept" =
                                          "application/xml")))
```

This illustrates that we can use standard *HTTP* operations in our *REST* requests to control how the server responds.

The result of querying the `datasets` URL is an *XML* document that tells us what data sets are available from the Web service. The document looks something like

```
<dataSetCollection totalCount="9" pageCount="1">
 <dataSet>
  <id>ANNUAL</id>
  <name>Annual Summary</name>
  <description>Annual Climatological Summary</description>
  <minDate>1831-02-01-05:00</minDate>
  <maxDate>2012-05-01-04:00</maxDate>
  <attributes>
   <attribute>
    <name>Units</name>
    <defaultValue> </defaultValue>
    <indexNumber>4</indexNumber>
   </attribute>
```

```
  </attributes>
 </dataSet>
 <dataSet>
  <id>GHCND</id>
  <name>Daily GHCND</name>
  <description>GHCN-Daily</description>
  <minDate>1763-01-01-05:00</minDate>
  <maxDate>2012-11-18-05:00</maxDate>
    ....
 </dataSet>
</dataSetCollection>
```

Each <*dataSet*> node contains information about a different resource within the service. We have the name we need to use to query that resource (e.g., ANNUAL or GHCND), a more human-readable description of that data set/resource, and metadata about it. We can get the names of the different datasets with

```
xpathSApply(xmlParse(xml), "//dataSet/id", xmlValue)
```

```
[1] "ANNUAL"      "GHCND"       "GHCNDMS"    "NORMAL_ANN" "NORMAL_DLY"
[6] "NORMAL_HLY" "NORMAL_MLY" "PRECIP_15" "PRECIP_HLY"
```

We can think of the *URL* http://www.ncdc.noaa.gov/cdo-services/services/datasets as the top-level resource for the NOAA Web service. The different data sets that we just listed are subresources. We refer to them by adding the data set identifier, e.g., http://www.ncdc.noaa.gov/..../datasets/ANNUAL When we send a request to that resource, we get the metadata for just that resource.

We can access various other subresources for a particular data set resource. For example, we can query all of the locations for which that data set is measured, or the details of the actual stations where the measurements are recorded. We can also query the types of the locations, and discover the data types collected for that data set. For example, we can get the types of locations for a given type of data set using a *URL* of the form http://www.ncdc.noaa.gov/.../datasets/ANNUAL/locationtypes . We have substituted ANNUAL into the *URL* to identify the specific data set we want and queried the locationtypes resource within this set. With a little massaging of the results, we see the possible values are

```
      id          name
 [1,] "CLIM_DIV" "Climate Division"
 [2,] "CLIM_REG" "Climate Region"
 [3,] "CNTRY"    "Country"
 [4,] "CNTY"     "County"
 [5,] "HYD_ACC"  "Hydrologic Accounting Unit"
 [6,] "HYD_CAT"  "Hydrologic Cataloging Unit"
 [7,] "HYD_REG"  "Hydrologic Region"
 [8,] "HYD_SUB"  "Hydrologic Subregion"
 [9,] "ST"       "State"
[10,] "ZIP"      "Zip Code"
```

Locations, stations, location types, and data types are types of resources that apply to all data sets. In other words, for a given data set or variable, we can filter by any of these resources, subsetting

for different values of each of these resources. We can drill down hierarchically to find, for example, descriptions of the stations at a particular location, for a given location type. To find the set of possible values for each resource, we query that resource directly. For example, we have found the location types for the data set `ANNUAL`. We can find the possible locations for the `ZIP` type for `ANNUAL` via the *URL* http://www.ncdc.noaa.gov/.../datasets/ANNUAL/locationtypes/ZIP/locations.xml Basically, we are subsetting on the location type being equal to ZIP, and then querying the available locations. The result is a document containing a sequence of *<location>* nodes of the form

```
<location>
 <id>ZIP:01370</id>
 <displayName>Shelburne Falls, MA 01370</displayName>
 <locationType>
  <id>ZIP</id>
  <name>Zip Code</name>
 </locationType>
 <minDate>1948-06-01-04:00</minDate>
 <maxDate>1977-05-31-04:00</maxDate>
 <stationCount>1</stationCount>
 <coverage>1</coverage>
</location>
```

We can use the value of the *<id>* node to query that particular location. We can then focus on a single location and find the information about its different stations http://www.ncdc.noaa.gov/.../datasets/ANNUAL/locationtypes/ZIP/ZIP:95616/locations.xml . There is a single station for this ZIP code and it is described as

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<stationCollection totalCount="1" pageCount="1"
   pageSize="100">
 <station>
  <id>COOP:042294</id>
  <displayName>DAVIS 2 WSW EXPERIMENTAL FARM, CA US</displayName>
  <minDate>1908-06-01-05:00</minDate>
  <maxDate>2012-05-01-04:00</maxDate>
  <latitude>38.5349</latitude>
  <longitude>-121.7761</longitude>
  <elevation>18.3</elevation>
  <locationLabels>
   <type>ZIP</type>
   <id>ZIP:95616</id>
   <displayName>Davis, CA 95616</displayName>
  </locationLabels>
  <coverage>0.9864</coverage>
 </station>
</stationCollection>
```

We can find the possible variables measured at this station via the *URL* http://www.ncdc.noaa.gov/.../datasets/ANNUAL/locationtypes/ZIP/ZIP:95616/datatypes.xml This returns a long list of possible measurements such as “extreme maximum daily temperature”

(EMXT) and "number days with minimum temperature less than or equal to 32.0 F" (DT32). We use the identifiers `EMXT` and `DT32` to access the values for these variables.

In general, we can combine the different subresources in different ways to make a query and subset the different resources. One of the general *URL* patterns is

```
/datasets/{dataSet}/locationtypes/{locationType}/
                                        datatypes/{dataType}
```

Here we substitute the terms enclosed by `{}` with specific values. For example, we use ANNUAL for `{dataSet}` and ZIP for `{locationType}`. We add terms to the path to restrict to a particular value of that resource, e.g., `locationtypes/ZIP/ZIP:95616` to restrict to a particular ZIP code.

Note that all of the queries to find the subresources were made by a call to getForm() using the appropriate *URL* and passing our access token.

Of course, most of what we have explored above returns metadata about what data may be available for the different subresources. We can retrieve the actual data values using the URLs to specify the resource and appending the term `data` to the end. This is described in the documentation at `http://www.ncdc.noaa.gov/cdo-web/webservices/cdows_data` and there we find that, in addition to the *URL* and access token, we need to provide additional parameters. We must specify the year, month, and starting and ending day to identify the time period. For example, to get the hourly precipitation for the Davis station for the first week in January, 2011, we use the query

```
getForm("http:.../PRECIP_HLY/locations/ZIP:95616/data",
        token = NOAAToken, year = 2011, month = 1,
        startday = 1, endday = 7)
```

The `startday` and `endday` are optional arguments, defaulting to 1 and 31, respectively.

If our query identifies a large amount of data, the server will divide it into pages and return it to us one page at a time. The total number of pages and the current page are returned in the result, and we can use this information to return to the server to retrieve the next page.

As with the other *REST* interfaces, it makes sense to hide the details of the *REST* queries by defining one or more *R* functions. The malleable hierarchy of resources makes this a little problematic. We will discuss approaches to this in Section 10.5, specifically Example 10-8 (page 374).

The NOAA *REST* API is more complicated than others because of the hierarchy of resources. The creators could have allowed us to specify the data set, location, station, data type, etc., as parameters in a request to a single method. However, treating these as resources probably makes more sense. The interface still uses parameters to specify the day, etc. There are other resources such as text and region searches that also take parameters to specify the details of the particular search. The example also illustrates how we use the access token as a parameter in each request. This is the same as for the Zillow API. In other interfaces, we might specify the token as part of the *HTTP* header.

10.4 Changing State with *REST*

In the examples we have seen so far in this chapter, we have been accessing public data, e.g., protein databases, climate simulation data, property value estimates, and weather station data. We used an access token for the latter two only to identify ourselves as the caller, not because the data were restricted to specific individuals. However, there are many Web services that manage data specific

to a particular individual or a particular account. Sites such as Flickr, Facebook, Twitter, Mendeley, and so on, allow users to create and share content. Dropbox also allows users to store files and to manage who has access to them. All of these sites provide Web service APIs that allow us to access public data. More importantly, they also allow account owners to create, retrieve, update, and delete—often referred to as CRUD—their own content (or other content for which they have the appropriate permissions).

The key difference in what we have described here and the earlier examples is that we are modifying the values of the resources and not just querying them. In addition to the **GET** operation, we can use *HTTP*'s **POST**, **PUT**, **DELETE**. The **POST** operation allows us to send a request with more complex content than with **GET**. **PUT** allows us to "upload" a resource to a Web server so that it is available in subsequent **GET** or **POST** requests. A **PUT** request may replace the actual value of the resource, and a **POST** request may update the value of the resource, possibly merging the content from the body of the request. **DELETE**, as you might expect, allows us to remove a resource.

These operations can make for very rich Web services. Indeed, we saw this with the CouchDB example where we created databases and documents, removed them, and updated them. We look here at two other examples focusing on the Web rather than local server applications. We describe how we can access Google Docs to access and publish data and documents. We also look at the Amazon S3 storage service. Unlike the PICR or NOAA services, these are general facilities that many *R* users can leverage as part of their modern workflow, along with many other Web services. In this section, we give a high-level description of these two Web services, and show how we can work with the service via the RGoogleDocs [16] and RAmazonS3 [19] packages, respectively, to actively modify the state of a Web resource. We focus on the high-level functions that hide the details of the *REST* requests, the authentication and authorization mechanism used to gain access to the resources, and also how we map the *REST* methods into *R* functions.

Example 10-5 The Google Docs API

Google Docs (http://docs.google.com) offers an online resource for creating and sharing work in the form of documents. We can create spreadsheet, presentation, and word processing documents via our Web browser in Google Docs. Of course, we can edit any of our documents. We can also create new documents or folders, delete existing ones, rename a document, or move one to a different folder. We can upload different types of documents from our own machines and have Google convert them, if appropriate, to a spreadsheet or word processing document. The contents of the documents live in the cloud on Google's disks. We can share these documents with other people, either to everybody on the Web or to just those with a Google account who we want to allow either to read or potentially edit the documents.

In addition to the Web interface for working with our collection of documents, Google Docs offers an API for querying and manipulating the collection programmatically. In the following sections, we discuss the basics of how to manage the collection of documents using this interface. We can use the API within *R* to programmatically publish data as spreadsheets, or even *R* data files created with save(). We can create more elaborate dynamic reports as spreadsheets and word processing documents that convey the results of analyses. We can read data that our collaborators make available. The important aspect of this is that we can do this programmatically within *R* and treat the documents as *R* objects. We do not have to manually interact with the browser, but can integrate Google Docs into an automated workflow.

We should mention that the Google Docs API also allows us to extract data from a document or to modify the content of an existing document. Indeed, the documents are represented via *XML* which we can manipulate in *R*. We will not, however, discuss these operations in this section since we explore

them in detail in Chapter 15 where we explore working with spreadsheets more broadly. Instead, we focus here on the management of the documents and the *REST* interface.

We also note that the version of Google Docs that we describe here is currently being deprecated. This is not a major issue. The high-level *R* functions we describe here can be adapted to work with both the old and new APIs. The example still illustrates the important aspects of *REST*, and also how higher-level interfaces in *R* make us more robust to changes in the underlying details of the *REST* API. API changes are a fact of life in the rapidly evolving and dynamic world of Web services.

10.4.1 Establishing a Connection with Google Docs from R

Before we can access data on Google Docs either via the Web browser or the API, we need an account. This account is associated with a particular person or group. We obtain this by registering with Google, creating a login name, say `rgdocs@gmail.com`, and creating a password for that account. We can then login to the account via the Web browser and create documents. From here on, we focus on using the API rather than the Web browser.

The documents we create on Google Docs are private unless we explicitly make them public. We do not want other people accessing our documents, or modifying or deleting them. So to access the documents, we need to convince Google that we are who we say we are. Unlike the Zillow and NOAA examples earlier, Google Docs does not give us a unique access token that we can then use forever with the Web service. Instead, Google Docs requires that we send the login and password for each session. We send these via secure *HTTP* (*HTTPS*) so that other people cannot intercept these values and masquerade as us. It is important not to share or reveal the login and password with others. As described in earlier examples, we do not want these values to appear in code, nor do we want to have to type them at the *R* prompt when they are needed. Instead, it is convenient and reasonably secure to set these as the values of the *R* option *GoogleDocsPassword* in the `.Rprofile` file that is read when *R* starts. We do this with

```
options(GoogleDocsPassword =
          c("rgdocs@gmail.com" = "my password"))
```

The value of the option is a character vector where the single element is the password and the name is the login/email address. It is important that the `.Rprofile` not be readable by other people on your machine.

Before we can work with the collection of documents, we need to send a request to get an access token that will give us access to the documents across the calls we make in our *R* session. We use the getGoogleAuth() function in RGoogleDocs to perform the authentication and get the token. We can either explicitly provide the login and password or leave the function to look for them as an *R* option. So the two commands

```
auth = getGoogleAuth(login, password)
auth = getGoogleAuth()
```

will work.

Assuming the call to getGoogleAuth() is successful, the result is an authorization token in the variable `auth` that we must use in each subsequent request to the Google service. This is a long string of seemingly random characters and digits. It has a class in *R* named `GoogleDocsAuthentication`. This makes its purpose and provenance clear, rather than simply being a string. We can use this class then to define methods for this type of string.

Before we explore the details of how to get the authentication/access token, we consider how we will use it to make requests and how we can simplify this in *R*. Given the token, we have to send it back to Google Docs within each request. We do this via a field in the *HTTP* header of each request. The field is named *Authorization* and the value is `GoogleLogin auth=....` where we substitute the token for the .... We can create one `CURLHandle` object and set the header field for that object just once. If we use that connection object for all of our requests, the *Authorization* field will be sent in each request and these will succeed. We can create this connection object with getGoogleDocsConnection(). Indeed, we can use this one function to both obtain the token (by calling getGoogleAuth() for us) and set it in the header with

```
con = getGoogleDocsConnection()
```

Once assigned, we simply pass `con` as the value for the *curl* or *con* parameter in calls to the other functions in the RGoogleDocs or RCurl packages.

To understand how we are working with the *REST* interface, it is useful to see how getGoogleAuth() works. Aside from finding the default value for the login and password, the function calls getForm() to send a request to Google Docs for the token. The call is

```
getForm("https://www.google.com/accounts/ClientLogin",
        accountType = "HOSTED_OR_GOOGLE",
        Email = login, Passwd = password,
        service = service, source = appID)
```

This is just a regular *REST* call with parameters that specify the login and password, the account type, and which of the Google Docs services we are accessing (spreadsheets or general document management) The default Google service is "writely" for working with all types of documents, i.e., word processing, presentation and spreadsheet documents; to work with only spreadsheet documents, we use the "wise" service.

On occasions, the attempt to call getGoogleAuth() returns with an error saying that the operation is Forbidden. The Google authorization service has decided that additional vetting is necessary. To "cure" the problem, use your Web browser to log into the same account on Google Docs directly. This will present you with a CAPTCHA that you must respond to and this will "unlock" your account for programmatic access.

This mechanism of getting a token still works but is being deprecated. While it is still useful to see, it is being replaced by use of *OAuth* 2.0. Indeed, many services use *OAuth* 1.0 or *OAuth* 2.0 to allow authenticated and authorized access to a user's content. This is a little more complicated and involved, but much richer both featurewise and in terms of the security it affords. Future versions of the RGoogleDocs package will use this, and how we use the *R* functions will remain essentially unchanged. This is one of the advantages of providing *R* functions to hide the details of the *REST* requests. We discuss *OAuth* in great deal in Chapter 13, including an example of accessing the Google Storage API.

While the explanation of how we get the token and cache it in a connection, and how we invoked the *REST* requests were lengthy, the short version of getting the authorized connection is quite simple. With the login and password set in the *GoogleDocsPassword* option, we use the command

```
con = getGoogleDocsConnection()
```

to obtain the connection. With that, we have access to our documents.

10.4.2 Managing Documents in Google Docs

Now that we have access to the Google Docs API, we can manage our documents. We assume that we have already created some documents, e.g., via the Web interface. What can we do with this collection from *R*? We can

1. retrieve the list of the documents, either by name or as objects that describe the details of each documents,
2. upload and create documents,
3. create a folder,
4. rename or move a document,
5. delete a document

Again, we leave discussion of how to access the contents of the documents to Chapter 15.

Listing the Documents: Titles and Objects
The function getDocs() in RGoogleDocs returns a list with an element for each document in the top-level Google Docs folder. We pass it our authenticated connection as

```
docs = getDocs(con)
```

We can see the names of the documents with

```
names(docs)
```

```
[1] "Testing"
[2] "OnCall"
[3] "OneColumn"
[4] "LLCP_VarLayout_11_OneColumn.RTF"
[5] "What is \"Data Science\" Anyway?"
[6] "Untitled document"
[7] "StringsFactors"
[8] "Untitled spreadsheet"
...
```

These names are just the names or titles we give the documents. They do not uniquely identify the document. We can have two documents with the same name/title. We want to think of each document as a resource with a *URL* associated with it. Indeed, there are several URLs associated with a given document, and different resources for different types of documents.

getDocs() returns more than just the names of the documents, but also a complete description of each document and its type. We can see the *R* classes of these document objects with

```
table(sapply(docs, class))
```

```
           GoogleDocument GoogleDocumentDescription
                        1                        39
             GoogleFolder        GooglePresentation
                       18                         3
       GoogleSpreadsheet
                       31
```

These are the names of (*S4*) classes we defined in *R* to represent and distinguish the different document object types. We use a common base class and then specialize this with sub-classes for the different document types, i.e., spreadsheet, presentation, word processing document, and folder. These classes correspond closely to the *XML* descriptions of the documents that Google Docs returns when we made the request in getDocs(). The slots in these classes include

```
slotNames(getClass(class(docs[[1]])))
```

```
 [1] "id"          "published"   "updated"   "category"    "title"
 [6] "content"     "alternate"   "self"      "edit"        "edit-media"
[11] "author"      "feedLink"    "access"    "connection"
```

We have information about when the document was published and subsequently updated (if at all), its title, the author, and a unique identifier for that document in the *id* slot. The *content*, *self*, *edit*, *edit-media*, *feedLink*, and *alternate* slots are all "links" to other resources associated with this document. These are URLs to access the document in different ways. For example, we can get the visible content from *content*, and we can modify the document using the *edit* link. A spreadsheet document has an additional link that identifies the URL for the individual worksheets within the workbook.

We can access the content of a document as an *HTML* document via the `src` element of the *content* slot, e.g.,

```
src = docs[["test"]]@content["src"]
hdoc = getURLContent(src, curl = con, followlocation = TRUE)
```

Note that we use the same authenticated connection so that we can access the document. We also added the curl option *followlocation*. A much better approach is to use getDocContent(). This hides the details of how we access the content and make the request, and also performs important error handling. We can simply pass it the document object and our connection:

```
getDocContent(docs[["test"]], con)
```

Although the `"test"` document is a spreadsheet, we retrieve an *HTML* file. RGoogleDocs has a set of functions for retrieving values from spreadsheets and converting this information into a data frame. How these functions that manipulate the *XML* representations of the documents are implemented is covered in detail in Chapter 15 in Part III.

Before we move on to other functions to manage the collection of documents, we examine how getDocs() and getDocContent() are implemented in terms of the *REST* operations. Both of these are very simple *HTTP* **GET** requests with no additional parameters in the request (other than the authorization field *HTTP* header). By default, getDocs() sends a simple request `https://docs.google.com/feeds/documents/private/full`. If we are working with the spreadsheet service (wise), it uses a different URL. It then converts the resulting *XML* document into a list of these document objects in *R*. getDocContent() is implemented as above with a call to getURLContent() using the *URL* given in the *content* slot of the object. This function allows the *R* user to specify the name/title of the document rather than the document object returned by getDocs(). We then have to call getDocs() to get the document object. This is convenient for the user, but it is quite expensive as it involves an additional request to Google Docs (and also processing descriptions of every document in the resulting *XML* when we only want one). When we write *R* functions to interface to *REST* services, we have to consider issues of network latency and caching results.

Adding and Uploading Documents

Now that we understand the concept of document objects in Google Docs and how we represent them in *R*, we can go further. We can use the *REST* interface to upload a document to Google Docs. The

"document" may be in a file or it may be an *R* object such as a data frame or matrix, which we will first convert to a CSV document and then upload that as a spreadsheet.

We can use the function uploadDoc() to perform the upload. We specify what to upload and the authenticated connection. When the *content* argument is a string and matches an existing file name, the function reads the contents of that file and uploads that. We want to tell Google Docs how to interpret the contents of the file, e.g., a spreadsheet or a CSV file. We do this via the *type* parameter. If this is not explicitly specified, the type of document (e.g., a spreadsheet, a Word document, a CSV file) is determined from the extension of the file name. We use a function (findType()) to match the extension with a MIME type from a table provided in Google's documentation. If the extension does not match any entry in this table, or if we are specifying the content directly and not in a file, we should specify a value for the *type* parameter explicitly. This can either be the MIME type, e.g., `text/csv`, `text/html` or `application/msword`, or alternatively, we can provide the corresponding extension, e.g., **csv**, **htm**, or **doc**. Note that Google cannot convert all types of documents and does not necessarily even handle "rich" CSV files.

As an example of uploadDoc(), suppose we have a CSV file named `data.csv`. We can upload it with

```
doc = uploadDoc("data.csv", con)
```

The function returns a `GoogleDocumentDescription` object or something more specific (e.g., a spreadsheet class). We can use this to query and manipulate the document just as we can objects returned by getDocs().

By default, the name/title of the new document in Google Docs will be the name of the file we uploaded, with the extension removed. We can, however, specify a new name for the document via the *name* parameter of uploadDoc(), e.g.,

```
doc = uploadDoc("data.csv", con, name = "sample data")
```

If the file we are uploading is a binary file rather than a text file, we can indicate this via the *binary* parameter, e.g.,

```
doc = uploadDoc("report.doc", con, binary = TRUE)
```

When the content is not in a file, but instead we have it directly in memory, say as a string, we can upload that directly without writing it to a file. For example, suppose we have a simple string containing data in CSV format, e.g.,

```
x = "1, 2, 3\n4, 5, 6\n"
```

We can upload that directly with

```
doc = uploadDoc(x, con, name = "my data", type = "csv",
                asText = TRUE)
```

Note that we explicitly indicate that the string is not the name of a file via the *asText* argument. Also, we specify the name for the document since we do not have a file name, and also we provide a value for the *type* parameter. Uploading content directly from memory suggests that we can upload an *R* data frame or matrix by first serializing it as CSV content and then uploading that. We do this by defining a method for the generic uploadDoc() function. We define this as

```
setMethod("uploadDoc", 'data.frame',
    function(content, con, name,
             type = as.character(findType(content)),
             binary = FALSE, asText = FALSE,
```

```
              folder = NULL, ...)
  {
    wcon = textConnection(NULL, "w", local = TRUE)
    on.exit(close(wcon))
    write.csv(content, file = wcon, row.names = FALSE)
    close(wcon); on.exit()
    uploadDoc(paste(textConnectionValue(wcon), collapse = "\n"),
              con, name, type = "csv", binary = FALSE,
              asText = TRUE, folder, ...)
})
```

Note that this creates the content as a string and then calls another method for uploadDoc() which actually uploads the content.

The method above shows that uploadDoc() has a parameter named *folder*. This allows the caller to specify that the new document should be uploaded into a particular folder. We can use the folder object returned from getDocs(), e.g.,

```
uploadDoc("mydata.csv", con, folder = docs[["Project1"]])
```

Here `Project1` identifies a document of class `GoogleFolder` in the list `docs`. As we see below, we can also create the folder with addFolder() and use the object it returns.

How does uploadDoc() work? It uses an *HTTP* **POST** request to send the contents of the document being uploaded. If we are uploading a file, it reads the contents into a vector in *R*. For a binary file, this is a *raw* vector; otherwise it is a *character* vector. This does not really matter since we pass this vector to the *HTTP* request via the curl option *postfields* which takes care of the different vector types. The *URL* for the upload request is either the top-level collection of documents, or the *URL* for the target folder, if the caller provides one.

In addition to uploading local files and content from *R*, we can create an empty spreadsheet within *R* and then populate it using functions we will discuss in Chapter 15. We create the spreadsheet with the addSpreadsheet() function, giving it the number of rows and columns for the sole worksheet. This function then uses the uploadDoc() function to add the document to our Google Docs connection. This illustrates how we can build higher-level functions via the fundamental functions for the interface.

Creating Folders

In addition to listing and uploading documents, the Google Docs API provides functionality to create folders and the *R* function addFolder() hides the details. We can give it the name of one or more new folders to create and an authenticated connection and it creates each of the folders in the top-level document collection. For example, we create two new folders at the top-level of the collection with

```
f = addFolder(c("Project1", "Project2"), con)
```

When we call getDocs() again, these new folders will appear. The value returned by addFolder() is either a single `GoogleFolder` object or a list of such objects with an element for each folder. We can use these objects in subsequent calls, e.g., to move a document to the new folder with moveToFolder().

The addFolder() function also allows us to specify the name of a new folder as a regular file path, e.g., `Project1/data`. This will create each folder within the path, as necessary. This involves testing to see if it already exists, which involves a request to Google Docs. This can be expensive in terms of time.

The implementation of the addFolder() function is a little more interesting than getDocs() or getDocContent(). Here, we are not querying the current value of a resource, but creating a resource.

We use a **POST** request to the URL for the top-level document collection, i.e., `https://docs.google.com/feeds/documents/private/full`. We use a **POST** command since Google Docs decides the detailed *URL* of the new folder, while we only specify its title. We have to specify information about the new folder, namely its title and that it is a folder rather than another type of document. We provide this information in the body of the **POST** request as an *XML* document. We create it to look something like

```
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>Project1</title>
 <category scheme="http://schemas.google.com/g/2005#kind"
   term="http://schemas.google.com/docs/2007#folder"
   label="folder"/>
</entry>
```

After we submit the *HTTP* request, we process the result and check for an error. If the request was successful, we convert the *XML* description of the new folder into an *R* `GoogleFolder` object.

Moving a Document to a Folder

We can move any document or folder into another folder with moveToFolder(). This takes the document to move and the target folder and performs the request, e.g.,

```
ndoc = moveToFolder(docs[["test"]], f$Project1)
```

This returns a potentially updated document object.

Note that we did not need to provide the authenticated connection in the call. This is because we added the connection to each document object when we created it. This allows them to be self-describing, i.e., we are able to make requests with that object without additional information. This can be a convenient idiom to use when creating an *R* interface to an API. Furthermore, we also overload the `$` operator for the connection so that we can make requests such as `con$getDocs()` rather than `getDocs(con)`. This is just syntactic sugar.

The moveToFolder() function is implemented via a **POST** request to the URL of the target folder's *content* slot. The body of the **POST** is an *XML* document that contains the details of the document to be moved, e.g., its *id* and *category* slots.

Changing Metadata on a Document

Note that when we moved a document into a different folder above, we did not need to specify the name of the document in the new folder. This may seem rather obvious since the same thing happens when we move a file on our hard drive to a folder. However, the reason for this is quite different for Google Docs documents. The name or title of a Google document is an attribute of the document itself, not the container in which it is located. The title will appear when we list the contents of the folder since it is extracted from the document. If we do want to change the name/title of the document, we can change it on the document itself. We do this by setting metadata on the document object in *R*. For example,

```
doc$title = "A new name"
doc["title", "author"] = list("A new name",
                              c(name = "me",
                                email= "me@gmail.com"))
```

When we list the documents, the new title will be used for that document.

Again, we have hidden the *REST* details with *R* (assignment) methods. The actual request is a **PUT** request that sends the changes/updates as an *XML* document.

Removing a Document

We can remove a document from Google Docs with the deleteDoc() function. We can pass it the document description object or the name of the object and the connection, e.g.,

```
deleteDoc(docs[["test"]], con)
deleteDoc("test", con)
```

When we update the list of documents, either in our Web browser or with getDocs(), that document will no longer be present.

As we might expect, the deleteDoc() function is implemented as a **DELETE** request. The *R* function is a generic function with methods for different types of inputs. We can provide the name/title as a *character* string or the `GoogleDocumentDescription` object returned by, e.g., getDocs(). We can also pass the object returned by getDocs() which is of class `GoogleDocList`. This would remove all the documents. However, when we subset a `GoogleDocList`, we get a new `GoogleDocList` object. This makes it easy to remove multiple documents in a single call, e.g.,

```
docs = getDocs()
deleteDoc(docs[ grep("^Project", names(doc)) ])
```

removes all documents where the name starts with `Project`.

There are many other tasks we can perform within *R* on our collection of Google Documents. We can change the permissions on a document to publish the document to everyone or share it with a few collaborators. We can search documents for particular phrases or values. All of these are done through *HTTP* requests, which are similar to what we have seen above and are quite simple.

10.4.3 Using an Access Token to Digitally Sign Requests

We now turn to an example that requires slightly more complex use of an access token to digitally sign requests. The company Amazon offers the Simple Storage Service—S3 for short—for hosting files and allowing them to be accessed from any machine on the Internet. In some respects, this is similar to Dropbox. However, Amazon also provides a cloud computing service with which we can reserve one or more computers (with different hardware and software characteristics) and use these to perform often intensive computations. This computing service and the S3 file system(s) are tightly integrated and we can use S3 to locate the data where they can be used in the cloud. We can even choose to store files at particular data centers across the world so that they are closer to where we will use them. (Note that throughout this section, S3 stands for Amazon S3, not the *S3* class mechanism in *R*.)

In the next example, we look at how we can use the S3 Web service from within *R*. We briefly outline the basic functionality the service provides and discuss how we implement the *REST* requests in *R*. These are actually provided by the RAmazonS3 package.

Example 10-6 Digitally Signing REST Requests to Amazon S3

Before we look at the functionality of the S3 service, it is useful to understand the basic computational/organization model. We store a file in a bucket. The bucket is analogous to a folder or directory. However, we cannot nest buckets as we can folders. In other words, we do not have buckets within buckets, and so there is no concept of a path or hierarchy. We can emulate paths within S3, but for our example, we focus on the core capabilities, which is working with files in top-level buck-

ets. We store a file in a bucket with a name we provide. We can only access a bucket or a file if we have the correct permissions. These are controlled via an Access Control Lists (ACL). These are similar to the standard file permissions in UNIX, but slightly richer. More general information about S3 is available at `http://aws.amazon.com/s3/`. Details of the *REST* interface are at `http://docs.amazonwebservices.com/AmazonS3/latest/API/APIRest.html`.

The set of operations we can perform with S3 is similar to that of managing the documents in the Google Docs API. Of course, the Google Docs API also allows us to modify the structured content of the documents, while S3 is just for storage and does not provide facilities for modifying the contents within a file. We can

- list all of the buckets available to us (listBuckets()),
- list the contents of a particular bucket (listBucket()),
- retrieve (the contents of) a file (getFile()),
- upload a file to a bucket (addFile()),
- set metadata for a file, e.g., its Content-Type,
- copy a file to a bucket (copyFile() & renameFile()),
- remove/delete a file (removeFile()),
- create a new bucket (makeBucket()),
- delete a bucket (removeBucket()),
- get & set permissions/ACLs on a file or bucket (getS3Access() and setS3Access()).
- get the geographical location of a file (getBucketLocation())

Here the *R* functions in the RAmazonS3 package that correspond to these operations are provided in parentheses. This package provides an *R* interface that hides the details of these operations and uses idioms similar to RGoogleDocs to make them more flexible and easier to use. For example, we allow content to be loaded from a file or from in-memory objects (using I() for the `AsIs` class). We allow file names to be specified as `bucket/file`, as a vector of two elements, or as separate arguments. We also build on these operations to provide higher-level functions to, for example, save *R* objects directly to a bucket and allow it to be load()ed directly back into an *R* session. We also allow a bucket in S3 to be treated as if it were a *list* in *R* with a length and names for the elements corresponding to the names of the files in the remote bucket. Subsetting and element-assignments work transparently.

The S3 API is one of the earliest in the history of modern Web services. Indeed, it helped to define the *OAuth* 1.0 approach and standard. Unlike Google Docs, we do not need to acquire an access token by sending our login and password. Instead, we are given a "secret" when we register with the service via a Web page and submit information so that we can be charged for our use. The secret is like the access token we were given for Zillow or NOAA; however, we use it in a quite different way. Like these other Web services, we do include an *Authorization* field in the *HTTP* request. However, we do not send the secret. Instead, we keep this secret private and use it to both digitally sign the contents of the request and to include that signature in the *HTTP* header. The S3 server then verifies that only somebody with our particular secret token could have generated the authorization signature for the given request and its contents.

Consider how we implement the function addFile() to upload a file to a bucket. We upload a simple string as the content to a bucket named `"dtl-test"` and name the new file `"myTest"` with

```
addFile(I("This is a test"), "dtl-test", "myTest",
        access = "public-read",
        meta = c(author = "DTL", version = "1.2"))
```

We have made the document readable by everybody and we have provided metadata giving the author and a version identifier. The addFile() function uses a **PUT** request to create the new resource, i.e., the

file in the bucket. It sends the request to the URL `http://dtl-test.s3.amazonaws.com/myTest` which uses the bucket name and the name of the file.

addFile() uses the content as the body of the **PUT** request. If the content is a binary vector, i.e., not text, we convert it to text using what is called base64 encoding. We indicate the type of the content via the *Content-Type* field in the header of the request.

In addition to the content type, we also provide any other details about the file and also the request in the *HTTP* header. We add the metadata for the file via fields in the header named *x-amz-meta-author* and *x-amz-meta-version*. We add the ACL information with the field *x-amz-acl*. We also add a *User-Agent* field and, very importantly, specify the date and time of the request. This is used by the server to verify that the request is not being resent much later than it was originally composed, e.g., by a third party who intercepted the request.

Having constructed the header and the body of the request, the last step is to sign this request. To do this, we need to know all of the details of the request including the operation (**PUT**), the target *URL*, the fields in the header, the date and time (available in the header), and the contents of the body. The function S3AuthString() hides the details of this operation, which ensures that the server can verify the signature and the request. Importantly, the function needs our signature so that it can sign the request. We call the function, in the case of addFile(), as

```
header = S3AuthString(auth, "PUT", bucket, name, h,
                      md5 = md5(contents, isFile = FALSE))
```

This returns our entire *HTTP* header with the *Authorization* field added and containing the digital signature based on our secret. It is something similar to

```
                              Date
    "Wed, 28 Nov 2012 11:18:18 PST"
                        User-Agent
            "RAmazonS3, R 2.14.2"
                x-amz-meta-author
                             "DTL"
               x-amz-meta-version
                             "1.2"
                      Content-Type
                      "text/plain"
                              Host
       "dtl-test.s3.amazonaws.com"
                        x-amz-acl
                    "public-read"
                    Authorization
"AWS AKIAJI......0/wxLJcUs+cbqjJM="
```

We can then use this as the value of the *httpheader* option for our call to httpPOST().

The S3 service processes the request and either returns successfully with information about the result as an *XML* document, or raises an error. As usual, the error may be an *HTTP* error and so will be raised by httpPOST(). Alternatively, it may be an error in the S3 request such as a problem with not having permission to write to that bucket. We need to process the *XML* document to find this and then raise an appropriate error, using a special class for that type of error.

Amazon has many other APIs for which we can develop *R* interfaces in a similar manner to RAmazonS3. For example, the Amazon DB service provides a simple key-value database mechanism. The RAmazonDBREST package [9] provides access to the service.

Services such as S3 are widely used and there are interfaces to access it for many different programming languages. In some cases, such as S3, there are *C* libraries that we can integrate into *R*. This is valuable since often they will be more widely used than *R*-specific code. Similarly, there may be a wider community supporting, maintaining and extending the code. We can bring the code into *R* either manually, or using code generation techniques such as offered by the RGCCTranslationUnit package [8]. We can use the .Call() mechanism with wrapper functions, or more direct and dynamic foreign function interfaces (FFIs) such as available in the `rdyncall` [1] and Rffi [10] packages. One advantage of developing the interface directly with *R* code is that it is easier to understand, debug, and extend. There is always a trade-off between these approaches of developing our own code and using general-purpose code. Ideally, we have choices rather than having to use one or the other.

10.5 Web Application Description Language: *WADL*

When we use a *REST* Web service, there is a typical sequence of events. We read the documentation for the different methods and then create a corresponding *R* function for each of these methods. Each function maps the *R* inputs to the *REST* parameters, submits the *HTTP* request, and then processes the result—typically either *JSON* or *XML*—back to an *R* object. We may have to use some form of authentication such as a passing an application identifier as a parameter or in the *HTTP* header, or use *OAuth* (1.0 or 2.0) to sign the contents of the request. The key aspect is that a human has to read the documentation and create the *R* function(s). In contrast, *SOAP*-based Web services typically provide a *WSDL* (Web Service Description Language) document that fully describes each available method and its data types. From the *WSDL*, we can programmatically generate all of the *R* functions to interface to the corresponding *SOAP* methods, and also define *S4* classes corresponding to the nontrivial data types. While *SOAP* may be more complex in some ways, being able to programmatically generate the interface makes it much simpler and more correct than having to read documentation and create the functions ourselves. For this reason, there have been efforts to create a similar facility for describing *REST* Web services with a description language. There is slow movement to using these, but in this section we describe two approaches, focusing on *WADL* documents. We also discuss strategies for mapping complex resource names to *R* function names and how to group methods and resources into a single *R* function.

10.5.1 Reflection Methods for REST Methods and Services

The Flickr API contains two reflection methods that allow us to find the names of all the available methods and then to query the details, i.e., inputs and outputs. We can then use this information for each method to generate a corresponding *R* function. For example, we can get a list of all of the methods in the Flickr API with

```
txt = getForm("http://api.flickr.com/services/rest/",
              method = "flickr.reflection.getMethods",
              api_key = getOption("flickr_api_key"))
```

where we use a token/key given to us when we register with the Flickr API. The result of our request is an *XML* document containing the names of the methods and we can convert it to an *R* object with

```
methods = unlist(xmlToList(I(txt), FALSE))
```

(We can also arrange to have the result returned as *JSON* using the `format` parameter, and then convert these with fromJSON().)

With the list of method names, we can retrieve the details for each method via the flickr.reflection.getMethodInfo method. For example, to get a description of the flickr.places.findByLatLon method, we can use

```
txt = getForm("http://api.flickr.com/services/rest/",
              method = "flickr.reflection.getMethodInfo",
              method_name = "flickr.places.findByLatLon",
              api_key = getOption("flickr_api_key"))
```

The result is an *XML* document (see Figure 10.1) that provides both documentation for the method that humans can read, and structured information about the expected inputs and outputs from the method, including possible errors. The <*arguments*> node contains an <*argument*> element for each of the inputs, including its name and whether it is optional or required. From this, we can generate simple *R* functions.

This approach of programmatically obtaining descriptions of methods and generating *R* functions is a significantly better approach than having to read the documentation for all 206 methods and correctly create *R* functions "manually." There are, however, a few problems. The format of the description of each method is not standardized. Also, how do we find the reflection methods in the first place? We would like to have a standard format for describing a collection of related Web services and their inputs. Ideally, we would be able to find this description in a searchable catalog, but we would like, at least, to be able to retrieve it via a *URL* associated with the Web service. This is where a *WADL* document can play an important role.

10.5.2 Working with WADL Documents

A *WADL* document uses *XML* to describe a Web service. The basic structure of the document is illustrated in Figure 10.2. At its simplest, each resource has a corresponding method. Each input to the method is described within a <*param*> node within the <*request*> node of the <*method*> element. This has the name and type of the parameter, along with other information about how to interpret it. The <*method*>'s parent <*resource*> element indicates the (relative) path for the request. This is interpreted relative to the base *URL* defined in the <*resources*> element. The *name* attribute on the <*method*> node identifies the *HTTP* operation used for making the request. The complete format of a *WADL* document is defined by the W3C submission/proposal and can be found at `http://www.w3.org/Submission/wadl/`.

Since a *WADL* document is written in *XML*, we can read it and extract the information in *R* using the XML package. The WADL package [12] does this. The function wadlMethods() reads the document and returns a list of method descriptions. The function makeFunctions() then takes these descriptions and generates the code for the corresponding *R* functions. We can control how the code is generated, which methods are processed and even modify the *WADL* document before it is processed. We can have makeFunctions() create the code as text so that we can use it in a package or source() it into a different *R* session. Alternatively, we can define the functions as regular *R* functions immediately and directly from the descriptions, putting them in the global environment or a list or whatever container we chose, e.g., a different environment. In addition to creating the

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method name="flickr.places.findByLatLon"
  needslogin="0" needssigning="0" requiredperms="0">
 <description>Return a place .... points.</description>
 <response>&lt;places
   latitude=&quot;37.76513627957266&quot;
   longitude=&quot;-122.42020770907402&quot;
   accuracy=&quot;16&quot; total=&quot;1&quot;&gt;
   &lt;place place_id=&quot;Y12JWsKbApmnSQpbQg&quot;
   woeid=&quot;23512048&quot;
   latitude=&quot;37.765&quot;
   longitude=&quot;-122.424&quot;
   place_url=&quot;/United+S.../Mission+Dolores&quot;
   place_type=&quot;neighbourhood&quot;
   place_type_id=&quot;22&quot;
   timezone=&quot;America/Los_Angeles&quot;
   name=&quot;Mission Dolores, San Fran...&quot;/&gt;
&lt;/places&gt;</response>
</method>
<arguments>
 <argument name="api_key" optional="0">
  Your API application key. ...  details.</argument>
 <argument name="lat" optional="0">
  The latitude whose valid range....</argument>
 <argument name="lon" optional="0">
  The longitude whose valid ... truncated.</argument>
 <argument name="accuracy" optional="1">
  Recorded accuracy ... default is 16.</argument>
</arguments>
<errors>
 <error code="1" message="Required arguments missing">
One or more required ..  with the API request.</error>
 <error code="2" message="Not a valid latitude">
The ..n.</error>
    ...
 <error code="116" message="Bad URL found">
One or more ... on Flickr.</error>
</errors>
</rsp>
```

Figure 10.1: *XML* Description of a Flickr Method. This shows a truncated *XML* document describing the Flickr API method named flickr.places.findByLatLon. There is a human-readable description of the method and sample output. The <*arguments*> node gives a description of each accepted input and whether it is optional or not. The <*errors*> node describes the possible errors that can arise when calling this method and the associated status code.

```
<application xmlns="http://wadl.dev.java.net/2009/02">
 <grammars><!-- Data type definitions --></grammars>
 <resources base="Base URL">
  <resource path="relative URL">
   <!-- Any template parameters that we substitute into URL -->
   <param name="parameter name" style="template" type="xs:string"/>
   <method name="GET">
    <request>
     <param name="parameter name" type="..." style="query"/>
           ...
    </request>
   </method>
  </resource>
  <resource path="other URL">
   <method>
   </method>
  </resource>
 </resources>
</application>
```

Figure 10.2: Example *WADL* Document. This illustrates the basic structure of a *WADL* document. The resources and methods provided by the Web service are described by listing the input parameters and their types, and the possible results. Data types can be defined within the `<grammars>` section.

functions, we can take the human-readable description of the methods from the *WADL* and generate help pages for the *R* functions we create. We do this with the makeWADLDocs() function.

These functions typically just work and generate the *R* functions without us having to specify any additional information. However, we can customize how they behave in various different ways and different stages of the processing. In the remainder of this section, we look at two more interesting and challenging examples of using the WADL package. The first provides an interface to the bioinformatics gene search facility at EuPathDB.org. The second revisits the NOAA Web service we discussed earlier in Example 10-4 (page 354). Both illustrate different aspects of the *WADL* and how we generate *R* functions. In both cases, we can customize the automated interface to make it more convenient for *R* users, and we discuss how to do this.

Example 10-7 The EuPathDB Gene Search Web Service

The EuPathDB Web service allows us to search for various genetic entities, such as genes, SNPs, and EST, that match different criteria, e.g., identifier, location, molecular weight and so on. The entire list of methods is available at `http://eupathdb.org/eupathdb/serviceList.jsp`. Rather than having a single *WADL* document, the EuPathDB provides a separate *WADL* document for each resource and method, and also a *WADL* document for each group of related methods. We can read each of these *WADL* documents describing the collection of related methods and then generate *R* code for those methods. We can retrieve these files from the EuPath documentation page, but we have downloaded all of the *WADL* files to a local directory. We find the names of the "grouped" *WADL* documents with

```
wadls = list.files("WADL", pattern = "Questions",
                   full.names = TRUE)
```

Now we can process each of these to generate the *R* functions. We start by reading all the methods in a given *WADL*:

```
methods = wadlMethods(doc)
```

The names of the methods —

```
[1] "http://eupa...ice/EstQuestions/EstBySourceId.xml"
[2] "http://eupa...ice/EstQuestions/EstBySourceId.json"
[3] "http://eupa...ice/EstQuestions/ESTsByGeneIDs.xml"
[4] "http://eupa...ice/EstQuestions/ESTsByGeneIDs.json"
....
```

illustrate that there are two separate methods for each resource. There is a *JSON* and an *XML* version for each method and this indicates the format of the results returned by each. We can generate separate functions for each version, e.g., ESTsByGeneIDs.json() and ESTsByGeneIDs.xml(). However, it is more *R*-like to have a single function for each pair of methods and allow the caller to specify the desired format via a parameter, e.g.,

```
ESTsByGeneIDs(..., .json = FALSE)
```

We will arrange for our functions to convert the result accordingly, using fromJSON() if the caller requested the results as *JSON*. In order to do this, we process only the *XML* versions of the methods and add an additional parameter when generating the functions. We subset the methods and change the names to remove the format extension:

```
methods = methods[grep(".xml", names(methods))]
names(methods) = gsub(".xml", "", names(methods))
```

Now we can create the *R* functions via a call to makeFunctions(). For some reason, the base *URL* in the *WADL* documents is incorrect. Instead of being `http://eupathdb.org/webservices/`, it should be `http://eupathdb.org/eupathdb/webservices/`. We can change this after parsing each of the *XML* documents. However, we can use makeFunctions()'s *rewriteURL* parameter to specify patterns for changing the URLs for the requests. So we can create the functions with

```
makeFunctions(methods = methods, eval = TRUE,
              rewriteURL = c("(/webservices/)",
                             "/eupathdb/webservices/"),
              hooks = list(makeSignature = mySig,
                           postCall = convertJSONCode))
```

By passing `TRUE` for *eval*, we are asking *R* to create and define the functions in the global environment and not just to return the functions as text.

The more complicated input to makeFunctions() is the *hooks* argument. This is a list of functions that makeFunction() calls when creating different parts of the code for each function we generate. The `makeSignature` element is called when we create the signature (collection of formal parameters) for the function. We use this to add our *.json* parameter to control the format of the result. The `postCall` element of *hooks* is added to our generated function after the *HTTP* request has been completed. We use this to add code to perform the conversion of the result from *JSON*, if this is what the caller wants. We can also use the `preCall` element of *hooks* to add code to perform any manipulation of the inputs before the *HTTP* request. For example, we might coerce the inputs to strings, or match abbreviated gene identifiers to their full form expected by the Web service.

Our mySig() function is defined as

```
mySig =
function(ids, defaultValues, extras, name, url)
```

```
{
  ans = makeSignature(ids, defaultValues, extras, name, url)
  ans[".json"] = ".json = TRUE"
  ans[".url"] = sprintf(".url = if(.json)
                                 '%s.json'
                               else
                                 '%s.xml'", url, url)
  ans
}
```

This function is called with the names of the *REST* method's parameters and their default values, any extra parameters, and the name of the function and *URL* for the request. We call the regular function makeSignature() to create the usual signature from the parameter's methods. This returns a character vector describing each of the parameters for the function, and then we add two additional parameters, along with the default values. We use text here rather than *R* language objects (e.g., expressions and calls) which we use in the XMLSchema [18] and SSOAP and other packages that programmatically create *R* functions.

We define the function convertJSONCode() function we use as the `postCall` hook with

```
function(params, url, name)
  "if(is.null(.convert) && .json)
     return(fromJSON(ans))"
```

This just returns a string giving the verbatim code that converts the body of the *HTTP* response to an *R* object using fromJSON().

After we create these functions, we can then call any of them. For example,

```
organism = "Cryptosporidium parvum,Leishmania major"
tmp = GenesByMolecularWeight(organism, 10000, 50000, .json = FALSE)
```

This returns the result as *XML*. We can convert it ourselves. Alternatively, we can specify a function for the function's *.convert* parameter. If there is a consistent way to interpret the results, we can add that as the default value for *.convert* or insert it as code into the function via the `postCall` hook.

We now move onto a different example, revisiting the NOAA Web services. Here we consider how to map the nested resource URLs and resources to a convenient *R* function or multiple functions.

Example 10-8 A WADL Interface to the NOAA Web Service

We start by reading the *WADL* document:

```
w = wadl("GitWorkingArea/WADL/inst/sampleWADLs/noaa.wadl")
```

This contains the parsed *XML* document.

The authentication token argument we need to provide in each request is not enumerated in the *WADL* document's methods. We need to add this to the description of each method. We can do this in the *XML* document or we can do this in the description objects returned by wadlMethods(). We can do this in the *XML* document with

```
rq = getNodeSet(w@ref, "//w:resource/w:method/w:request",
                c(w = "http://wadl.dev.java.net/2009/02"))
lapply(rq, function(node)
             newXMLNode("param",
```

```
                    attrs = c(name = "token",
                              style = "query",
                              type = "xs:string"),
                    parent = node))
```

Now that we have augmented the descriptions, we compute *R* descriptions of each method using wadlMethods():

```
methods = wadlMethods(w)
```

Alternatively, if we choose instead to add the parameter to the description objects returned by reading the original *WADL* document, we can do this with

```
m = wadlMethods("noaa.wadl")
m = lapply(m, function(x) {
                x[ nrow(x) + 1L, c("name", "type", "style")] <-
                            c("token", "xs:string", "query")
                x
          })
```

Either approach works equally well.

The final step in creating the *R* functions is to call makeFunctions(), passing it the method descriptions:

```
makeFunctions(w, m, eval = TRUE)
```

One of the problems with this approach is that the names of the functions come from the resource's *path* attribute. These are names such as `http://www.ncdc.noaa.gov/cdo-services/service/datasets/{dataSet}/datatypes`. We certainly do not want to use these as function names in *R*. We can change the names used by *R* by explicitly specifying a vector of names via makeFunctions()'s *funcNames* parameter, or by changing the names on the list of methods, i.e., the variable `m`.

While we know how to specify different names for the *R* functions, the more difficult issue is what should we name them. We can remove the common prefix of the base *URL* (`http://www.ncdc.noaa.gov/cdo-services/service/datasets/`) and so end up with names such as `{dataSet}/datatypes`. These are still not good names for *R* functions. We might replace each of the `'/'` characters with a `'.'` or a `'_'`, e.g., and remove the { and }. So the function names would be, e.g., `dataSet.datatypes` and `dataSet.locations.location`. This is better but still makes it difficult for the user to know the name of the function to call. A better approach might be to have a single general function that determines which inputs the caller provides and then determines the corresponding function. The WADL package can generate such a function, recognizing the template parameters. The function would look something like

```
noaa =
function(dataSet = NA, location = NA, dataType = NA,
         dates = NA, station = NA, ..., .funcNames, .append = "")
{
  missings = c(dataSet = missing(dataSet),
               location = missing(location),
               dataType = missing(dataType),
               dates = missing(dates),
               station = missing(station))
```

```
  fun = matchTemplateFunction(!missings, .funcNames, .append)
  args = list(...)
  other = list(dataSet = dataSet, location = location,
               dataType = dataType, dates = dates,
               station = station)[!missings]
  names(args)[names(other)] = other

  do.call(fun, args)
}
```

The important steps are identifying which of the template parameters were supplied (and which were missing) and using these in the call to matchTemplateFunction() to find the best matching function. For example, if we call our noaa() function and specify values for *dataSet* and *station*, this matches the method or URL `http://www.ncdc.noaa.gov/cdo-services/service/datasets/{dataSet}/stations/{station}`. The noaa() function then takes these template arguments, and any other arguments for the *REST* method (via the ...), and makes a call to that remote resource.

One problem with attempting to match the method using template parameters is that we cannot distinguish between, for example, `{dataSet}/stations/{station}` and `{dataSet}/stations/{station}/data`. The first returns metadata about the specified station; the second returns the actual data for a given station. The caller of the noaa() function can cause the second method to be called by using the *append* parameter. We specify the trailing part of the *URL* as "data" and the matchTemplateFunction() adds this when searching for a matching function.

The number of Web services that use *WADL* or any description language to describe APIs is small, certainly compared to the use of *WSDL* documents for *SOAP* APIs. There are several reasons for this. Firstly, manually creating client interfaces to *REST* APIs is simpler than for *SOAP*, so the work may be beneath the threshold of *needing* an automated approach. Secondly, many of the *REST* advocates claim that *WADL* does not reflect the essential ideas of *REST* in several different ways. This is unfortunate. Whether it be *WADL* or another approach to describing *REST* services, the benefits of having metadata about Web services and the resulting ability to programmatically generate client (and server) code for different languages are very significant. We can then focus on using the services. Version 2.0 of the *WSDL* specification allows it to be used to describe *REST* services, and perhaps this will encourage more widespread adoption.

When the owners of a *REST* Web service do not describe the API with a *WADL* document, we can of course develop *R* functions manually. However, it may be better for us to take the time to write the descriptions of the methods of the current API and then generate the code programmatically from these descriptions. We do not need to do this using a *WADL* document. Instead, we can use *R* objects such as those returned by wadlMethods() to describe the services and the data types. This, however, then limits the descriptions to being used in *R*. If we developed a *WADL* document for the service, people could use it in other programming languages and hopefully help to identify any errors or omissions in the descriptions. While developing descriptions in order to generate code is less direct, it can be valuable. If the code we need to generate is very complicated and specialized and not generated by the functions in the WADL package, then it may be easier to write the code manually. However, for Web services, the code is typically very generic and common across APIs. In these cases, programmatically generating code saves time and reduces errors. Furthermore, as the API

changes, we can adapt the description and regenerate the entire code without having to change each function.

10.6 Possible Enhancements and Extensions

Implement *REST* APIs
Any *R* programmer can develop an *R* package to provide an interface to any Web service or API. There are so many existing *REST* APIs, and new ones are emerging rapidly. The site `programmableweb.com` maintains a directory of known APIs, queryable by category. There are many scientific and government agencies making data available via Web services which would be useful to have access to from *R*. The rOpenSci project aims to create *R* packages for accessing many different scientific Web services. There are many social networking Web services or APIs. Similarly, there are many services such as cloud computing and storage that we can utilize from within *R*. There are many applications that we run locally rather than centralized Web services. For example, ElasticSearch and DynamoDB provide facilities we do not have in *R*, but which can be leveraged for new types of analyses.

Anyone can help with this effort. You can program interfaces, create *WADL*s and generate interface code or just identify interesting *REST* services and applications.

Enhance Existing Infrastructure
The WADL package works adequately for most *WADL* documents since the specification is quite simple. However, the package does not implement all features that are possible in a *WADL*. We encourage people to fork the project on Github and to either suggest patches or to volunteer taking over the development of new facilities. Similarly, the facilities for reading *WSDL* documents and, to a lesser extent, *XML* schema that occur in *WADL* and *WSDL* documents is adequate but not comprehensive or complete. Additions to read *WSDL* version 2.0 documents and generate code would be very valuable.

10.7 Summary of Functions for *REST* in *R*

In this chapter, we explored many examples that use *REST* services, e.g., Google Docs, Amazon S3, and CouchDB. Most of these examples have been wrapped up into *R* packages, e.g., RGoogleDocs, RAmazonS3, and `R4CouchDB`, respectively. Given the flexibility and variation in *REST* services, we do not provide a summary of the functions for these specific applications, but instead focus on the lower-level facilities needed to access *REST* services in *R*.

One of the advantages of *REST* is that it heavily leverages *HTTP*/*HTTPS* and so we need only be familiar with the functionality in the RCurl package (see RCurl and Chapter 8) to start working with *REST*. In many cases, we just send and receive simple inputs and outputs as strings. However, the content of both the request and response often use *XML* or *JSON* and again, we use existing functionality from either the XML or RJSONIO packages. As a result, the *REST*-specific functions are those related to reading the *WADL* files and generating interfaces to *REST* services programmatically. We describe the basic functions for generating the code from a *WADL* document below. There are other run-time functions in the WADL package that can be useful if one creates customized functions, e.g., checkValues() to check if a value specified by an *R* user is valid as an enumeration type.

wadlMethods() Create descriptions of the *REST* methods from a *WADL* document or `WADL` object. These descriptions are returned in a list with one element per method. Each method is represented by a data frame with a row describing each parameter—its name, type, whether it is required, etc.

makeFunctions() Generate *R* functions from the descriptions of the *WADL* methods returned by wadlMethods(). We can then use any of these functions within our *R* session to invoke the corresponding *REST* method on a remote server. The *methods* parameter controls which methods are to be converted into *R* functions, and the *eval* parameter provides the environment in which these functions are defined or a logical to indicate the global environment (`TRUE`) or to return the code as text (`FALSE`). There are various ways to customize how the *R* code corresponding to a *WADL* method is generated. We can provide our own function in place of makeFunction() via the *makeFun* parameter. If we use makeFunction() to create the code, we can customize it via a *hooks* parameter. This takes a list of functions to call when creating different parts of the code for each function, e.g., `makeSignature` for creating the signature of the function and `postCall` for after the *HTTP* request has been completed. In addition to overriding makeFunction() or customizing it, we can control how makeFunctions() computes the *URL* for each method. This allows us to easily redirect the requests to a different server.

makeWADLDocs() Produce *R* help pages for each of the *R* functions we have generated from a *WADL* document, using the documentation in the *WADL* file.

10.8 Further Reading

There are many resources available to learn more about *REST*. We have found [6] to be particularly useful. In addition, *WADL* is described briefly in Chapter 2 of [6].

References

[1] Daniel Adler. `rdyncall`: Improved foreign function interface (FFI) and dynamic bindings to *C* libraries (e.g., *OpenGL*). `http://cran.r-project.org/package=rdyncall`, 2012. *R* package version 0.7.5.

[2] J. Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[3] Thomas Bock. `R4CouchDB`: Collection of *R* functions for CouchDB access. `https://github.com/wactbprot/R4CouchDB`, 2011. *R* package version 0.08.

[4] Jeroen Ooms. `opencpu.encode`: Encodes *R* objects to a standardized *JSON* format. `http://cran.r-project.org/web/packages/opencpu.encode/`, 2012. *R* package version 0.22.

[5] *R* Core Team. *Condition Handling and Recovery*, 2012. `http://stat.ethz.ch/R-manual/R-patched/library/base/html/conditions.html`.

[6] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., Sebastopol, CA, 2007.

[7] Wendell Santos. 195 science APIs: Springer, EPA, and NCBI. `http://blog.programmableweb.com/2012/03/28/195-science-apis-springer-epa-and-ncbi/`, 2012.

[8] Duncan Temple Lang. RGCCTranslationUnit: *R* interface to GCC source code information. http://www.omegahat.org/RGCCTranslationUnit, 2009. *R* package version 0.4-0.

[9] Duncan Temple Lang. RAmazonDBREST: *REST*-based interface to Amazon's SimpleDB. http://www.omegahat.org/RAmazonDBREST, 2011. *R* package version 0.1-1.

[10] Duncan Temple Lang. Rffi: Interface to libffi to dynamically invoke arbitrary compiled routines at run-time without compiled bindings. http://www.omegahat.org/Rffi, 2011. *R* package version 0.3-0.

[11] Duncan Temple Lang. RJSONIO: Serialize *R* objects to *JSON* (*JavaScript* Object Notation). http://www.omegahat.org/RJSONIO, 2011. *R* package version 0.95.

[12] Duncan Temple Lang. WADL: Programmatically process Web Application Description Language documents. http://www.omegahat.org/WADL, 2011. *R* package version 0.2-0.

[13] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[14] Duncan Temple Lang. Zillow: Simple interface to Zillow.com's house price estimate API. http://www.omegahat.org/Zillow, 2011. *R* package version 0.1-1.

[15] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.

[16] Duncan Temple Lang. RGoogleDocs: Primitive interface to Google Documents from *R*. http://www.omegahat.org/RGoogleDocs, 2012. *R* package version 0.7-0.

[17] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.

[18] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.

[19] Duncan Temple Lang and Roger Peng. RAmazonS3: *R* interface to Amazon's S3 storage. http://www.omegahat.org/RAmazonS3, 2011. *R* package version 0.1-5.

[20] Eric van der Vlist. *XML Schema*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[21] Zillow, Inc. Zillow: A free online real estate marketplace. http://www.zillow.com/, 2012.

Chapter 11
Simple Web Services and Remote Method Calls with XML-RPC

Abstract In this chapter, we look at one of the early approaches used for Web services: XML-RPC. This is a form of remote procedure call, i.e., invoking functions that reside in a different process or machine. This is basically calling a "foreign" function. Like *REST*, this uses *HTTP* to communicate the requests and responses. However, XML-RPC uses *XML* to represent the different inputs/arguments to the request, and the value of the result. XML-RPC is quite simple and flexible, but not very widely used anymore, with *SOAP* and *REST* more common. However, it is still useful at times and somewhat interesting as an introduction to *SOAP*.

11.1 Using *XML* for Remote Procedure Calls: XML-RPC

After *HTTP* became so ubiquitous, people realized that it could be readily leveraged to communicate between any client and a server, and the technology named XML-RPC [17] arose. XML-RPC uses *HTTP* to send a request from a client to a server and then to return the result. This exchange is in *XML*. That is, XML-RPC uses *XML* to represent the call to the server's method, the data within the call, and the return result. This representation supports: scalars such as logical values, integers, real values, strings; dates and times; arrays of elements in order; arbitrary structures consisting of named fields; binary objects using base64 encoding; and the nil object. In short, XML-RPC has ways to represent data types from most languages.

The specification of XML-RPC is available at `http://xmlrpc.scripting.com/spec.html`. XML-RPC is a reasonably old technology, yet it has the advantage of simplicity while still being comprehensive. Although both *REST* and SOAP are more commonly used today, XML-RPC is still used in some contexts, either exclusively or as an alternative to the *REST* and SOAP interfaces also offered. We describe it here for completeness because it is still used and also because it illustrates some of the similarities and differences with both *REST* and SOAP interfaces.

The XMLRPC package [12] hides the details of XML-RPC from the *R* user's perspective and allows her to invoke a server's method via the function xml.rpc(). This function takes the *URL* of the server and the name of the server's method to be invoked. The *R* user can also pass inputs to the server's method as regular *R* arguments to the call to xml.rpc(). Unlike *R*, XML-RPC inputs must be specified in order rather than by name. The following illustrate basic xml.rpc() calls to the Advogato [1] XML-RPC server (Advogato is a "free software developer's advocate")

```
server = 'http://www.advogato.org/XMLRPC'
xml.rpc(server, 'test.sumprod', 9L, 10L)
```

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_11,

```
[1] 19 90
```

```
xml.rpc(server, 'test.capitalize', 'abCdef')
```

```
[1] "ABCDEF"
```

We can see that the names of the methods being invoked are `test.sumprod` and `test.capitalize`. In the first call, we pass two integers (9 and 10) as two separate arguments. In the second call, we pass a string.

The xml.rpc() function takes the inputs and creates the relevant call as an *XML* document, converting the *R* inputs to the corresponding *XML* representation. For example, the call to the *test.sumprod()* method appears as

```
<methodCall>
 <methodName>test.sumprod</methodName>
 <params>
  <param>
   <value><i4>9</i4></value>
  </param>
  <param>
   <value><i4>10</i4></value>
  </param>
 </params>
</methodCall>
```

xml.rpc() then sends the *XML* document via an *HTTP* **POST** request, filling in details in the *HTTP* header. The function then processes the result which is an *XML* document. For example, in our call to the *test.sumprod()* method, we receive the following *XML* in return

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value>
    <array>
     <data>
      <value><int>19</int></value>
      <value><int>90</int></value>
     </data>
    </array>
   </value>
  </param>
 </params>
</methodResponse>
```

xml.rpc() then converts this *XML* document to an *R* object and returns this object.

Since XML-RPC uses an *HTTP* request to send the *XML* invocation of the method, there are times when we may need or want to control that *HTTP* request. The xml.rpc() functions uses RCurl and a curl handle to send the *HTTP* request. We can use the *.opts* parameter to pass any options for the *HTTP* request as a named list of options. For example, we can use cookies, specify the user-agent in

the request, arrange to follow any Web server re-directions, and provide a user and password for the server using

```
xml.rpc(server, 'test.capitalize', 'abCdef',
        .opts = list(userpwd = "me:secret", cookiefile = "",
                     followlocation = TRUE,
                     useragent = "R XMLRPC"))
```

One can create and customize the curl handle ahead of the call to xml.rpc() and then pass it to xml.rpc(). For example,

```
library(RCurl)
con = getCurlHandle(userpwd = "me:secret", cookiefile = "",
                    followlocation = TRUE, useragent = "R XMLRPC")
xml.rpc(server, 'test.capitalize', 'abCdef', .curl = con)
```

This prior creation of the connection can be very useful when making several calls to the same XML-RPC server in a short period. Creating the curl handle once and reusing it in each call allows you to keep the connection between your machine and the server active and so avoid the overhead of re-establishing it in each call.

In most cases, the xml.rpc() function will convert the *XML* returned by the server to an *R* object appropriately. However, there are cases where you may want to interpret the *XML* directly. For example, if the result is a large *XML* document and you only want a small number of elements it contains, then you can avoid the overhead of processing all of the content. The *.convert* parameter to xml.rpc() is how one avoids the default conversion of the result. If this is `FALSE`, the *XML* content is returned as a string. You can then pass this string to xmlParse() and use functions such as getNodeSet() and xpathApply() to identify the nodes of interest and extract the content you want. Alternatively, you can pass an *R* function as the value for *.convert*. This function is called by xml.rpc() and passes the *XML* content as a string. The following two commands are equivalent:

```
xml.rpc(server, method, .convert = myFun)
myFun(xml.rpc(server, method, .convert = FALSE))
```

In most cases, we can pass *R* objects in our calls to XML-RPC methods, and the xml.rpc() function and its helper rpc.serialize() take care of converting each object to the appropriate *XML* representation. There is one potentially ambiguous situation in which the caller may need to be more specific about what is needed. Suppose the XML-RPC method expects an array of values, e.g., an array of real-valued numbers. If the *R* object we pass for that parameter is a numeric vector of length 1, i.e., a numeric scalar, then rpc.serialize() will represent this as a simple XML-RPC scalar. It unfortunately cannot know that the XML-RPC method expects an array with that value as the sole element. To force the scalar to be treated as an XML-RPC array, we can use the I() function to create an object of class `AsIs`, e.g.,

```
xml.rpc(server, "method", TRUE, I(3.1415))
```

The rpc.serialize() function then knows to create the array rather than the XML-RPC scalar.

In some programming situations, we may not be in a position to easily pass the arguments for the XML-RPC method to xml.rpc() directly in the call. Instead, we might have the arguments as a list. When this occurs, we can pass these directly to xml.rpc() via the *.args* parameter.

11.2 Classes for Representing the XML-RPC Server

When we call xml.rpc(), we specify the server, the method name, and the method's inputs. Often we will reuse the same curl connection for the same server. In this case, it is slightly more convenient to keep the *URL* for the server and the curl connection in a single object. The `XMLRPCServerConnection` class can be used to represent this object. We can create an instance of this class by calling the XMLRPCServer() function. We do this by specifying the server's *URL* in the *url* argument and the curl connection in the *curl* argument in the call to XMLRPCServer(). The *curl* argument can either be a simple logical value indicating to create a new curl connection or an explicit `CURLHandle` object that the caller has created previously. For example,

```
server = XMLRPCServer('http://www.advogato.org/XMLRPC', TRUE)
```

creates the server and a curl connection. The benefit is that the connection can be reused across calls. This keeps the connection with the server established and reduces the overhead of re-establishing it for each call with a new curl connection. Also, if you already have a connection to a server, then you can use it with XML-RPC too by passing the curl handle to XMLRPCServer().

Another benefit of this class is that we can invoke methods using a different syntax that avoids calling xml.rpc() explicitly. Instead, we can use a call of the form `server$method(arg1, arg2, ...)`. For example, we can call the method *test.sumprod()* via the command

```
server$test.sumprod(9L, 10L)
```

When we make calls of the form `server$method()`, the server's *URL* and curl connection are used.

At the least, XMLRPCServer() takes only the *URL* of the server and returns a simple `XMLRPCServer` object:

```
server = XMLRPCServer('http://www.advogato.org/XMLRPC')
```

The only benefit of this base class is that we can invoke methods using calls of the form `server$method(arg1, arg2, ...)`. That is, the default value of the *curl* argument implies that you do not want a curl handle to be reused across requests, i.e., each request will create one.

We can customize how the curl connection is created either by specifying explicit named curl options in the call to XMLRPCServer() or by passing an existing curl connection via the *curl* argument. The following are equivalent:

```
server = XMLRPCServer('http://www.advogato.org/XMLRPC',
                      verbose = TRUE, followlocation = TRUE,
                      cookie = "MyCookie=abcd",
                      ssl.verifypeer = FALSE)

curlHandle = getCurlHandle(verbose = TRUE, followlocation = TRUE,
                           cookie = "MyCookie=abcd",
                           ssl.verifypeer = FALSE)
server = XMLRPCServer('http://www.advogato.org/XMLRPC', curlHandle)
```

We can then use the `server` object in calls such as

```
server$test.capitalize('abCdef')
```

The code looks the same but in this case we are re-using the connection in `curlHandle` and not establishing a new connection.

One can also create more specific classes of `XMLRPCServer` for specific servers and provide more information about its available methods.

11.3 Writing *R* Functions to Use XML-RPC

In this section, we illustrate how to use the XMLRPC package to build an *R* interface to an XML-RPC server and its collection of methods. The primary example interfaces to a Wordpress [18] blog site. This is fully implemented in the RWordPress package [10] and so we illustrate some of the general strategies we used to create the interface. A second example interfaces to Ubigraph [15, 14], and involves communicating with a server that renders networks and graphs. The server provides methods for creating, displaying, and manipulating graphs. In this example we manage the dynamic state of the graph across calls to the server.

11.3.1 Programmatically Accessing a Blog

WordPress is a widely used service for blogging. One can register a new blog and then post entries in different categories or general topics, e.g., methodology, computing, hiking. WordPress hosts blogs at the wordpress.com domain. One can create a new blog there and get a new domain such as `http://omegahat.wordpress.com`. This is where you can read the Omegahat blog and also where the owner can post to it. Alternatively, the blog owner can host a WordPress-based blog on another machine, such as a machine maintained by the owner, using the WordPress software.

In addition to its Web browser interface, WordPress also supports an XML-RPC interface for programmatically querying, posting and updating blog entries and information about a blog. In fact, WordPress actually supports three common XML-RPC interfaces for blogging—Movable Type, metaWeblog, and Blogger. The WordPress API is the collection and extension of the methods in these interfaces and additional methods with WordPress-specific functionality. Information about the methods in the different interfaces is available at `http://codex.wordpress.org/XML-RPC_Support`. The *R* package RWordPress has *R* functions that provide access to all of these XML-RPC methods. In this section, we look at a few of these methods and how they are implemented in RWordPress.

In order to use the XML-RPC interface, all we need is the *URL* for the blog. We specify this via the *R* option *WordpressURL*. This allows us to use code that will work for any WordPress blog hosted on any machine. Most of the XML-RPC methods require a login and password for the blog. Rather than making the user specify them explicitly in the code, we make our functions read these from an option *WordpressLogin*. We use the Omegahat blog as an example by setting the *WordpressURL* and *WordpressLogin* login:

```
options(WordpressURL = "https://omegahat.wordpress.com/xmlrpc.php",
        WordpressLogin = c(login = "password"))
```

Of course, you will use your own blog location and login and password. Unfortunately, the API does not allow you to query multiple blogs with a given login.

To find the set of available methods, we can use the XML-RPC interface itself. There is a method named *mt.supportedMethods()* to do this. Note that the prefix “mt” to the method name indicates this

method is part of the Movable Type interface. WordPress-specific methods begin with the prefix `wp`. We can invoke *mt.supportedMethods()* with the call

```
o = xml.rpc("https://wordpress.com/xmlrpc.php",
            "mt.supportedMethods")
```

The result is a character vector with 76 elements.

Another one of the XML-RPC methods is named *mt.getRecentPostTitles()*. When we consult the documentation for this method at http://codex.wordpress.org/XML-RPC_MovableType_API, we see that it has four parameters. These are the name of the blog, the user name of the blog owner, the password for the account, and an optional integer giving the maximum number of posts to include in the return. We can invoke this with the call

```
ans = xml.rpc("http://omegahat.wordpress.com/xmlrpc.php",
              "mt.getRecentPostTitles", "", login, password, 100L)
```

Note that we pass an empty string for the blog name. The result is a list of each post to the Omegahat blog. Each element is a simple list with six elements with the following names

```
[1] "dateCreated"    "userid"         "postid"
[4] "title"          "post_status"    "date_created_gmt"
```

The return value includes the identifier for each post (`postid`), which allows us to later work with that particular post, e.g., retrieve comments for, or update, the post. We can combine these post entries into a data frame with a column for each of these six fields with

```
do.call(rbind, lapply(ans, as.data.frame))
```

We can combine all of these actions into a function for this XML-RPC method

```
getPosts = getRecentPostTitles =
function(num = 100, blogid = "",
         login = getOption("WordpressLogin",
                           stop("need a login and password")),
         ..., .server = getServerURL())
{
  ans = xml.rpc(.server, "mt.getRecentPostTitles",
                as.character(blogid), names(login),
                as.character(login), as.integer(num), ...)

  do.call("rbind", lapply(ans, as.data.frame))
}
```

Our function definition allows additional arguments that are passed on to xml.rpc() via the ... parameter. This can be used either to control the *HTTP* request by specifying a value for the *.opts* argument or to pass a curl handle via *.curl*.

As a side note, we can change the parameters of getRecentPostTitles() to allow the caller to specify the login and password separately or as a named character vector. We can also compute the name of the server by using the login name concatenated with `wordpress.com/xmlrpc.php`.

Next, we look at how to post a blog entry. Our post will contain text and both a table and a plot created within *R*. We construct the content of the post as *HTML* and include the text, table, and reference to our plot image file. The image will be created locally on our machine, and so we need to upload it to the blog server and then reference it there. We start by uploading to the blog the plot we created in *R* (as a PNG file).

The XML-RPC interface for WordPress provides a method named *wp.uploadFile()* to upload files. As with the other methods, this expects the blog identifier and user login and password as separate parameters. The final parameter provides the details about the image. The XML-RPC method expects an *XML* structure with members:

- `<bits>` – the binary contents of the image file
- `<type>` – the MIME type for the image, i.e., `image/png`
- `<name>` – a string giving the name to use for the file on the server
- `<overwrite>` – a logical value that controls whether the server overwrites an existing file on the server, or raise an error instead.

All we need to do in *R* is to create a list with named elements corresponding to the members of the structure and then fill them in appropriately. The `<bits>` member is the least obvious. However, for this we need to read the PNG file into *R* as a raw vector. We can use readBin() for this and pass it a raw vector with as many bytes as there are in the file. The function readBinaryFile() in the RWordPress package does this, so we can write the *R* function for uploading a file to WordPress as

```
uploadFile =
function(what, type = guessMIMEType(what), blogid = "",
         login = getOption("WordpressLogin",
                           stop("need a login and password")),
         remoteName = basename(what), overwrite = TRUE, ...,
         .server = getServerURL())
{
  if(inherits(what, "AsIs")) {
    content = what
  } else {
    if(!file.exists(what))
      stop("no such file ", what)
    content = readBinaryFile(what)
  }

  info = list(name = remoteName, type = type,
              bits = content, overwrite = overwrite)
  xml.rpc(.server, "wp.uploadFile",
          as.character(blogid), names(login),
          as.character(login),  info, ...)
}
```

Note the definition of the variable `info`. This is where we create the *R* structure for xml.rpc(), and xml.rpc() converts it into the *XML* format expected by the method. With this function, we can upload our file with the simple call

```
image = uploadFile("quakes.png")
```

This call to uploadFile() returns an *R* list with three named elements. The important one is `image$url` which holds the *URL* for the file on the blog server. We can use this in our blog post to refer to the plot.

The next step is to create the *HTML* content for the blog entry. There are several different *R* packages to help create this *HTML*, or we can create it manually with the tools in the XML package [8]. Once we have created the *HTML*, we are ready to post it to our blog. We call the XML-RPC method

metaWeblog.newPost() provided by the WordPress API to make the post. This method has five arguments; the blogid, username, password, the content to be published, and a boolean indicating whether to publish the content or leave it as a draft of a post. The content argument describes the post and has several members. The description member is the *HTML* for the post. We can also provide other members such as a title, an author, and a category vector. Our newPost() function can be defined as

```
newPost =
function(content, publish = TRUE, blogid = "",
         login = getOption("WordpressLogin",
                           stop("need a login and password")),
         .server = getServerURL())
{
  if(is.character(content))
    content = list(description = content)

  ans = xml.rpc(.server, "metaWeblog.newPost",
                as.character(blogid), names(login),
                as.character(login),
                content, as.logical(publish))
  ans
}
```

The result is a string identifying the blog entry.

There are many other methods in the WordPress API that we can explore. You can study the RWordPress package to see what has been implemented and how.

11.3.2 Interactive and Dynamic Network Graphs with Ubigraph

Ubigraph [15] (http://ubietylab.net/ubigraph/) is a tool for visualizing dynamic and interactive networks or graphs of nodes and edges. Dynamic means that we can add, remove, hide, and show nodes and edges in the graph or change their characteristics at any time and the graph will be immediately re-displayed. Interactive, in this context, means that we can spin and zoom the graph to look at different aspects of it rather than have a single static view such as with Graphviz http://www.graphviz.org/ [2, 4]. Ubigraph is free to download, but not Open Source. We run it as a stand-alone process that both displays the current graph and acts as an XML-RPC server. We can then invoke the XML-RPC methods it provides to create and manipulate the graph. We use Ubigraph to illustrate XML-RPC and also how easy it is to create an XML-RPC client. The RUbigraph package [9] is a reasonably complete implementation of an interface to the Ubigraph methods. Again, we illustrate some of the basic functions and how they use xml.rpc().

There are numerous other graph visualization tools. One example is Cytoscape [7] (http://www.cytoscape.org/), an open source software platform for visualizing complex networks via static graphs. Cytoscape has become a standard network visualization tool in molecular biology. In addition to a point-and-click interface for interactive use, Cytoscape has a plug-in which allows programmatic access via an XML-RPC connection. This allows us to create and manipulate graphs from applications such as *R*. The RCytoscape package [6] provides an interface for the XML-RPC methods in Cytoscape's XML-RPC plug-in, using XMLRPC to invoke these methods. This is another good resource for understanding how to use the XMLRPC package.

We first start the downloaded Ubigraph server. It may be necessary to run this as root, e.g.,

```
sudo $\tilde$/Downloads/UbiGraph-alpha-0.2.4-MacOSX10.4Intel/
          bin/ubigraph_server


4 processors
Using single-level layout.
Running Ubigraph/XML-RPC server.
```

This produces an empty black window on the screen. The server is now running and awaiting requests to the *URL* `http://127.0.0.1:20738/RPC2`.

The documentation for the Ubigraph API lists five basic methods to manipulate the structure of the network. The first is *ubigraph.clear()* to remove all of the nodes and start a new graph. The methods *ubigraph.new_vertex()* and *ubigraph.new_edge()* are for creating new nodes and edges in the graph. Both return the unique identifier of the new object—an integer. These identifiers can be used to manipulate those objects, e.g., to remove them with *ubigraph.remove_vertex()* and *ubigraph.remove_edge()*, and also to set their characteristics such as color, label, shape, and visibility. We set these characteristics via additional XML-RPC methods named *ubigraph.set_color_attribute()*, *ubigraph.set_arrow_position_attribute()*, *ubigraph.set_shapedetail_attribute()*, and so on. We program all of these in a reasonably simple manner.

The identifiers for the elements of the network—vertices and edges—are the essential data types for this interface. Given an identifier, we can perform many operations with it, e.g., change its color or shape. Accordingly, we define classes to represent these identifiers. We use a general identifier `UbigraphID` and two more specific subclasses that represent a vertex and an edge respectively:

```
setClass("UbigraphID", contains = c("integer"))
setClass("VertexID", contains = "UbigraphID")
setClass("EdgeID", contains = "UbigraphID")
```

These are simply (scalar) integers. The class label identifies the nature of the object that they identify.

Note that you can extend these to create classes for specific types of vertices and edges such as an *R* or **Bioconductor** package and an edge that represents a dependency between two packages. We may even have classes for a simple dependency, an import, or a suggests. Similarly, we may have classes to represent an *R* or *C* function as a vertex and an edge that shows how one function calls another.

With these classes defined, we can build the important functions to create new vertices and edges. The simple form of the function for creating a vertex can be defined as

```
newVertex =
function(server = "http://127.0.0.1:20738/RPC2",
         curl = getCurlHandle(), .opts = list())
{
  ans = xml.rpc(server, "ubigraph.new_vertex",
                .curl = curl, .opts = .opts)
  ans = new("VertexID", ans)

  ans
}
```

The *server* argument of this function allows us to specify an alternative location for the Ubigraph server and with *curl*, we can also pass an existing curl handle to use in multiple calls to the server. Note that the function returns an object of class `VertexID`.

The *R* function to create an edge requires identifiers for the source and destination vertices of the edge. We can define it as

```
newEdge =
function(src, dest, server = "http://127.0.0.1:20738/RPC2",
         curl = getCurlHandle(), .opts = list())
{
  if(length(src) > 1 || length(dest) > 1) {
    return(mapply(newEdge, src, dest, id,
                  MoreArgs = list(server = server, .attrs = .attrs,
                                  curl = curl, .opts = .opts)))
  }

  ans = xml.rpc(server, "ubigraph.new_edge", as.integer(src),
                as.integer(dest), .curl = curl, .opts = .opts)
  new("EdgeID", ans)
}
```

Note that this function is vectorized in that it allows the caller to specify two collections of identifiers for both the source and destination vertices and this will create an edge for each one.

In the RUbigraph package, we actually define these two functions in a slightly more flexible manner. Firstly, we allow the caller to specify attributes (e.g., color, label, ...) for the vertex and edge being created. These can be specified as additional named arguments in the call and are collected in the ... parameter, or as a single named list via the *.attrs* parameter. The names identify the characteristic being set and the value is the value of that characteristic.

We also allow the caller to specify the identifier for the vertex or edge being defined. This can be a single identifier or a collection that will cause multiple objects to be created. If the identifier is provided for creating an object, we use a different XML-RPC method by appending `_w_id` to the name of the regular method. The newVertex() function in the RUbigraph package is defined as

```
newVertex =
function(id = NA, ..., server = "http://127.0.0.1:20738/RPC2",
         .attrs = list(...), curl = getCurlHandle(), .opts = list())
{
  if(length(id) > 1)
     return(sapply(id, newVertex, .attrs = .attrs, server = server,
                   curl = curl, .opts = .opts))

  if(is.na(id))
      ans = xml.rpc(server, "ubigraph.new_vertex",
                    .curl = curl, .opts = .opts)
  else {
     if(xml.rpc(server, "ubigraph.new_vertex_w_id", as.integer(id),
                .curl = curl, .opts = .opts) == 0)
       ans = as.integer(id)
     else
       stop(sprintf("id (%d) is already in use", as.integer(id)))
   }

  ans = new("VertexID", ans)
```

```
  setAttributes(ans, .attrs = .attrs, .server = server,curl = curl)
  ans
}
```

Again, the function returns one or more `VertexID`s. Also in this function, the additional parameters are collected into the *.attrs* list and then are passed to a setAttributes() function.

As mentioned previously, each vertex and edge support numerous characteristics or attributes. For example, edges and vertices can have a `color`, `label`, `fontcolor`, `fontsize`, `fontfamily` and `visible` attribute. Also, a vertex can have a size attribute, while an edge has a width attribute. Vertices also have a `shape` and `shapedetail` attribute, while edges have the attributes named `arrow`, `arrow_position`, `arrow_radius`, `arrow_length`, `arrow_reverse`, `stroke`, `showstrain`, `oriented`, `strength`, `spline`. The setAttributes() function handles all of these with a small piece of generic code by using either the *ubigraph.set_vertex_attribute()* or *ubigraph.set_edge_attribute()* method according to the type of object whose attribute is to be set. We know which to use based on the class of the identifier, i.e., `VertexID` or `EdgeID`. The setAttributes() function calls the appropriate Ubigraph method and passes it the identifier of the object, the name of the attribute being set, and the value of that attribute.

```
setAttributes =
function(id, ..., .attrs = list(...),
         .type = if (is(id, "EdgeID"))
                   "edge"
                 else
                   "vertex",
         .server = "http://127.0.0.1:20738/RPC2",
         curl = getCurlHandle())
{
   op = sprintf("ubigraph.set_%s_attribute", .type)
   sapply(names(.attrs),
          function(at) {
            val = .attrs[[at]]
            if((at == "color" || at == "fontcolor") &&
               substring(val, 1, 1) != "#") {
              tmp = col2rgb(val)/255
              val = rgb(tmp[1], tmp[2], tmp[3])
            }
            if(is.logical(val))
              val = tolower(as.character(val))
            xml.rpc(.server, op, as.integer(id), at,
                    as.character(val), .curl = curl) == 0
          })
}
```

Since we have taken the time to define classes for the object identifiers, we can make setting attributes simpler. For each of the possible attributes, we define a generic function for setting that attribute, e.g., a function named `label<-()`. Then we define methods for the appropriate class of Ubigraph object, i.e., `VertexID` or `EdgeID`. We do this programmatically in *R* since we know the names of all of the supported attributes and for which classes they are supported. This then allows us to set attributes in a more *R*-like manner as

```
label(node) = "RUbigraph"
width(edge) = 3L
color(edge) = rgb(255, 0, 0)
color(edge) = '#FF0000'
```

We have covered *R* functions for the XML-RPC methods to create nodes and edges and set attributes on the nodes and edges. RUbigraph provides wrapper functions for other methods. For example there are methods that clear, or empty, the network and remove an individual edge or vertex, which are named clear(), removeEdge() and removeVertex(), respectively. The implementation of these *R* functions is quite straightforward, given the fundamentals of the XMLRPC package. See the RUbigraph package for details.

Example 11-1 Creating an Ubigraph Network Display

Given all of the functions available in RUbigraph for working with Ubigraph, we can now create a network. As an example, we will display the collection of *R* packages available on the Omegahat repository and their dependencies. Each package will be a vertex in our display, and the edges between them illustrate the dependencies. We color the Omegahat packages red so the color of the node shows in which repository each package is located.

We start by loading the RUbigraph package and ensuring that the server is running.

```
library(RUbigraph)
stopifnot(isUbigraphRunning())
```

If this fails, we must start the Ubigraph server!

The next step is to retrieve a list of the Omegahat packages and also find all of their dependencies, both within the repository and on packages in other repositories.

```
dep = available.packages(contrib.url('http://www.omegahat.org/R',
                                     type = "source"))
pkgs = unique(rownames(dep))
g = utils:::.make_dependency_list(pkgs, dep)
all.pkgs = unique(c(unlist(g), pkgs))
```

Now we are ready to create the corresponding nodes and edges. Since there will be multiple calls to the server, we create a single curl handle and use this in all of our requests.

```
curl = getCurlHandle()
```

We next make certain to clear any existing nodes in the Ubigraph display, and then create the nodes for each package:

```
clear(curl = curl)
ids = lapply(all.pkgs,
             function(i)
               newVertex(.attrs = list(label = i), curl = curl))
names(ids) = all.pkgs
```

Now we change the color of each of the Omegahat packages to red, leaving the others unchanged. These multiple calls are connected operations in that we are setting attributes on existing objects, i.e., the nodes corresponding to the Omegahat packages. We use the node `ids` to do this as follows

```
sapply(ids[pkgs], function(id) color(id) = '#FF0000')
```

Finally, we incorporate the dependencies by drawing edges from each Omegahat package to each of the packages on which it depends:

```
sapply(pkgs,
       function(i) {
         d = g[[i]]
         if(length(d))
           sapply(ids[d],
                  function(o) newEdge(ids[i], o, curl = curl))
       })
```

11.4 Handling Errors in XML-RPC

The xml.rpc() function makes it reasonably straightforward to invoke methods on a server that supports XML-RPC. By hiding the details, there are fewer opportunities for things to go awry. However, there are many things that can still go wrong. We can pass inappropriate values in the call or inputs of the wrong type which the server was not programmed to handle. These are problems associated with the XML-RPC method. Other problems can occur because of the *HTTP* connection. For example, we may not be able to connect to the host due to too many previous requests, an incorrect password, or simply not being able to resolve the *URL*. In the case of problems with the XML-RPC method, the XML-RPC server will return an *XML* document that explicitly identifies the "fault." In the case of an issue with the *HTTP* request, the error will be generated by the Web server or the `libcurl` engine underlying the RCurl package. In either case, xml.rpc() will raise an *R* error that inherits from the *S3* class `simpleError`. For the high-level XML-RPC error, the *R* error will have class `XMLRPCError`. When designing an *R* package to interface to an XML-RPC application, you will want to make your package as robust as possible with respect to these kinds of errors.

Example 11-2 Understanding an Input Type Error in XML-RPC

Let's look at an example of a call that gives an XML-RPC error:

```
xml.rpc("http://www.advogato.org/XMLRPC", 'test.sumprod', 9, 10)
```

This gives us an error

```
faultCode:  1  faultString:  param 1:
  expecting <int>, got <double>
```

which is of class

```
[1] "XMLRPCError" "simpleError" "error"       "condition"
```

What's the problem? Basically, the XML-RPC methods expected an integer value, but it was given a real value instead; in fact, it was given two of them. In *R*, the literal value `9` corresponds to a numeric vector of length 1, i.e., a real or double value. This is why we used `9L` in our original call since that explicitly treats the literal value as an integer value.

This illustrates one of the problems with XML-RPC and weakly-typed languages generally. The xml.rpc() function cannot know the types the remote method expects. It takes the *R* values it is given and maps them to their corresponding *XML* representation. These inputs are only interpreted and checked when the request arrives at the remote server. The server can accept different types of inputs or else reject values it was not expecting. That is what is happening in this case.

In order to avoid problems with mismatched input types, it is much more sensible to create an *R* function corresponding to each XML-RPC method and have the function coerce its inputs to the appropriate *R* type and then pass them to xml.rpc(). There are two advantages to this approach. Firstly, it ensure the types are correct or a failure occurs before the XML-RPC request is made. Secondly, the *R* "wrapper" function can explicitly enumerate the expected arguments and give them informative names. This contrasts to the general xml.rpc() function which takes any number of inputs for the XML-RPC method. One can also provide *R* documentation for the method and its parameters.

The class of the errors allow us to handle them using *R*'s tryCatch() function. For example,

```
e = tryCatch(xml.rpc("http://www.advogato.org/XMLRPC",
                     'test.sumprod', 9, 10),
             error = function(e)  e)
```

Here we just collect the error so that we can find its class, which in this case is

```
[1] "XMLRPCError" "simpleError" "error"     "condition"
```

The error object's `message` element contains information about the problem,

```
e$message
```

```
[1] "faultCode:  1  faultString:  param 1:
   expecting <int>, got <double>"
```

Example 11-3 An HTTP Error in an XML-RPC Request

A simple example of an *HTTP* error occurs when we mistakenly use an *HTTPS URL* in the request:

```
err = tryCatch(xml.rpc("https://www.advogato.org/XMLRPC",
                       'test.sumprod', 9, 10L),
                error = function(e) e)
```

In this case, the class of the error is

```
[1] "COULDNT_CONNECT"  "GenericCurlError" "error"       "condition"
```

The hierarchy of error classes allows us to handle different situations in different ways. For example, we can catch XML-RPC and *HTTP* errors in different ways and allow all other errors, such as errors in the evaluation of the *R* code to compute the inputs, to be handled as usual. We can do this with

```
tryCatch(xml.rpc("https://www.advogato.org/XMLRPC",
                 'test.sumprod', 9, 10L),
         GenericCurlError = function(e)
                             cat("Problem with the HTTP request:",
                                 e$message, "\n"),
         XMLRPCError = function(e)
                           cat("Problem in the XMLRPC request:",
                               e$message, "\n"))
```

When developing functions within a package, it is good to deal with these different classes of errors appropriately and insulate the user from the differences.

11.5 Under the Hood of xml.rpc()

As we mentioned, the xml.rpc() function takes care of all of the details of marshalling the *R* inputs to *XML*, sending the request to the relevant *URL*, and converting the resulting *XML* document to an *R* object. Most users of XML-RPC services will not directly call the xml.rpc() function, but will use a "wrapper" function developed by somebody that corresponds to a specific XML-RPC method and that itself calls xml.rpc(). This means that in almost all cases, nobody needs to know the precise details of how these conversions to and from *XML* work. However, such information can be useful both for dealing with any problems that arise such as debugging problematic calls and for customizing calls, if necessary, by defining methods for serializing specific *R* classes. In this section, we discuss the details of the marshalling to and from *XML* for XML-RPC.

In XML-RPC, we represent a call to a server's method with an *XML* document of the form

```
<methodCall>
 <methodName>name of server's function</methodName>
 <params>
  <param>
    <value>....</value>
  </param>
 </params>
</methodCall>
```

The *XML* for the method call is quite simple and contains all of the information for the call. As we have seen already, in the simple case of calling the *test.sumprod()* method, this call appears as

```
<methodCall>
 <methodName>test.sumprod</methodName>
 <params>
  <param>
   <value><i4>9</i4></value>
  </param>
  <param>
   <value><i4>10</i4></value>
  </param>
 </params>
</methodCall>
```

The <methodName> element specifies the name of the method or function on the server that is being called. If the client provides inputs to the call, these are contained in a <params> element. In <params>, each input is within a child <param> element. Each <param> element has a <value> subnode that contains the value for the input parameter. The name of the node indicates the type of the value, e.g., <i4> or <double>.

The XMLRPC package provides high-level functions that allow us to make an XML-RPC call in one step. That is, the function xml.rpc() takes all the information describing the call, a *URL*, the name of the method and the *R* objects that are arguments for the method. It generates the appropriate *XML* document and makes the *HTTP* request. It then converts the result from *XML* to *R*.

The rpc.serialize() function performs the mapping of the *R* objects that are inputs to the XML-RPC function and creates the corresponding *XML* representations for them using the XML-RPC format. Table 11.1 shows the mapping of *R* objects to the *XML* elements that XML-RPC recognizes. For the most part, the mapping from *R* types to the XML-RPC representation is quite straightforward.

Vectors of length 1 in *R* are mapped to scalar values in XML-RPC. Since *R* has few primitive data types, the mapping from these to XML-RPC is clear. For example, *integer* values in *R* correspond to <i4> types in XML-RPC, and *numeric* values map to <double> types in XML-RPC. *R* vectors with more than one value naturally map to <array> elements in XML-RPC. There is some ambiguity in converting vectors of length 1 in *R* to XML-RPC as we do not know whether this should be a scalar value or an <array> with just one element. By default, as we mentioned, such vectors are mapped to scalar values. However, if we know that the function expects an array, we can force rpc.serialize() to map it to an <array> element by making the *R* vector to be of class AsIs. One way to do this is via the I() function, e.g., I(x).

We should note that rpc.serialize() ignores the names on *vector* objects. Indeed, it ignores all *R* attributes on a vector. However, for a named *list* object, rpc.serialize() creates a <struct> element in the *XML* document. This treats each of the elements in the *R* *list* as a field in the XML-RPC structure.

S3 classes are often either *vector* or *list* objects and so, by default, will be serialized as we described above. However, we see below that we can define methods to serialize particular *R* classes in different ways. In *R*, *S4* classes and objects are much richer than simple lists and *S3* classes. However, XML-RPC does not have an equivalent structure that represents class inheritance. Accordingly, rpc.serialize() just maps the collection of named elements to a list and then serializes that as a <struct> element in XML-RPC. The following example illustrates how this works. We start by defining two simple classes:

```
setClass("Bar", representation(xyz = "logical"),
         prototype = list(xyz = c(FALSE, FALSE, FALSE)))
setClass("Foo", representation(a = "integer", b = "numeric",
                               c = "character"),
         contains = "Bar")
```

Foo extends the base class Bar and so inherits the slot *xyz*. Next, we create an instance of the class Foo

```
obj = new("Foo", a = 1L, b = 3.1415, c = "Hello world")
```

We inherit the general method for serializing this *S4* object and so a call to rpc.serialize() produces

```
rpc.serialize(obj)
```

```
<struct>
 <member>
  <name>a</name>
  <value><i4>1</i4></value>
 </member>
 <member>
  <name>b</name>
  <value><double>3.1415</double></value>
 </member>
 <member>
  <name>c</name>
  <value><string>Hello world></string></value>
 </member>
 <member>
  <name>xyz</name>
```

```
  <value>
   <array>
    <data>
     <value><boolean>0</boolean></value>
     <value><boolean>0</boolean></value>
     <value><boolean>0</boolean></value>
    </data>
   </array>
  </value>
 </member>
</struct>
```

Table 11.1: Mapping between *R* and XML-RPC Data Types

***R* type**	**Description**	***XML* element**
logical scalar	Boolean value that maps to 1 or 0 for `TRUE` and `FALSE`	`<boolean>`
integer scalar	Positive or negative integer value that fits into 4 bytes	`<i4>` or `<int>`
numeric scalar	Real-valued number	`<double>`
character scalar	Single string consisting of a sequence of characters	`<string>`
POSIXt scalar	Represents date–time in the form of Year-MonthDayTHour:Minute:Seconds, i.e., the format `%Y%m%dT%H:%H:%S`	`<dateTime.iso8601>`
Date	Converts date to a date–time with time being 00:00:00 and then represents that as an XML schema dateTime type	`<dateTime.iso8601>`
raw	Represented as base64 encoded binary	`<base64>`
empty vector–`NULL`	Represented as empty XML-RPC array	`<array>`
vector with at least 2 elements	Array	`<array>` consisting of a single `<data>` element and 0 or more `<value>` elements
objects of class `AsIs`	Convert an *R* vector of length 1 to an array rather than a scalar value	`<array>`
named list	Represents the container as a struct with named elements that are the members of the struct	`<struct>` with the named elements given by `<member>` elements consisting of `<name>` and `<value>` pairs. The content of the `<name>` is text and not a `<string>`.
S4 object	Converts to a named list made up of the slots and then represented as a `<struct>`	`<struct>`

This table shows the correspondence between R data types and XML-RPC data types. It shows how R objects are mapped to XML-RPC representations when making a call with xml.rpc(), and similarly how XML-RPC responses are converted to R objects.

If for some reason the representation of an *R* object is not appropriate for the XML-RPC call, we can define our own method for serializing a particular class of object to an XML-RPC format. Such a method will typically transform the *R* object to some other representation in *R* and then call rpc.serialize() on the resulting subelements and combine them into an *XML* subtree. However, the method can create the *XML* content directly itself.

Example 11-4 Serializing the `timeDate` Class to XML-RPC

The `timeDate` class in the package of the same name is used to represent a sequence of date–time pairs. While it uses the `POSIXct` class to store the times, it has slots for additional information such as the format of the time–date specification and the time zone. We would want to serialize these by just using the `POSIXct` values. We can do this by defining the rpc.serialize() method as

```
setMethod("rpc.serialize", "timeDate",
          function(x, ...) rpc.serialize(as(x, "POSIXct"), ...))
```

Here we transform the `timeDate` object trivially by suppressing its additional slots. In other cases, we may have to combine the internal information to create a more typical representation and then call rpc.serialize().

We can also define our own methods for controlling how to serialize *R S3* or *S4* objects to *XML* for XML-RPC requests. We look next at an example of how to do this.

Example 11-5 Serializing an S3 `lm` Object to XML-RPC

Suppose we have a linear model of class `lm` that is represented in *R* as a list with named elements containing information about the fitted model, including the coefficient estimates, the residuals, and the model matrix. Also, suppose that an XML-RPC method expects the array of coefficients of the fitted model and the corresponding names of the variables, but not the additional components. When calling the XML-RPC method, we can extract the coefficients and the names of the terms and pass those as a new list, e.g.,

```
xml.rpc(url, method, list(coefficients = coef(fit),
                          variables = attr(fit$terms,
                                           "term.labels")))
```

However, we can arrange to do this automatically for any `lm` object passed in any XML-RPC method by registering a method for rpc.serialize() for the class `lm`. This is done with

```
setOldClass("lm")
setMethod("rpc.serialize", "lm",
          function(x, ...) {
            rpc.serialize(list(coefficients = coef(fit),
                               variables = attr(fit$terms,
                                                "term.labels")),
                          ...)
          })
```

Again, we create the new object and then pass this to the generic rpc.serialize() function that will create the appropriate *XML* document. When an *R* user calls xml.rpc() and passes an object of class `lm`, this method will be used to serialize that argument.

Example 11-6 Serializing Specific Classes in XML-RPC

Creating a global method to convert all instances of a particular *R* class to XML-RPC format may be overkill or inappropriate for specific classes. However, it can be useful if there is a dominant and consistent way to represent information in that class other than its basic serialization as an *R* object. For example, suppose we have a simple `ts` object such as the one created with

```
tt = ts(1:10, frequency = 12, start = c(1959, 4))
```

When displayed in *R*, we see

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959                 1   2   3   4   5   6   7   8   9
1960  10
```

However, this is actually an integer sequence with an attribute `tsp` that gives the time periods:

```
unclass(tt)

 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"tsp")
[1] 1959.25 1960.00   12.00
```

When serializing this as input to an XML-RPC method, we do not want the array of values 1, 2, 3, . . . , 10, which is the default behavior. Instead, we can think of this as an array of `dateTime` objects. In this case, we can convert the `ts` object to `POSIXct` and then serialize this. The general details are more involved than we suggest here as the `ts` class, in fact, allows for different frequencies, time periods, etc., but the basic approach is the key concept.

This ability to define methods for serializing *R* objects allows us, for example, to specify how to map `matrix` and data.frame() objects to their XML-RPC representations.

11.5.1 The HTTP Request

The XML-RPC specification covers both the format of the *XML* documents and also the *HTTP* request. The *HTTP* request should be a **POST** operation. The body of the **POST** request is the *XML* document. In the header of the request, the *Content-Type* should be `text/xml`. The caller should specify the *User-Agent* to identify the application that is making the method call. This can be left as XML-RPC or be adjusted to specify more information about the nature of the code making the request, e.g., *R* and/or the name of the *R* package.

11.6 Possible Enhancements and Extensions

xrdl: XML-RPC Description Language
Like most *REST* interfaces, to implement an XML-RPC interface, we have to read documentation describing all of the methods and the inputs and outputs for each. We use this information to ensure that the inputs to our *R* functions are appropriate before making the request and coercing them to the appropriate types. Rather than doing this manually, we would like to have software that reads a description of the methods and their inputs and outputs and then generates *R* code and local documentation. We do this via a *WSDL* (Web Service Description Language) [3] document for SOAP (Simple Object Access Protocol) services (see Chapter 12). There is an approach to doing this for XML-RPC described at `http://code.google.com/p/xrdl/`. Similarly, it is possible to use *WSDL* to describe an XML-RPC API. A task is to implement either or both of these approaches for

R. These will build directly on the XMLRPC package and the xml.rpc() function. They can mimic the approach in SSOAP [11] and XMLSchema [13]. Each XML-RPC method is mapped to an *R* function that checks and coerces the inputs to the types specified in the method description. There are opportunities to define new *R* classes corresponding to data structures described in the XML-RPC methods.

11.7 Summary of Functions to use XML-RPC from *R*

The XMLRPC package exports one key function, xml.rpc(). This is all that is needed to invoke XML-RPC methods. The XMLRPCServer() function offers a mechanism for treating the XML-RPC server as an object in *R* and syntactic sugar for invoking methods. If we want to control how *R* objects are serialized to *XML* to be inserted into the body of an XML-RPC request, we can provide methods for the rpc.serialize() generic function.

xml.rpc() Invoke an XML-RPC server's method, passing the inputs to the method as regular *R* arguments. To provide an alternative conversion for the result, an *R* function can be provided via the *.convert* parameter, or this parameter can be set to `FALSE`, indicating the *XML* content is to be returned as an string. If the inputs are in a list, then these can be passed to xml.rpc() via the *.args* parameter.

XMLRPCServer() Create an object that represents an XML-RPC server to which we can send requests. We can then use this object to make one or more requests, possibly reusing the same *HTTP* connection. One of the simple benefits of the server object is that we can make calls of the form `server$method(arg1, arg2, ...)` rather than using the xml.rpc() function directly.

rpc.serialize() Convert an *R* object to its equivalent *XML* form for use in an XML-RPC request. We can define methods for different *R* classes to control how these are converted, and we can reuse the methods for the *R* primitive types.

11.8 Further Reading

An introduction to XML-RPC [16] is available online at `http://scripting.com/davenet/1998/07/14/xmlRpcForNewbies.html`. For the details of the XML-RPC specification see [17] (`http://xmlrpc.scripting.com/spec.html`). [5] provides descriptions of how to use XML-RPC to implement clients in *C*, *C++*, *Python* and other languages.

References

[1] Advogato. The Advogato Community resource site for developers of free software. `http://www.Advogato.org`, 2011.

[2] Arif Bilgin, Don Caldwell, John Ellson, Emden Gansner, Yifan Hu, and Stephen North. Graph visualization software: Drawing graphs since 1988. `http://www.graphviz.org/`, 2012.

[3] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[4] Jeff Gentry, Robert Gentleman, and Wolfgang Huber. How to plot a graph using Rgraphviz. http://bioconductor.org/packages/2.6/bioc/vignettes/Rgraphviz/inst/doc/Rgraphviz.pdf, 2010.
[5] Eric Kidd. XMLRPC how to. http://tldp.org/HOWTO/XML-RPC-HOWTO, 2001.
[6] Paul Shannon. RCytoscape: Interactvive viewing and exploration of graphs, connecting *R* to Cytoscape. http://rcytoscape.systemsbiology.net, 2011. *R* package version 1.8.1.
[7] Michael Smoot, Keiichiro Ono, Johannes Ruscheinski, Peng-Liang Wang, and Trey Ideker. Cytoscape 2.8: New features for data integration and network visualization. *Bioinformatics*, 27:431–432, 2011.
[8] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.
[9] Duncan Temple Lang. RUbigraph: Interface to Ubigraph server via XML-RPC. http://www.omegahat.org/RUbigraph/, 2012. *R* package version 0.1-0.
[10] Duncan Temple Lang. RWordPress: Interface to WordPress blogs. http://www.omegahat.org/RWordPress, 2012. *R* package version 0.2-3.
[11] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.
[12] Duncan Temple Lang. XMLRPC: Remote procedure call (RPC) via *XML* in *R*. http://www.omegahat.org/XMLRPC, 2012. *R* package version 0.2-5.
[13] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.
[14] Todd Veldhuizen. Dynamic multilevel graph visualization. http://arxiv.org/abs/0712.1549, 2007.
[15] Todd Veldhuizen. UbiGraph: Free dynamic graph visualization software. http://ubietylab.net/ubigraph, 2007.
[16] Dave Winer. XML-RPC for newbies. http://scripting.com/davenet/1998/07/14/xmlRpcForNewbies.html, 1998.
[17] Dave Winer. XML-RPC specification. http://xmlrpc.scripting.com/spec.html, 1999.
[18] WordPress Community. Blog tool, publishing platform, and CMS. http://wordpress.org/, 2012.

Chapter 12
Accessing *SOAP* Web Services

Abstract This chapter looks at another mechanism used to access Web services — *SOAP* — which is similar to XML-RPC and to some extent *REST*. *SOAP* is more structured and general than XML-RPC and *REST*. It is also ostensibly more complex and more difficult to understand because of all of the details about how to format and send requests as *XML* documents. However, much of the complexity of *SOAP* is hidden from users as we can programmatically generate code in *R* (and other languages) to access any *SOAP*-based Web service and its collection of methods using the genSOAPClientInterface() function. This hides all of the details underlying *XML* and *HTTP* while still leaving the user with a lot of flexibility and control. In this chapter, we describe how to access *SOAP* Web services from within *R* and how to generate the interfaces from *WSDL* documents describing the service's methods and data structures. For more advanced programmers, we also discuss approaches to customize how this code is generated and how to use the .SOAP() function directly to gain more control for more sophisticated applications of *SOAP* in *R*.

12.1 Introduction: What Is *SOAP*?

SOAP, Simple Object Access Protocol [1, 4], is an alternative approach to invoking methods on a Web server than *HTML* forms, XML-RPC, and *REST*. This technology grew out of XML-RPC, but is both a lot richer and, at its lowest level, a lot more complex. Fortunately, at the high-level that most people use, *SOAP* is actually both simpler and more advanced than XML-RPC and *REST*. As with most competing technologies, each has its own merits and demerits. *REST* is becoming increasingly common and popular, but *SOAP* is also widely used. We discuss why later in the chapter.

We will first describe the high-level functionality (processWSDL() and genSOAPClientInterface()) of the SSOAP package [13] and how to use it without the need for much understanding of the details. We also discuss how this high-level approach can be customized in different ways. We then discuss the intermediate level-functions (e.g., .SOAP()) that *R* programmers may wish to use to adapt and customize some of the approaches provided by the high-level functions. We also give an overview of the *SOAP* mechanism for those who want to understand the low-level details. We illustrate each of these levels with, hopefully, simple but relevant examples.

Similar to XML-RPC, *SOAP* uses *HTTP* to send a method invocation to a Web server and receive the result. It also uses *XML* as the format for representing the method invocation and the resulting value. As with the XMLRPC package [14], there is a single function — .SOAP() — that takes care of

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_12,

all the details of invoking a remote method. Additionally, just as with xml.rpc(), each call will specify the server, the name of the method, and the *R* objects that are the arguments/inputs to that method.

For XML-RPC, the format of the *XML* for representing the call and more importantly the data within the call is fixed. We are allowed to have scalar values with different types: booleans, three different types of numbers, strings, date–times and base-64 encoded data. Nonscalar values are limited to ordered arrays and "structs," which are very similar to lists with named elements or members. This set of data types is all that is allowed in XML-RPC, and this leads to its simplicity. In contrast, *SOAP* uses the power of *XML* schema to both allow a richer set of primitive data types and also to allow new data types to be defined and used in application-specific ways. Schema allow us to define complex composite data types corresponding to classes and object-oriented systems generally. *SOAP* permits us to send such data described by *XML*, and defined by an *XML* schema, as inputs or arguments to Web service methods. The method returns an object that also is represented by *XML* and is defined by a schema. Each Web service can define its own data structures but still use the generic *SOAP* framework for representing and delivering the request. This makes *SOAP* much richer and capable of representing more complex data, types, and classes. It elevates the computational model to an object-oriented model rather than simply sending data. We can send a matrix or a linear model, rather than sending its individual parts without the enclosing context and relationship between the parts, e.g., a vector of numbers or a vector of vectors for a matrix. This structure leads to more higher-level, more structured, and better software.

The ability to use application-defined data types in *SOAP* requests adds greatly to the complexity of the technology. Since there is not a fixed set of data types, we cannot use generic code to map each *R* input to *XML*, and vice versa. Instead, we need to map *R* inputs to *XML* expected by the *SOAP* method and defined by the schema. This added structure is beneficial in two respects. As we said, more structure leads to a more abstract model and to robust code that can check that conditions are satisfied. Secondly, and more importantly for us, the strict structure of the strongly-typed approach allows us to use the computer to generate, or "compile," code for the different data structures and remote methods, and to hide almost all of the details of converting *R* objects to the appropriate *XML* content. While there can be arbitrarily complex data types, they are all based on *XML* schema and the primitive data types that the schema standard defines, e.g., boolean, integer, real scalars, dates, times, and also arrays and structures. Again, we are building on shared *XML* technologies, and we can leverage the XMLSchema package [15] (see Chapter 14) to take care of most of the marshalling of *R* inputs to *SOAP* methods and the result back from *XML* to *R*.

12.2 The Basic Workflow: Working with *SOAP* in *R*

Unlike XML-RPC, most *SOAP* Web services provide a machine-readable description of both the methods they provide and the data structures involved as inputs and outputs. These descriptions are provided by a *WSDL* document. *WSDL* stands for the Web Service Description Language [3]. The *WSDL* document provides the information to fully describe all of the methods and the data types of their parameters and results. This information is enough for us to create *R* functions and classes for each of the Web service methods and data types described in the *WSDL* and so provide a programmatically generated interface to an entire Web service. This is similar to the approach and the code that we use in the XMLSchema package for mapping a schema to *R* classes and converter functions for reading and writing *XML*. Indeed, we leverage XMLSchema to do precisely this for *SOAP*.

The genSOAPClientInterface() function is the primary function in *R* for transforming a *WSDL* document, which describes a Web service's methods and data types, into *R* functions and classes.

The genSOAPClientInterface() function first uses the processWSDL() function to read the *WSDL* and its schema and develop descriptions of the Web server, the methods, and the data types. It then converts these descriptions into *R* classes and functions that mirror the Web service methods as local *R* functions.

Given a *WSDL* document (either as a local file, a *URL*, or in-memory document), we use the following command to generate the entire interface in *R*:

```
iface = genSOAPClientInterface(wsdlURL)
```

This defines the necessary *R* classes corresponding to the schema types used in the Web service methods, and returns a collection of functions that are interfaces with those methods. The variable `iface` contains the generated functions and is an *S4* object of class `SOAPClientInterface`. The functions are available in the *functions* slot and we can invoke them immediately within this *R* session or write them to a file for later use. We look at some examples.

12.2.1 Accessing the KEGG Web Service

The KEGG (Kyoto Encyclopedia of Genes and Genomes) [5] used to provide a *SOAP* interface for searching many genomic-related databases. According to the Web site, this API "allows customization of KEGG-based analysis, such as for searching and computing biochemical pathways in cellular processes or analyzing the universe of genes in the completely sequenced genomes." One can find information about organisms, genes, pathways, and motifs. We can use the Web service to query dynamic, up-to-date remote databases in structured ways.

The *WSDL* for the KEGG Web service is located at `http://soap.genome.jp/KEGG.wsdl`. We generate the interface with

```
uK = "http://soap.genome.jp/KEGG.wsdl"
iface = genSOAPClientInterface(uK)
```

We can find the names of the available functions with

```
names(iface@functions)
```

```
 [1] "list_databases"              "list_organisms"
 [3] "list_pathways"               "list_ko_classes"
 [5] "binfo"                       "bget"
 [7] "bfind"                       "btit"
 [9] "bconv"                       "get_linkdb_by_entry"
  ...
```

Each of these names corresponds to an *R* function that will invoke a method in the Web service.

We start by querying which databases KEGG consults. We do this by calling the list_databases() function:

```
dbs = iface@functions$list_databases()
```

The result is a list in *R* with 55 elements describing each database. The first two elements illustrate the general structure of the elements:

```
dbs[1:2]
```

```
[[1]]
An object of class "Definition"
Slot "entry_id":
[1] "nt"

Slot "definition":
[1] "Non-redundant nucleic acid sequence database"


[[2]]
An object of class "Definition"
Slot "entry_id":
[1] "aa"

Slot "definition":
[1] "Non-redundant protein sequence database"
```

The `dbs` object is actually of class `ArrayOfDefinition` and each element is an object of class `Definition`. A `Definition` object has two slots: *entry_id* and *definition*, both strings. These two classes were defined by genSOAPClientInterface() since the *list_databases()* method was described in the *WSDL* document as returning a schema array of Definition values. We have been able to preserve this rich structure of classes identifying the nature of the string values. Contrast this with just having a list consisting of character vectors with two elements.

The list_databases() function required no arguments, so we look at another method that does. The get_genes_by_organism() function returns a collection of gene identifiers for a particular organism. It takes a string identifying the organism and two numbers giving the starting and ending positions in the set of genes. These two numbers allow us to limit how many gene identifiers are returned in a given call. We query 10,000 genes for Homo sapiens. The identifier for Homo sapiens is `'hsa'` which we can find via a call to list_organisms(). We can invoke the get_genes_by_organism() method as

```
genes = iface@functions$get_genes_by_organism('hsa', 1, 10000)
```

The get_genes_by_organism() function we generated knows to convert the two numbers (1 and 10000) to integer values and to serialize these values and the string `'hsa'` in the correct manner expected by the server. The result is

```
head(genes)

[1] "hsa:728819"    "hsa:645954"    "hsa:283711"
[4]  "hsa:400645"   "hsa:100128416" "hsa:645142"
```

This is an array of string values that genSOAPClientInterface() mapped to a simple *R* character vector.

There are numerous examples of calling KEGG methods at http://www.kegg.jp/kegg/soap/doc/keggapi_manual.html.

Before moving onto a more involved example, we note that we can ask genSOAPClientInterface() to define the *R* functions as regular variables within the *R* global environment. We can then call the functions directly as, for example, `list_databases()` and `get_genes_by_organism('hin', 1, 100)` rather than having to access them via `iface@functions$list_databases()`, etc. We do this by specifying `TRUE` for the *putFunctions* parameter, e.g.,

```
genSOAPClientInterface(uK, putFunctions = TRUE)
```

This is more convenient for immediate use of the functions.

12.2.2 Accessing Chemical Data via the ChemSpider SOAP API

Mass spectrometry is a mechanism for measuring the mass-to-charge ratio of molecules. It can be used to identify both the mass of a compound and chemical structure or composition, i.e., the primitive elements it contains [16]. The site `http://www.chemspider.com` provides an open (for academic use) database of chemical structures that allows us "fast text and structure search access to over 26 million structures from hundreds of data sources" [8]. ChemSpider provides several Web services (see `http://www.chemspider.com/AboutServices.aspx?`), including one for searching for chemicals based on their molecular mass. The site provides a *WSDL* that describes several methods such as *GetDatabases()*, *SearchByMass2() SearchByMassAsync()* and *GetExtended-CompoundInfoArray()* and *GetExtendedCompoundInfo()*. We focus immediately on the last of these as it requires a token to process the *SOAP* request. We look at the others later to show how we obtained the identities of the compounds in which we are interested.

Some of these Web service methods are publicly available to everybody; others are free for academic users. For instance, both *SearchByMassAsync()* and *GetExtendedCompoundInfoArray()* methods require the caller to provide a token that grants them access to the information. We obtain this by registering with ChemSpider to obtain the free token which can be used for any of the restricted methods. We do this once, ahead of time and separately from any *SOAP* requests we make. We have done this and assigned the value of the secret token to the *R* variable `token` We can then pass this to the wrapper functions that require it, e.g., GetExtendedCompoundInfo(), along with any other parameters the method requires, and obtain the results. Here the authorization is given to the Web service by an argument in the call. We look at a different approach in Section 12.7 which puts the information at the level of the *SOAP* request and not the Web service method.

As before, we read the *WSDL* document and generate the wrapper functions for the Web service. We do this with

```
url = "http://www.chemspider.com/MassSpecAPI.asmx?WSDL"
massSpec = genSOAPClientInterface(url, putFunctions = TRUE)
```

Here we use *putFunctions* to have the functions assigned into *R*'s global environment. This will allow us to call them directly by name.

We are going to use the Web service to find compounds that have a mass value between 218 and 238 (atomic mass units) and then get additional information about each of these. For this, we first need to search the databases for compounds with a mass in this range. We need to specify which databases we want to search and for this, we need to know the names of the different databases. Once we have identified the compounds matching this mass range, we look up the complete information for each. The basic sequence of commands is given by the pseudo code

```
dbs =  GetDatabases()
csid = SearchByMass(mass, range, dbs, token)
info = GetExtendedCompoundInfoArray(csid, token)
```

We start by calling GetDatabases() to give us the names of all the databases. The call

```
dbs = GetDatabases()
```

returns a character vector, (currently) giving the names of 466 databases that ChemSpider can search:

```
[1]   "Abacipharm"          "Abblis Chemicals"
[3]   "ABI Chemicals, GmbH" "ACB Blocks"
[5]   "Accela ChemBio"      "ACD/Labs"
  ...
[465] "Zerenex Molecular "  "ZINC"
```

We can now use these database names, or a subset of them, to search for compounds with a mass between 218 and 238 atomic mass units. There are two methods in the Web service that we can use for this search: *SearchByMass2()* and *SearchByMassAsync()*. We use the latter in this example as: a) it allows us to specify which databases to search, unlike the *SearchByMass2()* method; and b) it allows us to illustrate additional *SOAP* requests. The "`Async`" in the name of the method means asynchronous. By this, we mean that we submit the request to search for compounds and the method returns almost immediately with a promise to continue to do the search in the background. The method returns us a ticket or identifier for our task/request and we can use this to periodically check if the task has been completed and/or to retrieve the actual results. In contrast, *SearchByMass()* returns the result. Essentially, with *SearchByMassAsync()*, we are running the task without having to wait for it. When the task is computationally intensive and takes some time, this frees us to do other tasks and return to get the results. It also allows the server to schedule the different tasks as it decides. The call to SearchByMassAsync() will ultimately yield the ChemSpider ID (CSID) values of compounds with masses within a given molecular range, and we will fetch this result with a different call given below.

We call the generated *R* SearchByMassAsync() function as

```
task.id = SearchByMassAsync(list(mass = 228., range = 10.,
                                 dbs = dbs, token = token))
```

We specify this molecular range by giving the midpoint of the interval and half of its width, i.e., mass + or − width. We also specify a vector of the database names in which to search for the matching compounds. Finally, we provide our token. The result of the call is the identifier for our task or request. We can use this to query the status of the request and to retrieve the result, but to do this we need to use the Search API that ChemSpider provides, which is another *SOAP* Web service.

We generate *R* wrapper functions for the Search API with

```
uCS = "http://www.chemspider.com/Search.asmx?WSDL"
genSOAPClientInterface(uCS, putFunctions = TRUE)
```

This generates the *R* functions and data types corresponding to that API's methods. We are ready to use them so we check the status with

```
GetAsyncSearchStatus(list(rid = task.id, token = token))
```

This call returns a string from a set of possible values describing the status and hopefully its value is `"ResultReady"`. That tells us we can retrieve the results with GetAsynchSearchResults(), e.g,

```
csids = GetAsyncSearchResult(list(rid = task.id, token = token))
```

This is an integer vector containing 79 CSID values. Each of these identifies a compound; the collection is the set of compounds with mass in our range of interest.

We can now obtain information about each of these compounds by calling GetExtendedCompoundInfo() and passing it the CSID identifying the compound and our token, e.g.,

```
info = GetExtendedCompoundInfo(CSID = 23543, token = token)
```

The resulting object is an instance of the class `ExtendedCompoundInfo` and has slots named

```
[1]  "CSID"            "MF"                "SMILES"
[4]  "InChI"           "InChIKey"          "AverageMass"
[7]  "MolecularWeight" "MonoisotopicMass" "NominalMass"
[10] "ALogP"           "XLogP"             "CommonName"
```

These slots provide details about the compound. The class `ExtendedCompoundInfo` was defined by genSOAPClientInterface() and corresponds to a type defined in the *WSDL*'s schema.

We can call GetExtendedCompoundInfo() for each of these compounds. This involves a separate *SOAP* invocation for each CSID and that many connections with the server. We can use a single `CURLHandle` to keep the connection open. We can even use multiple parallel curl requests using the curl "multi" interface. (See Section 8.5 in Chapter 8 on advanced RCurl topics.) However, in this situation, the *SOAP* API provides a "vectorized" method, GetExtendedCompoundInfoArray(), that takes an array of CSIDs and returns an array with information for each of them. We can call this method with

```
info = GetExtendedCompoundInfoArray(CSID = csids, token = token)
```

and the result is the same as if we called GetExtendedCompoundInfo() separately for each of the elements of `csids`.

12.2.3 Other Useful Features of *genSOAPClientInterface()*

Many readers might expect that we would write the code (functions and classes) to a file and then source() them into *R* rather than creating the functions and classes directly in *R*. We could then, for example, put this code in an *R* package. In some ways, this is desirable as we would only have to process the *WSDL* and generate the code once, and then multiple users could use it without needing to go through these initial steps. The function writeInterface() is used to write the code to a file. We pass it both the object returned from genSOAPClientInterface() and the name of the file to which the code is to be written, e.g.,

```
iface = genSOAPClientInterface(uK)
writeInterface(iface, "KEGG.R")
```

We can then source() this function back into *R* or have it as a file in an *R* package and have it processed when the package is installed and loaded into an *R* session.

This approach is often quite convenient and for the most part adequate. However, we should note that if the Web service's methods, and hence the *WSDL*, change after we generate the code, this code will no longer be the appropriate code to use. It may lead to odd and confusing errors or, even worse, yield the wrong results. For this reason, it is good practice to generate the code when it is needed. Of course, code that used to work with a previous version of the Web service may then fail, but this is always a problem with client-server computing.

We can also save the object returned by genSOAPClientInterface() and then restore that in another *R* session. Similarly, we see below that we can save the functions in an environment and load that into another *R* session and use them there.

Controlling where the Functions and Classes Are Defined

As we saw in the previous section, genSOAPClientInterface() returns an object that contains a list of the *R* functions that correspond to the Web service methods. These can then be accessed by name in this list, e.g.,

```
iface@functions$list_databases
```

and invoked. In many cases, it is convenient to keep the functions grouped together. In other cases, however, *R* users may prefer to have them as regular top-level functions in the *R* work space. We can arrange for genSOAPClientInterface() to also do this for us via the *putFunctions* parameter. Passing `TRUE` for this causes the functions to be defined in the environment specified by the *where* parameter, which by default is the global environment, i.e., globalenv(). If we want to put the functions in a different environment, we can specify the environment explicitly, e.g.,

```
codeEnv = new.env()
codeEnv$.packageName = "KEGG"
iface = genSOAPClientInterface(uK, putFunctions = codeEnv)
```

or

```
iface = genSOAPClientInterface(uK, where = codeEnv,
                               putFunctions = TRUE)
```

The environment `codeEnv` would then contain all of the functions and also any classes and methods that genSOAPClientInterface() defines. We can access the functions with `get("list_databases", codeEnv)` or more conveniently with `codeEnv$list_databases`. Alternatively, we can add the environment to the search path using attach() and then refer to the functions as variables:

```
attach(codeEnv)
list_databases()
```

It is possible to put the functions and the classes and methods in different environments. We can do this by specifying different environment objects for *where* to control the location of the classes and for *putFunctions* to house the functions. It is important that the functions can find those classes and methods, and so the environments must be "related" or creatively connected.

We can use this ability to write the classes and functions to an environment in order to serialize the functions to a file and then use them in another *R* session. For example,

```
codeEnv = new.env()
iface = genSOAPClientInterface(uK, where = codeEnv,
                               putFunctions = TRUE)
```

puts the classes and functions into `codeEnv`, an environment we created. We serialize the environment `codeEnv` to a file

```
save(codeEnv, file = "codeEnv.rda")
```

and then load it into another *R* session, either on this or another machine with

```
library(SSOAP)
load(codeEnv)
```

We can then invoke the functions, e.g.,

```
codeEnv$list_databases()
```

to call the list_databases() function. This is another way to distribute the generated functions. (Currently, the classes are not properly defined when we load the environment.)

Specifying the *WSDL* in Different Ways

We typically read the *WSDL* document directly from a *URL* so that we are guaranteed to get the

most up-to-date version that corresponds to the actual Web service that is available. However, since the *WSDL* is being processed by the xmlParse() function in the XML package [11], we can specify the *WSDL* in any form that function understands. This means we can generate the interface using a local file or even a compressed (GZIP) file by passing its name to either processWSDL() or genSOAPClientInterface(). Similarly, we can pass the actual content of the *WSDL* document having read it into *R* via, e.g., a nontrivial *HTTP* request. For instance, if the *WSDL* document is available only via secure *HTTP* (i.e., *HTTPS*), then xmlParse() cannot retrieve it. However, we can retrieve it with

```
wsdlText = getURLContent(uWSDL, ssl.verifypeer = FALSE)
```

Then we can pass this string to genSOAPClientInterface() as

```
doc = xmlParse(wsdlText, asText = TRUE)
iface = genSOAPClientInterface(doc)
```

12.3 Understanding the Generated Wrapper Functions

Here we examine the functions that genSOAPClientInterface() creates to interface to each of the methods. It is useful to understand their design so that we understand what they are doing with our inputs and how they make the *SOAP* request and marshal the inputs and outputs to and from *XML*.

Let's look at the KEGG method get_genes_by_organism again. This took three arguments — the string identifying the organism, a number giving the starting gene in the list, and a second number giving the final position in the list to return. It returns an array of string elements. The method can be thought of, in pseudo code, as

```
character vector
  get_genes_by_organism again(
                      string org, integer offset, integer limit)
```

The *R* function corresponding to this description in the *WSDL* is given by

```
iface@functions$get_genes_by_organism

An object of class "WSDLGeneratedSOAPFunction"
function (org, offset, limit, server = .defaultServer,
          .convert = .operation@returnValue,
          .opts = list(), nameSpaces = "1.2",
          .soapHeader = NULL,
          .header = SSOAP:::getSOAPRequestHeader(
                              .operation@action, .server = server),
          curlHandle = RCurl::getCurlHandle())
{
  .SOAP(server, .operation@name,
        .soapArgs = list(org = as.character(org),
                         offset = as.integer(offset),
                         limit = as.integer(limit)),
        action = .operation@action,
        xmlns = .operation@namespace,
```

```
        .types = .operation@parameters,
        .convert = .convert,
        .opts = .opts, nameSpaces = nameSpaces,
        .elementFormQualified = TRUE,
        .returnNodeName = "return", .soapHeader = .soapHeader,
        .header = .header,  curlHandle = curlHandle)
}
<environment: 0x7fb44c55c6d0>
```

The first three parameters in the *R* function are the Web service method's parameters, i.e., *org*, *offset*, and *limit*. They have no default values and so must be provided by the caller. The only command in the body of the function is a call to .SOAP() and that call passes the values of these three arguments via the *.soapArgs*. This is passed as

```
.soapArgs = list(org = as.character(org),
                 offset = as.integer(offset),
                 limit = as.integer(limit))
```

We see that the function is converting the values to the target types—a string, and integers for the other two arguments. These are simple primitive types so we can use as.character() and as.integer(). If the expected data types were more complex classes, e.g., defined by the *WSDL* document, the generated code would use the as() function to coerce the *R* value for the parameter to the expected target type. We can then serialize that object to *XML* in the manner expected by the server. Since the generated functions coerce the individual arguments to their target types, users can define their own coercion methods from different *R* types to those target classes. They can use setAs() to specify these coercion methods and they then will be used where appropriate. When the target class is an *S4* class, there is a built-in coercion mechanism for mapping a list of named elements to an *S4* class. This is generic code that uses names of the elements in the *list* as the slots of the target object. It uses the type of each slot to coerce the *R* element in the *list* to that type. This allows us to call the generated functions without actually creating the *S4* objects expected by the function.

Each generated function also has additional parameters such as *.soapHeader*, *.opts*, *curlHandle*, and even *.convert*. You can read about the purpose of each of these in Section 12.5. It suffices to point out here that we can pass these to our generated function, and they are passed to the .SOAP() call.

There are several additional things to note about the get_genes_by_organism() function and all of the functions created and returned by genSOAPClientInterface(). It has a class named `WSDLGeneratedSOAPFunction`. This identifies it as a function generated from a *WSDL* for *SOAP*.

The function also has an environment that is displayed at the end of the function, e.g., `<environment: 0x7ff6cef50228>`, although the value will change for each *R* session. This is important since the "free" variables describing the Web service method are contained in the environment (and its parent environment) and are available to the body of the function when it is called. Specifically, the variables `.defaultServer` and `.operation` are not local, but are in the environment of the function or its parent environment. This allows us to share the server across the functions generated by genSOAPClientInterface() and also slightly simplify each function.

This function-specific environment causes a minor problem when we write one of these generated functions as text to a file via writeInterface(). There are several approaches to dealing with this. We can arrange to write the environment and its contents to the same file to which we write each function, and then add code to associate that environment with the function. Another approach, and the one we use when writing the functions to a file, is to put the "free" variables—`.operation`, `.header`, `.convert`, and `.defaultServer`—into the function, and remove the need for the environment. We

can insert the values into the body of the function where they are used. However, instead, we add them as formal parameters to the function with default values that are those currently in the environment. This allows the caller to override them if desired. The result is that the functions have more parameters than those generated directly in the *R* session via genSOAPClientInterface(). However, there is no difference in how they perform, by default. We have chosen to use the environments when we create the functions within an *R* session via genSOAPClientInterface() as this makes the code more readable. However, we add the free variables to the formal parameters when we write the functions to a file as text.

There is no issue with the environment if we serialize the functions via save() and restore them with load().

12.4 The Basics of *SOAP*

While we can use genSOAPClientInterface() and the generated functions to invoke remote *SOAP* methods, it is sometimes important to understand the basic structure of the *SOAP* mechanism. At its simplest, we send an *XML* document that describes the method call to the *SOAP* server. While there are many different transportation protocols for communicating with the server, the most common is *HTTP* or *HTTPS* and we use a **POST** request to send the *XML* document as the body of the *HTTP* **POST** request. The root node of the document is `<Envelope>`. Similar to an *HTML* document, it consists of an optional `<Header>` node and a single, required `<Body>` node. The `<Body>` contains a node identifying the method name and, within that, an *XML* serialization of each of the arguments to the method.

It is instructive to look at what is actually sent and received during a *SOAP* request. Consider our example that called the KEGG method *get_genes_by_organism()* which expects three arguments—the name of the organism as a string, and two integers identifying the start and end positions within the list of genes to return. The following is what the .SOAP() function sends

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <get_genes_by_organism xmlns="SOAP/KEGG">
   <org xmlns="SOAP/KEGG" xsi:type="xsd:string">hsa</org>
   <offset xmlns="SOAP/KEGG" xsi:type="xsd:int">1</offset>
   <limit xmlns="SOAP/KEGG" xsi:type="xsd:int">10</limit>
  </get_genes_by_organism>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The topmost node is the `<Envelope>`. Its namespace is the `http://schemas.xmlsoap.org/soap/envelope/` which is declared in the node. We also define namespaces for *XML* schema (`xsi` and `xsd` in this snippet).

The <Body> node is the only child of the <Envelope> node in this example. We can also have a <Header> node, but this is not common for public APIs and basic *SOAP* interfaces. Within the <Body> node, we have the details of the method invocation. We have a node with the name of the method, i.e., <get_genes_by_organism>. Note that this also defines its own namespace (SOAP/KEGG). This namespace is for the application-specific content—method and data types—defined in the *WSDL*. In this example, the namespace is not a URI but a simple string SOAP/KEGG. This is unusual.

Within the node identifying the name of the method, we have an element for each argument to the method. Next, we look at how we represent an *R* vector as *XML* in a *SOAP* call. When we call the *get_pathways_by_genes()* Web service method, we pass it a collection of gene identifiers. For example,

```
iface@functions$get_pathways_by_genes(c('eco:b0077' , 'eco:b0078'))
```

The method part of the *SOAP* envelope is

```
<get_pathways_by_genes xmlns="SOAP/KEGG">
 <genes_id_list xmlns="SOAP/KEGG" xsi:type="SOAP-ENC:Array"
                SOAP-ENC:arrayType="xsd:string[2]">
  <item xsi:type="xsd:string">eco:b0077</item>
  <item xsi:type="xsd:string">eco:b0078</item>
 </genes_id_list>
</get_pathways_by_genes>
```

There is a single argument named "genes_id_list". This is an array of string items as described by its *arrayType* attribute. In this call, it has two elements so this length is added to the array description, i.e., "xsd:string[2]". Each element of the *R* character vector is mapped to an <item> node that has a *type* attribute identifying the nature of the value. Then the node houses the value itself. In this case, each <item> node contains just a string. However, generally, each <item> node in the array, and indeed each node corresponding to an argument in the call, can be a complex *XML* node with subnodes and attributes that represent the data. This is described in the *XML* schema as part of the *WSDL* document. We can see an example of this in the way the *SOAP* result is represented in *XML*.

Consider the call to the list_databases method in the KEGG API. This call takes no arguments and so the <Body> of the *SOAP* request is just

```
<SOAP-ENV:Body>
  <list_databases xmlns="SOAP/KEGG"/>
</SOAP-ENV:Body>
```

That is, the node identifying the name of the method being called has no children. The interesting part for us is the *XML* returned from the *SOAP* method. An abbreviated version is

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <soap:Body>
  <list_databasesResponse xmlns="SOAP/KEGG">
   <return xmlns:r="urn:SOAP/KEGG"
```

```
      soapenc:arrayType="r:Definition[55]"
      xsi:type="soapenc:Array">
    <item xsi:type="r:Definition">
     <entry_id xsi:type="xsd:string">nt</entry_id>
     <definition xsi:type="xsd:string">
       Non-redundant nucleic acid sequence database
     </definition>
    </item>
    <item xsi:type="r:Definition">
     <entry_id xsi:type="xsd:string">aa</entry_id>
     <definition xsi:type="xsd:string">
       Non-redundant protein sequence database
     </definition>
    </item>
        ......
  </list_databasesResponse>
 </soap:Body>
</soap:Envelope>
```

Again, the root node is <*Envelope*> and it contains namespace definitions for various different vocabularies used in the content. The <*Body*> node contains the response, and its child is <*list_databasesResponse*>. The name `list_databasesResponse` identifies the purpose of the node, but this form of the node name is not mandated by the *SOAP* specification. The name of the node identifying the response is defined within the *WSDL*. The <*return*> child node is the one that contains the actual results. As with all the *XML* elements in the result, the <*return*> node has a *type* that identifies the data type it contains. In the case of <*return*>, this is an `"Array"` type and the *arrayType* attribute tells us it is an array of 55 `Definition` objects. Our *R* function actually knew that the result was an array of elements of type `Definition` from the *WSDL* document. However, this information is included in case we are working directly without a *WSDL* document. There are 55 <*item*> child elements of <*return*>. Each of these describes a `Definition` object. This object is defined in the *XML* schema in the *WSDL* document as

```
<xsd:complexType name="Definition">
 <xsd:all>
  <xsd:element name="entry_id" type="xsd:string"/>
  <xsd:element name="definition" type="xsd:string"/>
 </xsd:all>
</xsd:complexType>
```

This schema node simply means that a <*Definition*> object has two components, <*entry_id*> and <*definition*>, and both are string values. The genSOAPClientInterface() function maps this definition to an *S4* class with two string slots with those names.

A *SOAP* <*Envelope*>, either in the request or response, can have an optional <*Header*> element. This element can contain various types of information, and there can be multiple blocks or parts to the <*Header*>. Each block in the header provides some information about the context of the call. This might be authentication information such as a login and password identifying the caller. It might also be a digital signature for the <*Body*> of the request. It might contain information about how to direct the request within the Web server. The format of the <*Header*> elements is very flexible and the server defines what it expects and understands. Whether the receiver must be able to make sense of the contents of the <*Header*> before proceeding can be controlled by the *mustUnderstand* attribute.

The *SOAP* header is different from the *HTTP* header. The latter can contain standard *HTTP* fields such as *User-Agent*, *Accept*, and so on. For a *SOAP* request, however, the *HTTP* request must provide the action string in the header of the request via the *SOAPAction* field, e.g.,

```
POST /MassSpecAPI.asmx HTTP/1.1
Host: www.chemspider.com
Accept: text/xml
Accept: multipart/*
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://www.chemspider.com/GetDatabases"
Content-Length: 439
```

12.5 The .SOAP() Function

When we make a *SOAP* call, either using a function programmatically generated from a *WSDL* document or directly ourselves, the actual call is done via the .SOAP() function. Accordingly, it is sometimes useful or necessary to understand how the .SOAP() function works. This is the function that actually makes the *SOAP* requests, even for the wrapper functions we generate from the *WSDL* document. For instance, in some cases, we will either not have a *WSDL* document describing the Web service methods and data types, or alternatively we may decide not to use genSOAPClientInterface() to generate the *R* code to call the Web service methods. Instead, an *R* user can invoke a Web service method by directly calling the .SOAP() function or writing their own functions that call it. The .SOAP() function is the primary function for invoking *SOAP* requests. It takes care of creating the *XML* document representing the request by serializing the *R* arguments and then sends the message via *HTTP*. By default, it also attempts to convert the result to an *R* object, either as a regular result or a *SOAP* error. This section discusses the .SOAP() function and its parameters and describes how we can use them to control how the *SOAP* requests are made.

The essential arguments for a call to .SOAP() are the:

1. *URL* for the method,
2. name of the Web service method being invoked,
3. arguments for the Web service method, and
4. *SOAP* action, required by *SOAP*.

The following is a call to the *getsubset()* method provided by the MODIS, Moderate Resolution Imaging Spectroradiometer, service [6]. MODIS is a satellite instrument that acquires data in 36 spectral bands with the goal of studying global dynamics on the land, oceans, and lower atmosphere. When calling any of the service's methods, we send the *SOAP* request to the same *URL*, i.e., `http://daac.ornl.gov/cgi-bin/MODIS/GLBVIZ_1_Glb_subset/MODIS_webservice.pl`, which we store in the *R* variable `MODIS_URL`. We pass seven arguments in the call to the *getsubset()* method.

```
.SOAP(MODIS_URL,  1
      "getsubset",  2
      40.115, -110.025,  3
      "MOD11A2",  4
      "all",  5
      "A2001001", "A2001025",  6
```

```
    1, 1,  7
    action = MODIS_Action)  8
```

1 We specify the *URL* for the top-level MODIS service.
2 The second argument to .SOAP() identifies the method, i.e., *getsubset()*.
3 The *getsubset()* method has seven arguments; the first two are the latitude and longitude of the location for which we want the data.
4 The string `"MOD11A2"` indicates which of the available data "products" or systems to query.
5 The string `"all"` indicates that we want all of the available bandwidths.
6 These two arguments (`"A2001001"` and `"A2001025"`) give the start and end dates in which we are interested.
7 The last two arguments (1 and 1) give the size (in kilometers) of the box around the location for which we want the data.
8 The action argument identifies the target of the *SOAP* request for the *HTTP* server. The Web server can examine the header and determine how to process the *SOAP* request using this information, without having to look at the body of the request. The `MODIS_Action` argument has the value `"http://daac.ornl.gov/MODIS_webservice#getsubset"`

12.5.1 The *server* Parameter

The *server* parameter identifies the *URL* to which the request will be sent. This can be specified in several different forms. We can identify the *URL* either as a string, or alternatively, via an object of the general class `SOAPServer`. The latter is an *S4* class that identifies the host name or IP (Internet Protocol) address, the port, and the protocol (e.g., *HTTP*) for communicating with the sever. The subclasses and constructor functions support *HTTP*, *HTTPS*, and FTP. We can also specify the *server* as a curl connection object (i.e., of class `CURLHandle`) However, this `CURLHandle` object must have the *url* option already set, or the *url* must be specified in the *.opts* argument passed to this .SOAP() call. The *.opts* argument is used to customize the *HTTP* request used to send the *SOAP* request. Typically if a `CURLHandle` object is used, one sets the server once and reuses the connection across multiple calls. This can reduce the overhead of re-establishing the connection with the server each time and also of resetting all of the options. It also allows us to enable cookies that persist across multiple calls to the same server.

While we can use an existing `CURLHandle` as the *server* argument and explicitly set its parameters before the call, we can also use the *.opts* parameter to pass configuration parameters for the curl request. These are then passed to a new `CURLHandle` object created for the *SOAP* request. As with creating any `CURLHandle` object, any settings specified in the *R* option named *RCurlOptions* will be used, unless overridden by values in *.opts*. Useful default settings include options such as *followlocation* to follow any redirections by the Web server and *cainfo* for *SSL* certificates used to verify the authenticity of the Web server when using *HTTPS*.

12.5.2 The *method* Parameter

The *method* argument is very simple. It is a string that identifies the *SOAP* method being called. This is used when creating the *XML* content for the *SOAP* request.

12.5.3 Arguments for the SOAP Method: ... and .soapArgs Parameters

The arguments in a call to the .SOAP() function can be divided into three separate groups: values to be passed as arguments to the *SOAP* method on the server, arguments controlling the *SOAP* request, and other arguments that are local to the *R* function .SOAP(). Here we are focusing on the first of these—the arguments to the *SOAP* method. These arguments can be passed via the ... mechanism. The names of these arguments are used when creating the *XML* representation of each argument. Specifically, the name of the argument is used as the name of the node holding that value. For example, in the call

```
.SOAP(server = "http://www.chemspider.com/Search.asmx",
      method = "SimpleSearch", query = "Azithromycin",
      token = token,
      action = I("http://www.chemspider.com/SimpleSearch"),
      xmlns= "http://www.chemspider.com/")
```

the *XML* representation for the two arguments is

```
<ns:query xsi:type="xsd:string">Azithromycin</ns:query>
<ns:token xsi:type="xsd:string">
xxxxx-xxxx-yyy-xxxxxxxx
</ns:token>
```

That is, the names of the arguments to the Web service method are `"query"` and `"token"` and these are used in the *XML* content. (The `token` argument is a string in *R* that holds our secret key that allows us access to some of the restricted methods in the ChemSpider API.)

For a call to .SOAP(), the arguments for the *SOAP* method should be specified in the order they are expected by the *SOAP* server for that method. *R* has no way of matching names of arguments for the *SOAP* method at this level as it has no description of the method itself. Indeed, *R* also does not know the expected type of each parameter of the method. The .SOAP() function will simply serialize the *R* object using a general approach, mapping the *R* values to their corresponding type using the generic *XML* schema. This is one of the significant advantages of using a *WSDL* and genSOAPClientInterface() to generate a wrapper function that calls .SOAP(). That wrapper function can match the arguments by name and will coerce each of them to the expected type. For example, if a method expects an integer, but we pass a value such as `1` in a call to .SOAP(), this will be sent as a real-valued number in the *SOAP* request since `1` is of type *numeric* in *R*, not an *integer*. The generated wrapper function, however, would first coerce the argument to the expected `integer` type and so the request would be correct.

When calling .SOAP() directly, the *R* programmer can specify the arguments for the Web service method directly in the call via *R*'s ... mechanism. In some cases, this is not desirable. Firstly, suppose a Web service method had a parameter named `"server"`. This is the same name as .SOAP()'s first parameter. In a call such as `.SOAP("http://www.server/a/b", server = 1)`, *R* would use the value 1 as the *URL* to which the *SOAP* request will be sent. It will match the actual *URL* in the first argument to a different parameter, the method name in this case. This will cause confusion or incorrect results. The problem is that we have two sets of parameters, those of .SOAP() and those of the Web service method. We are better off separating them explicitly rather than using *R*'s ... mechanism and relying on distinct names.

The second reason that we may not want to use the ... mechanism is that the arguments may already be in an *R* `list`. For example, they may be returned by a call to another function as a single collection of arguments. Alternatively, we may generate them by a call to lapply(). Rather than performing gymnastics to create the call (e.g., via `do.call`), we can specify the arguments for the

SOAP method via the *.soapArgs* parameter of the .SOAP() function. For example, we can make our earlier .SOAP() call as

```
args = list(40.115, -110.025, "MOD11A2", "LST_Day_1km",
            "A2001001", "A2001025", 1, 1)
.SOAP(MODIS_URL, "getsubset", .soapArgs = args,
       action = MODIS_Action)
```

In this approach, we can treat these arguments as a single unit, but continue to use the regular *R* mechanism for specifying the other *R* arguments controlling how the *SOAP* document and *HTTP* request are processed. Here the arguments are not named. However, in the call at the start of this section to the *SimpleSearch()* method with the token and query parameters, we would pass these as

```
.SOAP(server = "http://www.chemspider.com/Search.asmx",
      method = "SimpleSearch",
       .soapArgs = list(query = "Azithromycin",  token = token))
```

There is now no chance of a conflict between the method's parameter names and those of .SOAP().

12.5.4 The action Parameter

The *action* parameter is used in the header of the *HTTP* request. It identifies the request as being a *SOAP* operation and enables the Web server to handle the request in a different manner than it typically would for any non-*SOAP* request. For instance, it can dispatch the request to a specific generic *SOAP* script rather than to the actual *URL* in the request. It might use the local path in the *URL* to allow the generic *SOAP* script to behave differently. Alternatively, the server might use the content of the action to identify a specific script for the *SOAP* method. Of course, the Web server can already do this with the name of the *SOAP* method. However, it would have to process the body of the request to determine this name. Putting the action in the *HTTP* header reduces the processing for the Web server to identify and redirect *SOAP* requests.

For many services, the action is computed by concatenating the *URL* for the request and the method's name, separated by the '#' character, and the .SOAP() function uses this if we do not explicitly specify an action. This is a convention and Web services can use any action string for a method. We typically get this from the *WSDL* document and do not need to concern ourselves with knowing the actual value of the action string. However, when we are calling .SOAP() directly, we may need to explicitly specify the action. In some cases, this will be a string we obtain from the documentation for the Web service. In other cases, the default value may be correct, but we may need to have it in quotes in the value of the *SOAPAction* header field. A somewhat common situation is that the action for a given method consists of a common string used for all methods but concatenated with the method name, separated by '#'. The .SOAP() function tries to assist here by allowing the caller to specify the common string, and the function appends the method name to form the final value to use for the action. However, this raises a problem when we want to specify the action string directly without the .SOAP() function appending the method name to it. To do this, we pass the string as an object of class `AsIs`. We do this by enclosing the string in a call to the I(), e.g.,

```
action = I("http://www.chemspider.com/SimpleSearch")
```

This causes .SOAP() to use the string without any modifications.

The observant reader may notice that in our initial example of this section when calling the *getsubset()* method in the MODIS Web service, we did not need to use I() for the action even though it

was not the default. The reason for this is that the value is a *URL*. The .SOAP() function recognizes a *URL* and assumes it should be left as is. However, it is advisable to explicitly use the I() to mark the value as `AsIs` rather than rely on .SOAP() determining what to do.

12.5.5 Passing Curl Options via the .opts Parameter

The .SOAP() function sends the *SOAP* request via an *HTTP*(S) **POST** request. It does this using the RCurl package [12] and ultimately its curlPerform() function. We can specify a multitude of different options that control this communication between *R* and the Web server. We can specify these options for curlPerform(), etc., via the .SOAP() function's *.opts* parameter. For example, we can enable verbose reporting of the request and specify a cookie to send in the request with

```
v = .SOAP(MODIS_URL, "getsubset", 40.115, -110.025, "MOD11A2",
          "LST_Day_1km", "A2001001", "A2001025", 1, 1,
          .opts = list(verbose = TRUE, cookie = "my cookie"),
          action = MODIS_Action)
```

We can also populate a curl handle with these options and pass that handle to .SOAP() via the *curlHandle* parameter. We can do the same as above with

```
curl = getCurlHandle(verbose = TRUE, cookie = "my cookie")
v = .SOAP(MODIS_URL, "getsubset",
          40.115, -110.025, "MOD11A2", "LST_Day_1km",
          "A2001001", "A2001025", 1, 1,
          curlHandle = curl, action = MODIS_Action)
```

One potential point of confusion is in specifying values for the header in the *HTTP* request via the curl option *httpheader*. The .SOAP() function sets this curl option itself using the value of its own *.header* parameter. This defaults to, for example,

```
Accept: text/xml
Accept: multipart/*
Content-Type: text/xml; charset=utf-8
SOAPAction: "SOAP/KEGG#get_genes_by_organism"
```

Here the action is the value of the *action* parameter. Since .SOAP() sets this *httpheader*, we cannot set its value via the curl options (*.opts*). Instead, if we want to change it, we have to specify it via .SOAP()'s *.header* parameter. To do this, we can either call the getSOAPRequestHeader() to get the default values and then add to those, or else we can create the character vector ourselves, making certain to add the appropriate *SOAPAction* entry. We should note that we can specify other curl parameters that appear in the *HTTP* header, such as *useragent* or *cookie*, in the *.opts* argument. It is just the *httpheader* option we cannot specify via *.opts*.

12.5.6 The .convert Parameter

By default, the .SOAP() function processes the result returned to it by the Web server and converts it to the appropriate *R* value. There are several reasons why a call to .SOAP() may fail and we discuss these in Section 12.6. Here we focus on the case where the call to the Web service method was

successful and how .SOAP() attempts to convert the result to an *R* object. The *.convert* parameter is used to control how the conversion of the *XML* content is processed.

The basic value for this *.convert* argument is either `TRUE` or `FALSE`. For `TRUE`, the standard deserializing of *XML* to *R* is performed, which is to find the relevant *XML* node in the body of the result and pass it to the fromXML() function. For `FALSE`, the *XML* content is returned as a string and the caller can process it as he or she wants.

When we have defined classes to represent the data types for the *SOAP* types, we can pass a simple string as the value for *.convert*. This should specify the name of an *R* class and the *XML* will then be converted to an object of that class. This will then call the method coerce() for coercing an *XML* node to this class. When we process a *WSDL* document with genSOAPClientInterface(), methods will be defined for doing this coercion from an *XML* node to each of the new classes. However, we can also specify our own coercion methods for different classes and easily control how the .SOAP() converts the result, given just the name of the target class. This approach allows us to use a global method for coercing *XML* nodes to a particular class that is available throughout an *R* session and across .SOAP() calls.

An alternative to specifying a string and using global coercion methods is to specify an *R* function as the value for *.convert*. By default, this function is called with a single argument which is the *XML* node containing the result from the Web service method. The .SOAP() function finds the actual node in the document returned by the server and then passes this to the user-specified coercion function. The function can then extract the relevant data it wants and return that as an *R* object, or even a collection of *XML* nodes. How does the .SOAP() function find the node containing the result in the *XML* document? Firstly, there is a sensible default. It is typically the single grandchild of the *<Body>* node. If this is not the case, we can specify the name of the node using the *.returnNodeName* parameter.

The approach of using a function for *.convert* allows us to control the conversion for an individual call, not all calls. The function only applies for the particular call to .SOAP(). The coercion method applies to all calls to .SOAP() and any *R* commands that coerce an *XML* node to the target class.

There are circumstances in which we want the entire *XML* response from the Web service method and not just the result node that the .SOAP() function identifies. For example, we may want access to the complete result to get at the *<Header>* node in the *SOAP XML* message or fields in the header of the *HTTP* response. In such cases, we pass a function that inherits from the class `RawSOAPConverter`. In this case, the function will be passed all of the information from the *HTTP* response, i.e., both the header information of the *HTTP* response and also the content of the response, as a string. This allows the function to process the entire *HTTP* response in arbitrary ways. For example, the function can process any additional content in the *XML* response, and also examine the fields in the *HTTP* header. If we are only interested in the *XML* in the response, the function can parse this document and then process this using the usual functions from the XML package.

A convenient way to specify a function of class `RawSOAPConverter` is to enclose it in a call to structure(), e.g.,

```
v = .SOAP(MODIS_URL, "getsubset",
          40.115, -110.025, "MOD11A2", "LST_Day_1km",
          "A2001001", "A2001025", 1, 1,
          .convert = structure(function(x) { ... },
                               class = "RawSOAPConverter"),
          action = MODIS_Action)
```

If no more specific form of conversion is specified, then .SOAP() uses a simple but effective mechanism to transform the *XML* into an *R* object. This amounts to parsing the *XML* *<Envelope>*,

finding the main node in the <Body> and then passing this to the function fromXML() in the XMLSchema package. While this is a generic function and can be given a description of the target type in the form of a schema type, the default conversion does not have the target type and so processes the *XML* content based on the name of the top-level *XML* node in the result. If there is no method for fromXML() for this target type, the conversion will end up in the default method for fromXML() and this function attempts to interpret the *XML* tree as an *R* named list. The fromXML() function iterates over the children of the node, converting each of those by calling fromXML() recursively. The important part of the function is that it examines each node's *type* attribute (if present) which indicates the nature of the value in the node. These use the *XML* schema types and allow us to distinguish strings from booleans, integers, floats, dates, times, and arrays. For example, the three *XML* nodes

```
<id>123</id>
<id xsi:type="xsd:int">123</id>
<id xsi:type="xsd:float">123</id>
```

have the same content (123), but the fromXML() function knows to treat the latter two as an integer and a real-valued number, respectively, and leave the first as a string as there is no type information.

We look now at the *XML* that is returned by our call above to the *getsubset()* method in the MODIS API. (Note that we have stripped the first three columns in each <item> node in order to fit on the printed page.)

```
<tns1:getsubsetReturn
  xmlns:soapenc=
    "http://schemas.xmlsoap.org/soap/encoding/"
  xsi:type="tns1:ModisData">
 <xllcorner
   xsi:type="xsd:float">-9357990.22</xllcorner>
 <yllcorner xsi:type="xsd:float">4458921.58</yllcorner>
 <cellsize
   xsi:type="xsd:float">926.62543305583381</cellsize>
 <nrows xsi:type="xsd:float">3</nrows>
 <ncols xsi:type="xsd:float">3</ncols>
 <band xsi:type="xsd:string">all</band>
 <units xsi:type="xsd:string"/>
 <scale xsi:type="xsd:float"/>
 <latitude xsi:type="xsd:float">40.115</latitude>
 <longitude xsi:type="xsd:float">-110.025</longitude>
 <header xsi:type="xsd:string">
  HDFname,Product,Date,Location,Processed_Date,...
 </header>
 <subset soapenc:arrayType="xsd:string[4]"
   xsi:type="soapenc:Array">
  <item xsi:type="xsd:string">
2006350190424,0,64,64,0,0,0,0,0,0
2006350190424,254,254,254,254,254,254,254,254,254
2006350190424,255,79,79,255,255,255,255,255,255
2006350190424,255,113,113,255,255,255,255,255,255
  </item>
  <item xsi:type="xsd:string">
```

```
2006352085954,32,32,32,32,32,32,32,32,32
2006352085954,37,37,101,37,37,37,37,37,101
2006352085954,89,89,89,89,89,89,89,89,90
2006352085954,114,114,114,114,114,114,114,114,114
  </item>
  <item xsi:type="xsd:string">
2006354035405,2,2,2,2,2,2,0,0,0
2006354035405,183,183,183,183,183,183,183,183,183
2006354035405,45,45,45,45,45,45,255,255,255
  </item>
  <item xsi:type="xsd:string">
2006355205007,160,161,161,160,161,161,161,161,161
2006355205007,211,211,210,210,210,210,210,210,210
2006355205007,79,72,72,79,72,72,72,72,72
  </item>
 </subset>
</tns1:getsubsetReturn>
```

We can see that there are nodes such as <*xllcorner*>, <*yylcorner*>, <*cellsize*>, <*nrows*>, <*band*>, <*units*>, and <*header*> that contain single scalar values. (The <*header*> element here is a comma-separted list of strings, but is itself identified as a single string via its *type* attribute.) The <*subset*> node is a container for four <*item*> nodes, each of which is a string. It is very natural to map this entire *XML* tree to an *R* list with an element for each of the immediate child nodes of the root (i.e., <*getsubsetReturn*>). The names of the elements should be the names of the nodes. The type of each element is determined from the <*type*> attribute on the node. Only the <*subset*> node contains a nonscalar value. We can map this to *R* in the same manner as the root node, i.e., by converting each of its children and creating a named list. This is what fromXML() does. However, it goes further and recognizes that it can reduce this list to a character vector since all the elements have the same type in *R*. This is the same simplification that sapply() does to reduce a list to a homogeneous vector without losing information.

The result of calling fromXML() on this <*getsubsetReturn*> node is a list with elements described as

```
sapply(fromXML(node), class)
```

```
xllcorner   yllcorner    cellsize       nrows
"numeric"   "numeric"   "numeric"   "numeric"
    ncols        band       units       scale
"numeric" "character" "character"   "numeric"
 latitude   longitude      header      subset
"numeric"   "numeric" "character" "character"
```

Note that the content of each of the <*item*> nodes within the <*subset*> element are lines of comma-separated values. These look like different pieces of a CSV document. We can turn them into a data frame using read.csv() and a text connection in the following manner:

```
vals = xmlSApply(subsetNode, xmlValue)
con = textConnection(vals)
data = read.csv(con)
close(con)
```

We put this all together as a function that we can pass as the value for the *.convert* argument.

```
modisConvert =
function(node)
{
  ans = XMLSchema::fromXML(node)
  vals = xmlSApply(node[["subset"]], xmlValue)
  con = textConnection(vals)
  on.exit(close(con))
  ans$subset = read.csv(con)
  ans
}
```

We can then use this with

```
val = .SOAP(MODIS_URL, "getsubset",
            40.115, -110.025, "MOD11A2", "all",
            "A2001001", "A2001025", 1, 1,
            action = MODIS_Action, .convert = modisConvert)
```

12.5.7 Additional Arguments

The .SOAP() function has hooks for user-defined customization for several additional aspects of the *SOAP* invocation. We can specify the namespace definitions to allow switching from *SOAP* version 1.2 to *SOAP* version 1.1 via the *nameSpaces* parameter. The simplest way to specify this is by calling SOAPNameSpaces(). We can get the 1.1 namespaces with the call

```
SOAPNameSpaces(version = "1.1")
```

and then pass this as the value for *nameSpaces*.

We can also specify information to pass in the header of the *HTTP* request via the *.header* parameter. This value is used as the value of the *httpheader* option for the `CURLHandle`. This enables us to specify *HTTP* header fields such as cookies, acceptable response types, a user agent string, and so on. It should include an appropriate value for the *SOAPAction* field as if you override the default value, you are responsible for setting this. Note that we can also specify the *HTTP* header by populating a `CURLHandle` with curl options and then passing this as the value for either the *server* or *curlHandle* parameters.

12.6 Handling Errors in *SOAP* Calls

In this section, we look at how we can handle errors in *SOAP* requests in different ways. There are three general types of errors that can occur in a call to the .SOAP() function.

1. Firstly, we can get a regular *R* error because of syntax errors, nonexistent variables specified in the call, and so on. These occur before we even get to making the *SOAP* request to the Web server.

2. The second type of error is a problem in the *HTTP* request. The .SOAP() function checks the *HTTP*'s response header for an indication of a communication/*HTTP* error, e.g., the status that indicates that the host is unavailable, or the *URL* is incorrect or nonexistent. If there is an error at that point, .SOAP() raises that error with the appropriate class related to the communication problem, e.g., with class **COULDNT_RESOLVE_HOST** . Each of these will inherit from the class `GenericCurlError`.
3. The third type of error arises from the actual *SOAP* method request and comes from the Web service itself, not the *HTTP* transportation layer. We might have provided too few arguments, arguments of the wrong type that cannot be understood, or values that are not valid, e.g., a date that is out of the range the Web service can process. In this case, .SOAP() raises an error of class `SOAPError` that contains information about the error as explained by the Web service.

One of the reasons for enumerating how the errors can occur and the classes of the resulting errors is so that programmers can catch specific types of errors using the tryCatch() mechanism in *R*. For example, we can react to different errors using

```
tryCatch(.SOAP(....),
         COULDNT_RESOLVE_HOSE = function(e) {
           cat("The host doesn't appear to exist: ",
               e$message, "\n")
         },
         GenericCurlError = function(e) {
            cat("Problem in the HTTP communication",
                " for the SOAP request:",
                e$message, "\n")
            NULL
         },
         SOAPError =  function(e) {
             cat("Problem in the actual SOAP call:",
                 e$message, "\n")
         })
```

The different classes of errors can be used to implement different error handling strategies, including recovering and continuing to complete the request. See the condition system in *R* [7].

12.7 Using the <Header> Element in a *SOAP* Request for Authentication and Security

In this section, we briefly discuss the case where access to *SOAP* methods is restricted to particular users, i.e., access requires some authentication and/or authorization to use the services. There are several approaches to dealing with this. One approach is to rely on some form of *HTTP* authentication. This might be passing a login and password as part of the *URL*, or specifying it in the *HTTP* request via the curl option *userpwd*. If we are using *HTTP* as the *SOAP* communication mechanism (rather than some other protocol such as SMTP), then this will work correctly as we can specify the authentication information via the *.opts* parameter in our calls (e.g., via the *userpwd* option). We should note, generally, that in order to keep the authentication information (login and password or token) secret, we should use secure *HTTP*, i.e., *HTTPS*, when communicating with the server to ensure that the content is not sent as readable text by an intermediary. However, *SOAP* can use other transport layers such

as email via SMTP or FTP where the *HTTP* authentication is not available. This is why people have developed *SOAP*-specific approaches for security. In the ChemSpider example earlier in the chapter (Section 12.2.2), we saw that one can register with the service to get a token and then send this as an argument in some of our calls to the Web service methods. The method then verifies that the token is valid and corresponds to a user who has permission to call the method and then proceeds. This process delegates the verification to the method. A second scenario involves passing the authentication and/or authorization information separately from the methods' arguments, but still as part of the *SOAP* request. For this, we put the information in the `<Header>` node of the *SOAP* `<Envelope>`.

The *SOAP* specification provides great flexibility in what one can put in the `<Header>` node and how authentication is done using the `<Header>`. There are many ways that a server can validate the call. It may require a simple login and password. It may require more elaborate details such as including a "nonce" (a number that is used just once across all requests and so inhibits others from intercepting and running exactly the same request at a later time) and a time stamp as *OAuth* 1.0 does (see Chapter 13). It may also require a three-stage negotiation in which the caller first sends a partially complete request, the server responds with information that should be put in the `<Header>` for all subsequent requests that are considered complete, and the client takes this information and uses it in these subsequent requests. There are yet many other approaches, so we can see that an *R* programmer may have to customize how the authentication information is provided for a particular *SOAP* server. However, there is a commonality to these and that is that the information is passed as *XML* content within the `<Header>` node of the *SOAP* document. We look at some approaches to using the `<Header>` for authorization and how the .SOAP() supports these and other approaches. Before we do so, we note that there are also very different general approaches to security in *XML*. These include *XML* document signatures, for example.

As we have mentioned previously, a *SOAP* message consists of an `<Envelope>` node and this element can have an optional `<Header>` node. We can use this to provide additional information about the request, including authentication details. For example, a particular server may expect a login and password to be specified in the `<Header>` in the form

```
<SoapAuthHeader>
 <UserName>duncan</UserName>
 <Password>simple password</Password>
</SoapAuthHeader>
```

(We would, of course, replace `"duncan"` and `"simple password"` with the actual login and password for the caller.) Alternatively, the *SOAP* server may expect this information in a different form such as using Web Service (WS)-Security, e.g.,

```
<ws:Security soapenv:mustUnderstand="1">
 <ws:UsernameToken  wsu:Id="UsernameToken-1">
  <ws:Username>duncan</ws:Username>
  <ws:Password Type="....#PasswordText">
    simple password</ws:Password>
 </ws:UsernameToken>
</ws:Security>
```

There may be other formats we need to use to provide this information. For our purposes, the key thing is that this information must be in *XML* content that we need to send in the *SOAP* request. We need to add it to the `<Header>` node of the request's `<Envelope>` element. So we need a mechanism in the .SOAP() function to do this.

First, we create the *XML* content. Here, we do this for the first of our simple examples above with the *R* command

```
auth = newXMLNode("SoapAuthHeader",
                  newXMLNode("UserName", "duncan"),
                  newXMLNode("Password", "simple password"))
```

We can then instruct the .SOAP() function to add this to the *XML* document being sent by passing this <SoapAuthHeader> node via the *.soapHeader*[1] parameter. For example, we can do this with the call

```
.SOAP("http://www.chemspider.com/MassSpecAPI.asmx",
      "GetDatabases", .soapHeader = auth, action = I(uCSGD))
```

(Note that the variable uCSGD contains the *URL* http://www.chemspider.com/MassSpecAPI.asmx#GetDatabases.) Similarly, we can use the function generated from the *WSDL* document for the Web service and specify a value for *.soapHeader* as

```
dbs = massSpec@functions$GetDatabases(.soapHeader = auth)
```

Both of these produce the (abbreviated) *XML* document

```
<SOAP-ENV:Envelope ...>
 <SOAP-ENV:Header>
  <SoapAuthHeader>
   <UserName>duncan</UserName>
   <Password>simple password</Password>
  </SoapAuthHeader>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
  <namesp1:GetDatabases
    xmlns:namesp1="http://www.chemspider.com/GetDatabases"/>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is sufficient to create the correct *SOAP* request.

The *.soapHeader* parameter is quite flexible. It can accept a value that is an *XML* node, or a string that it then parses as *XML* content and converts to a node. The value can also be an *R* function that will generate *XML* content (as a node or string). The function is called with two arguments—the *XML* document being created with the <Envelope> as its root node, and the name of the method being called. This allows the content for the <Header> to be constructed dynamically. It also allows the header to examine the <Body> of the document to query information about the *SOAP* invocation that it may need, e.g., to digitally sign the contents. We also note that this mechanism provides a "back-door" for modifying the *XML* content sent as part of the .SOAP() function. In other words, we can provide a function as the value for *.soapHeader* that modifies the <Body> node, but that may not add a <Header> node.

Before we conclude the discussion of authentication and the *SOAP* <Header> node, we should mention that the *XML* in the *SOAP* response for our method invocation can also contain a <Header> node. The server can return information in this <Header> node that we may want to process. For instance, it may return a session token that we should use in subsequent calls. We would extract

[1] Note that the *.header* parameter is for specifying fields for the *HTTP* request's header such as *Accept*, *SOAPAction*, *User-Agent*, not the header of the *SOAP* <Envelope>.

this from the response's <Header> node and add this to future requests via their <Header> node. The .SOAP() function does not currently process a <Header> node in the response. Instead, we, the caller, must do this ourselves. In order to access the <Header> node, we need to disable the conversion of the response to an *R* object by .SOAP() via `.convert = FALSE`. Then the .SOAP() function returns both the *HTTP* header and the *XML* document as a string. We would then parse the <Header> node in the *XML* document and extract that information.

12.8 Customizing the Code Generation

Ideally, an *R* user will be able to call genSOAPClientInterface() with just the *URL* of the *WSDL* document and that this will create all of the functions, class definitions, and converters between *R* and *XML* without any extra information. The *R* user will then be able to invoke any of the generated functions and be completely isolated from the details of *SOAP* underlying the functions. Unfortunately, this may not be possible for two reasons. One reason is that genSOAPClientInterface() and, more particularly, the code in the XMLSchema package is not necessarily able to deal with all possible *WSDL* documents and schema types. In this case, you can report the problem to the maintainer of the SSOAP package and we can generalize the code to handle these new characteristics. This is a win for the user and all the other potential users of the software, so we strongly encourage this approach.

The second problem may arise because the *SOAP* methods and associated data types are different from what the genSOAPClientInterface() expects, *by default*. In this case, we can use some of the additional functionality in genSOAPClientInterface(), processWSDL() and other functions that read and process the *WSDL* and schema. These can be used to both post-process the *WSDL* and schema before we generate the code, and also to customize how the code is generated. We focus on various aspects of this customization in this section.

When customizing the code generation, it is often best to first read the *WSDL* document with processWSDL(), make any modifications to the descriptions it returns, and then pass the modified descriptions to genSOAPClientInterface(), or your own functions for generating the code. Throughout this section, we will use this two- or three-step approach, with the modification of the descriptions sometimes being omitted. The processWSDL() function returns an object of class `SOAPServerDescription`; genSOAPClientInterface() returns an object of class `SOAPClientInterface`. These are quite similar in structure in that they both have collections describing the *SOAP* functions/methods and the data types from the schema. The `SOAPServerDescription` is just a description of these elements, while the `SOAPClientInterface` has converted those into *R* objects and definitions. The earlier description has not assembled and resolved the references to the data types, but merely collected the information so that we can use it when it is needed.

12.8.1 Specifying the Port and Bindings

One of the first customizations we can make is to specify the "port" in the *WSDL* document with the methods we want. Basically, a *WSDL* method can contain several "ports" and in principle these can offer different methods. However, often we have different ports that offer different mechanisms to access the same methods. These different mechanisms may be different versions of *SOAP*, i.e., 1.1

or 1.2, or the ports may offer an *HTTP* **POST** and an *HTTP* **GET** approach for how to deliver the request. For example, the *WSDL* for ChemSpider contains the following *XML*:

```
<wsdl:port name="SearchSoap" binding="tns:SearchSoap">
 <soap:address location="http://www.chemspider.com/Search.asmx"/>
</wsdl:port>
<wsdl:port name="SearchSoap12" binding="tns:SearchSoap12">
 <soap12:address
     location="http://www.chemspider.com/Search.asmx" />
</wsdl:port>
<wsdl:port name="SearchHttpGet" binding="tns:SearchHttpGet">
 <http:address location="http://www.chemspider.com/Search.asmx"/>
</wsdl:port>
<wsdl:port name="SearchHttpPost" binding="tns:SearchHttpPost">
 <http:address location="http://www.chemspider.com/Search.asmx"/>
</wsdl:port>
```

This *WSDL* lists four different `<port>` elements, each with is own *name* attribute and each referencing a different *binding*, e.g., `"tns:SearchSoap12"`. Elsewhere in the *WSDL*, these bindings are defined and they list the different methods provided by that binding, e.g.,

```
<wsdl:binding name="SearchSoap12" type="tns:SearchSoap">
 <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="SimpleSearch2IdList">
       ...
 <wsdl:operation/>
   ...
</wsdl:binding>
```

This means we can specify which port, and accordingly the binding and set of methods we want for our interface, via the processWSDL()'s *port* parameter. We can identify the port by position/index or, preferably, by its *name* attribute. For example, either of the following

```
w = processWSDL(url, port = "SearchSoap12")
w = processWSDL(url, port = 2L)
```

will use the *SOAP* version 1.2 binding and its methods.

12.8.2 Processing Only Relevant Functions

Another common simple customization is in our call to genSOAPClientInterface(). We can focus on a subset of the methods rather than generating wrappers for all of them. If the *WSDL* specifies many methods in which we have no interest, we can save time and memory by discarding those. We can do this by subsetting the list of functions in the `SOAPServerDescription` and either overwriting this collection with the smaller set, or explicitly passing the sublist of operations to genSOAPClientInterface(). For example, let's read the mass-spectrometry API from ChemSpider and examine the operations it provides:

```
u = "http://www.chemspider.com/MassSpecAPI.asmx?WSDL"
w = processWSDL(u)
head(names(w@operations[[1]]))
```

```
[1] "GetDatabases"     "SearchByMass"     "SearchByMass2"
[4] "SearchByFormula"  "SearchByFormula2" "GetRecordMol"
```

Note that processWSDL() returns an object that may have multiple sets of operations corresponding to different bindings and interfaces. For this reason, we use `w@operations[[1]]` to access the methods for the first set and in this case there is only one set of methods. Suppose we only want *R* functions for the GetDatabases(), SearchByMass2() and GetExtendedCompoundInfoArray() methods. We can specify this with

```
funNames = c("GetDatabases", "SearchByMass2",
             "GetExtendedCompoundInfoArray")
w@operations[[1]] = w@operations[[1]] [ funNames ]
iface = genSOAPClientInterface(w)
```

or directly as

```
iface2 = genSOAPClientInterface(w@operations[[1]][funNames], w)
```

Clearly, the first approach discards the other functions permanently, while the second approach keeps them around for later use, but only processes the three we want.

12.8.3 Changing and Adding Formal Parameters

There are cases where we want to change the functions that are created by genSOAPClientInterface(). For example, we may want to change the default server. We may want to change the default value for the *curlHandle* parameter so that it picks up an existing curl handle that contains some authentication information. We may also want to add additional parameters to a function and have it passed on to the .SOAP() call or add some additional code within the body of our function that makes use of it. We look at these examples in order to illustrate and discuss more general changes one can make.

12.8.3.1 Changing the Default Server

Suppose we want to change the default server to which the *SOAP* requests are sent. For instance, there may be a local mirror that provides faster responses or we may want to test an experimental server. Alternatively, we may simply want to switch to *HTTPS* rather than use *HTTP*. There are three ways we can do this. The essential differences between the approaches are: changing the server before code generation, changing all functions by changing a shared variable, and changing the default value of a parameter in one or more functions. Each of these has its advantages given a particular goal.

The first approach is to change the server information before we call genSOAPClientInterface(). We can read the *WSDL* document via

```
wsdl = processWSDL("http://soap.genome.jp/KEGG.wsdl")
```

The *server* slot in the `wsdl` object has class `HTTPSOAPServer`. Let's change this to use *HTTPS* and switch the host and port to `"localhost"` and 8080, respectively, but leave the *path* slot unchanged. We do this with

```
wsdl@server = new("HTTPSSOAPServer", host = "localhost",
                  port = 8080L, path = wsdl@server@path)
```

or

```
wsdl@server = as(wsdl@server, "HTTPSSOAPServer")
wsdl@server@host = "localhost"
wsdl@server@port = 8080L
```

This change will be used when we call genSOAPClientInterface() –

```
genSOAPClientInterface(wsdl)
```

The new functions will use this server information.

This approach hopefully illustrates that we can change the descriptions of the methods and the data types returned by processWSDL() before we generate the code. We can add new parameters to some, or all, functions, or change some of the classes and types and then generate the code.

A second approach to changing the default server used by the new functions is to change the shared variable they all use. As we mentioned in Section 12.2.3, each of the functions generated by genSOAPClientInterface() has its own environment. This is used to store the variable `.operation` that provides a description of the Web service method at "run-time." This environment has a parent environment shared by all the generated functions. This is where the `.defaultServer` variable is located, and this is used as the default value of the *server* parameter for each of the functions. This means that we can change the value of that variable, and all the functions will use the new variable. We can do this with

```
e = environment(parent.env(list_databases))
e$.defaultServer = new("HTTPSSOAPServer", host = "localhost",
                       port = 8080L, path = wsdl@server@path)
```

This is different from the previous approach above as we are changing the behavior of the functions after they are created.

A third approach to changing the default server is to change this in each function directly. The formal arguments or signature for the list_databases() function, for example, are shown as

```
function (server = .defaultServer,
          .convert = .operation@returnValue,
          .opts = list(), nameSpaces = "1.2", .soapHeader = NULL,
          .header = SSOAP::getSOAPRequestHeader(.operation@action,
                                                .server = server),
          curlHandle = getCurlHandle()) {}
```

The default value for the *server* parameter is `.defaultServer`. We can change this by directly assigning a new value with the command

```
formals(list_databases)$server =
      new("HTTPSSOAPServer", host = "localhost", port = 8080L,
           path = wsdl@server@path)
```

This new `HTTPSSOAPServer` object has become the default value for that argument.

One of the advantages of this last approach of changing the default value of one of the function's formal parameters is that we can do this to a subset of the functions, should we so want.

12.8.3.2 Changing the Default Value of Service-level Parameters in All Functions

Suppose we want to change the default value of the *curlHandle* parameter in our generated functions. We can combine two of the approaches from Section 12.8.3.1 to do this. We can 1) change the default

value in each function to refer to a variable, and 2) place that variable in shared environment. We first create that shared variable. We call the variable `.defaultCurlHandle` and assign the value to the shared environment. We can do this in several steps. We first get the parent environment of one of the functions with

```
e = parent.env(environment(iface@functions[[1]]))
```

This is the shared environment. Then we assign `.defaultCurlHandle` to that environment:

```
e$.defaultCurlHandle =
     getCurlHandle(verbose = TRUE, userpwd = "duncan:myPassword",
                   maxredirs = 4L, followlocation = TRUE)
```

The next step is to change the functions to refer to this as the default value of the *curlHandle* parameter. We can do this for one function with the command

```
formals(iface@functions[[1]])$curlHandle =
                                    quote(.defaultCurlHandle)
```

The result is that this function will look for the variable `.defaultCurlHandle` and find it in the environments it inherits. Of course, we can loop over the functions to do this for each of them. Also, it should be clear that we can change the default value for the *.soapHeader* or any of the other parameters to specify our own default.

12.8.3.3 Adding a Parameter to a Function

We can also add a parameter to a function. This is unusual and we can often use other approaches such as writing a wrapper function that calls our newly generated function or taking over how the functions are generated altogether (see Section 12.8.3.4). However, let's explore this idea as it also shows how we can modify function objects directly in *R*. Suppose we want to change the way we convert the result of the *SOAP* request for a particular method. For simplicity, we use the list_databases method in the KEGG Web service. Recall this takes no arguments and the generated function is

```
function(server = .defaultServer,
         .convert = .operation@returnValue,
         .opts = list(), nameSpaces = "1.2",
         .soapHeader = NULL,
         .header = SSOAP::getSOAPRequestHeader(.operation@action,
                                               .server = server),
         curlHandle = RCurl::getCurlHandle())
{
  .SOAP(server, .operation@name, action = .operation@action,
        xmlns = .operation@namespace,
        .types = .operation@parameters, .convert = .convert,
        .opts = .opts, nameSpaces = nameSpaces,
        .elementFormQualified = TRUE, .returnNodeName = "return",
        .soapHeader = .soapHeader, .header = .header,
        curlHandle = curlHandle)
}
```

We want to override the conversion of the result and we want to allow the caller to control how we do this with a parameter, say, *simplify*. So we need to add the *simplify* parameter to the function, give it a default value (`TRUE`), and then use it when converting the result. We want the code to look like

```
function(simplify = TRUE, server = .defaultServer,
         .opts = list(), nameSpaces = "1.2",
         .soapHeader = NULL,
         .header = SSOAP::getSOAPRequestHeader(.operation@action,
                                               .server = server),
         curlHandle = RCurl::getCurlHandle())
{
  ans <-
    .SOAP(server, .operation@name, action = .operation@action,
          xmlns = .operation@namespace,
          .types = .operation@parameters, .convert = FALSE,
          .opts = .opts, nameSpaces = nameSpaces,
          .elementFormQualified = TRUE, .returnNodeName = "return",
          .soapHeader = .soapHeader, .header = .header,
          curlHandle = curlHandle)
  converterFunction(ans, simplify)
}
```

In this version, we've changed the call to .SOAP() to specify `FALSE` as the value for *.convert*. We have also assigned the value from the .SOAP() call to `ans` and then added a new call to converterFunction() with both the result and *simplify* as the two arguments.

How do we modify the original function to create the one we want? There are several steps. We drop the *.convert* parameter and add *simplify* as the first parameter. It is easiest to do this by changing the *.convert* to *simplify*, replacing its default value, and changing the order of the parameters, e.g.,

```
i = match(".convert", names(formals(list_databases)))
formals(list_databases)[[i]] = TRUE
names(formals(list_databases))[i] = "simplify"
formals(list_databases)[1:2] = formals(list_databases)[2:1]
```

The next part of our changes is to modify the call to .SOAP() and assign the result. We can do the first step with the commands

```
soapCall = body(list_databases)[[2]]
soapCall[[".convert"]] = FALSE
```

We assign this to the variable `ans` and replace the original call to .SOAP() in the new function with

```
body(list_databases)[[2]] = call("<-", as.name("ans"), soapCall)
```

The final step is to add a call to converterFunction() and pass it both `ans` and `simplify`. We do this with

```
body(list_databases)[[3]] = quote(converterFunction(ans, simplify))
```

This completes our changes. Having modified a function, we can use it immediately or we can use writeInterface() to write it to a file for use in a different *R* session.

We have manipulated the function's parameters and the body of the function and its calls. We have changed language objects. This is called "programming on the language." It seems complex, but is

quite straightforward with a little understanding of the structure of *R* language objects. For the most part, a function, its parameters (formal arguments), and its body can be treated as *R* lists. This allows us to change particular elements of those lists to create different language objects.

Many people would prefer to work with the function definitions as strings and to insert the code using pattern matching and text substitution. This seems simple, but can be very ad hoc, fragile, and error-prone. It is hard to match a pattern precisely and unambiguously. This is especially true across multiple lines. There is no context to the pattern matching, e.g., find the *.convert* argument in the call to .SOAP() where the value is `FALSE` and change that, but ignore the parameter definition in the function named *.convert*. Think about how to write a regular expression that captures these semantics based solely on text. Manipulating the language objects as we have done above is one of the less commonly known strengths of the *R* language and is far more flexible, general, and extensible. It is our preferred way to generate *R* code in *R*.

12.8.3.4 Changing How the Functions Are Generated

If you need to significantly change the way the functions are created, you can specify a different function in the call to genSOAPClientInterface() for generating the code for a function via the *opFun* parameter. For example, you may need to use an *OAuth* approach (see Chapter 13) to make the request and so call another function instead of .SOAP(). In many cases, you can adapt the functions that genSOAPClientInterface() generates by default, e.g., replace the .SOAP symbol with another function. However, in some cases, it may be simpler to take over the creation of the functions directly. genSOAPClientInterface() allows us to do this via its *opFun* parameter. We pass it a function and that is called for each method in the *WSDL*, and it is expected to return a function object.

The function passed as the value of *opFun* is called for each *SOAP* method being processed, and it is responsible for generating the *R* function corresponding to that method. It is passed six arguments. These are:

- A description of the *SOAP* method as an object of class `WSDLMethod`. This contains all the information about the *SOAP* method, including its name, the list of data types for the inputs, the type of the return value, the *SOAP* action, the name of the node containing the value of the result, documentation, and the details of the *SOAP* mechanism to use for the input and output.
- An object of a subclass of `SOAPServer` that provides the details of the *URL* for the server to which the *SOAP* request should be sent
- A collection of all the data types defined in the different *XML* schema defined in the *WSDL*. This is an object of class `SchemaCollection` and is a list of elements that describe each schema. Each element is of class `SchemaTypes` and is a list describing the individual data types and *XML* elements definitions. These classes are described in Chapter 14
- An environment that is used to store global/nonlocal variables shared by all of the functions, e.g., the default server. This is typically used as the environment of the resulting function so that it can refer to these global variables.
- A string identifying which *SOAP* namespaces to use, i.e., version 1.1 or 1.2. This can be passed to .SOAP() directly or used to inline the namespaces.
- A logical value that indicates whether the function should provide a parameter for the caller to provide a *SOAP* header object.

As with other packages described in this book, there are numerous helper functions in the SSOAP package. You should explore the code in the package and use these rather than reinventing them. We encourage you to suggest changes to make them more flexible and useful.

12.9 Serializing *R* Values to *XML* for *SOAP*

The .SOAP() function is responsible for creating the *XML* document that represents the method invocation. This consists of an <*Envelope*> and a <*Body*> node within that. The <*Body*> has a node that identifies the name of the method. The children of this node are the serializations to *XML* of the *R* values passed as arguments to the Web service method. What controls this serialization? Essentially, it is the generic function toSOAP(). Since this is a generic function, we can override or define new methods in order to customize how some objects are serialized to *XML*.

The definition of a toSOAP() method should have the signature

```
function(obj, con = xmlOutputBuffer(header = ""),
         type = NULL, literal = FALSE,
         elementFormQualified = FALSE, ...)
```

The methods used for toSOAP() are selected based on the class of the object being serialized (*obj*), and the target type (*type*) that defines the expected data structure, i.e., an *R* object describing the type from the *WSDL*'s schema. These types of objects are derived from the `GenericSchemaType` class in the XMLSchema package (see Chapter 14). These describe *XML* element definitions, complex types, sequences, etc., in the *XML* schema.

Suppose we decided that objects of class `matrix` should always be represented for *SOAP* as an array of rows, and that each row is an array. For example, the matrix

```
  [  1, 2, 3
     4, 5, 6]
```

would appear in *SOAP* as

```
<matrix xsi:type="SOAP-ENC:Array"
  SOAP-ENC:arrayType="xsd:int[2,3]>
 <item xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="xsd:int[3]">
  <item xsi:type="xsd:int">1</item>
  <item xsi:type="xsd:int">2</item>
  <item xsi:type="xsd:int">3</item>
 </item>
 <item xsi:type="SOAP-ENC:Array"
   SOAP-ENC:arrayType="xsd:int[3]">
  <item xsi:type="xsd:int">4</item>
  <item xsi:type="xsd:int">5</item>
  <item xsi:type="xsd:int">6</item>
 </item>
</matrix>
```

We can define a method for this type with

```
setMethod("toSOAP", c("matrix", "XMLInternalElementNode"),
  function(obj, con = xmlOutputBuffer(header=""),
           type = NULL, literal = FALSE,
           elementFormQualified = FALSE, ...) {
                ...
  })
```

The body of the method needs to create the appropriate *XML* content. It does this by adding children to the *XML* node passed to it as the value of the *con* argument. This way, we can implement our function by first creating the outer <matrix> node and then looping over each row and creating an <item> element for that row and a separate <item> element for each of the values in that row. We need to specify the *type* and *arrayType* for top-level <item> elements and the *type* for each value in the matrix. We do this with the following code:

```
dims = sprintf("xsd:int[%d, %d]", nrow(obj), ncol(obj))
rowDims =  dims = sprintf("xsd:int[%d]", ncol(obj))

mnode = newXMLNode("matrix",
                   attrs = c("xsi:type" = "SOAP-ENC:Array",
                             "SOAP-ENC:arrayType" = dims),
                   parent = con)
apply(obj, 1,
      function(values) {
         row = newXMLNode("item",
                          attrs = c("xsi:type" = "SOAP-ENC:Array",
                              "SOAP-ENC:arrayType" = "xsd:int[]"),
                          parent = mnode)
         sapply(values, function(v)
                           newXMLNode("item", v,
                                      attrs = c("xsi:type"="int"),
                                      parent = row))
      })
```

Of course, this function should check the actual type of the values and replace int with the corresponding *XML* schema type. It can do this by using functions in the XMLSchema package.

The `xsi` and `SOAP-ENC` namespace prefixes we used in our function correspond to the `http://www.w3.org/2001/XMLSchema-instance` and `http://schemas.xmlsoap.org/soap/encoding/` namespace URI. We can define the namespaces ourselves, but they are already part of the larger *SOAP* document being created when our toSOAP() method is called. These namespaces are used to encode the type of the content in each *XML* node. Consult a book such as [9] for more details on *SOAP* encoding.

In our function, we had to manually create each node in the *XML* content we generated. We can, however, utilize some of the existing methods to do the work for us. For example, there are functions and methods that can create a *SOAP* array for each of the rows. In this case, we can call toSOAPArray(), e.g., for the first row

```
SSOAP:::toSOAPArray(obj[1, ], mnode)
```

We can use this approach to process each row of the matrix.

In other cases, we may call the generic function toSOAP() again for the subparts of our *R* object, rather than a particular function such as toSOAPArray(). Typically, we have to provide a description of the data type we are trying to create. We pass this via the *type* parameter. This involves using the classes in the XMLSchema package to describe a data type. For a row, this is an `ArrayType` with an integer as the element type, which we can describe with the class `SchemaIntType`. We can call toSOAP() with

```
toSOAP(1:3, newXMLNode("doc"),
       type = new("ArrayType", elType = new("SchemaIntType")))
```

where here the value for *type* describes our array of integer values.

We should note that our example is intentionally simple. In practice, we would not want a method for toSOAP() to create the same *XML* representation regardless of what is expected by the consumer of the *XML*. Instead, we would want the method to use the information in the *type* argument to determine what was expected. We might define one method for an `SimpleSequenceType` and another for a `Element` type, and so on. *S4* methods allow us to specify methods for combinations of *R* type and *XML* schema type, and to select the methods based on the classes of both arguments simultaneously.

While we can use either genSOAPClientInterface() or .SOAP() for most cases, more advanced or atypical *SOAP* calls may require more control over a) creating the *XML* document representing the request and/or b) how we send the request. The SSOAP package provides some additional functions at a lower-level than .SOAP(), which programmers can call to take care of the different pieces. For instance, if one wants to customize the creation of the *XML* document, then one might call writeSOAPMessage() to create the standard document and then modify the resulting *XML* tree.

12.10 Possible Enhancements and Extensions

Documentation
A *WSDL* document often contains documentation for the methods, their parameters and the data types defined in the schema. It would be valuable to be able to process this information and generate *R* help files for the generated functions and classes. One could use *R*'s existing Rd format or an *XML* format such as the one provided by the RXMLHelp package [10].

12.11 Summary of Functions for Working with *SOAP* in *R*

The SSOAP package provides both high-level and low-level functionality to access *SOAP*-based Web services from within *R*. We can pass genSOAPClientInterface() the *URL* of a *WSDL* document describing a *SOAP* server and its methods and data types, and the function will generate *R* functions and classes that allow us to immediately invoke those methods. We can write the generated *R* code to files for use in other *R* sessions or packages via writeInterface(). The function that actually makes the *SOAP* requests is .SOAP(), analogous to .C() or .Fortran(). We describe these functions in the SSOAP package below.

genSOAPClientInterface() Programmatically generate an interface to a *SOAP* Web service from a *WSDL* document. This function uses processWSDL() (see below) to obtain descriptions of the *SOAP* service, and then creates *R* class definitions and functions that mirror the Web service methods as local *R* functions. The *putFunction* parameter controls where the newly created functions are defined. By default, they are simply returned as a list. However, they can be assigned to the global environment or another environment identified by the *where* parameter. The genSOAPClientInterface() function allows us to control and customize how the functions are created.

processWSDL() Read a *WSDL* document and develop descriptions of the Web server, methods, and data. This function is used to understand the *WSDL* document describing a server and its

methods. The genSOAPClientInterface() function is then used to map this description to *R* functions and class definitions.

.SOAP() Make a *SOAP* request. This is the workhorse function of the SSOAP package that actually constructs the *SOAP* request, sends it to the server, and converts the result to an *R* object. There are many ways to customize how this function does each of these steps. The *server* parameter identifies the *URL* to which the request is sent, and the character string provided in the *method* parameter identifies the *SOAP* method being called. Arguments for the *SOAP* methods are passed via ... or the *.soapArgs* parameter. We can pass an existing `CURLHandle` to reuse an open, or perhaps authenticated, connection with the server. We can control whether, or how, the result is converted to *R* using *.convert*, specifying a logical value, the name of an *R* class, or a function.

writeInterface() Write the object returned from genSOAPClientInterface() to a file. This allows us to create the interface code just once and reuse it in a different *R* session, perhaps including it in an *R* package.

12.12 Further Reading

Both [9] and [2] are excellent resources on *SOAP*.

References

[1] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP), version 1.1. Worldwide Web Consortium, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[2] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[3] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weeraearana. Web Service Description Language (WSDL) 1.1. Worldwide Web Consortium, 2001. http://www.w3.org/TR/wsdl.

[4] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework*. Worldwide Web Consortium, 2007.

[5] Minoru Kanehisa. KEGG: Kyoto Encyclopedia of Genes and Genomes. http://www.genome.jp/kegg/, 2012.

[6] National Aeronautics and Space Administration. Moderate resolution imaging spectroradiometer: MODIS Website. http://modis.gsfc.nasa.gov/, 2012.

[7] *R* Core Team. *Condition Handling and Recovery*, 2012. http://stat.ethz.ch/R-manual/R-patched/library/base/html/conditions.html.

[8] Royal Society of Chemistry. ChemSpider: The free chemical database. http://www.chemspider.com/, 2012.

[9] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, Inc., Sebastopol, CA, 2001.

[10] Duncan Temple Lang. RXMLHelp: *XML* format and tools for *R* documentation. http://www.omegahat.org/RXMLHelp, 2011. *R* package version 0.1-0.

[11] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.
[12] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.
[13] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.
[14] Duncan Temple Lang. XMLRPC: Remote procedure call (RPC) via *XML* in *R*. http://www.omegahat.org/XMLRPC, 2012. *R* package version 0.2-5.
[15] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.
[16] Wikipedia contributors. Mass spectrometry. http://en.wikipedia.org/wiki/Mass_spectrometry, 2012.

Chapter 13
Authentication for Web Services via *OAuth*

Abstract Some *REST* APIs require authentication to access data and services, and many are starting to require use of a more general mechanism named *OAuth*. This avoids logins and passwords and allows secure three-party interactions between the owner of the data, the application accessing it, and the host of the data and Web service. In this chapter, we describe the *OAuth* mechanism and how to use the `ROAuth` package in *R* to work with *REST* APIs requiring authentication with *OAuth* 1.0. We also describe how to work with *OAuth* 2.0 in *R*. We illustrate these authentication mechanisms using Dropbox and Google Storage as examples.

13.1 Introduction: Securely Accessing Private Data with *OAuth*

When we use the Web, we are using a client–server model where our browser or *R* (or another application) is the client, and the Web server is the *provider* of the resource, e.g., an *HTML* document or a CGI script. In this case, the client is anonymous. The server knows the IP address of the client, but not the identity of the person making the request. For public sites and pages, this is fine as the server does not need to know who is requesting the page/*resource* in order to return the requested information. Of course, when the service contains data that is restricted to a specific person or group of people, the client cannot access it anonymously. Instead, the client must identify the user correctly, providing the appropriate credentials. For this, we typically use a login and password. We can specify these in an *HTTP* request using RCurl [11] via the *userpwd* curl option of the form `"login:password"`, or the *username* and *password* pair of options if we want to separate the login and password. (See Chapter 8 for details on how to perform *HTTP* requests using curl.)

This two-way or *two-legged* authentication is quite simple and familiar. Increasingly, however, we encounter more complex and richer situations that involve three parties in the interaction. Suppose we have a user who wants to allow an application on his or her desktop or a Web site to access data/resources on a different Web site. The *OAuth* documentation [7] describes the following example. Suppose we have photos on a Web site such as Flickr. We want to use an online printing service—PhotoPrint—to order physical copies of the photos. While we can download the photos from Flickr and then upload them to the printing site, PhotoPrint provides a neat feature that presents us with a list of our photos and allows us to select which ones we want to print. How does the photo service get a list of our photos? This is private information and should require us to log in to Flickr. In this case, we have three parties in the interaction. Flickr is the *provider* of the resources—the photos. We are the *user* who owns the resources (but we are not the provider). PhotoPrint is the *application* that

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_13,

needs access to these resources from the provider and must be given permission by us—the user and owner—to access them.

Under no circumstances do we want to give the photo service our login and password for Flickr. We do not want to allow them to delete any photos, change our account in any way, etc. However, we do want to authorize them to get a list of the photos, and also to be able to access the photos we select. That is, we want to give them limited privileges for a short period of time. We also want to be able to revoke these privileges or have them naturally expire. Instead of giving PhotoPrint our login and password, the application, user/owner, and provider communicate and give the application a token, or a ticket, that grants them explicit and specific privileges. The application then uses this token in each request it makes to the provider's API so that it is allowed access to the user's resources. The *OAuth* mechanism allows us to do all this. The *OAuth* guide on hueniverse.com describes *OAuth* in the following way: "*OAuth* provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.)."

OAuth is much more general and powerful than the simple login-password used for client–server authentication. *OAuth* overcomes many of the insecurities and limitations in basic authentication that use a login and password. However, it is more complex. Not only do we have to coordinate all three parties involved to grant and gain the relevant permissions, the *OAuth* 1.0 mechanism requires that we sign each request to access the privileged data. This signature mechanism is quite complicated. The `ROAuth` package [3] hides these details. *OAuth2*, the next generation version of the authentication mechanism, is a great deal simpler and more direct. It uses secure *HTTP* (*HTTPS*) to remove the complex signature mechanism. While *OAuth2* is not a finalized specification, it is in use by significant providers such as Google, Facebook (for their Graph API), and LinkedIn, and will be used increasingly in the future.

Our focus on *OAuth* in this book is using it from within *R*. Specifically, we are looking at invoking methods offered by a provider's API so that we can access private resources that belong to a user from within *R*. We use *OAuth* to negotiate between *R*, the user, and the provider to gain an access token. We then use *OAuth* and this access token to invoke the provider's methods via an *HTTP* request. We use the `ROAuth` package, rather than RCurl directly, to make these *HTTP* requests in order to hide all of the extra details *OAuth* requires.

In this chapter, we start by looking at *OAuth* 1.0 and the `ROAuth` package. We use Dropbox's API as a case study, and show how to download and upload files from and to our Dropbox account from within *R*. We illustrate *OAuth* 2.0 by accessing the Google Storage API. We should note that in both cases, there are packages that provide high-level *R* functions that interface with the methods of these two APIs—`rDrop` [10] and RGoogleStorage [12]. The functions in these packages hide much of the details of how we use *OAuth* in *R*. However, we illustrate some of these in this chapter. In this chapter, we do not focus on the low-level details, but instead look at how to use *OAuth* in *R* via the `ROAuth` package. You can read a great deal more about the lower-level details of *OAuth* in the *OAuth* specification [8], on the Web [7], and in the book [9]

13.1.1 The OAuth Model and R

Before we discuss the `ROAuth` package and specific *R* functions, it is important to understand how we think about the *OAuth* model in the context of using it from *R* to call methods in a provider's API. The general *OAuth* mechanism is for situations where there are three participants. There is the resource *owner/user* (i.e., us) who needs to authorize an *application* to have access to the resources

maintained by a *provider/server*. The user, application, and provider are the three parties. When we are working within *R*, the provider is the remote Web service with the methods we want to invoke. In most cases, the owner will be the *R* user and the application will also be a combination of *R* and the owner. In other words, the user/owner will have his or her own application in *R*. While there are three parties, two of them (the user and application) are very tightly coupled. As a result, the regular three-participant *OAuth* approach is a lot more complicated than it need be for this situation. Indeed, a login and password would be much more convenient for us. However, since *OAuth* is much more flexible and also widely used in other situations, we often need to use it. Accordingly, the common approach to using *OAuth* in *R* is that owners will create their own *R*-specific applications. (We will see how to do this for Dropbox below.) To connect all of this to *OAuth*'s three-legged model you can think of yourself (the user) as being the resource owner, *R* as being the application, and the provider, e.g., Dropbox, as being the server, or host, of the resources.

When a user creates an application, he/she registers it with the provider (typically via a Web page). The user specifies a name and a description of the application. (Of course, the application is much more than a name and description.) When the user registers the application, the provider gives two tokens—a consumer key and secret. These uniquely identify the application to the provider.

Why should owners have to create their own application? Instead, we might be tempted to use *OAuth* in the more common three-legged manner. In the case of Dropbox, for example, we can create an *R* package (say `rDrop`) that provides an interface with the Dropbox API and that acts as a separate third-party application. The developer of the `rDrop` package would register it with Dropbox. The package would have its own consumer key and secret. We can put these in the code for the package and then use them to obtain the access token by having each user grant permission for `rDrop` to access that user's own data. This appears to make a lot more sense than having each user create a consumer key and secret for the same *R* package. However, it is a *very* bad idea. If we did have a key and secret for the `rDrop` package, then these credentials must be in the *R* code for the package so that it can use them to perform the handshake(). This means that other programmers can easily obtain the application's private key and secret and use them in their own nefarious packages. When such a masquerading application asks a user to grant permission for their code pretending to be `rDrop`, it can use the key and secret in other code outside of the `rDrop` package. Users would unknowingly grant access to code that they thought was from the `rDrop` package, and the other package would then be able to do what it wanted with the access privileges, and the user would blame `rDrop`.

Similarly, keeping the key and secret for the application in the package but not in the *R* code still makes them accessible. For example, one might also consider putting the consumer key and secret in compiled code. Again, this is a problem if you give the source code for the package to others to install as an *R* package. Even with binaries, people can still use a program such as *strings* to extract those values. Furthermore, if the *R* code in the package can query the key and secret from the native code and then use them within *R* code, then an *R* user can step through *R* code and discover these private data directly. To avoid this, the compiled code would have to do all the signing of the *HTTP* requests directly itself. There are *C++* libraries to help with this (e.g., `liboauth` [2] and an *R* package that uses it), but this is less desirable, as it slightly complicates installation of packages, makes the code less flexible and accessible to *R* programmers, and still does not adequately protect the data.

We might also consider using an application-specific consumer key and secret, if we are running *R* code on a server where nobody can see that code. However, if we want the code to run on other systems not under our control, we will need to distribute the *R* package. Again, the consumer key and secret will no longer be secure. There is also no point in locating the key and secret at a *URL* as they are still accessible to others, no matter how obscure the *URL*s are.

Basically, anytime we have an open source package, it cannot restrict access to the keys. Instead, we must have each user provide the key and secret, and that is what we are doing when a users registers her own application and gets her own pair of key and secret.

13.1.2 Creating/Registering an Application with the Provider

Before we can access a provider's API in *R*, we need to register an application with the provider. This will give us the consumer key and secret that uniquely identifies the application to the provider. How we register the application differs slightly for each provider, but the basic steps are the same. We log in to the provider's Web page using our account for that site, find the relevant Web page for registering developer applications, and fill in a form. Assuming there are no errors, the provider will show you the consumer key and secret pair for the new application. You must store these and ensure that they are private, i.e., nobody else can read them.

In order to follow along with our example, you can register an application for Dropbox. We visit the page `https://www.dropbox.com/developers/apps` and click on the button in the middle of the page entitled "Create an App". Note that this page also lists the existing applications we have registered and we can find their information, including the key and secret, if we should lose them. We can also delete them and so revoke access for those applications. We specify a name for the application. Almost any name is fine. You may use the login name for the Dropbox account prefixed by the letter `"R"`. We use `"RJaneDoe"` for this chapter. You also provide a brief description. It is important to change the default `Access level` option to `Full Dropbox`. This is how we control the scope of the privileges. We click on the `Create` button and the Dropbox site shows us a new page with the key and secret for our new application. We will need to record those. It is important to keep these private.

13.2 The `ROAuth` Package

Rather than describing the *OAuth* mechanism and its low-level details, we start by focusing on how we use it in *R* via the `ROAuth` package. After we see it in practice, we give a high-level overview of two aspects of how *OAuth* works (negotiating the access token and signing the *HTTP* requests) in the hope that this may demystify the steps and help when things go awry.

13.2.1 The Basic Workflow in R for OAuth 1.0

There are three pieces to the computational model for *R* that you need to understand to make use of `ROAuth`. The first is that we use the oauth() function to create an *R* object that contains the application's consumer key and secret as well as information about how to communicate with the provider. The key and secret are for our "application," and are associated with our account on the provider, e.g., Dropbox. The second step is to use the consumer key and secret to get an access token for making actual *HTTP* requests to the provider's API. The handshake() function does the negotiating, involving the user to grant permissions via a Web page on the provider's site. The final piece is that each request needs to use the credentials object returned by handshake() to sign the

content of the request. This will happen automatically by using the OAuthRequest() function for each call to a method in the provider's API.

The basic sequence of operations in pseudo-code is

```
cred = oauth(key, secret, requestURL, authURL, accessURL)
cred = handshake(cred)
OAuthRequest(cred, methodURL, listOfArgs, method = "GET",
             curl_option, curl = curlHandle)
```

First, oauth() combines the application information for the *OAuth* provider. Then we use handshake() to negotiate between the application, provider, and user to get an access token. Lastly, we use this access token to invoke methods in the provider's API with a call to OAuthRequest().

Creating the Application `OAuthCredentials` Object

The oauth() function assembles the necessary details in order to obtain an access token from the provider. This function typically requires five arguments. The first two are the application's consumer key and secret that were created by the provider when we registered our application. The remaining three arguments are *URL*s that tell the oauth() function how to negotiate for an access token. These are the request *URL*, authorize *URL*, and access *URL*. We see later that *OAuth* uses these in sequence to obtain the final access token that has been verified by the user, i.e., owner of the resources we are going to access. For Dropbox, these *URL*s are listed and explained on the API reference page. Each *OAuth* provider will provide documentation that tells us its *URL*s for these tasks.

We can now call the oauth() function as

```
uRT = "https://api.dropbox.com/1/oauth/request_token"
uAu = "https://www.dropbox.com/1/oauth/authorize"
uAT = "https://api.dropbox.com/1/oauth/access_token/"
cred = oauth(consumerKey = cKey, consumerSecret = cSecret,
             requestURL = uRT, authURL = uAu, accessURL = uAT,
             post = FALSE)
```

We assume here that the consumer key and secret are stored in the variables `cKey` and `cSecret`. This is a good practice as they are secret and should never be visible in code. (See Section 13.2.3 for ideas about how to store them.)

The oauth() function creates an *R* object of class `OAuthCredentials`. It does not communicate with the provider (Dropbox) at this point.

The Handshake to Get the User Permission and Access Token

The negotiations to get the access token and secret necessary for making the actual requests on behalf of the owner of the data involve several steps. They also involve additional security including signing the contents of the request in such a way that the receiver can verify that they come from our registered application and are not intercepted and changed, or resubmitted (e.g., transferring money for a second time). This signature process also involves several steps and some less common technologies, such as digital signatures. The `ROAuth` package hides these complexities from you, and allows you to focus on invoking the Web service methods.

We start the negotiation with Dropbox to obtain authorization and the relevant request and access tokens by calling the handshake method:

```
msg = "hit return in R when you authorize access"
cred = handshake(cred, post = FALSE, verify = msg)
```

Note that we can also use the form

```
cred = cred$handshake(post = FALSE, verify = msg)
```

After we call this function, there is a slight delay and then we see our message we specified via the *verify* parameter. Then, our Web browser comes to the foreground and displays a page asking us to grant permission to the application we registered with the provider, Dropbox. If we are not already logged into our account on Dropbox in our Web browser, the browser will direct us to the login page and after entering our login and password, the actual page with which we grant permission to our application will appear. This looks something like the screenshot shown in Figure 13.1. We, the owner of the Dropbox account and the data of interest, then grant permission by clicking the `Allow` button and we see the page shown in Figure 13.2. For Dropbox, we then return to the *R* prompt and hit enter so that the handshake operation can proceed.

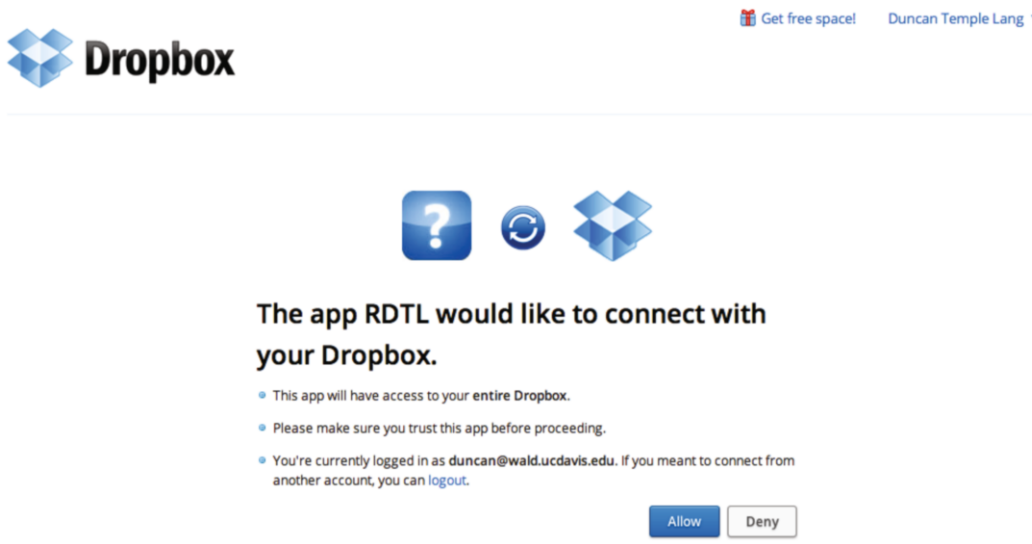


Figure 13.1: Example Dropbox Page Allowing the Application to Connect with the Owner's Resources. This screenshot shows the Web page that permits the user to grant permission to access data from his or her Dropbox account to the *R* application `RDTL`.

We should note that most *OAuth* providers will issue a new token after you grant permissions to the application. This will be shown on a Web page when you grant access. You then copy that from the browser back into *R* where the handshake() functions is waiting for the token. This is necessary as the oauth() function needs this token before it can proceed to the next stage to get the final access token. Dropbox is unusual in that it does not issue a new token. This is why we use the *verify* parameter to specify a message to prompt the user to hit the return key after they grant permission to the application. Generally, oauth() will prompt the user with something like (reformatted)

```
To enable the connection, please direct your web browser to:
https://www.dropbox.com/1/oauth/authorize?oauth_token=xxxxxxx
When complete, record the PIN given to you and provide it here,
or hit enter:
```

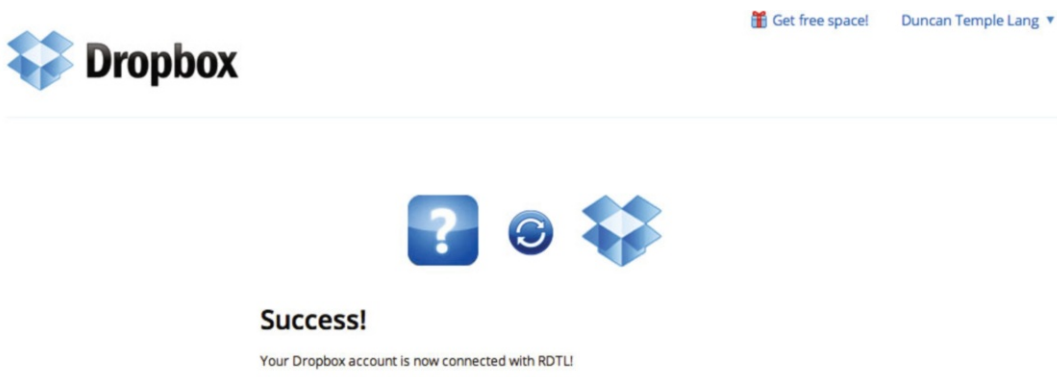


Figure 13.2: Example Dropbox Confirmation that the Application has Connected to the Owner's Resources. This is the Web page that Dropbox.com displays when the user has granted access to the *R* application RDTL.

This tells the user which *URL* to visit in case there is a need to visit the site manually, and also to copy the new token back to *R*.

It is important to note that the handshake() function returns the *updated* OAuthCredentials object with the additional key and secret that allows us to make requests. It is essential that you assign the object to an *R* variable to be able to use it in future calls. (We typically reassign it to the same variable name we used as the first argument to handshake(), thus simply updating that variable.) The OAuthCredentials object you created with the call to oauth() does not have this key and secret, just your own consumer key and secret. If you do not assign the updated version, it will be as if you never went through the handshake process.

The *post* argument in the call to handshake() controls how we perform the handshaking *HTTP* (or *HTTPS*) requests, using either **GET** or **POST** operations. **POST** is the default approach as many *OAuth* servers use **POST** and this is the recommended approach. We had to override it in the case of Dropbox.

As with all of the *OAuth* calls we are making, we are using an *HTTP* request and we can customize this with curl options. You can specify these in any *OAuth* call (handshake() or OAuthRequest()) as individual arguments or as a list of options via the *.opts* parameter. We can also pass a curl handle to use for the request via the *curl* parameter.

Invoking Methods with the *OAuth* Access Token

Once we have completed the handshake to acquire the authentication tokens, we can use the resulting credentials repeatedly to call privileged methods in the provider's API. We do this by calling the OAuthRequest() function. The first argument is the OAuthCredentials object (with the newly updated access key and secret). The second is the *URL* for the request. Next, we pass a list or character vector of the arguments for the call. We can also specify the type or *method* of the *HTTP* request, i.e., **GET**, **POST**, **PUT**, **DELETE**, etc. The default is **GET**. For example, the Dropbox API has a method for getting meta-information about the user's account. (See https://api.dropbox.com/developers/reference/api.) This method takes no inputs and is a regular **GET** request. We can invoke it with

```
u = "https://api.dropbox.com/1/account/info"
val = OAuthRequest(cred, u)
```

The result is a *JSON* string, which we can convert to an *R* object with

```
fromJSON(val)
```

Again, we can specify curl options or a curl handle to control the *HTTP* request. One can pass options to control how the *HTTP* request is performed via the . . . parameter of OAuthRequest(). For example, we can specify the user-agent for the request and also instruct the request to follow redirects with

```
val = OAuthRequest(cred, u, httpheader = c('User-agent' = 'rDrop'),
                   followlocation = TRUE, verbose = TRUE)
```

These arguments are passed on to oauthGET(), oauthPOST() and similarly named functions, selected based on the value of the *method* argument.

In most cases, we will pass actual arguments to parameterize the method. For example, if we want to create a new folder in Dropbox, we need to specify the root folder (either `dropbox` or `sandbox`) and the path or name of the new folder. This is a **POST** request rather than a **GET** operation. We can invoke this request as

```
uCF = "https://api.dropbox.com/1/fileops/create_folder"
val = OAuthRequest(cred, uCF,
                   c(root = "dropbox", path = "NewFolder"),
                   method = "POST",
                   httpheader = c('User-agent' = 'rDrop'),
                   followlocation = TRUE, verbose = TRUE)
```

Again we specify curl options, but the interesting argument to our function is the collection of individual arguments passed to the *REST* method. These are *root* and *path*. Here they are passed as a vector, but this can also be a list.

We have mentioned that *OAuth* 1.0 uses a regular *HTTP*/*HTTPS* request, but involves a somewhat complex process of signing the requests and adding information to the header in the *HTTP* request. The OAuthRequest() function does not require input from us to deal with this and uses the information in the `OAuthCredentials` object to take care of all of the details on our behalf. For the curious, we discuss the details later in this chapter.

There are some sites that do not require the user-authorization step in the *OAuth* negotiations. They use *OAuth* so that they can reliably identify the application making the requests, or simply when there is no specific user but general access. They may want to do this for auditing purposes or to provide different services for different applications, e.g., provide faster response for premium applications. In these cases, we can simply omit the authentication *URL* when creating the `OAuthCredentials` object, or directly in the call to handshake(). For example, the site `http://term.ie/oauth/example` provides a test site for *OAuth*1. It does not require user authentication as there is no user. It uses the literal strings `"key"` and `"secret"` as the consumer key and secret for all applications. We can obtain our access token using the call

```
uXR = "http://term.ie/oauth/example/request_token.php"
uXA = "http://term.ie/oauth/example/access_token.php"
cred = oauth("key", "secret", requestURL = uXR, accessURL = uXA)
```

We simply omit the *authURL* in the call. Then we can call handshake() and this performs the two-step request to gain the access token.

13.2.2 Using an Access Token Across R Sessions

At the end of a successful call to handshake(), the cred object will have the information to make authorized requests to access the protected resources. It would be nice to only have to do the handshake and grant permissions for a particular application a single time. We can achieve this by saving the cred variable in *R*'s usual way, i.e., via the save() function. Then we can load it into other *R* sessions and the tokens will still be valid (unless the user has revoked the permissions on the server). Of course, one can also call the handshake() function again and repeat the negotiations and authorization. Remember that the contents of this object are secret so it is important to be careful that nobody else can read the saved file. If somebody does access it, they can call any of the provider's API methods as that user.

13.2.3 Keeping the Consumer Key and Secret Private

We need to provide our application's key and secret in the call to oauth(), but they should not appear in code that can be read by another person. There are several approaches to help with this. One is to set these values as an *R* option, typically in our .Rprofile file that is read when *R* starts. (It is important to ensure that this file is not readable by anybody else on your computer and also that you do not display your options in an *R* session.) Our code can refer to those options, and then the values will never be seen in the code. For example, we can call oauth() with

```
oauth(getOption("DropboxKey"), getOption("DropboxSecret"),
       requestURL, accessURL, authURL)
```

Similarly, we can also assign the values to variables and then refer to those in our call oauth() as we did above, e.g. cKey and cSecret.

13.2.4 Extending the OAuthCredentials Class

It is convenient to combine and store the consumer key and secret along with the *URL*s needed for the *OAuth* negotiations to gain an access token. By combining them in an OAuthCredentials object, we can easily pass all of these values to handshake(). Of course, it is just as easy to pass them to handshake() individually, e.g.,

```
cred = handshake(c(consumerKey, consumerSecret), uRT, uAu, uAT)
```

We can also use the resulting OAuthCredentials object cred in another call to handshake() to obtain a new access token.

One of the benefits of using the OAuthCredentials object in the first place is that we can define simple subclasses of it for different services. For example, we might define one for Dropbox and another for Mendeley using

```
setClass("DropboxCredentials", contains = "OAuthCredentials")
setClass("MendeleyCredentials", contains = "OAuthCredentials")
```

The purpose of these classes is that we can create more specific OAuthCredentials objects and pass them to *R* functions that are wrappers for the Web service methods for that particular service.

These functions need only check that the credentials are of the appropriate class rather than checking the *URL* string for the request. For example, we might implement downloading a file from Dropbox as

```
dropbox_get =
function(cred, filename, binary = NA, ..., curl = getCurlHandle())
{
  if(!is(cred, "DropboxCredentials"))
     stop("invalid credentials")

  OAuthRequest(cred, "https://api-content.dropbox.com/1/files/",
               c(root = "dropbox",  path = filename),
               binary = binary, ..., curl = curl)
}
```

Using a subclass is good practice as it catches mistakes of confusing credentials for one service with another. The way we have written this wrapper function above (dropbox_get()) also illustrates several related good practices. The function takes the credential object and then explicit arguments for the parameters expected by the *REST* request. Importantly, it also takes a curl handle via the *curl* parameter and any number of other arguments (via the . . . parameter). It uses these in each *HTTP* request within the function. This allows the caller to control the *HTTP* request and also reuse an existing curl handle.

13.2.5 An Alternative Syntax for Invoking OAuth Requests

It can be convenient to think of the request as being a part of the `OAuthCredentials` object. We can call the OAuthRequest() function as

```
cred$OAuthRequest(url, params, methods, ...)
```

This helps to avoid the situation where we forget to pass the `cred` object as the first argument, but instead just focus on the parameters for the specific method we are calling.

The OAuthRequest() function looks at the *method* and determines which helper function to invoke, e.g., oauthGET() or oauthDELETE(). We can bypass this by using the invocation form

```
cred$get(url, params, ...)
```

where we substitute the `get` in `cred$get` with any of `post`, `put`, `delete`, `head` to specify the method.

Note that this form of invocation—`cred$OAuthRequest()`—is also compatible with the original reference-class–based `OAuth` class in the package. That particular implementation can cause problems when the credential objects are serialized and reused after the `ROAuth` package has been changed and improved. Instead of using the new package's methods, the older object uses the older methods from the earlier version of the package. That approach is still available in the package but, for various reasons, we encourage people to use the *S4* class `OAuthCredentials` and the handshake() function for *OAuth* 1.0.

13.2.6 Low-level Details of OAuth 1.0: The Handshake

In this section and the next, we give brief descriptions of two important aspects of *OAuth*. In this section we describe how we use the application's consumer key and secret to get an access token that we can use to make actual *HTTP* requests to access the protected data. In Section 13.2.7 we show how we digitally sign the *HTTP* requests to ensure their integrity and validity.

We use the handshake() function to allow the application to obtain the important access token to access the user's resources on the provider. It can then use this to call methods in the provider's API to access the user's resources. We do not need to understand how the handshake() function gets the access token to use it. However, we explain the details next for those who are interested or need to know.

Generally, in a call to oauth(), we have to specify the request *URL*, the authentication *URL*, and also the access *URL* for *OAuth*. However, we only seem to visit one page in the Web browser. Is it reasonable to ask what is the purpose of these *URL*s? The client uses the request, authorization, and access *URL*s in three sequential steps. In each of these, the purpose is to get a token that it can use in the next step. At the end, the application has a token it can then use to access the resources owned by the user and made available via the provider. The user is us, say JaneDoe. The application is RJaneDoe that we registered with the provider, Dropbox. When we registered the application, we were given a key and secret for the application and this is the key–secret pair we pass to oauth(). For accessing Dropbox, we create the `OAuthCredentials` with a call such as

```
cred = oauth(consumerKey = cKey, consumerSecret = cSecret,
             requestURL = uRT, authURL = uAu, accessURL = uAT)
```

If it helps, we can think of the `OAuthCredentials` object as the application in the *OAuth* workflow. We, the *R* user, are the user in the *OAuth* workflow. Of course, Dropbox is the provider, and the files in our Dropbox account are the resources.

Step 1: Application asks provider for an unverified request token
When we call the handshake() function with the value of the `cred` variable, the application (our `cred` object) contacts the provider (Dropbox) using the request *URL* to ask the provider for a *request token*. This is a token that the application can use to ask the user for permission to access the resources on the provider.

The application asks the provider for the request token using its own consumer key and secret. The provider checks these are valid application identifiers and issues the token. If the key or secret is not correct, then the provider does not issue a token and the application will not be able to move to the next step.

Step 2: User grants authorization to the application
The application takes the unverified request token and asks the user to authorize it via the provider's Web site, specifically the authorization *URL* (given by the *authURL* argument in oauth()). This requires the user to be logged into the provider's Web site so this might first bring the user to the provider's login page and then to the actual authorization *URL*. When the user grants the permissions requested by the application, the provider will issue a verified request token. This is the string that the user copies from the browser back to the *R* session where the oauth() function is waiting before it can proceed to the third and final step.

Dropbox, in fact, does not issue a new token for the verified request token, reusing the request token. In this case, we can skip the step of copying the new verified request token to *R*, since there is no new token.

Step 3: Application exchanges the verified request token for an access token
The final step in the handshake involves the application exchanging the user-verified request token for an access token that it can use to actually access the user's resources on the provider. It sends the user-verified request token (along with the application's own key and secret) to the provider via the access *URL* (given by the *accessURL* parameter). The provider returns a new access token. The application then stores that. The application can then use this access token in any later calls to methods in the provider's API in order to access the user's resources.

As we mentioned, the entire handshake process focuses on tokens and there are three tokens—the (unverified) request token, the verified request token, and the access token. The user is involved in the second step; only the application and provider are involved in the negotiations for the first and third steps.

13.2.7 Low-level Details of OAuth 1.0: The Digital Signature

The `ROAuth` package also digitally signs each *HTTP* request performed via OAuthRequest() and its helper functions (oauthGET(), oauthPOST(), etc.). This digital signature involves combining the *URL* of the method request, all of the parameters in the request, the access token, the consumer key, and a time stamp to identify to the server when the request was constructed (and avoid people replaying them at a later date). These are combined into a single string and then the "signature" or hash-based authentication code (HMAC) (or other signing methods) for that string is computed using the consumer secret. This string is then attached to the request in the *HTTP* header via an *Authorization* field so that the server can verify the contents of the request and ensure that they have not been modified by anybody else. For example, the following illustrates the header for a request to Dropbox to retrieve the contents of a file

```
GET /1/search/dropbox/?
 query=crime-data.csv&include_deleted=FALSE HTTP/1.1
Host: api.dropbox.com
Accept: */*
Authorization:
     OAuth oauth_consumer_key="nm3ihzplerdat6l",
     oauth_nonce="vgXYKqo3qXjVvDPHtv7HA5YetuHwm",
     oauth_signature="WWIbRAdajyz5wgBXIy3akKCOHVU%3D",
     oauth_signature_method="HMAC-SHA1",
     oauth_timestamp="1351700581",
     oauth_token="275cr75wkqa5aps",
     oauth_version="1.0"
```

We see the different elements that go into creating the signature that the server needs to verify the value.

The server verifies the signature by looking up its own copy of the secrets associated with the consumer key and the access token. It then has all of the information that the signer had and so can reproduce the signature. If they do not match, the server rejects the request. The signatures may not match because of an error in the client's signing code (or indeed the server too). However, when the code is correct, any mismatch indicates that somebody changed the actual request, e.g., one or more of the parameters. Others cannot create an appropriate new signature because they do not have the consumer and access secret. The server also checks the time stamp and will reject a request if it is

from too long ago. It has to allow for some network latency delay in the request, but it will reject a request if it exceeds some threshold. The signature also uses a "nonce," a random number with the sole purpose of being used just once. This nonce is contained in the signature and in the Authorization field. The server examines the nonce and looks at its records. If the nonce was used before, the request is rejected as being invalid. This prohibits a malicious man-in-the-middle from intercepting the request and replaying it another time, e.g., transferring money from an account.

There are many more details in the signing process that we have not discussed here, such as (lexicographically) ordering the parameters in the entire request, escaping certain characters by converting them to base-64, etc.

13.3 *OAuth* 2.0 and Google Storage

Up to now in this chapter, we have described *OAuth* 1.0. As mentioned, *OAuth* is powerful, but the details are complex, specifically the digital signing of each request. Fortunately, the `ROAuth` package hides these details. However, *OAuth2* is a good deal simpler because it uses secure *HTTP* (*HTTPS*) and avoids the need for a complicated signature process. Since it is quite simple, we can manually and explicitly implement the necessary *OAuth2* steps to both obtain a token and use the token to make a request. We show how to do this using Google Storage [4] as a specific example. You can read the code to get the idea. However, to run the code you will need to register an application with Google Storage API. Before that, you will need to enable the Google Storage service for your Google login.

Google Storage is a *REST*ful Web service that allows people to store, access, and share data in the cloud in a secure manner, similar in spirit to Amazon's Simple Storage Service (S3) [1]. We can upload, download, and copy files in buckets (corresponding to folders) and query, set, and change permissions (access control lists) using *HTTPS* requests. Authentication is done using *OAuth* 2.0. See `https://developers.google.com/storage/` for more details about setting up an account, using the API, etc. The RGoogleStorage package [12] provides an *R* interface to this API and hides much of the *OAuth* 2.0 details. However, in this section, we look at how the package performs the authentication to illustrate how we can work with *OAuth* 2.0 in *R* for other providers.

The workflow for using *OAuth* 2.0 has a basic similarity to that of *OAuth* 1.0. We still have the application, user/owner, and provider. In this case, Google Storage API is the provider. We, again, need to register an application with the provider. As with *OAuth* 1.0, this results in a private key and secret so must be done for each user wanting to access Google Storage from within *R*. We create the application via Google's API console [5]. The steps involve creating a new project, selecting the API or service of interest, specifying a product name, setting the `Application type` option to `Installed application` (rather than a Web application or Service account) and generating an *OAuth* 2.0 client id. We do not describe specifically how to do all of these here as there is perpetually up-to-date documentation on the API Web page. The successful creation of an application will be a Web page that looks like the page shown in Figure 13.3. The important information we will need are the Client ID and the Client secret. We assign these to an *R* variable or set them as options() in *R* so we can refer to them without showing them in the code.

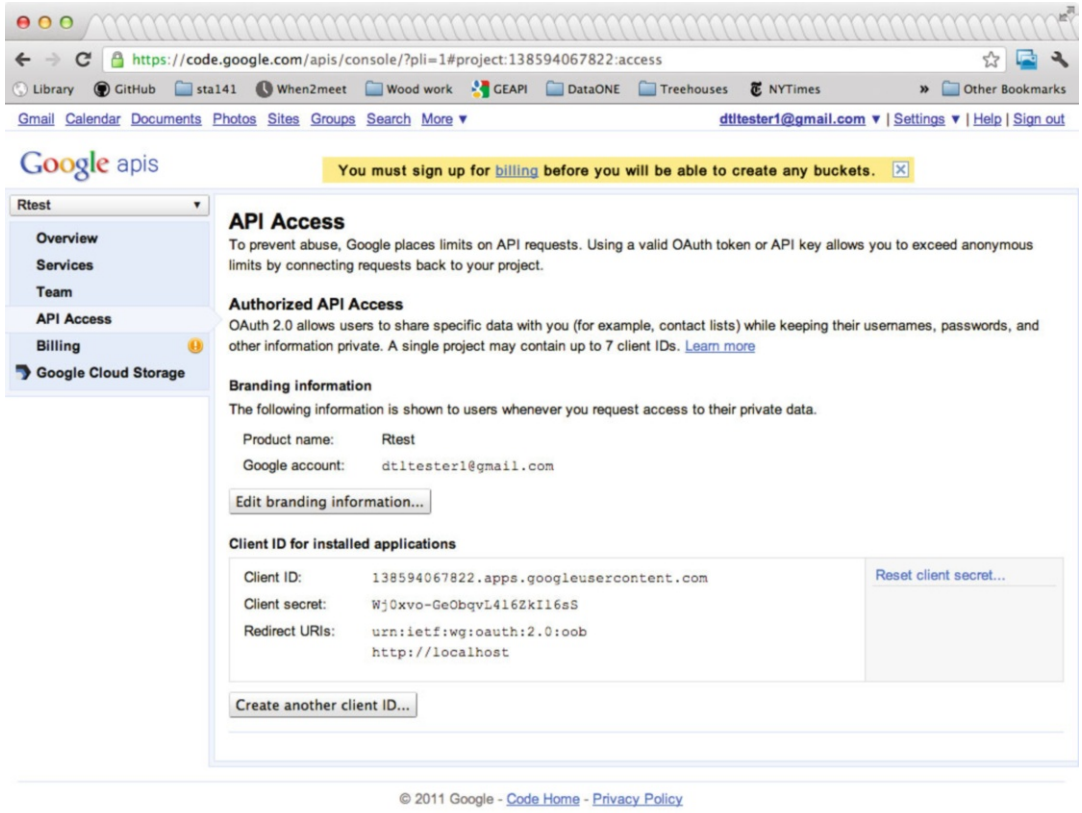


Figure 13.3: Google Storage Page for Authorized API Access. Having created the application to access Google Storage, we need the values of the Client ID and Client secret in *R* in order to get authorization and access tokens.

13.3.1 Getting the User's Permission and the Authorization Token

We now have our application set up. Next, we need to have the user (also us) authorize access to our Google Storage account and its buckets and files. As with *OAuth* 1.0, this is a negotiation between all three parties—the user, the application, and the provider. However, it is a little simpler than with *OAuth* 1.0. We leverage the Web browser to have the user visit a Web page to grant our application the authorization it needs. To do this, we construct the *URL* to visit in *R* and tell the user's Web browser to view that page via the browseURL() function in *R*. We cannot do this with an RCurl request as we need the Web browser so that the user can log in to their Google account and interact with the page to grant authorization and obtain the authorization token.

To create the *URL*, we need to combine our application identifier (Client ID from above) and also information indicating the scope, or set of permissions, we want access for, e.g., read, read and write, or just write. We send the request to `https://accounts.google.com/o/oauth2/auth`. Note that we are using *HTTPS* in all of our *OAuth* 2.0 requests. We add on the different parameters as part of a **GET** request, i.e., as `name=value` pairs, separated from each other via `'&'` and separated from the *URL* by `'?'`. The *URL* might look something like

```
https://accounts.google.com/o/oauth2/auth?
  redirect_uri=urn:ietf:wg:oauth:2.0:oob&
  scope=https://www.googleapis.com/auth/devstorage.read_only&
  client_id=xxxxxxx.apps.googleusercontent.com&response_type=code
```

We, of course, replace the `xxxxxxx` in the `client_id` parameter value with the actual identifier for our application. We have also set the `redirect_uri` parameter to the reserved string `'urn:ietf:wg:oauth:2.0:oob'`. This URI indicates that there is no Web page for the application to callback to and that the application is a stand-alone/desktop application. The `"oob"` in this URI stands for "out of band." This means the token will be returned to us directly in the Web browser rather than the browser being redirected to a Web page specifically for our application.

Once we have created the *URL* string in *R*, we pass it to browseURL() to display this in our Web browser. This will show us the page to grant permissions, bringing us first to the login page of our Google account, if necessary, to grant the permissions. The page to grant permission will look like the screenshot displayed in Figure 13.4. We click on the `Allow access` button and then we are shown a page with a single text field containing the all-important access token. The page will be similar to that shown in Figure 13.5.

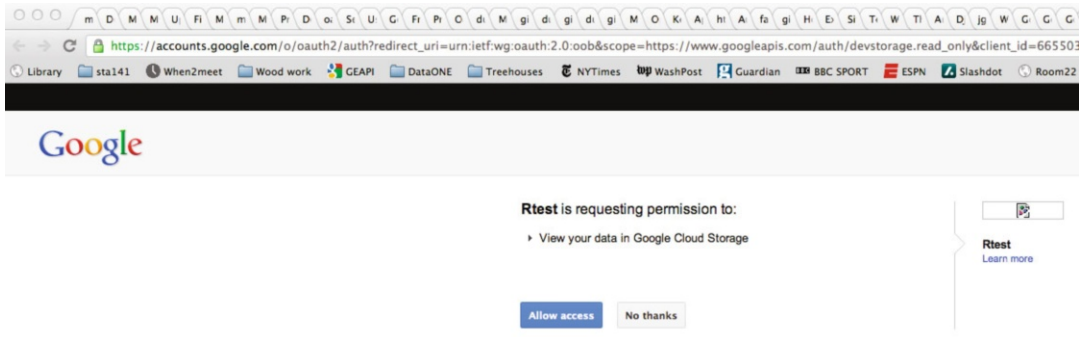


Figure 13.4: Google Storage Page of an Application Asking a User for Access. This screenshot is of the Web page the user is shown when an application asks for access to the user's files. The user clicks on one of the buttons to permit or deny access.

We then cut-and-paste this access token string into *R* and assign it to a variable, say

```
token = "4/Bf18Xp0_CC-NVUZGG_uVBIOaZ8IL"
```

Since this is just a string and it does not tell us anything about its purpose or content, we like to define a class for representing the string and indicating its purpose. When we forget where we got the value, we can check the class. To do this, we use the following code:

```
setClass('OAuth2PermissionToken', contains = 'character')
token = new('OAuth2PermissionToken', token)
```

Better yet, we can make this a Google token or even a Google Storage token with

```
setClass('GoogleOAuth2Token', contains = 'OAuth2PermissionToken')
setClass('GoogleStorageToken', contains = 'GoogleOAuth2Token')
token = new('GoogleStorageToken', token)
```

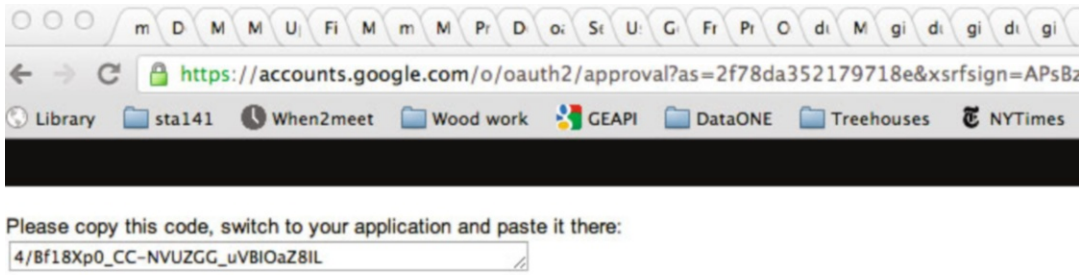


Figure 13.5: Google Storage Page with the Authorization Token. This is the next page the user sees in the Web browser when granting access to an application. The user copies the authorization token in the text field back to *R* so that the application can exchange it for an access token to be able to make requests to the API.

We, of course, only need to define these classes once and create different instances at any time.

13.3.2 Exchanging the Authorization Token for an Access Token

We now need to exchange the authorization token for an access token. The user is no longer involved in this negotiation, which is simply between our application and Google. Of course, we are doing this in *R*, so we are involved, but as the application. For this step, we need to use our client identifier and also its secret. We send an *HTTPS* **POST** request to `https://accounts.google.com/o/oauth2/token`, and include as arguments the:

1. authorization token we just received from the user,
2. client id and client secret identifying our application,
3. `grant_type` as `'authorization_code'`, and
4. `redirect_uri` as before (`'urn:ietf:wg:oauth:2.0:oob'`)

We do not need the Web browser at this point as the user is not involved. Instead, we send the request directly from *R* to Google. We can use postForm() to do this with the code

```
args = c(client_id = client_id, client_secret = client_secret,
         grant_type = 'authorization_code',
         code = as(token, "character"),
         redirect_uri = 'urn:ietf:wg:oauth:2.0:oob')
txt = postForm('https://accounts.google.com/o/oauth2/token',
               .params = args, style = "POST")
```

The result contains the actual access token. Google returns the information in *JSON* format, such as

```
{
 "access_token" :
 "ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNtCqaI7FJ7VTPAK3qDRQ",
 "token_type" : "Bearer",
 "expires_in" : 3600,
```

```
 "refresh_token" : "1/bVto2xxzbv_Pz8kHluGR5idfTy8qdiSFBvwIVsUEvYM"
}
```

Other providers may use *XML* or another format.

The *JSON* content gives us the access token and also its type (Bearer) and when it expires (as number of seconds). Some providers will just return these and the token will be valid forever. Most providers, however, will have the token expire after a given time, say 60 minutes as here. Rather than have the user grant permissions again after the access token expires, *OAuth* 2.0 allows the provider to return a refresh token along with the access token. The application can use this to renew the access token when it expires, without the user being involved. We discuss precisely how to do this later.

To get the access token from the *JSON* content, we can use the fromJSON() function (see Chapter 7 on how to use this function) and then save the information as an *S4* class. We define a class `OAuth2AuthorizationToken` that holds the access token, the refresh token, and also the expiration time. We calculate this time by adding the value of the `expires_in` field to the current time `Sys.time()`.

13.3.3 Using the Access Token in an API Request

We are now going to use the access token to access data in Google Storage. First, we list the contents of an existing bucket named `proj1`. To do this, we send an *HTTPS* request to `https://commondatastorage.googleapis.com/proj1` We put the access token and our client identifier into the header of the *HTTP* request. This looks something like

```
                                          Authorization
"OAuth ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNtCqaI7F...3qDRQ"
                                    x-goog-project-id
                                       "xxxxxxxxxxxx"
                                                Date
                          "March, 29 Mar 2012 13:11:16 PDT"
                                    x-goog-api-version
                                                "2"
```

The Authorization field contains the access token in the `OAuth2AuthorizationToken` object that we obtained above. We prefix this access token with the string `"OAuth"`. The project ID is just the numbers identifying our application. We add the current date and time so that Google can verify that the request is not being "replayed" at some later time. Finally, we specify the version of the Google API to which we are sending the request. Our request can be made manually as

```
hdr = c(Authorization =
        "OAuth ya29.AHES6ZR9ZrB4kqY2W9iGm2fm5QNt...3qDRQ",
        'x-goog-project-id' = clientID,
        Date = format(Sys.time(), "%B, %d %b %Y %H:%M:%S %Z"),
        'x-goog-api-version' =  "2")
url = "https://commondatastorage.googleapis.com/proj1"
txt = getURLContent(url, httpheader = hdr,
                    useragent = "RGoogleStorage")
```

The result is an *XML* document that contains an element for each "file" in the bucket (corresponding to a folder). We can then process this and convert it to a data frame in *R* listing the name of the document, when it was last modified, its size in bytes, the owner's name, and unique identifier.

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://doc.s3.amazonaws.com/2006-03-01">
 <Name>proj1</Name>
 <Prefix/>
 <Marker/>
 <IsTruncated>false</IsTruncated>
 <Contents>
  <Key>bar</Key>
  <LastModified>2011-05-18T13:05:11.187Z</LastModified>
  <ETag>"5578833a0c6cb26394a1414140718cab"</ETag>
  <Size>12</Size>
  <StorageClass>STANDARD</StorageClass>
  <Owner>
   <ID>00b4903a97f8e9...0edc6aaee</ID>
   <DisplayName>Duncan Temple Lang</DisplayName>
  </Owner>
 </Contents>
 <Contents>
  <Key>myPlot</Key>
  <LastModified>2011-05-18T12:59:26.190Z</LastModified>
  <ETag>"0e49b507686b4ad978ef53832c11c157"</ETag>
  <Size>14184</Size>
  <StorageClass>STANDARD</StorageClass>
  <Owner>
   <ID>00b4903a97f8e9...edc6aaee</ID>
   <DisplayName>Duncan Temple Lang</DisplayName>
  </Owner>
 </Contents>
</ListBucketResult>
```

We have manually created the content for the *httpheader* setting in several places above. We should do this in a function since the format is the same in each of these places. The makeHeader() function in the RGoogleStorage package can be used for this task.

Let's look at an API method where we can create a document within a bucket. We specify the path of the file we want to create (or overwrite) and send the request to `https://commondatastorage.googleapis.com/path/to/file`. Here we are sending content from *R* to Google Storage. We do this via a **PUT** operation. We get the content of the document either from a file or an *R* object in memory and then upload it using the curl options *readfunction* and *infilesize*. Again we create our own value for *httpheader* including the Authorization token and the client secret.

In this case, we are writing to the storage facility. For this we need write privileges to the user's Google Storage account. However, we obtained our token for reading only. This means we need to create a new permission and access token to be able to perform this operation. To do this, we repeat the steps in Section 13.3.1. The only thing we need to change is the scope, which we set to `https://www.googleapis.com/auth/devstorage.read_write`.

We have not covered how to send arguments in a request other than upload the contents of a file. It turns out that in the Google Storage API, this is how we pass arguments other than identifying the bucket or document in the *URL* of the request. For instance, to download the contents of a document in a bucket, we use the full *URL* to that document, e.g., `https://commondatastorage.googleapis.com/proj1/abc`. To get the access control list (ACL), or in other words, the permissions for a bucket, we use the `acl` parameter as part of a **GET** request, e.g.,

```
getForm("https://commonda...googleapis.com/phase1/foo",
        acl = "", .opts = list(httpheader = oauth2Header))
```

where `oauth2Header` is our header including the token, client secret, and date as above. When setting an ACL on a bucket, we actually upload an *XML* document that describes the permissions. When copying a document, we specify the target via the *URL* and the original document by specifying its path in the *x-goog-copy-source* field in the *HTTP* header.

13.3.4 Refreshing an OAuth2 Access Token

As we saw when we got the access token, we also received the number of seconds until the token expired and a refresh token. We put the expiration time in the `OAuth2AuthorizationToken` object. When we use this object in one of our functions, we should check to see if this access token has expired (by comparing the expiration time to the current time). If it has, we can use the `refresh_token` to obtain a new access token. This is very similar to the step we used to exchange the original user-granted token for our access token. We **POST** a request to `https://accounts.google.com/o/oauth2/token` with the client id and secret. Instead of the *code* and *redirect_uri* arguments, we specify arguments named `refresh_token` and `grant_type`. For the former, we provide the value of the refresh token we were given; for the latter, we specify the string `'refresh_token'`. A request might look like

```
args = c(client_id = getOption("Google.storage.ID"),
         client_secret = getOption("Google.storage.Secret"),
         grant_type = 'refresh_token',
         refresh_token = token@refresh_token)

postForm('https://accounts.google.com/o/oauth2/token',
          .params = args, style = "POST")
```

This again returns a new access token and the expiration duration in *JSON* format and we can again turn this into an `OAuth2AuthorizationToken`.

We can encapsulate getting and refreshing a token into a function so that the details are hidden from the caller. The getAuth() function in the RGoogleStorage package does this and the function getPermission() takes care of making the initial request and guiding the caller to the Web browser to grant the permissions.

We should write our functions for API methods so that they check the expiration and update the `OAuth2AuthorizationToken` as necessary. Unfortunately, if any of these functions update the token, then that function may find it difficult to return the new token as part of the regular value returned from the API method called. For example, suppose we have a function download() to retrieve a document from Google Storage. This function should return the contents of the document, but if it refreshes the token, that new access token will be discarded when the function returns. This is fine as

the other functions can also refresh an expired token when they are called, but there may be a lot of unnecessary refresh requests. Instead, we can issue a warning and ask the user to explicitly refresh the token before the next function call. However, an alternative is to use a mutable object to represent the token and refresh token, i.e., the `OAuth2AuthorizationToken`. We can pass this mutable object to our functions which will refresh them if necessary. Since they update that mutable object, the new token and expiration time are in the original object passed to the function, and there is no need for the function to return this additional information. This is an example where using a reference class is appropriate simply because we cannot easily return the updated token as well as the actual result of a function.

13.4 Summary of Functions for Using *OAuth* in *R*

The `ROAuth` package provides facilities in *R* to use the *OAuth* mechanism to enable *R* to access data owned by a user and hosted by a third-party Web service. The package negotiates the creation of request and access tokens and also the signing of *HTTP* requests with the access token for *REST* services that require *OAuth* authentication. There are both *S4* and corresponding reference-class methods to use *OAuth* in *R*. The *S4* methods for handling this process are described below. The reference-class mechanism is essentially the same but with the form `method(cred = auth, arg1, arg2, ...)` replaced by `auth$method(arg1, arg2, ...)`

oauth() Create an `OAuthCredentials` object that contains the necessary information with which we can later obtain an access token from a Web service provider. This information includes the consumer key and secret for the application, which are provided in the *consumerKey* and *consumerSecret* parameters, as well as three *URL*s used to negotiate the request for an access token. These *URL*s are provided via the *requestURL*, *authURL*, and *accessURL* parameters, which are the *URL*s used to initiate the request, grant authorization, and access the authorization key, respectively. We pass this object to the handshake() function to perform the negotiations to obtain the access token. An updated object is returned that includes the access token with which we can then make authenticated requests.

handshake() Negotiate with the service provider to gain an access token to make authenticated requests to the provider. This function takes as input the `OAuthCredentials` object created by oauth() (passed to handshake() in the *cred* parameter) and returns an updated `OAuthCredentials` object that contains an additional key and secret needed to make requests.

OAuthRequest() Invoke a *REST* Web service method using *OAuth* signing. This function signs the request with the credentials object returned from the call to handshake(). We specify the *URL* to which we send the request via the *URL* parameter. Input values (or arguments) to the request can be supplied via the *params* argument. We can make different types of *HTTP* requests (e.g., **GET**, **DELETE**, etc.) with this generic request function via the *method* parameter. We can also specify different `libcurl` options and a `CURLHandle` connection object to use for the *HTTP* requests.

13.5 Further Reading

There are several good books and online documents that describe *OAuth* 1.0 and *OAuth* 2.0. The guide by Hammer [7] has a good explanation of *OAuth* 1.0 written in a manner that explains it to users of

OAuth in very clear terms. It is written by the primary author of the *OAuth* 1.0 specification. The guide explains all the details of *OAuth* 1.0, from high-level concepts to details about how to sign each request. There is even an interactive document to illustrate all the steps of the signing process for those who need to understand this.

Chapter 9 of [9] provides a very readable description of *OAuth* 1.0 and also *OAuth* 2.0. If the discussion at `hueniverse.com` is not entirely clear, this book should help to clarify those difficult concepts from a different perspective. The book is also interesting for other topics it covers.

Google has a good overview of *OAuth* 2.0 [6] and how to use it. This is quite clear and comprehensive.

The official specification for *OAuth* 1.0 is [8]. As such, it is a little more pedantic than explanatory. It is a good reference when writing the code to implement *OAuth* 1.0.

References

[1] Amazon Web Services, Inc. Amazon simple storage service (Amazon S3). `http://aws.amazon.com/s3/`, 2012.

[2] Robin Gareus. liboauth: *OAuth* Library functions, version 1.0.0. `http://liboauth.sourceforge.net`, 2012.

[3] Jeff Gentry and Duncan Temple Lang. `ROAuth`: *R* interface for *OAuth*. `http://cran.r-project.org/web/packages/ROAuth/index.html`, 2012. *R* package version 0.9.2.

[4] Google, Inc. Google cloud storage: A *REST*ful service for storing and accessing data on Google's networking infrastructure. `https://developers.google.com/storage/`, 2011.

[5] Google, Inc. Google APIs console. `https://code.google.com/apis/console/`, 2012.

[6] Google, Inc. Using *OAuth* 2.0 to access Google APIs. `https://developers.google.com/accounts/docs/OAuth2`, 2012.

[7] Eran Hammer. The *OAuth* 1.0 guide. `http://hueniverse.com/oauth/guide/`, 2011.

[8] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF), 2010. `http://tools.ietf.org/html/rfc5849`.

[9] Johnathan LeBlanc. *Programming Social Applications: Building Viral Experiences with OpenSocial, OAuth, OpenID, and Distributed Web Frameworks*. O'Reilly Media / Yahoo Press, Sebastopol, CA, 2011.

[10] Karthik Ram and Duncan Temple Lang. `rDrop`: Dropbox *R* interface. `https://github.com/karthikram/rDrop/`, 2012. *R* package version 0.3.

[11] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. `http://www.omegahat.org/RCurl`, 2012. *R* package version 1.95-3.

[12] Duncan Temple Lang. RGoogleStorage: Accessing the Google storage API from *R*. `http://www.omegahat.org/RGoogleStorage`, 2012. *R* package version 0.1-0.

Part III
General *XML* Application Areas

Overview

In this part of the book, we turn our attention to some different applications of *XML*. Our intent is to illustrate where *XML* is being used in interesting general areas that should be of relevance to data scientists. These are more than specialized applications. Instead, they are general topics such as spreadsheets, dynamic text documents, interactive and dynamic visualization, spatial–temporal displays with Google Earth, and programmatic code generation from descriptions of data structures. These provide rich possibilities and plenty of opportunities to do new things.

We start by looking at *XML* schema. We encountered these when looking at *SOAP* and *WSDL* documents. Schema are used to define how particular data structures are represented in *XML*. A schema describes the structure of a family of related *XML* documents, specifying the possible attributes and child nodes of different *XML* elements. As we did implicitly in the chapter on *SOAP*, we can read an *XML* schema in *R* and map the data types it describes to corresponding *R* classes and also generate code to read and write *XML* documents. Chapter 14 discusses these ideas and the *R* functionality to work with *XML* schema.

The remaining chapters in this part of the book do relate to specific *XML* vocabularies, but the formats are not the primary focus. Instead, the chapters illustrate general-purpose, commonly used technologies or technologies that we imagine will become more commonly deployed by statisticians and that use *XML* as the primary representation of the content.

We start by looking at spreadsheets. Today, all office suites store spreadsheets as **zip** archives that contain *XML* files that represent the entire state of a spread sheet and its workbooks. By understanding the general structure, we can extract data from spreadsheets and also create new spreadsheets and use the platform to provide interactive displays of data, results from data analyses, and so on. This provides interesting opportunities for us to convey results in new ways. The format for spreadsheets also shares many similarities with word processing documents and presentations (slides) and so understanding spreadsheets helps us to access all types of common documents.

After spreadsheets, we turn to relatively new opportunities for graphical displays. Scalable Vector Graphics (*SVG*) is an *XML* vocabulary for representing interactive, dynamic two-dimensional vectorized graphical displays. *SVG* is well supported in Web browsers and allows us to embed interactive plots with Web pages. We can create *SVG* plots in *R*, but more importantly we can use *R*'s existing graphics capabilities to create the plots and then enhance them to add interactivity, animation, and linked plots. *SVG* provides some of these features itself, but others are provided by embedding the *SVG* document within Web pages and leveraging the *JavaScript* programming language to operate on the *SVG* elements. This combination allows us to create very rich displays that combine *SVG* and other *HTML* components and plug-ins. It also connects to using *JSON* to serialize data from *R* to *JavaScript*.

While *SVG* is used for general two- and three-dimensional displays, *KML* is an *XML* vocabulary that is used for representing spatial/geographical and spatial–temporal data. Importantly, *KML* is the format used to create displays in Google Earth and Google Maps. We can display points or surfaces or more complex structures on a globe and then viewers can interact with these displays and control what other features to display. We can also project *R* plots directly onto the globe. We can make all of these displays interactive by integrating the *KML* with *JavaScript* code within a Web page. This represents tremendous potential for interactive spatio–temporal displays.

In Chapter 18, we briefly outline some of the potential for authoring documents (journal articles, tutorials, books, etc.) using an *XML* vocabulary such as *DocBook*. The goal of this chapter is to suggest possible ways to improve authoring productivity, and also to hint at a qualitatively different type of document that contains all of our work and not just what the reader sees. The document becomes a notebook or database of a data analysis or simulation experiment. We can project it into different formats (static and interactive) and also create different versions, or views, for different audiences.

Our goal with each of these topics is to not only introduce the reader to the concepts and existing functionality in *R*, but also to encourage further advances in these and similar areas.

Chapter 14
Meta-Programming with *XML* Schema

Abstract This chapter discusses the functionality to read, process, and use *XML* schema within *R*. The primary goal is to be able to programmatically generate *R* classes and code to work with *XML* documents. Recall from Section 2.7 that an *XML* schema describes the structure, content and data types for *XML* documents in a particular *XML* vocabulary, grammar or format (e.g., *PMML* or *KML*). The information in the schema can be turned into *S4* class definitions and *R* functions for reading and generating *XML* in *R* for that *XML* vocabulary. We can use these *S4* classes to represent data that appear in the *XML* documents as *R* objects corresponding to the schema. This means we can both read *XML* into objects of these classes and also generate *XML* by serializing these *R* objects to *XML*. The key idea is that these classes and methods for reading and writing *XML* are generated programmatically. The XMLSchema package provides code that generates code for a given schema. *R* users do not need to interact directly with the contents of a schema, but can benefit from them to create structured *R* code and data.

14.1 Introduction: Using Information from *XML* Schema

We discussed *XML* schema in Section 2.7. Schema can be used to validate an *XML* document—either one we have received or one we generated. The validation would tell us if it was not just well-formed *XML*, but also conformed to the type of document(s) described by the schema. However, a schema can be used for much more than just validating an *XML* document. Consider the following snippet of an *XML* document:

```
<item>
 <value>1.3</value>
 <when int="P1D">2012-1-10</when>
 <col>ff00aaee</col>
 <status>ok<code>0</code></status>
 <location>
  <long>W122.41</long><lat>N37.77</lat>
  <zip>94129</zip>
 </location>
</item>
```

If we convert this to *R* using xmlToList() in the XML package [16], we get a named list with elements corresponding to each node, and each individual value in the resulting tree is a single string, e.g.,

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_14, © Springer Science+Business Media New York 2014

```
xmlToList(doc)

$value
[1] "1.3"

$when
$when$text          $when$.attrs
[1] "2012-1-10"       int
                    "P1D"
$col
[1] "ff00aaee"

$status
$status$text        $status$code
[1] "ok"            [1] "0"

$location
$location$long      $location$lat      $location$zip
[1] "W122.41"       [1] "N37.77"       [1] "94129"
```

(We have reformatted the above *R* output for readability.) Unfortunately, we have no information about the types of the values other than that they are strings. It turns out that this node describes a measurement taken by a device. The text within the <*value*> node is a real-valued number. It is limited to be between 0 and 10. The <*when*> element contains a date and its *int* attribute is an interval/duration, with a value of one day (`"P1D"`). The value `"ff00aaee"` for <*col*> is a hexadecimal value consisting of exactly four components made up of pairs of hexadecimal digits (i.e., from the set 0, 1, 2, ..., 9, a, b, ..., f). The <*status*> element contains a string, but it is one of a limited set of possible strings from "ok", "low power", "test", "failing", and "failed". The <*code*> node gives us the status/error code, which is a non-negative integer between 0 and 27. The <*location*> element gives us the latitude and longitude as numbers with direction qualifiers (i.e., N and W for north and west), and we also have the five-digit ZIP code in <*location*>. We want to be able to read this *XML* content into *R* and have the data types be computed automatically. (We also want to decide whether to include the attributes in the *R* content, e.g., *int* and *PID*.) We do not want to manually determine these data types and then transform the string values to these.

Information about the data types is typically described in an *XML* schema, along with documentation for the different elements and attributes. This information allows us to represent the values much more meaningfully in *R*. We can represent the <*when*> value as a `Date`, the contents of <*value*> as a numeric value or even by a more specific class constrained to be between 0 and 10. The <*status*> value can be more usefully represented as a *factor* in *R*. When we convert the *XML* content to *R* objects with these more structured types, it makes it easier and more reliable to immediately work with these values. We can, of course, convert them after we convert from *XML* to *R* in a second step. However, this requires human intervention and knowledge about the types, and also involves a second round of processing. The goal is to make this more automated, less error-prone, and more complete.

When generating *XML* from *R* objects, we have similar problems as when reading *XML*. For example, how do we convert the simple string "abc" to *XML*? It depends on the target schema or expectations of the application(s) that will process the resulting *XML*. Of course, we put it in an appropriate *XML* element, but should we put the value as the child of the element, or as an attribute?

Since this is a vector of length 1 in *R*, we can treat it as either a vector or as a scalar value. Should we map it to a scalar value or an array with just one element? For example,

```
<genes_id_list>eco:b0078</genes_id_list>
```

or

```
<genes_id_list xsi:type="Array" arrayType="string[1]">
  <item>eco:b0078</item>
</genes_id_list>
```

This difference matters in many applications that consume the resulting *XML*, e.g., a SOAP server. Again, the schema tells us what is expected and required and we can remove the need to manually specify this information, perhaps erroneously. We can do this manually, but it is preferable to avoid any of these details. Instead, we can take this information from the schema and use it to create *R* functions that actually generate the appropriate *XML* from a suitable *R* object.

While there are many *XML* documents that have no schema, many of the commonly used *XML* formats provide, and are defined by, a schema. For example, we have a schema for *PMML* and separate ones for *KML* and *GML*. The schema identifies all the *XML* elements within that vocabulary and defines their possible content and any attributes and the possible values or types those attributes can have. It indicates that a particular element or attribute may be a date, a date–time, a year, an enumerated string, a number, a constrained number, a sequence with values between *k* and *n*, and so on. The schema also defines how the *XML* elements are related by specifying which elements are possible children of other elements and how many are expected in what positions. Within a schema, there may also be intermediate data types defined, which are reused when defining elements and attributes.

If we can understand an *XML* schema, we have full knowledge of the structure of a class of *XML* documents, i.e., an *XML* grammar. If we can gain this knowledge programmatically, a multitude of opportunities arise. We can generate *S4* class definitions corresponding to the data structures represented in the *XML* document. We can programmatically generate *R* code that will map an *XML* document adhering to this schema to corresponding *R* data structures. Similarly, we can automate the creation of code to serialize, or write, an *R* object of a particular type to a suitable *XML* format corresponding to the schema.

The ability to process schema allows us to dynamically "interface" with different *XML* vocabularies. We mean dynamic in the sense that we can, during an *R* session, "discover" an *XML* schema and generate code on-the-fly. We can use this code within the *R* session or write it to an *R* package for others to use in the future. The XMLSchema package [19] can be used to interface to any *XML* schema. It is extensively used as part of the SSOAP package [17] to read the data type definitions with a Web Service Description Language (*WSDL*) [1] document, i.e., the schema within the *WSDL* document (see Chapter 12). We generate functions to invoke those Web service methods and the functions in the XMLSchema package handle creating the *S4* class definitions and code to read and write the *XML* used to marshal inputs to the methods and the output back to *R*. In this chapter, we see how we can use it on different schema to create this code for direct use in parsing and also generating *XML*.

Before we explore how to use schemas to generate code to read *XML* into *R* objects and generate *XML* from *R* objects, we should point out that there is a great advantage to this approach generally. The reason is because it removes humans from a significant part of the processing work. This saves time, reduces the overhead of getting started, and results in fewer errors. However, while it is quite general, it may not be the best approach for every task. In different circumstances, it may be best to parse and process the *XML* documents directly oneself. For instance, if we want a single piece of data or variable from a document, then using getNodeSet() and *XPath* is likely to be more efficient and straightforward. Similarly, if we have an *R* object that is not compatible with the *S4* structures

that we generated programmatically, then it may be best to bypass the generated code and generate an *XML* representation for the *R* object using the tools described in Chapter 6. For example, suppose we have a data frame containing latitudes and longitudes, but not in the *S4* structure generated by the *KML* schema. Should we want to convert it into a *KML* document, we might use functions such as newXMLNode() to create the desired *XML* document rather than spending time converting the data frame to an intermediate type for which there is a method to convert to *KML*.

Finally, in this chapter we predominantly use two example schemas, one for *PMML* and the other for *KML*. Before continuing, we provide in the following example a snippet of *PMML* and a brief discussion to help motivate the usefulness of programmatically generating *R* class definitions from schema.

Example 14-1 Generating S4 Classes Programmatically for PMML

The Predictive Model Markup Language (*PMML*) [3] is an *XML* format for describing the results of one or more statistical models and concepts related to prediction and predictive analytics, e.g., linear model/regression, classification tree, support vector machine, neural network, time series, predictor, parameter, document term matrix, etc. It facilitates reporting and exchanging the result of fitting models, classifiers, etc., between applications. *PMML* is defined by its schema and different applications implement (parts of) the schema. For instance, the `pmml` package [21] provides some facilities for generating *PMML* representations of models created in *R*, i.e., serializing them from *R* to *XML*. Currently there is support for linear and generalized models, support vector machines, random forests, and *k*-means clusters.

Below is a snippet of a *PMML* document that results from fitting a classification tree using the rpart() function in *R*. The dependent variable in the fit is `Kyphosis`, which is a categorical variable with two levels, "absent" and "present". The independent variables are `Age`, `Number`, and `Start`. We see from the snippet that the root node of the classification tree that was fitted to the data can be reconstructed using the *<Node>* elements and their *<simplePredicate>* children, including the criteria for the split at each node in the tree and the prediction that would be used if the tree were trimmed to that node.

```
<?xml version="1.0"?>
<PMML version="3.1" xmlns="http://www.dmg.org/PMML-3_1"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Header copyright="Copyright (c) 2007-2008 Togaware"
   description="RPart decision tree model">
  <Extension name="timestamp"
    value="2008-04-21 18:49:28" extender="Rattle"/>
  <Extension name="description" value="deborahnolan"
    extender="Rattle"/>
  <Application name="Rattle/PMML" version="1.1.6"/>
 </Header>
 <DataDictionary numberOfFields="4">
  <DataField name="Kyphosis" optype="categorical"
    dataType="string">
   <Value value="absent"/> <Value value="present"/>
  </DataField>
  <DataField name="Age" optype="continuous" dataType="double"/>
  <DataField name="Number" optype="continuous" dataType="double"/>
  <DataField name="Start" optype="continuous" dataType="double"/>
 </DataDictionary>
```

```
 <TreeModel modelName="RPart_Model" functionName="classification"
   algorithmName="rpart" splitCharacteristic="binarySplit">
  <MiningSchema>
   <MiningField name="Kyphosis" usageType="predicted"/>
   <MiningField name="Age" usageType="active"/>
   <MiningField name="Number" usageType="active"/>
   <MiningField name="Start" usageType="active"/>
  </MiningSchema>
  <Node score="absent" recordCount="81">
   <True/>
   <Node score="absent" recordCount="62">
    <SimplePredicate field="Start"
      operator="greaterOrEqual" value="8.5"/>
    ...
    </Node>
   </Node>
   <Node score="present" recordCount="19">
    <SimplePredicate field="Start" operator="lessThan"
       value="8.5"/>
   </Node>
  </Node>
 </TreeModel>
</PMML>
```

PMML offers a good example of the benefits of the programmatic approach taken in XMLSchema. For instance, there are several other types of models in *PMML* that we may want to represent in *R*. Furthermore, apparently the `pmml` package is based on a slightly older version of the *PMML* schema, and we may want to update the code to include any additional or modified definitions. This is a case where it would be preferable to have a means of mapping the schema into *R* classes and for programmatically generating *R* code to create the *XML* from an *R* object.

This task can be challenging as the *R* objects may not have a structure that corresponds to the *PMML* description. This can happen with any language or application, and is one of the problems with having a generic representation shared by many languages, but not used by any directly. However, we can generate the classes and the conversion code from the *PMML* definitions. Then we can map the existing *R* structures to these and generate the *XML* or operate on them in different ways.

14.2 Reading *XML* Schema and Generating Code and Classes

The primary purpose of the XMLSchema package is to automate the conversion between *XML* documents and structured *R* objects. The first step is to read and process the schema for a class of *XML* documents and understand the data types. Then we can use these to read any number of documents or nodes from that class of documents.

The basic computational workflow in using the XMLSchema package is described by the following pseudo code. First, we read the schema with

```
schema = readSchema(schemaUri)
```

Then we create the associated classes for the data types the schema contains:

```
defineClasses(schema)
```

This also defines the appropriate methods for the fromXML() function so that after we parse an *XML* document, we pass the tree to fromXML() and it will convert the tree into objects of the appropriate class:

```
doc = xmlParse(docUri)
fromXML(doc)
```

As an alternative to fromXML(), we can coerce an *XML* node (of class `XMLAbstractNode`) via the as() function (implicitly coerce()):

```
as(doc, RClassName)
```

This latter approach is more familiar to *R* programmers than fromXML(). However, we have to know the target class or use the name of the *XML* node and assume that this is an actual class. We use *KML* as an example.

Example 14-2 Reading KML Schema

We read the older *KML* schema[1] and define the classes and methods with

```
kx = "https://developers.google.com/kml/schema/kml21.xsd"
kml = readSchema(kx)
defineClasses(kml)
```

We need only do this once per *R* session. Given these classes and converter functions defined from the schema, we can read any *KML* document and convert its contents to *R*.

We first parse the document

```
doc = xmlParse(kmlUri)
```

We can then convert the entire *XML* tree and its nodes to an *R* object (and sub-objects) with either of the *R* calls

```
fromXML(doc)
```

or

```
as(doc, "kml")
```

These return an object of class `KmlType` that contains the `Document` object and the list of `Folder` objects and different classes of `FeatureType` objects.

It is useful to compare this with the fromJSON() function in RJSONIO and xmlTreeParse() in XML. Both of these functions can read an entire document (in the different formats) and convert all of the elements to *R* objects. The fromJSON() function can recognize numbers and logical values while xmlTreeParse() cannot. However, fromXML() and the as() method we used above recognize more data types. They recognize numbers and boolean values, but also constrained numbers, strings, times, and dates, etc. They also map composite objects (i.e., collections of values) to *S4* objects rather than simple lists, and validate that the contents are correct, both in type and count. Once we have

[1] We are using version 2.1 of the *KML* schema in this chapter rather than the newer ogckml2.2 schema. The latter has directives that explicitly make it more extensible and this adds extra content that is not the focus of this chapter.

processed a schema, this approach is just as easy as fromJSON(), and the resulting objects contain more information.

Now, suppose we are interested in converting just `<GroundOverlay>` nodes to *R*. For example,

```
<GroundOverlay>
 <name>Tampa</name>
 <Icon>
  <color>#88FFFFFF</color> <href>TampaFla.png</href>
 </Icon>
 <LatLonBox>
  <north>29.47</north> <south>26.47</south>
  <east>-81.53</east> <west>-83.53</west>
 </LatLonBox>
</GroundOverlay>
```

We find these with the getNodeSet() function, for example, and then convert them to an *R* class with

```
ov = getNodeSet(doc, "//x:GroundOverlay", "x")
fromXML(ov[[1]])
```

In addition to fromXML(), we can also use as() to coerce an *XML* node to an *R* class, e.g.,

```
as(ov[[1]], "GroundOverlay")
as(ov[[1]], "GroundOverlayType")
as(ov[[1]], xmlName(ov[[1]]))
```

The as() function takes no additional arguments, just the *XML* node and the target class. The fromXML() function, however, allows us to perform the same conversion, but gives us control over some of the conversion. The simple call:

```
fromXML(ov, type = "GroundOverlay")
```

is equivalent to `as(ov, "GroundOverlay")`. Both of these are useful if you expect the target class to be defined and you want an error if that is not the case, i.e., the function fails when the class is not found. However, the simpler call `fromXML(ov)`, where we do not specify the target type, can typically find the mapping from the name of the *XML* node ("GroundOverlay") to the appropriate *R* class (`GroundOverlayType`), using the schema information.

The fromXML() function also allows us to specify the mapping from element names to *R* classes, as well as several other details in the conversion. Finally, if fromXML() cannot identify a target *R* class, it performs a generic conversion. This is similar to a call to xmlToList() but a bit smarter as it looks at type attributes in nodes. This allows it to convert strings to numbers, dates, etc., appropriately.

14.2.1 Writing the Generated Code to a File

When we want to work with a schema within a single *R* session, defineClasses() works well. In other cases, we may want to write the new class definitions and fromXML() and coerce() methods to a file so that they can be used in another *R* session, e.g., at a later time, on another machine, or as part of an *R* package. The XMLSchema package has some facilities for this. We start again by reading the schema and defining the classes, e.g.,

```
kml = readSchema(kx)
o = defineClasses(kml, baseClass = "VirtualXMLSchemaClass")
```

We then take these class names and determine the order in which they should be written to the file. This is often necessary as the class should be defined before it is referenced in another class. We use computeOrder() to determine this, as in,

```
classNames = computeOrder(unlist(names(o[[1]])))
```

We can then open a writeable connection and use writeClassDef() to serialize the code. For example,

```
con = file("/tmp/kml.R", "w")
invisible(sapply(classNames, writeClassDef, file = con,
                 where = globalenv()))
close(con)
```

That code can then be read into *R* with source() or be part of a package.

14.2.2 Customizing the Code Generation

In this section, we describe how the classes are generated by defineClasses(). This function processes all of the *R*-level descriptions of a schema's contents, which are in a `SchemaCollection`. It defines the classes and creates the coercion methods (for both coerce() and fromXML()) for mapping an *XML* node to the corresponding new class. We can influence how defineClasses() and its subfunctions operate via its parameters. We can control in which environment the classes and methods are defined via the *where* argument to defineClasses(). We can also specify the base class for the new classes being defined via the *baseClass* parameter. Having a different base class can be convenient if we want to define methods that apply to all of these new classes associated with the schema, but not classes from other schema. We can also control whether we create a class for *XML* `<element>` types, and not just the explicit data types defined at the top level in the schema, e.g., `<complexType>` and `<simpleType>`.

The defineClasses() function also has a parameter named *opts*. We pass an object of class `CodeGenOpts` as the value for *opts*. This object provides a collection of options that control how the code is generated. Currently, `CodeGenOpts` has slots *makePrototype*, *allowMissingNodes*, and *defineEnumVars*. Each of these is a *logical* value that enables or disables the corresponding feature.

The *makePrototype* option controls whether we create and specify the *S4* prototype when defining a new class from the schema. If *makePrototype* is `TRUE`, we compute the default values for the different slots in our new class, as specified in the *XML* schema via the *default* attribute for each of the different components. If the value of this option is `FALSE`, we ignore the prototype and just pass `NULL` as the value for the prototype and use *R*'s default prototype. The default values are clearly useful as they provide appropriate and intended values for *R* when the element is not present in *XML* content that we receive and process.

The option *allowMissingNodes* controls how we create each function that converts an *XML* node to the corresponding class created from the schema. When this option is `TRUE`, each step in the converter function that accesses a child node or an attribute checks to see if it is present in the *XML* content being processed. It performs this check even if the node or attribute is required and so allows for *XML* content that does not quite conform to the schema. When the *XML* does correspond strictly to the schema, there is no need for these tests and they add additional unnecessary calculations. Specifying *allowMissingNodes* as `TRUE` enables these tests and makes the generated converter

functions more forgiving, at the expense of a marginal speed decrease. A value of `FALSE` means that the converter functions will raise an error when a required node or attribute is not present in the *XML* content.

When we define a class corresponding to an enumerated restricted string, there is a collection of possible values for the string. For example, the `colorModeEnumType` in the (OGC) *KML* schema has possible values "normal" and "random". In many languages, we can refer to these values by a variable with the same name. The option *defineEnumVars* controls whether or not defineClasses() and its helper functions define such variables in *R*. The purpose of these is to catch errors caused by "typos". For instance, if we mistyped "normal" as "norma" (i.e., without the trailing "l"), the validity function will raise an error when it is called. However, if we have defined variables corresponding to these values, typing `norma` in our code will lead to *R* raising an error about the variable `norma` not being found. In short, we get a different type of error at a different point in the computations. The `variable not found` error is slightly easier to understand in many cases.

14.3 Reading *XML* Schema in *R*

In this section, we look more closely at the details of readSchema(). We start by discussing readSchema() and the structure of the information it returns. Then we discuss the classes of the *R* objects describing the different components of a schema, e.g., the descriptions of a `complexType`, a `sequence` element, and so on. Then we illustrate how we map those schema type descriptions to *R* class definitions and the details for defining the methods to read *XML* content and convert it using these different classes. The aim is to allow readers to understand what these functions do and what they generate. This helps us to know what to expect from the generated code. We also hope that others will be able to work with these objects to make further use of the schema information.

We use readSchema() to read an *XML* schema document. This function takes either an already parsed *XML* document (e.g., with xmlParse()) or it takes the name of a local file, a *URL*, or a string containing the *XML* content, and then parses that to get the parsed *XML* document to process. Since schema often appear in a *WSDL* document associated with a Web service, readSchema() can also be given the name of a *WSDL* file and it will extract the schema from it.

Having parsed and obtained the top-level <*schema*> node, readSchema() processes its contents and organizes the descriptions of *XML* elements and data types into *R* objects that represent that information. For example, we read the *PMML* schema with

```
pxsd = "pmml-4-1.xsd"
f = system.file("samples", pxsd, package = "XMLSchema")
pmml = readSchema(f)
```

The readSchema() function typically returns a `SchemaCollection`. This is a collection (*list*) used to represent one or more schema, not the contents of a single schema. This can be a little confusing at first. We called readSchema() with a single schema and can expect to get back a single schema. The reason for returning a collection of schema is reasonably straightforward. Many schema are entirely self-contained, defining all the data types internally within the schema or relying on the primitive data types defined by the general *XML* schema language, e.g., string, float. However, many schema are not self-contained but build on other schema. An *XML* schema can include and import other schema (using <*include*> and <*import*>, respectively), and these in turn can import/include other schema, and so on. This allows us to reuse sub-schema and share definitions of types and elements across *XML* grammars and formats. We can define new data types using those in other schema,

which is a good thing, but it does add some complexity. When we define a data type in one schema that refers to a data type in another schema, we need a way to find the definition of that second type. Furthermore, we need to be able to unambiguously identify that second type. The `SchemaCollection` class helps us to do this.

PMML is a stand-alone schema. As a result, readSchema() returns only one schema, but as the sole element within a `SchemaCollection` object. We are interested in that single element, e.g., `pmml[[1]]`. The Keyhole Markup Language schema developed and standardized by Google and the Open Geospatial Consortium (OGC) does build on two other schema. We can read that schema with

```
u = "http://schemas.opengis.net/kml/2.2.0/ogckml22.xsd"
kml = readSchema(u)
```

Again, the result is a `SchemaCollection`. However, this contains three elements because the top-level `ogckml22.xsd` schema document imports both the `atom-author-link.xsd` and the `http://docs.oasis-open.org/election/external/xAL.xsd` schema.

A `SchemaCollection` object is a list of `SchemaTypes` objects. The `SchemaTypes` class is used to contain the descriptions of the data types, elements, attributes and groups defined in the actual schema, e.g., *<complexType>*, *<element>*, *<simpleType>*, *<attributeGroup>* elements. Another way to think about this is that a schema can be hierarchical due to *<import>* or *<include>* nodes, and these elements can be at any level of the hierarchy. The readSchema() function flattens this hierarchy into a list, i.e., a `SchemaCollection`. Each element of a `SchemaCollection` represents a single schema and is of class `SchemaTypes`.

It is important to recognize that readSchema() returns descriptions of elements and data types with named *references* to these other elements and data types on which they are based. As a result, we need to be able to find those references later. This is why we maintain the list of different schema involved in the topmost schema. Why do we not just incorporate the definitions from these other schema directly? Because we want to be able to process each element globally and avoid multiple re-definitions. We can also merge them into a single list, but it is convenient to group them by their originating schema.

The elements of a `SchemaCollection` are named. We do not use the name of the document, but instead the target namespace associated with that particular schema. This namespace provides the "scope" in which the elements of the schema are defined, and allows us to distinguish between two types defined in different schema but with the same name. We use these namespaces for names within the `SchemaCollection` to enable resolving references to different types. The names for the *KML* schema are

```
names(kml)
```

```
[1] "http://www.opengis.net/kml/2.2"
[2] "http://www.w3.org/2005/Atom"
[3] "urn:oasis:names:tc:ciq:xsdschema:xAL:2.0"
```

For the *PMML* schema, we get

```
names(pmml)
```

```
[1] "http://www.dmg.org/PMML-4_1"
```

This naming scheme allows us to readily and uniquely identify a definition in another schema. For example, consider the schema content from the *KML* schema defining the `AbstractFeatureType` below.

```
<complexType name="AbstractFeatureType" abstract="true">
 <complexContent>
  <extension base="kml:AbstractObjectType">
   <sequence>
    <element ref="kml:name" minOccurs="0"/>  1
    <element ref="kml:visibility" minOccurs="0"/>
    <element ref="kml:open" minOccurs="0"/>
    <element ref="atom:author" minOccurs="0"/>  2
    <element ref="atom:link" minOccurs="0"/>
    <element ref="kml:address" minOccurs="0"/>
    <element ref="xal:AddressDetails" minOccurs="0"/>  3
    <element ref="kml:phoneNumber" minOccurs="0"/>
       ......
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

1 To resolve the reference `kml:name`, we find the local definition of the "kml" namespace prefix and find its URI. We then use this to index the `SchemaCollection`. In this case, the prefix refers to this same schema in which this type, `AbstractFeatureType`, is defined.

2 The sequence includes an element for author and this references the definition `atom:author`. As with `kml:name`, we find the local definition of the `atom` namespace prefix, find its URI, and use it to index the `SchemaCollection` to obtain the correct definition of author, even if another schema in our collection has an element or type named `author`.

3 Similarly, we find a description of the *<AddressDetails>* node within the `SchemaCollection` by mapping the prefix `xal` to the namespace URI `urn:oasis:names:tc:ciq:xsdschema:xAL:2.0`.

The descriptions of individual elements and data types defined in a schema may refer to data types and elements in that schema or in other schemas that are part of the overall collection of schemas. We use namespaces to resolve these references across the schema unambiguously. Much of this can be done simply using pointers, but since we do not have these in *R*, we delay resolving references to other elements and types until they are needed and we have read the entire collection of schemas. In this way, readSchema() creates "lazy" and partially complete descriptions of the types in order to avoid forward references and having to resolve references immediately. This is enough when we want to explore a schema at its top level without looking at the subelements of any particular type, e.g., looking at what elements are defined and the inheritance hierarchy. However, it is awkward to deal with references rather than actual definitions in place. We have to resolve each reference via the namespace URI and name. Implicitly, functions such as defineClasses() and defClass() look-up the references via the resolve() function and create locally complete descriptions of the data types and elements before processing the actual type. This is done after the call to readSchema() in defineClasses().

Example 14-3 Exploring the KML Schema via the `SchemaCollection` Class

Once we have the `SchemaCollection` and its `SchemaTypes` elements, we can examine the contents of the schema. We can see how many top-level descriptions there are in each of the *KML*-related schemas with

```
sapply(kml, length)
```

```
          http://www.opengis.net/kml/2.2
                                      357
             http://www.w3.org/2005/Atom
                                        9
urn:oasis:names:tc:ciq:xsdschema:xAL:2.0
                                       33
```

The Atom schema defines only nine types. The first few are named

```
head(names(kml[[2]]))
```

```
[1] "atomPersonConstruct" "name"                "uri"
[4] "email"               "author"              "link"
```

These provide definitions for different pieces of contact information. We can also look at the names for the other schema and explore what types they define, but there are too many to include here.

Note that we have seen that a type called `name` is defined in both the Atom and the *KML* schema (in the definition of the `AbstractFeatureType` type). Since they have unique namespaces, we can determine which one is meant in each context they are referenced. Note also that each reference to a schema type has the actual namespace URI identifying that namespace, and not just the namespace prefix. When readSchema() encounters a reference such as `atom:name`, it maps the prefix `atom` to the corresponding namespace's URI and this allows us to resolve the reference later within the `SchemaCollection` correctly and unambiguously.

Let's look at the different classes of *R* objects we have describing the different schema components. We can do this across all three schema by flattening the list of `SchemaTypes` into a single container of all their elements:

```
table(sapply(unlist(kml, recursive = FALSE), class))
```

```
                    AttributeGroup                     ClassDefinition
                                 2                                  22
                           Element              ExtendedClassDefinition
                               230                                  55
                  RestrictedDouble                  RestrictedHexBinary
                                 5                                   1
                RestrictedListType            RestrictedStringDefinition
                                 2                                  11
RestrictedStringPatternDefinition                      SchemaVoidType
                                 3                                  60
                     SimpleElement                    SimpleSequenceType
                                 2                                   5
                   UnionDefinition
                                 1
```

These classes are the ones we will work with.

The classes listed in this example are the core classes used to describe components in all schema. Each corresponds to a different type within a schema. The purpose of having these different classes in *R* is to allow us to work more easily with generic schema types by mapping them to more *R*-based concepts. There is a short description of each of these classes in Table 14.1.

Table 14.1: Primary *R* Classes for *XML* Schema Types

***R* class name**	**Purpose**
ClassDefinition	Defines a data type in the *XML* schema corresponding to a new class in *R*. The data type often corresponds to a `<complexType>` element in the schema and has slots including the name of the type, a count for how often it occurs (optional minimum and maximum), and the types of its elements.
ExtendedClass-Definition	Describes a data type in the *XML* schema that extends another schema data type.
RestrictedDouble	Restricted classes refer to a situation when we have a regular type that has restrictions on the possible value(s) it contains. A RestrictedDouble describes a real-valued number type with a limitation on the range of the values, e.g., a positive real value or a value between 0 and 1.
RestrictedHexBinary	Describes a hexadecimal value with some restrictions such as the number of components, i.e., its length. The *KML* schema defines a color as a restricted hexadecimal with four components giving the alpha, blue, green, and red levels, in that order, as values between 00 and FF.
RestrictedListType	Describes a restricted list, which is a regular list with a constraint on the type(s) of its elements. For example, we may have a list of only strings, or of only Definition objects.
RestrictedString-Definition	Describes a restricted string. Typically this describes a string that comes from a finite set of possible values, such as an enumerated constant in other languages.
RestrictedString-PatternDefinition	Describes a string data type where the contents must match a particular pattern, i.e., a regular expression.
Element	Describes an *XML* element definition in the schema, consisting of a name, attribute definitions, and possible children given by its *type* field.
SimpleElement	Corresponds to an *XML* element (rather than an explicit data type) that contains only attributes and optionally text, but no subchildren. The `<Level>`, `<Constant>`, and `<MatCell>` elements in the *PMML* schema are examples of this.
SimpleSequenceType	Corresponds to a collection of items. This is used for `<sequence>` elements in the schema, or elements or data types that can occur more than once in a particular context, i.e., that have a *maxOccurs* attribute with a value greater than 1.
UnionDefinition	Describes a type that contains exactly one of a possible set of types.
SchemaVoidType	Corresponds to the empty data type, such as to a `void` in *C* and perhaps a `NULL` in *R*. It occurs, for example, when we have an element with no type or content, e.g., `<xs:element name="ARIMA" />` in the *PMML* schema.

This lists the different R classes that are generated via readSchema(). These describe XML element and data type definitions in XML schema.

14.4 *R* Classes for Describing *XML* Schema Types

In this section, we look at the descriptions created by readSchema() of the different types of elements in a schema. We look at the *XML* in the schema and see how these map to different classes described in Table 14.1. We use examples from both *PMML* and *KML*.

Example 14-4 A Description of PMML Schema Elements in R

The current *PMML* schema is available at `http://www.dmg.org/v4-1/pmml-4-1.xsd`. We can read it into *R* with

```
pmml = readSchema("http://www.dmg.org/v4-1/pmml-4-1.xsd")
```

The result is a `SchemaCollection` that contains a single schema. For simplicity and since we are not resolving types here but merely looking at the classes produced by readSchema(), we work directly with the `SchemaTypes` object, i.e.,

```
pmml = pmml[[1]]
```

There are 287 elements in the schema with the following class distribution:

```
table(sapply(pmml, class))
```

```
        ClassDefinition                 Element
                      2                     222
ExtendedClassDefinition RestrictedStringDefinition
                     10                      31
          SimpleElement           SchemaGroupType
                      5                      14
         SchemaVoidType
                      3
```

There are twelve explicit data types

```
names(pmml)[sapply(pmml, is, "ClassDefinition")]
```

```
 [1] "VECTOR-ID"     "NN-NEURON-ID"  "NN-NEURON-IDREF"
 [4] "ELEMENT-ID"    "NUMBER"        "INT-NUMBER"
 [7] "REAL-NUMBER"   "PROB-NUMBER"   "PERCENTAGE-NUMBER"
[10] "FIELD-NAME"    "ArrayType"     "COUNT-TABLE-TYPE"
```

and ten of these extend other data types. The 31 `RestrictedStringDefinition` objects correspond to enumerated constants, i.e., a small subset of specific strings. For example, the COMPARE-FUNCTION type is defined as

```
<xs:simpleType name="COMPARE-FUNCTION">
    <xs:restriction base="xs:string">
        <xs:enumeration value="absDiff" />
        <xs:enumeration value="gaussSim" />
        <xs:enumeration value="delta" />
        <xs:enumeration value="equal" />
        <xs:enumeration value="table" />
    </xs:restriction>
</xs:simpleType>
```

The COMPARE-FUNCTION is a <*simpleType*> and so corresponds to an *XML* element that has only text as content, and no *XML* elements as children. The <*restriction*> indicates that the content is a string and that it must be one of the values in the listed <*enumeration*> elements. The class of the descripition in *R*, i.e., `pmml[["COMPARE-FUNCTION"]]`, is `RestrictedStringDefinition`. This has the name of the type, COMPARE-FUNCTION, and also the set of possible values:

```
pmml[["COMPARE-FUNCTION"]]@values
```

```
 "absDiff" "gaussSim"   "delta"   "equal"   "table"
```

In addition to the `RestrictedStringDefinition` class, there are many other restricted types. We describe several of them here and provide examples.

`RestrictedStringPatternDefinition`

This type is a restriction on strings by specifying a pattern. For instance, we might define a postal ZIP code type as

```
<xs:simpleType name="ZIP">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{5}"/>
  </xs:restriction>
</xs:simpleType>
```

The readSchema() function uses the class `RestrictedStringPatternDefinition` to represent the schema description for this type of item.

`RestrictedDouble`

Another type is a restriction on the range of a number. For example, the *KML* schema includes a definition for an angle between 0 and 90 degrees. This is defined as

```
<simpleType name="anglepos90">
 <restriction base="double">
  <minInclusive value="0.0"/>
  <maxInclusive value="90.0"/>
 </restriction>
</simpleType>
```

The readSchema() function describes this with an instance of the `RestrictedDouble` class and stores within it the minimum and maximum and whether these are inclusive or not.

There are also classes for restrictions for integers and hexadecimal representations. For example, the *KML* schema defines the type color as a hexadecimal value that consists of exactly four elements, e.g., `ff000000`. This consists of the values of the alpha-level and blue, green, and red levels, in that order.

`RestrictedListType`

We have also introduced the `RestrictedListType` to represent a collection of elements that have to be of a specified type. For example, again in the *KML* schema, the definition of `CoordinatesType` is given as

```
<simpleType name="CoordinatesType">
 <list itemType="string"/>
</simpleType>
```

This indicates a list for which each element is a string and we can represent this via `RestrictedListType` with a *type* slot that identifies the *XML* schema string type.

SimpleSequenceType

The `RestrictedListType` is very similar to the `SimpleSequenceType` class. This corresponds to a schema type defined using the `<sequence>` element. For example, in the Atom schema for author information (included in the ogckml22 schema), we have the definition

```
<complexType name="atomPersonConstruct">
 <choice minOccurs="0" maxOccurs="unbounded">
  <element ref="atom:name"/>
  <element ref="atom:uri"/>
  <element ref="atom:email"/>
 </choice>
</complexType>
```

This defines a type named `atomPersonConstruct` that has 0 or more elements that can be from the set "name", "uri" or "email". The elements can be mixed rather than having to be all of one element type. So `SimpleSequenceType` is a little more flexible than `RestrictedListType`, which is defined to have a single type of element.

SimpleElement

A `SimpleElement` is an *XML* element (rather than just an explicit data type within the schema) that only contains attributes and optionally text, but no subchildren. The `<Level>`, `<Constant>`, and `<MatCell>` elements defined in the *PMML* schema are examples of this. For example, `<MatCell>` is defined as

```
<xs:element name="MatCell">
 <xs:complexType>
  <xs:simpleContent>
   <xs:extension base="xs:string">
    <xs:attribute use="required" type="INT-NUMBER" name="row" />
    <xs:attribute use="required" type="INT-NUMBER" name="col" />
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

This means that the *XML* element `<MatCell>` must contain a string and it must have *row* and *col* attributes that are integer values. (This is defined in the schema, but it does not say whether the value corresponds to 0- or 1-based indexing.)

Element

The `Element` types correspond to definitions of *XML* elements. These are actual elements we will find in an *XML* document. These differ from `<complexType>` and other types defined in the schema in that those are descriptions of structures, but not of *XML* node names that appear in *XML* documents. An element definition in a schema corresponds to a node name and also a data type. An element acts as an alias or concrete name for a data type. The `<GaussianDistribution>` element is an `Element` object. It appears in the schema as

```
<xs:element name="GaussianDistribution">
 <xs:complexType>
  <xs:sequence>
```

```
   <xs:element maxOccurs="unbounded" ref="Extension"
       minOccurs="0" />
  </xs:sequence>
  <xs:attribute use="required" type="REAL-NUMBER" name="mean"/>
  <xs:attribute use="required" type="REAL-NUMBER" name="variance"/>
 </xs:complexType>
</xs:element>
```

Note that the body defines a <*complexType*> and this corresponds to a `ClassDefinition` at the top level of the schema. A <*GaussianDistribution*> element must have both a *mean* and a *variance* attribute that are numbers. It can also contain (as child elements) zero or more <*Extension*> elements. When we convert an *XML* node corresponding to an `Element`, we will map it to the corresponding data type defined for that `Element`.

ExtendedClassDefinition

The AbstractFeatureType defined in *KML* is an example of an `ExtendedClassDefinition`. This is very similar to a `ClassDefinition` in that it has a name of a new type with its own components (elements and attributes). However, it extends an existing type and inherits that types components. As an example, `AbstractFeatureType` extends the `AbstractObjectType` as its "base" class, as specified by the *XML*

```
<complexType name="AbstractFeatureType" abstract="true">
 <complexContent>
  <extension base="kml:AbstractObjectType">
      ....
```

UnionDefinition

The `UnionDefinition` class is used to represent an *XML* schema that can be one of two or more types. These can arise from two different types of schema definitions. One involves the <*union*> element. For example, the *KML* schema defines a type named `dateTime` as

```
<union memberTypes="dateTime date gYearMonth gYear"/>
```

The result is a `UnionDefinition` where the value is one of these four types listed in the *memberTypes* attribute.

`UnionDefinition` objects also arise from the <*choice*> schema element. A <*choice*> element means that there can be any one of the types/elements contained in <*choice*>. As an example, consider the definition of the data type `BalloonStyleType` in the *KML* schema. This includes the *XML*

```
<choice>
 <annotation>
  <documentation>color deprecated in 2.1</documentation>
 </annotation>
 <element ref="kml:color" minOccurs="0"/>
 <element ref="kml:bgColor" minOccurs="0"/>
</choice>
```

This indicates that there will be one of the two elements <*kml:color*> and <*kml:bgColor*>. Since both have a *minOccurs* with a value of 0, it is possible in this case that neither element will appear. However, we definitely cannot have both elements appear, or any other element.

`UnionDefinition` objects from *XML* schema are analogous to unions in *C* or `ClassUnions` in *R*, both of which allow a value from one of the several classes to be used as the value of the union type.

As an example, XMCDA is an *XML* grammar for representing MultiCriteria Decision Analysis (MCDA) [7] data elements. This standard was developed by Decision Deck (`http://www.decision-deck.org/`), which is an open source project to support MCDA techniques. The `RXMCDA` package [8] allows you to transform XMCDA documents into *R* objects for use with MCDA algorithms and to reciprocally create XMCDA documents from *R*. The XMCDA schema `XMCDA-2.0.0.xsd` defines the type `criterionReference` as

```
<xs:complexType name="criterionReference">
 <xs:choice>
  <xs:element name="criterionID" type="xs:string"/>
  <xs:element name="criteriaSetID" type="xs:string"/>
  <xs:element name="criteriaSet" type="xmcda:criteriaSet"/>
 </xs:choice>
</xs:complexType>
```

This describes a type named criterionReference and indicates that it should contain one child element. That child must be a *<criterionID>*, *<criteriaSetID>*, or *<criteriaSet>* element.

`SchemaGroupType`

The `SchemaGroupType` class corresponds to a *<group>* element in the *XML* schema. These act as containers for one or more *XML* elements. The group can be referenced by name in defining other elements or complex types and the contents inlined in that definition. As a result, we do not necessarily need to map these to *R* data types. We can create them as intermediate classes, or we can just inline the definitions of the elements in each case they are used.

14.5 Mapping Schema Type Descriptions to *R* Classes and Converter Methods

In this section, we again look at specific *XML* schema examples, but here we illustrate how defineClasses() and its helper defClass() map these descriptions to actual *R* class definitions and generate the *R* functions to convert an *XML* node or document to instances of those classes. This is not an exhaustive coverage of all of the schema types and how they are mapped. However, we discuss the important elements in the design. This information will help readers understand what is being generated by defineClasses() and how the converter functions behave. It provides the reasoning behind these decisions and hopefully provides advanced readers with a starting point for defining alternative mapping schemes.

14.5.1 Mapping Simple Elements to R Types

Consider the following type in the *PMML* schema.

```
<xs:element name="Constant">
 <xs:complexType>
  <xs:simpleContent>
   <xs:extension base="xs:string">
```

```
    <xs:attribute type="DATATYPE" name="dataType" />
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

This *XML* type is used to represent a constant value in a *PMML* document, i.e., a value that does not change and is not a variable in some model. For this, we need both the value and also its data type. This might appear in *XML* as

```
<Constant dataType="float">3.141593</Constant>
```

The schema defines this as an *XML* element named "Constant". Its content consists of a `<complexType>` node and a `<simpleContent>` node within it. This basically means we have an element that has no *XML* child elements, but simply text, and that it has one or more attributes. We can map this element to *R* as a class that extends the *R* class `string` (which is a character vector with one element) to represent the text content within the *XML* element. The new class also has a slot corresponding to the *dataType* attribute. We can define this as

```
setClass("Constant",
         representation("string", dataType = "DATATYPE"))
```

Note that the slot *dataType* is defined to be an instance of the class `DATATYPE`. This is actually defined in the *PMML* schema as

```
<xs:simpleType name="DATATYPE">
 <xs:restriction base="xs:string">
  <xs:enumeration value="string" />
  <xs:enumeration value="integer" />
  <xs:enumeration value="float" />
  <xs:enumeration value="double" />
  <xs:enumeration value="boolean" />
  <xs:enumeration value="date" />
  <xs:enumeration value="time" />
  <xs:enumeration value="dateTime" />
  <xs:enumeration value="dateDaysSince[0]" />
  <xs:enumeration value="dateDaysSince[1960]" />
  <xs:enumeration value="dateDaysSince[1970]" />
  <xs:enumeration value="dateDaysSince[1980]" />
  <xs:enumeration value="timeSeconds" />
  <xs:enumeration value="dateTimeSecondsSince[0]" />
  <xs:enumeration value="dateTimeSecondsSince[1960]" />
  <xs:enumeration value="dateTimeSecondsSince[1970]" />
  <xs:enumeration value="dateTimeSecondsSince[1980]" />
 </xs:restriction>
</xs:simpleType>
```

This defines a type that is a string restricted to being one of these possible values listed in the `<enumeration>` elements. Before we define the class `Constant`, defClass() ensures that the *R* class for `DATATYPE` was defined. This is described by readSchema() as a `RestrictedStringDefinition` object, which we saw in Section 14.4. The defClass() function would map this new class to a simple extension of the `string` class with

```
setClass("DATATYPE", contains = "string")
```

However, we cannot allow instances of `DATATYPE` to be arbitrary strings, so along with the simple definition, we ensure that the value of such an object is one of the enumerated values. We do this by defining a validity method that *R* uses to check the contents of the object. This looks something like

```
setValidity("DATATYPE",
  function (object) {
    values = c("string", "integer", "float", "double", "boolean",
               "date", "time", "dateTime", "dateDaysSince[0]",
               "dateDaysSince[1960]",  "dateDaysSince[1970]",
               "dateDaysSince[1980]",  "timeSeconds",
               "dateTimeSecondsSince[0]",
               "dateTimeSecondsSince[1960]",
               "dateTimeSecondsSince[1970]",
               "dateTimeSecondsSince[1980]")

    if (any(i <- !(object %in% values)) )
      paste( paste(object[i], collapse = ","",
              " is not a recognized value ",
              paste(sQuote(values), collapse = ", ")))
    else TRUE
  })
```

This function is associated with the class either in the call to setClass() or via setValidity().

When we use defClass() (directly or via the defineClasses() function), any classes referenced in the type being defined will be defined before the target class is defined. This include base types for `ExtendedClassDefinitions` and any class corresponding to slots in the target class. These dependent classes are defined only once. When we need to define them, we also define any of their dependent classes, and so on, recursively. Given this sequence, the `DATATYPE` class would have been defined before the `Constant` class that required it.

When defining the `Constant` class, we also define a method for coercing an *XML* node to an instance of the class. It can be defined as

```
setAs("XMLAbstractNode", "Constant",
      function(from)
        new("Constant", xmlValue(from),
            dataType = xmlGetAttr(from, "dataType")))
```

When we encounter a <*Constant*> element in an *XML* document, this creates a new `Constant` object in *R* and fills in its string and *dataType* values. The node

```
<Constant dataType="integer">3</Constant>
```

yields

```
new("Constant", "3", dataType = "integer")
```

It is reasonably clear that we can use a more appropriate and context-specific conversion from a <*Constant*> node to an actual value in *R* in our example. For instance, we can map the string `"3"` to the integer 3L. We can define a function that examines the *dataType* attribute and performs the relevant conversion of the value in the <*Constant*> element to the corresponding *R* type. Then we can override the default coercion method created by defineClasses() with this alternative function. This would look something like

```
setAs("XMLAbstractNode", "Constant",
      function(from) {
         val = xmlValue(from)
         switch(xmlGetAttr(from, "dataType"),
                integer = as.integer(val),
                float = ,
                double = as.numeric(val),
                boolean = as.logical(val),
                date = as.Date(val),
                # handlers for other types ...
               )})
```

Note that the function does not return an instance of the class `Constant`, which may cause problems. Rather than using this as a method for coerce()/as(), we may want to use the function as a method for the fromXML() function instead, as this provides no expectations on the class of the return value. To do this, we also want to explicitly set this in a map of schema node names to *R* types.

14.5.2 Class Inheritance in R for Schema Derived Types

Consider the <*LatLonBoxType*> element in the *KML* schema. This is defined as

```
<complexType name="LatLonBoxType">
 <complexContent>
  <extension base="kml:ObjectType">
   <sequence>
    <annotation>
     <documentation>
      Yes, north/south range to 180/-180
     </documentation>
    </annotation>
    <element name="north" type="kml:angle180"
      minOccurs="0" default="180.0"/>
    <element name="south" type="kml:angle180"
      minOccurs="0" default="-180.0"/>
    <element name="east" type="kml:angle180"
      minOccurs="0" default="180.0"/>
    <element name="west" type="kml:angle180"
      minOccurs="0" default="-180.0"/>
    <element name="rotation" type="kml:angle180" minOccurs="0"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

This schema element defines a data type (not an element) named "LatLonBoxType" which extends the type "ObjectType" defined earlier in the schema. When we use readSchema() to process the schema document, the LatLonBoxType element corresponds to an object of class `ExtendedClassDefinition`, and the description of the ObjectType coming from read-

Schema() is a `ClassDefinition`. In addition to the slots `ExtendedClassDefinition` inherits from `ObjectType` (*id* and *targetId*, which we will see later), the definition introduces new slots for the class, identified by the *XML* elements named north, south, ..., rotation. Each of these are optional since the *minOccurs* attribute is set to 0. We can have either 0 or 1 of each of these elements in the parent *XML* node. Each of these elements is of type "angle180". This is defined elsewhere in the schema and we look at that below. For now, we can assume that defClass() has defined an *R* class corresponding to `angle180`. What will the *R* class definition for `LatLonBoxType` look like based on this schema description? A natural mapping is described by

```
setClass("LatLonBoxType", contains = "ObjectType",
         representation(north = "angle180", south = "angle180",
                        east = "angle180", west = "angle180",
                        rotation = "angle180" ),
         prototype = list(north = new("angle180", 180),
                          south = new("angle180", -180),
                          east = new("angle180", 180),
                          west = new("angle180", -180)))
```

We do not include the slots inherited from `ObjectType`. Instead, we use *R*'s inheritance to do this for us in a natural manner via the *contains* parameter for setClass().

For the `LatLonBoxType` class, we have provided a prototype. This is because some of the elements within the schema description provide a default value via their *default XML* attribute, e.g., 180 and −180. These are provided as strings in the schema, but we have converted these default values to the corresponding types in *R* as required by the type of each slot. Note also that in the *XML* above, there is no default value for *rotation* so we do not provide a value for it in the prototype.[2]

Note that in defining `LatLonBoxType`, there was no need for a validity method. The *S4* class mechanism ensures we can only assign objects of class `angle180` to each of the slots. There is a validity method on the `angle180` class to verify that the value is within the correct range. This way, there is no need for an explicit validation method on `LatLonBoxType`, just on the classes of each of its slots.

Now we look at how we might implement the converter from an *XML* node to an instance of `LatLonBoxType`. What should the converter do? Firstly, it should create a new instance of the `LatLonBoxType`. Then we can process each of the possible elements in the *XML* (i.e., <*north*>, <*south*>, ...) and convert them to the appropriate type, i.e., `angle180`. We also must process the content inherited from the base `ObjectType`. We can find out what those elements and attributes are, and then process them too. However, it is easier and better to simply inherit the coercion method from that class. We start by converting the *XML* node to that base class and then convert the result to an instance of the `LatLonBoxType`, e.g.,

```
.obj = as(from, "ObjectType")
.obj = as(.obj, "LatLonBoxType")
```

After this, we can process each of the possible elements and assign the values to the slots of the new `LatLonBoxType` object. Since these are all optional, we need to test whether each one is present and then convert it, if it is. For example, for <*north*>, we would have code

```
if(!is.null(tmp <- from[["north"]]))
   .obj@north = as(tmp, "angle180")
```

[2] In the actual *KML* schema, there is a default value for the rotation. We removed it here to illustrate how the lack of a default value maps to the *R* prototype.

This code goes into the body of the converter function for `LatLonBoxType`. It retrieves the child node named `<north>` in the `<LatLonBoxType>` node and, if it is not `NULL`, calls the method to convert that node to an instance of the `angle180` class. This dependent class and its coercion method from an *XML* node will be generated as part of defining `LatLonBoxType`, if it does not already exist.

The complete converter function is given by

```
f = function(from)
{
  .obj = as(from, "ObjectType")
  .obj = as(.obj, "LatLonBoxType")

  if(!is.null(from[["north"]]))
    .obj@north = as(from[["north"]], "angle180")
  if(!is.null(from[["south"]]))
    .obj@south = as(from[["south"]], "angle180")
  if(!is.null(from[["east"]]))
    .obj@east = as(from[["east"]], "angle180")
  if(!is.null(from[["west"]]))
    .obj@west = as(from[["west"]], "angle180")
  if(!is.null(from[["rotation"]]))
    .obj@rotation = as(from[["rotation"]], "angle180")
  .obj
}
```

Let's look at the class `angle180`. This class will be defined by defClass() when it is needed to specify the class of the slots for the `LatLonBoxType`. The `angle180` class corresponds to the schema description

```
<simpleType name="angle180">
 <restriction base="double">
  <minInclusive value="-180"/>
  <maxInclusive value="180"/>
 </restriction>
</simpleType>
```

This schema merely indicates that `angle180` is a number and it should be between -180 and 180, inclusive. We can define the corresponding class in *R* with

```
setClass("angle180", contains = "numeric",
         validity = function(object) {
           if(!all(object >= -180 & object <= 180))
             "some values are outside of [-180, 180]"
           else
             TRUE
           })
```

This class is a simple extension of the `numeric` vector class, but we add a validity method that will catch erroneous values.

The function to convert an *XML* node to this type is also quite simple. We can define it as

```
function(from) new("angle180", xmlValue(from))
```

A value outside of $[-180, 180]$ will raise an error in the constructor function because of the validity method. Again, this converter function will be created automatically in a call to defineClasses() or if this is one of the dependent classes needed in a call to defClass().

Now we look at the base class `ObectType` that `LatLonBoxType` extends. In the schema, the data type named `ObjectType` is defined as

```
<complexType name="ObjectType" abstract="true">
 <attributeGroup ref="kml:idAttributes"/>
</complexType>
```

and the `idAttributes` group is defined as

```
<attributeGroup name="idAttributes">
 <attribute name="id" type="ID" use="optional"/>
 <attribute name="targetId" type="NCName" use="optional"/>
</attributeGroup>
```

Basically, we see that our `ObjectType` class should have two slots corresponding to "id" and "targetId". The types ID and NCName are primitive/builtin types for an *XML* schema. An ID is a string that acts as a unique identifier within the entire *XML* document, i.e., each *id* attribute must be unique. NCName is any valid *XML* node name.

The corresponding *S4* class definition should be

```
setClass("ObjectType",
         representation(id = "ID", targetId = "NCName"),
         contains = "VIRTUAL")
```

The schema for ObjectType indicates that this is an abstract type. This means that we cannot create instances of this directly and so we might use this concept in *R* by extending the "VIRTUAL" class. Unfortunately, this causes grief in our converter function for `LatLonBoxType` above since we have

```
.obj = as(from, "ObjectType")
```

The converter must return an `ObjectType`, and create an instance of the would-be virtual class. We can change this converter to create an instance of `LatLonBoxType` and then fill in the slots inherited from `ObjectType`. This would work, but change how we develop the converter function. Instead, we currently ignore the abstract attribute of a type in the schema and use the coercion approach that creates an instance of the base class. Therefore we define `ObjectType` with

```
setClass("ObjectType",
         representation(id = "ID", targetId = "NCName"))
```

and omit the "VIRTUAL" base class.

We define the conversion from *XML* to `ObjectType` as

```
setAs("XMLAbstractNode", "ObjectType",
      function(from) {
        obj = new("ObjectType")
        obj@id = new("ID", xmlGetAttr(from, "id", character()))
        obj@targetId = new("NCName", xmlGetAttr(from, "targetId",
                                                character()))
        obj
      })
```

At this point, we have defined all the pieces for the `ExtendedClassDefinition` description of `LatLonBoxType`, including the base class and the `angle180` class for the slots.

14.5.3 Collections, Lists, and Recurring Elements

Let's look at a different but quite simple schema. This is `egquery.xsd` which describes the inputs and outputs for a query of the National Center for Biotechnology Information's (NCBI) Entrez cross-database search engine [10]. This Web service allows you to search multiple life sciences databases for a term and get information about the number of matches in each database. The schema describes data types and elements for both the database query and also the result. We focus on the result types. An example of the result is

```
<?xml version="1.0"?>
<Result xmlns="http://www.ncbi.nlm.nih.gov/soap/eutils/egquery">
 <Term>stem cells</Term>
 <eGQueryResult>
  <ResultItem>
   <DbName>pubmed</DbName>
   <MenuName>PubMed</MenuName>
   <Count>166703</Count>
   <Status>Ok</Status>
  </ResultItem>
  <ResultItem>
   <DbName>pmc</DbName>
   <MenuName>PubMed Central</MenuName>
   <Count>52733</Count>
   <Status>Ok</Status>
  </ResultItem>
 </eGQueryResult>
</Result>
```

The top-level node is *<Result>* and it has an *<Term>* node and a *<eGQueryResult>* node as its immediate children. The *<Term>* node gives the query string that led to the result. The *<eGQueryResult>* node contains one or more *<ResultItem>* elements. Each *<ResultItem>* represents the query results for one of the different databases that was searched and had a match. These *<ResultItem>* nodes each have *<DbName>*, *<MenuName>*, *<Count>*, and *<Status>* nodes. The content of each of these is a simple string. The relevant part of the *XML* schema that describes this structure appears in Figure 14.1

Given the basic structure of the *<Result>* node, let's think about how we map this to an *R* class. We start with the *<ResultItemType>* element, as this is how the defClass() function operates, working with the dependencies first. *<ResultItemType>* corresponds to a `ClassDefinition` object in our `SchemaTypes` object. *<ResultItemType>* has four elements (*<DbName>*, *<MenuName>*, *<Count>*, and *<Status>*), each of which has a *type* attribute identifying it as a string. This type corresponds to a class in *R* that has a slot for each of these four elements, each of which is of type `string`. We define this with the class

```
setClass("ResultItemType",
         representation(DbName = "string",MenuName = "string",
                        Count = "string", Status = "string"))
```

The most interesting part of the schema for this example is the `eGQueryResultType` definition. This is a *<sequence>* element and is represented in *R* by readSchema() as a `SimpleSequenceType`. The *<ERROR>* element is optional as it has an attribute *minOccurs* with

```
<xs:element name="Result">  1
 <xs:complexType>
  <xs:sequence>  2
   <xs:element name="Term" type="xs:string"/>  3
   <xs:element name="eGQueryResult"
          type="tns:eGQueryResultType"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:complexType name="ResultItemType">  4
 <xs:sequence>
  <xs:element name="DbName" type="xs:string"/>
  <xs:element name="MenuName" type="xs:string"/>
  <xs:element name="Count" type="xs:string"/>
  <xs:element name="Status" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="eGQueryResultType">  5
 <xs:sequence>
  <xs:element name="ERROR" type="xs:string" minOccurs="0"/>
  <xs:element name="ResultItem" type="tns:ResultItemType"
         maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>
```

1 <*Result*> is the top-level node. It is a complex type, so can have child nodes and attributes.
2 The <*complexType*> node and its <*sequence*> children indicate that a <*Result*> node will have both a <*Term*> and a <*eGQueryResult*> element.
3 The <*Term*> element is defined locally within the <*Result*> element. The <*Term*> element will have simple text content.
4 The `ResultItemType` data type consists of four elements, <*DbName*>,<*MenuName*>, <*Count*>, and, <*Status*>. All of these elements have string content.
5 The <*eGQueryResult*> element in <*Result*> corresponds to an `eGQueryResultType` data type in the schema. This is a type that consists of an optional <*ERROR*> element describing a problem with the query, and 0 or more <*ResultItem*> nodes, one for each database in the result. Each of these <*ResultItem*> elements is described by the `ResultItemType` component in the schema.

Figure 14.1: Schema for the <*Result*> Node in EGQuery. This portion of the `egquery.xsd` schema describes the format for the result returned from the NCBI cross-database query. It defines three components in the output—the <*Result*> element, and the data types `ResultItemType` and `eGQueryResultType`.

a value of 0. The <*ResultItem*> element has a *maxOccurs* attribute with a value "unbounded". This means that there is potentially an "infinite" number of these elements within the result, which really corresponds to an arbitrary number of databases queried during the search. Each of these is an instance of our `ResultItemType` class that we defined previously. How do we represent the `eGQueryResultType` class? There are two different possibilities. One approach is to have a slot for the optional <*ERROR*> element and another for the collection of <*ResultItem*> elements. The former slot is a string; the latter is a *list* since there can be any number of `ResultItemType` elements. This *list* should verify that all items are of class `ResultItemType`, so we define a new class for this type of list named `ListOfResultItemType`. This extends the regular *list* class, but ensures that all its elements are of the appropriate class. It does this with a validity method. We can define the class as

```
setClass("ListOfResultItemType", contains = "list",
         validity =
             function (object) {
                if (!all(sapply(object, is, "ResultItemType")))
                   stop("not all elements are of type ",
                        "ResultItemType")
                else if (length(object) < 1)
                   "too few elements"
                else TRUE
             })
```

The validity method tests that all of the elements are of class `ResultItemType` and then ensures that there is at least one item. Since this is a type we consume rather than generate, we may not need to verify the number of items but can expect that the provider ensures this. However, the test for the length does not cost much in computing time.

The function to convert an *XML* node to a list of this type is quite simple. We loop over each of the *XML* elements and convert it and put the results in the appropriate element of the list. For our `ListOfResultItem` class, we can implement this as

```
setAs("XMLAbstractNode", "ListOfResultItemType",
      function(from)
         new("ListOfResultItem",
               xmlApply(from, as, "ResultItemType")))
```

Note that this function call explicitly converts each element to the class `ResultItemType`. If the elements in the *XML* are not homogeneous but of different types that are compatible with `ResultItemType`, i.e., derived from that base type, we will potentially end up with an incorrect and incomplete conversion because we map the *XML* nodes to the base class and not the complete class, thus discarding data. For that reason, it is best to use the fromXML() function to convert each element, e.g.,

```
new("ListOfResultItem", xmlApply(from, fromXML))
```

This examines the *XML* node and attempts to figure out the most appropriate *R* class to coerce it to.

With the `ListOfResultItemType` class defined, we can define `eGQueryResultType` as

```
setClass("eGQueryResultType",
         representation(ERROR = "string",
                        ResultItem = "ListOfResultItemType"))
```

Given that the class is defined along with the converter functions for this class and the `ListOfResultItemType` class, we can convert the *<Result>* *XML* document above easily with

```
e = as(xmlRoot(doc), "Result")
```

We talk about creating these converter functions below.

The second approach to defining the `eGQueryResultType` class is to, again, have a slot corresponding to the *<ERROR>* element. However, rather than *having* a slot for the collection of `ResultItemType` objects, we can define the class to *extend* a *list* class. We can again use the `ListOfResultItem`, this time as our base class, e.g.,

```
setClass("eGQueryResultType",
          representation("ListOfResultItem", ERROR = "string"))
```

This allows us to avoid having to access the `ResultItemType` via the extra slot reference, e.g., `obj[[2]]` rather than `obj@ResultItem[[2]]`.

One problem with using the *list* as a base class is when the *XML* schema description has two or more components with an unbounded number of elements. If we extend a *list* class, which of these sequences of elements should we use as the base for the list? Each of the other collections would have its own separate slot. The choice of which one to use as the base is somewhat arbitrary and the approach of extending a *list* class is only really effective when there is a single unbounded component. The approach of always using a slot for each sequence provides a little more consistency for users at the expense of making some situations (e.g., when a single slot is a *list*) slightly more complex.

14.5.3.1 Collections of Simple Types

Before we leave the concept of *XML* schema types that map to vectors or lists in *R*, we look at another example. Consider the `CoordinatesType` data type defined in the *KML* schema as

```
<element name="coordinates" type="kml:CoordinatesType"/>
<simpleType name="CoordinatesType">
  <list itemType="string"/>
</simpleType>
```

We see a data type named `CoordinatesType` which consists of a list of string elements. A *<list>* node in an *XML* schema means that the "elements" of the *<list>* in a document are text values, separated by white space. An example of the *<coordinates>* element is

```
<coordinates>-122.082,37.422,0 -121.156,38.193,20</coordinates>
```

Here the two elements of the list are `"-122.082,37.422,0"` and `"-121.156,38.193,20"`. Each element is, of course, itself a comma-separated list, but these are sublists that are not described in the `CoordinatesType` definition and are up to the client to interpret.

How do we want to map this `CoordinatesType` to *R*? In this case, we want to extend the `character` type, i.e.,

```
setClass("CoordinateTypes", contains  = "character")
```

The conversion method in this case for an *XML* node is

```
function(from)
  new("CoordinateTypes",
      strsplit(xmlValue(from), "[[:space:]]+")[[1]])
```

That is, we split the contents of the node by whitespace to get the elements.

For the *<coordinates>* element, the contents are simple strings, although they might have been declared as numbers. Consider the following schema types:

```
<simpleType name="itemIconStateType">
 <list itemType="kml:itemIconStateEnum"/>
</simpleType>
<simpleType name="itemIconStateEnum">
 <restriction base="string">
  <enumeration value="open"/>
  <enumeration value="closed"/>
  <enumeration value="error"/>
```

```
  <enumeration value="fetching0"/>
  <enumeration value="fetching1"/>
  <enumeration value="fetching2"/>
 </restriction>
</simpleType>
```

Here, each of the elements in the collection will be an object of class `itemIconStateEnum`. This type is just one of the string values from the enumerated set "open", "closed", etc. Since the individual elements are strings, we may be inclined to define the list type as extending a character vector. Unfortunately, this will lose information when we combine individual `itemIconStateEnum` values or subset a collection of them. To see this, we explore the definition

```
setClass("itemIconStateType", contains = "character")
```

Now we create an instance of this class such as

```
tmp = new("itemIconStateType",
          c(new("itemIconStateEnum", "open"),
            new("itemIconStateEnum", "closed")))
```

Querying the class of each of the elements with

```
sapply(tmp, class)
```

indicates that when we combine the values, we also lose the information. The problem is that the c() function combines the values and discards the class information about the individual elements. We can see this with

```
class( c( new("itemIconStateEnum", "open") ) )
```

This yields "character" and the `itemIconStateType` has disappeared. However, the class of `tmp` is `itemIconStateType`. If we insert individual elements rather than use the c() function, the class of the container (`itemIconStateType`) is maintained, but not that of the individual elements (`itemIconStateEnum`). For instance,

```
tmp[3] = new("itemIconStateEnum", "open")
class(tmp)
```

Subsetting also discards the class information. For instance, `tmp[1]` gives a simple `character` vector. We have lost the information that this is an `itemIconStateType` vector. We can remedy this by defining a method for the subset operator.

```
setMethod("[", "itemIconStateType",
          function(x, i, j, ...)
            new("itemIconStateType", callNextMethod()))
```

This uses the standard inherited method and then wraps the resulting object into the appropriate class.

Because of the loss of information about the class of the individual elements when we extend a primitive vector type, we may use the *list* class as the base for an *XML* schema `<list>`. We can define `itemIconStateType` as

```
setClass("itemIconStateType", contains = "list")
```

With this definition, the class of the individual information is maintained, e.g.,

```
tmp = new("itemIconStateType",
          list(new("itemIconStateEnum", "open"),
               new("itemIconStateEnum", "closed")))
```

When we subset individual elements from this, they maintain their class, e.g., `tmp[[1]]`. However, again, `tmp[1:2]` returns just a *list*, although the elements are each of class `itemIconStateEnum`. If we want `tmp[]` to maintain the class `itemIconStateType`, we need to again define a method for `[` that sets this class on the result, as we did above.

In short, we have a choice as to how we map a `<list>` type in an *XML* schema with elements that correspond to a primitive type in *R*. We can extend the primitive vector type or a list.

14.6 Working with Included and Imported Schema

In this section we address some of the facilities in the XMLSchema package that make it easier to work with *XML* schema documents. These facilities also make it easier to explore the *XML* content and to perform computations other than mapping the types directly to *R* classes and methods.

14.6.1 Processing Sub-schema

The readSchema() function by default will read other schemas that are referenced within the schema. This action is recursive so we can read a schema that refers to other schemas, which in turn refer to other schemas. It uses the location (file name or base URL) of the current schema to compute the location of relative references to other schema, e.g.,

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.ncbi...gov/soap/eutils/efetch_pubmed"
 elementFormDefault="qualified" >
<xs:include schemaLocation="efetch_pubmed.xsd" />
   ........
</xs:schema>
```

There are times when we only want to process the contents of the immediate schema and do not want to follow the imported and included schema. For instance, we may be interested in knowing the names of the top-level types in a schema and visualizing the inheritance hierarchy they form. We are not interested in the types of the slots in these classes, but just the relationships between the top-level classes. For these cases, we can control whether the `<include>` and/or `<import>` elements are processed recursively with the *followIncludes* and *followImports* parameters. We can specify `FALSE` for *followImports* and this applies to both. However, we can also chose to process only imports or includes and not the other.

14.6.2 Local Schema Files and XML Catalogs

When reading schemas that include or import other schemas, it is sometimes convenient to have the files locally so that we do not need network access or need to wait to download these additional documents. If the references to these other documents are simple relative links, we can just download them once and put them locally in the appropriate directory relative to the topmost schema. Unfortunately, some of the references to the "sub-schema" are absolute links to remote documents, and so it would

appear we cannot readily read all the files locally without editing them each time we download them. This is a case where we can use catalogs to map remote URIs to local files.

For example, the Statistical Data and Metadata eXchange (SDMX) [4] is an initiative to foster standards for the exchange of statistical information. This initiative includes a schema for creating queries for SDMX structures and data files. This query schema contains several *XML* tags of the form

```
<xsd:import namespace="http://www.SDMX.org/resources/SDMXML/
   schemas/v2_0/structure"
 schemaLocation="http://sdw-ws.ecb.europa.eu:80/
   services/SDMXQuery?xsd=5"/>
```

These schemas import other *XML* schema documents. Suppose we have downloaded these files and stored them in a directory named `/tmp/Schema/SDMX/`. That is, we have files named, for instance, `/tmp/Schema/SDMX/SDMXQuery?xsd=1`, `/tmp/Schema/SDMX/SDMXQuery?xsd=2`, etc. We can add an *XML* catalog entry and redirect URIs that start with `http://sdw-ws.ecb.europa.eu:80/services/` to `/tmp/Schema/SDMX/`. We can do this for a particular *R* session using

```
catalogAdd("http://sdw-ws.ecb.europa.eu:80/services/",
           "/tmp/Schema/SDMX/")
```

Alternatively, we can do this permanently by adding a *<rewriteURI>* element in our external catalog file, e.g., /catalog.xml or /etc/catalog.xml. That is, we can add to the catalog file:

```
<rewriteURI rewritePrefix="/tmp/Schema/SDMX/"
  uriStartString="http://sdw-ws.ecb.europa.eu:80/services/"/>
```

Once added, all the remote references will map to this `/tmp/Schema/SDMX/` directory and we can work with the local files.

14.6.3 Computations on a Schema Hierarchy

As mentioned earlier, when a schema imports or includes another schema (using *<import>* and *<include>*, respectively), we end up with a hierarchy of schema. We can process these recursively and separately. There are occasions, however, when we want to work at the level of an *XML* document that contains the actual sub-schema and not just the indirect *<import>* and *<include>* directives. We want to effectively process the *<import>* and *<include>* tags in the same way as *XInclude* would, which is to simply replace the element with the content of the file it references. For example, the function directDependencies() in the XMLSchema package makes use of this inlined *XML* document to determine what data types and elements are directly used in the definition of other top-level elements within the overall schema. This inclusion technique is used to find circular references. We use the function parseSchemaDoc() to create the necessary *XML* document. It reads the top-level *XML* document, and then finds the *<import>* and *<include>* nodes. It replaces these with the actual contents of the documents they reference. It does this recursively, and so we end up with a single *XML* document that contains all of the schemas and their nodes hierarchically. We can then use, for example, *XPath* expressions to find all *<element>* nodes with a name attribute, or all references to other types due to a *ref* attribute.

14.7 Possible Enhancements and Extensions

***S4* Classes to Schema**
Typically, interfaces we develop are bidirectional. In this case, we primarily have a one-directional interface, being able to convert *XML* documents to *R* objects based on the types in a schema. We generate the classes and methods for the fromXML() converter. We should also be able to generate the code to convert from an *R* object to *XML*. This is less developed than the code generation for fromXML(). There is code in the XMLSchema package and also in the SSOAP package to do this, but there is a great deal more that can be done. In the SSOAP package, the conversion is done generically using the descriptions of the schema types returned from readSchema(), but not by generating converter functions. Creating the functions for particular types and using these can make things slightly faster. The toSOAP() function is the primary function responsible for this. This can be greatly enhanced.

Visualizing Schema
A schema defines data types and elements. Some data types extend others, while many include others as slots. The former defines a class hierarchy and the latter a network of related classes. It would be valuable to be able to visualize these relationships. We can use the graph package and Rgraphviz [5, 6] or `igraph` [2] to create plots of graphs in *R*. Ideally, we want to add interactive capabilities to these plots to be able to examine the documentation strings (in the `<annotation>` nodes within the schema nodes) and to dynamically expand a class to see its slots and their types. We can do this in *HTML* or with an interactive *SVG* plot via SVGAnnotation [11] or `gridSVG` [9].

A General System for Data Types
Analyzing the data types in *XML* schema is quite similar to processing types from Windows Type Libraries, *C* and *C++* data structures, *Java* classes and so on. In each of these, we map data types in these external libraries to *R* classes and create code that provides an interface between the two systems. There are packages that work with each of these: SWinTypeLibs [15], RGCCTranslationUnit [12], RCIndex [13] `rJava` [18], and SJava [14]. Each of these analyzes the data structures in a similar manner, but with very different code. Creating a general infrastructure for analyzing these types of structures would be valuable not only for these existing frameworks, but also for new ones that continue to arise. Indeed, the code in the XMLSchema package would greatly benefit from being restructured within such a framework.

Efficiency and Optimization
The code we generate to convert *XML* to *R* code is reasonably efficient. However, for very large amounts of data, it can still be slow. There are opportunities to make it faster. We should profile the code and find ways to improve the performance. When we recognize particular data type definitions, e.g., arrays of string elements, we can use specialized functions for extracting the values, e.g., *C* code that does the equivalent of

```
xmlSApply(node, xmlValue)
```

We can even generate *C* code generally to read the *XML* for arbitrary data types in order to convert instances to *R*.

14.8 Summary of Functions to Work with *XML* Schema

The XMLSchema package provides functionality to read an *XML* schema definition and to create *R* class definitions and code corresponding to the data types in a schema. These can be used to parse an *XML* document into appropriate data structures in *R* and, reciprocally, to serialize *R* objects into a suitable *XML* format corresponding to the schema. There are several functions in the package that are used by the generated code and exported by the package, but not of direct interest to readers. Similarly, there are other functions that readers who want to customize how the code is generated may be interested in learning about. However, we list the high-level functions below that are most likely to be used by readers.

readSchema() Create an *R*-level description of the contents of an *XML* schema, including sub-schema imported and included by the top-level schema.

defineClasses() Create an *R* class definition for each of the data types defined in an *XML* schema, where the schema is an *R* object returned from readSchema(). This also defines the appropriate methods for the fromXML() function so that parsed *XML* content can be converted into *R* objects.

writeClassDef() Generate *R* code that can be used in another *R* session or package to define classes and coercion methods that were created in a session via defineClasses() or defClass().

14.9 Further Reading

The books [19] and [20] provide comprehensive descriptions of the *XML* schema language and its details. They explain the different components of a schema and also describe how to use and think about them. These are good references when trying to understand aspects of a schema and also how to define one.

There are, of course, numerous resources on the Web discussing *XML* schema at different levels. A good, brief introduction to the main aspects of schema is available at `http://www.w3schools.com/schema/default.asp`.

References

[1] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[2] Gabor Csardi. `igraph`: Network analysis and visualization. `http://cran.r-project.org/web/packages/igraph/index.html`, 2012. *R* package version 0.6.4.

[3] Data Mining Group. Predictive Model Markup Language. `http://www.dmg.org/pmml-v3-2.html`, 2011.

[4] Economic Commission for Europe. Common open standards for the exchange and sharing of socio-economic data and metadata: The SDMX initiative. `http://sdmx.org/docs/2002/wp11.pdf`, 2002.

[5] Jeff Gentry, Li Long, Robert Gentleman, Seth Falcon, Florian Hahne, Deepayan Sarkar, and Kaspar Hansen. Rgraphviz: Provides plotting capabilities for *R* graph objects. `http://www.bioconductor.org/packages/2.11/bioc/html/Rgraphviz.html`, 2011.

R package version 2.2.1.

[6] Florian Hahne. Rgraphviz: A new interface to render graphs using Rgraphviz. http://www.bioconductor.org/packages/2.11/bioc/vignettes/Rgraphviz/inst/doc/newRgraphvizInterface.pdf, 2012.

[7] Murat Koksalan, Jyrki Wallenius, and Stanley Zionts. *Multiple Criteria Decision Making: From Early History to the 21st Century*. World Scientific Publishing Co. Pte. Ltd., Singapore, 2011.

[8] Patrick Meyer and Sebastien Bigaret. RXMCDA. http://cran.r-project.org/package=RXMCDA, 2012. *R* package version 1.4.2.

[9] Paul Murrell. gridSVG: Export grid graphics as *SVG*. http://www.stat.auckland.ac.nz/~paul/R/gridSVG/gridSVG_0.5-10.tar.gz, 2010. *R* package version 0.5.

[10] National Center for Biotechnology Information. Entrez, the life sciences search engine. http://www.ncbi.nlm.nih.gov/gquery, 2011.

[11] Deborah Nolan and Duncan Temple Lang. SVGAnnotation: Tools for post-processing *SVG* plots created in *R*. http://www.omegahat.org/SVGAnnotation, 2011. *R* package version 0.9.

[12] Duncan Temple Lang. RGCCTranslationUnit: *R* interface to GCC source code information. http://www.omegahat.org/RGCCTranslationUnit, 2009. *R* package version 0.4-0.

[13] Duncan Temple Lang. RCIndex: *R* interface to the clang parser's *C* API. http://www.omegahat.org/RCIndex, 2010. *R* package version 0.2-0.

[14] Duncan Temple Lang. SJava. http://cran.r-project.org/package=SJava, 2011.

[15] Duncan Temple Lang. SWinTypeLibs: Type library information reader. http://www.omegahat.org/SWinTypeLibs, 2011. *R* package version 0.6-0.

[16] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[17] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.

[18] Simon Urbanek. rJava: Low-level *R* to *Java* interface. http://cran.r-project.org/package=rJava, 2011. *R* package version 0.9-3.

[19] Eric van der Vlist. *XML Schema*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[20] Priscilla Walmsley. *Definitive XML Schema*. Prentice Hall PTR, Upper Saddle River, NJ, 2001.

[21] Graham Williams, Michael Hahsler, Hemant Ishwaran, Udaya Kogalur, and Rajarshi Guha. pmml: Generate PMML for various models. http://cran.r-project.org/web/packages/pmml/index.html, 2012. *R* package version 1.2.30.

Chapter 15
Spreadsheets

Abstract In this chapter, we explore the possibilities for data exchange offered by the Office Open XML (*OOXML*) standard. Many of the office suites have adopted *OOXML* for their spreadsheets, word processing, and presentation tools. We demonstrate the kinds of functionality that can be built using the tools in the XML package to interface with *XML*-based spreadsheets from within *R*. Examples include: reading an entire **xlsx** file into an *R* data frame (or list of data frames, one per sheet); extracting and setting cell values in a worksheet; and adding style information on cells, *R* plots to sheets, and **rda** files to the **xlsx** archive. While the focus is on Excel and **xlsx** files, the ideas presented in this chapter can be extended to other spreadsheet applications, e.g., Google Docs and Open Office, and to other office tools, e.g., Word and PowerPoint. The ROOXML [19] package provides the basic infrastructure for Microsoft Office, and, for example, RWordXML provides facilities for working with word processing files.

15.1 Introduction: A Background in Spreadsheets

Spreadsheets have had a questionable reputation in the field of statistics. Statisticians have published papers warning users about the computational algorithms in spreadsheets [8, 9], e.g., they can produce negative variances, and it is a commonly held opinion that a comma-separated format is preferable to a spreadsheet because it is easy to use with a variety of statistical software languages. For example, in the online manual, *R* Data Import/Export [12] , it states that

> The most common *R* data import/export question seems to be "how do I read an Excel spreadsheet". . . . The first piece of advice is to avoid doing so if possible! If you have access to Excel, export the data you want from Excel in tab-delimited or comma-separated form, and use read.delim() or read.csv() to import it into *R*.

While this is often a reasonable approach, there is potentially much more information in the spreadsheet file format, e.g., data formatted as currency, dates, times, etc., which would be lost in the export. Of course, the situation is often more complex than that and there are many reasons why statisticians may need or even want to use spreadsheets. Moreover, spreadsheets are a widely used format for data storage and exchange as evidenced by the question "how do I read an Excel spreadsheet?"

When the ISO standardized the Office Open XML format [4] for spreadsheets, word processing documents, and presentations, all the major office suites, including MicroSoft Office [13], Apple iWork [2], Libre Office [7], KOffice [6], and Open Office [1] incorporated variants of *OOXML*. The widespread adoption of this standard creates a compelling argument for using spreadsheets as an exchange format. In working with *OOXML*, we have found many advantages to this format:

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_15,

- The **xlsx** file contains meta information about (among other things) the data format. This information can be used to automate the reading of the data, removing the need for us to specify column classes, quotes, comment characters, character encoding, etc. Using this meta information can both simplify the task of reading data from a spreadsheet into *R* and make it more verifiable and reproducible.
- The workbook may have multiple spreadsheets or spreadsheets that are not simple tables, e.g., they can contain ragged arrays or multiple tables. In this case, the simple solution of using the "Save As" feature to create a CSV file requires multiple exports or results in a potential loss of information.
- The *OOXML* format makes it relatively easy to extract data into *R* in a programmatic, repeatable, reproducible fashion. By programmatically extracting the data from the spreadsheet into *R*, we eliminate the possibility of working with data that differ from the spreadsheet. That is, we eliminate the "middle man" and extra steps in the exchange. If the spreadsheet changes, and we have not exported it as CSV since it was changed, then we will be working with the wrong data. Furthermore, we may need to repeat this process multiple times or want to keep a record of what we have done, and hence will want to access the data in the spreadsheet programmatically.
- The GUI-nature of the spreadsheet makes it a useful format for displaying the results of statistical data analysis. For example, the spreadsheet recomputes dependencies when cell values change, and graphical displays can be included directly in the display, as well as interactive controls. We may want to use the spreadsheet as a report format that contains plots, pivot tables, and formulas that recompute cell values and charts when inputs are updated. Additionally, a spreadsheet can provide attractive titles, headers, etc.
- The *OOXML* format also makes it relatively easy to *generate* spreadsheets from *R* in a programmatic, repeatable, reproducible fashion, including generating tables, formulas, and charts. If the data change, then we can repeat this generative process reliably to revise the **xlsx** file.
- The *OOXML* spreadsheet can be treated as a queryable, updateable database. This is very different from thinking of the spreadsheet as something to "throw away" after exporting the data as CSV. For example, we can query the database for hyperlinks, footnotes, and other meta-information.

As data analysts, we want to remain current with the emerging technologies for accessing and presenting quantitative information, and build new tools to access and publish data in these technologies. We have developed several *R* packages that take advantage of the *XML* structure of *OOXML* documents to work with spreadsheets from within *R*. These are RExcelXML [18], ROpenOffice [20], and RGoogleDocs [23].

The focus in this chapter is on the RExcelXML package, which provides an *R* interface with **xlsx** documents. However, ROpenOffice and RGoogleDocs behave similarly; they offer an *R* interface with Open Office and Google Docs documents, respectively, that are based on the same principles developed in RExcelXML. These packages offer functionality for *R* users to both query and modify the contents of workbooks and worksheets. The essential theme is that because the spreadsheet is a **zip** archive that contains *XML* structured content, we can manipulate this content from within *R*. Furthermore, the basic approach demonstrated here carries over to word processing documents and presentation files that follow the *OOXML* format. For example, the RWordXML package [24] offers facilities for accessing common elements from within *R* of an *OOXML* word processing document. These packages are by no means complete and polished. Instead, they serve as case studies for exploring the power of an *XML*-based data format and how it can be used within *R*.

There are several other *R* packages for working with Excel spreadsheets. These include `XLConnect` [16], `xlsReadWrite` [17], `gdata` [27], `RExcel` [10], `RODBC` [14], `WriteXLS` [15], `dataframes2xls` [25], and `xlsx` [3]. Some of these have more limited scope,

such as gdata and xlsReadWrite, and most work only with the older **xls** format. The package xlsx works with the **xlsx** format. XLConnect handles both formats and is very robust. The XLConnect and xlsx packages take an entirely different approach from RExcelXML and are based on a *Java* library for working with **xlsx** files. The approach is an excellent one as it piggy-backs on the development of toolkits in another language so the focus need only be on making the methods in these other toolkits available in *R*. When that happens, a lot of functionality becomes immediately available in *R*, and updates to the toolkit happen as the other toolkit developers revise their software. One downside, however, is that it is not easy to add specialty functionality, such as adding an **rda** file to the archive.

The packages described in this chapter are not as robust as, e.g., XLConnect. Our intention here is to demonstrate an alternative *XML*-based approach for working with *OOXML* formatted files. The goal is to provide an example of how an *R* programmer might design a package to handle spreadsheets that leverages knowledge of *OOXML* and generic tools for handling *XML*. We describe the philosophy behind this implementation and the advantages gained.

Finally, we should also mention interactive spreadsheet capabilities with *R*. When we create a spread/work-sheet from within *R*, the results are fixed. It is possible to add code to the worksheet, and workbook generally, which can make calls back to *R* to dynamically compute values within the worksheet using the *R* engine. This is feasible on Windows via *DCOM* (Distributed Component Object Model). With *DCOM*, we can communicate with Microsoft Excel, Word, PowerPoint, and various other applications as they are running. With this connection, we can programmatically query and modify the spreadsheets, documents, presentations directly from within *R* in much the same way that we can here. For example, we can access sheets, cells, and so on. The way we access these is very different, but the document model is very similar.

This chapter includes discussion of both high- and intermediate-level functions that we have created to interface with *OOXML* formatted spreadsheets. The approach we take is to first put in place functionality that handles the vast majority of spreadsheets easily and simply. Next, by delving more deeply into *SpreadsheetML* (the *OOXML* vocabulary for spreadsheets), we see how a few basic tools for accessing *XML* from within *R* can provide the programmer with the flexibility to handle more complex formats. We describe these functions that give easy access to the *XML* spreadsheet structure, e.g., functions to extract and insert values and formulas in cells and to work with styles. Finally, with an understanding of *OOXML*, we use these intermediate-level functions and the generic tools in the XML [21] and Rcompression [22] packages to produce customized functionality to, e.g., add *R* plots to a spreadsheet, extract meta information from complex spreadsheets, and use the **xlsx** archive to store atypical auxiliary data such as an **rda** file.

15.2 Simple Spreadsheets

Many workbooks consist of a single spreadsheet with a simple rectangular collection of values, where the columns correspond to variables and the rows to records, and there may be a header row at the top of the sheet to indicate column names. When this is the case, we typically want to read these data simply and directly into a data frame, just as we may read a CSV-formatted file into a data frame with read.csv(). The function read.xls() in RExcelXML does this for us. It determines which columns and rows have data in them and the type of data in each column, and it extracts the values in the cells of the spreadsheet into a data frame. We demonstrate how to use it with two examples.

15.2.1 Extracting a Spreadsheet into a Data Frame

Our first example demonstrates how to use read.xls() with a spreadsheet that contains only one simple sheet. The first row contains a header column, and the data are arranged in a simple rectangular form.

Example 15-1 Extracting Data from a World Bank Spreadsheet

The World Bank (`http://www.worldbank.org/`) provides financial and technical assistance to developing countries. In 2010, the Bank launched an Open Data Website (`http://data.worldbank.org/`) that provides access to their data, including Excel tables and reports on topics such as GDP, education, health, and the environment. For example, the World Development Report 2011 on Conflict, Security, and Development [29] has an accompanying Excel spreadsheet [28]. This spreadsheet includes 50 years of data relevant to civil war, terrorism, and trafficking for different countries. A subset of rows and columns has been extracted into a smaller spreadsheet and appears in the screenshot in Figure 15.1. We see that it has six columns/variables, each beginning with a variable name. The columns `"Year"` and `"Cwbattledeaths"` are numeric, whereas the others contain character strings.

We read the contents of the spreadsheet into the data frame with the following call to read.xlsx():

```
cwd = read.xlsx("WorldBank/MiniWDR.xlsx", header = TRUE)
```

The names of the variables in the data frame match the values in the first row of the spreadsheet. We confirm this with

```
names(cwd)
```

```
[1] "Countrycode"    "Country"        "Year"
[4] "RegionA"        "RegionB"        "Cwbattledeaths"
```

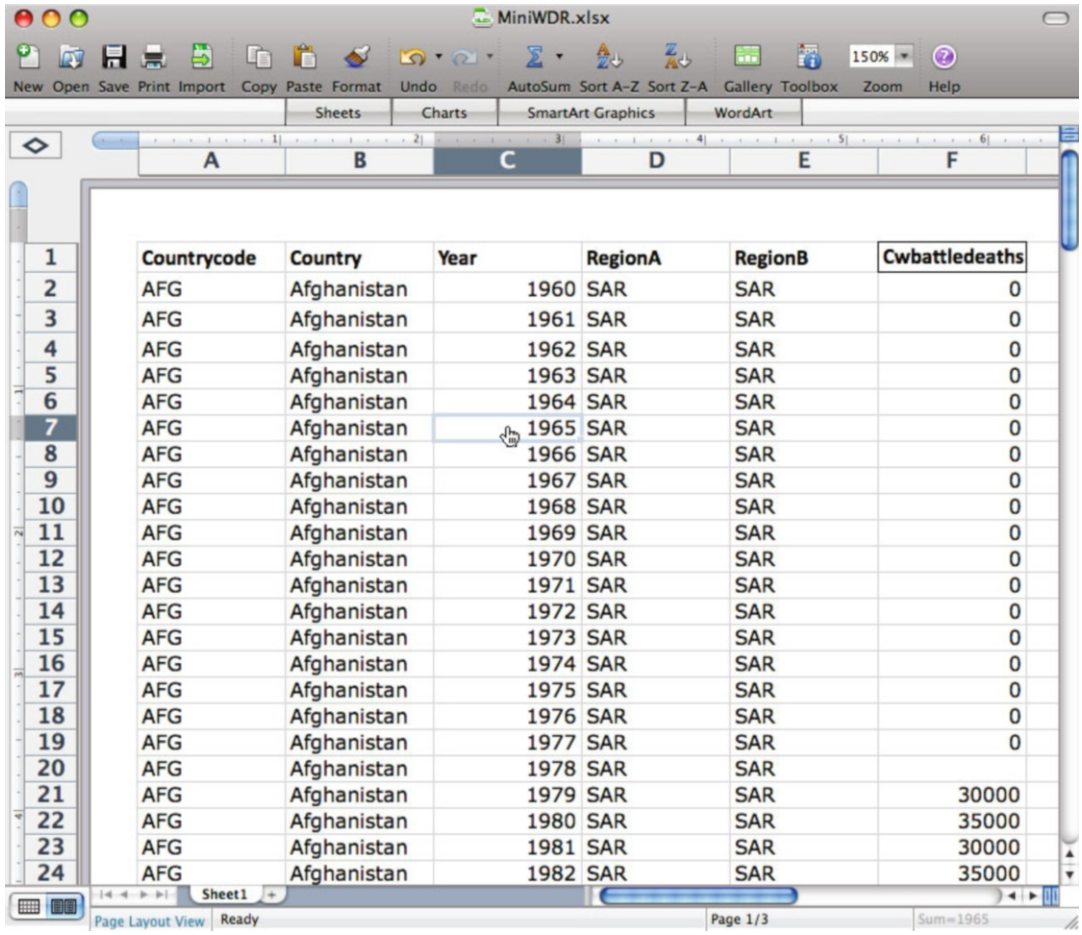
Also, we check that the character strings and numeric values in the spreadsheet are stored in *R* as factor and numeric data types, as we might expect:

```
sapply(cwd, class)
```

```
    Countrycode         Country            Year
       "factor"        "factor"       "numeric"
        RegionA         RegionB Cwbattledeaths
       "factor"        "factor"       "numeric"
```

15.2.2 Extracting Multiple Sheets from a Workbook

The read.xlsx() function has a few additional arguments to handle somewhat more complex workbooks than the one-table, one-sheet workbook in Example 15-1 (page 504). For example, if a workbook has multiple sheets, we can specify the sheet that we want to read via the *which* parameter. If there are multiple header rows at the top of the spreadsheet, then we can use the *skip* argument to specify the number of rows to ignore. Moreover, we can read multiple sheets into a list of data frames by supplying a vector to *which*. All of the arguments that control the import of a spreadsheet can be



	Countrycode	Country	Year	RegionA	RegionB	Cwbattledeaths
1						
2	AFG	Afghanistan	1960	SAR	SAR	0
3	AFG	Afghanistan	1961	SAR	SAR	0
4	AFG	Afghanistan	1962	SAR	SAR	0
5	AFG	Afghanistan	1963	SAR	SAR	0
6	AFG	Afghanistan	1964	SAR	SAR	0
7	AFG	Afghanistan	1965	SAR	SAR	0
8	AFG	Afghanistan	1966	SAR	SAR	0
9	AFG	Afghanistan	1967	SAR	SAR	0
10	AFG	Afghanistan	1968	SAR	SAR	0
11	AFG	Afghanistan	1969	SAR	SAR	0
12	AFG	Afghanistan	1970	SAR	SAR	0
13	AFG	Afghanistan	1971	SAR	SAR	0
14	AFG	Afghanistan	1972	SAR	SAR	0
15	AFG	Afghanistan	1973	SAR	SAR	0
16	AFG	Afghanistan	1974	SAR	SAR	0
17	AFG	Afghanistan	1975	SAR	SAR	0
18	AFG	Afghanistan	1976	SAR	SAR	0
19	AFG	Afghanistan	1977	SAR	SAR	0
20	AFG	Afghanistan	1978	SAR	SAR	
21	AFG	Afghanistan	1979	SAR	SAR	30000
22	AFG	Afghanistan	1980	SAR	SAR	35000
23	AFG	Afghanistan	1981	SAR	SAR	30000
24	AFG	Afghanistan	1982	SAR	SAR	35000

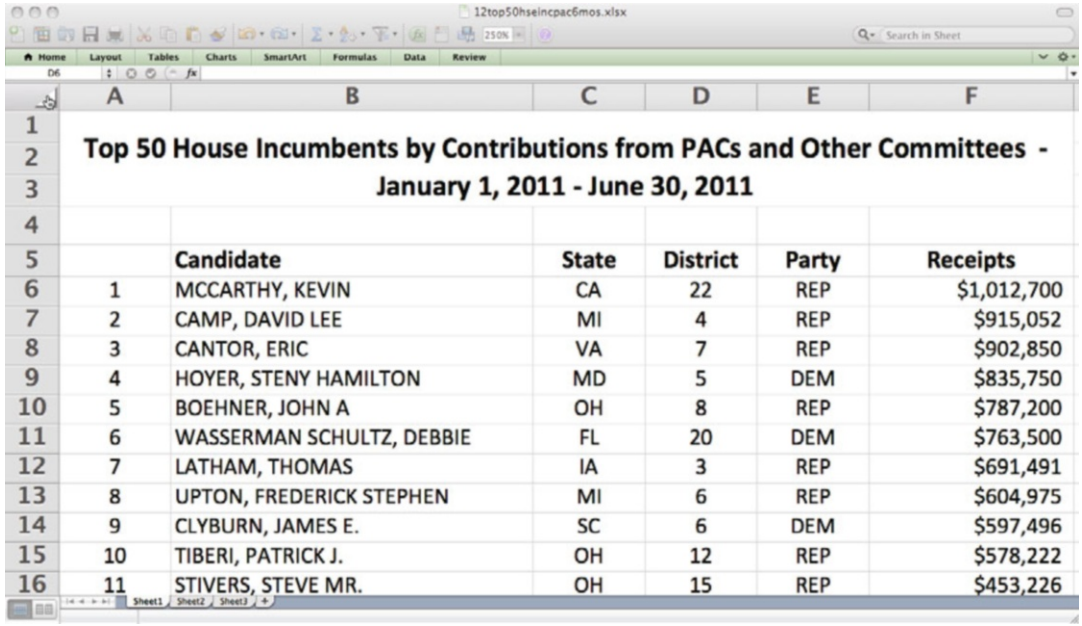
Figure 15.1: World Bank Excel Report on Conflict. This screenshot shows the simple structure of an Excel spreadsheet, which is an extract of the spreadsheet provided by the World Bank Report 2011 on conflict security and development for different countries in different years. The workbook contains only one sheet, which has six columns including the name of the country, year, and the number of deaths due to civil war battles. This information was downloaded from `http://data.worldbank.org/` in September 2011.

passed a vector. This way, we can specify how to handle each sheet independently. We demonstrate how to use these vector arguments in the next example.

Example 15-2 Extracting Federal Exchange Commission (FEC) Data from Multiple Worksheets

The US Federal Election Commission (FEC) is an independent regulatory agency set up to disclose campaign finance information and oversee the public funding of US federal elections. The FEC makes its data publicly available at `http://www.fec.gov/`. One example is its six-month summary, for the first half of 2011, of contributions from Political Action Committees (PAC) and other committees to incumbents running for re-election in the House of Representatives [5]. These data are provided in an Excel workbook (see Figures 15.2 and 15.3), which contains three worksheets. We are primarily interested in the first of these. Although the data in the first sheet are in a rectangular form, the sheet

has a title that occupies the first three rows, the fourth row is empty, and the fifth row contains column headers, which we would like to use as variable names.



	A	B	C	D	E	F
1						
2		Top 50 House Incumbents by Contributions from PACs and Other Committees -				
3		January 1, 2011 - June 30, 2011				
4						
5		Candidate	State	District	Party	Receipts
6	1	MCCARTHY, KEVIN	CA	22	REP	$1,012,700
7	2	CAMP, DAVID LEE	MI	4	REP	$915,052
8	3	CANTOR, ERIC	VA	7	REP	$902,850
9	4	HOYER, STENY HAMILTON	MD	5	DEM	$835,750
10	5	BOEHNER, JOHN A	OH	8	REP	$787,200
11	6	WASSERMAN SCHULTZ, DEBBIE	FL	20	DEM	$763,500
12	7	LATHAM, THOMAS	IA	3	REP	$691,491
13	8	UPTON, FREDERICK STEPHEN	MI	6	REP	$604,975
14	9	CLYBURN, JAMES E.	SC	6	DEM	$597,496
15	10	TIBERI, PATRICK J.	OH	12	REP	$578,222
16	11	STIVERS, STEVE MR.	OH	15	REP	$453,226

Figure 15.2: FEC Spreadsheet Report of Campaign Contributions. The screenshot shows the format of the first worksheet in the Excel spreadsheet `12Top50HouseIncumbentByCPAC6Months.xlsx`. Notice that the column names appear in row five, the data begin in row six, and the first column has no column name. The spreadsheet was downloaded from `http://www.fec.gov` in September 2011.

To extract the data, we use the read.xlsx() arguments *which*, *skip*, and *header* to specify that we want the data in the first sheet and that we want to skip the first four rows and use the next row as a header. That is,

```
FEC = "FEC/12Top50HouseIncumbentByCPAC6Months.xlsx"
top50 = read.xlsx(FEC, which = 1, skip = 4, header = TRUE)
names(top50)
```

```
[1] "Candidate" "State" "District" "Party" "Receipts"
```

```
dim(top50)
```

```
[1] 50  5
```

The second sheet contains additional information (see Figure 15.3), which we can read into *R* with a second call to read.xlsx(). Instead, we demonstrate how to read the data from both worksheets into *R* at once with

```
top50 = read.xlsx(FEC, which = 1:2, header = c(TRUE, FALSE),
                  skip = c(4,0))
```

The return value is a list of two data frames. The second data frame contains the values from `"Sheet2"`. Note that there is no header in this sheet, i.e., the data begin in the first row. This means the variable names in *R* are generic, as would happen with read.csv(). We confirm this with

```
names(top50[[2]])
```

```
[1] "V1" "V2" "V3"
```

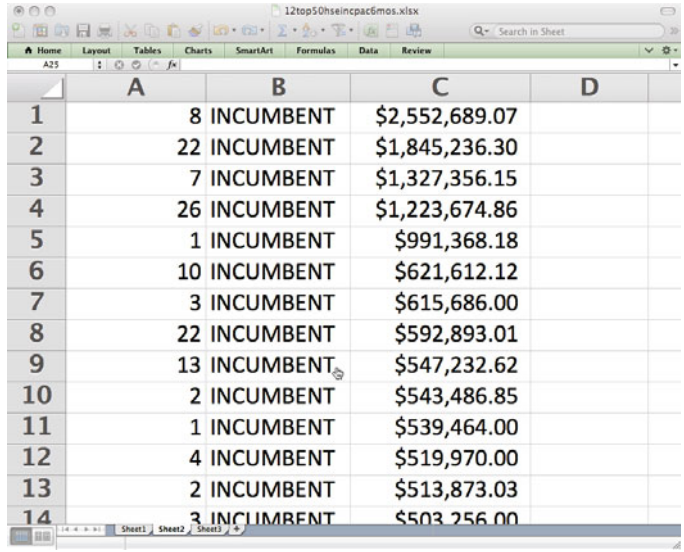


Figure 15.3: Second Worksheet in an FEC Workbook. This screen shot shows the contents of the second worksheet of `12Top50HouseIncumbentByCPAC6Months.xlsx`. It has no column names and column C contains currency amounts that contain dollar signs and commas. When we read this sheet into *R* with read.xlsx(), each column is extracted into a vector in a data frame and given a generic name, e.g., `"V1"`, and the cell values are converted into the appropriate type, e.g., the currency is converted to numeric.

The read.xlsx() function identifies the types of the columns and collapses them to the appropriate type in *R*, e.g., we convert the currency to numeric, not strings with `"$"`, i.e.,

```
sapply(top50[[2]], class)
```

```
       V1        V2        V3
"numeric"  "factor" "numeric"
```

Note also that we convert `"INCUMBENT"` to a factor rather than a string. In addition, if there are blank regions in the data rectangle, then we convert these to missing values.

15.3 Office Open XML

The basic function read.xlsx() handles most cases where the data are in a simple rectangular form. How we do this is by extracting information from the *XML* documents within the **xlsx** archive. The read.xlsx() function uses the core tools in the XML package to extract cell values, find cell formats, etc. When we have more complex spreadsheets that, e.g., have a more flexible format, we can use these same tools to extract content. Rather than converting a worksheet into a data frame all in one step, we provide functions that access various components of the **xlsx** file. These intermediate-level functions offer functionality between the high-level approach of read.xlsx() and the basic parsing functions of XML (see Chapters 3, 4, and 5). To use these intermediate-level functions, we need to have some knowledge of the **xlsx** archive because these functions are not as all-inclusive as read.xlsx(). In this section, we describe the basic structure of the **xlsx** archive and how information is stored in it. This will give us enough understanding to handle more complex extractions and to also create spreadsheets from within *R* that contain, e.g., *R* plots.

15.3.1 The xlsx Archive

The Office Open XML (*OOXML*) standard [4, 26] uses a **zip** archive to store the contents of a spreadsheet, word document, or presentation, e.g., PowerPoint or Libre Office. The **zip** files contain data files as well as files that indicate the relationships between the content files. To explore the **zip** archive and the contents of its files, we examine the **xlsx** archive for the simple spreadsheet shown in Figure 15.4.

The documents for this spreadsheet are packaged in a **zip** archive that contains several directories and about 20 files. There is one file for each worksheet and each image, in addition to files that contain information about styles. Other files contain information that is used to connect the styles and images to the spreadsheet that uses the styles or "contains" the images. We access the files in the archive with the excelDoc() function. This function creates an object which allows us to treat the **zip** file as a list in *R* and access the individual files it contains. We call it with:

```
ed = excelDoc("test.xlsx")
```

The excelDoc() function relies on the Rcompression package to provide general access to the **zip** archive (without writing the files on disk). It provides a convenient interface to the zipArchive() function in Rcompression. We can examine `ed` to find that our simple spreadsheet consists of 22 files with the following names:

```
names(ed)
```

```
 [1] "[Content_Types].xml"
 [2] "_rels/.rels"
 [3] "xl/_rels/workbook.xml.rels"
 [4] "xl/workbook.xml"
 [5] "xl/styles.xml"
 [6] "xl/theme/theme1.xml"
 [7] "xl/worksheets/sheet2.xml"
 [8] "xl/worksheets/_rels/sheet1.xml.rels"
 [9] "xl/worksheets/_rels/sheet2.xml.rels"
[10] "xl/drawings/_rels/drawing1.xml.rels"
```

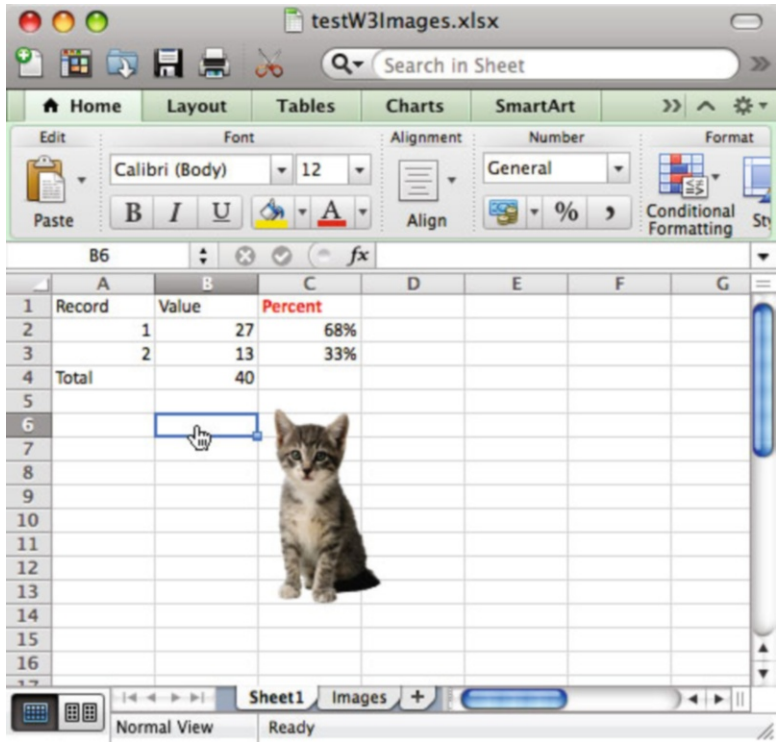


Figure 15.4: Example Spreadsheet. This screenshot shows a small spreadsheet that has a few interesting features. There are two sheets. On the first sheet, the column header for the third column is formatted in a bold-face, red font, and the values in this column are computed using a formula and formatted as a percentage. A PNG file is included on this sheet as well. The second sheet, called `"Images"`, contains two PNG files: a photo and an *R* plot.

```
[11] "xl/drawings/_rels/drawing2.xml.rels"
[12] "xl/worksheets/sheet1.xml"
[13] "docProps/thumbnail.jpeg"
[14] "xl/media/image1.png"
[15] "xl/drawings/drawing2.xml"
[16] "xl/drawings/drawing1.xml"
[17] "xl/media/image2.JPG"
[18] "xl/sharedStrings.xml"
[19] "xl/media/image3.png"
[20] "docProps/core.xml"
[21] "xl/calcChain.xml"
[22] "docProps/app.xml"
```

This structure is quite similar across all **xlsx** files (and is similar to Libre Office, etc.). At its root, the **xlsx** archive contains an *XML* file called `[Content_Types].xml` and three directories: `_rels/`, `xl/`, and `docProps/`. Here, we see that the `xl/` directory contains a central workbook file named `workbook.xml` and files for the two worksheets called `worksheets/sheet1.xml` and `worksheets/sheet2.xml`. The core content of the file the user sees is found in this application-

specific directory. The other directories contain information about the data and how they are to appear in the application's interface.

OOXML defines *XML* vocabularies for spreadsheets, as well as word processing documents and presentations. These are *SpreadsheetML*, *WordprocessingML*, and *PresentationML*, respectively. The word processing and presentation archives have similar structures, where the `xl/` directory is replaced by one specific to the document type, e.g., a `word/` directory is used in the word processing **zip** archive. In the following sections, we examine the *SpreadsheetML* vocabulary and some of the auxiliary files in the archive. We see that the **xlsx** document is organized in a distributed manner. The information in the spreadsheet can be stored in a single file, but has been organized for maximum re-use of parts of a document and maximum flexibility in replacing parts, etc. It is confusing at first with the large number of files and level of indirectness, but the structure is quite sensible.

15.3.2 The Workbook

The workbook file called `xl/workbook.xml` keeps track of the worksheets, global settings and other shared components of the workbook. It points to the worksheets via its *<sheets>* node as shown with `xmlRoot(ed[["xl/workbook.xml"]])`:

```
<workbook
 xmlns="http://schemas.../spreadsheetml/2006/main"
 xmlns:r="http://...officeDocument/2006/relationships">
 <fileVersion appName="xl" lastEdited="5"
    lowestEdited="5" rupBuild="21405"/>
 <workbookPr showInkAnnotation="0" autoCompressPictures="0"/>
 <bookViews>
  <workbookView xWindow="0" yWindow="0" windowWidth="25600"
    windowHeight="14840" tabRatio="500" activeTab="1"/>
 </bookViews>
 <sheets>
  <sheet name="Sheet1" sheetId="1" r:id="rId1"/>
  <sheet name="Images" sheetId="2" r:id="rId2"/>
 </sheets>
 <calcPr calcId="140000" concurrentCalc="0"/>
 <extLst>
  <ext xmlns:mx="http://.../mac/excel/2008/main"
    uri="{7523E5D3-25F3-A5E0-1632-64F254C22452}">
   <mx:ArchID Flags="2"/>
  </ext>
 </extLst>
</workbook>
```

We see that neither the worksheet itself nor its file name appears in the workbook. Instead, there is an *r:id* attribute which we use to determine that the first worksheet file name is `xl/worksheets/sheet1.xml`. The information to tie the workbook and worksheet files together is stored in a relationships (i.e., `.rels`) file. This extra layer of indirection can be a bit messy to work with, but it is also very flexible. We describe it in greater detail in Section 15.7 where we provide examples of adding a worksheet to a workbook and an image to a worksheet.

15.3.3 Cells and Worksheets

We next examine the contents of the worksheet *XML* file for the first sheet, i.e., `sheet1.xml`. The root node of this document has eight children with the following names:

```
sheet1 = xmlRoot(ed[["xl/worksheets/sheet1.xml"]])
names(sheet1)
```

```
      dimension      sheetViews   sheetFormatPr       sheetData
    "dimension"    "sheetViews" "sheetFormatPr"     "sheetData"
    pageMargins       pageSetup         drawing          extLst
  "pageMargins"     "pageSetup"       "drawing"        "extLst"
```

The <*dimension*> element is:

```
sheet1[["dimension"]]
```

```
<dimension ref="A1:C4"/>
```

It has a *ref* attribute that specifies the extent of the rectangular region containing any data in the sheet. The actual cell values are found in the <*sheetData*> node. There is one <*row*> element for each row in the spreadsheet containing content. We examine the first <*row*> node in <*sheetData*> with `sheet1[["sheetData"]][[1]]`. This returns the row with the column headers, `"Record"`, `"Value"`, and `"Percent"`:

```
<row r="1" spans="1:3">
 <c r="A1" t="s"> <v>0</v> </c>
 <c r="B1" t="s"> <v>1</v> </c>
 <c r="C1" s="1" t="s"> <v>2</v> </c>
</row>
```

We see that there is a <*c*> (for cell) element for each nonempty cell in the row. The <*v*> child of <*c*> contains the value of the cell. These are 0, 1, and 2, rather than the strings for the column headers `"Record"`, `"Value"`, and `"Percent"`, as might be expected. The reason for this is that to optimize the use of strings, a single instance of each unique string is stored in a shared strings table for all the worksheets and their cells, and not in the cell's node. This shared table is stored in a separate file called `xl/sharedStrings.xml` in the **xlsx** archive. The cells then reference the string by its index in the shared table. The standard uses zero-based counting so the value of 0 in the A1 cell refers to the first element in the shared strings table. We know that the value refers to a shared string and not the number 0 because the *t* (type) attribute on the parent <*c*> has a value of `"s"` to denote that the cell contains a string data type. Also, the *s* (style) attribute on the third cell references style information for that cell.

In our example, the shared string table is small, with only four entries, and the first entry contains the string `"Record"`. We confirm this with `ed[["xl/sharedStrings.xml"]]`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<sst xmlns="http://schemas.../spreadsheetml/2006/main"
     count="4" uniqueCount="4">
 <si> <t>Record</t> </si>
 <si> <t>Value</t> </si>
 <si> <t>Percent</t> </si>
```

```
 <si> <t>Total</t> </si>
</sst>
```

The "Total" string is in the fourth row of the first column of the spreadsheet. The second sheet has no strings, just images so has no entries in this table. If a string is used multiple times, possibly in multiple worksheets, then this approach to storing strings can be very efficient. For example, the spreadsheet in Example 15-1 (page 504) repeats each country name 50 times, once for each year of reporting. Rather than having 50 cells with "Afghanistan" as the value in <v>, there are 50 references to the shared string "10". This is very similar to *R*'s factor type.

Lastly, we examine the second row of the worksheet (sheet1[["sheetData"]][[2]]) to find:

```
<row r="2" spans="1:3">
 <c r="A2"> <v>1</v> </c>
 <c r="B2"> <v>27</v> </c>
 <c r="C2" s="2"> <f>B2/B4</f> <v>0.67500000000000004</v> </c>
</row>
```

In this row, the value 1 found in the A2 cell corresponds to the number 1 because <c> has no *t* attribute so the type defaults to number. The third cell in the row, i.e., C2, has two children, <f> and <v>. The <f> element contains a formula and <v> contains the cached value from the last time the formula was calculated. Notice also that this value is 0.675..., rather than 68% as seen in Figure 15.4. The "68%" is a representation of this value. It appears in Excel as 68% because of the formatting applied to it and specified by this style, i.e., the parent <c> has a style of "2" that is used to format the cell value.

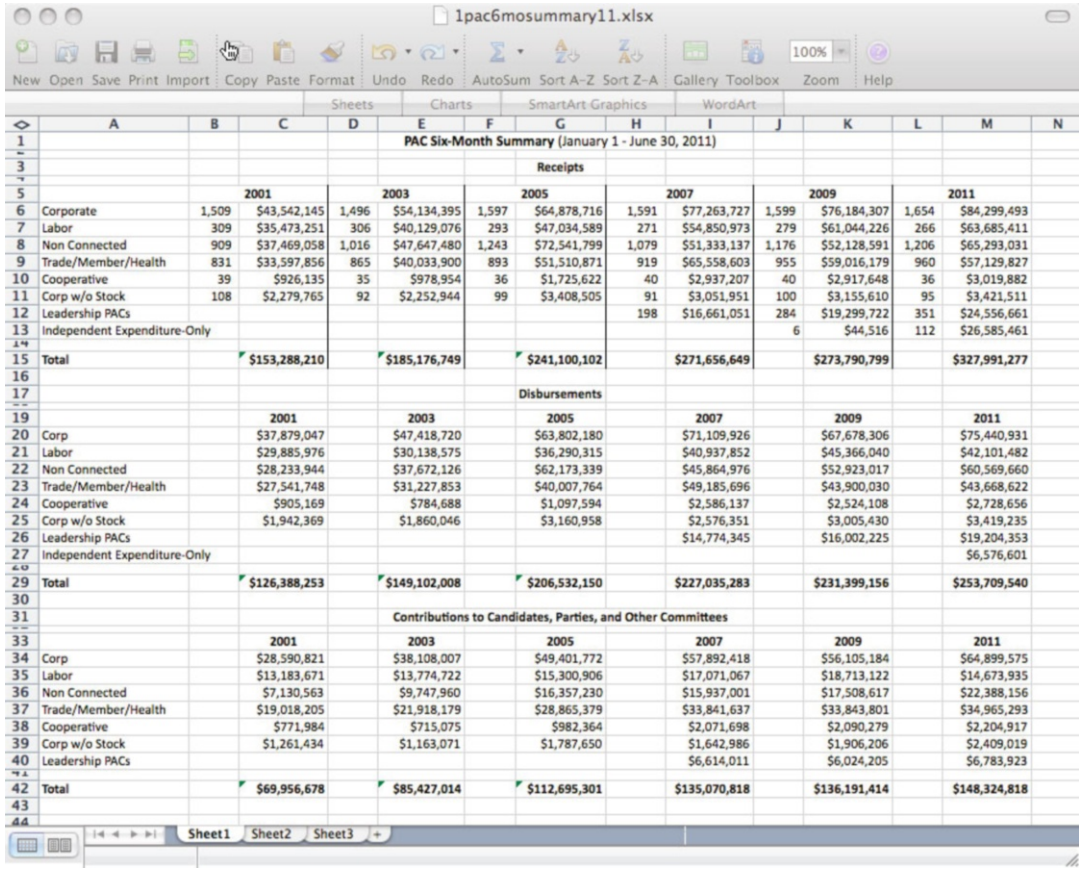
15.4 Intermediate-Level Functions for Extracting Subsets of a Worksheet

Data can be organized quite flexibly in a spreadsheet, and when the contents of the worksheet are not in a simple tabular form, read.xlsx() may give us lots of NAs where there are holes in the table. That is, spreadsheets may be well formatted for people to view and not for data exchange. When this is the case, we may want to work more directly with the **xlsx** archive and the sheet files in order to extract subsets explicitly rather than getting all of the cells in one go. Of course, we can use read.xlsx() to extract all of the data from the sheet as one huge data frame and work within *R* to clean it up. However, we may instead want to use the structure of the spreadsheet to extract specific pieces of data.

Another spreadsheet made available by the FEC at http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx provides information about PAC contributions in the first half of 2011 (see Figure 15.5). It contains three tables in one sheet. The rows in each table correspond to the source of the contribution, e.g., corporate, labor. The first table provides information about funds received, the second about disbursements, and the third contributions. Blank lines in rows 2, 4, 14, 16, 18, 28, 30, and 32 visually separate the tables from each other and from rows that report column totals for the tables. Notice also that the first table has two columns for each reporting year; one holds the number of receipts and the other the total amount of funds received. The other tables have only one column of information per year.

There are many ways that we may want to organize these data in *R*. We take a simple approach and create separate data frames for each table in the worksheet. When the data are in *R*, we can combine and restructure them into a single data frame more suitable for analysis. Our goal here is

to demonstrate how to work with the Excel file as we extract pieces of a worksheet to create a data frame.



1pac6mosummary11.xlsx

New Open Save Print Import Copy Paste Format Undo Redo AutoSum Sort A-Z Sort Z-A Gallery Toolbox 100% Zoom Help

Sheets Charts SmartArt Graphics WordArt

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1					PAC Six-Month Summary (January 1 - June 30, 2011)									
3							Receipts							
5			2001		2003		2005		2007		2009		2011	
6	Corporate	1,509	$43,542,145	1,496	$54,134,395	1,597	$64,878,716	1,591	$77,263,727	1,599	$76,184,307	1,654	$84,299,493	
7	Labor	309	$35,473,251	306	$40,129,076	293	$47,034,589	271	$54,850,973	279	$61,044,226	266	$63,685,411	
8	Non Connected	909	$37,469,058	1,016	$47,647,480	1,243	$72,541,799	1,079	$51,333,137	1,176	$52,128,591	1,206	$65,293,031	
9	Trade/Member/Health	831	$33,597,856	865	$40,033,900	893	$51,510,871	919	$65,558,603	955	$59,016,179	960	$57,129,827	
10	Cooperative	39	$926,135	35	$978,954	36	$1,725,622	40	$2,937,207	40	$2,917,648	36	$3,019,882	
11	Corp w/o Stock	108	$2,279,765	92	$2,252,944	99	$3,408,505	91	$3,051,951	100	$3,155,610	95	$3,421,511	
12	Leadership PACs							198	$16,661,051	284	$19,299,722	351	$24,556,661	
13	Independent Expenditure-Only									6	$44,516	112	$26,585,461	
15	Total		$153,288,210		$185,176,749		$241,100,102		$271,656,649		$273,790,799		$327,991,277	
16														
17							Disbursements							
19			2001		2003		2005		2007		2009		2011	
20	Corp		$37,879,047		$47,418,720		$63,802,180		$71,109,926		$67,678,306		$75,440,931	
21	Labor		$29,885,976		$30,138,575		$36,290,315		$40,937,852		$45,366,040		$42,101,482	
22	Non Connected		$28,233,944		$37,672,126		$62,173,339		$45,864,976		$52,923,017		$60,569,660	
23	Trade/Member/Health		$27,541,748		$31,227,853		$40,007,764		$49,185,696		$43,900,030		$43,668,622	
24	Cooperative		$905,169		$784,688		$1,097,594		$2,586,137		$2,524,108		$2,728,656	
25	Corp w/o Stock		$1,942,369		$1,860,046		$3,160,958		$2,576,351		$3,005,430		$3,419,235	
26	Leadership PACs								$14,774,345		$16,002,225		$19,204,353	
27	Independent Expenditure-Only												$6,576,601	
29	Total		$126,388,253		$149,102,008		$206,532,150		$227,035,283		$231,399,156		$253,709,540	
30														
31							Contributions to Candidates, Parties, and Other Committees							
33			2001		2003		2005		2007		2009		2011	
34	Corp		$28,590,821		$38,108,007		$49,401,772		$57,892,418		$56,105,184		$64,899,575	
35	Labor		$13,183,671		$13,774,722		$15,300,906		$17,071,067		$18,713,122		$14,673,935	
36	Non Connected		$7,130,563		$9,747,960		$16,357,230		$15,937,001		$17,508,617		$22,388,156	
37	Trade/Member/Health		$19,018,205		$21,918,179		$28,865,379		$33,841,637		$33,843,801		$34,965,293	
38	Cooperative		$771,984		$715,075		$982,364		$2,071,698		$2,090,279		$2,204,917	
39	Corp w/o Stock		$1,261,434		$1,163,071		$1,787,650		$1,642,986		$1,906,206		$2,409,019	
40	Leadership PACs								$6,614,011		$6,024,205		$6,783,923	
42	Total		$69,956,678		$85,427,014		$112,695,301		$135,070,818		$136,191,414		$148,324,818	
43														

Sheet1 Sheet2 Sheet3

Figure 15.5: FEC Worksheet with Multiple Tables. This screenshot shows an Excel spreadsheet published on the Web by the Federal Election Commission. It holds three tables that disclose the source of political contributions by (`"Receipts"`, `"Disbursements"`, and `"Contributions to Candidates, Parties and other Committees"`). It was downloaded from `http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx` in October, 2011.

15.4.1 The Excel Archive in R

To begin our extraction, we call excelDoc() to access the files within the **xlsx** archive:

```
FECExcelDoc = excelDoc("FEC/1pac6mosummary11.xlsx")
```

Although the `ExcelArchive` object is of interest for several tasks, users will typically want to work with a `Workbook` object as this is the entity that contains the worksheets and data.

15.4.2 The Excel Workbook in R

The workbook() function creates a different view of the **xlsx** file, i.e., an object that has class `Workbook`. This is an object with two slots: *content*, which is an *XML* document, and *name*, which is a vector of length 2 giving the name of the **xslx** file and the name of the *XML* file that is associated with this workbook. The name is used to remember from whence the content came so that we can save any changes we make to the correct place.

The workbook() function takes either the name of the **xslx** file or the `ExcelArchive` object and returns an object of class `Workbook`. We pass it the archive with

```
wbFEC = workbook(FECExcelDoc)
```

The `Workbook` class has methods for the names() and `[[` functions. The names() function returns the names of the worksheets it contains. We see that for our workbook we have three sheets with the typical Excel names:

```
names(wbFEC)
```

```
    rId1     rId2     rId3
"Sheet1" "Sheet2" "Sheet3"
```

These are the titles that appear in the tabs along the bottom of the Excel app.

The `[[` method allows you to extract an individual worksheet, either by name or by position. The following are equivalent:

```
ws1 = wbFEC[[1]]
ws1 = wbFEC[["Sheet1"]]
```

The return value is an object of class `Worksheet`.

15.4.3 The Excel Worksheet in R

A `Worksheet` is an object that represents the sheet in the archive. An object of this class describes the sheet in the archive, not its data/contents, i.e., it describes the sheet symbolically. We can think of it like a data frame or matrix. For example, we can ask for its dimensions, e.g.,

```
dim(ws1)
```

```
[1] 43 16
```

This tells us that there are 43 rows and 16 columns of active cells in the first worksheet of our workbook. This function ignores the empty columns and rows before and after the main block of data, but it does not ignore the empty columns and rows within the populated region.

In addition to extracting a worksheet with the `[[` operator, we can use getSheet() directly to retrieve worksheets as a list of `Worksheet` objects, e.g.,

```
sheets = getSheet(wbFEC, which = 1)
```

In the next example, we will extract the cell values from the middle table (labeled `"Disbursements"`) of the first worksheet. We will perform three extractions to build the data frame. First, we will extract the first column and use these cell values as row names in the data frame.

Then, we extract the data values, and lastly we obtain the row containing the years to use as variable names in our data frame. We use the example to demonstrate several approaches for accessing specific regions in the spreadsheet.

Example 15-3 Extracting a Rectangular Region from an FEC Worksheet

We begin by using the typical Excel specification of a column, e.g., A20:A27, to get the row names for the data frame from the first column of the spreadsheet. We do this with

```
rowNs = ws1["A20:A27"]
head(rowNs)
```

```
[1] "Corp"                "Labor"
[3] "Non Connected"       "Trade/Member/Health"
[5] "Cooperative"         "Corp w/o Stock"
```

Notice that we need to know the extent of this information (e.g., it is in the first column, rows 20–27) in order to extract it.

We next extract the numeric content of the worksheet. We are interested in the region C20:M27, but we first examine a few of the rows with

```
ws1["c20:J22"]
```

```
           C  D        E  F        G  H        I  J
20 37879047 NA 47418720 NA 63802180 NA 71109926 NA
21 29885976 NA 30138575 NA 36290315 NA 40937852 NA
22 28233944 NA 37672126 NA 62173339 NA 45864976 NA
```

We see that columns D, F, H, and J contain `NA`s as these columns are used only for the first table and we are working with the second table. We can extract only those columns with numbers as follows:

```
disburse = ws1[20:27, seq(3,13, by = 2)]
disburse[ 1:3, 1:4 ]
```

```
           C        E        G        I
20 37879047 47418720 63802180 71109926
21 29885976 30138575 36290315 40937852
22 28233944 37672126 62173339 45864976
```

Here we have used *R* terminology to specify the extent, i.e., rows 20:27 and columns 3, 5, 7, etc. That is, the `[` operator has been extended to accept either the spreadsheet cell terminology or *R* terminology for subsetting.

Lastly, we demonstrate how to subset the worksheet by "name" and by logicals. For example, we can extract the row names with

```
ws1[20:27, "A"]
```

and we can extract the years from row 19 with

```
years = ws1[ 19, c(rep(FALSE, 2), rep(c(TRUE, FALSE), 6)) ]
years
```

```
      C    E    G    I    K    M
19 2001 2003 2005 2007 2009 2011
```

Now that we have these three pieces, we complete the creation of the data frame for disbursements with

```
row.names(disburse) = rowNs
names(disburse) = paste0("Y", years[1, ])
```

The first row in the spreadsheet in Figure 15.5 has a title that spans multiple columns. Extracting such titles is the topic of the next section.

15.5 Accessing Highly Formatted Spreadsheets

A spreadsheet or workbook can be very effective for displaying data because it allows us to annotate the data with additional information such as explanations of variables, provenance of the data, units for variables, and footnotes about particular values or columns. It also allows us to provide rich headers or titles for columns in each sheet. Often these will be formatted to span multiple columns. One example of a complex, richly formatted spreadsheet is `c07_tabB1.xlsx` that we downloaded from the Census Bureau at `http://blueprod.ssd.census.gov/statab/ccdb/`. Similar data can be found at `http://factfinder.census.gov/`. In the next example, we demonstrate how to read into *R* various parts of the spreadsheet, e.g., footnotes and titles that span multiple columns.

Example 15-4 Working with Detailed Titles and Footnotes in a US Census Spreadsheet

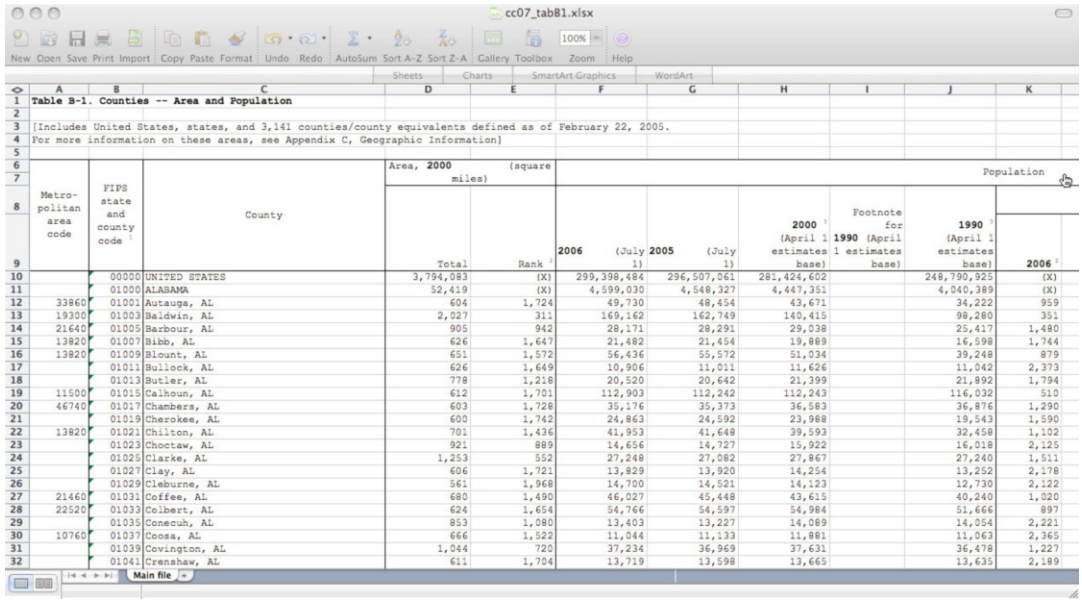
If we view the `c07_tabB1.xlsx` spreadsheet in Excel (or any spreadsheet application), we can see how the different parts deviate from the simple tabular displays in the earlier examples in this chapter (see Figure 15.6). There is a title in cell A1. Cells A3 and A4 contain additional qualifying information. The data are in columns A–Q and in rows 10–3209, inclusive. The values are formatted with commas separating the thousands. The headers span three rows with main titles and subtitles. Additionally, there are footnotes in the document that we want to recover to include in a data description (see Figure 15.7). In this example, we demonstrate how to handle the header information and retrieve the footnotes.

We begin by opening the spreadsheet from within *R* and accessing the sheet called `"Main file"`. Rather than first call excelDoc() and then pass the workbook() function the `excelArchive` object, we provide workbook() with the file name,

```
wb = workbook("cc07_tabB1.xlsx")
sh = wb[[1]]
```

Getting the Header

When we look at the document in a spreadsheet application, we can see that the titles and labels for the columns are above the data and just below the initial title and notes. The titles are recognizable to humans, but hard to recognize algorithmically. We have `"Metropolitan area code"`, `"FIPS state and county code"` (with a footnote index), `"County"`, `"Area, 2000 (square miles)"` and `"Population"`. However for the last two of these, the titles span multiple columns and there are subtitles that connect these to the specific columns. For instance, for `"Area, 2000"` we have Total and Rank. For `"Population"`, we have five columns for four different years—2006, 2005, 2000 and 1990—and an extra column for footnotes associated with the 1990 values. To the right of these, we have columns for `"Rank"` and `"Persons per square mile of land area"`, again within the general title of `"Population"`, and within these there are subcategories



cc07_tab81.xlsx

	A	B	C	D	E	F	G	H	I	J	K
1	Table B-1. Counties -- Area and Population										
2											
3	[Includes United States, states, and 3,141 counties/county equivalents defined as of February 22, 2005.										
4	For more information on these areas, see Appendix C, Geographic Information]										
5											
6				Area, 2000 (square miles)		Population					
7											
8	Metro-politan area code	FIPS state and county code [7]	County	Total	Rank [8]	2006 (July 1)	2005 (July 1)	2000 [3] (April 1 estimates base)	Footnote for 1990 (April 1 estimates base)	1990 [3] (April 1 estimates base)	2006 [4]
9											
10		00000	UNITED STATES	3,794,083	(X)	299,398,484	296,507,061	281,424,602		248,790,925	(X)
11		01000	ALABAMA	52,419	(X)	4,599,030	4,548,327	4,447,351		4,040,389	(X)
12	33860	01001	Autauga, AL	604	1,724	49,730	48,454	43,671		34,222	959
13	19300	01003	Baldwin, AL	2,027	311	169,162	162,749	140,415		98,280	351
14	21640	01005	Barbour, AL	905	942	28,171	28,291	29,038		25,417	1,480
15	13820	01007	Bibb, AL	626	1,647	21,482	21,454	19,889		16,598	1,744
16	13820	01009	Blount, AL	651	1,572	56,436	55,572	51,034		39,248	879
17		01011	Bullock, AL	626	1,649	10,906	11,011	11,626		11,042	2,373
18		01013	Butler, AL	778	1,218	20,520	20,642	21,399		21,892	1,794
19	11500	01015	Calhoun, AL	612	1,701	112,903	112,242	112,243		116,032	510
20	46740	01017	Chambers, AL	603	1,728	35,176	35,373	36,583		36,876	1,290
21		01019	Cherokee, AL	600	1,742	24,863	24,592	23,988		19,543	1,590
22	13820	01021	Chilton, AL	701	1,436	41,953	41,648	39,593		32,458	1,102
23		01023	Choctaw, AL	921	889	14,656	14,727	15,922		16,018	2,125
24		01025	Clarke, AL	1,253	552	27,248	27,082	27,867		27,240	1,511
25		01027	Clay, AL	606	1,721	13,829	13,920	14,254		13,252	2,178
26		01029	Cleburne, AL	561	1,968	14,700	14,521	14,123		12,730	2,122
27	21460	01031	Coffee, AL	680	1,490	46,027	45,448	43,615		40,240	1,020
28	22520	01033	Colbert, AL	624	1,654	54,766	54,597	54,984		51,666	897
29		01035	Conecuh, AL	853	1,080	13,403	13,227	14,089		14,054	2,221
30	10760	01037	Coosa, AL	666	1,522	11,044	11,133	11,881		11,063	2,365
31		01039	Covington, AL	1,044	720	37,234	36,969	37,631		36,478	1,227
32		01041	Crenshaw, AL	611	1,704	13,719	13,598	13,665		13,635	2,189

Figure 15.6: US Census Spreadsheet of County Population. This screenshot captures the display of `c07_tabB1.xlsx` in Excel. The original file was downloaded in 2008 as an **xls** file from `http://blueprod.ssd.census.gov/statab/ccdb/` and converted to a 2007 **xlsx** format. While this spreadsheet is no longer available on the Web, similar files can be found at `http://factfinder2.census.gov`.

for different years. It is often simpler to specify the names manually, rather than trying to extract them programmatically. We will show here how we can examine the content of the sheet to programmatically determine which cells are merged.

The important part of this computation is to determine which titles span multiple columns. These are "merged cells." [1] We can find which groups of cells act as one by looking at the *<mergeCells>* node in the worksheet's *XML* document in the slot *content*. Then for our sheet,

```
m = getNodeSet(sh@content, "//x:mergeCells", "x")
m[[1]]
```

```
<mergeCells count="14">
 <mergeCell ref="H8:H9"/> <mergeCell ref="J8:J9"/>
 <mergeCell ref="I8:I9"/> <mergeCell ref="D8:D9"/>
 <mergeCell ref="E8:E9"/> <mergeCell ref="C6:C9"/>
 <mergeCell ref="B6:B9"/> <mergeCell ref="K8:N8"/>
 <mergeCell ref="A6:A9"/> <mergeCell ref="O8:Q8"/>
 <mergeCell ref="D6:E7"/> <mergeCell ref="F6:Q7"/>
 <mergeCell ref="F8:F9"/> <mergeCell ref="G8:G9"/>
</mergeCells>
```

[1] To create these in Excel, you highlight the cells and then right click and select `"Format cells"`. Within this, we move to the Alignment tab and click OK.

Note that we need to ensure that there is a <mergeCells> node before we subset m as not all work sheet documents will have a <mergeCells> element.

From the node

```
<mergeCell ref="A6:A9"/>
```

we can see that A6, A7, A8, and A9 are merged together. When we access them in the sheet,

```
sh["A6:A9"]

[1] "Metro- politan  area   code" NA
[3] NA                             NA
```

we see that A7–A9 are NA. Similarly, the node

```
<mergeCell ref="D6:E7"/>
```

indicates that the four cells—D6, D7, E6, and E7—are combined; this region in the sheet corresponds to the "Area, 2000 (square miles)" title. The other <mergeCell> nodes locate the middle-level titles and lowest-level titles in the spreadsheet.

A possible next step would be to merge the names together to get variable names. One approach is to repeat the values in each of the merged cells and then collapse these down the columns to get the complete names, e.g., F6 becomes "Population.2006.July.1".

Finding the Footnotes

The footnotes are located at the bottom of the document (see Figure 15.7). They are identified by the string "FOOTNOTES" in the first column. To locate them, we first need to find the cell containing that text. We can do this in various ways. One way is to do this directly with *R* string comparisons. We can get the values of each cell in the first column and find which matches "FOOTNOTES":

```
col1 = sh[,1]
start = which(col1 == "FOOTNOTES") + 1L
```

We can next find the cell in the column that is blank:

```
end = start + which(is.na(col1[-(1:start)]))[1] - 1L
```

Now, we can get the content of each footnote with

```
fn = col1[start:end]
```

However, several of the footnotes span multiple rows which we want to group together. The pattern is that a new footnote starts with a number; otherwise the text is to be combined with the previous line. Unfortunately, this rule is not quite right. The third footnote has three lines and the third of these starts with the text "1990", which is a year, not the start of a new footnote. We can use a regular expression that looks for one or two digits at the start and this would suffice in this circumstance. For example,

```
i = grepl("^[0-9]{1,2} ", fn)
footnotes = as.character(by(fn, cumsum(i), paste, collapse = " "))
```

But, let's look for a more general and robust approach.

If we have to find the footnotes in a more robust manner that avoids the confusion of the "1900", then we can look at the formatting of the footnotes. To be more specific, we look at the *XML* content of the cells containing a footnote. These will have the footnote number in a separate (sub-)node from the actual text so we need to find the *XML* node. Since these cells contain text, and text is stored in a

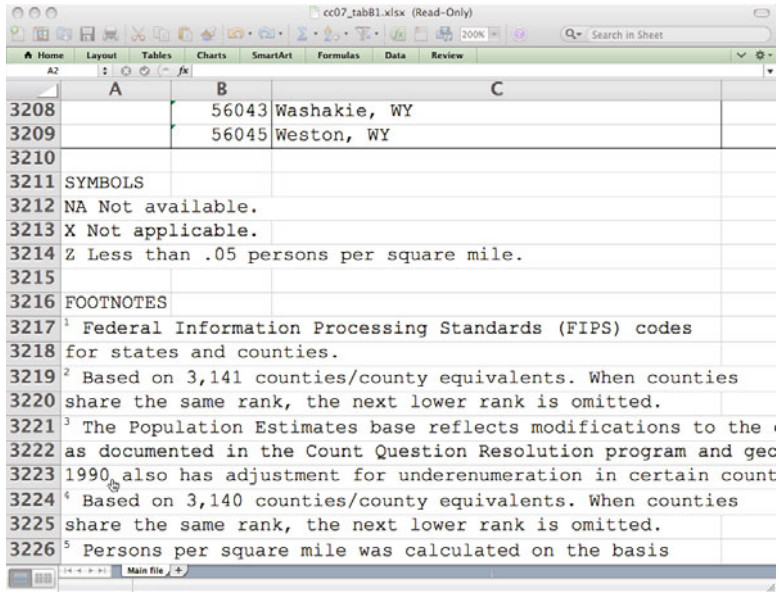


Figure 15.7: Footnotes in a US Census Spreadsheet. This screenshot displays the footnotes that appear at the bottom of spreadsheet `c07_tabB1.xlsx` (see Figure 15.6 for a screenshot of the top of the spreadsheet). Some footnotes occupy more than one row, e.g., the third footnote appears in rows 3221 through 3223.

shared string table, we must first get the cell value and then use it to look up the corresponding *XML* node in the collection of shared strings. To do this, we first collect the identifiers for the cells we want

```
ids = sprintf("A%d", start:end)
fnNodes = sh[ids, asNode = TRUE]
```

Then from the nodes in the sheet, we can find the indices of the shared strings

```
idx = as.integer(sapply(fnNodes, function(x)
                                    xmlValue(x[["v"]]))) + 1L
idx[1]
```

```
368
```

Note that we add 1 to each index since 0-based counting is used in the Excel document. Now we can locate the shared string nodes. To get the value, we index with `idx`. We examine the first one to determine the pattern for which we are looking,

```
ss = getSharedStrings(wb, asNode = TRUE)
ss[idx][1]
```

```
$si
<si>
 <r>
  <t>1</t>
```

```
 </r>
 <r>
  <rPr>
   <sz val="12"/>
   <rFont val="Courier New"/>
   <family val="3"/>
  </rPr>
  <t xml:space="preserve">
Federal Information Processing Standards...</t>
 </r>
</si>
```

We see in this case that there are two <*r*> elements making up the shared string node. The first <*r*> contains a <*t*> element that holds a single number, i.e., the number of the footnote. Basically, any shared string that has two elements is the start of a footnote:

```
which(sapply(ss[idx], xmlSize) > 1)
```

```
si si si si si si si si si
 1  3  5  8 10 12 13 15 17
```

We can collect the strings for the footnotes in the same manner as before with

```
footnotes = as.character(by(sapply(ss[idx], xmlValue),
                            cumsum(sapply(ss[idx], xmlSize) > 1),
                            paste, collapse =  " "))
footnotes = gsub("^[0-9] ", "", footnotes)
```

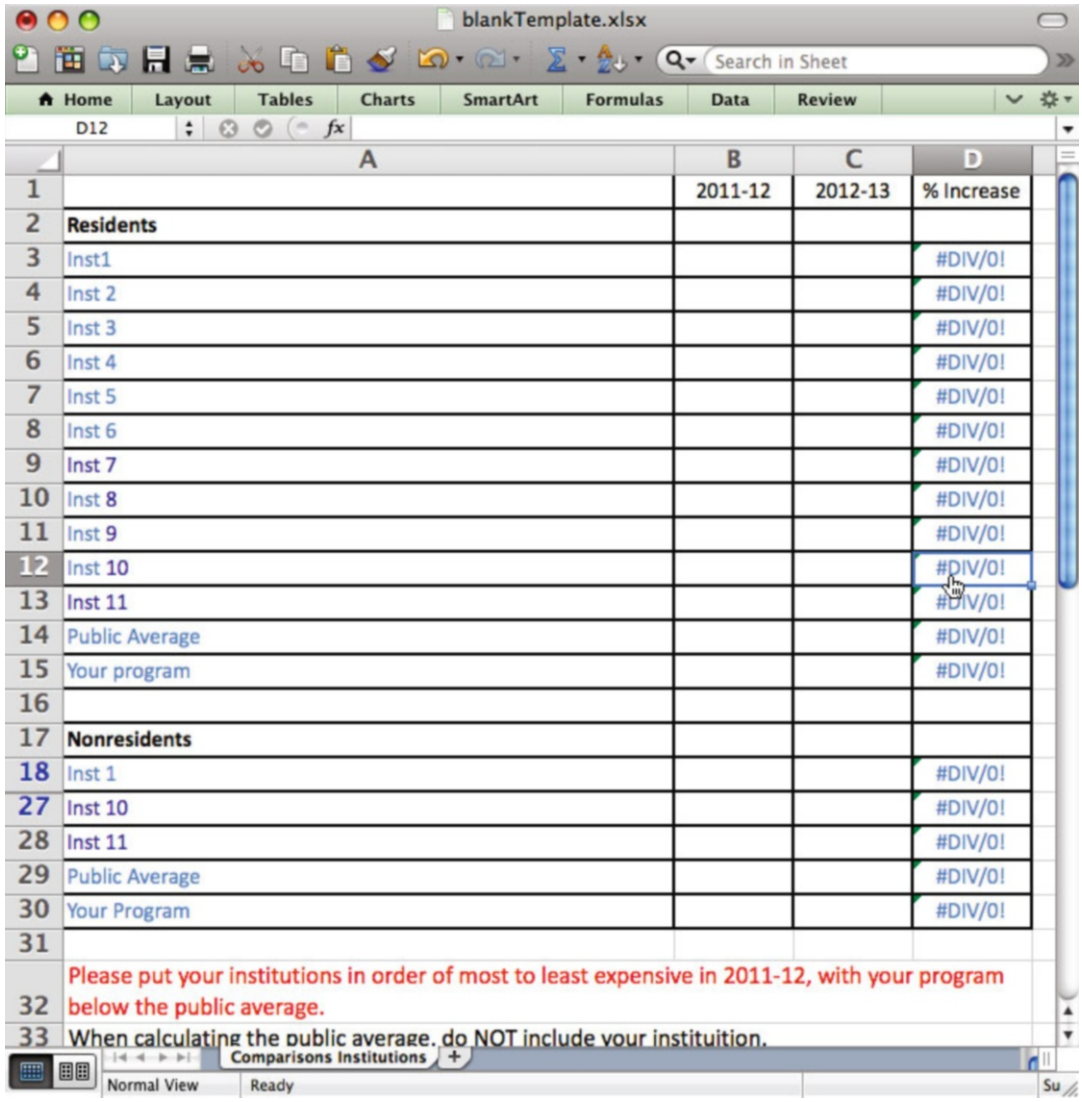
For example, the third footnote that potentially caused problems is

```
footnotes[3]
```

```
[1] "The Population Estimates base reflects ...
 for underenumeration in certain counties and cities."
```

15.6 Creating and Updating Spreadsheets

Spreadsheets can be useful for displaying the results from a data analysis or for organizing data for presentation and interactive display. As an example, when preparing an application for permission to raise tuition for our graduate program, we were provided an Excel template in which to insert the requested comparison data. The template standardized the information and its presentation, making the review process more streamlined and consistent. We want to programmatically read this Excel document and add our data to the existing cells in the worksheet. Moreover, we wish to generate a new workbook that contains our report and leave the template untouched as a backup. We may even want to include an **rda** file containing the source data and a script file with the code that was used to create the new **xlsx** archive. That way, we have a self-contained archive for sharing and for future program updates. We demonstrate how to carry out these various actions with the Excel template shown in Figure 15.8.



	A	B	C	D
1		2011-12	2012-13	% Increase
2	**Residents**			
3	Inst1			#DIV/0!
4	Inst 2			#DIV/0!
5	Inst 3			#DIV/0!
6	Inst 4			#DIV/0!
7	Inst 5			#DIV/0!
8	Inst 6			#DIV/0!
9	Inst 7			#DIV/0!
10	Inst 8			#DIV/0!
11	Inst 9			#DIV/0!
12	Inst 10			#DIV/0!
13	Inst 11			#DIV/0!
14	Public Average			#DIV/0!
15	Your program			#DIV/0!
16				
17	**Nonresidents**			
18	Inst 1			#DIV/0!
27	Inst 10			#DIV/0!
28	Inst 11			#DIV/0!
29	Public Average			#DIV/0!
30	Your Program			#DIV/0!
31				
32	Please put your institutions in order of most to least expensive in 2011-12, with your program below the public average.			
33	When calculating the public average, do NOT include your institution.			

Figure 15.8: Example Report Template in a Spreadsheet. This screenshot shows a spreadsheet called `reportTemplate.xlsx` to be completed for a report. A copy of this template needs to be filled in with each institution's name and tuition. We need to add a formula for the average tuition for the public institutions so that the `"Public Average"` cells will automatically be computed.

15.6.1 Cloning the Excel Document and Entering Cell Values and Formulae

The workbook in Figure 15.8 displays the spreadsheet that we want to fill in by adding data for comparison institutions (stored in a data frame). It is a simple report with four columns, where the rightmost column (`"% increase"`) is calculated by a formula from the cells in columns B and C. We need to fill in the cells in columns A, B, and C. We also want to add formulas to B14 and C14

to compute averages for those rows that correspond to public institutions. (This formula is not part of the original template.)

Briefly, content can be added to an existing worksheet via simple assignments such as `sh[i, j] = "some text"`. The row index `i` can be numeric and the column index can be numeric or the name of a column, e.g., `"B"`, `"AC"`. Alternatively, Excel syntax can be used, e.g., `sh["A3"] = 0`. We can also add more complex objects to the sheet. A vector can be assigned to the worksheet along either a horizontal or vertical axis, e.g., `sh["A2:E2"] = 1:5` sets the cells A2, B2, C2, D2, and E2 to 1, 2, 3, 4, 5, respectively. Similarly, `sh["A2:A6"] = 1:5` sets the cells A2, A3, ..., A6. For a two-dimensional object on the right-hand side of the assignment, a rectangular collection of cells is populated. For example, `sh["C6:E7"] = matrix(1:6, nrow = 3)` sets C6, D6, E6 to 1, 3, 5, respectively and C7, D7, E7 to 2, 4, 6, respectively. When we insert data, we have the choice of updating only the *XML* document in memory, or we can also add the updated *XML* document to the **xslx** archive. This is controlled through the `update` attribute, which takes a logical value indicating whether to update the archive or not.

Example 15-5 Generating an Excel Report from a Template

Rather than change the template document, we create a clone for our actual report and modify that. The function excelDoc() creates an Excel document, when the name of the **xlsx** file that we pass to it is not found and the *create* argument is `TRUE`. In this case, the function creates a new, generic document that has one empty worksheet. However, if we also provide the name of a spreadsheet in the function's *template* argument, then a copy of that spreadsheet is used as the new document. For our report, we clone the report template and then access the worksheet we want to fill in as follows:

```
report = excelDoc("myReport.xlsx", create = TRUE,
                  template = "reportTemplate.xlsx")
reportWB = workbook(report)
sh = getSheet(reportWB, which = 1)[[1]]
```

Next we load the comparison data that will be added to the spreadsheet to create the report:

```
load("comparisons.rda")
head(comparisons)
```

```
  Institution Pr  Time Res11 Res12 NonRes11 NonRes12
1      U Penn  Y 2.000 59503 62478    59503    62478
2         USC  Y 1.500 43871 46065    43871    46065
3     Cornell  Y 1.500 43304 45469    43304    45469
4         CMU  Y 1.500 37185 39044    37185    39044
5    Stanford  Y 1.125 43953 46151    43953    46151
6    Columbia  Y 1.000 44180 46389    44180    46389
```

In the report, we want to list the institutions in the order of their 2011 resident tuition (in `Res11`). We determine this ordering and pull out Berkeley's values, as follows:

```
comparisons = comparisons[ order(comparisons$Res11,
                                 decreasing=TRUE), ]
whichB =  which(comparisons$Institution == "Berkeley")
berkeley = comparisons[whichB, ]
comparisons = comparisons[- whichB, ]
```

We begin by filling in column A with institution name,

```
sh[3:13, 1] = comparisons$Institution
sh[15, 1] = berkeley$Institution
```

Next we add the tuition values:

```
sh["B3:B13"] = comparisons$Res11
sh["C3:C13"] = comparisons$Res12
sh["B15:C15"] = berkeley[1, c("Res11", "Res12")]
```

The last step is to add the formulas that compute averages for the public institutions. Different from putting in values, the formula allows the recipient of the spreadsheet to update the data and recalculate.

The function excelFormula() takes a formula as a string. Below, we determine the rows in the worksheet to be averaged, and construct the formula as a string,

```
whichPu = which(comparisons$Pr == "N")
formulaStr = paste0("=AVERAGE(",  paste("B", (2 + whichPu),
                                        sep = "", collapse = ","),
                    ")")
```

Now that we have constructed the formula, we call excelFormula() to assign it to the appropriate cell in the worksheet:

```
sh["B14"] = excelFormula(formulaStr)
```

We similarly construct the formula for cell C4. The same additions can be made to the nonresident part of the worksheet, which we also do not show here.

The changes that we have made to the spreadsheet have all been made in memory. The actual **xlsx** archive has not been updated because the default assignment is to update the parsed document and to not write the changes to the archive. Our final step then is to update the **zip** file. We call update(), passing it the worksheet:

```
update(sh)
```

```
Zip Archive: myReport.xlsx
 [1] "[Content_Types].xml"        "_rels/.rels" ...
 [9] "docProps/core.xml"          "xl/calcChain.xml"
[11] "docProps/app.xml"
```

15.6.2 Working with Styles

In addition to adding content to a worksheet, we can also format the content for display. We can do this for cells or groups of cells and rows and columns by explicitly specifying the appearance of each cell. However, it is best to use an extra layer of abstraction where we assign one or more cells a style that we have separately defined. Then, if we want to update the cell's appearance, we need only make changes to the style definition to have all the associated cells change their appearance. This is one of the benefits of using centralized styles.

We can either define a new style with createStyle() or access the styles in the Excel archive with getStyles(). Also, getDocStyles() allows us to work directly with the styles in the *XML* document,

rather than using the *R* representations that getStyles() returns. Once a style has been defined, we can use setCellStyle() to assign that style to a cell. We show how in the next example.

Example 15-6 Adding Styles to Cells for an Excel Report

To make it easier to distinguish between public and private universities in the report created in Example 15-5 (page 522), we use different color fonts for the institution name: green for public and blue for private. We create two new styles for this purpose as follows:

```
ft = Font(sz=12L, face="b")
newStPu = createStyle(sh, font = ft, fg = "00FF00")
newStPr = createStyle(sh, font = ft, fg = "0000FF")
```

Next we associate each style with the appropriate rows in the worksheet. The variables `puRows` and `prRows` determine which rows in the worksheet correspond to the public and private institutions. We determined them based on the *R* variable `Pr` in `comparisons`. We assign the two styles to the cells as follows:

```
instPu = paste("A", puRows, sep = "")
instPr = paste("A", prRows, sep = "")
instNames = cells(sh)
setCellStyle(instNames[instPu], newStPu)
setCellStyle(instNames[instPr], newStPr)
```

Since we did not supply the *update* argument when we called setCellStyle(), we have taken the default action for this function. This is to update the archive with each modification so there is no need to call update() at this point.

15.6.3 Inserting Other Content into the Archive

We have seen that the archive is a collection of files, and we also can store additional files in the archive. The spreadsheet applications will ignore them, but carry them around in the archive for us to use. Take our report for example. We can add to the archive an **rda** file that contains the data frame that was used to produce the report. This addition makes the archive a self-contained document. The data frame contains all the necessary information to fill in and style the cells, so by storing it in the archive, the archive has the data needed to reproduce or update the report.

The RExcelXML package uses the Rcompression package to access the files within the **xlsx** archive. We can use it for the same purpose, e.g., to access and add an extra file to an archive. In the following example, we demonstrate two ways to do this. The first uses the excelDoc() function and `[` available in RExcelXML. The second approach directly uses the zipArchive() function in Rcompression; the function on which the excelDoc() relies.

*Example 15-7 Adding an **rda** File to an Excel Archive*

We want to add the serialized data frame `comparisons` to our report `myReport.xlsx`, which was created in Example 15-5 (page 522). We begin by opening the archive with the following call to excelDoc():

```
report = excelDoc("myReport.xlsx")
```

Then we serialize the `comparisons` object to a raw vector and add it to `report` with:

```
report[["comparisons.rda"]] = serialize(comparisons, NULL)
```

We can check to see that the **rda** file is indeed in the archive:

```
length(report)
```

```
[1] 12
```

```
names(report)[12]
```

```
[1] "comparisons.rda"
```

Alternatively, we can simply use the zipArchive() function in the Rcompression package to add the serialized data to the Excel archive with

```
library(Rcompression)
z = zipArchive("myReport.xlsx")
z[["comparisons.rda"]] = serialize(comparisons, NULL)
```

Finally, to restore the data frame, we extract it from the archive as a raw vector and use unserialize() as shown here:

```
rw = report[["comparisons.rda", mode = "raw"]]
head(unserialize(rw))
```

```
    Institution Pr  Time Res11 Res12 NonRes11 NonRes12
1        U Penn  Y 2.000 59503 62478    59503    62478
2           USC  Y 1.500 43871 46065    43871    46065
3       Cornell  Y 1.500 43304 45469    43304    45469
4           CMU  Y 1.500 37185 39044    37185    39044
5      Stanford  Y 1.125 43953 46151    43953    46151
6      Columbia  Y 1.000 44180 46389    44180    46389
```

15.7 Using Relationship and Association Information in the Archive

When we add a worksheet to a workbook or an image or chart to a worksheet, the relationships between files in the archive change. This can be complicated and error-prone so if we want to build functions to make these additions, we need an understanding of the relationships between files in the archive. For example, the `workbook.xml` file keeps track of the worksheets, global settings, and other shared components of the workbook. However, neither the worksheets nor their filenames are embedded in this file. There is an extra layer of indirection used to connect `sheet1.xml` to `workbook1.xml`. A **rels** file in the archive is used to tie the pieces of the workbook together, amongst other relationships. Specifically, `xl/_rels/workbook.xml.rels` contains information to connect the worksheets to the workbook.

We provide two examples in this section demonstrating how to add a worksheet to a workbook and an image to a worksheet. These use different **rels** files in the archive. We use these examples to outline

our approach of leveraging *XML* tools to parse and build *XML* files for applications. The functions addWorksheet() and addImage() formalize these examples into more general functionality. We note that these functions are not polished; they merely offer a starting point for this functionality. The first example explains the basic approach needed for working directly with *XML* files in the Office Open *XML* archive, in contrast to other approaches that interface with a toolkit from another language, e.g., *Java*, for providing access from within *R* to the contents of a **xlsx** file.

We saw in Section 15.3, that `workbook1.xml` has a `<sheets>` element with two `<sheet>` children, one for each worksheet in the document. Each of these `<sheet>` elements has a *name* attribute that provides the label to display on the tab in the Excel user interface, `"Sheet1"` and `"Images"`, respectively, and each has a *sheetId* attribute, which is used for ordering the sheets in the display. The `<sheet>` node does not contain the file name that holds the sheet contents, e.g., `sheet1.xml`; instead, the relationship identifier, *r:id* on `<sheet>`, locates this file.

For example, for the sheet labeled `"Images"`, the *r:id* is `"rId2"`. We look in the file called `xl/_rels/workbook.xml.rels` (shown below) for the `<Relationship>` element that has an *Id* attribute value of `"rId2"`. The value of this element's *Target* attribute provides the file name for the sheet. In this case, it is `xl/worksheets/sheet2.xml` (file names are relative to xl/).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Relationships xmlns="http://.../relationships">
 <Relationship Type="http://...calcChain"
    Id="rId6" Target="calcChain.xml"/>
 <Relationship Type="http://...styles"
    Id="rId4" Target="styles.xml"/>
 <Relationship Type="http://...worksheet"
    Id="rId1" Target="worksheets/sheet1.xml"/>
 <Relationship Type="http://...worksheet"
    Id="rId2" Target="worksheets/sheet2.xml"/>
 <Relationship Type="http://...theme"
    Id="rId3"Target="theme/theme1.xml"/>
 <Relationship  Type="http://...sharedStrings"
    Id="rId5" Target="sharedStrings.xml"/>
</Relationships>
```

Figure 15.9 provides a diagram of the layout of the part of the archive that contains the information about the `"Images"` sheet. In addition to the **rels** and `workbook1.xml` files, the file called `[Content_Types].xml` provides information on the kind of content in each of the files in the archive, and `app.xml` within `docProps/` has information pertaining to the layout of the GUI, e.g., it indicates the spreadsheet has two pages.

Example 15-8 Adding a Worksheet to a Workbook

In this example we demonstrate how to add a worksheet to a spreadsheet, including updating the subsidiary *XML* and **rels** files in the archive. Our basic approach is to use a template worksheet, i.e., a blank `sheet.xml` that has the structure defined by Office Open *XML* for sheets. In addition to adding this worksheet to the archive, we must also update the auxiliary files as shown in Figure 15.9. That is, we follow the steps below.

- Add a `<Relationship>` element to `workbook.xml.rels`. This element has a unique identifier in its *Id* attribute and file name of the worksheet in its *Target* attribute.
- Add a `<sheet>` element to `workbook.xml`, which has an *r:id* that matches the identifier in *Id* on the newly created `<Relationship>` node in the **rels** file.

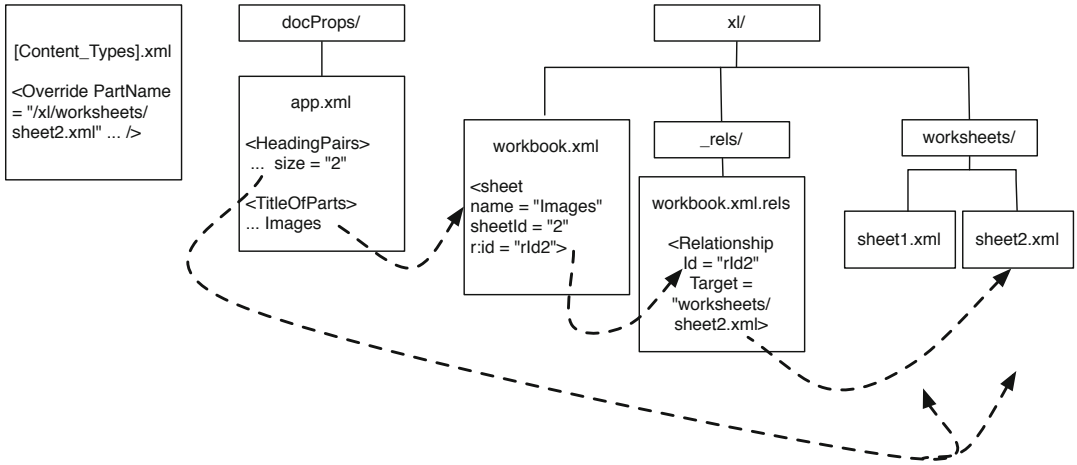


Figure 15.9: Diagram of the Interconnections between Files in an Excel Archive. This diagram shows the connections between various *XML* and **rels** files in the Excel Archive. For example, rather than `workbook1.xml` containing the names of the worksheet files, e.g., `worksheets/sheet2.xml`, it contains a reference to a <*Relationship*> node in a **rels** file that holds the worksheet file name.

- Update `app.xml` to indicate the new *size* of the workbook and related properties.
- Update `[Content_Types].xml` with information about the content of the new worksheet file.
- Insert a generic worksheet into the archive with the file name given in the *Target* attribute of the new <*Relationship*> element.

We carry out each of these steps here. After accessing the archive, we determine how many sheets are already in the archive. The following code does this and then creates the new sheet's file name based on this information:

```
ed = excelDoc("testAddWS.xlsx")
num = length(grep("xl/worksheets/sheet.*\\.xml", names(ed))) + 1
filename = sprintf("xl/worksheets/sheet%d.xml", num)
```

We must enter this file name into the **rels** file, but first we need to determine a unique identifier to associate with the file name. We extract all <*Relationship*> nodes from the **rels** file and construct a unique identifier based on the number of relationships already present. Here we use tools in XML, e.g., xpathSApply() to parse and identify node sets:

```
rels = ed[["xl/_rels/workbook.xml.rels"]]

schemas = "http://schemas.../relationships"
namespace = c(x = schemas)
eids = xpathSApply(rels, "//x:Relationship", xmlGetAttr, "Id",
                   namespaces = namespace)
relId = sprintf("rId%d", length(eids) + 1L)
relId

[1] "rId7"
```

Now that we have constructed the unique id and file name for the new worksheet, we can add a *<Relationship>* node to the **rels** document with this information. We use newXMLNode() in XML to do this as follows:

```
newXMLNode(
 "Relationship",
 attrs = c(Id = relId,
           Type = "http://schemas.../worksheet",
           Target = sprintf("worksheets/%s", basename(filename))),
 parent =   xmlRoot(rels))
```

```
<Relationship Id="rId7"
  Type="http://schemas.../worksheet"
  Target="worksheets/sheet3.xml"/>
```

Next, we update the `workbook.xml` file to include a *<sheet>* node that points to the *<Relationship>* element we just added to the **rels** file. For convenience, we again use `num`, the number of sheets, in the sheet title. We do this with

```
name = paste("New Sheet", num)
main = "http://schemas.../spreadsheetml/2006/main"
wbook = ed[["xl/workbook.xml"]]
sheets = getNodeSet(wbook, "//x:sheets", c(x = main))
newXMLNode("sheet",
           attrs = c(name = name, sheetId = num, "r:id" = relId),
           parent = sheets[[1]])
```

```
<sheet name="New Sheet 3" sheetId="3" r:id="rId7"/>
```

The last files that need updating are `app.xml` and `[Content_Types].xml`. We do not provide the code here but instead refer the reader to the addWorksheet() function in RExcelXML for these final details.

This approach has been generalized and encapsulated into the function addWorksheet(). It takes the Excel archive and sheet title as arguments. The following call performs the task of adding a new sheet to a workbook for us:

```
ed = excelDoc("testAddWS.xlsx")
addWorksheet(ed, name="New Sheet")
```

Functions similar to addWorksheet() have been developed to add image files and Excel charts to worksheets. As with the relationship between worksheets and workbooks, the *XML* describing the image or chart is not directly contained or referred to in the worksheet. Relationship files are used to connect the image to the sheet that displays it. Figure 15.10 provides a visual representation of the relationships for the image displayed in `"Sheet1"` of the worksheet shown in Figure 15.4.

These relationships are slightly more complex than those for sheets because they involve two files—the PNG file containing the actual image and an *XML* file called `xl/drawings/drawing1.xml`, which holds information about the placement, size, and shape of the images in the sheet. Following is a snippet of this file:

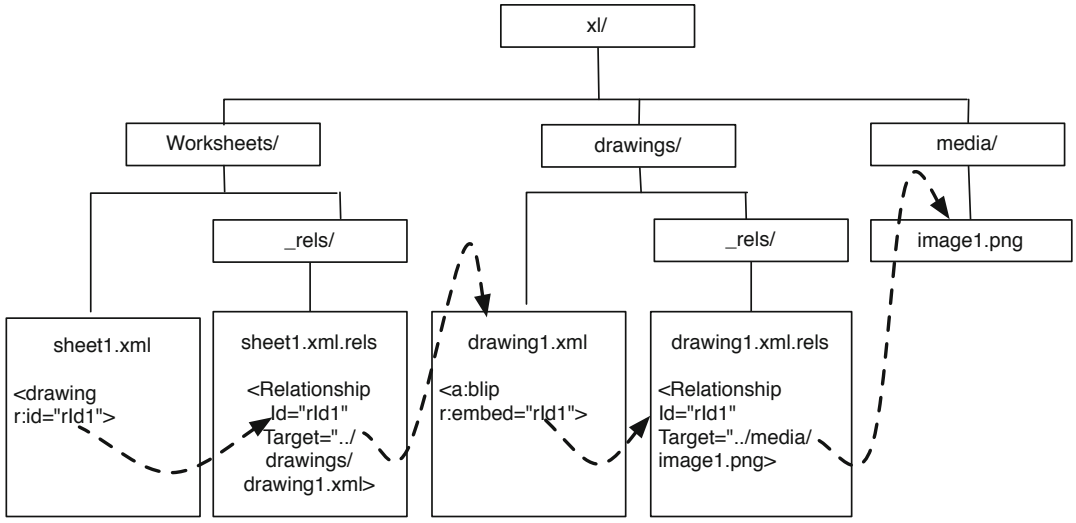


Figure 15.10: Diagram of File Relationships for an Image in a Worksheet. The hierarchy displayed here shows the connections between the files pertaining to the image in the worksheet shown in Figure 15.4. For example, rather than `sheet1.xml` containing the name of the drawing file (`drawings/drawing1.xml`), it contains a reference to a *<Relationship>* that holds the target file name. Similarly, the name of the image file is determined via the relationship in `drawings/_rels/drawing1.xml.rels` that connects `drawing1.xml` to `image1.png`.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xdr:wsDr xmlns:xdr="http://...spreadsheetDrawing"
    xmlns:a="http://...drawingml/2006/main">
 <xdr:twoCellAnchor editAs="oneCell">
  <xdr:from>
   <xdr:col>2</xdr:col> <xdr:colOff>114300</xdr:colOff>
   <xdr:row>4</xdr:row> <xdr:rowOff>139700</xdr:rowOff>
  </xdr:from>
  <xdr:to>
   <xdr:col>3</xdr:col> <xdr:colOff>177800</xdr:colOff>
   <xdr:row>13</xdr:row> <xdr:rowOff>12700</xdr:rowOff>
  </xdr:to>
  <xdr:pic>
   ...
   <xdr:blipFill>
    <a:blip xmlns:r="http://...relationships" r:embed="rId1">
...
```

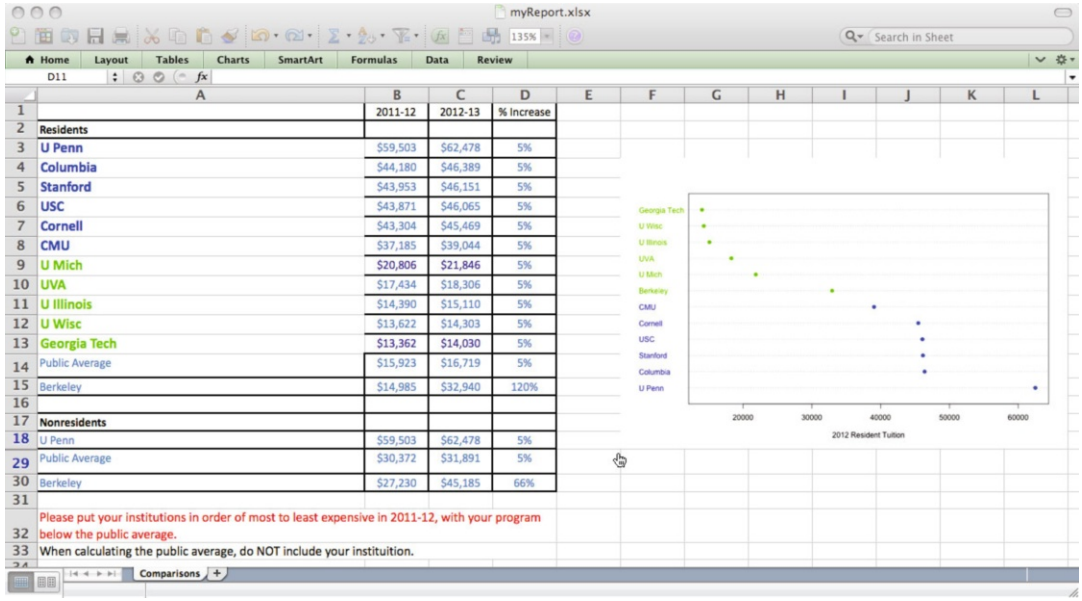
The *<twoCellAnchor>* element contains *<from>*, *<to>*, and *<pic>* elements. The *<from>* and *<to>* elements provide the location of the image. These have *<col>*, *<colOff>*, *<row>*, and *<rowOff>* children, where *<col>* and *<row>* are the (zero-based) indices of the column and row. The *<colOff>* and *<rowOff>* elements are in English metric units (EMU) and they are offsets measured from the edge of the specified row or column. (An EMU is 1/360,000 of a centimeter.) The

<pic> element holds the information about the image file. The <blip> node contains a relationship *ID* that leads to the the file name. Again, rather than explicitly containing the file name, this element refers to an identifier, which we look up in the **rels** file associated with this document, i.e., `xl/drawings/_rels/drawing1.xml.rels`, to obtain the name of the image file: `xl/media/image1.png`.

The addImage() function in RExcelXML handles the creation and editing of these various *XML* files much the same way as we did when we added a sheet to a workbook in Example 15-8 (page 526). We next provide an example of how to use this intermediate-level function.

Example 15-9 Adding an Image to a Worksheet

We continue with the report created in Example 15-5 (page 522) and add an *R* plot of the data to the worksheet. While the format of the table in the report was prescribed by the template, we wanted to add an alternative visual comparison of the tuition values. In this example, we create a dot plot in *R* as a PNG file and add that image to the worksheet (see Figure 15.11).



	A	B	C	D
1		2011-12	2012-13	% Increase
2	Residents			
3	U Penn	$59,503	$62,478	5%
4	Columbia	$44,180	$46,389	5%
5	Stanford	$43,953	$46,151	5%
6	USC	$43,871	$46,065	5%
7	Cornell	$43,304	$45,469	5%
8	CMU	$37,185	$39,044	5%
9	U Mich	$20,806	$21,846	5%
10	UVA	$17,434	$18,306	5%
11	U Illinois	$14,390	$15,110	5%
12	U Wisc	$13,622	$14,303	5%
13	Georgia Tech	$13,362	$14,030	5%
14	Public Average	$15,923	$16,719	5%
15	Berkeley	$14,985	$32,940	120%
16				
17	Nonresidents			
18	U Penn	$59,503	$62,478	5%
29	Public Average	$30,372	$31,891	5%
30	Berkeley	$27,230	$45,185	66%
31				
32	Please put your institutions in order of most to least expensive in 2011-12, with your program below the public average.			
33	When calculating the public average, do NOT include your instituition.			

Figure 15.11: Completed Excel Report Template with Dot Plot. This screenshot shows a completed report. Starting from the blank template shown in Figure 15.8, we filled in cells with values from a data frame, added styles to some of the cells, inserted formulae into B14 and C14 to compute averages, and included a dot plot made in *R*. Also, "hidden" in the **xlsx** archive is the **rda** file used to complete the report.

We use the data from `comparisons` to create the bar chart. The colors of the dots match the text color used in the table to distinguish private and public institutions.

```
cI = comparisons[order(comparisons$Res12, decreasing=TRUE), ]
png("TuitionDotChart.png", width=720)
dotchart(cI$Res12, pch=19, col=c("green","blue")[cI$Pr],
         labels = cI$Institution, xlab = "2012 Resident Tuition")
dev.off()
```

The dot plot has been saved as a PNG file, which we now add to the worksheet.

We read the Excel archive and access the spreadsheet with the usual commands:

```
reportWB = workbook("myReport.xlsx")
sh = getSheet(reportWB, 1)[[1]]
```

The addImage() function handles all of the details required to add an image to a worksheet. It updates the appropriate **rels** files to make the connection between the worksheet and the image; creates a `drawing.xml` file containing the image-specific information such as where to place the image in the sheet; and inserts the PNG file in the archive. The function takes as input, the worksheet in which to place the image, the name of the image file, and the top left and lower right corners where the image will be placed.

```
addImage(sh, "TuitionDotChart.png", from = c(4, 6), to = c(25, 12))
```

The function also has an *update* argument with a default value of `TRUE` so the archive has been updated with this new information.

15.8 Google Docs and Open Office Spreadsheets

We have primarily focused in this chapter on Excel spreadsheets; however, the same functionality can be developed for other spreadsheet software that follow the Office Open *XML* standard. The *R* packages RGoogleDocs and ROpenOffice are starting points for handling Google Docs and Open Office files, respectively. These packages are not as complete as RExcelXML, and the authors invite contributions that will flesh out the functionality. The following example, highlights the similarities and differences in functionality between the addWorksheet() functions in RExcelXML and RGoogleDocs.

Example 15-10 Inserting a Worksheet into a Google Docs Spreadsheet

Google Docs are dynamic documents that are accessible via the Internet. They can be shared with others and edited simultaneously. They need to be accessed with an Internet connection, login, and password. This access can be handled programmatically within *R*. The following two-step process is described in more detail in Example 10-5. Briefly, we first authenticate by shipping our user id and password with the following call to getGoogleAuth():

```
auth = getGoogleAuth("deb.nolan@gmail.com", "my password", "wise")
```

Note that to access a spreadsheet, we must request "wise" service from Google. Once authenticated, we get the connection with the following call to the getGoogleDocsConnection() function:

```
con = getGoogleDocsConnection(auth)
```

Now that we have established a connection, we use the getDocs() function to retrieve a list of available documents from the Google Docs account. Given the "wise" constraint, the call below only retrieves the spreadsheets from the account:

```
ssdocs = getDocs(con)
```

Although we needed application-specific code to first access the GoogleDocs spreadsheet, after we obtain that access, the functions to operate on the spreadsheet are essentially identical to those

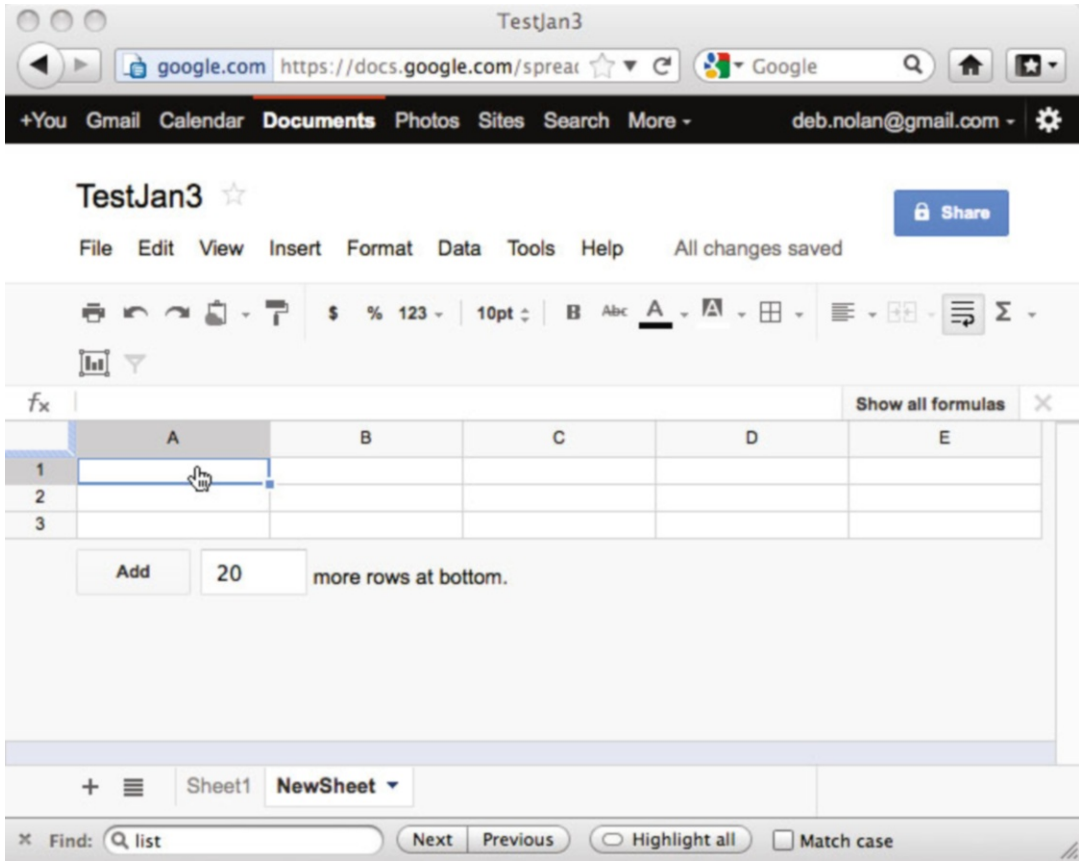


Figure 15.12: Example Google Docs Spreadsheet. This screenshot shows a Google Docs spreadsheet called `TestJan3`. It has two sheets, `"Sheet1"` and `"NewSheet"`. The latter was added to the spreadsheet via a call to addWorksheet() in the RGoogleDocs package.

in RExcelXML. For example, we can add a worksheet to a spreadsheet by calling the function addWorksheet(), which has a signature very similar to addWorksheet() in RExcelXML:

```
addWorksheet(ssdocs[[1]], con, title = "NewSheet")
```

One key difference is that we must pass the connection with each of our function calls. The new worksheet is shown in Figure 15.12.

15.9 Possible Enhancements and Extensions

Adding Charts to Worksheets

The *XML* documents for charts can be reused and shared among spreadsheet, presentation, and word processing applications. Currently, RExcelXML has basic functionality to add a bar chart to a spread-

sheet. Similar to addWorksheet(), the addChart() function uses a template *XML* document as a starting point. The function adds this template to the Excel archive and handles updating the relationships between files. The addChart() function is a proof of concept, and can be extended to include other chart types.

Other *OOXML*-Formatted Office Suites
As mentioned in Section 15.8, the *R* packages for handling Google Docs (RGoogleDocs) and Open Office documents (ROpenOffice) are in the preliminary stages of development. The approach described in this chapter with RExcelXML can be extended in a straightforward manner to these other implementations of the Office Open *XML* standard. The authors invite contributions that will flesh out the functionality of these packages.

Word Processing and Presentation Documents
Again, while the focus in this chapter has been on creating and manipulating Excel documents, Office Open *XML* includes mark-up for word processing documents (*WordprocessingML*) and presentations (*PresentationML*) in addition to *SpreadsheetML*. The same general approach to using the XML package with *XML* files carries over to creating and modifying Microsoft Word and presentation files, e.g., RWordXML offers a starting point for creating this interface.

15.10 Summary of Functions in RExcelXML

Understanding the *SpreadsheetML* vocabulary and structure helps us develop functionality that provides easy access to cell values, formula, and styles. These functions offer a convenient means for working with various parts of the **xlsx** archive. This approach enables the programmer to extend and customize the interface to the spreadsheet, workbook, and Excel document. Many of these functions are demonstrated via examples in Sections 15.4 and 15.5. We provide below brief summaries of the functions available in RExcelXML.

read.xlsx() Retrieve the contents of the worksheets in an **xlsx** document. The contents are represented as a list of data frames, one for each worksheet. This function extracts the contents for simple worksheets (*which* specifies which sheets in the workbook to extract) and can skip rows (*skip*), convert a header to variable names (*header*), and automatically identify the class of the content of each column.

excelDoc() Retrieve the ZIP archive of an **xlsx** document. This function provides access to all of the files within the archive. It can also create (when *create* is `TRUE`) a new Excel document object from a template (*template*), which can be supplied in the function call.

workbook() Create a `Workbook` object from an `excelDoc` or a file name. The `Workbook` provides access to its worksheets through its `[[` and `[` methods. The worksheets are `Worksheet` objects. See `[[` and `[` function descriptions below for how to access and set cells in the worksheet.

getSheet() Retrieve worksheets from a `Workbook` object or an `ExcelArchive` object. The return value is a list of `Worksheet` objects. The contents of the worksheet can be accessed with `[` and cells().

cells() Return a list of all *XML* cell nodes from a `Worksheet` object. An *XPath* expression can be provided (in the *xquery* parameter) to restrict the set of nodes returned.

`[[` and `[` Subset worksheets in a workbook and cells in a worksheet. The cells can be subsetted via Excel-style or *R*-style indexing or a mixture of both, e.g., `sheet["A3:B4"]`, `sheet[3:4, 1:2]` and `sheet[3:4, c("A", "B")]` are all equivalent. The *update* argument controls whether the **zip** archive or the *XML* file only will be modified, e.g.,

```
sheet["A3:B4", update = FALSE] = 1:4
```

does not update the archive.

update() Update the contents to an **xlsx** file (the **zip** archive) by adding or overwriting existing entries. For efficiency, many of the functions that create and change the contents of a workbook have an *update* argument to control when the archive is updated. This includes, `[[` and `[`.

getSharedStrings() Retrieve as a character vector the shared strings of a workbook or worksheet.

getStyles() Retrieve the shared styles for a workbook or worksheet. The return value will be an *R* representation of the styles.

getDocStyles() Retrieve the *XML* `<styleSheet>` for the `ExcelArchive`. The return value is an `XMLInternalDocument` rather than an *R* representation of the styles.

createStyle() Define a new style for a cell. The foreground color (*fg*), fill color (*fill*), font (*font*), alignment (*halign* and *valign*), and border (*border*), among other styles can be specified.

setCellStyle() Set the style for a cell using an existing style.

excelFormula() Construct from a string an Excel formula that can be added to a cell.

addWorksheet() Add an empty worksheet to an Excel archive with the title provided in *name*.

addImage() Add an image file, e.g., PNG, *PDF*, to a region of a worksheet. Specify the extent of the image with *from* and *to*.

addChart() Construct an Excel chart and add it to a region of a worksheet.

15.11 Further Reading

The official documentation for OOXML is voluminous, but Part 3 of the documentation provides a primer with examples and diagrams [4]. Also, [11] provides a brief overview to the documentation and a discussion of the goals and properties of the standard. Reference [26] offers a brief introduction to the basics of OOXML.

References

[1] Apache Software Foundation. OpenOffice: The free and open productivity suite; 3.0 New Features. http://www.openoffice.org/dev_docs/features/3.0/, 2011.

[2] Apple, Inc. Numbers for iOS: Supported file formats. http://support.apple.com/kb/HT4642, 2011.

[3] Adrian Dragulescu. xlsx: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files. http://cran.r-project.org/package=xlsx, 2011. *R* package version 0.5.0.

[4] ECMA International. Ecma Office Open *XML* file formats standard, Part 3: Primer. http://www.ecma-international.org/news/TC45_current_work/TC45_available_docs.htm, 2011.

[5] Federal Election Commission. Top 50 house incumbents by contributions from PACs and other committees, January 1, 2011 – June 30, 2011. http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx, 2011.

[6] KDE e.V. KOffice: Standards-compliant office and productivity applications. http://userbase.kde.org/KOffice, 2011.

[7] LibreOffice; The Document Foundation. Calc: The LibreOffice spreadsheet program. http://www.libreoffice.org/features/calc/, 2011.

[8] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2000 and Excel XP. *Computational Statistics & Data Analysis*, 40:713–721, 2002.

[9] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52:4570–4578, 2008.

[10] Eric Neuwirth. RExcel: Interface between *R* and Excel. http://cran.r-project.org/package=RExcel, 2011. *R* package version 3.2.6.

[11] Tom Ngo. Office Open *XML* overview. http://www.ecma-international.org/news/TC45_current_work/OpenXMLWhitePaper.pdf, 2005.

[12] *R* Core Team. *R Data Import/Export*, 2012. http://cran.r-project.org/doc/manuals/R-data.html.

[13] Frank Rice. Introducing the Office (2007) Open *XML* file formats. http://msdn.microsoft.com/en-us/library/aa338205(v=office.12).aspx, 2006.

[14] Brian Ripley. RODBC: ODBC database access. http://cran.r-project.org/package=RODBC, 2011. *R* package version 1.3-3.

[15] Marc Schwartz. WriteXLS: Cross-platform *PERL*-based *R* function to create Excel 2003 (XLS) files . http://cran.r-project.org/package=WriteXLS, 2011. *R* package version 2.3.0.

[16] Miria Solutions. XLConnect: Manipulate Excel files from *R*. http://cran.r-project.org/package=XLConnect, 2011. *R* package version 0.2-3.

[17] Hans-Peter Suter. xlsReadWrite: Natively read and write Excel files. http://cran.r-project.org/package=xlsReadWrite, 2011. *R* package version 1.5-4.

[18] Duncan Temple Lang. RExcelXML: Tools for working with Excel *XML* documents. http://www.omegahat.org/RExcelXML, 2011. *R* package version 0.5-0.

[19] Duncan Temple Lang. ROOXML: Simple tools for Open Office *XML* documents. http://www.omegahat.org/ROOXML, 2011.

[20] Duncan Temple Lang. ROpenOffice: Basic reading of Open Office spreadsheets and workbooks. http://www.omegahat.org/ROpenOffice, 2011. *R* package version 0.4-1.

[21] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[22] Duncan Temple Lang. Rcompression: In-memory decompression for GNU **zip** and bzip2 formats. http://www.omegahat.org/Rcompression, 2012. *R* package version 0.94-0.

[23] Duncan Temple Lang. RGoogleDocs: Primitive interface to Google Documents from *R*. http://www.omegahat.org/RGoogleDocs, 2012. *R* package version 0.7-0.

[24] Duncan Temple Lang and Gabriel Becker. RWordXML: Tools for Open Office word processing *XML* documents. http://www.omegahat.org/RWordXML, 2010. *R* package version 0.1-0.

[25] Guido van Steen. dataframes2xls: Write data frames to xls files. http://cran.r-project.org/package=dataframes2xls, 2011. *R* package version 0.4.5.

[26] Wouter van Vugt. Open *XML*: The markup explained. http://openxmldeveloper.org/blog/b/openxmldeveloper/archive/2007/08/13/1970.aspx, 2007.

[27] Gregory Warnes. gdata: Various *R* programming tools for data manipulation. http://cran.r-project.org/package=gdata, 2011. *R* package version 2.12.0.

[28] World Bank Group. WDR2011 dataset. http://databank.worldbank.org/databank/download/WDR2011Dataset.xlsx, 2011.

[29] World Bank Group. World development report 2011 on conflict, security and development. http://data.worldbank.org/data-catalog/wdr2011, 2011.

Chapter 16
Scalable Vector Graphics

Abstract This chapter explores a powerful two-dimensional graphics format named *SVG* (scalable vector graphics), which is becoming widely used to represent object-based, vector (nonraster) graphics both on the Web and elsewhere. In addition to displaying scalable plots, the *XML*-based *SVG* format provides functionality for interactivity, animation, and a number of filtering effects that allow us to create rich graphical displays, and also to integrate them with Web pages. *R* can create static *SVG* displays, and we can post-process them within *R* as *XML* documents to add interaction and animation. We show how to provide simple interaction and animation with high-level *R* functions, and also how to add more customized interaction using *R* commands that use *JavaScript* at viewing time. We also discuss how to use both *SVG* and *HTML* to create rich "applications" and mash-ups in the Web browser.

16.1 Introduction: What Is *SVG*?

Graphical representations of data have significantly changed in recent years, where viewers expect to interact with a plot on the Web by clicking on it to get more information, produce a different view, or control an animation. Scalable Vector Graphics (*SVG*) [9] offers these capabilities; it is an *XML* format for describing two-dimensional (2D) graphical displays that also supports interactivity, animation, and filters for special effects.

Given that *SVG* is an *XML* grammar, *SVG* images can be constructed "from scratch" using the *XML* writing facilities described in Chapter 6. However, it is also possible to use *R*'s plotting facilities to create high-level, high-quality *SVG* graphics through the *SVG* graphics device (svg()) and the cairo rendering engine [7, 38]. Once created, we can post-process these *SVG* plots to add interactivity, such as hyperlinks, tool tips, linked plots, and arbitrary *JavaScript* code [10, 35] for handling events on user controls, e.g., sliders, buttons, and animation. This approach takes advantage of the standard plotting functions in *R* (including lattice and the traditional grz-model plotting functions) and leaves us to augment the resulting *SVG* plots, which we can accomplish through the *XML* parsing and writing facilities in the XML package [32].

A powerful aspect of *SVG* is that we can combine it with *JavaScript* to provide interaction and animation programmatically during the rendering of the *SVG* rather than declaratively through *SVG* elements. While this means using two programming languages, we note that the *SVG* plots are being displayed in a very different medium and use a language (*JavaScript*) that is widely used for Web content. These *JavaScript* additions can make the plot interactive for the viewer to control in a variety

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_16,

of ways, e.g., linking points across scatter plots, animating a time series, and adding/removing subsets of data from a plot.

Similar to JPEG and PNG files, *SVG* documents can be included in *HTML* documents; they can also be in-lined within *HTML* content. Many commonly used Web browsers directly support *SVG* (Chrome, Firefox, Opera, Safari), and there is a plug-in for Internet Explorer. For example, Firefox can act as a rendering engine that interprets the vector description and draws the objects in the display within the *HTML* page. Quite differently from other image formats, *SVG* graphics, and their subelements, remain interactive when displayed within an *HTML* document and can participate as components in applications that interact with other graphics and *HTML* components. There are also non-Web-browser viewers for *SVG* such as Inkscape (`http://www.inkscape.org/`) and Squiggle (based on Apache's Batik [2] `http://xmlgraphics.apache.org/batik/tools/browser.html`). Furthermore, *SVG* can be included in *PDF* documents via an *XML*-based page description language named Formatting Objects (*FO*) [23], which is used for high-quality typesetting of *XML* documents. For these reasons, *SVG* is a rich and viable alternative to the ubiquitous PNG and JPEG formats. While *SVG* may not be the most ideal approach to interactive graphics, it has many advantages that come from its simplicity and increasing use on the Web and in publishing generally.

Another advantage of *SVG* is that it uses a vector-based system that describes an image as a series of geometric shapes, in contrast to a raster representation that uses a rectangular array of pixels (picture elements) that represents what appears at each location in the display. That is, an *SVG* document includes the commands (in an *XML* format) to draw shapes at specific sets of coordinates, and these shapes are infinitely scalable because they are created from vector descriptions. The viewer can adjust and change the display, e.g., to zoom in and refocus, and maintain a clear picture.

In this chapter, we demonstrate some of the high-level functions for working with *SVG* documents to create interactive, animated displays from within *R* via the SVGAnnotation package [21]. As in other chapters, we first describe high-level functionality developed in SVGAnnotation that handles many of the common cases. In particular, we provide functionality to add tool tips and hyperlinks on plot components for many of the plots produced by the graphics functions in *R*. Next we delve deeper into the *SVG* grammar and some of the facilities it provides, and then demonstrate how we can use the package's intermediate-level functions, which offer access to the components of the plot—e.g., point, axis, label, and title—in order to create arbitrary customized interactive graphical displays. The examples include linking points across scatter plots, controlling a spline smoother with a slider, animating points in a scatter plot, highlighting subsets of points using a choice menu, and using hyperlinks on a map to select *HTML* tables for display. Before describing these high-level and intermediate-level functions, we discuss the computational model behind them.

16.1.1 A Model for Adding Interactivity to SVG Plots

Our model for creating interactive plots uses the built-in *R* plotting tools to create the initial graphical display. (We plot to the *SVG* device available in *R*). Given the *SVG* tree, we augment (or annotate) the plot to include additional *SVG* elements and attributes and, possibly, *JavaScript* code. Since *SVG* is a grammar of *XML*, the graphical display created by the *SVG* graphics device in *R* is highly structured. This means it is relatively easy to examine and modify the *SVG* document programmatically. We use knowledge of the document structure to identify the various elements of an *SVG* plot/document that correspond to the components of the plot, e.g., point, axis label, title. Once identified, we augment the *SVG* document with information that enables interactivity and/or animation. In other words, there

are two distinct steps to creating the interactive displays. The first uses *R*'s plotting functions to create the base plot and the second uses the tools available in the XML package to modify the *SVG* produced in the first step. This augmented document is then written to a text file and opened in an *SVG* viewer. *R* is not available from within the viewer. The plot's interactivity is made possible by the *SVG* elements/attributes and *JavaScript* code that have been added to the document.

One of the features of this approach is that we can post-process the output of an existing *R* graphics device to provide interactivity and animation. We do not have to replace the device with our own to intercept *R* graphics operations and assemble the results within the device. This is made possible because of the *XML* structure of the generated *SVG*. It is not nearly as simple to provide interactivity with binary formats such as *PDF* or rasterized formats such as PNG and JPEG.

The post-processing approach does, however, give rise to a potential problem. Since we post-process the output from the svg() function and associated device, we are exploiting a format and structure that may change. There are two layers of software, each of which may change. The first is the implementation of the *SVG* device in *R*. The second is the third-party C-level `libcairo` library on which the *R* graphics device is based. Since the generic *R* graphics engine and also the device structure are well-defined and very stable, it is unlikely that there will be significant changes to the format of the *SVG* generated by *R*-related code. Changes to `libcairo` are more likely to cause changes to the *SVG*. Very old versions of `libcairo` (e.g., 1.2.4) do yield *SVG* documents that are nontrivially different from more recent versions (e.g., version 1.10.0). Future versions of `libcairo` *may* introduce new changes to the format and cause issues for SVGAnnotation. We do not, however, expect significant changes.

In many regards, the approach we have taken in designing SVGAnnotation is intended to cause minimal disruption to the existing tool chain. We do not require the user to deploy a different graphics device. We do not modify the code of the existing graphics device. Instead, we work to make sense of the output (e.g., identify shapes) rather than knowing the graphical operations, e.g., circle, polygon, rectangle, text. This introduces the perils of a change in the output structure, but is a worthwhile goal of direct software reuse. It synchronizes the annotation facilities to the primary *SVG* graphics device in use within *R*. The alternative is to provide our own device and generate the *SVG* content ourselves and control its format. This would protect us from changes in the format. However, we would not benefit from passively incorporating enhancements to the *R* graphics device or `libcairo`. To address this issue, we could use a "proxy" device that acts as a "front" for the regular *SVG* device. This device would identify the different *R*-level components and then forward the device calls to the existing *SVG* `libcairo`-based graphics device. This would help us to map the high-level *R* components of the graphical displays to the lower-level *SVG* content.

The SVGAnnotation package provides high-level functions for "standard" plots in which we can readily identify the *R* elements. It also allows an *R* programmer to use intermediate-level functions to operate on axes, legends, etc. There are also low-level facilities for identifying the shape of visual elements, e.g., polygon, line, vertical line, text. The functions in the package can also add high-level type identifiers to nodes in the *SVG* document that, for example, identify data points, axes, labels, titles, and frames. These facilities allow us to handle cases from regular *R* graphics, `lattice` [28], `ggplot2` [37], and `grid` [17, 19]. By leveraging the *XML* facilities in *R*, SVGAnnotation offers capabilities for creating rich new plots. Most importantly, the approach and the concept it implements is readily extended to other contexts and types of plots. The abstract idea and approach is the main contribution of the package.

Post-processing of *SVG* Plots

The higher-level functions in SVGAnnotation make it quite easy to add different forms of interactivity to an *SVG* display. To do this, we follow a three-step process.

- **Plotting Stage**
  Create the base plot in *R* as an *SVG* document. That is, open an *SVG* graphics device and plot to it using *R*'s plotting functions. The svgPlot() function in SVGAnnotation does this and also adds the plotting commands to the document and returns the parsed *XML* document (or writes it to a file).
- **Annotation Stage**
  Analyze and modify the *SVG* file created in the plotting stage, i.e., add *SVG* content, and possibly *JavaScript* code, to the document to make the display interactive or animated. Depending on the type of plot to be annotated and the kind of annotations desired, the high-level functions in SVGAnnotation may be able to identify the graphical elements and handle all aspects of these annotations. Otherwise, intermediate-level functions can be used to locate and amend the relevant plotting elements. The basic approach is to use the XML package to parse, examine, and modify the *SVG* document from within *R*. Once the annotation is completed, we save the *SVG* document to a file.
- **Viewing Stage**
  Load the annotated *SVG* document into an *SVG* viewer, such as Opera, Firefox, Safari, or Squiggle, where the reader views and interacts with the image. In this stage, *R* is not available. The interactivity is generated by *SVG* elements themselves and/or *JavaScript* code that has been added to the document in the annotation stage.

16.1.2 Other Approaches to Making Interactive SVG Plots in R

The approach we have taken to add interactive features to graphical displays uses the rendering engine provided by `libcairo` from within *R*. There are two other *SVG* graphics devices available for use within *R*. These are found in the `RSvgDevice` [14] and the derived `RSVGTipsDevice` [24] packages. The former generates an *SVG* document that contains *SVG* elements corresponding to the graphical primitives the *R* graphics engine emits, e.g., lines, rectangles, circles, text. The `libcairo` system is more widely used and more robust, and `libcairo` also deals with text in more advanced and sophisticated ways (specifically drawing letters as infinitely scalable shapes/paths rather than using font sets). This is the primary reason we use the `libcairo` approach even though the format of the *SVG* documents that `RSvgDevice` produces is simpler and more direct.

The `RSVGTipsDevice` package builds on the code from `RSvgDevice`. It allows *R* programmers to add *SVG* annotations and does so when the *SVG* is being created, rather than via post-processing additions. *R* users can set text for tool tips and URLs for hyperlinks that are applied to the next graphical primitive that the *R* graphics engine emits. This works well when the *R* programmer is directly creating in their own code all graphical elements of a display. However, it does not work when calling existing plotting functions, each of which creates many graphical elements in a single call. The computational model does not provide the appropriate resolution for associating an annotation with a particular element, but just the next one that will be created. As a result, we cannot annotate just the vertical axis label when calling a function that creates an entire plot.

The post-processing approach we present is not limited to using `libcairo`. We can also use `RSVGTipsDevice` to generate the *SVG* content and then programmatically manipulate that. The documents generated by the two different devices will be somewhat different (e.g., the use of groups of characters for strings, inline *CSS* styles versus separate classes, *SVG* primitives for circles rather than paths). However, because the elements of the graphical display were generated by the *R* graphics engine with the same *R* expressions, the order of the primitive elements and the general structure will be very similar.

The package `gridSVG` [18] is an earlier and quite different approach to taking advantage of *SVG*. As the name suggests, it is focused on *R*'s grid graphics system [16]. For this reason, it cannot work with displays created with the traditional and original graphics model, but it does handle any graphics produced via grid such as all of the plot types in `lattice` and `ggplot2`. The package `gridSVG` takes advantage of the structured self-describing information contained in grid's graphical objects. As one creates the grid display, information about the locations and shapes is stored as objects in *R*. These can then be translated to *SVG* without using the *R* graphics device system. New graphics primitives have been added to the grid system to add hyperlinks, animations, etc., corresponding to the facilities provided in *SVG*.

The approach provided by `gridSVG` is quite rich and more direct than SVGAnnotation. It removes the need to post-process the graphics that we presented here. Instead, one proactively and explicitly specifies graphical concepts. One limitation that is similar to that of `RSVGTipsDevice` is that we still run into problems with interleaving *SVG* annotations. While a user can issue grid commands that add *SVG* facilities to the output, higher-level functions that create plots produce entire sequences/hierarchies of grid objects in a single operation. If these do not add the desired annotations, we need to post-process the grid hierarchy to add them. This post-processing would be very similar to what we are doing in SVGAnnotation; however, it would be done on *R* objects.

The `imagemap` package [26] offers another approach to creating interactivity within *R* graphical displays. The basic concept is that a plot is created in *R* in the usual manner. Then, the creator specifies different regions within the plot that correspond to areas of interest to viewers, e.g., the axes labels, points in a scatter plot, bars in a histogram. These regions define an image map that can be displayed within an *HTML* document. The creator specifies actions in the form of *JavaScript* code that are triggered as the viewer moves the mouse over the different regions or clicks within a region. This approach does not map the elements of the plot, such as a title, hexagonal bin, or state polygon, to mouse events. It instead creates an overlay for the image from separately identified regions. The elements in the image cannot be programmatically changed at viewing time, e.g., we cannot change the color of a line. We also cannot move elements to achieve animation effects.

The `animation` package [39] provides functionality to create movies (e.g., animated **gif** files or **mpeg** movies) entirely within *R*, using some additional external software. The idea is that one draws a sequence of plots in *R*. These are combined and then displayed at regular intervals as frames of the movie to give the appearance of motion. Although animated displays are common to both the `animation` and SVGAnnotation packages, the goals and infrastructure are very different. Being able to work at the level of graphical objects within the plot is richer and more efficient for many applications.

The interactivity and animation described here are very different from what is available in more commonly used statistical graphics systems such as GGobi [30], iPlots [34], or Mondrian [33]. Each of these provide an interactive environment for the purpose of exploratory visualization and are intended primarily for use in the data analysis phase. Instead, *SVG* is mostly used for creating presentation graphics that typically come at the end of the data analysis stage. Rather than providing general interactive features for data exploration, we use *SVG* to create application-specific interactivity, including graphical interfaces for a display.

16.2 Simple Forms of Interactivity

Tool tips and hyperlinks are very simple forms of interactivity that are easily created with *SVG*. With tool tips, when we rest the mouse on a portion of the plot, say an axis label or an observation in a scatter plot, a small window pops up with additional information. Figure 16.1 shows an example where the mouse is on a point in a scatter plot and a tool tip has appeared that provides a more detailed description for that particular observation. This interactivity is available in any *SVG*-supported browser and does not require an *R* or *JavaScript* "plug-in".

Our first step in creating this plot (or any *SVG* plot that we want to make interactive) is to open an *SVG* graphics device with a call to the svg() function, issue *R* plotting commands to the *SVG* device (as with any other graphics device), and close the device. For example,

```
svg("quakes.svg")
plot(lat ~ lon, data = quakes)
title("Fiji Region Earthquakes")
dev.off()
```

creates the file named `quakes.svg`, which contains the *SVG* that renders a scatter plot of the locations of earthquakes. The SVGAnnotation package provides a convenience function, svgPlot(), for plotting to the *SVG* device. Specifically, svgPlot() opens the device, evaluates the code to create the plot(s), and then closes the device, all in a single function call. For example,

```
svgPlot({ plot(lat ~ lon, data = quakes)
          title("Fiji Region Earthquakes")
        }, "quakes.svg")
```

performs the same function calls as the prior code. An advantage to using svgPlot() is that it also inserts the *R* code that generated the plot as metadata into the *SVG* file and provides provenance and reflection information. This can be convenient for post-processing the resulting *SVG*. For example, for lattice plots, svgPlot() adds the number of panels, strips, conditioning variables, levels, and details about the legend. This extra information allows one to more easily and reliably identify the *SVG* elements that correspond to the components of the *R* plot(s) to be annotated.

An additional reason for using the svgPlot() function is that it can return the *SVG* document for further processing, instead of saving the *SVG* to a file and requiring the user to parse the *SVG* document in order to annotate it. We often want the *SVG* document generated by *R*'s graphics commands as an *XML* tree and not written to a file so if no file name is specified by the caller, then the return value of svgPlot() is a tree structure, e.g.,

```
doc = svgPlot({ plot(lat ~ lon, data = quakes)
                title("Fiji Region Earthquakes") })
```

With this *SVG* tree structure in `doc`, we can modify the plot to, e.g., add a `<title>` node to each point where the text content of `<title>` will be displayed as a tool tip when the mouse hovers over the point. We show how to do this in the following example.

Example 16-1 Adding Tool Tips and Hyperlinks to an SVG Plot of Earthquakes

We use code in [27] to make the plot of `quakes` shown in Figure 16.1. The only change we have made to this code is to embed the call to plot() in a call to svgPlot(). We do this as follows:

```
depth.col = gray.colors(100)[cut(quakes$depth, 100, label=FALSE)]
depth.ord = rev(order(quakes$depth))
doc = svgPlot(plot(lat ~ long, data = quakes[depth.ord, ],
```

```
        pch = 19, col = depth.col[depth.ord],
        xlab = "Longitude", ylab="Latitude",
        main = "Fiji Region Earthquakes"))
```

Since we did not specify a value for the *file* parameter of svgPlot(), the function returns the parsed *XML/SVG* tree, which we can then post-process and enhance.

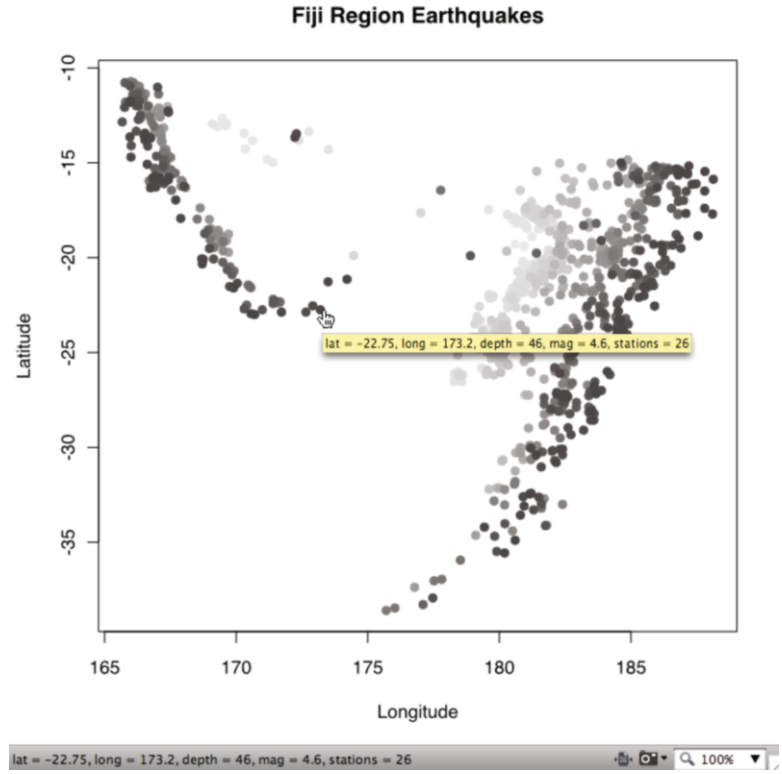


Figure 16.1: Tool Tips and Hyperlinks on an *SVG* Scatter Plot of Earthquakes. This screenshot shows a scatter plot where each point has a tool tip on it. The tool tips were added using the addToolTips() function in SVGAnnotation. When viewed in a browser, as the mouse hovers over the point, information about the values of each variable for that observation is provided in a pop-up window. When the viewer moves the mouse away from the point, the pop-up window disappears. In addition, the title of the plot ("Fiji Region Earthquakes") has a hyperlink associated with it, so when the viewer clicks on the title, the browser visits the target URL.

The default operation for the addToolTips() function is to add text to each of the points in a scatter plot, which the *SVG* viewer will display when requested. We simply pass the *SVG* document to addToolTips() along with a vector of strings, one for each point in the plot. For example, adding the row names from the data frame as tool tips for the points is as simple as

```
addToolTips(doc, rownames(quakes[depth.ord, ]))
```

If we want the tool tip to provide values for each of the variables in our data frame, we can do this by creating the string for each row of the data frame and then passing this character vector to addToolTips(), e.g.,

```
addToolTips(doc, apply(quakes[depth.ord, ], 1,
                       function(x)
                          paste(names(quakes), x,
                                sep = " = ", collapse = ", ")))
```

We can also use addToolTips() to provide tool tips on axis labels. By default, addToolTips() adds the text to the data points in the plot, so now we tell addToolTips() to annotate the labels by passing it the particular axis-label nodes to be annotated rather than the entire *SVG* document. We find these axis-label nodes using another function in SVGAnnotation, getAxesLabelNodes(). This function locates the nodes in the *SVG* document that correspond to the title and axes labels:

```
ax = getAxesLabelNodes(doc)
```

This call gets more than just the axes labels but also the title, so we discard the title node, and call addToolTips() to place tool tips on just the axes labels:

```
addToolTips(ax[c("xaxis", "yaxis")],
            c("Degrees east of the prime meridean",
              "Degrees south of the equator"), addArea = TRUE)
```

The *addArea* argument places a bounding box around the text in the label so that this bounding region (not just the letters themselves) are responsive to mouse activity.

The addToolTips() function also has an `addCSS` parameter. This indicates the inclusion of a cascading style sheet file into the *SVG* document to control the appearance of elements. We can add the default style sheet available in the package or specify our own *CSS* document using a reference to a document or inlining the content in the *SVG*. If the argument is `TRUE`, then the default *CSS* is added. The default value for *addCSS* is `NA`, and in this case, the function determines whether or not the *CSS* is needed. In our situation, since a tool tip is to be placed on a rectangular area surrounding the text of an axis label (i.e., `addArea = TRUE`), a *CSS* will be added. We also can add a specific *CSS* file via a call to addCSS().

We sometimes want to click on a phrase or an individual point in a plot and have the browser jump to a different view or a Web page. *SVG* supports hyperlinks on elements, and we can readily add this feature to *R* plots. To demonstrate, we add a hyperlink to the plot title so that when a viewer clicks on the phrase "Fiji Region Earthquakes", the Web browser displays the USGS Web page containing a map of recent earthquakes in the South Pacific, i.e., `http://earthquake.usgs.gov/.../recenteqsww/Maps/region/S_Pacific.php`.

We have already noted that the title of the plot is in the first element of the return value from the call to getAxesLabelNodes(), which we saved in the *R* variable `ax`. We also saved the *URL* in `usgsUrl`. To add a hyperlink to the title, we simply associate the target *URL* with this element in the document via a call to addAxesLinks() as follows:

```
addAxesLinks(ax$title, usgsUrl)
```

Now that we have completed our annotations of the *SVG*, we save the modified document:

```
saveXML(doc, "quakes_tips.svg")
```

This document can then be opened in an *SVG* viewer (e.g., a Web browser) and the user can interact with it by mousing over the points and axes labels to read the tool tips and visit the link that we have provided.

Currently, many, but not all, common plots in *R* can be annotated with tool tips and hyperlinks using addToolTips(), addLink(), and addAxesLinks(). For example, hyperlinks can be added to a map so that when the viewer clicks on a particular region in the plot, the browser opens the referenced Web page. The following state map is created in *SVG* with each state filled with an appropriate color:

```
doc = svgPlot(map('state', fill = TRUE, col = stateColors))
```

We can use getPlotPoints(), one of the functions in the SVGAnnotation for handling *SVG* elements, to find the polygonal regions for the states:

```
polygonPaths = getPlotPoints(doc)
```

We then add hyperlinks to each polygon with a call to addLink() with

```
addLink(polygonPaths, urls, css = FALSE)
```

Note that if we do not fill the polygons with color, then the interiors are not explicitly drawn, and therefore only the boundaries (i.e., the paths) are active. This means that the interiors of the states do not respond to a mouse click. This may seem strange, but is actually a feature of *SVG*. A similar approach can be used to add tool tips to the regions of a hexagonal bin plot [8]. Suppose we want to add a tool tip to each hexagonal region to show its actual cell count. We proceed to call hexbin() and not create the plot but save the return value, which is an *S4* object,

```
library("hexbin")
hbin = hexbin(quakes$long, quakes$lat)
doc = svgPlot(plot(hbin))
```

The `count` slot in the `hbin` object contains a vector of cell counts for all of the bins, which we can use as tool tips. A call to getPlotPoints() locates the hexagonal regions within the *SVG* document:

```
ptz = getPlotPoints(doc)
```

Now we add tool tips to these regions in the plot with

```
tips = paste("Count: ", hbin@count)
addToolTips(ptz, tips, addArea = TRUE)
```

16.3 The Essentials of *SVG*

The previous section introduced the basic functionality for accessing elements of simple plots, e.g., getAxesLabelNodes() and getPlotPoints(), and functions to enhance those nodes, e.g., addLink() and addToolTips(). For more complex plots, we may have to find the graphical elements ourselves so in this section we look at the structure of *SVG* to understand the landscape. To do more advanced things, i.e., adding different *SVG* features on nodes or finding the graphical elements, we need to know more about the structure of *SVG*.

In this section, we provide a brief overview of the most common elements in *SVG* and the basics of the drawing model. For more detailed information on *SVG*, readers should consult [9, 36]. Figure 16.2 provides a simple *SVG* image that uses many common *SVG* elements. This simple document was created manually (not with *R*). The image contains a gray rectangle with a red border, two pairs of green and pink circles, a JPEG image of a pointer, a path that draws the perimeter of the state of Oregon and fills the resulting polygon with blue, and the text `"Oregon"`. Below the figure is the corresponding *SVG* and a brief description of various nodes and attributes of the document.

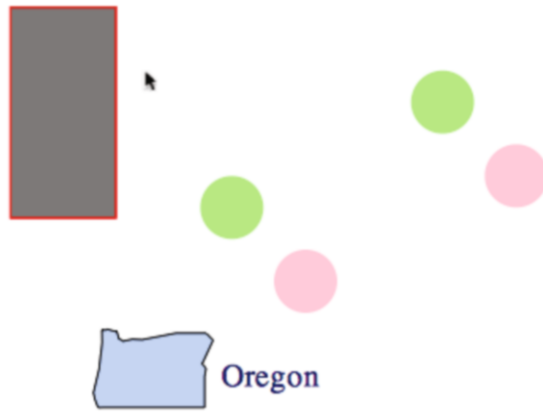


Figure 16.2: Example *SVG* Document. This simple *SVG* document displays different shapes and *SVG* objects.

```
<svg xmlns = "http://www.w3.org/2000/svg"  1
 xmlns:xlink = "http://www.w3.org/1999/xlink"
 width = "300pt" height = "300pt"  2
 viewBox = "0 0 300 300" version = "1.1">  3
 <defs>  4
  <g id="cId">  5
   <circle id = "greencirc" cx = "15" cy="15"  6
      r = "15" fill = "lightgreen"/>
   <circle id = "pinkcirc" cx= "50" cy="50"
      r = "15" fill = "pink"/>
  </g>
  <style type="text/css">  7
   <![CDATA[
 .recs fill: rgb(50%, 50%, 50%); fill-opacity: 1; stroke: red; ]]>
  </style>
 </defs>
 <g id = "main">  8
  <rect x = "10" y = "20" width = "50"  9
     height = "100" class = "recs"/>
  <use  x = "100" y = "100" xlink:href = "#cId"/>  10
  <use  x = "200" y = "50" xlink:href = "#cId" />
  <image xlink:href = "examples/pointer.jpg"  11
     x = "70" y = "50" width = "10" height = "10"/>
  <path style = "fill: rgb(100, 149, 237);  12, 13
     fill-opacity: 0.5;stroke-width: 0.75;
     stroke-linecap: round; stroke-linejoin: round;
     stroke: rgb(0%,0%,0%); stroke-opacity: 1;"
     d = "M 102.8 174.6 L 102.8 174.6  14
           ... L 88.5 174.6 L 102.8 174.6 Z"
```

```
      id = "oregon"/>
  <text x = "110" y = "200" fill = "navy"
    font-size = "15">  15
      Oregon
  </text>
 </g>
</svg>
```

1 The root of an *SVG* document is <svg>. Possible elements it can have are: a <title> node that contains the text to be displayed in the title bar of the *SVG* viewer; a <desc> node that holds a description of the document; and other tags for grouping and drawing elements.

2 The *width* and *height* attributes of <svg> specify the size of the canvas, which can be measured in points (pt), pixels (px), inches (in), centimeters (cm), etc.

3 The *viewBox* attribute of <svg> provides the minimum x and y and the width and height, respectively, which establishes the coordinate system for all descendant elements. Note: the top left of the viewBox is at $(0,0)$.

4 The <defs> node is a container for *SVG* elements that are defined and given a label, but not immediately put in the *SVG* display. These definitions can be augmented, displayed, and reused within the *SVG* display through a reference to the element's unique identifier. The <defs> element acts as a dictionary of template elements.

5 We define a pair of circles, one green and the other pink. These are combined together into a single group using the <g> element and identified by "cId". Since these circles are in the <defs> element, they are not drawn until used within the *SVG* document.

6 This <circle> element will render a circle when used in the *SVG* document (see the <use> element below). The circle is located at (15, 15) relative to the position were it is used. The radius is also 15. The circle is filled with a "lightgreen" color. *SVG* supports specifying colors as names or in RGBA format.

7 We can provide styles in <defs> for controlling the appearance of elements with cascading style sheets. The approaches for doing this are described in Section 16.6.3.

8 This <g> node is not within a <defs> node so the elements within it are drawn. The use of <g> is helpful when, e.g., we want to treat the collection of objects as a single object in order to transform it as a unit, or place the same appearance characteristics on each element in the collection. The style placed on <g> will apply to all of its subelements, although any element in <g> can override particular attributes. Grouped elements can be defined and then inserted multiple times in the document. It is also possible to nest other <svg> elements within a <g> element, and so create compositions reusing previously and separately created displays.

9 Instructions to draw basic shapes are provided via the <line>, <rect>, <circle>, <ellipse>, and <polygon> tags. Here we have an instruction to draw a rectangle with its upper left corner at (10, 20), a width of 50, and height of 100. The size of the rectangle is specified in the coordinates of the *viewBox*. This coordinate system places the smallest values at the upper left corner of the canvas and the maximum values for x and y at the lower right corner. The *class* attribute contains style information about the color of the interior of the rectangle and its border. Here we use the style called "recs" that was defined earlier in the <style> node in <defs>.

10 The pair of circles defined in the `"cId"` element in `<defs>` is rendered via this `<use>` tag at the location (100, 100) in the viewBox. [Actually, the center of the first circle will be at (115, 115) as its relative location is (15, 15).] As with *HTML*, we specify the reference to an internal element (or "anchor") by prefixing the name/id with a `'#'`, e.g., `'#cId'`. This suggests that we can link to elements in other files, and indeed in *SVG* we have the full power of *XLink* [29], which is a general linking facility for any *XML* vocabulary.

11 Images in, e.g., PNG or JPEG formats can be included in the *SVG* display. Here we place an arrow at (70, 50) in the viewBox and specify the width and height as well. The link to the image is specified via *XLink* using the `xlink` namespace prefix as defined in the *SVG* element.

12 The `<path>` tag provides information to draw a "curve". It contains instructions for the placement of a "pen" on a canvas and the movement of the pen from one point to the next in a connect-the-dots manner. The instructions for drawing the path are specified via a character string that is provided in its *d* attribute. This has a reasonably rich syntax. See callout 14 below for more details.

13 Styles can be placed inline on elements as well as via *CSS*. Here we specify the fill color and style for the curve being drawn. See Section 16.6.3 for more details on specifying styles in *SVG*.

14 The path for the Oregon border is provided in the *d* attribute of this `<path>` node. The instructions begin by picking up the pen and moving it to the starting position (102.8, 174.6). This starting point is given either as an absolute position (`"M x,y"`) or a relative position (`"m x,y"`), i.e., the capitalization of the letter determines whether the position is relative (m) or absolute (M). Either a comma or blank space can be used to separate the *x* and *y* coordinates. From the starting point, the pen draws a line segment to the next point, and so on. The segment may be a straight line (L or l) or quadratic (Q or q) or cubic (C or c) Bezier curve. The final instruction `"Z"` closes the path by drawing a line segment from the pen's current position back to the starting point. These paths provide very succinct notation for drawing curves.

15 The contents of `<text>` is placed at (110, 200) in the viewBox and the other attributes specify its appearance. The `libcairo` rendering engine (and hence the svg() device in *R*) uses `<path>` nodes for rendering all shapes including characters and text (see Section 16.6.1). One benefit to this approach of using a `<path>` command for drawing letters is that there is no reliance on fonts when the *SVG* document is viewed. It also means that scaling the *SVG* preserves the shape of the letters with very high accuracy. However, when we add text and shapes to an *SVG* document, we may want to use the simpler higher-level shortcut tags, e.g., `<text>` and `<circle>`.

Later in this chapter (see Section 16.6.1) we delve more deeply into *SVG* and the basic structure of an *SVG* document created by the *SVG* device in *R*. In particular, we examine the *SVG* document produced from a simple call to plot(). If we wish to annotate other types of plots, i.e., one that is not covered by the high-level functions in SVGAnnotation, such as a ternary plot, then it can be helpful to understand the structure of the documents produced by the *SVG* device. Also in that section, we provide mode details on how to use styles to customize the *SVG* document, and we describe the *SVG* elements that create animations.

16.4 General Interactivity on *SVG* Elements via *JavaScript*

The examples shown in Section 16.2 have added *SVG* elements and attributes to create simple interactive effects such as tool tips and hyperlinks. These are examples of how the *SVG* viewer reacts to user events. Other types of plots, in addition to scatter plots, hexbin plots, and maps, can be modified

in this same way. We can place hyperlinks and tool tips on, e.g., mosaic plots, boxplots, lattice plots, etc. In this section, we create more complex forms of interactivity with the help of *JavaScript*. For example, the linkPlots() function in SVGAnnotation adds *JavaScript* code to an *SVG* plot to respond to mouseover events on points in a scatter plot to highlight observations that are linked across plots. That is, with *JavaScript* we can make points in a scatter plot change color in response to a mouse event. In general, whether it is changing a point's color, visibility, location, or size, mouse actions can initiate the modification of *SVG* dynamically, thus enabling a very rich set of user interactions with a plot. In this section, we demonstrate how functions such as linkPlots() post-process an *SVG* plot to add calls to *JavaScript* functions so that when a mouse event occurs these functions modify the attributes on elements in the *SVG* document to change their appearance. To do this, we also augment these *SVG* graphical elements with unique identifiers so that they are easily identified by the *JavaScript* functions.

General Approach to Using *JavaScript* to Add Interactivity to *SVG* Documents.

Our approach can be summarized by the following set of tasks performed in the annotation stage of document creation (see Section 16.1.1).

- Within *R*, add unique identifiers to the relevant graphics elements in the *SVG* document so they can be retrieved easily in the viewing stage via the *JavaScript* function *getElementById()*.
- Create special attributes on the graphics elements to store the default values of settings that are subject to change, e.g., *original-style*. These attribute names are made up by the developer, and should not conflict with *SVG* attribute names.
- Add attributes, such as *onmouseover*, *onmouseout*, and *onclick*, to the elements that are supposed to respond to mouse actions. The values of these attributes are *JavaScript* function calls. Our philosophy is to pass all element-specific information needed to respond to a request/mouse-event in the call. This way, the *JavaScript* functions can be used in multiple situations, e.g., with other data and other types of plots, and not rely on auxiliary global variables.
- Embed in the *SVG* document the *JavaScript* functions that respond to the mouse activity, e.g., that modify and reset the element attributes.

16.4.1 Adding JavaScript Event Handlers to SVG Elements

In this section, we examine the steps involved in modifying an *SVG* display to link points across plots (see Figure 16.3). We show how to do this for a pair of plots. We start by placing unique identifiers on each point in the two plots so that we know which point is in which plot and which points correspond to the same observation in the original data frame. Next, we add three more attributes to these point elements. The *onmouseover* attribute holds the *JavaScript* function call to change the color of the point (and its associated points in the other plots) when the mouse hovers over it; the *onmouseout* attribute contains a call to a *JavaScript* function to undo the coloring when the mouse moves off the point; and the last attribute, *originalFill*, stores the point's original color so these *JavaScript* functions know how to change and restore a point's color. We also store in the *SVG* plot any *JavaScript* variables and functions needed to perform these operations.

We start by creating the collection of plots. We might use a simple call to the pairs() function to create a draftsman's display of pairwise-scatter plots. Alternatively, one can create the plots with

individual *R* commands and arrange them in arbitrary layouts with par() or layout(). We use par() to create two side-by-side plots:

```
doc = svgPlot({
        par(mfrow = c(1,2))
        plot(Murder ~ UrbanPop, USArrests, main="", cex = 1.4)
        plot(Rape ~ UrbanPop, USArrests, main = "", cex = 1.4)
              }, width = 14, height = 7)
```

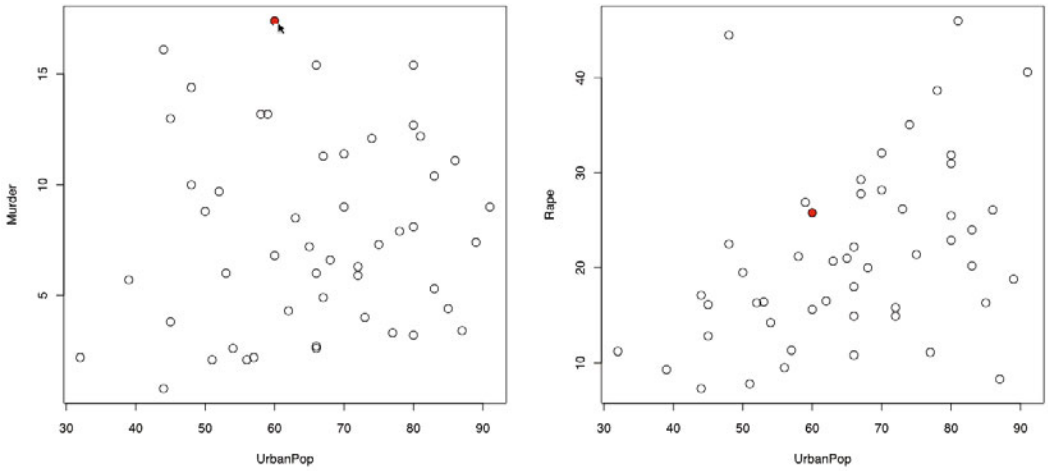


Figure 16.3: Linked Scatter Plots in *SVG*. The points in the two scatter plots shown here are linked. When the mouse covers a point in one plot, that point changes color as does the corresponding point in the other plot for the same observation. When the mouse moves off the point, then the linked points return to their original color(s).

The first annotation step is to place unique identifiers on each point in the two plots. We want these ids to include information as to which plot region and observation the point belongs. Whether we use pairs() or `par(mfrow = ...)`, the points within each plotting region appear in a regular order; that is, the first *SVG* element in each plotting region corresponds to the first row in the data frame, the second element to the second row, and so on. Thus, elements across plots can be matched by simply using the order in which they appear in their respective plotting regions (missing values can cause problems, however). We add *id* attributes of the form `"plotI-J"` where `"I"` is the plot index and `"J"` is the observation index within the plot/data frame. We control these *id*s as we will use them in *JavaScript* code created in *R*.

The function getPlotRegionNodes() retrieves the plot regions:

```
reg = getPlotRegionNodes(doc)
```

The children of each region node will be the points. We retrieve the children in the first region:

```
points = xmlChildren(reg[[1]])
```

and add an *id* attribute to each point,

```
mapply(function(id, node) xmlAttrs(node) = c(id = id),
       sprintf("plot1-%s", 1:xmlSize(points)), points)
```

We examine the, say, 17th point to see that it has the correct *id* value:

```
<path style="stroke-width:0.75; ..."
   d="M 208.347656 568.269531 C 208.347656 ... " id="plot1-17"/>
```

Next we add *onmouseover* and *onmouseout* attributes to each point. The value for these attributes is a *JavaScript* function call to perform the linking and unlinking action, e.g., to change the color of the linked points. Below is the augmented *SVG* node that corresponds to another point in the first plot. This is the point colored red in the left-hand plot shown in Figure 16.3:

```
<path style="stroke-width:0.75; ... "
   d="M 260.417969 72.800781 C 260.417969 ..." id="plot1-10"
   onmouseover="color_point(evt, 10 , 2 , 'red' )"
   onmouseout="reset_color(evt, 10 , 2 )"
   fill="none" originalFill="none"/>
```

Notice that the *onmouseover* event will call the *JavaScript* function *color_point()* (not an *R* function), passing it the event and three additional pieces of information: the index of the point, the total number of plots/regions in the display (two in our example), and the value for *fill*. Similarly, the call to *reset_color()* passes the point's index and number of plot regions. It does not need to pass a value for *fill* because it resets the color to its original value, which is found in the *originalFill* attribute. All together, five attributes have been added to the <path> element for this point: the identifier *id*, the *JavaScript* calls for *onmouseover* and *onmouseout*, and style attributes *fill* and *originalFill*. The *originalFill* is an attribute we have created and it will be ignored by the *SVG* renderer.

Finally, we must add code to the *SVG* document defining the two *JavaScript* functions *color_point()* and *reset_color()*. The code is placed in a <script> node within the *SVG* document. These *JavaScript* functions are relatively general and available in the SVGAnnotation package for making customized plots. They appear below.

```
function color_point(evt, which, numPlots, color) {
  for(i = 1; i <= numPlots ; i++) {
    path = document.getElementById("plot" + i + "-" + which);
    path.setAttribute("fill", color);
  }
}

function reset_color(evt, which, numPlots) {
  for(i = 1; i <= numPlots ; i++) {
    path = document.getElementById("plot" + i + "-" + which);
    path.setAttribute("fill", path.getAttribute("originalFill"));
  }
}
```

The SVGAnnotation package wraps these annotation tasks into the function linkPlots(). The next example shows how to use it.

Example 16-2 Pointwise Linking Across SVG Scatter Plots

Once we have created the graphical display (within a call to svgPlot()), the linkPlots() function does all of the additional annotation work for us. It finds and annotates the points in each panel/plot, and

adds our generic *JavaScript* functions for handling the interactivity to the document. We simply call linkPlots() passing it the *SVG* plot-document with

```
linkPlots(doc)
```

and then we save the modified document to text with

```
saveXML(doc, "USArrests_linked.svg")
```

The *SVG* display is ready for viewing and responding to viewer interactions.

16.4.2 Using JavaScript to Create Graphical Elements at Run-time

In the previous section, we showed how we can link points across plots with *JavaScript* code that simply alters the attributes of existing *SVG* elements in the plot created by the *R* graphics engine. That is, all of the computations on the data were done in advance, in *R*, in either the plotting stage or the annotation stage. At view time, the *JavaScript* functions simply change and reset attributes on the elements in the *SVG* display. An alternative approach is to place data from *R* in the *SVG* document as *JavaScript* variables, and in the viewing stage, use *JavaScript* and these variables to change the plot, adding or changing elements using computations it does with the "original data." For example, we might use *JavaScript* to draw line segments on a scatter plot to connect a point to its *k*-nearest neighbors, where the information as to which points are nearest is calculated in *R* and placed in the *SVG* document in *JSON*-formatted variables. The difference between this and the approach of the previous section is that we are drawing in two places: in *R* via the plotting routines, and in *JavaScript* at the time the document is viewed (and *R* is no longer available). We use *R* to draw the initial scatter plot, and *JavaScript* to draw lines on the plot in response to a mouseover event on a point and to discard these lines when the mouse moves off the element. We demonstrate how to do this in the next example. Of course, we can follow the previous approach and do all of this drawing in *R*. We can draw the line segments connecting each point to its nearest neighbors and hide these segments in the display. Then, we only use *JavaScript* to reveal and hide the relevant line segments in response to a mouse event.

Example 16-3 Drawing Nearest Neighbors on an SVG Scatter Plot with JavaScript

We use *JavaScript* to dynamically (i.e., at viewing time) augment a scatter plot so that it displays the four nearest-neighbors from our data set to a point (also in our data set) (see Figure 16.4). We use *JavaScript* to look up the elements in the *SVG* display that correspond to the nearest points and to add new <*line*> elements from the active point to its neighbors. Note that we are not adding *SVG* elements to the original *SVG* document, but rather adding line objects to the *JavaScript* rendering of the *SVG* display at view time. The browser will have already discovered the *SVG* document and processed the *SVG* into elements in a display.

Now that we have the strategy, we can perform the steps. We begin by creating the basic scatter plot:

```
doc = svgPlot(plot(mpg ~ wt, mtcars, pch = 19,
                   main = "Motor Trend Car Road Tests",
                   col= c("green", "blue")[(am+1)]))
```

In the annotation stage, we add unique identifiers to the points in the plot. Specifically, we add an *id* attribute to each point that simply has the index of the observation in the data frame. Since *JavaScript*

uses 0-based indexing, we find it easier to start the indexing at 0. We also add calls to the *JavaScript* functions (*showNeighbors()* and *hideLines()*) to handle the mouseover and mouseout events on each point, giving each function enough information to draw and undraw the lines for that point:

```
ptz = getPlotPoints(doc, simplify = FALSE)[[1]]
sapply(seq(along = ptz),
       function(i)
         addAttributes(ptz[[i]], id = i - 1L,
                       onmouseover =
                              "showNeighbors(evt, 4, neighbors)",
                       onmouseout = "hideLines(evt)"))
```

Also in the annotation stage in *R*, we calculate the distances between points and use these distances to identify the nearest neighbors as follows:

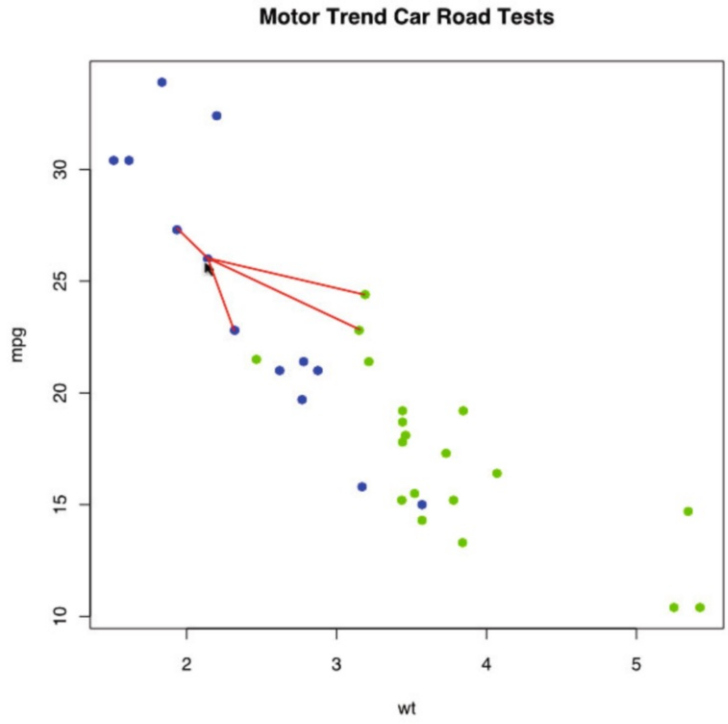


Figure 16.4: Interactive *SVG* Scatter Plot that Displays a Point's Nearest Neighbors. This interactive scatter plot can show the four closest points to any point in the display. When the mouse moves over a point, a *JavaScript* function is called to determine the coordinates of the four nearest neighbors and draw line segments from the active point to these neighbors. It removes them when the mouse moves away from the active point. The *SVG* document contains *JavaScript* variables that hold the indices of the nearest neighbors for each point, in order, from closest to farthest. These are computed easily in *R* and serialized from *R* to *JavaScript* as *JSON* content. We insert this *JSON* string into the *SVG* document so the values are available to the *JavaScript* code at viewing time for drawing the lines.

```
DD = as.matrix(dist(mtcars[, c("mpg", "wt")]))
D = t(apply(DD, 1, order)) - 1
```

D is a matrix that has a row for each observation and that row contains the indices of the nearest observations to it, in order from nearest to furthest. Notice that we use the order of the observation in the matrix so that it will match the identifier that we added to the point in the plot. We make this information about each point's neighbors available to the *JavaScript* code as a two-dimensional *JavaScript* array, serialized from *R*. The following call to addECMAScripts()[1] illustrates how we serialize the *R* matrix D to *JavaScript* as the variable *neighbors*. In addition we add the *JavaScript* code defining our *JavaScript* functions with

```
dimnames(D) = list(NULL, NULL)
jscript = list.files(path = system.file("examples", "Javascript",
                                        package = "SVGAnnotation"),
                     full.names = TRUE, pattern="knn")
addECMAScripts(doc, jscript, TRUE, neighbors = D)
```

The addECMAScripts() function has a ... argument that accepts *R* objects in the form `name = value`. These objects are added to the *SVG* document as *JavaScript* variables with the argument name in addECMAScripts() (e.g., *neighbors*) being used as the name for the *JavaScript* variable. The TRUE value for the third argument indicates that we want the *contents* of the *JavaScript* file to be copied into the *SVG* document rather than using a reference to the local file as a link. We choose to copy the contents to make the resulting *SVG* file self-contained and independent of auxiliary files.

We have completed the annotation of the *SVG* document. Now let's examine the *JavaScript* code. The *JavaScript* function *showNeighbors()*, which we call when the user moves the mouse over a point, has three arguments: the event object, the number of neighbors of interest, and a two-dimensional array identifying the nearest neighbors for all points (*neighbors*). This array is a *JavaScript* variable that contains the contents of the *R* matrix D that we computed earlier. Notice that the data are separated from the function and explicitly passed as arguments in the function call so that the function can be used with other data in other contexts. With its helper functions, *showNeighbors()* retrieves from the active point the index of each of the *k* neighboring points from the *neighbors* array. It then extracts the *SVG* object corresponding to each of the neighbors and gets its coordinates in the *SVG* canvas. With these coordinates, we create new line segments between each of the neighboring points and the active point. We place these line segments inside a new group element (a <*g*> node), which makes it easier to remove them when the mouse moves off the point.

We provide here the complete code for one of the helper functions, *addLines()*, which creates the line objects. We include it to show how *JavaScript* methods are used to construct new elements in the display at run time:

```
function addLines(obj, neighbors, numNeighbors)
{
  var x, y, x1, y1;
  var tmp = obj.getBBox();
  x = tmp.x + tmp.width/2;
  y = tmp.y + tmp.height/2;

  lineGroup = document.createElementNS(svgNS, "g");
  obj.parentNode.appendChild(lineGroup);
```

[1] *ECMAScript* is essentially another name for *JavaScript*. *ECMAScript* is the formal specification of the language standardized by Ecma International. The acronym ECMA stands for the European Computer Manufacturer's Association.

```
  var ids = obj.getAttribute('id') + ": ";
  for(var i = 1; i <= numNeighbors ; i++) {
    var target;
    target = document.getElementById(neighbors[i]);
    ids = ids + " " + neighbors[i];

    tmp = target.getBBox();
    x1 = tmp.x + tmp.width/2;
    y1 = tmp.y + tmp.height/2;

    var line = document.createElementNS(svgNS, "line");
    line.setAttribute('x1', x);
    line.setAttribute('y1', y);
    line.setAttribute('x2', x1);
    line.setAttribute('y2', y1);
    line.setAttribute('style', "fill: red; stroke: red;");
    line.setAttribute('class', "neighborLine");
    lineGroup.appendChild(line);
  }
  window.status = ids;
}
```

Note that although `libcairo` uses only <*path*> elements to draw objects, we can use higher-level elements such as <*line*> in our *JavaScript* drawing. We store the line segments as a global variable across calls to *JavaScript*. This makes it easy to remove the lines in one operation when the mouse moves off the point. We illustrate this operation with the definition of the *hideLines()* function which simply checks if the value of the variable *lineGroup* is defined and removes it and its children if it exists.

```
function hideLines()
{
  if(typeof lineGroup != "undefined") {
    lineGroup.parentNode.removeChild(lineGroup);
    lineGroup = null;
  }
}
```

With Examples 16-2 and 16-3, we have given a brief introduction to the kinds of interactivity that are possible with *SVG* and *JavaScript*; specifically, we have included *JavaScript* code within *SVG*. These two examples take somewhat different approaches. Example 16-2 (page 551) performed all of the drawing tasks in *R* and used *JavaScript* to manipulate attribute values of the *SVG* elements. On the other hand, Example 16-3 (page 552) used both *R* and *JavaScript* to draw at the different stages; we used *R* when creating the *SVG* and *JavaScript* when viewing the *SVG*. There are run-time benefits to the first approach because we do not need to draw each time the user, e.g., mouses over a point. However, there are many more elements in the document, and loading the *SVG* file may be slower. This trade-off, of course, involves the issue of where the computations are performed.

We have also seen that SVGAnnotation provides facilities for identifying elements of the *SVG* document that correspond to particular parts of plots. For example, the intermediate-level function

getPlotPoints() locates the *SVG* elements that correspond to, e.g., points in a scatter plot or hexagonal bins in the *R* display. These functions can assist developers in creating new high-level functions for annotating the output from "nonstandard" plotting functions in *R*. Section 16.8 provides a list of the intermediate-level functions available in the SVGAnnotation package.

16.4.3 Interaction with HTML User Interface Elements

The interactivity for the examples considered so far has been entirely contained within the *SVG* document. That is, the *SVG* functionality for tool-tips and hyperlinks and the *JavaScript* code for "onmouse" events that responds to user interaction have been located in the *SVG* file. In this section, we demonstrate how to integrate *SVG* and *HTML* when the *SVG* is housed within an *HTML* document and the *JavaScript* is in the *HTML*, not the *SVG*.

There are several benefits to this integration of *HTML* and *SVG*. For example, we can use *HTML* forms to control interactivity where we arrange controls in a form using *HTML*'s layout facilities, e.g., `<div>`s, lists, and `<table>`s. Additionally, we can embed multiple *SVG* displays in one *HTML* document, link *SVG* documents together using *JavaScript*, and control the width and height of the region in which the *SVG* is displayed, inducing scrollbars if necessary. We can also have *JavaScript* code both in the *HTML* document and in the *SVG* document(s). This increased locality and flexibility can be slightly more complicated, but leads to very rich displays.

HTML forms [13] provide built-in controls, such as a button, choice menu, check box, radio button, and textbox, that receive input from the user. *HTML5* provides additional controls such as sliders. These built-in *HTML* form elements are typically not as rich as those that can be drawn in *SVG* with *JavaScript*. However, they are simple to deploy in an *HTML* page.

With forms, we can integrate controls in *HTML* and a plot in *SVG*. We embed the *SVG* display in an *HTML* document using the *HTML* `<object>` element, e.g.,

```
<object id="PlotID" data="plot.svg"
        type="image/svg+xml" width="960" height="800"/>
```

This `<object>` specifies the name of the *SVG* file or *URL* (via its *data* attribute), the MIME type, dimensions, and unique identifier. When we want to operate on pieces of the *SVG* document (e.g., when the viewer clicks on an *HTML* button), we use *JavaScript* to retrieve the *SVG* object using the *id* attribute. Once we have this *SVG* object, we can operate on it with *JavaScript*, just as we have done when we placed the scripts in the *SVG* document. The *JavaScript* code below shows how we might retrieve the embedded *SVG* object and treat it as an *SVG* document.

```
doc = document.getElementById('PlotID');
doc = doc.getSVGDocument();
```

We can then deploy *JavaScript* callback functions that a) respond to viewer input via the controls in an *HTML* form and b) interact with *SVG* documents embedded in the *HTML* document. For example, when a viewer selects an option from a choice menu, a *JavaScript* function can be called to modify an *SVG* display in the document. We illustrate this approach in the next example.

Example 16-4 Using Forms to Highlight Points in a Lattice Plot

Here, we allow the viewer to explore the relationship between horsepower and miles per gallon for manual and automatic shift cars that have three, four, or five gears. We use panels in a lattice plot to compare manual and automatic cars. In addition, the viewer can highlight the points in each panel that have the selected number of gears. The viewer does this by choosing three, four, or five from a choice

menu that we have placed in an *HTML* form (see Figure 16.5). We have embedded the *SVG* display in an *HTML* document, and this document also contains an *HTML* form with the choice menu for subsetting the observations.

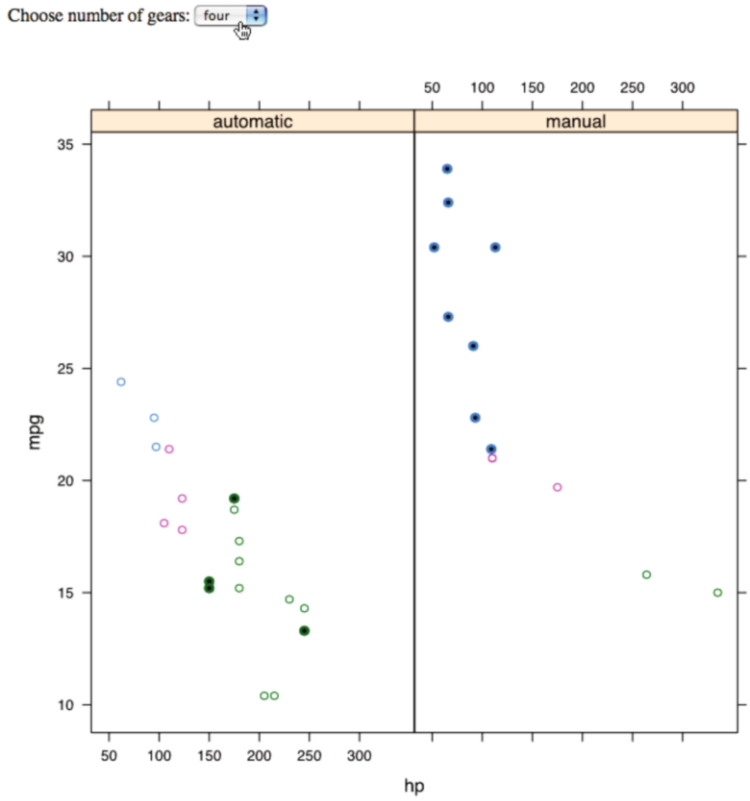


Figure 16.5: *HTML* Select Menu Used to Link Groups of Points Across *SVG* Conditional Plots. This screenshot shows an *HTML* page containing a form and an *SVG* lattice plot. The viewer controls the *SVG* display with the choice menu located in the form. When the viewer changes the selection, a *JavaScript* function in the *HTML* document is called to respond to this event, and it highlights the appropriate subset of points in the *SVG* plot.

Unlike the earlier examples in this chapter, the *SVG* plot contains no *JavaScript* code. However, all of the plotting elements do have identifier attributes (i.e., the *id* attribute indicates to which group a point belongs). For example, the following element for a point:

```
<path style="fill: none; stroke-width: 0.75; ... "
      d="M 195.441406 381.136719 C 195.441406  ... " id="1-2-3"/>
```

has an *id* of `"1-2-3"`, meaning it is the third point in the second subset in the first panel of the plot. This is a simple naming convention that we made up for this application in order to easily identify the points in the *JavaScript* code.

The *HTML* document rendered in Figure 16.5 contains the *SVG* plot by including it using the following `<object>` tag:

```
<object id="latticePlot"  data="mt_lattice_Choice.svg"
        type="text/svg+xml" width="960" height="768" />
```

This *HTML* document also contains a <*form*> element with a <*select*> menu that has options to specify the number of gears so that we dynamically highlight the observations with that number of gears. The form appears below:

```
<form> Choose number of gears:
 <select name="gear" onchange="showChoice(this)">
  <option value="0" default="true">Select</option>
  <option value="1">three</option> <option value="2">four</option>
  <option value="3">five</option>
 </select>
</form>
```

Notice that when the viewer changes the selection, the <*onchange*> event handler calls the *showChoice()* function. This function is passed the *JavaScript* object corresponding to the choice menu, and it uses this object to query the value the viewer has selected. The *showChoice()* function then calls *highlight()*, passing it the information it needs to highlight the new group of points. The *showChoice()* function is

```
var oldgroup = 0;
var group = 0;
var doc;

function showChoice(obj) {
  doc = document.getElementById('latticePlot');
  doc = doc.getSVGDocument();
  group = Math.floor(obj.value);
  if (group > 0) {
    highlight(group, pointCounts[(group - 1)], true);
  }

  if (oldgroup > 0) {
    highlight(oldgroup, pointCounts[(oldgroup - 1)], false);
  }
   oldgroup = group;
}
```

The *highlight()* function called from *showChoice()* extracts the elements in the *SVG* document corresponding to the points that belong to the selected group, changes their style attributes so they are highlighted, and resets the previously highlighted points to their original state. The *highlight()* function relies on the plotting elements having unique *id*s that indicate to which group the points belong, e.g., the "1-2-3" attribute value described before.

Notice that now that the *JavaScript* code no longer resides in the *SVG* document, we have accessed the *SVG* elements differently. We use the *JavaScript* methods *document.getElementById()* and *getSVGDocument()* to access the *SVG* document and its contents.

The *SVG* document created for the previous example had no *JavaScript* code in it and no calls to *JavaScript* functions. The interactive functionality is supplied entirely by *JavaScript* code in the

HTML document. The *JavaScript* code can access the plot elements in the *SVG* document and modify them dynamically. In general, provided the plot elements have appropriate ids, "plain" *SVG* can easily be made interactive via *JavaScript* embedded in *HTML* rather than in the *SVG*. One advantage of this approach is that plots made without intentionally or explicitly having interactivity can be given interactive features.

16.4.4 Adding Event Handlers to SVG Elements via JavaScript Code in HTML

We will consider another example that uses *JavaScript* code in an *HTML* document to programmatically modify elements in the *SVG* document at the time the *HTML* document is loaded in the browser. When the document is loaded, we will use *JavaScript* to add event handlers to elements in the *SVG*, such as *onmouseover*, *onmouseout*, and *onclick* handlers. This approach can be useful when the plots have been produced in *SVG*, but the creator has no reason to add any interactivity. If the *JavaScript* code can identify the appropriate elements in the *SVG* document, either by contextual knowledge or using known *id* attribute values, it can make static *SVG* interactive at viewing time. As an example, we consider the case where we place mouse event handlers on regions in a map after the map has been loaded in the browser, i.e., we use *JavaScript*, not *R*, to put event handlers on the map regions.

Example 16-5 Adding Event Handlers to SVG Maps When the Map Is Loaded into an HTML Page

We begin with an *HTML* page that displays the canonical red–blue map of the United States, where the states are colored red or blue according to whether the majority of votes in the 2008 presidential election were Republican or Democrat, respectively (see Figure 16.6). We want the viewer to be able to interact with the map by clicking on a state to display summary statistics for the selected state.

We have laid out the *HTML* page using nested `<div>` tags as follows:

```
<body onload="loaded()">
<center> <h3>Presidential Election 2008</h3>
 <p>
This map is interactive... results about the voting in that state.
 </p>
</center>
<div id="main">
 <div id="summaryMap">
  <div id="stateSummary"></div>
  <div id="map">
   <object id="svgMap" data="stateMap.svg"
     type="image/svg+xml" width="700" height="700">
   </object>
  </div>
 </div>
</div>
```

We have embedded the *SVG* map in the `<div>` element that has an *id* of `"map"`. This plot was created within *R* using the `maps` package [4]. We annotated the polygon elements for each state in the display to add an *id* attribute to each polygon that indicates the state to which it corresponds. Although we placed these unique identifiers on each polygon in *R*, we did not place any event handlers on the plot elements or add any *JavaScript* code to the *SVG*. Instead, we use *JavaScript* code in the

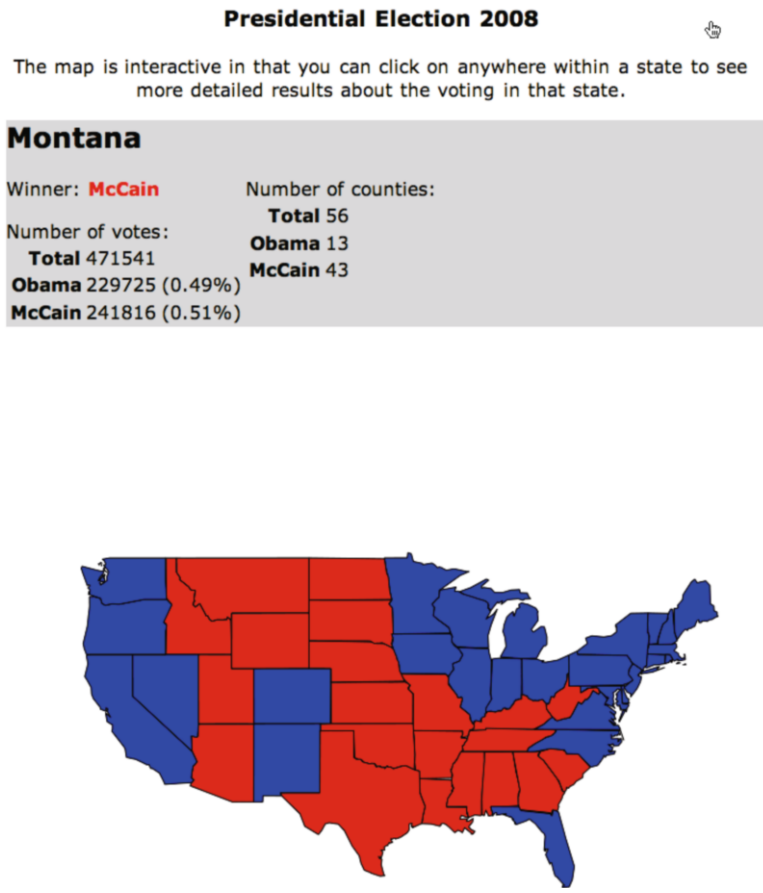


Figure 16.6: *HTML* Page with *JavaScript* Code that Connects an *SVG* Map to *HTML* Tables. This screenshot shows a map of election results by state. Interactivity with the map is controlled via *JavaScript* code located in the *HTML* page. When the viewer clicks on a state, an *HTML* table of summary votes for that state is displayed within the light-grey region above the map. The results for Montana appear in this screenshot.

HTML page to add *onclick* attributes when the page is loaded. The *JavaScript* code adds these *onclick* attributes as follows:

```
function loaded() {
  show("national");
  var doc;
  doc = document.getElementById('svgMap');
  doc = doc.getSVGDocument();

  for(var i = 0; i < polyIds.length ; i++) {
    var el = doc.getElementById(polyIds[i]);
    var txt = "parent.show('" + polyStates[i] + "')";
    el.setAttribute("onclick", txt);
```

```
    }
}
```

The *show()* function is responsible for displaying the requested *HTML* table. Alternatively, we can post-process the *SVG* before we display it. To do this, we use *XPath* expressions within *JavaScript* to find the elements.

We note that the mouse clicks occur on elements in the *SVG* document, but the *JavaScript* code to handle these events are in the *HTML* document. This means that our event handler needs to call a function located in its parent, e.g., `parent.show`. The *JavaScript* variables *polyIds* and *polyStates* are used to match the polygons in the map with the names of the states to know which state's results to display. These variables are available in a `<script>` tag in the `<head>` of the *HTML* document. We also place the *show()* function there. When *show()* is called, it retrieves a reference to the `<div>` element, named `"stateSummary"`, in the *HTML* document; this element is where the *JavaScript* code will place the table of statistics for the selected state. We modify the contents of this `<div>` element, adding the new *HTML* table.

The main difference between the approach presented here and that found in the earlier examples is that some of the post-plotting annotations have been done in the viewing stage. The post-processing occurs outside of the *R* environment, at the time the document is loaded. Thus, we can take *SVG* documents, which were not made for interactivity, and modify them using *JavaScript*. In this case, the interactivity may require access from the browser to the data and statistical routines used to generate the plot.

16.4.5 Embedding GUI Controls Within an SVG Plot

If we want a stand-alone *SVG* document that is interactive, we can draw user interface controls, such as buttons, check boxes, and sliders in the *SVG* display, and provide additional forms of interaction entirely within the *SVG*. For example, the Carto:Net community [5, 20] has developed an open source library of *SVG* graphical user interface (GUI) controls to incorporate into *SVG* figures and documents. This library consists of *JavaScript* functions to build the controls as *SVG* elements and functions to respond to user interaction. Carto:Net is available from `http://carto.net/`, and is also distributed with the SVGAnnotation package for convenience. It includes controls such as a check box, radio button, button, slider, text box, and selection menu. These are rendered using native *SVG* elements and so are quite different from their equivalents in *HTML* forms and pages. With *SVG*, there is the potential to design unique, sophisticated GUI elements, such as sliders, dial knobs, and color choosers. Another potentially useful feature of *SVG*-based GUI components is that they can be rendered with *SVG* filters and transformations, e.g., at an angle, with Gaussian blurring, or even animated. The main disadvantage is complexity. *SVG* GUI elements are somewhat more complex to use in programs than the "built-in" *HTML* form elements. Fortunately, the Carto:Net library offers a variety of GUI elements that can be easily embedded in the *SVG* document.

Although the GUI control is being drawn with *JavaScript* commands, the basic approach to incorporating Carto:Net UI components in an *SVG* document involves a few additional steps when annotating the *SVG*:

- Enlarge the viewing area so that there is room for the GUI control (i.e., change the *viewBox* attribute on the root element).

- Add initialization *JavaScript* code to the document in order to create the GUI object when the document is loaded (i.e., add a call to *JavaScript* initialization code via the *onload* attribute on the `<svg>` element). This script needs to call the *JavaScript* function(s) provided by Carto:Net to create the GUI object.
- Populate the `<defs>` node with elements that contain the generic drawing instructions for the UI control(s) that are provided by Carto:Net. We also locate the GUI in the display by adding an empty `<g>` tag that the Carto:Net scripts use to draw the graphical representation of the control.
- Add the Carto:Net scripts to the document along with our application's *JavaScript* event handler functions for the UI. This *JavaScript* code connects user activity on the UI to changes in the display, e.g., a change in the location of a slider's thumb calls a function that reveals a new curve on a plot and hides from view the old curve.

The SVGAnnotation package provides two generic functions, addSlider() and radioShowHide(), to embed a Carto:Net GUI component in an *SVG* document. These functions take care of the steps listed above. A slider created with the Carto:Net library can be queried to find the location of its slider thumb, set its value, move its slider thumb, and remove the slider from its display. This functionality is provided via the *JavaScript* slider object's methods *getValue()*, *setValue()*, *moveTo()*, and *removeSlider()*, respectively, and are invoked as, e.g., `slider.getValue()`. The check boxes and radio buttons available via radioShowHide() have equivalent functionality.

As an example, consider the interactivity in Figure 16.7. This display contains two side-by-side scatter plots and a slider below. The scatter plot on the left has a fitted curve and the one to the right displays the residuals from the fit. The slider corresponds to the degree of smoothness of the fitted curve. When the viewer moves the slider thumb to a new position, the corresponding fitted curve is added to the scatter plot on the left, the previously displayed curve is removed, the residuals for this new curve are rendered in the plot on the right-hand side, and the residuals from the earlier fit are hidden. Similar to the approach of many of the previous examples, the computations for fitting the curve to the data are performed in *R* for every possible selectable value of the smoothing parameter. The resulting curves and residuals are plotted for all possible values of the smoothing parameter in the *SVG* document. Then in the annotation stage, the *SVG* elements corresponding to each curve are given unique *id*s and a *visibility* attribute of `"hidden"`. Similarly, the residuals are given ids that can connect each residual to its curve, and they are hidden in the residual plot. When this *SVG* document is displayed in the browser, if the viewer moves the slider thumb, then the *JavaScript* callback function handles the hiding and showing of the appropriate elements in the two plots. If *R* were available in the browser, these computations could be done at viewing time.

16.5 Animation

SVG supports two types of animation: declarative, which solely uses *SVG* facilities; and scripted, which relies on the animation functionality in *JavaScript*. The declarative animation is a "pure" *SVG* animation in that the *SVG* elements contain directions in *SVG* on how to move themselves and how to change their shape or appearance/style. These animation instructions are provided via *SVG* elements that we add as children to the element we want to animate. The *SVG* renderer handles the changes. The alternative approach uses a built-in *JavaScript* timer, which we set to run *JavaScript* code at regular intervals. This code moves and changes attributes on the *SVG* elements in the display.

In this section, we look at both styles of animation—declarative and scripted. The SVGAnnotation package contains a high-level function, animate(), to provide declarative animation for a scatter plot.

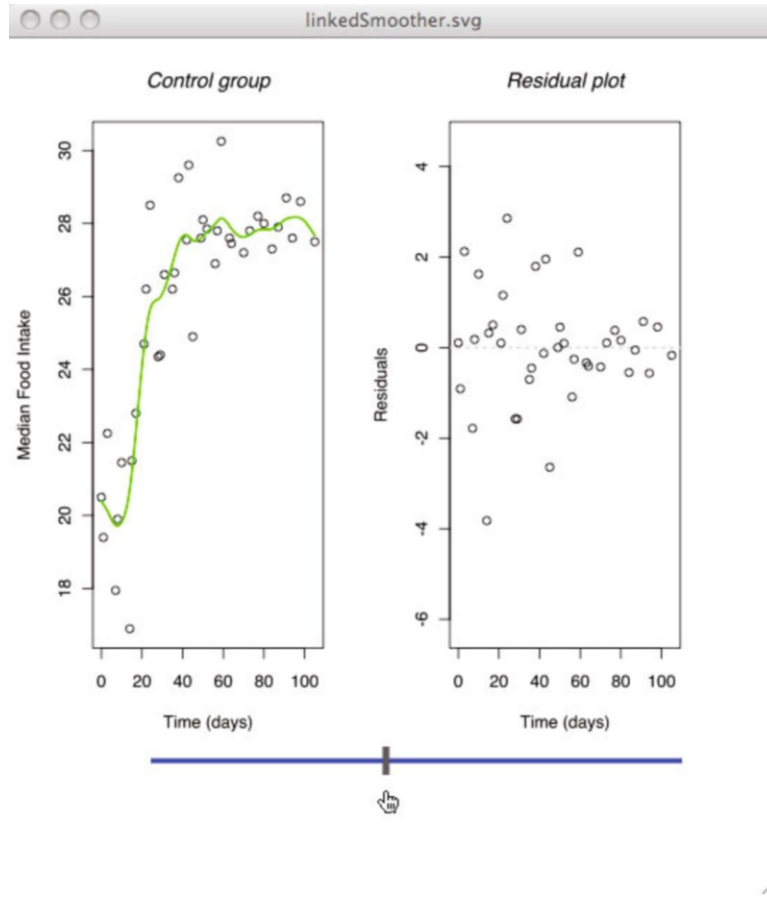


Figure 16.7: *SVG* Slider Added to a Plot. This pair of plots was made in *R*. Hidden within the plot on the left are curves from the fit for many different values of the smoothing parameter, and hidden in the right-hand plot are the residuals for each of those fits. The slider displayed across the bottom of the image specifies the smoothing parameter. It is provided by Carto:Net [5]. The slider is added to the *SVG* display using addSlider(). When the viewer changes the location of the slider thumb, the curve and residuals corresponding to the new bandwidth are displayed and the previously displayed curve and residuals are hidden. *JavaScript* code added to the *SVG* document responds to the change in the position of the slider thumb and takes care of hiding and showing the various parts of the two plots.

It is described next. Then in Section 16.5.2, we demonstrate how to use the *JavaScript* approach to animate a map. More details about the *SVG* animation elements can be found in Section 16.6.4.

16.5.1 Declarative Animation with SVG

The animate() function in SVGAnnotation uses the declarative approach to move points in a scatter plot around on the canvas. This function annotates a scatter plot adding additional elements that contain instructions on where and when to move each point in the plot. When viewed, each point moves

continuously in a synchronized fashion with the other points in the plot. The points move through their given sets of positions. In addition, the animate() function can also change the sizes and colors of the points during each step in the animation.

In the following example, we demonstrate how to use the animate() function to create an animation. The effect is similar to that of GapMinder [25].

Example 16-6 Animating Scatter Plots Through Snapshots in Time

In this example, we display, via a scatter plot animation, eleven decades of data provided by the United Nations (see Figure 16.8). The following is a snippet of the data we use,

```
  longevity    income population   yr    country
1     39.70 4708.2323    6584000 1900  Argentina
2     63.22 6740.9394    4278000 1900  Australia
3     42.00 5163.9268    6550000 1900    Austria
4     22.00  794.8383   31786000 1900 Bangladesh
5     51.11 4746.9662    7478000 1900    Belgium
6     32.90  705.4758   21754000 1900     Brazil
```

We want to plot the variable `income` along the *x*-axis and `longevity` along the *y*-axis for each country. We plot these values for 1900 and then animate the points over time (in `yr`). In addition, we use the values in `population` to determine the radius of the plotting symbol.

The first step is to plot the points for the initial decade, i.e., 1900, which creates the starting frame of the animation. We control much of the layout, placing the tick marks, grid lines, and labels ourselves, as shown here:

```
doc =
  svgPlot({
        plot(longevity ~ income, subset(gapM, yr == 1900),
             axes = FALSE, pch = 21, col = colsB, bg = colsI,
             xlab = "Income", xlim = c(-400, 50000),
             ylab = "Life Expectancy", ylim = c(20, 85))
        box()
        y.at = seq(20, 85, by = 5)
        x.at = c(200, 400, 1000, 2000, 4000, 10000, 20000, 40000)
        axis(1, at = x.at,
             labels = formatC(x.at, big.mark = ",", format = "d"))
        axis(2, at = y.at, labels = formatC(y.at))
        abline(h = y.at, col="gray", lty = 3)
        abline(v = x.at, col="gray", lty = 3)
        legend(35000, 40, longCont, col = colsB,
               fill = colsB, bty = "n", cex = 0.75)
    })
```

Here `colsB` holds the colors for each circle border, and `colsI` has the colors for the interior of the circles. We color the points according to the continent to which they belong. The `longCont` variable in the call to legend() contains the names of the continents. We would also like to provide tool tips for each of the points that display the particular country's name. We annotate the plot using addToolTips() with

```
addToolTips(doc, as.character(gapM$country[ gapM$yr == 1900 ]))
```

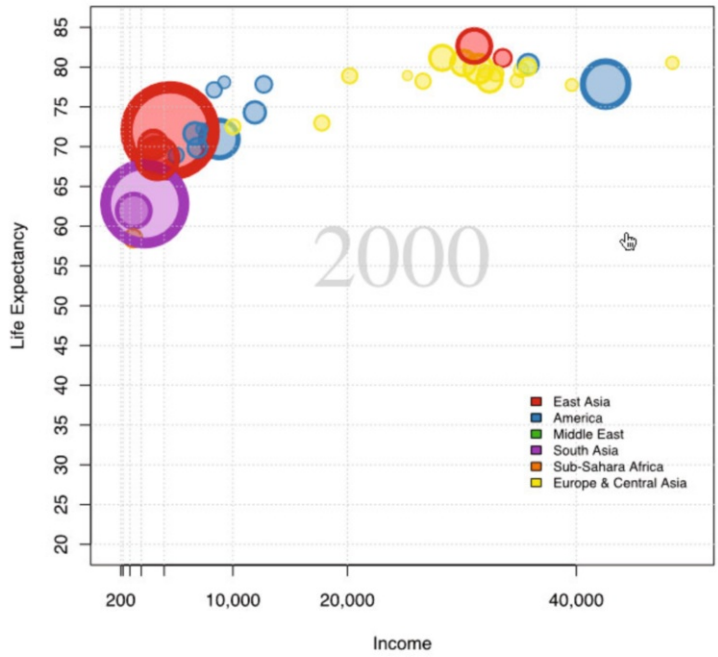


Figure 16.8: Animated Scatter Plot Drawn in *SVG*. This screenshot shows a scatter plot at the end of an animation. Each circle represents one country. The location of a circle corresponds to (income, life expectancy) pairs, and its area is proportional to the population for that country in that year. These circles move and change size as the animation progresses through each decade. Reloading the page restarts the animation.

The initial view or starting point of the animation is now in the variable `doc`. We provide this to animate(), along with the subsequent slices of the data for each decade. We specify the duration of the animation via the *interval* parameter of each step (the total duration of the animation can also be specified via *dur*). The radii of each circle, for each time period (decade), is provided in *radii*. The call to animate() is

```
animate(doc, data = gapM[ , c("income", "longevity")],
        which = gapM$yr, dropFirst = TRUE,
        labels = seq(1900, length = 11, by = 10),
        begin = 0, interval = 3, radii = radL)
```

The animate() function also needs to know the range of the horizontal and vertical data used to create the initial plot so that it can scale the motion of the points appropriately. It computes these from the data by default, but we can specify this information with the `xlim` or `ylim` parameters in plot(). We can also supply animate() with background labels and point colors that can be changed for each time period. We specify these via the *labels* and *colors* arguments to the function, respectively.

The animate() function handles the complication that arises from the different coordinate systems of the data and the *SVG* canvas. However, problems may arise when we call plot() with the *log*

argument. If we do this and also specify the motion of a point in its original untransformed units, then its position will not be properly translated into the units of the canvas. To remedy this, we recommend logging the variable, instead of using the *log* parameter in plot().

A second complication arises due to the *SVG* element for an *R* plotting symbol being a <path> rather than a <circle> element. The animate() function examines the paths to determine whether or not the plotting symbol is a circle. If the *radii* parameter is provided and the plotting symbol for a point is a circle, then <path> is replaced by <circle> so that it can easily grow and shrink as it moves through the stages of the animation. Currently, the *radii* parameter for other symbols is ignored.

16.5.2 Programming Animation with JavaScript

In *JavaScript*, we use the *setInterval()* function to create animations with *SVG* documents. This function sets a timer or alarm to go off at repeated intervals. When this alarm goes off, *JavaScript* code that was specified when we set the timer is evaluated. For example, in the code:

```
animationHandle = setInterval("animationStep()", Interval);
```

the *setInterval()* function is called with the values "animationStep()" and *Interval*. This means that after *Interval* milliseconds, a call to *animationStep()* will be made. The return value from the *setInterval()* function is a reference to the timer. When the timer triggers an evaluation of this code, that code can change the interval for the next "alarm." This way, we can have different time intervals between the calls to *animationStep()*. To stop the animation, we call the *JavaScript* function *clearInterval()*. This allows us to terminate a scheduled animation step.

In the next example, we demonstrate how to use this *JavaScript* timer to animate an *SVG* display that was created in *R*.

Example 16-7 Animating a Map with JavaScript and SVG

We animate a political map of the states within the United States, where each state is painted the color of the party of the presidential candidate who won that state: red for Republican; blue for Democrat; and green for Independent. For any given year, this gives us a visualization of geographical voting patterns, and the animation enables us to view changes in these patterns over time. We obtained this information for 1900 to 2008 from http://electionatlas.org.

We start by plotting a state map of the United States where each state is gray. To this map, we add the word "Start" which we will make interactive so the viewer can control the animation by clicking on this word. We make this plot with

```
doc = svgPlot({
       m = map('state', fill = TRUE, col = 'grey66')
       title('Presidential Elections 1900-2008')
       text(m$range[1] + 3, m$range[3] + 1, "Start", col = "green")
             })
```

Next, we add an *onclick* event handler to the word "Start" so that it acts as a button. When this word is toggled, we will begin the animation and change the text of this button to "Pause" (to indicate to the viewer that we will pause the animation if the viewer clicks on it). Our event handler for the "button" is the *JavaScript* function *toggleAnimation()*, which we add to the element with

```
start = getTextNodes(doc)[[1]]
start = asTextNode(start, "Start")
xmlAttrs(start) = c(id = "Start", onclick = "toggleAnimation(evt)")
```

The *SVG* graphics device in *R* uses `<path>` elements to draw each letter in a title or label in a plot. To make things simpler, we use `<text>` elements instead. The second line of the above code converts this "text" node that holds the drawing instructions for "Start" to a simple `<text>` node.

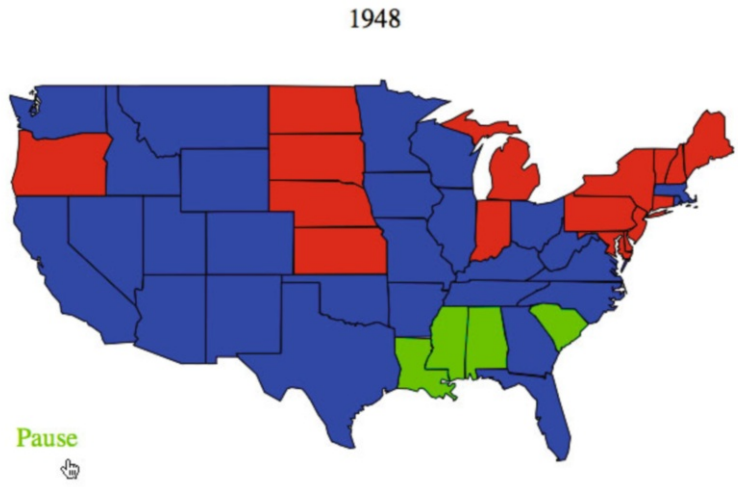


Figure 16.9: Animated *SVG* Map controlled by *JavaScript*. This screenshot was captured in the middle an animation where the color of a state changes to reflect the party of the presidential candidate who received the most votes in an election year (red for Republican, blue for Democrat, and green for Independent). The animation is controlled by a *JavaScript* timer, where at regular intervals when the timer goes off, *JavaScript* code modifies the *SVG* to fill each state with the color for the next election result and changes the year shown above the map. A mouse-click on `"Pause"` halts the animation.

The *toggleAnimation()* function changes the color of each state in the map to match the winning party in the state for that year. This function also checks to see if the animation has completed and, if so, sets up the map so that the viewer can start the animation over. As in Section 16.4.1, we place unique identifiers on the plot title, Start "button", and regions in the map to make it easier for us to retrieve these elements for dynamic updating during the animation.

To initiate the animation, we place a call to *setInterval()* within an *init()* function, which is run at the time the *SVG* document is loaded. The top of the *SVG* document appears as

```
<svg xmlns="http://www.w3.org/2000/svg"
   width="504pt" height="504pt" viewBox="0 0 504 504"
   version="1.1" onload="init(evt)">
```

The *onload* attribute contains the call to *init()* to initiate the animation.

The relevant *JavaScript* code (which we place in the *SVG* document) for this initialization is

```
var animationHandle = null;
var currentYear = 0;
var Interval = 1000;

function init(evt) { ...
```

```
  animationHandle = setInterval("animationStep(ids, colors,
                                    yearLabels, stateNames)",
                                Interval);
}
```

The *Interval* variable is set to 1000 milliseconds so at 1-second intervals we will update the title to the next election year and change the color of the states.

The function *animationStep()* is called at each step of the animation. This function increments the value of *currentYear* and checks to see if the animation has completed. If it has finished, we reset the display. If not, we update the display by calling *displayNextStep()* with the next set of input values. The *animationStep()* function appears below:

```
function animationStep(ids, colors, yearLabels, stateNames)
{
  currentYear++;
  if (currentYear >= yearLabels.length) {
    var el = document.getElementById("Start");
    setTextValue(el, "Start");
    clearInterval(animationHandle);
    animationHandle = null;
    for (var i = 0; i < ids.length; i++) {
       var el = document.getElementById(ids[i]);
       if (el) el.setAttribute('style', 'fill: ' + '#A8A8A8');
    }
    var title = document.getElementById('title');
    setTextValue(title, 'Presidential Elections 1900-2008');
    return;
  }
  displayYear(currentYear, ids, colors, yearLabels, stateNames);
}
```

The complete collection of *JavaScript* functions required for this animation (e.g., *displayYear()*) can be found in the examples supplied with the SVGAnnotation package.

16.6 Understanding Low-level *SVG* Content

The purpose of this section is to introduce and explain some aspects of working with *SVG* content more directly. In previous sections, we tried to use *R* functions to find and modify *SVG* nodes. We rarely needed to understand much about the structure or content of the nodes, but worked locally, e.g., adding one or more attributes to a specific node. However, when post-processing *SVG* generated by *R* and `libcairo` to create interactive customizations, some understanding of the *SVG* structure may be needed. For example, when we made the text in the map interactive so that we can control the animation in Section 16.5.2, we needed to know that the *SVG* elements for labels produced by the *SVG* graphics device in *R* were `<path>` elements, rather than `<text>` elements. Also, when we want to change the appearance of a graphical element dynamically, we need to understand how

to specify styles in *SVG*, and when we want to develop an *SVG*-based animation, we need to know more about the animation elements in *SVG*. These are the topics of this section.

16.6.1 The SVG Display for an R Plot

Since *SVG* is *XML*, the *SVG* documents produced in *R* are highly structured, and we can capitalize on this structure to locate particular elements and enhance them with additional attributes, parent them with new elements, and insert sibling elements in order to create various forms of interactivity and animation. To introduce this structure, we examine the *SVG* generated by a typical call to plot(), e.g., the call that was used to make the scatter plot in Example 16-1 (page 542):

```
doc = svgPlot(plot(lat ~ long, data = quakes[depth.ord, ],
                   pch = 19, col = depth.col[depth.ord],
                   xlab = "Longitude", ylab="Latitude",
                   main = "Fiji Region Earthquakes") )
```

The tree diagram in Figure 16.10 provides a sketch of the hierarchy of the resulting *SVG* document.

Most of *R*'s plotting functions are very regular and predictable so it is possible to determine which *SVG* nodes correspond to which graphics objects in the *R* display. The approach we use in the SVGAnnotation package is based on understanding the default operation of the plotting functions. For example, notice that the first <*g*> sibling of the <*rect*> element in Figure 16.10 has 1000 children, which matches the number of rows in quakes. We can confirm this with

```
kids = getNodeSet(doc, "/x:svg/x:g/*", "x")
sapply(kids, xmlSize)
```

```
[1]    0 1000   25    5
```

The next <*g*> element has 25 children, and it corresponds to the axes of the data region of the plot and their tick marks. The last <*g*> contains the title and the two axes labels. It has five children, one each for the title, *x*-axis label, and *y*-axis label, and two additional children that were added to create rectangular regions around the axes labels that respond to mouse events.

The high-level functions in SVGAnnotation make use of these default locations for the most common plots. If we wish to annotate less common plots, then we may need to use other functions in SVGAnnotation, or directly handle the *XML* nodes ourselves with the functionality available in the XML package. When we find and identify the nodes corresponding to the elements of the plots, which are different for different plot types, we then develop functions for the new plot type. Since we can draw anything with *R*, the structure may not follow any particular order.

Next, we examine the first of these 1000 nodes to see how svg() draws points (these correspond to the points in the scatter plot). We see below that the first node is a <*path*> element:

```
kids[[2]][[1]]
```

```
<path style="fill-rule: nonzero; ..."
   d="M 337.144531 191.292969 C 337.144531 194.894531 ..."
   type="plot-point"
   xlink:title="lat = -20.32, long = 180.88, ... stations = 22">
 <title>
lat = -20.32, long = 180.88, depth = 680, mag = 4.2, stations = 22
```

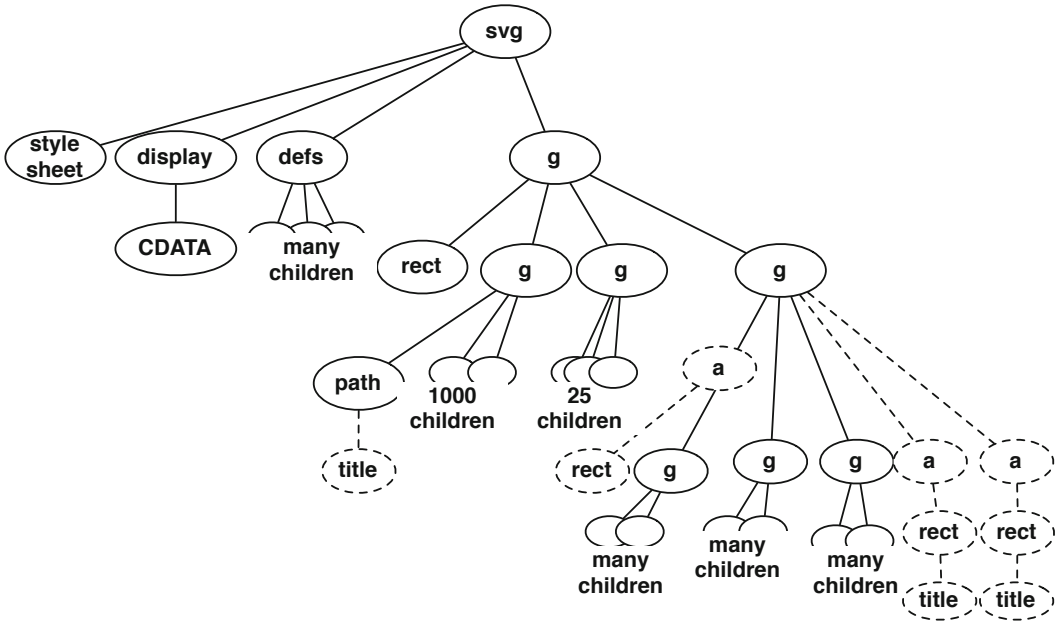


Figure 16.10: Example *SVG* Tree. This tree provides a high-level diagram of the organization of the *SVG* document produced by a nested call of plot() within svgPlot(). The style sheet and the <*display*> element on the left of the tree were added by svgPlot() to include a *CSS* file and also a description of the *R* plot. The <*CDATA*> child of <*display*> contains the *R* code passed in the call to svgPlot(). The nodes with dotted lines are those that have been added to the *SVG* document by the addLink() and addToolTips() functions. For readability, not all nodes are displayed, and in some cases the number of nodes is provided to make it clear to which part of the plot these elements correspond. The "1000 children" refers to the point elements as they correspond to the 1000 observations in the data frame, and the "25 children" corresponds to the axes of the data region.

```
 </title>
</path>
```

Although the symbol used to represent this point in the plot is a circle, the *SVG* graphics device in *R* (via libcairo) does not use the <*circle*> element to draw it. Instead, the <*path*> node provides instructions for drawing the "circle" using Bezier curves to connect the points supplied in the <*path*> node's *d* attribute. Locating nodes in the document corresponding to points is not trivial even for scatter plots because the coordinate system for the plot and the data frame are different. The *x*, *y* coordinates used to specify the path are in the coordinate system of the *SVG* canvas, not the coordinate system of the data (in the *R* data frame). As a result, we cannot directly use the values in our data to identify the corresponding *SVG* elements in the document; the data values first need to be converted into this alternative *SVG* coordinate system and that depends on the data region of the plot. The *SVG* coordinate system supports various units of measurement, but the device in *R* uses only points (abbreviated as 'pt' or 'pts'). A point is approximately 1/72 of an inch.

This *SVG* document has been modified to add a <*title*> element to each point element, where the text content of <*title*> is displayed on a tool tip when the mouse hovers over the corresponding point in the scatter plot. Functionality to add this kind of interactivity is available with addToolTips().

Additionally, once we identify particular plot elements, e.g., axes labels, plot titles, points, and other regions in a plot, we can pass these to addToolTips(). Over 25 functions are available in SVGAnnotation to access these elements, such as getPlotPoints() and getAxesLabelNodes(). (See Section 16.8 for a list of these "get" functions.)

16.6.2 Text in the SVG Display

The `libcairo` engine used by *R* generates all shapes exclusively with the <*path*> tag. This applies to the text in axes and plot labels too; that is, the text is explicitly drawn via *SVG* paths rather than with <*text*> elements. The resulting letters scale extremely well and do not rely on special fonts, which may not be available at the time of rendering. More specifically, the path for the glyphs that correspond to individual letters are created and placed in a <*defs*> element, and a <*use*> element brings in the glyph at the proper location in the plot. This allows for reuse of common strings. However, this representation introduces some difficulties for us because the text for legends and axes labels do not appear as plain text in the *SVG* document so are not as easily located for post-processing. Also, the placement of the glyphs in <*defs*> means that there is one additional level of indirection that needs to be handled when annotating text.

Below is a <*g*> element used to draw the axis label `"Latitude"`. The attribute *type* on <*g*> identifies this element as an "axis-label". We have added this attribute in the post-processing stage to make it easier to locate and annotate the axis.

```
<g style="fill: rgb(0%,0%,0%); fill-opacity: 1;" type="axis-label">
 <use xlink:href="#glyph1-7" x="14.398" y="266.152"/>
 <use xlink:href="#glyph1-8" x="14.398" y="259.478"/>
 <use xlink:href="#glyph1-9" x="14.398" y="252.804"/>
 <use xlink:href="#glyph1-10" x="14.398" y="249.470"/>
 <use xlink:href="#glyph1-9" x="14.398" y="246.804"/>
 <use xlink:href="#glyph1-11" x="14.398" y="243.470"/>
 <use xlink:href="#glyph1-12" x="14.398" y="236.796"/>
 <use xlink:href="#glyph1-13" x="14.398" y="230.123"/>
</g>
```

This <*g*> node has eight references to glyphs, one for each letter in `"Latitude"`. These glyphs are located in the <*defs*> node of the document. The references are made via the *href* attribute. For example, the letter "a" is the second child of <*g*>; it appears in the <*defs*> node with an *id* of "glyph1-8":

```
<symbol overflow="visible" id="glyph1-8">
 <path style="stroke: none;" d="M -1.671875 -1.578125
   C -1.367188 -1.578125 -1.128906 -1.6875 -0.953125 -1.90625 ...
   Z M -6.421875 -3.265625 "/>
</symbol>
```

16.6.3 Styles in SVG

In this section, we describe how to specify *CSS* styles in an *SVG* document. Styles can be specified in four ways.

1. *Inline CSS styles.* One approach is to place the style information directly in the graphical element (e.g., <*circle*>, <*path*>) by setting the value of a *style* attribute. For example, the Oregon polygon in Figure 16.2 has the following *style* attribute:

   ```
   <path style = "fill: rgb(100, 149, 237);
      fill-opacity: 0.5;stroke-width: 0.75;
      stroke-linecap: round; stroke-linejoin: round;
      stroke: rgb(0%,0%,0%); stroke-opacity: 1;" ... />
   ```

   The value of *style* is provided as a *CSS* list of properties arranged in name:value pairs separated by `";"`. This particular style attribute provides the color (cornflower blue) and opacity for filling the polygon, the color of the border (black), and other details about the path, such as the thickness of the line used to draw the path. Note that colors in *SVG* can be represented by text name, e.g., `cornflowerblue`, the red–green–blue triple `rgb(100, 149, 237)`, or the hexadecimal representation of the triple, e.g., `#6495ED`.
2. *Internal Style Sheet.* Alternatively, we can place *CSS* information in the <*defs*> node within the *SVG* document. As an example, the following <*style*> node is placed in the <*defs*> portion of an *SVG* document,

   ```
   <style type="text/css">
    <![CDATA[
   .recs {fill: rgb(50%,50%,50%); fill-opacity: 1; stroke: red;}
     ] ]>
   </style>
   ```

   This internal style sheet defines the `"recs"` class. This class is available for placing styles on graphical elements in the document. Below we see how a <*rect*> node references this `"recs"` class in the rectangle (see Figure 16.2 for the rendered image):

   ```
   <rect x="10" y="20" width="50" height="100" class="recs"/>
   ```

   The rectangle is connected to this style via the value of its *class* attribute, i.e., `"recs"`.
3. *External Style Sheet. CSS*'s may also be located in an external file for reuse in multiple files. It can be included via the xml-stylesheet processing instruction such as

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

4. *Presentation Attributes.* An alternative to using a *CSS* for specifying styles, whether inline, internal, or external, is to provide individual *XML* "presentation" attributes directly in the *SVG* element. For example, the following circle has a *fill* attribute specifying the circle's color:

   ```
   <circle id="pinkcirc" cx="50" cy="50"  r="15" fill="pink"/>
   ```

These various approaches can be mixed throughout the document and even within a node. That is, style information can be provided from a combination of inline, internal, and external style sheets, as well as presentation attributes. Presentation attributes are very straightforward and easy to use. We can just add a simple attribute, e.g., *fill*, on an element and avoid the extra layer of indirectness. This approach allows us to easily modify the presentation of an element in response to a user action. However, the

downside of this approach is that presentation information is mixed with content. For this reason, the inline, internal and external cascading style sheets are often used rather than presentation attributes.

For more information about cascading style sheets, see [15].

16.6.4 SVG Animation Elements

The *SVG* animation elements include `<animate>`, `<animateMotion>`, `<animateTransform>`, `<animateColor>`, and `<set>`. These elements act on attributes of the animation element's parent. For example, we can use `<animateTransform>` to change the *width* attribute of the parent element, which causes the parent element to grow or shrink. We can also use `<animateTransform>` to change the *visibility* attribute of an element to make it hidden/visible. We can move a graphical element to a new position on the canvas by specifying the coordinates of this new location in the `<animateMotion>` element. The element will move along a smooth path for the given interval of time.

As an example of how declarative animation works, we describe the action taken by the *SVG* element shown below. This `<circle>` element moves from its original location (95, 370) at the time of loading the *SVG* document (i.e., at 0 seconds) to a new location (67, 412) in 2 seconds. As it moves, this circle simultaneously shrinks from its original radius of 5.4 to 3. This mini-animation is the building-block for the animation created in Figure 16.8. In that example, each point represents a country with the *x* and *y* locations corresponding to the country's average life expectancy and income for a particular decade. As the points move according to changing year, they also grow or shrink according to population size.

```
<circle x="95" y="370" r="5.4"
 style="fill-rule: nonzero; fill: rgb(100%,0%,0%);...">
 <animateMotion id="move1"  1
     from="95, 370" to="67, 412"  2
     fill="freeze" 3
     dur="2s" begin="0s"/">  4
 <animateTransform  5
     attributeName="transform"  6
     type="scale"  7
     fill="freeze"  8
     to="3"  9
     dur="2s" begin="0s"/">
</circle">
```

1 This `<animateMotion>` node contains instructions for moving its parent, the `<circle>` node, along a path.

2 The attributes *from* and *to* on `<animateMotion>` provide the beginning and ending location for the circle.

3 This *fill* attribute is not to be confused with the *fill* attribute of the circle. It refers to what happens at the end of the animation. The value of `"freeze"` indicates that the final value should be kept. Without it, the attribute being animated would return to its original value at the end of the animation period, e.g., this circle would return to its starting position.

4 The attributes *begin* and *dur*, indicate that the movement of the circle (and its change in size) are to start at `"0s"`, i.e., when the page is loaded, and last for 2 seconds. The time can be specified relative to when the document was loaded or relative to completion of another animation.
5 This `<animateTransform>` element provides instructions on how to resize the circle. This can be used for animating several different characteristics of an *SVG* element, e.g., *scale*, *visibility*, or *rotate*.
6 This *attributeName* attribute value on the `<animateTransform>` element is `"transform"`, which indicates that the `<circle>` is being transformed in some way. Other values of *attributeName* can be the name of an attribute on the parent element that we want to animate/change.
7 The value of `"scale"` for this *type* attribute indicates that the circle is to be re-sized; other possible values for *type* are rotate, skewX, skewY, and translate, which we can use to reshape a graphical element.
8 This *fill* attribute on `<animateTransform>` keeps the circle from returning to its original size at the end of the animation, i.e., after the `"dur"` time has elapsed.
9 The *to* attribute on this `<animateTransform>` node indicates the final size of the circle.

We coordinate the movement of the circle and the changing of its size by specifying that both animations are to begin at the same time (*begin* is 0 seconds) and take the same amount of time (*dur* is 2 seconds). Any animation element requires a *dur* attribute to specify its duration. The value can be provided in seconds, e.g., `dur="10s"`, or minutes, e.g., `dur="2min"`. It is also possible to use clock values, e.g., `dur="2:10"` for 2 minutes and 10 seconds. The *SVG* clock starts ticking when the document has completed loading.

In addition to specifying the duration of an animation, it is also possible to specify when the animation is to begin or end. The values of these attributes can be absolute values, as with *dur*, or they can be relative to the start or end of another animation. For example, if we change the *begin* value in the above `<animateTransform>` to `"move1.end+3s"`, then the circle would start shrinking 3 seconds after it finished moving to its new location, i.e., 3 seconds after the animation with the *id* `"move1"` ends. This can be useful when we want to synchronize various parts of an animation.

With `<animateMotion>`, we can also specify the path the circle takes in traveling from one location to the other.

The `<set>` element animates the values of *SVG* attributes, such as *fill*. For example, the color of a circle can be changed when it reaches its new location by embedding the following element as a child of `<circle>`:

```
<set attributeName = "fill" attributeType = "CSS"
   begin = "move1.end" to = "#FFDB00FF" fill = "freeze"/>
```

The attribute called *attributeName* specifies which attribute of its parent node is to be "animated." In this example, it is the *fill* attribute, which controls the color with which we fill the circle. The change of color begins when the circle has finished moving because the *begin* attribute has a value of `"move1.end"`. The color changes *to* the new color of `"#FFDB00FF"`. The *fill* attribute on this element should not be confused with the value of *attributeName*. This *fill* refers to whether the attribute being animated will return to its original value at the end of the animation or will keep the new color. Its value of `"freeze"` indicates that the final color will remain once the animation ends.

One additional point we make is that the *attributeType* attribute in the `<set>` element indicates where the attribute being changed can be found. A value of `"CSS"` means that the fill color is located in the *style* attribute of `<circle>`, e.g., `style="fill: rgb(100%,0%,0%);"`. If the circle's color had been specified in a presentation attribute, e.g., `fill = "rgb(100%,0%,0%);"`, then we would have given *attributeType* a value of `"XML"`.

Below is an abbreviated version of the point element produced by animate() for India in Example 16-6 (page 564) . Notice the similarities to the sample *SVG* shown above.

```
<circle x="82.093" y="394.221354" r="2.699"
  style="fill-rule: nonzero; fill: rgb(59.60%,30.58%,63.92%); ..."
  type="plot-point">
 <title xmlns="http://www.w3.org/2000/svg">India</title>
 <animateMotion begin="0s" dur="3s" id="move1"
    from="82.093,394.221" to="82.217, 396.429" fill="freeze"/>
 <animateTransform attributeName="transform"
    begin="0s" dur="3s" type="scale" fill="freeze" to="5.38376"/>
 <animateMotion begin="3s" dur="3s" id="move2"
    from="82.217, 396.429" to="82.334, 377.378" fill="freeze"/>
 <animateTransform attributeName="transform"
    begin="3s" dur="3s" type="scale" fill="freeze" to="5.38376"/>
...
 <animateMotion begin="27s" dur="3s" id="move10"
    from="89.661, 196.133" to="94.979, 190.222" fill="freeze"/>
 <animateTransform attributeName="transform"
    begin="27s" dur="3s" type="scale" fill="freeze" to="9.69793" />
</circle>
```

Within the *<circle>* are a sequence of 10 pairs of *<animateMotion>* and *<animateTransform>* nodes for each of the 10 transitions from one decade to the next.

16.7 Possible Enhancements and Extensions

Flash* and *Flex

The motivation behind the SVGAnnotation package is the ability to exploit and harness new media such as interactive Web pages. The aim is to enable statisticians to readily create new graphical displays of data and models using existing tools that can be rendered in rich, interactive, and animated venues such as Web browsers, Google Earth, Google Maps, GIS applications, etc. The SVGAnnotation package focuses on facilitating *R* users in leveraging existing *R* graphical functionality by post-processing the output to make it interactive and/or animated. It is a complete computational model in that it enables the author of a display (or a third party) to modify all elements in that display. This approach can be used for other modern formats.

Another format for graphical displays and applications is Adobe's *Flash* and *Flex* [1, 12] (`http://www.adobe.com/products/flash`). This is an alternative to *SVG* that is a widely used framework (the ActionScript programming language, collection of run-time libraries, and compiler suite) for creating general interfaces and displays. The range of applications include rich graphical user interfaces and interactive, animated business charts. *Flash* and *Flex* are freely available, but not Open Source. The format and tools are mostly controlled by Adobe. The Adobe chart libraries for *Flash* are only available in commercial distributions.

There are publicly available Open Source libraries for creating certain types of common plots, e.g., flare `http://flare.prefuse.org`. Alternatively, we can use *R* to create statistical graphical displays by generating *ActionScript* code to render the different elements in the display. We have developed a prototype *R* graphics device (in the FlashMXML package [31]) that creates *Flash* plots

within *R* using the regular graphics framework. We can then post-process the generated content in much the same way we do with the SVGAnnotation package in order to provide interaction and animation.

Rather than considering *Flash* and *JavaScript* as competitors to *SVG*, we think that each has its own strengths. *SVG* is more straightforward and direct than *Flash* and *JavaScript* for creating graphical displays. For one, we have an excellent *R* graphics device that creates the initial *SVG* content. We can add GUI components, but *Flash* is better for this as it provides a much richer collection of GUI components such as a data grid. Drawing displays with *JavaScript* avoids depending on the presence of support for either *SVG* or *Flash* and so can be deployed in more environments.

Another approach is to use the *HTML5* canvas element that has been recently introduced in several browsers. We can create *JavaScript* objects for drawing, e.g., circles, lines, text on a canvas. Again, we have developed a prototype *R* graphics device that generates such code for an *R* plot. We can also annotate this to add animation and interaction.

We also mention the Vector Markup Language (VML) (`http://msdn.microsoft.com/en-us/library/bb263898(VS.85).aspx`) which is quite similar to *SVG*. It is used within Microsoft products such as Excel, Word and PowerPoint. However, it is not widely supported by other applications.

***R* Available at Viewing Time**

A fundamental aspect of what we have described with respect to the SVGAnnotation package is that *R* is used to create the display but is not available at viewing time when the *SVG* document is rendered. If *R* is available for the *SVG* viewer, then the *JavaScript*/*ECMAScript* code within an *SVG* or *HTML* document can make use of *R* at run-time. It can invoke *R* functions and access data to update the display in response to user interaction and animation. Additionally, we can use *R* code to manipulate the *SVG* and *HTML* content and so enable programming in both *JavaScript* and *R* at view time. One approach is to have *R* available on the server side of a client–server setup via, e.g., `rApache` [11]. Alternatively, *R* can be plugged into the *HTML* viewer and perform *R* computations on the client side. We are developing such an extension for Firefox [3] and other browsers that embeds *R* within a user's browser. This modernizes previous work in 2000 on the SNetscape package that embedded *R* and *R* graphics devices within the Netscape browser. We believe the combination of *SVG* (or *Flash* or the *HTML5* canvas) with *R* at viewing time will prove to be a very rich and flexible environment for creating new types of dynamic, interactive, and animated graphics and also allow Google Earth and Google Maps displays to be combined with *R* plots and computations.

The increasing importance of Web-based presentations is a space where statisticians need to be engaged. To accomplish this, we need more tools for creating these presentations. SVGAnnotation offers one approach; we are working on others. We ask that the reader think of this package as a rich starting point that enables a new mode of displaying *R* graphics in an interactive, dynamic manner on the Web. It establishes a foundation on which we and others can build even higher-level facilities for annotating *SVG* content and providing rich plots in several different media.

16.8 Summary of Functions in SVGAnnotation

There are approximately 60 functions available in SVGAnnotation for annotating plots. Among these are high-level functions that add interactivity through a single function call. These functions are listed below. Other, intermediate-level functions are also available. These roughly fall into two types: functions that find nodes in the *SVG* that correspond to particular parts of a plot, e.g., points, legend,

panel; and those that annotate or add a node. The functions that locate the nodes begin with the prefix `"get"` and those that annotate a node or add a node begin with `"add"`.

We recommend creating the *SVG* plot with the function svgPlot() in SVGAnnotation. It opens the *SVG* device, evaluates the commands to generate the plot, and closes the device. It also adds the *R* plotting commands to the *SVG* document and can either return the parsed *XML* document as a treelike structure or write it to a file.

linkPlots() Implement a simple linking mechanism of points within an *R* plot consisting of multiple subpanels/plots. When the mouse is moved over a point in a subplot, the color of that point and all corresponding points in other plots/panels change. The subplots can be created by, for example, arranging plots via the *mfrow* parameter in par(), via the function pairs(), or using splom() in the `lattice` package.

animate() Create an animated scatter plot where points in the scatter plot move from one location to the next in a synchronized fashion, possibly changing size and color as they move.

addAxesLinks() Associate target *URL*s with an axis label or plot title so that a mouse click on an axis label or title jumps to the specified URL in the viewer's Web browser.

addToolTips() Associate text with elements of the *SVG* document so that the text can be viewed in a tool tip when the mouse is placed over that object/element in the display. The default action of this function is to add tool tips to points in a scatter plot.

The following are a list of intermediate-level functions that are available in SVGAnnotation.

addLink() Associate target URLs with any *SVG* element so that a mouse click on an, e.g., axis label, title, point or some other plot component, displays the specified *URL* in the viewer's Web browser. When the *addArea* parameter is `TRUE`, the link is associated with a bounding rectangle around the *SVG* element.

addECMAScripts() and addCSS() Add *JavaScript* and *CSS* code to an *SVG* document, respectively. These functions either directly insert the content or put a reference (*href* attribute) to the file/URL in the document.

addSlider() Add an interactive slider to the *SVG* document. This function uses Carto:Net [5].

radioShowHide() Add radio buttons to the *SVG* document. This function uses Carto:Net.

getPlotPoints() Locate the *SVG* elements/nodes in the document that represent the "points" within the *R* graphic. These may be in multiple plots or panels, and they may be, e.g., the polygonal regions in a map or hexagons in a hexbin plot.

getAxesLabelNodes() Locate the *SVG* elements/nodes in the document that represent the text of the main title and the *x* and *y* axes for each subplot within the *R* graphic.

getPlotRegionNodes() Retrieve the *SVG* elements that house the contents of the data regions of the subplots within the display. This works for traditional and lattice plots, and also for histograms and bar plots.

getStyle(), setStyle(), modifyStyle() These functions query, set, and reorganize the value of a style attribute of an *SVG* node, respectively. Use these functions to determine and set the vector of *CSS*-based style settings.

getLatticeLegendNodes() Retrieve the legend in a lattice plot.

enlargeSVGViewBox() Change the dimensions of the viewing box so that extra elements can be added to the document and displayed within its viewing area, e.g., a slider below a plot or check boxes alongside a plot.

convertCSSStylesToSVG() Convert a *CSS style* attribute into presentation attributes. The purpose is to make it easier to change a single attribute in response to a user event. See Section 16.6.3 for a description of these types of style specifications.

asTextNode() Replace a <*g*> element (that contains directions for drawing text with <*path*> tags) with a <*text*> node. This makes it easier to modify text in a display. Although it does not have the benefit of scalability and the variety of fonts, it is much simpler for creating interactivity. See Section 16.6.1 for more details on the text created by *R*'s graphics system(s) and `libcairo`.

There are many more `"get"` functions available in SVGAnnotation. These can make it easier for the developer to build other high-level functions for annotation. Here is a list of all the get functions currently in the package:

```
 [1] "getAxesLabelNodes"      "getAxesLabelNodes.mosaic"
 [3] "getBoundingBox"         "getCategoryLabelNodes.mosaic"
 [5] "getCSS"                 "getECMAScript"
 [7] "getEdgeElements"        "getEdgeInfo"
 [9] "getGeneralPath"         "getJavaScript"
[11] "getLatticeLegendNodes" "getLatticeObject"
[13] "getMatplotSeries"       "getNodeElements"
[15] "getPanelCoordinates"    "getPanelDataNodes"
[17] "getPlotPoints"          "getPlotRegion"
[19] "getPlotRegionNodes"     "getRCommand"
[21] "getRect"                "getShape"
[23] "getStripNodes"          "getStyle"
[25] "getSVGNodeTypes"        "getTextNodes"
[27] "getTextPoints"          "getTopContainer"
[29] "getTopG"                "getUSR"
[31] "getViewBox"
```

16.9 Further Reading

For more detailed information on *SVG*, including its animation capabilities, consult [9]. To learn more about *HTML* consult [13], *JavaScript* read [10], cascading style sheets see [6, 15]. Lastly, the examples in this chapter and other examples are described in more detail in [21, 22].

References

[1] Adobe. Flash player software, version 10.3. http://get.adobe.com/flashplayer/, 2011.

[2] Apache Software Foundation. The Apache Batik project: A *Java*-based toolkit for applications or applets that want to use images in scalable vector graphics (*SVG*), version 1.7. http://xmlgraphics.apache.org/batik/, 2008.

[3] Gabriel Becker and Duncan Temple Lang. RBrowserPlugin: *R* in the Web browser. https://github.com/gmbecker/RFirefox, 2012. *R* package version 0.1-5.

[4] Richard Becker, Allan Wilks, Ray Brownrigg, and Thomas Minka. maps: Draw geographical maps. http://cran.r-project.org/web/packages/maps/, 2011. *R* package version 2.1.

[5] Alex Berger, Alex Pucher, Alexandra Medwedeff, Andreas Neumann, Andre Winter, Christian Furpass, Christian Resch, Florent Chuffart, Florian Jurgeit, Georg Held, Greg Sepesi, Iris Fibinger, Klaus Forster, Martin Galanda, Nedjo Rogers, Nicole Ueberschar, Peter Sykora, Sudhir Kumar Reddy Maddirala, Thomas Mailander, Till Voswinckel, Tobias Bruehlmeier, Torsten Ullrich, and Yvonne Barth. Carto.net software. http://www.carto.net, 2010.

[6] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. Cascading style sheets, level 2, revision 1 (CSS 2.1) specification. Worldwide Web Consortium, 2011. http://www.w3.org/TR/CSS2/.

[7] Cairo Graphics. Cairo: A 2D graphics library with support for multiple output devices, version 1.1. http://www.cairographics.org, 2006.

[8] Dan Carr, Nicholas Lewin-Koh, and Martin Maechler. hexbin: Hexagonal binning routines. http://www.bioconductor.org/packages/2.6/bioc/html/hexbin.html, 2009. *R* package version 1.22.

[9] J David Eisenberg. *SVG Essentials*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[10] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[11] Jeffrey Horner. rApache: Web application development with *R* and Apache. http://rapache.net/, 2011. *R* package version 1.2.1.

[12] Chafic Kazoun and Joey Lott. *Programming Flex 3: The Comprehensive Guide to Creating Rich Internet Applications with Adobe Flex*. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[13] Bill Kennedy and Chuck Musciano. *HTML and XHTML: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[14] Jake Luciani. RSvgDevice: An *RSVG* graphics device. http://cran.r-project.org/web/packages/RSvgDevice/, 2009. *R* package version 0.6.

[15] Eric A Meyer. *CSS Pocket Reference*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[16] Paul Murrell. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL, 2005.

[17] Paul Murrell. gridBase: Integration of base and grid graphics. http://www.stat.auckland.ac.nz/~paul/grid/grid.html, 2006. *R* package version 0.4.

[18] Paul Murrell. gridSVG: Export grid graphics as *SVG*. http://www.stat.auckland.ac.nz/~paul/R/gridSVG/gridSVG_0.5-10.tar.gz, 2010. *R* package version 0.5.

[19] Paul Murrell. grid: The grid graphics package. http://cran.r-project.org/package=grid, 2011. *R* package version 2.16.0.

[20] Andreas Neumann and Andre Winter. Vector-based Web cartography: Enable *SVG*. http://www.carto.net/papers/svg/index_e.shtml, 2003.

[21] Deborah Nolan and Duncan Temple Lang. SVGAnnotation: Tools for post-processing *SVG* plots created in *R*. http://www.omegahat.org/SVGAnnotation, 2011. *R* package version 0.9.

[22] Deborah Nolan and Duncan Temple Lang. Interactive and animated scalable vector graphics and *R* data displays. *Journal of Statistical Software*, 46:1–88, 2012.

[23] Dave Pawson. *XSL-FO: Making XML Look Good in Print*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[24] Tony Plate. RSVGTipsDevice: An *RSVG* graphics device with dynamic tips and hyperlink. http://cran.r-project.org/web/packages/RSVGTipsDevice/, 2011. *R* package version 1.0.

[25] Hans Rosling. Gapminder: World. http://www.gapminder.org/world, 2008.

[26] Barry Rowlingson. imagemap: Create *HTML* imagemaps. http://www.maths.lancs.ac.uk/Software/Imagemap/, 2004. *R* package version 0.9.

[27] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. http://lmdvr.r-forge.r-project.org/figures/figures.html.

[28] Deepayan Sarkar. lattice: Lattice graphics. http://cran.r-project.org/web/packages/lattice/, 2011. *R* package version 0.19.

[29] John Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[30] Debby Swayne, Dianne Cook, Duncan Temple Lang, and Andreas Buja. GGobi software, version 2.1. http://www.ggobi.org, 2010.

[31] Duncan Temple Lang. FlashMXML: A simple Flash graphics device for *R*. http://www.omegahat.org/FlashMXML, 2011. *R* package version 0.2-0.

[32] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[33] Martin Theus. Interactive data visualization using Mondrian. *Journal of Statistical Software*, 7:1–9, 2002.

[34] Simon Urbanek and Tobias Wichtrey. iplots: Interactive graphics for *R*. http://cran.r-project.org/web/packages/iplots/, 2011. *R* package version 1.1.

[35] W3Schools, Inc. *JavaScript* tutorial. http://www.w3schools.com/JS/default.asp, 2011.

[36] W3Schools.com. *SVG* tutorial. http://www.w3schools.com/svg/default.asp, 2011.

[37] Hadley Wickham. ggplot2: An implementation of the Grammar of Graphics. http://cran.r-project.org/web/packages/ggplot2/, 2010. *R* package version 0.8.

[38] Carl Worth and Keith Packard. Cairo: Cross-device rendering for vector graphics. http://cworth.org/~cworth/papers/xr_ols2003/, 2003.

[39] Yihui Xie. animation: A gallery of animations in statistics and utilities to create animations. http://cran.r-project.org/web/packages/animation/, 2011. *R* package version 2.0.

Chapter 17
Keyhole Markup Language

Abstract In this chapter, we explore two innovative tools for interactive visualization: Google Earth and Google Maps. Google Earth renders Keyhole Markup Language (*KML*) documents for viewing on a virtual earth browser, and similarly, Google Maps displays *KML*-formatted data on two-dimensional maps in a Web browser. *KML* is a grammar of *XML* for marking up spatial data. With it we can place plotting symbols on a map, augment these points with additional information, overlay *R* plots on the earth, animate points that have times associated with them, and draw arbitrary objects on the earth. We describe plotting functions in *R* that we have developed for creating *KML* displays with these features. We also discuss how to embed Google Earth in a Web page to create interactive mash-ups using *KML*, *SVG*, *HTML*, *JavaScript*, and *R*.

17.1 Introduction: Google Earth as a Graphics Device

Google Earth [5] and Google Maps [6] offer yet another example of *XML* because the instructions for rendering data on them are written in the Keyhole Markup Language (*KML*).[1] These interfaces provide exciting new ways to display spatial and spatial–temporal data, whether they are locations of homes, stores, and hiking paths, or more scientific data such as earthquake locations, predictions from climate models, and mash-ups of world health statistics. That is, Google Earth and Google Maps offer qualitatively different ways to interact with and explore data. To get a sense of the possibilities, suppose we are interested in creating a display of earthquakes. We can, for example, load into Google Earth a *KML* document that contains the locations of earthquakes that occurred around the world in a two-week time period, and Google Earth will render these locations as pushpins on the surface of the earth. We can augment each point with additional information so that when we click on its pushpin, a small window pops up with this information, e.g., the window might show the date the quake occurred or a scatter plot of the magnitude and depth of major earthquakes in that region over the past 100 years. If we want to examine the relationship between the location of the quakes and tectonic plates, we can layer onto the earth browser plate boundaries that the US Geological Survey has made available, also in *KML*. Google Earth renders these plate boundaries as paths on the surface of the earth. We can zoom in and rotate the earth to get a closer look at the earthquakes that occurred along, say, the Okhostk

[1] *KML* is an open standard maintained by the Open Geospatial Consortium, Inc. (OGC) [16] (http://www.opengeospatial.org/standards/kml/). Many applications display *KML*, including Google Earth, Google Maps, NASA WorldWind (http://worldwindcentral.com/wiki/Main_page), ESRI ArcGIS Explorer (http://www.esri.com/software/arcgis/explorer/index.html), AutoCAD (http://usa.autodesk.com/), and Yahoo! Pipes (http://pipes.yahoo.com/pipes/).

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_17, © Springer Science+Business Media New York 2014

Plate to the north of Japan. When zooming in, additional features, such as terrain, political boundaries, roads, and buildings, come into focus and the pushpins for the quakes appropriately scale to the new view. Moreover, we can mark the earthquake locations with symbols other than pushpins, e.g., circles that are color-coded to convey magnitude and scaled according to quake depth. Alternatively, we can add a spatial smooth of these data to the earth in the form of a three-dimensional surface or a heat map. Finally, if we arrange the earthquakes in folders by magnitude, then we can easily filter the view of the data by hiding/removing quakes with magnitudes, say, under 3.0.

We see from this description that the earth browser is a technological development that allows us to move substantively beyond the static plot. Moreover, Google Earth and Google Maps are well-established standards that make it easy to incorporate information from other applications. Both offer opportunities for interactive presentation-style spatial graphics that are not directly available in *R*. The RKML package [14] offers functionality to create displays for Google Earth and Google Maps. With RKML, we can produce plots on Google Earth for exploratory data analysis in an *R* session and to formally present results. The plots can be simple displays of spatial data, richer displays that incorporate additional information via styles and balloon windows, and more complex interactive displays via links between Google Earth, *R* plots, and *HTML* forms in Web pages.

As with many of the chapters in this book, this chapter is divided into two parts: 1) an introduction to high-level functions in the *R* package that offer basic functionality to create entire plots with one function call; and 2) descriptions of intermediate-level functions for customization. In Section 17.2, we describe the high-level functions available in RKML for creating Google Earth and Google Maps displays. These are kmlPoints() for geospatial displays, and kmlTime() for locations that have times associated with them. Then in Section 17.4, we describe the high-level function kml(), which is an overarching interface to kmlPoints() and kmlTime() that uses a formula interface. We have developed this high-level functionality in the spirit of *R*'s traditional graphics functions for creating data visualizations. After describing the core elements of *KML* in Section 17.5, we discuss (in Section 17.6) how to create more customized Google Earth documents that require us to work more directly with *KML* elements. In Section 17.6, we also demonstrate how to use the graphics device in the RKMLDevice package [20] to create any *R* plot in *KML* and have it drawn directly on the virtual earth. In the final examples of this chapter (Section 17.7), we show how to create interactive displays in Web pages containing *SVG* plots, Google Earth plug-ins, *HTML* buttons and forms, and *JavaScript* for communication amongst these various elements of the Web page.

Other work in this area includes the `RgoogleMaps` [12], R2GoogleMaps [21], and `plotKML` [7] packages. With `RgoogleMaps` we can download directly into *R* a static image from Google Maps of a rectangular latitude–longitude region. We can then use this static image for a background on which to overlay *R* plots and provide some geographical context for the plot. The result is a static plot within *R*. The R2GoogleMaps package works in the opposite direction. With the functions it provides, we can process data in *R* to generate a display that can be viewed in a Web browser that presents the data via Google Maps. This generates both an *HTML* document and *JavaScript* code that maps the data by latitude and longitude pairs. In short, this package exports a display from *R* to Google Maps similar to RKML, but in a different format. The `plotKML` package provides methods to write `Spatial` objects from the `sp` package [17] and `Raster` objects from the `raster` package [8] into *KML*.

17.1.1 The Google Earth and Google Maps Interfaces

In this section, we introduce briefly the Google Earth and Google Maps tools. If you are already acquainted with Google Earth and Google Maps, you may want to simply review the summary of features of a virtual earth browser (shown in the box below) before continuing to the next section.

With Google Earth and Google Maps, we can interact with the globe to examine particular locations that are denoted by what are called "placemarks." Figure 17.1 annotates the controls found in the Google Earth browser. The browser interface has three basic parts: the main 3D viewer where you see a view of the earth and the placemarks that have been added to it; the sidebar where you can manage the display of placemarks and other elements on the earth; and the tool bar across the top of the browser for tools to, e.g., add new graphical elements. In the 3D viewer, we find the navigation controls (on the right side) for adjusting the view, e.g., moving about the earth, zooming in and out, and changing the orientation of the view. Not shown in the figure are the time controls that allow us, as users, to restrict the display of placemarks to a particular time period. The time controller appears in the 3D viewer when times are associated with placemarks or other graphical elements. It is a slider with two thumbs. We can use the thumbs to restrict the display of placemarks to user-specified windows of time, and we can run animations by having the thumbs automatically move across the slider making placemarks appear and disappear as their associated times enter and leave this moving time window. More information about the Google Earth interface, including video tutorials, is available at `http://www.google.com/earth/learn/index.html` and `http://earth.google.com/support/`.

Summary of Features Available in a Virtual Earth Browser

In general, virtual earth browsers offer many attractive, easy-to-use features for interacting with spatial data. These include:

- navigation controls to rotate, zoom in on, and spin the virtual earth;
- time controls to limit the display of placemarks to a window of time and to run animations where placemarks appear and disappear as this window moves across a slider;
- folders for layering and integrating data of different types and from various sources;
- styles for specifying the appearance of placemarks, lines, and polygons, e.g., color, scale, plotting symbol;
- pop-up balloon windows that provide additional information about a placemark or data folder, and can include rich *HTML* content;
- legends that can be located on the view port and so remain visible as the virtual earth rotates and zooms;
- ground overlays for draping an image (e.g., a PNG/JPEG file) on the earth, such as a smooth surface from kriging;
- complex graphical objects, such as representations of buildings that are constructed from composites of graphical elements and ground overlays that are made visible at specific elevations.

Google Earth is usually viewed in a stand-alone app, but can be embedded in a Web page too. Embedding Google Earth in a Web page gives us a lot of leverage to combine different technologies, e.g., *SVG*, *HTML*, and *JavaScript*, to create novel interactive displays. The topic of how to embed Google Earth in a Web page will be addressed in Section 17.7.

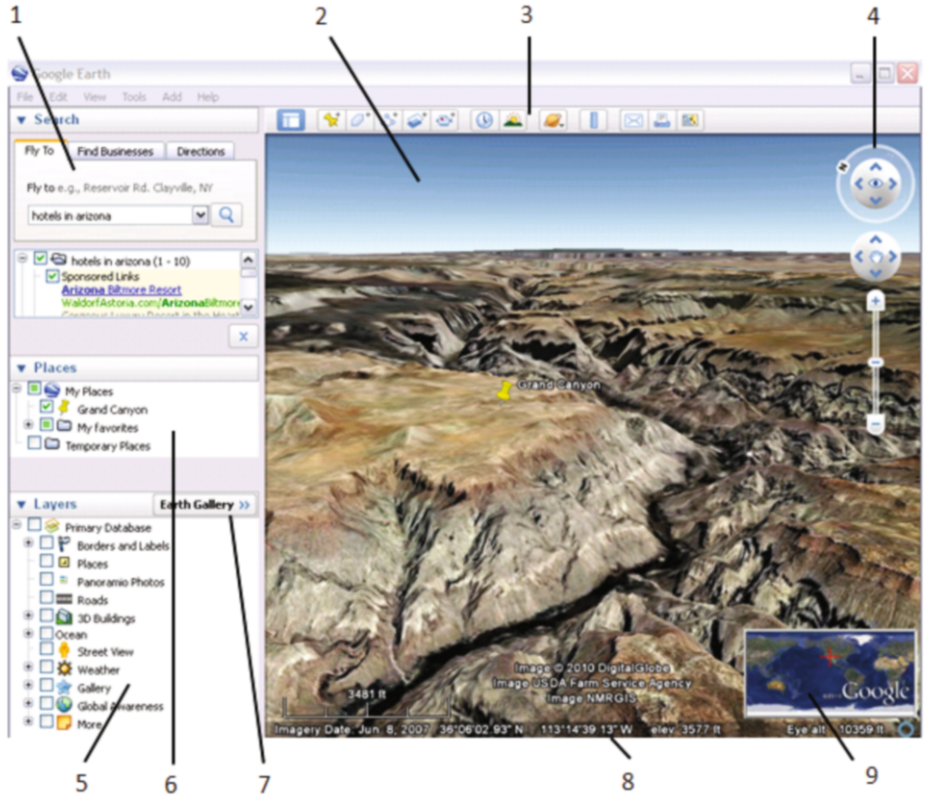


Figure 17.1: Google Earth Interface. This screenshot is from Google Earth's help pages, which are available at `http://earth.google.com/support`.

1. The Search panel is used to find places on the earth (not elements in Google Earth) and manage search results. To find a specific place, enter its address or description, e.g., `"white house"`, into the `"Fly To"` tab in this panel.
2. The 3D viewer displays a portion of the globe. The user can change the view, zooming in and out and moving to nearby regions, with the navigation controls described in item 4 below.
3. The tool bar along the top of the browser contains buttons to hide or display the sidebar, and to add a placemark, polygon, path, or image overlay to the globe. There are also buttons to record a "tour," display historical images, add sunlight to the earth, view the sky, measure a distance, email a view, print, and show the view in Google Maps.
4. The navigation controls along the right side of the 3D viewer consist of, from top to bottom: a circular "look joystick" to orbit around the center of the current view; a "move joystick" to fly around the globe to different locations; and a vertical "zoom slider" to change the elevation and hence resolution.
5. With the Layers panel, the user can add and remove points of interest that Google Earth makes available to all users, such as political borders, roads, and 3D buildings.
6. When a placemark is added to the globe (e.g., the yellow pushpin marking the Grand Canyon in this screenshot), information about it appears in the Places panel, and when a *KML* document is opened, the document and folders within the document are displayed in the Places panel. Folders

contain collections of placemarks (and other graphical elements), and the Places panel is a hierarchical viewer used to organize and show/hide individual placemarks and collections of them.
7. The Earth Gallery button is for importing content from the Earth Gallery, a collection of *KML* documents created and contributed by the *KML* users community.
8. The status bar at the bottom of the 3D viewer displays the current latitude, longitude, and elevation of the mouse. It also displays the status of downloads as they happen.
9. The overview map provides an additional global perspective about the current local view, e.g., it locates the region shown in the 3D viewer on the 2D map of the entire earth.

Google Maps offers an interactive 2D view of the world that is contained in a Web browser, i.e., it provides a flat map in a Web browser. We can display *KML* content in either Google Earth or Google Maps. We can load a *KML* document into Google Maps by 1) visiting `http://maps.google.com`, 2) entering the *URL* of the file in the textfield, and 3) hitting the blue `"Search Maps"` button with the magnifying glass icon. Alternatively, we can create an *HTML* document that uses *JavaScript* to create the graphical elements to be displayed on Google Maps. The maps come with interactive controls for navigation and layering, and buttons that allow the user to view the file in Google Earth, print or email the map, or copy its link location. These controls are briefly described in Figure 17.2.

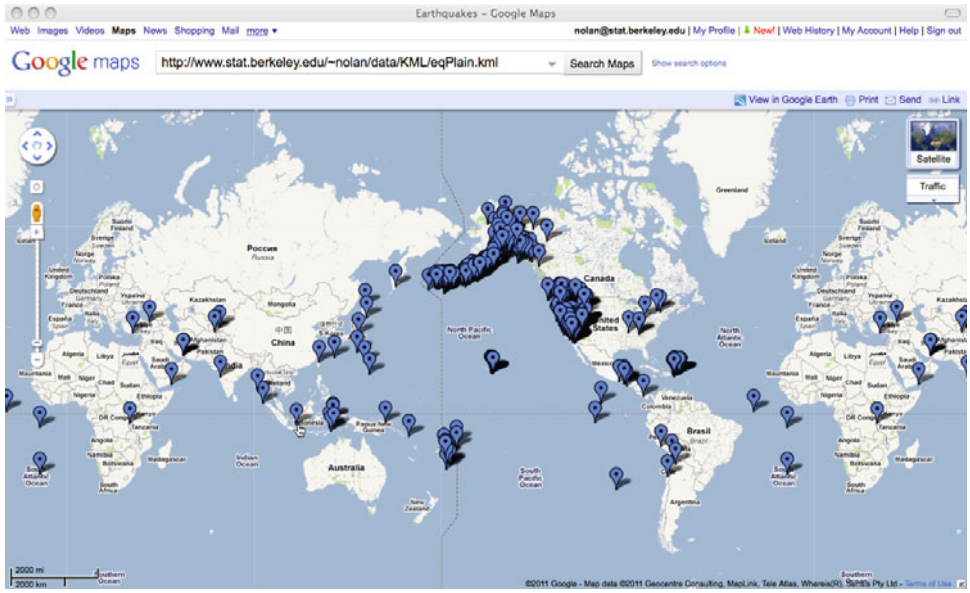


Figure 17.2: Google Maps Interface. This screenshot of Google Maps displays the locations of earthquakes over a 7-day period. These data were retrieved from the USGS Web site [25] at `http://earthquake.usgs.gov/earthquakes/catalogs/merged_catalog.xml.gz`. The *KML* document displayed here was created in *R* using the kmlPoints() function in RKML. Each paddle denotes the location of an earthquake. The map also contains many user controls. On the left side, navigation controls allow for flying to other locations and zooming; the images on the top right of the map enable the addition of traffic, photos, and terrain information; the buttons in the bar along the top of the map provide options to view the file in Google Earth, print or email the map, or copy the link location of the map.

The functions kmlPoints(), kmlTime(), and kml() in RKML can create *KML* documents for display on Google Earth or Google Maps. After they have been created, we can open these *KML* documents in Google Earth or Google Maps and we have the navigation tools and other forms of interactivity immediately available to us. Viewers can then conduct their own exploration of these data. In the next section, we describe the simple process for creating these *KML* documents in *R*.

17.2 Simple Displays of Spatial Data

In developing the model for creating *KML* documents in *R*, we have tried to be consistent with *R*'s plot() function. We imagine Google Earth as an open graphics device where the plot already exists but is empty. The coordinate system of the plot is longitude (x) and latitude (y), with x ranging from -180 to 180 and y from -90 to 90. Later, the viewer can change the coordinate system by zooming in and out, but this is our starting point. This situation is entirely similar to having used the `type="n"` argument to plot() to create the canvas, and the next step is to call points() to add points to the plot. We have a parallel situation with Google Earth as our "open" graphics device, and our next step is to call the kmlPoints() function to create a *KML* document that holds the plotting instructions to add points to Google Earth and Google Maps. Similarly for data with a time component, kmlTime() includes temporal information with each point added to the plot. This temporal information allows the user to watch a movielike animation of the points appearing and disappearing over time.

The functions kmlPoints() and kmlTime() provide high-level functionality that make it easy to create *KML* documents for display on Google Earth. There is no need to know anything about the *KML* grammar to use these functions. A third high-level function is kml(). It provides a formula interface to kmlPoints() and kmlTime(). In this section, we provide examples of how to use kmlPoints() and kmlTime() and in Section 17.4 we demonstrate kml(). Those who want to create custom plots will need to know more about *KML*, which is the subject of Section 17.5.

17.2.1 Adding Points to the Google Earth and Google Maps Canvas

To create a *KML* document for display on Google Earth or Google Maps, we need only provide kmlPoints() with a data frame of latitude and longitude pairs. The basic call in pseudo code is

```
doc = kmlPoints(dataFrame)
saveXML(doc, "kmlDoc.kml")
```

The `dataFrame` variable is an *R* data frame object and must contain *numeric* vectors named `longitude` and `latitude`. The kmlPoints() function creates a *KML* object using these lon–lat pairs for the locations of placemarks. The return object, `doc`, from the call to kmlPoints() is a *KML* parsed document in *R*. We serialize it to the file `kmlDoc.kml` with the call to saveXML(). Then we can load `kmlDoc.kml` directly into Google Earth or Google Maps for viewing. Since we have not specified an icon to use for the placemarks, the default plotting symbol will be used, i.e., a yellow pushpin for Google Earth and blue paddle for Google Maps. The next example demonstrates this.

Example 17-1 Plotting Earthquake Locations as Paddles on Google Maps

We create the display of the locations of earthquakes for Google Earth and Google Maps shown in Figure 17.2. The locations of the quakes are provided in the variables `longitude` and `latitude` in the data frame `quakes` available in the RKML package. We call the kmlPoints() function as

```
doc = kmlPoints(quakes, docName = "Earthquakes")
```

The *docName* value goes in the *KML* document and appears in the Places panel as the label for the document; it has nothing to do with the file name of the document.

We save `doc` with

```
saveXML(doc, "eqPlain.kml")
```

After it has been saved, we can load `eqPlain.kml` into Google Earth immediately for viewing. Also, we can treat Google Earth as a live, "in session" graphics device of sorts that displays the plot when we create it in *R*. That is, we can open the document in Google Earth from within *R* with a system call on OS X, such as,

```
system("open eqPlain.kml")
```

and something similar on other operating systems. In Windows, we can also communicate directly with a Google Earth process using *DCOM* (see [1, 22]). Lastly, to view this *KML* document in Google Maps, we must first place this file on a Web site as described in Section 17.1.1.

17.2.2 Associating Time with Points

Placemarks as well as other features in *KML* can have time values associated with them. When this is the case, Google Earth automatically provides a slider and play button in the top left corner of its 3D viewer. The slider contains two thumbs that correspond to the endpoints of a time window. Those placemarks with a time that falls between the two thumbs are displayed and the other placemarks are hidden. To change the placemarks that are visible, you can change the location and size of the time window. You do this by dragging the entire time window along the slider or by moving one or both of the thumbs.

We can also display "animations" using the play button that accompanies the time slider. When we click on the play button, the time window starts to automatically move along the slider. As the time window moves, placemarks are revealed and hidden when they enter and exit the time window, respectively. This activity creates an animation of sorts.

Like kmlPoints(), the high-level function kmlTime() creates the *KML* necessary for displaying places on Google Earth, with the main difference being that it accepts times associated with the lon–lat pairs. This time-related information can be passed to kmlTime() as the vector `time` or `date` in the input data frame, or separately via the function's *times* parameter. The format for time can be either `POSIX` or the *S3* class `Dates`. Also, unless otherwise specified, the time vector is used to label the placemarks. Since these times can create rather long labels, we can specify alternative labels with the *name* parameter. The next example shows how to create an animation using kmlTime().

Example 17-2 Plotting Elephant Seal Locations on Google Earth

Here we create a *KML* document containing the movements of a female elephant seal as she forages in the North Pacific Ocean. The data (available from [3]) contain the daily locations of a single elephant seal over a 75-day migration; they are available in the data frame `elephantSeal` in the RKML package. Since we have a time associated with each location, we can animate the seal's journey. To do this, we anchor the left thumb of the time window at the left end of the slider so only the right thumb moves when we click on the play button. This way, the time window grows larger as the right thumb moves across the slider, and once a placemark appears, it remains visible for the remainder of

the animation. Figure 17.3, displays a screenshot of the animation taken on the fifteenth day of the journey, March 9, 1992.

We have the dates available in two formats in the data frame; `day` is the day of the year according to the Julian calendar (the observations range from day 54 to 128) and `date` is in `POSIXt` format.

```
head(elephantSeal)
```

```
  day latitude longitude       date
1  54     34.0    -120.0 1992-02-24
2  55     35.0    -121.3 1992-02-25
3  56     36.0    -122.6 1992-02-26
4  57     36.5    -123.6 1992-02-27
5  58     36.8    -124.6 1992-02-28
6  59     36.9    -125.4 1992-02-29
```

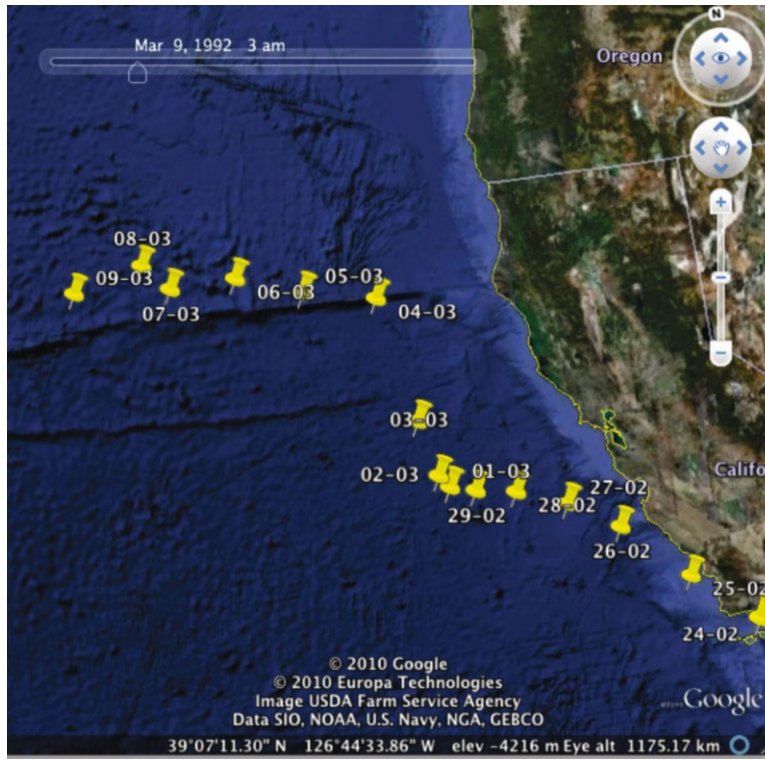


Figure 17.3: Google Earth Display of an Elephant Seal's Daily Location. This Google Earth screenshot displays the daily location of an elephant seal. Each yellow pushpin denotes the location of the seal and the text next to the pushpin gives the date in dd-mm format. The slider at the top left of the viewer shows the right thumb located at March 9, 1992, fifteen days into the seal's 75-day journey. We anchored the left thumb of the time window to the left end of the slider so that the placemarks remain displayed once revealed.

Our first step is to make a short date format to use as a label for the placemarks. We do this with

```
dateNames = format(elephantSeal$date, "%d-%m")
```

Next we call kmlTime() and provide the shortened label via its *name* parameter as follows:

```
doc = kmlTime(elephantSeal, name = dateNames)
saveXML(doc, "simpleElephantSeal.kml")
```

Again, after we serialize `doc` to text with saveXML(), we can load it into Google Earth to view the placemarks and animate them.

17.2.3 Using Styles to Customize Graphical Elements

We can imagine wanting to change the appearance of placemarks from the default yellow pushpin and blue paddle to more interesting and informative icons. For example, we may want to specify the color and size of the plotting symbol to convey auxiliary information. We may also want to control the appearance of the information about a placemark in its pop-up balloon window. We can supply all of this style information via arguments to kmlPoints() and kmlTime(), and as a result, customize a display to be more informative and aesthetic. For example, we can change the elephant seal animation shown in Figure 17.3 by substituting the standard yellow pushpin with paddle icons that are colored according to whether the seal is heading out to sea or back to the rookery. We also can connect the paddles with line segments to make it easier to see the path the seal takes. See Figure 17.4 for a revised display. As another example, we can replace the paddles that mark the earthquakes shown in Figure 17.2 with circles that descend the yellow/orange/red color spectrum according to the magnitude of the quake, and we can scale the icons according to quake depth, as shown in Figure 17.5. We demonstrate how to create these more informative displays in this section.

In both examples, we want to use the same style settings for many placemarks. *KML* supports this by having a collection of style descriptions in a separate part of the document that act as a dictionary of styles. The approach is similar to styles in word processors and somewhat similar to styles in cascading style sheets. We first create these styles and give them unique identifiers. Then, we use these identifiers to associate a style with each placemark in calls to kmlPoints() and kmlTime(). The basic call in pseudo code is

```
kmlPoints(dataFrame, docStyles = styleList, style = styleIds)
```

Here `styleList` holds a named list of styles that will be stored at the top level of the document and available for use by all the graphical elements. The variable `styleIds` is a vector of style names that associates each lon–lat pair in our data frame with a particular style. We use the *style* parameter here in a similar, but slightly more indirect, way that we use the *col* and *pch* parameters in plot(). The indirection comes from specifying the style name rather than the actual style information. However, it is also possible to specify style details in *style*. We prefer instead to refer to the styles by name and use *docStyles* to avoid redundancy in the *KML* document. Also, this way, we can combine styles.

Again, the benefits of using these document-level styles include the fact that we do not repeat the same information for multiple elements with the same appearance. This reduces the size of the file, avoiding redundant information, and also greatly simplifies changing the appearance of points by only having to change the information in one location in the file. We provide a few more details about how to describe a *KML* style before delving into the examples.

We use three basic style settings in kmlPoints() and kmlTime(); these are referred to as "IconStyle", "LineStyle", and "BalloonStyle". The IconStyle holds information about the icon that represents the placemark, possibly including the name of an image file for the icon and a scaling factor. LineStyle provides style information for line segments, such as color and width. The appearance of the pop-up balloon window is controlled via a BalloonStyle; it holds specifications for the background color of the window, text color, the text itself, etc. The most common style settings for these three types of styles are summarized in Table 17.1.

Table 17.1: Style Specifications for *KML* Features

Name	Feature	Style Specifications
IconStyle	Point	**Icon** An href that points to either a local file or a *URL* of the image file to be used as the placemark icon. **scale** A numeric value used to scale the icon.
LineStyle	Line segment	**width** A numeric value that denotes the width (in pixels) of the line segment. **color** The color of the line segment. Note that colors are specified in hexadecimal format of the form ABGR, where A is the alpha (opacity) level, and the others are blue–green–red values. This is different from the more common RGBA format that *R* supports. The rgbToKMLColor() function in RKML converts a named *R* color or a color in RGB or RGBA format into that expected by *KML*.
BalloonStyle	Pop-up window	**text** A template for the text content of the balloon window. The content can be plain text or *HTML* and will be appropriately formatted and displayed as in a Web browser. This style supports text substitution for the contents of the pop-up window using the placemark's content. When `"$[name]"` and `"$[description]"` appear in the text, the placemark's name and description elements are substituted, respectively, for these placeholders. This notation, with the use of `"$"`, e.g., `"$[name]"`, has nothing to do with *R* syntax. **bgColor** The background color (ABGR format) for the balloon window. **textColor** The color (ABGR format) of the text in the balloon window.

This is a brief description of the different types of styles in KML. The styles control the appearance of the different KML objects when they are displayed. Elements of a centralized style can be used, merged and overridden within a particular element.

17.2.3.1 Styles for Placemarks and Lines

To create a *KML* style, we specify the style's details as a list with named elements that match the names in Table 17.1. This is like the specification of options for lattice graphics. The functions in RKML convert this style information into the format expected by *KML*.

In the next example, we demonstrate how to use IconStyle, LineStyle, and BalloonStyle to create four styles that are used in Figure 17.4. We use these styles to distinguish between the placemarks when the seal is headed away from and returning to the rookery. We use green paddles and line segments when the seal heads away from shore and red ones when she turns around and heads back. In addition, the balloon window for each placemark contains the placemark's description element.

Example 17-3 Customizing a KML Display of an Elephant Seal's Movements

We will augment the Google Earth animation from Example 17-2 (page 587) with style settings. The revised display appears in Figure 17.4. These settings are in the *list*, `esStyles`. The names of the style elements in `esStyles` act as the unique identifiers for each style. These are

```
names(esStyles)
```

```
[1] "paddle_out" "paddle_back" "line_out" "line_back"
```

The `paddle_out` element is

```
esStyles[["paddle_out"]]

$IconStyle
$IconStyle$Icon
                                                    href
"http://maps.google.com/mapfiles/.../grn-blank.png"

$BalloonStyle
$BalloonStyle$text
[1] "Date: $[description] 1992"
```

We see that `paddle_out` combines both IconStyle and BalloonStyle information. The IconStyle specifies the *URL* of a green paddle provided by Google Maps, which is to be used as the icon. Google Maps provides a set of different colored paddles for this purpose. Also notice that the BalloonStyle contains the placeholder `"$[description]"` so we can easily tailor the balloon window for each placemark. Lastly, `line_out` specifies a color in hexadecimal format (ABGR values) and width (in pixels) for the line as

```
esStyles[["line_out"]]

$LineStyle
$LineStyle$color
[1] "ff8adfb2"

$LineStyle$width
[1] 2
```

We again use kmlTime() to prepare the *KML* document for display in Google Earth. However, we now provide additional style information to the function. To determine which style to use for each placemark, we use the *factor* `direction`, which indicates whether each lon–lat pair is outbound or returning. We create the display with the call

```
doc = kmlTime(elephantSeal,
              name = format(elephantSeal$date, "%d-%m"),
              docStyles = esStyles,
              style = c("paddle_out", "paddle_back")[direction],
              lty = c("line_out", "line_back")[direction],
              addLines = TRUE )
```

Here we have passed our *list* `esStyles` to kmlTime() via *docStyles*. This argument provides styles at the document level so we can use them with any placemark or other graphical element. We customize each placemark according to the name of the style found in the *style* parameter. We also provide the styles for the line segments that connect the placemarks via the *lty* argument, and we indicate that the points are to be connected with line segments by setting *addLines* to `TRUE`. Recall that the values in *name* specify the labels for the placemarks.

Figure 17.4: Customized Placemarks and Line Segments for an Elephant Seal's Journey. This screenshot captures the animation of an elephant seal's movements. The placemarks are green paddles rather than the default yellow pushpin (see Figure 17.3), and they are connected by green line segments. The paddle to use for each placemark and the details for the appearance of each line segment are specified in style elements within the *KML* document.

17.2.3.2 Creating Icons in *R* and Using *HTML* in Pop-up Windows

In the previous section, we used icons made available by Google for representing placemarks and we constructed simple names and descriptions for the placemarks. (The name and description of a placemark appear in the pop-up window when the placemark's icon is clicked.) In this section, we show how these icons and descriptions can be further customized using *R*. For example, we can use *R*'s plotting functions to draw any image we want and use it as an icon. We also can construct contextual information for a placemark's description by using additional data that we have about each observation. This information can be formatted in *HTML* and we can even include plots we made in *R*. The following example demonstrates these capabilities.

Example 17-4 Annotating Earthquake Locations in KML with Depth and Magnitude

We enhance the Google Earth display from Example 17-1 (page 586) by swapping the pushpin for a circular icon that we construct in *R*. The color and size of each earthquake's circle is determined by its magnitude and depth, respectively. We also supply more detailed information about each earthquake than just its date. We construct this information in *R* using available data about the quakes, and we format this information in *HTML* for the pop-up window. In addition, we provide more information

about the source for the data, and this is presented in Google Earth's Places panel. See Figure 17.5 for a screenshot of the display of the modified *KML* document.

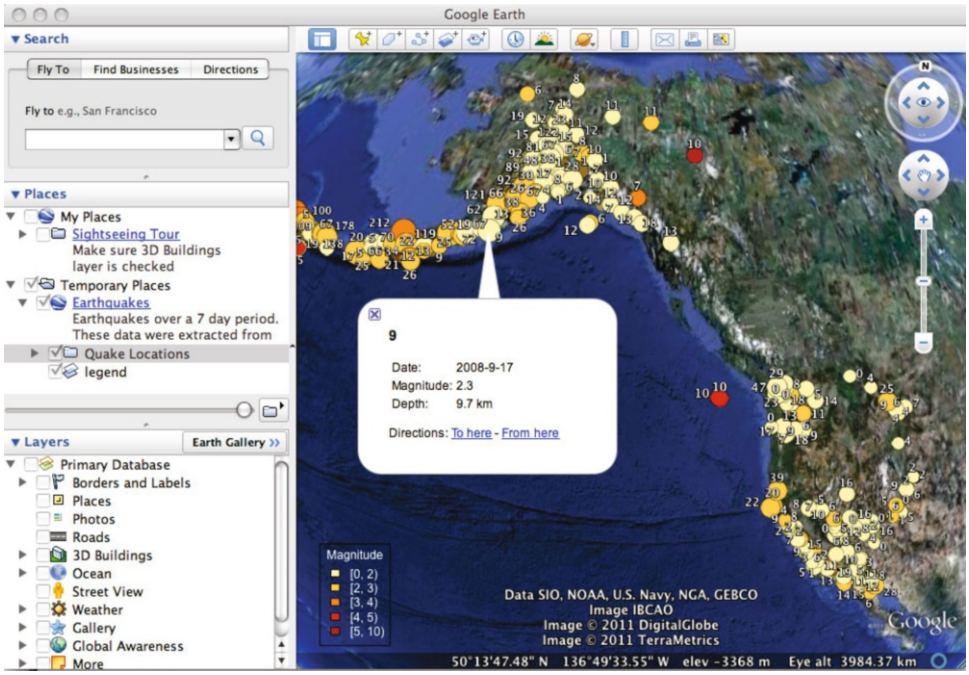


Figure 17.5: Customized Icons and Balloon Windows for Earthquakes in Google Earth. The screenshot shown here displays the locations of earthquakes with icons we have created in *R*.
We created five images, each a different color, in *R* and created 25 styles corresponding to all possible combinations of five colors and five scales. The color and size of the circle provide information about the earthquake's magnitude and depth, respectively. Also, each placemark has a balloon window with additional information about the quake, formatted as an *HTML* table. Note the legend in the lower left corner of the display. The legend belongs to the "view box" of the display, meaning that when the user rotates the earth or zooms in, the legend remains fixed on the 3D viewer.

We begin by using *R* to make the image files for the icons. Since we are using *R*, we can draw anything for an icon. Here, we make five circles of different colors that will correspond to five levels of magnitude. These icons are circles that range in color from yellow to orange to red along the `"YlOrRd"` palette in `RColorBrewer` [13] with the file `yor1ball.png` being pale yellow and `yor5ball.png` dark red. Pseudo code to do this is

```
mapply(makeBallIcon, quakecols, filenames,
       MoreArgs = list(width = 60))
```

The makeBallIcon() function is available in RKML for making circles of specified sizes and colors. Rather than make circles of different sizes, we will use the scale element in IconStyle to adjust the size of the icon. That is, we will use these five circles to create 25 styles, where each circle is used five times, once each with five different scales that correspond to five levels of earthquake depth.

We have placed these 25 styles in the *list* `ballStyles`. They have names `"ball_m-k"` where `"m"` ranges from 1 to 5 and corresponds to magnitude and `"k"` also ranges from 1 to 5 but corresponds

to depth. We examine the first element of ballStyles which will be used to represent earthquakes that fall into the smallest magnitude and depth ranges

```
ballStyles[["ball_1-1"]]

$IconStyle
$IconStyle$scale
[1] "0.525"

$IconStyle$Icon
          href
"yor1ball.png"
```

We see that placemarks for earthquakes with the smallest depth will have a scale of 0.525 and those with the smallest magnitude will be represented by the circle in yor1ball.png.

With these style settings in hand, we match each observation to the name for its appropriate style. To do this we use the cut() function to discretize magnitude and depth in quakes. The binned values of magnitude and depth for each observation are in magCut and depthCut, respectively. Together these values determine which style to use for each placemark. The pseudo code for this follows:

```
quakeStyle = sprintf("ball_%d-%d", magCut, depthCut)
```

We now have the list of 25 styles in ballStyles and the associated style for each lon–lat pair in quakeStyle. We pass kmlPoints() the ballStyles list as the value of *docStyles* so it is placed at the top level of the document for reference by all placemarks. We also pass in *style* the associated style for each placemark. We call kmlPoints() with

```
kmlPoints(quakes, docStyles = ballStyles, style = quakeStyle)
```

However, before we create this *KML* document, we want to further augment the display with more informative descriptions for the pop-up windows and Places panel.

Each placemark's description will be displayed in a pop-up balloon window when the viewer clicks on the placemark. We construct a character vector that holds contextual information about each point and is formatted as an *HTML* table. We do this with

```
ptDescriptions = sprintf(paste(
   "<table><tr><td>Date:</td><td>%s-%s-%s</td></tr>",
   "<tr><td>Magnitude:</td><td>%s</td></tr>",
   "<tr><td>Depth:</td><td>%s km</td></tr></table>"),
         quakes$year, quakes$month, quakes$day,
         quakes$magnitude, quakes$depth)
```

The functions in R2HTML [11] can also be used to create an *HTML* table from a data frame. See Figure 17.5 for a screenshot that shows one of these pop-up windows.

We also set up a few additional strings that will customize the document-level information, e.g., the document name and description and the folder name that appear in the Places panel. We create these strings as

```
docName = "Earthquakes"
docDescription = "Earthquakes over a 7 day period... "
folderName = "Quake Locations"
```

The `docName` variable has a label used for the *KML* document, `docDescription` offers a brief description of the *KML* document that appears below the document label in the Places panel, and `folderName` has a label for the collection of placemarks within the document.

We are now ready to create the *KML* document. The following call to kmlPoints() supplies all the information we have created to customize the display as well as the essential lon–lat pairs:

```
doc = kmlPoints(quakes, docName = docName,
                docDescription = docDescription,
                docStyles = ballStyles,
                folderName = folderName,
                style = quakeStyle,
                description = ptDescriptions,
                .names = floor(quakes$depth))
```

Note that we override the default label that appears next to each placemark with the argument *.names*. In this case, we supply the depth of the earthquake for the placemark label.

Before saving our document, we add to it a legend for the colors. We use the kmlLegend() function, which has arguments that are similar to those of the regular legend() function in *R*. We call kmlLegend() and pass it argument values that indicate where to place the legend on the 3D viewer, the colors to fill the boxes, and the labels associated with each color, among other things. The pseudo code appears as

```
kmlLegend(x = 20, y = 20, title = "Magnitude",
          legend = magLegend, fill = quakecols,
          text.col = "white", dims = c(100, 108), parent = doc)
```

Google Earth adds the legend as an overlay to the viewing window. The *x* and *y* argument values in this function call provide the coordinates in pixels for the location of the legend. The coordinates of the legend are not supplied in longitude and latitude because the legend remains fixed in the 3D viewer so it can be seen in all views of the earth. The legend also remains the same size as the display zooms in and out. For this reason, it does not make sense to create a legend for circle size. The *parent* argument to kmlLegend() provides the *KML* document or node to which the legend is to be added.

Finally, we save the document for viewing in Google Earth with

```
saveXML(doc, "eqFancy.kml")
```

Although, we have given the file the name `"eqFancy.kml"`, it will be named `"eqFancy.kmz"`. The **kmz** extension means the file is a zipped collection of files; these are the *KML* file and the five PNG files for the icons, e.g., `yor1.png` is one of the PNG files. This file format and the reasons for using it are described in Section 17.3.

17.3 Zipped *KML* Documents

If our *KML* document is very large or if we use local image files, then we typically want to **zip** the *KML* document. The **zip**ed file will include any accompanying image files. It is simpler to use because then we work with one, smaller file when we transfer files, i.e., publish to the Web and download for viewing. Google Earth has created the **kmz** file extension for this purpose. A **kmz** file is a **zip**ed

archive that contains the original **kml** file, which is renamed as `doc.kml` within the archive, and any locally referenced image files. The **zip** format is widely used and portable.

The saveXML() function **zip**s the file in memory and writes only the **kmz** file to disk. It checks all references to icons, etc. and **zip**s the *KML* document if any of these references are to local files. This function uses the Rcompression package [24] to create the **kmz** file. Also, with the updateArchiveFiles() function, we can add extra files to the **kmz** archive, such as the *R* data used to created the document.

Since the **kmz** file is a **zip** file, we can un**zip** it and extract all our individual files from it. To do this, we may need to change the extension to **zip** in order to use it with some software.

There are occasions when we do not want saveXML() to create a **zip** file, such as when we are editing a local file that we plan to post-process. To avoid saveXML() from **zip**ing the document, we can use the I() function to tell it to use the name "as is", e.g.,

```
saveXML(doc, I("eqFancy.kml"))
```

Alternatively, we can pass `FALSE` as the value of the *useKMZ* argument, e.g.,

```
saveXML(doc, "eqFancy.kml", useKMZ = FALSE)
```

17.4 A Formula Language for Making *KML* Plots

For convenience, kmlPoints() and kmlTime() allow the caller to specify the latitude and longitude of the locations either as named elements of a data frame or by explicitly passing them via the *.longitude* and *.latitude* arguments. It is simpler to pass a data frame, but this requires that the location variables be named `"longitude"` and `"latitude"`. Abbreviations such as `"lon"` and `"lat"` are not supported. An alternative and richer approach is to provide the information identifying the longitude and latitude via a formula. *R* users will be familiar with the formula interface used in both the modeling framework and the `lattice` graphics systems [18]. In `lattice`, the formula identifies the relationship between the variables being plotted, and it allows for conditioning variables that create separate panels for the different levels of these conditioning variables. We have developed in RKML the high-level function kml() that accepts a formula. The formula expresses relationships for spatial and spatial–temporal data that extends the syntax of the traditional formula language in *R*. This spatial formula language adapts the notions of the modeling symbol `~` and the conditioning symbol `|` to the Google Earth graphics model, and also introduces the new symbol `@` to express temporal relationships.

In the extended syntax, the formula `~long + lat` encapsulates the notion of locations of placemarks in the coordinate system of the virtual earth. Generally, this formula expects two variables to be specified to the right of the `~` symbol, where the first variable identifies longitude and the second latitude. The kml() function finds these variables by first looking in the data frame and then in the call frame of the caller.

Unlike with plots in `lattice`, there are no panels in Google Earth. Therefore, we use the conditioning variables to the right of `|` in a formula to divide the placemarks into groups that we organize into folders. That is, the levels of the conditioning variable are used to partition observations into folders. These folders appear in the Places panel and allow us to toggle on and off a collection of placemarks. We show how to use the `|` in Section 17.4.2.

Our formula language has the additional concept of time which we represent with `@`. Of course, the `@` and `|` symbols can be used together in a formula, e.g,

```
~ longitude + latitude @ date | cond1 + cond2
```

The order is important, with the `@` preceding the `|` operator. Additionally, one can use parentheses in the formula. In the next section, we demonstrate how to use `@` in a formula in kml().

17.4.1 Including Time in the Formula for Geospatial–Temporal Plots

The formula in the following pseudo code

```
kml(~ longitude + latitude @ time, data)
```

provides a date (in the vector `time`) for each longitude–latitude pair. As an example, we show how to use kml() with a formula to recreate the display of the journey of the elephant seal from Example 17-3 (page 590). The locations and time for the elephant seal journey are all in the data frame `elephantSeal` and are `longitude`, `latitude`, and `date`, respectively. We express the relationship with the formula `~ longitude + latitude @ date`. When we give this formula to kml(), the function examines the formula to detect whether to use kmlPoints() or kmlTime(). In this case, since we have an `@` in our formula, kmlTime() will be called. To stylize the plot, we can pass to kml(), through its ... argument, the arguments and values that we used previously in the call to kmlTime(). We invoke the kml() function with

```
doc = kml(~ longitude + latitude @ date, elephantSeal,
          name = format(elephantSeal$date, "%d-%m"),
          docStyles = esStyles,
          style = c("paddle_out", "paddle_back")[direction],
          lty = c("line_out", "line_back")[direction],
          addLines = TRUE)
```

17.4.2 Grouping Placemarks into Folders on Google Earth

The RKML package's extension to the formula language adapts the `|` symbol for the Google Earth setting. We use the variables appearing to the right of `|` in the formula to partition the placemarks into folders. This way, groups of placemarks can be toggled on and off the earth. We similarly adapt the *groups* argument that appears in `lattice` functions such as xyplot(). We use *groups* to specify colors or glyphs for placemarks via styles. The value for the *groups* argument can be a vector of *KML* style identifiers that is the same length as the vector of longitudes/latitudes. Alternatively, it can be a `factor` with levels that correspond to the *KML* document styles. The latter possibility is shown in the next example.

Example 17-5 Plotting Earthquake Locations in KML via the Formula Language

We continue with Example 17-4 (page 592) and demonstrate some of the advantages to using the formula-language in making Google Earth plots. In this next display of earthquake locations, we will condition on magnitude. This allows the viewer to, e.g., interactively remove from the display the multitude of small quakes and to toggle on and off medium and larger quakes for making comparisons. With the formula: `~ longitude + latitude | magCut`, the quakes will be grouped according to their magnitude (recall `magCut` is a discretized version of `magnitude` with five levels). Each subset of placemarks are collected into a folder and displayed in the Places panel.

Furthermore, we use the *groups* argument to kml() to specify styles for the placemarks. We provide `depthCut` as the value for the *groups* argument. The `depthCut` variable is a factor with five levels for depth, and this factor is used as an index into the document-level styles. These styles are provided to kml() via its *docStyles* argument. For styles, we use the *list* `pmStyles`, where each style specifies an image to use as an icon, e.g.,

```
pmStyles[[1]]

$IconStyle
$IconStyle$Icon
          href
"yor1ball.png"
```

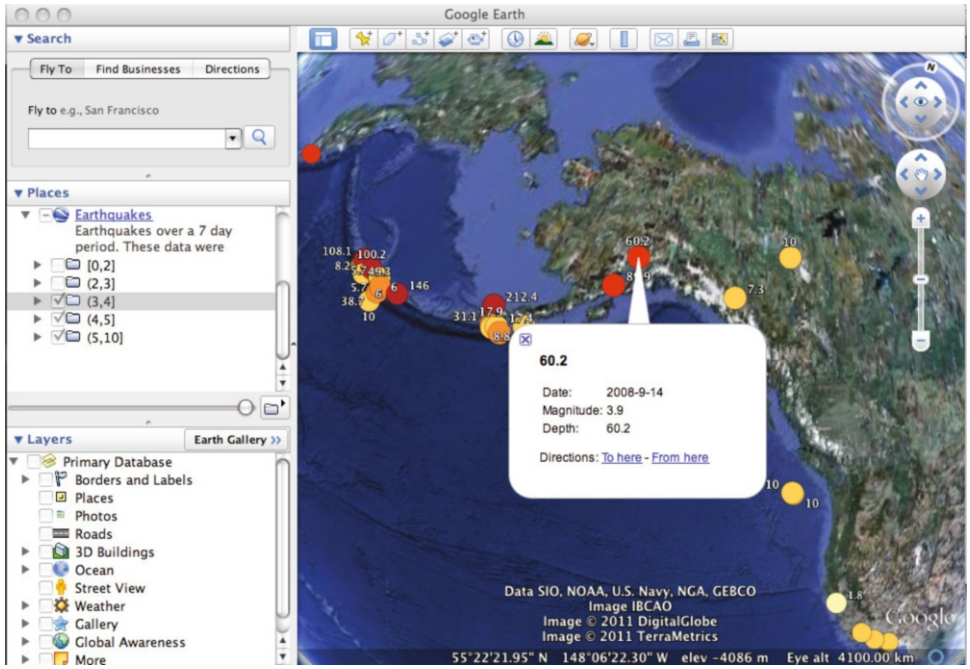


Figure 17.6: Alternative Customization of a Google Earth Earthquake Display. This particular rendition of the earthquake data was created using the formula `~ longitude + latitude | magCut`. Conditioning on `magCut` leads to the creation of the five folders within `"Earthquakes"` in the Places panel. The viewer can use these folders to choose subsets of quakes to view. We also specified a discretized version of the depth of the quake in the *groups* argument to kml(). As a result, five icons (colored circles), one for each level of depth, are used to represent the earthquakes.

The complete call to kml() is shown here:

```
doc = kml(~ longitude + latitude | magCut, quakes,
          groups = depthCut, docName = docName,
          docDescription = docDescription, docStyles = pmStyles,
          description = ptDescriptions, .names = quakes$depth)
```

The values for *docName*, *docDescription*, and *description* are the same as in Example 17-4 (page 592). That is, they supply, respectively, the title of the document that appears in the Places panel, the content of the balloon window that pops-up when the viewer clicks on the document, and the contents for each placemark's balloon window. We have changed two arguments to reflect the grouping by earthquake depth. The value for *docStyles* is the list containing the five icon styles (colored circles) and the *.names* argument supplies the values for depth; these appear as labels next to the icons. We see a screenshot of the resulting display in Figure 17.6.

17.5 The *KML* Grammar

Up to this point, we have not discussed or seen the format of a *KML* document because this knowledge was not needed to use the high-level functions kmlPoints(), kmlTime(), and kml(). However, to create a new kind of plot for Google Earth or to customize a display, then knowledge of the *KML* grammar is necessary. Several intermediate-level functions are available in RKML for building and adding elements to *KML* documents. These functions generate snippets of *KML*, and their inputs typically include a *KML* document or elements. To build your own displays and to use these intermediate functions, it will be useful to understand *KML*. In this section, we briefly introduce the structure of a *KML* document and many of the most commonly used elements in the language. A more extensive tutorial is offered at `http://code.google.com/apis/kml/documentation/kml_tut.html` and the full *KML* API [4] is available at `http://code.google.com/apis/kml/documentation/kmlreference.html`.

17.5.1 A Sample KML Document

A diagram of the general structure of a *KML* document appears in Figure 17.7. In this figure, we see that the root node of a *KML* document is <*kml*> and it has one child, <*Document*>. The document-level information includes the <*name*> and <*description*> of the document, which appear in the Places panel of Google Earth. We provide this information to the high-level functions in RKML via the *docName* and *docDescription* parameters. We also see in the diagram that the <*Style*> children of <*Document*> hold the document-level style information. Notice that the element names of the children of <*Style*>, e.g., <*IconStyle*>, <*Icon*>, and <*scale*>, match the names of the elements we used in the *R* list for providing style information to our high-level plotting functions. Also note that each placemark is represented as a <*Placemark*> element and these are grouped in this particular example into a single <*Folder*> element. An excerpt of the actual *KML* on which this diagram is based appears below, where we provide brief descriptions of the various *KML* elements via callouts.

```
<kml xmlns="http://earth.google.com/kml/2.2">  1
 <Document>  2
  <name>Elephant Seal Journey</name>  3
  <description> Elephant seal data... </description>  4
  <LookAt>  5
    <longitude>-135.5</longitude> <latitude>42.0</latitude>
```

```
    <altitude>4100000</altitude>
    <tilt>0</tilt> <heading>0</heading>
    <altitudeMode>absolute</altitudeMode>
  </LookAt>
  <Style id="paddle_out">  6
    <IconStyle>  7
      <scale>0.5</scale>
      <Icon>http://maps.../grn-blank.png</Icon>
    </IconStyle>
    <BalloonStyle>$description</BalloonStyle>  8
  </Style>
 ...
  <Style id="line_back">
    <LineStyle>  9
      <color>ff999afb</color>
      <width>2</width>
    </LineStyle>
  </Style>
  <Folder>  10
    <name>Daily Tracks</name>
    <Placemark>  11
      <name>24-02</name>  12
      <description>24-02</description>  13
      <TimeStamp>  14
        <when>1992-02-24T00:00:00</when>
      </TimeStamp>
      <styleUrl>#paddle_out</styleUrl>  15
      <Point>  16
        <coordinates>-120.000,34.000,0</coordinates>
      </Point>
    </Placemark>
...
    <Placemark>
      <styleUrl>#line_back</styleUrl>
      <TimeStamp>
        <when>1992-05-07T00:00:00</when>
      </TimeStamp>
      <LineString>  17
        <tessellate>1</tessellate>
        <extrude>0</extrude>
        <coordinates>-122.900,34.600,0
                     -120.000,34.000,0</coordinates>
      </LineString>
    </Placemark>
  </Folder>
 </Document>
</kml>
```

1 The root of the *KML* document is <kml>.
2 <Document> contains all of the styles, folders, and placemarks for the display. There may be only one <Document> child in <kml>.
3 The <name> node specifies a title that will be displayed in the Places panel.
4 The content of this <description> element will appear in a pop-up window when we click on the blue-and-white document icon in the Places panel.
5 This <LookAt> node specifies the initial view of the document, i.e., when we load the document into Google Earth, the globe spins and zooms to this location.
6 Each <Style> element can be used to specify the appearance of a placemark, line segment, ground overlay, etc.
7 This <IconStyle> child of <Style> holds various pieces of style information for a placemark icon. It includes the image to use for the icon (in <Icon>) and a scale factor for the image (in <scale>).
8 Style information for a pop-up window is provided in <BalloonStyle>. In this case, text substitution is used in the content to create a description that is customized to the placemark, e.g., the contents of a placemark's <description> will be substituted for $description.
9 The <LineStyle> element in the <Style> element called line_back contains a color and width that can be used with a line segment (see the <LineString> element below).
10 The <Placemark> nodes are organized into one or multiple <Folder>s. The <Placemark>s in a <Folder> can be added to and removed from the earth by selecting the folder icon in the Places panel.
11 The <Placemark> element identifies a location on earth. It can contain geometric elements such as <Point>, <LineString>, and <Polygon>. These elements will be drawn on the earth at the locations specified in the <coordinates> node.
12 The text content of this <name> node will appear as a label next to the location of the placemark. This is different from the <name> child of <Document> and the <name> child of <Folder>. These other elements supply the titles for the document and folder, respectively.
13 The content of <description> will appear in a pop-up window when the viewer clicks on the placemark.
14 The <TimeStamp> element contains information about the time when the placemark is visible (this information is in <when>). If a <TimeStamp> element appears in a document, then Google Earth provides a time slider so the user can control the interval of time being viewed.
15 The reference to the paddle_out style in <styleUrl> indicates the particular style that should be applied to this placemark. This style information appears in the <Style> element that has an *id* of "paddle_out".
16 The <Point> node provides the latitude, longitude, and elevation of a point in its <coordinates> child. These three values are supplied as a comma-separated list. Google Earth adds an icon to the globe at this location.
17 <LineString> provides the latitude, longitude, and elevation for the start and end points of a line segment. These coordinates are supplied in the <coordinates> child; they are separated by white space. Other specifications in <LineString> include whether the line follows the terrain (<tessellate>) and whether it is connected to the ground (<extrude>).

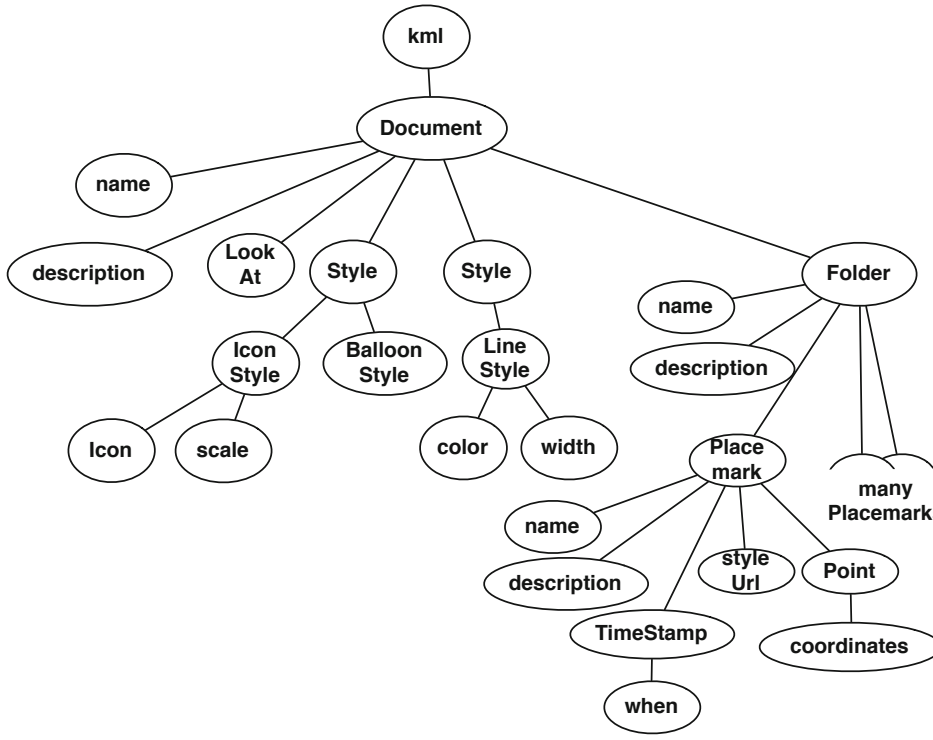


Figure 17.7: Example *KML* Document Tree. This diagram illustrates the structure of a typical *KML* document. This document was produced by the call to kmlTime() in Example 17-3 (page 590). No text content is included in the diagram to keep the display simple.

17.5.2 Strategies for Working with and Debugging KML Documents

We provide in this section a few ideas that we have found helpful when developing a *KML* display. First, when we see a feature in a Google Earth display that we may want to replicate, we can right-click on it in the Places panel, copy it, and paste it into a text editor to view the *KML* source for that feature. This can be especially useful when we want to see how a feature is constructed. Also, when debugging, we have found it helpful to edit the *KML* document in a text editor and, say, change the color or size of a feature so that it can be easily identified on Google Earth to confirm the placemark is in the correct location, etc. Or, we may add style information directly to a `<Placemark>`, rather than using the indirect approach, to help in debugging our *KML*.

When we are building a *KML* document or designing a new type of plot, the contents of the document may be changing frequently. We would ordinarily have to reload the revised document explicitly in Google Earth to view the update. We can facilitate this and have Google Earth regularly check to

see if it should reload the file because it has changed. The createUpdatingDoc() function is designed to create a document with the sole purpose of reloading another *KML* file. We can use it as follows:

```
dyn = createUpdatingDoc("eqFormula.kml")
saveXML(dyn, "dyn.kml")
```

This creates a new document `dyn.kml`, and that reloads `eqFormula.kml` at regular intervals. This means that we can load `dyn.kml` into Google Earth and then continue to update and save `eqFormula.kml`. Any updates will be loaded into Google Earth automatically.

17.6 Working More Directly with *KML* to Create Custom Displays

With kml(), kmlPoints(), and kmlTime() we can quickly and easily create alternative presentations of spatial data. However, these types of displays are just the beginning of what we can create with *KML*, and we imagine the programmer wanting to make use of other features in Google Earth to design an alternative data display. For example, Google Earth has the notion of a ground overlay, which is an image draped across the surface of the earth. We may want to create a plot (or plots) in *R* and use it as a ground overlay. To do this, we can create the *R* plot, save it as a PNG image, and post-process an existing *KML* document to include this image file as an overlay. As another example, we may want to draw a plot directly on the virtual earth. The *KML* language has the notion of a geometric object, including point, line segment, and polygon, and we can draw these objects at different locations on the virtual earth. With these basic primitives we can draw a plot directly on Google Earth. The RKMLDevice package [20] implements a *KML* graphics device (using pure *R* code) for this purpose. With it, we can create arbitrary *R* plots in *KML* and they can be drawn on Google Earth.

To use an *R* plot as a ground overlay or to draw an *R* plot directly on the virtual earth requires more control of the document than is offered by the high-level functions kml(), kmlPoints(), and kmlTime(). These functions create the entire document in a single call. In this case, we want the ability to work at a lower level to build a document from scratch or to post-process a document produced by one of the high-level functions. The RKML package provides intermediate-level functions to help with this in many cases. These functions enable us to build a wide range of *KML* elements and incrementally add them to a document. For example, with createKMLDoc() we get a skeleton document with no placemarks or other features. Then we can add to this document styles, folders, placemarks, ground overlays, etc. using functions in RKML such as addStyles(), addFolder(), addPlacemark(), addGroundOverlay(), respectively. These helper functions generate a snippet of *KML* and add it to a *KML* node that the caller has specified.

In this section, we will demonstrate how to create a *KML* document from scratch and how to augment an existing document. We will also show how to make ground overlays and use the *KML* graphics device. In this process, we demonstrate how to use the intermediate functions in RKML and the basic workflow for creating and modifying *KML* documents within *R*.

17.6.1 Overlaying Images Made in R on Google Earth

In this section we demonstrate how to create a *KML* document that has ground overlays made from static image files of *R* plots. The basic workflow is to

- Make the *R* plots that will be overlaid on the virtual earth and save them as, say, PNG files.
- Create a skeleton *KML* document that will contain the ground overlays.
- Add a <Folder> to the document to hold the ground overlays.
- Add a <GroundOverlay> element for each image file to this folder. Each element includes specification of the image's file name and the coordinates of the region to which we "pin" the overlay.
- Add any other features to the document. These may include a legend, an additional folder, placemarks in this new folder, and document-level styles for the placemarks to reference.

With knowledge of *KML*, we can use the functions in XML, such as newXMLDoc() and newXMLNode(), to create the document (see Chapter 6 for more details on these and other functions for generating *XML*). For example, we can create the "stub" of a *KML* document using newXMLDoc(). Then we continue with calls to newXMLNode() to add the root <kml> to the document, <Document> to <kml>, and <name> and <description> to <Document>, etc., i.e.,

```
kml.doc = newXMLDoc()
kml = newXMLNode("kml", doc = kml.doc)
doc = newXMLNode("Document", parent = kml)
newXMLNode("name", docName, parent = doc)
newXMLNode("description", docDescription, parent = doc)
```

However, this code has been generalized and wrapped into the intermediate-level function createKMLDoc(). It also includes the capability of specifying the <LookAt> and other document-level information.

In general, the intermediate-level functions in RKML provide wrappers to newXMLNode() to add elements to a *KML* document. For example, addStyles(), addFolder(), addPlacemark(), and addGroundOverlay() functions add, respectively, <Style>, <Folder>, <Placemark>, and <GroundOverlay> elements to a *KML* document. The kmlPoints() and kmlTime() functions use many of these intermediate-level functions to create a *KML* document.

These intermediate-level functions all work in a similar fashion. The newly generated *XML* elements are added to the document at the location specified in the *parent* parameter of the function. In the cases where we are unambiguously adding an *XML* element to the top-level <Document> node, e.g., adding a style definition, we can pass the *KML* document object as the parent rather than explicitly finding the <Document> subnode. Children of the element can typically be specified via parameters to the function call. For example, to include a <description> child in a <Placemark> node, we provide the content of the <description> element as a character string to the *description* parameter of addPlacemark(). These intermediate-level functions take care of assembling the element and its children. The following is an example of how to call addPlacemark():

```
addPlacemark(coordinates, name = placemarkLabel,
             description = placemarkBalloonContent,
             styleUrl = styleID, parent = folderNode)
```

As a result of this function call, a new <Placemark> element will be added to the <Folder> element in `folderNode`. This <Placemark> will contain <name>, <description>, and <styleUrl> children. The contents of these children are the values of the *name*, *description*, and *styleUrl* arguments. The goal of these functions is to hide the details of the *XML* from the user.

In the next example, we demonstrate how to use these intermediate functions to create a custom *KML* document.

Example 17-6 R Plots of Average Daily Temperature as Ground Overlays on Google Earth

Here we examine weather patterns in 100 cities in the United States, where each city has its individual weather time series. We drape across the earth, at each city's location, the PNG image of the time series. These simple line graphs have been created in *R* using the plot() function. Such a *KML* document cannot be generated by kmlPoints(). Instead, we demonstrate how to create the *KML* from "scratch" using the intermediate-level functions in RKML. Our *KML* document has several components:

- document details, e.g., its name and description;
- style information for the placemarks;
- placemark at each city's location to give access to a balloon window for the city;
- balloon window for each city that includes an image generated by a standard *R* graphics function;
- ground overlay at each city's location.

See Figure 17.8 for a screenshot of the Google Earth display.

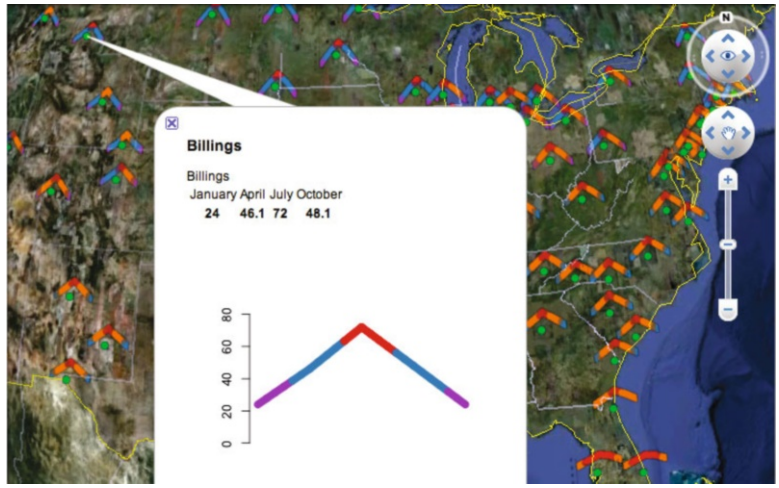


Figure 17.8: *R* Plots as Ground Overlays on Google Earth. A set of 100 PNG line graphs of daily temperature in 100 US cities are made in *R* and included in this *KML* archive as ground overlays. These ground overlays are placed on Google Earth at each city's location. The images have transparent backgrounds so as to not obscure one another nor the terrain. Additionally, the *KML* document contains a small green circular placemark for each city. This enables the user to interactively query a city for more detailed data. In this screenshot, the user has queried Billings, MT. The description window contains another line graph made in *R*. The archive contains 100 of these PNG plots too.

We demonstrate how to create each of these components of the *KML* document with the help of the intermediate functions. The data for making the display are available in `temperature` in the RKML package. Suppose that we have already used `temperature` to construct the images for the ground overlays and the images for the balloon windows. Also suppose, each city's image files have names such as `BillingsMontOverlay.png` and `BillingsMontBalloon.png`, respectively, for, e.g., Billings, MT.

We begin by creating the basic *KML* template with its `<kml>` root and `<Document>` child node. We use the createKMLDoc() function for this purpose. We provide this function the name and description for the document. In addition, we use the *window* argument to specify the initial latitude and

longitude, and optionally elevation, for viewing the document when it is first opened in Google Earth. We create this document with

```
doc = createKMLDoc(docName = "City Temperatures",
                   description = 'Temperatures for 100 cities ...',
                   window = c(latitude = 38.5, longitude = -96))
```

Next we use addStyles() to add <*Style*> elements to the document. Since we want the placemarks to appear as small green circles, we create the necessary style information with

```
gF = system.file("Icons/greendot.png", package="RKML")
style = list(IconStyle = list(scale = .25, Icon = c(href = gF)),
             LabelStyle = list(scale = 1, color = "#FFFFFF00"))
```

The styles are specified in a *list* as described in Section 17.2.3. We want the style element to be available to all placemarks in the document so it needs to be a child of <*Document*>. We access this parent element with

```
docNode = xmlRoot(doc)[["Document"]]
```

Then we pass this parent element and the style information to addStyles() as follows:

```
addStyles(docNode, list(citySymbol = style))
```

The addStyles() function converts the *list* of style information into the appropriate *KML* nodes and adds this <*Style*> node to its intended parent, which is in `docNode`. Later, our placemarks can refer to this style as `"citySymbol"`.

Next we construct a placemark for each city. We demonstrate how to do this for Billings, Montana. We first create a folder to hold Billings' placemark and its ground overlay. The folder's parent will again be the <*Document*> element. We use addFolder() to create the folder element with the name `"Billings"` as follows:

```
folder = addFolder("Billings", parent = docNode)
```

We add a <*Placemark*> node along with its coordinates, label, and style identifier,

```
mark = addPlacemark(c(-108.53, 45.80), name = "Billings",
                    styleUrl = "#citySymbol", parent = folder)
```

We can use the *description* argument to add a description for the placemark, but instead we separately construct the <*description*> element and then add it to the <*Placemark*>.

Suppose we have constructed the content of the description from `temperature` and formatted it as *HTML* content, e.g.,

```
<p>Billings</p>
<table>
<tr><td>January</td><td>April</td>...</tr>
<tr><th>24</th><th>46.1</th>...</tr>
</table>
<img src="BillingsMontBalloon.png"/>
```

We add this *HTML* description to the placemark (which is in `desc`) with

```
addDescription(desc, parent = mark)
```

Our final step is to add the ground overlay to the Billings folder. To do this, we specify a rectangular region on the surface of the earth that the *png* file will cover. This region is slightly offset from the

location of Billings' placemark so that it is easier to click on the small green dot we have already added. The addGroundOverlay() function creates this node for us as follows:

```
off = c(1, 1.5)
loc = c(-108.53, 45.80)
box = c(north = loc[2] + off[2], south = loc[2] - off[2],
        east = loc[1] + off[1], west = loc[1] - off[1])
addGroundOverlay(name = "Billings",
                 Icon = c(color = "#88FFFFFF",
                          href = "BillingsMontOverlay.png"),
                 LatLonBox = box, parent = folder)
```

We can abstract this code into a function so that we can easily create folders containing a placemark and ground overlay for each city.

When we save the document, we will get a **zip**ed file that contains the *KML* document, the 100 ground overlays, the 100 images for the balloon windows, and the green dot icon.

17.6.2 KML-Formatted Plots on Google Earth

As mentioned earlier in this section, *KML* has the notions of a geometric object and the placement of these objects on a canvas. We can use these geometric primitives to draw plots on Google Earth, and we would like to use *R* to do the drawing. We can achieve this with RKMLDevice [20]. This package implements a *KML* graphics device (using purely *R* code). With it, we can create arbitrary *R* plots as *KML* documents and they can be displayed in Google Earth. As the *R* graphics engines emits points, adds axes tick marks and labels, generates polygons for maps, etc., it makes calls to the graphics device's primitive operations for drawing circles, lines, text, and polygons. The RKMLDevice package merely generates the corresponding *KML* and adds it to a node in a local *KML* document associated with that device. When the plotting is complete and the device closed in *R*, that document has the *KML* to display the plot(s).

There are several advantages to drawing on the virtual earth with *KML* over using image files as overlays. One benefit is that the image is vectorized so it scales well when zooming in and out. Another is that Google Earth determines the resolutions at which the image should be shown, hiding it when it would be too small. Also, the plot elements (e.g., legend, tick mark labels, etc.) are separate elements in the *KML* display and so can be hidden or displayed at any time by the viewer (via the Places panel), thus allowing more detailed control of the information.

The simple idea behind the device is that the *R* user opens a device as a rectangular region in the coordinate system of Google Earth, i.e., in latitude and longitude, to define the boundaries in which the *R* plot should be drawn. The dimensions are similar to the number of pixels or inches in other devices for the horizontal and vertical extent. *R* uses the dimensions of the box to define the effective drawing region of the device. It then translates the coordinate system of the data to the graphics device and the *R* plot is drawn within that region. In essence, the *KML* graphics device enables us to plot right on a region of the earth. In the following example, we explore how to add boxplots drawn in *KML* on Google Earth.

Example 17-7 Creating Temperature Boxplots with a KML Graphics Device

Here we make three separate plots in *R* showing monthly boxplots of daily temperatures for each of three cities: Seattle, Portland, and San Francisco. These are not PNG files, but *KML* documents that are displayed on Google Earth at each city's location. Each plot has twelve boxplots showing the distribution of temperature for the days in a month. See Figure 17.9 for a screenshot.

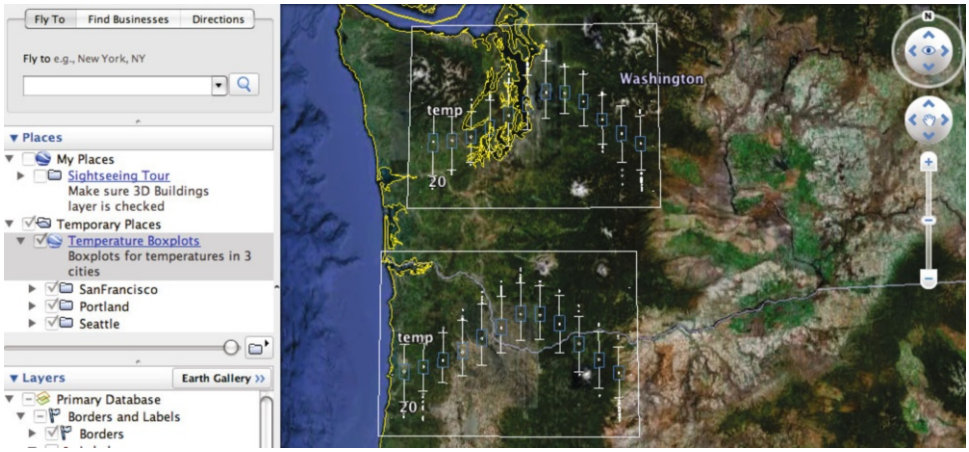


Figure 17.9: Boxplots of Daily Temperatures Drawn in *KML* on Google Earth. The boxplots displayed in this screenshot are created in *R* using bwplot() and the *KML* device in RKMLDevice. These plots are *KML* drawings (not PNG files). They can be hidden or shown by clicking on the check box next to the corresponding city's folder in the "Temperature Boxplots" document in the Places panel.

The basic approach to plotting with the *KML* device goes as follows:

- Open the *KML* device by calling kmlDevice(), and specify the longitude–latitude of the bottom left corner and the width and height of the rectangular region that the plot will occupy. Assign the return value to a variable, say `doc`;
- Make a plot and close the device;
- Save the *KML* tree in `doc` with a call to saveXML().

This sequence of steps will create a new *KML* document containing the plot.

We can also add a plot to an existing *KML* document. We do this by passing in an existing *KML* document to kmlDevice() when we open the device. Then, the *KML* instructions for drawing the plot will be added to this document. This is the approach that we take here so that all three sets of boxplots are in the same *KML* document, which will make it easier to load the plots in Google Earth. We save the updated *KML* document after we have made all three plots.

We begin by creating a new and separate *KML* document. This is the document to which we will add the plots. We use createKMLDoc() to do this with

```
doc = createKMLDoc(docName = "Temperature Boxplots",
                   description = "Boxplots for temperatures...",
                   window = c(longitude = -122, latitude = 42,
                              altitude = 2000000))
```

We then access the <*Document*> node of this newly created *KML* file with

```
docNode = xmlRoot(doc)[["Document"]]
```

Now we are ready to initiate a graphics device for the first plot.

When we call kmlDevice(), we pass it `docNode` so that the plot will be added to this node. We start with San Francisco and open the *KML* device with

```
kmlDevice(SFLoc, 3, 1.5, folder = "San Francisco", doc = docNode)
```

Here `SFLoc` contains the longitude and latitude of the lower left corner of the region that will hold the plot for San Francisco: 3 is the width in degrees and 1.5 is the height in degrees of the rectangular region. The device will create a new folder in `docNode` to hold the plot. In this call, we have also passed a name for the new folder.

The next step is to make the plot for San Francisco and close the device. We use bwplot() in `lattice` to do this with

```
print(bwplot(temp ~ months, SFTemperatures))
dev.off()
```

We repeat this process for the other two cities, each time specifying `docNode` as the parent node of the plot when we open the device. Also, the location of the region and the folder name will change to reflect the change in city.

After we make all three plots, we save the updated document with

```
saveXML(doc, "boxplotsTemps.kml")
```

An alternative approach is to create three different *KML* documents, one for each plot. Then we can combine these into one document by extracting the `<Folder>` element from each document and adding them to a new document. These operations are reasonably straightforward using the general tools in the XML package [23] to manipulate and search *XML* trees.

17.7 Embedding Google Earth in a Web Page

Google Earth is most often used as a stand-alone application. However, Google also offers a plug-in for embedding the earth browser into a Web page. This allows for more interesting and complex visualizations as data displayed on the earth can be viewed in the context of other plots and information and even linked with forms and plots in the Web page.[2]

While there are different ways to create interactivity and links between a standard *R* graphical display and Google Earth, e.g., with image maps, in this section, we create plots as *SVG* documents using the *SVG* graphics device in *R* and then enhance them to add interactivity using the SVGAnnotation package [15] (see Chapter 16). With these technologies (Google Earth, *SVG* plots, and *HTML* forms), we can create interesting mash-ups. The glue underlying these technologies that enables the interactivity is *JavaScript*. *JavaScript* event handlers can, for example, respond to a viewer's mouse activity on a plot element or an *HTML* button and update the earth view or add and remove placemarks from Google Earth. The action can go in the other direction as well, e.g., a mouse click within the Google Earth plug-in can result in a change in a plot in the *HTML* page.

[2] Currently the Google Earth plug-in is supported on Chrome, Firefox, Safari, and some versions of Internet Explorer Web browsers. Opera is not supported.

In this section, we demonstrate some of these capabilities. We first show how to use Google Earth as a plug-in and then proceed to create a Web page with more complex interactions between the Google Earth plug-in, *HTML* buttons, and *SVG* plots. The *KML* documents displayed in the plug-in and the *SVG* plots embedded in the page are all created in *R* using RKML and SVGAnnotation.

17.7.1 Using the Google Earth Plug-in

In order to use the Google Earth code, you must register with Google for an API key and use that key when referencing the Google code. This is required and free. The sequence of steps to use the plug-in in a Web page are as follows:

- load the *HTML* document,
- load the Google Earth plug-in,
- create the view or the Google Earth instance,
- load one or more *KML* documents,
- insert the *KML* document(s) into the view.

The Web browser handles the first step; we supply the *JavaScript* to handle the rest. Actually, our *JavaScript* calls methods in the Google Earth API to carry out these tasks.

As an example, we consider housing data made available by the *San Francisco Chronicle*. These data contain information about sales in the San Francisco Bay Area, including the date of sale, sale price, square footage, and location of each house sold from April 2003 to May 2006. These data were scraped from the Web and analyzed in [26]. Google Earth is a powerful way to explore both the individual observations and the general aggregates. Viewers can also layer, for example, crime reports, public school performance measures, or census data on income levels for the different neighborhoods, and explore house prices as these variables increase and decrease across neighborhoods.

Example 17-8 Displaying San Francisco Housing Market Data in a Google Earth Plug-in

This example takes the first step toward creating a Web mash-up of the San Francisco housing data. Here, we simply embed a Google Earth display in a Web page and add a placemark for each of the cities in the Bay Area. For each city, we also display a time-series plot of the weekly median housing sales in its balloon window. The *KML* file for this display is the same whether it is shown on the Google Earth browser or on Google Earth embedded in a Web page. To make this Google Earth document, we use an approach similar to Section 17.2.1. Our focus in this example is on the *HTML* and *JavaScript* needed to view this *KML* file on Google Earth embedded in the browser.

The *HTML* document shown below is the source file for the simple page shown in Figure 17.10. We have annotated it to highlight the basic pieces needed to use a Google Earth as a plug-in.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>  1
 <title>Google Earth in Browser</title>
 <script src="http://www.google.com/jsapi?key=ABQ...">
 </script>  2
 <script  rel="text/javascript" src="geCity.js"></script>  3
</head>
<body onload='init();'  id='body'>  4
 <h1>Bay Areas Housing Prices</h1>
 <p> Catalog of houses sold ... </p>
```

```
<div id='myGE' class="ge"  5
     style='border: 3px solid silver; float: left;
            height: 600px; width: 600px;'>
</div> </body> </html>
```

1 The <head> element contains the necessary *JavaScript* code in two <script> elements. Each <script> element must have a separate closing tag because most browsers are currently unable to properly parse a self-closing <script> tag.

2 This first <script> node references code located on the Google site: http://code.google.com/. This code is needed for the plug-in to be created and programmatically manipulated via our *JavaScript* functions. This is essentially a *C* library for Google Earth.

3 This second <script> element references our *JavaScript* code in geCity.js. It includes functions that will be invoked asynchronously by the Web browser as, e.g., the page is loaded, the plug-in is loaded, and the viewer interacts with the plug-in, etc. This code is shown below.

4 The *onload* attribute of the <body> element is a call to our *JavaScript* *init()* function that initializes Google Earth. In other words, once the *HTML* page is loaded, the Web browser calls this function which then creates the Google Earth plug-in viewer in the *HTML* via *JavaScript*.

5 The element labeled "myGE" is the place where the Google Earth viewer is displayed. The appearance of this can be controlled via a cascading style sheet. Instead, we have simply added the style information directly to the tag via its *style* attribute, which includes the width and height of the space that the browser will allocate to display the plug-in in the page.

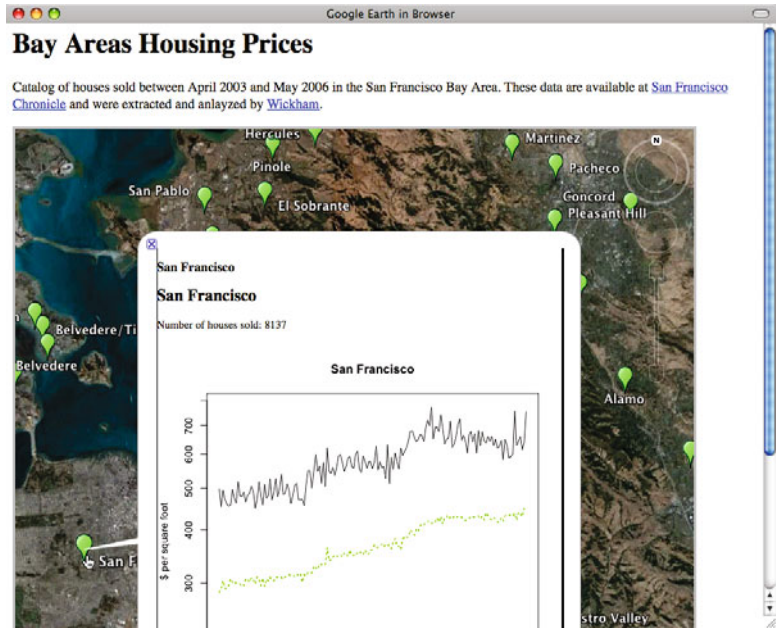


Figure 17.10: Median House Prices Displayed in a Google Earth Plug-in. This screenshot shows Google Earth embedded in a simple Web page. The viewer can interact with the virtual earth in the same way as in the stand-alone browser, e.g., zooming, rotating, and clicking on a placemark to view its details. However, the Places panel is currently not available for adding and removing folders.

Our *JavaScript* code in `geCity.js` contains the code necessary to load the plug-in and add a *KML* document to the embedded Google Earth. This script (shown below) starts with a call to *google.load()* as the *HTML* document is being processed. This function does not create the view; it just loads the plug-in. Once the page has been loaded (the plug-in will be loaded too), the *init()* function is called. The *init()* function calls *createInstance()* to create a view or Google Earth instance. We pass *createInstance()* the location in the *HTML* document to place the Google Earth viewer and two functions: one to call when the Google Earth instance is successfully created, and another to call if the instance is not successfully created. These functions are *initCallback()* and *failureCallback()*, respectively. The *initCallback()* function calls methods in the Google Earth API to make the view visible in the *HTML* window, add navigation controls, change the view to a particular coordinate set and most importantly to start downloading our *KML* document. Our *KML* document is added to the view after it has been downloaded (the *fetchCallback()* function does this). The code is shown below.

***JavaScript* for Plugging Google Earth into a Browser**

- We load the plug-in into the browser with

```
google.load("earth", "1");
var ge = null;
```

  We have also set up a global variable to hold our Google Earth instance.
- The *init()* function is called when the page is loaded because `onload='init();'` appears on the <*head*> element. This function creates an instance of the Google Earth viewer.

```
function init() {
  google.earth.createInstance("myGE", initCallback,
                              failureCallback);
}
```

  If successful, the *initCallback()* function is called with the viewer object, and if unsuccessful, we call *failureCallback()*.
- The *initCallback()* function places the Google Earth viewer in our global variable `ge` so that our handler functions have access to it. We also call the viewer's methods to make it visible in the page, orient the view, and load the **kmz** file.

```
function initCallback(object) {
  ge = object;
  ge.getWindow().setVisibility(true);
  ge.getNavigationControl().setVisibility(ge.VISIBILITY_AUTO);

  var lookAt =
    ge.getView().copyAsLookAt(ge.ALTITUDE_ABSOLUTE);
  lookAt.setLongitude(-122.27);
  lookAt.setLatitude(37.87);
  lookAt.setAltitude(4000.0);
  ge.getView().setAbstractView(lookAt);

  google.earth.fetchKml(ge, 'http://.../cityHousePrices.kmz',
                        fetchCallback);
}
```

- If there is a problem in creating the Google Earth viewer, then *failureCallback()* is called. Here we simply notify the user of the problem with a message in a pop-up window.

```
function failureCallback(object) {
   alert("Failed to start Google Earth");
}
```

- The **kmz** file is loaded when *initCallback()* calls the *fetchKml* method. In addition to providing *fetchKml* with the name of the file to be loaded, we also provide the *fetchCallback()* function. This function is called when the *KML* document has finished loading. It is responsible for adding the parsed document to the Google Earth plug-in's tree hierarchy.

```
function fetchCallback(obj) {
   alert("Should be loading " + obj);
   ge.getFeatures().appendChild(obj);
}
```

  We also open a pop-up window letting the user know that the *KML* document is loading.

This sequence of asynchronous *JavaScript* calls can be a little confusing and indirect for those more familiar with procedural, sequential programming. This approach allows the Web browser and other *JavaScript* code in our page to continue and respond to user interaction while we wait for different downloads to complete.

Note that it is possible to create multiple Google Earth displays in one Web page. We simply add to the *HTML* a `<div>` node for each Google Earth object. For each instance, we make a separate call to *google.earth.createInstance()* to create it and we call *google.earth.fetchKml()* to load one or more *KML* files into each instance.

17.7.2 Linking the Plug-in to Other Elements in a Web Page

We can go a great deal further than just adding *KML* documents to the Google Earth plug-in. We can combine regular statistical plots with Google Earth and link them to make rich interactive displays. For example, imagine an interface for exploring the housing data used in Section 17.7.1 that enables us to see the relationship between median price and size for houses in the various cities in the Bay Area and to link this information to placemarks on Google Earth so that the regions of high- and low-priced cities emerges. We can create a scatter plot of city median house price and size where a click on a city's point in the scatter plot leads the Google Earth viewer to spin so the view is centered on that city. At the same time, when we click on the city's point in the scatter plot, a second scatter plot of the price and size for each house sold in that city is revealed. Additionally, we can design this second scatter plot so that each point is linked to its corresponding placemark in Google Earth. This way, when we click on an observation in that scatter plot, a brightly colored paddle is added to the Google Earth display so it is easy to locate the placemark and look up more information about that house. With these interconnected displays, we can, e.g., explore anomalies in more depth. The example below develops a Web interface with this functionality.

Example 17-9 Linking SVG Scatter Plots and a Google Earth Plug-in

The *HTML* page shown in Figure 17.11 includes two scatter plots and a Google Earth plug-in. As described above, there are many ways the user can interact with the plots and plug-in. The *KML* to create the Google Earth display is similar to the examples shown in Section 17.2.1. We describe briefly how the interactivity between the various components in the Web page is achieved.

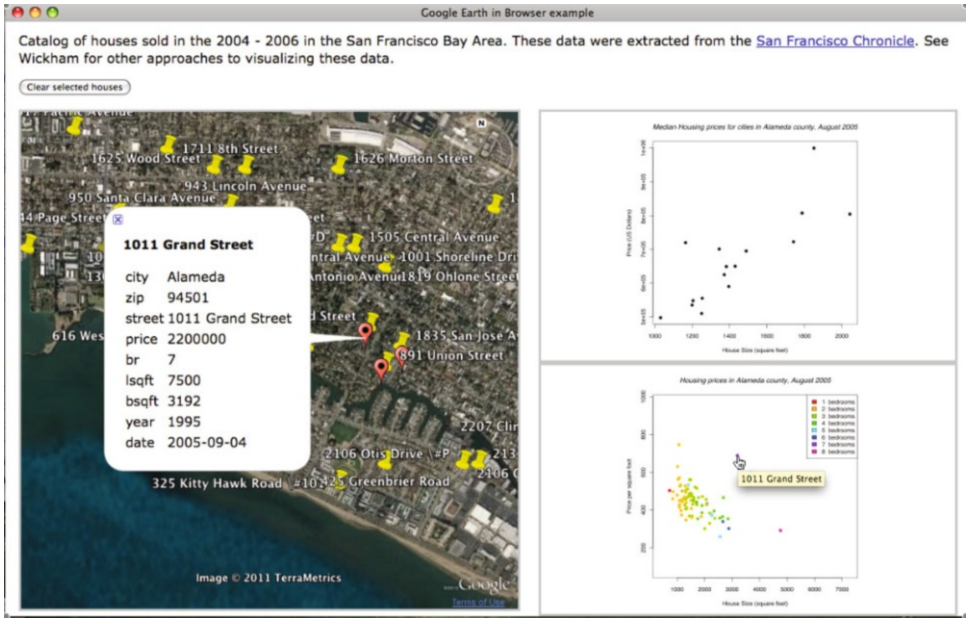


Figure 17.11: Web Page of San Francisco Housing Prices, Linking Google Earth and *SVG* Scatter Plots. This screenshot displays an *HTML* page with Google Earth embedded in it. To the right of Google Earth are two scatter plots, which are Scalable Vector Graphics (*SVG*) documents embedded in the *HTML* page. The *SVG* documents, Google Earth plug-in, and *JavaScript* in the *HTML* page enable the viewer to dynamically interact in several ways with these displays of housing sales. As shown heret, when the mouse hovers over a point, the address appears in a tool tip, and when we click on this point, a paddle is added to Google Earth to make it easier to see it. Additionally, when we click on this paddle, further information about the house pops up. A click on the button above the plug-in will remove all of these additional paddles from the display so only the yellow pushpins remain.

The *HTML* document for this display is similar to that in Example 17-8 (page 610). The only difference is that it lays out four objects: the virtual browser, two *SVG* documents, and an *HTML* button. We again use `<div>` tags to place the Google Earth display in the page.

The `<head>` of the document holds the *JavaScript* needed to set up Google Earth as well as the *JavaScript* responsible for the interactivity. The various forms of interactivity are created via *SVG*, using the SVGAnnotation package, *JavaScript*, and *HTML* forms. The model behind the interactivity is the same as described in Chapter 16. Briefly, we do the following.

- Create scatter plots as *SVG* documents using svgPlot() and *R*'s plot() function.
- Add tool tips to each point in the *SVG* documents (city name for city summaries and street address for the housing data) via addToolTips().

- Add to each point in the city-level plot an *onclick* attribute to call *selectCity()*, a *JavaScript* function. We pass this function the longitude, latitude, city name, and number of houses in the city. It will re-orient the Google Earth display to place the city in the center of its view, and hide all of the points in the current house-level scatter plot except those for the houses sold in that city.
- Add to each point in the house-level plot an *onclick* attribute, which calls the *JavaScript* function *addPlacemark()*. This function takes as input the longitude and latitude of the house, and it adds a red paddle to Google Earth at this location.

As in Example 17-8 (page 610), the *JavaScript* code in the *HTML* document creates an instance of Google Earth and loads the *KML* files. In addition, this code includes the definitions of the *JavaScript* functions that respond to mouse events on the *SVG* plots. For example, *selectCity()* resets the view port of the Google Earth object and calls the *JavaScript* function *updateVisibility()* to update the house-level plot. The *selectCity()* function is implemented as

```
function selectCity(long, lat, city, houseCt) {
  var lookAt = ge.getView().copyAsLookAt(ge.ALTITUDE_ABSOLUTE);
  lookAt.setLongitude(long);
  lookAt.setLatitude(lat);
  lookAt.setAltitude(1000);
  ge.getView().setAbstractView(lookAt);
  updateVisibility(city, houseCt);
}
```

The *updateVisibility()* function hides all points in the second scatter plot for those houses in the previously displayed city. It also makes visible the points for the houses in the newly selected city. We use a naming scheme for the points that makes it easy to identify the points corresponding to each house in a city. We implement the *JavaScript* code to hide and make visible these points as

```
function updateVisibility(city, houseCt) {
     /* Hide the previously displayed houses */
  for(i = 0 ; i < oldHouseCt; i++) {
    el = houseDoc.getElementById(oldCity + "-" + (i+1));
    el.setAttribute('visibility', 'hidden');
  }
     /* Display the new city's houses */
  for(i = 0 ; i < houseCt; i++) {
    el = houseDoc.getElementById(city + "-" + (i+1));
    el.setAttribute('visibility', 'visible');
  }
      /* Save name of current city in order to hide
         its houses when a new city is selected */
  oldCity = city;
  oldHouseCt = houseCt;
}
```

17.8 Possible Enhancements and Extensions

Including Plots in the Formula Language
Although not yet implemented, our formula interface has been developed to allow a plot specification on the left-hand side of the formula, e.g.,

```
kml(xyplot(price ~ bsqft) ~ long + lat | city)
```

The idea is that we can create separate plots based on the subsets induced by the conditioning variable (`city` in this example). This is equivalent to creating separate panels via

```
xyplot(price ~ bsqft | city)
```

and then placing these at the "center" longitude and latitude for each subgroup of locations. We can create the individual subplots/panels using either the *KML* device or an image as a ground overlay.

Google Sky
While the primary focus of Google Earth and Google Maps is on earth-related data, Google Earth is also capable of displaying astronomical data. There is increasing interest in visualizing cosmological data and with Google Earth and Google Sky these data can be displayed in context.

Making *R* Available from Google Earth
We have discussed that Google Earth and Google Maps can be used as a "live" graphics device for interactive exploratory data analysis within an *R* session. As we make the functions richer and easier to use, these facilities can be used for extensive exploratory analysis to discover characteristics of data. An improvement is to make the devices interactive. While these devices are already interactive in the sense of allowing viewers to change the resolution and toggle the visibility of elements, we mean here the ability for *R* to respond to viewer actions. Currently, when content is displayed on Google Earth or Google Maps, there is no subsequent communication with the *R* session. Ideally, we would be able to program responses from *R* to viewer interactions. For example, as we zoom in on a Google Earth view, we may want to change the elements displayed on the map to provide higher resolution information, or we may want to respond to user clicks on placemarks by updating plots within the Web page. The mechanism to achieve this interactive response from *R* is essentially available with RBrowserPlugin [2] (or `qtbase` [10]) by embedding a Web browser within *R*, or vice versa, i.e., *R* as a plug-in to Firefox. This then allows us to display one or more Google Earth plug-ins, respond to viewer interaction/events with either *R* or *JavaScript* functions, and update plots and the contents of the Google Earth plug-in. Alternatively, we can also achieve this sort of interactivity with *DCOM* and RDCOMEvents [19].

17.9 Summary of Functions in RKML

The RKML package provides both high-level functions for making "standard" plots on Google Earth and Google Maps and intermediate-level functions for working with *KML* nodes to create customized plots. Below are descriptions of the high-level functions.

kml() Create a *KML* document from the formula and/or data frame provided. The formula `~ Longitude + Latitude` also accepts conditioning variables for organizing placemarks into folders and a time stamp variable, e.g., `~ Lon + Lat @ Time | condition`.

kmlPoints() Display the locations (longitude and latitude) given in the data frame as placemarks in a *KML* document. Additional information for creating the *KML* document can be provided.

This includes: a document title (*docName*), description (*docDescription*), and style definitions (*docStyles*); a folder name (*folderName*); and placemark styles (*style*), labels (*.names*), and descriptions (*description*).

kmlTime() Display the locations (longitude and latitude) given in the data frame as placemarks in a *KML* document and associate a time stamp (*time*) with each location. Additional descriptors and styles are available as with kmlPoints().

kmlLegend() Place a legend (*legend*) on the Google Earth viewing window at the position specified in *x*, *y* and with box colors (provided in *fill*) and labels (in *legend*).

If the user has a basic understanding of the format of a *KML* document then it is possible to create custom displays of data. Next we provide descriptions of functions in RKML for creating and modifying nodes in a *KML* document for this purpose.

createKMLDoc() Create the skeleton of a *KML* document, including the root node, the root's sole child, `<Document>`, and a name (*docName*) and description (*description*) for the document.

addStyles() Add style definitions (*docStyles*) to the document (*doc*) for use by the *KML* objects, e.g., placemarks and balloon windows.

addFolder() Put a `<Folder>` element in the node provided in *parent*, giving it a *name* and an optional *id*.

addPlacemark() Create a `<Placemark>` element with the coordinates provided (in the *point* parameter), as a child of the *parent* node, giving it a *name* and optional *description*, unique identifier (*id*), style (*styleUrl*), and *time*.

addTimeStamp() Add a `<TimeStamp>` element to the `<Placemark>` node provided in *parent*, giving it an optional *format*.

addDescription() Create a `<description>` element as a child of the *parent* node.

addGroundOverlay() Create a `<GroundOverlay>` element as a child of the *parent* node.

makeBallIcon() and makeRectIcon() Create an icon PNG image for use as a placemark, giving it a color (*col*), file name (*file*), and size (*width* and *height* for a rectangle or *radius* for a circle).

17.10 Further Reading

See [16] for the *KML* standards and [4] for a developer's guide. In addition, Chapters 4 and 5 of [9] contain a brief description of *KML* and other geo-location formats.

References

[1] Thomas Baier. `rcom`: *R* COM client interface and internal COM server. `http://cran.r-project.org/web/packages/rcom/index.html`, 2011. *R* package version 2.2-5.

[2] Gabriel Becker and Duncan Temple Lang. `RBrowserPlugin`: *R* in the Web browser. `https://github.com/gmbecker/RFirefox`, 2012. *R* package version 0.1-5.

[3] David Brillinger and Brent Stewart. Elephant-seal movements: Modelling migration. *Canadian Journal of Statistics*, 26:431–443, 1998.

[4] Google, Inc. Keyhole markup language (*KML*) reference. `https://developers.google.com/kml/documentation/kmlreference`, 2010.

[5] Google, Inc. Google Earth: A 3D virtual earth browser, version 6. http://www.google.com/earth/, 2011.
[6] Google, Inc. Google Maps: A Web mapping service application. http://maps.google.com/, 2011.
[7] Tomislav Hengl, Pierre Roudier, Dylan Beaudette, and Edzer Pebesma. plotKML: Visualization of spatial and spatio-temporal objects in Google Earth. http://cran.r-project.org/web/packages/plotKML/, 2012. *R* package version 0.3-2.
[8] Robert Hijmans and Jacob van Etten. raster: Geographic data analysis and modeling. http://cran.r-project.org/web/packages/raster/, 2012. *R* package version 2.0-41.
[9] Anthony T. Holdener. *HTML5 Geolocation*. O'Reilly Media, Inc., Sebastopol, CA, 2011.
[10] Michael Lawrence and Deepayan Sarkar. qtbase: Interface between *R* and Qt. http://cran.r-project.org/package=qtbase, 2011. *R* package version 1.0.4.
[11] Eric Lecoutre. R2HTML: *HTML* exportation for *R* objects. http://cran.r-project.org/package=R2HTML, 2011. *R* package version 2.2.
[12] Markus Loecher. RgoogleMaps: Overlays on Google map tiles in *R*. http://cran.r-project.org/package=RgoogleMaps, 2011. *R* package version 1.2.0.
[13] Erich Neuwirth. RColorBrewer: ColorBrewer palettes. http://CRAN.R-project.org/package=RColorBrewer, 2011. *R* package version 1.0-5.
[14] Deborah Nolan and Duncan Temple Lang. RKML: Simple tools for creating *KML* displays from *R*. http://www.omegahat.org/RKML/, 2011. *R* package version 0.7.
[15] Deborah Nolan and Duncan Temple Lang. SVGAnnotation: Tools for post-processing *SVG* plots created in *R*. http://www.omegahat.org/SVGAnnotation, 2011. *R* package version 0.9.
[16] Open Geospatial Consortium, Inc. OGC *KML* standards. http://www.opengeospatial.org/standards/kml/, 2010.
[17] Edzer Pebesma, Roger Bivand, Barry Rowlingson, and Virgilio Gomez-Rubio. sp: Classes and methods for spatial data. http://cran.r-project.org/web/packages/sp/, 2012. *R* package version 1.0-5.
[18] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. http://lmdvr.r-forge.r-project.org/figures/figures.html.
[19] Duncan Temple Lang. RDCOMEvents: Responding to *DCOM* events with *R* functions . http://www.omegahat.org/RDCOMEvents/, 2005. *R* package version 0.3-1.
[20] Duncan Temple Lang. RKMLDevice: Creating *R* plots in *KML*. http://www.omegahat.org/RKMLDevice/, 2010. *R* package version 0.1.
[21] Duncan Temple Lang. R2GoogleMaps: Create *HTML*/*JavaScript* documents to display data using Google Maps. http://www.omegahat.org/R2GoogleMaps, 2011. *R* package version 0.2-0.
[22] Duncan Temple Lang. RDCOMClient: Accessing *DCOM* services within *R*. http://www.omegahat.org/RDCOMClient/, 2011. *R* package version 0.93-0.
[23] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.
[24] Duncan Temple Lang. Rcompression: In-memory decompression for GNU **zip** and bzip2 formats. http://www.omegahat.org/Rcompression, 2012. *R* package version 0.94-0.
[25] USGS Earthquakes Hazards Program. Latest earthquakes: feeds and data. http://earthquake.usgs.gov/earthquakes/catalogs/, 2010.
[26] Hadley Wickham, Deborah Swayne, and David Poole. Bay Area blues: The Effect of the housing crisis. In Toby Segaran and Jeff Hammerbacher, editors, *Beautiful Data: The Stories Behind Elegant Data Solutions*, pages 303–322. O'Reilly Media, Inc., Sebastopol, CA, 2009.

Chapter 18
New Ways to Think about Documents

Abstract This chapter explores some ideas about how we can think of documents in several different ways from the traditional write–format–publish cycle. Instead, we think about documents in some of the same ways we do about software. We want to be able to programmatically verify that a document is "correct," i.e., unit testing for documents. We want modular components that we can reuse across documents and contexts. We want to be able to express relationships between elements of the document to map to our actual workflow, e.g., tasks and alternative approaches and not just sections, subsections and code chunks. We go beyond dynamic documents and report generation to try to capture the actual research process, concepts, and workflow, and not just their output. This involves semantic markup in the document. While we talk about *XML*, many of the ideas apply to any mark-up format which we can query and modify programmatically with convenient tools. However, some aspects exploit the richer hierarchical structure of *XML*, as opposed to the linear format of most document mark-up systems. The goal of this chapter is not to try to convert people to using the system we use to author documents, although it is available and quite powerful. Instead, we want to encourage people to demand more of the tools they use to author documents and to think about authoring documents in new, richer ways.

18.1 The Process of Authoring and Creating Documents

Most of us write different types of documents for different purposes. We author help files for *R* functions or data sets, vignettes, and tutorials describing a package, reports about data analyses we perform, journal articles about statistical methods or software we develop, and even lengthy books! We use various different systems to create the final document: a word processor, a text editor to write LaTeX, *Markdown*, or Sweave [2] content. Then, we generate the *PDF* or *HTML* version of the document and examine it to see if it is correct. Often, we have to return to the "code" for the document to change the formatting and generate the output again. And so the edit–view cycle continues. Aside from the formatting, we proofread the document to find errors in the "real" content, i.e., what we want the reader to understand.

If we think about this edit–view cycle, we recognize that we are essentially writing *code* for LaTeX or some other formatting system. We are running this code and debugging it based mostly on the output as it appears on the pages. We are slaves to the application that renders the final document, combing its output for errors and modifying the code to create the output. When we think about the proofreading step, this is done almost entirely by hand, or by human eye. Natural language processing

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0_18,

tools are getting better and can detect many grammatical issues. Unfortunately, many checks still have to be done by a human. However, there are many tasks in verifying the content of a document, especially technical documents involving, or about, computation, that can be done programmatically. That is, we can catch content errors via software we write. This is important as proofreading content is time-consuming, tedious, and hence very error-prone and not reproducible. We also have to do it multiple times as we refine the document and possibly introduce new errors.

In this chapter, we will discuss and illustrate ways to

1. programmatically validate aspects of a document;
2. treat the document as a database of text, code (in multiple languages), data, metadata, etc. that incorporates more than just what is presented to the reader;
3. "evaluate" parts of the document in *R*;
4. reuse content within multiple documents to avoid repeating ourselves (the DRY principle);
5. create an extensible mechanism for converting *XML* documents to other formats that is not as extensive as pandoc, but more general and extensible;
6. capture more than just the text and code and what we want the reader to see, but also the logic, the thought process and workflow in the actual research process, including how decisions were made, the different tasks in the workflow, different approaches for each task, and alternative implementations of these approaches.

All of these goals constitute a large and detailed topic, and we had originally planned on them being three chapters in a separate part of the book. However, this chapter will just give a brief overview of some of the ideas and also some of the tools we have developed to work with documents in these ways. We focus mainly on writing documents that involve computations, but the ideas can be relevant to all human-authored documents.

18.2 Validating a Document

To motivate validating documents programmatically, consider how we wrote this book and different articles and papers. We made numerous minor errors while focusing on the ideas we were trying to express and how to write about them. The following are some examples of these errors.

- We sometimes misspelled the names of *R* packages or *R* functions, e.g., .xmlrpc() instead of xml.rpc().
- Sometimes, we changed the name of a function in the source code of a package but did not update the material in the document or help files.
- We added a figure, planned to add the caption later, but forgot. Similarly, we did not put titles on some sections or left them empty, expecting to complete them at a later time.
- We mistyped the name of a graphics file we want to display.
- We used the wrong identifier or label when adding a cross-reference to another section or chapter in the text.
- We added a citation to a bibliography entry that we had not yet created and then either forgot to add it or used a different identifier than the one we had first used to refer to it.
- We forgot to add a bibliography entry for some packages.
- We did not use mark-up for some words or phrases that should appear consistently through the text, e.g., R rather than *R*, and XML rather than *XML*.
- We used the wrong mark-up for a concept, e.g., we used mark-up for denoting a file name when we meant file extension, or `<r:pkg>` when we meant `<bioc:pkg>` for a **Bioconductor** package.

- We had multiple entries for the same entity in the bibliography, some tables, etc.

In addition to finding mistakes, we also want to capture information in the text so that we can:

- know which functions to include in the table at the end of some of the chapters summarizing the important functions;
- programmatically add bibliography entries for the packages we have mentioned in the text but for which we have not yet already created a manual bibliography entry;
- verify that all the code that is supposed to work is working;
- determine all of the mark-up terms we use and ensure there are rules to render each of them;
- find all the hyperlinks to Web pages and verify they are correct;
- quickly find the locations of different parts of the text, e.g., all the titles, all the captions with no text, the sections with no titles, in order to read them or efficiently visit the correct file and line;
- add an attribute to each function to identify in which package it is located so that when we render it, we can have a hyperlink to its help page;
- find any chapter with an introduction containing more than one paragraph, as the publisher wants introductions in a single paragraph;
- identify the first use of an acronym and make certain its expansion/definition appears beside it, but not for other instances of that acronym, perhaps doing this per chapter;
- export the code in each example to a separate file based on the example name so that we can put it on the Web.

Each of these tasks is quite small, but taken together they represent a nontrivial amount of work for even a moderately sized or lengthy document, or one that undergoes multiple revisions. Our experiences are reasonably common and general. There are also many other types of errors people make, or information they want to query or insert into different types of documents. So, think of those examples above as just the sort of things we would like to be able to do and have tools to do them for us. Essentially, what we want are general regression tests for a document, the ability for an author to easily write context-specific tests, and also to be able to programmatically generate content for the document. If the tools were easy to create, we would think of many more tasks we can perform programmatically.

If we use LaTeX, Sweave or knitr, for example, to create the rendered version of our document, we will see error and warning messages about *some* of these errors, e.g., incorrect cross-references and bibliography citations. Unfortunately, some of these may get lost in the copious amount of output that is generated. It is nice to have a tool that collects these for us and allows us to explore them in different ways and groups. These are structural errors. We will not be informed of the errors such as misspelled *R* function or package names, missing captions or titles. We may have to identify those visually by reading the document. We also may not be using LaTeX as we may want a different target format. It is possible that pandoc may help us. In general, however, we want to be able to verify and correct the content without rendering it, easily find the different types of errors separately from the rendering, and easily build new tools to answer different queries.

Next, we look at some tools with which we can address some of these issues. Before we do, however, let's think about how we might do this if the document were written in LaTeX, using Sweave or a form of *Markdown*. These formats allow us to differentiate between (*R*) code and the text in a document. Unfortunately, they do not provide any facilities for structured access to the elements of the text, e.g., sections, titles, captions, figures, *R* function names. We can use regular expressions to find mark-up such as `\Rpkg{XML}` or `\Rfunc{plot}`. These are quite easy to find and will work in most cases. For more complex mark-up such as finding empty captions in a figure or missing titles within a section, we have an issue that the content of interest may be split across lines. For example, how do we find the caption inside a figure, but not a table, or the paragraphs within a

chapter's abstract? This requires not just simple pattern-matching of text, but also understanding the relationship of the text to structural and hierarchical elements of the document, e.g., chapters, figures, tables, sections, and so on. Regular expressions are very powerful and convenient for many simple or quick tasks. The following is an amusing comment about regular expressions by then Netscape engineer Jamie Zawinski in 1997 (https://groups.google.com/forum/?#!msg/alt.religion.emacs/DR057Srw5-c/Co-2L2BKn7UJ): 'Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.' This captures the fact that regular expressions can be quite brittle and awkward to use to develop robust and extensible software. They make some tasks easy. However, they cannot easily manage more complex tasks.

Rather than using regular expressions, we want a parser that understands the nature and structure of the document, identifies the different tokens, and provides us with a representation of the document in a structured, hierarchical manner, e.g., a list of chapters, a list of sections within each chapter, code, references to *R* functions and packages, etc. Building a parser for LaTeX or *Markdown* is not a major undertaking, but it is also nontrivial. To represent the result as anything more than a simple hierarchy involves some design decisions and class/type definitions. However, if the document was written using *XML* as the mark-up format, instead of LaTeX, Sweave or *Markdown*, we already have a parser that can return the entire hierarchical structure. We also have tools such as *XPath* to find the elements of interest. We do not have classes describing chapters, sections, figures, etc., but they are not needed as we can identify and query them by element names.

Many people shudder at the thought of authoring documents in *XML*. However, for writing articles, books, etc., the difference between *XML* and any of LaTeX, Sweave, and knitr is not significant. We still have to use mark-up. The verbose nature of *XML* to which many people object is similar to LaTeX. Compare

```
\begin{RCode}
 ....
\end{RCode}
```

and

```
<r:code>
 ....
</r:code>
```

Similarly, for knitr's *Markdown* format, we use

````
```{r}
 ....
```
````

XML, in contrast, is a general-purpose format that people use in many other contexts, as we have seen throughout the book. It provides namespaces, attributes, well-documented standards for entities, character encoding, etc., and, of course, many general tools for working with the content.

Tools such as nxml-mode for emacs makes authoring and editing *XML* reasonably straightforward. Additionally, we can create structured *XML* documents in a WYSWIG/visual word processor (e.g., Libre Office and Microsoft Word) and specialized *XML* editors such as XML Spy. Moreover, the features we are talking about are not exclusively tied to *XML*. Rather, we want some form of structured mark-up that captures semantics rather than formatting information, and one that we can easily query, manipulate, and modify programmatically. Given our use of *XML* for so many other applications, it makes sense to reuse these tools and technologies. If we had to create the entire development cycle for writing *XML* documents—developing the vocabulary/mark-up, the validation tools, the query tools,

projecting them to different formats or "views", and so on—this might be too much work. However, we exploit *DocBook* for the vocabulary, the *DocBook XSL* mappings to transform documents to *HTML*, *FO* and LaTeX formats, and LaTeX for creating the *PDF* output.

We addressed some of the example problems above that we ran into when writing this book and other documents with functions in the XDocTools [13] package, e.g., correcting *R* function and package names. Others such as finding summaries for chapters which have more than one paragraph are quite specific, so we wrote short *R* functions and *XPath* queries to implement them.

Verifying Package Names

Let's look at how we check the names of *R* packages and also verify there is a corresponding bibliography entry for each. We marked up packages as <*r:pkg*>, <*omg:pkg*> or <*bioc:pkg*>. Before we check if the spellings are correct, we can determine if we refer to any **Bioconductor** packages. We use a simple *XPath* query for this:

```
doc = xmlParse("book.xml")
length(getNodeSet(doc, "//bioc:pkg"))
```

There are some, so we do have to check these also.

Before we do this, we consider how we have determined the existence of some <*bioc:pkg*> elements in the actual text of the document. The xmlParse() function performs all of the relevant *XInclude* directives for just the parts we specified. The resulting document in `doc` is the correct subset of all of the different files in which we wrote the content of the book. Imagine if we had used regular expressions to find the text <*bioc:pkg*>. For convenience, we probably do this in a single call that operates recursively on the directories and all of their files. This includes files that we never explicitly include in the final document. It would also search parts of the files we do not *XInclude* since we can include subparts of an *XML* document and not necessarily the whole document. If the mark-up were LaTeX, it would search lines after `\endinputs`. Essentially, simple regular expressions have no context, and those that do typically become very complex and not line-oriented. In contrast, with *XML* mark-up, we can specify context. For example, we can omit any references to **Bioconductor** packages that are within an <*ignore*> or <*invisible*> node, or part of the bibliography with

```
getNodeSet(doc, "//bioc:pkg[not(ancestor::ignore) and
                            not(ancestor::invisible) and
                            not(ancestor::bibliography)]")
```

This gives much more specificity than regular expressions.

Now we can return to checking the function and package names. We can retrieve all of the different varieties of the <*pkg*> nodes with

```
pkgRefs = getNodeSet(doc, "//bioc:pkg | //r:pkg | //omg:pkg")
```

However, we will exclude those we mentioned above, i.e., in <*ignore*> and <*bibliography*> nodes. To do this, we construct our *XPath* expression programmatically:

```
pred = paste(sprintf("not(ancestor::%s)",
                     c("ignore", "bibliography")),
             collapse = " and ")
xp = sprintf("//%s[%s]", c("r:pkg", "bioc:pkg", "omg:pkg"), pred)
pkgRefs = getNodeSet(doc, paste(xp, collapse = " | "))
```

We could have worked with the package names directly (e.g., `XML`) rather than the nodes (e.g., `<r:pkg>XML</r:pkg>`. However, we want the nodes so that we can determine the name of the

repository in which to search for the package name, i.e., omg, r, or bioc. We also want the nodes so that we can report the file and line number of any that seem incorrect.

After we have the nodes, verifying the names of the packages is mostly unrelated to *XML*. We can query the different repositories for a list of their available packages and then check that each package name is in the specified repository. This also allows us to check that we have not marked up a **Bioconductor** package as being on CRAN (i.e., as <*r:pkg*>).

Sections with No Title
How do we find sections that have no title? A reasonably simple *XPath* expression will suffice:

```
//section[not(./title)]
```

Again, we probably want to skip those <*section*> elements that are within an <*ignore*> element. We can also find sections where the title is empty with

```
//section[not(./title) or (count(./title/*) = 0
           and count(./title/text()) = 0)]
```

We can find figures or tables that do not have a caption in much the same way with

```
//table[not(./caption)]
```

Finding the First Mention of a Package
We mentioned that we want to find the first mention of a package so that we can put a citation for it at that point and not at others. How do we find the first reference to a package? For the package XML package, we can use

```
fn = getNodeSet(doc, "//omg:pkg[text() = 'XML'][1]")
```

We can then find the file and line number with

```
getNodeLocation( fn )
```

We want to ignore elements within each chapter's overview/abstract so we can be more specific with

```
//omg:pkg[not(ancestor::summary) and text() = 'XML'][1]
```

How do we find the first reference to each package? We get the names of all the packages in the document and then loop over these separately in *R*. We can do this with a complex *XPath* query, but it is simpler in *R*. We do this for acronyms, but the approach is identical for packages. First, we find all the unique acronyms:

```
xp = "//acronym[not(ancestor::ignore)]"
acronyms = unique(xpathSApply(doc, xp, xmlValue))
```

Again, we add a predicate to ensure the node is not within an <*ignore*> element, analogous to `\begin{comment}` in LaTeX. We can loop over each of these and find the first instance with

```
sapply(acronyms,
       function(a) {
         xp = sprintf("//acronym[text() = '%s' and
                                 not(ancestor::ignore)]", a)
         getNodeSet(doc, xp)[[1]]
        })
```

We can either manually or programmatically check that these nodes have the necessary expansion either as an attribute or in the immediate preceding or following text.

We can also do this kind of querying and validation for word processing documents. We can find tables and figures with no captions, and so on. These are structural. We can also validate content. As long as people use styles to identify the concepts to which they are referring, e.g., *R* package, *R* function, *R* code, *HTTP* header element, we can also identify and validate this information programmatically. This is because the document is represented in a highly structured manner that we can manipulate within *R* or other languages. The use of named styles is the same as using conceptual mark-up such as `\Rfunc{plot}` rather than formatting information such as `\texttt{plot()}`. Not only does it allow us to define the formatting of all *R* functions consistently in a single location, it also conveys semantic meaning. Almost everyone has agreed this is the "right" thing to do, as evidenced by the dominance of *CSS* for Web pages.

We can also validate *HTML* documents and also the *CSS* documents used to display them. We can check *HTML* documents to verify links (both internal and external) are valid, the structure make sense and so on. We can also query the relationships between *HTML* elements and a *CSS* document. The RCSS [11] package interfaces to `libcroco`, a facility for parsing and querying *CSS* documents. RCSS allows us match a node in a *HTML* document to a *CSS* document and determine what style will apply. This allows us to validate that all the nodes we expect to have a style actually do, and allow us to examine these.

In addition to validating documents that we write manually in text editors or word processors, we can also validate and manipulate other documents such as spreadsheets, presentation slides produced with, e.g., Apple's Keynote or Microsoft's PowerPoint applications. We can also operate on figures we create with tools such as OmniGraffle. Since these are structured documents stored in *XML*, we can find elements of interest. We cannot use the same mark-up in these graphical interfaces, e.g., `<r:func>`, but we can perform some of the same kinds of verification for the names of *R* packages or functions in the text. This is especially true if authors use styles to identify the concept of particular content, e.g., *R* code, function names or packages.

18.3 Treating a Document as *R* Code

Authoring our documents to combine text and code is a very old idea. We put comments in code and then we have a hybrid document. However, the document is still source code. Knuth's literate programming concept is a significant advance on this, elevating text from comments to first-class elements in the document, and allowing interleaving of text and code in "chunks." This is the form Sweave and knitr [17] use. This transforms the document from being a source code file, to one that cannot be read directly by the language's parser, e.g., *R*. This turns things inside out. We have a document that contains *R* code, not an *R* source code document that contains text, e.g., roxygen2-annotated *R* code. So we need functionality to extract the code.

Sweave and knitr have tools to extract, or tangle, the code from the input document. For an *XML* document, it is quite easy to write *XPath* expressions to find the code. For example, we can use

```
getNodeSet(doc, "//r:code | //r:plot")
```

to find all `<r:code>` and `<r:plot>` nodes. The function xmlSource() in the XML package does this for us and handles many additional issues such as ignoring code that we explicitly do not want to evaluate by adding an *eval* attribute with a value of "false" or enclosing it within an `<ignore>` element. However, we can go further than this. We can evaluate the code in specific `<r:code>` or `<r:plot>` nodes using their *id* attributes, e.g.,

```
xmlSource("myDoc.R", ids = c("readData", "plot2"))
```

We can also read just the code within a particular section or subsection. We can identify the nodes within the desired section with an *XPath* expression. The xmlSource() function takes care of this for us when we specify the section's identifier via the *section* parameter. This can be a number or the section's identifier, i.e., its *id* attribute. This is not something that can be done easily or robustly with Sweave and knitr. The reason is that these do not try to examine the contents of the text chunks and cannot determine the hierarchy of the document. Therefore, they can not determine the path to a particular section or subsection.

18.3.1 Accessing Code Chunks via Variables

Regardless of the mark-up format of the document, we would rather not have to explicitly identify the code chunks to evaluate, e.g., by their identifiers (*id* attributes). Similarly, we do not necessarily want to have to evaluate all of the code chunks. Instead, we want to be able to easily instruct *R* to evaluate all the code up to (and including) where a particular variable is created, but not any of the chunks that are not needed to define that variable. This allows us to think in terms of the code, not the structure of the document. We would like to be able to do something analogous to

```
sourceDoc("myDoc.Rnw", variables = c("fit2"))
```

to evaluate all of the necessary expressions on which `fit2` depends. It is relatively easy to analyze *R* code in *R* itself, and so we can find the code chunks in which the variable of interest is defined. We can then evaluate all the code chunks up to that chunk. However, we can do better and determine the chunks or even individual expressions on which our target variable depends. We can do this by looking at the inputs to defining our variable, and then looking at where they are defined, and their own inputs, and so on recursively. This is not trivial, but the CodeDepends [14] package does this for us, in addition to other useful tasks.

Using CodeDepends, we start by reading our document with readScript(). This can read regular *R* source files, Sweave and *XML* documents and LaTeX documents using the Journal of Statistical Software's style. It can be easily extended to read code from other formats. Given the script object, we can evaluate the code corresponding to a particular variable with the sourceVariable() function. We can call it is as `sourceVariable(varName, script)` or indirectly with `script$varName`.

We use sourceVariable() with *R* scripts that do several related but separate tasks. Often there is some common code near the start of the script to do the general initialization, and then the separate tasks follow in a somewhat ad hoc sequence. Sometimes the tasks will share other general variables and code to initialize them. We do not want to have to determine which lines of code have to be run before others, and then have *R* evaluate those expressions by cutting-and-pasting them into *R* or highlighting them one after another in an editor and sending them to *R*. Instead, we use sourceVariable() and it determines what needs to be evaluated and does it for us.

The readScript() and sourceVariable() functions work well when the script is fixed and we just want to run parts of it. Often, we are iteratively editing the script and then evaluating parts of the modified or added code. If we use the object originally returned by readScript(), it will not know about the changes and so use the old code. We want a script object that checks to see if the original file has changed, and if so, to update itself and use the new contents. The updatingScript() function creates a script object very similar to readScript(). However, each time we use the script in a computation, it checks to see if the file from which it originally read the code has been updated. If the file is more recent, it reads it again to get the new contents and uses this new content for the computations.

The analysis the CodeDepends package does to determine dependencies can be used to implement caching for dynamic documents. Currently, the caching mechanisms used for Sweave and knitr and other dynamic systems use simple text matching, using hash strings for entire code chunks. If we reformat the same code in a chunk, or if we add a simple call to print() in a chunk, that will trigger re-evaluation of that chunk and all the chunks that depend on it. The CodeDepends mechanism gives us higher resolution, being able to work on individual top-level expressions and also allows us to use the semantics of the chunks rather than their lexical content. This allows us to determine which expressions within each chunk need to be re-evaluated to bring the overall state up-to-date. This can avoid lengthy unnecessary re-computation.

The CodeDepends package provides various other functionality that may be valuable for understanding code, chunks, and tasks, and generally the workflow associated with one or more scripts and collections of functions. For example, it can display the lifespan of variables in scripts, visualize relationships between the code chunks, and determine which packages are used by which chunks. We use the CodeDepends package as part of the `rProv` package [9] for tracking provenance of results in *R*. As a result, we expect that CodeDepends will evolve to become more valuable and robust over time. It currently does not detect dependencies via implicit changes to variables through indirect side-effects, e.g., reference classes or explicit calls to assign() in functions. The latter are rare, and identifying references classes is an issue that we hope to address.

18.4 Reusing Content in Different Documents

An important principle in software development is to avoid repeating ourselves and writing the same code in different places. We do not want to have to update the code in more than one place if we find a bug in it or need to enhance the code. We put the common code in a single place—a function. What is the analog for documents? We can include the same text content in different places within an overall document. For example, suppose we are writing two documents about the same topic but for different audiences. There may be parts that are identical such as particular paragraphs or a table of results. We want to have a single version of each these and incorporate those centralized versions into the two documents. Similarly, when we document *R* functions in a package, we document the parameters of each function in a separate Rd file. Often, several functions will have the same parameter with the same purpose. Currently, the *R* tools require us to repeat the documentation in each file. The `roxygen2` package [16] allows us to reuse the documentation for parameters, but not for other parts such as see also, value, references, or details sections.

We would like to have a user-level mechanism for specifying how to include part of one document within another. The key here is "part of" so that we can be very specific about which pieces we want. The *XInclude* mechanism for *XML* gives us this ability and control. We can include an entire file with `<xi:include href="filename"/>`. However, we can also use *XPointer* to specify the particular subset of elements to include in the other document. For example, we can include the second table with

```
<xi:include href="otherDoc.xml"
   xpointer="xpointer(//table[2])"/>
```

We can even include individual rows of a table or the caption of a figure with

```
<xi:include href="otherDoc.xml"
   xpointer="xpointer(//figure[@id='myFig]//caption)"/>
```

LaTeX and other systems have a means to include content from other files. We can use `\input` to include the contents of an entire file as if they were in the document itself. We have a small amount of control to limit what we include via the `\endinput` command. However, this does not allow us to specify which bits to include as *XInclude* does. It only stops the `\input` and ignores everything from that point onwards in the file being included. We cannot skip over parts at the beginning and we cannot include individual, noncontiguous pieces. To get the behavior we want, we define one or more conditions in the two LaTeX documents so that we can disable processing some content. Unfortunately, this means that we need to explicitly change the document to include some of its contents. This requires active organization if we own the two documents, and is not possible if we cannot change the document being included. In contrast, *XInclude* allows us to do what we want with local and remote documents whether we can edit them or not.

18.5 Capturing the Process and Paths of the Workflow

One of the limitations of Sweave, knitr and other dynamic document systems in *R* is that they are linear. By this, we mean that a text chunk is followed by a code chunk is followed by a text chunk, and so on. The chunks can be out of order, but when processed, the content is sequential. This is adequate for most situations and works very well for the primary uses of these systems, namely being able to generate a document that shows a particular set of results.

We think of documents as being richer than a single sequence of results. We want a document to contain all of the work we do on a particular project or task.[1] These activities include: dead-ends where we try something but it does not lead to an interesting result or conclusion; an alternative implementation of the same computations, perhaps to verify our original computations or a slow version we used initially; using an alternative statistical method to see if the results are similar; re-analyzing the data with outliers removed; or simply noting some interesting ideas or approaches we might try if we had more time. The document acts as a research notebook that captures the process of our work and the different paths through the activities. We can also annotate these with our thought process about why we pursued certain steps in the path and how these might have changed if the data or the results were slightly different. All of these additional pieces of information make a dynamic document into *reproducible research* as they capture the research steps, and not just the final steps and results. They allow reviewers and general readers to see if a particular line of reasoning was considered or even performed. They allow readers to understand how the conclusions may have changed with different data or a different choice of statistical method or assumptions. It allows researchers to keep all the work for reference, while showing only the important parts to the reader.

Documents that contain all of the activities within a research project are very rich. They may be made up of a combination of many collaborators' notebooks, making them even more complex. When we publish the results, we typically want to *project* the document to show just a subset. We call such a projection a view. This is the sequential document we typically write for Sweave or knitr. However, the notebook can be projected in different ways for different audiences into different views, each of which is sequential. This is appropriate for static documents appearing in printed form. However, when we can view the notebook in a Web browser, we can make all of the elements available to the reader in flexible ways. We can show them the different paths through the documents and let the reader select which to view. Indeed, the reader might view two or more computational paths through the document side-by-side to compare the choices made and the different results. We can also show them the steps

[1] The document itself may not actually contain all the text. We may use multiple files and use *XInclude* to combine these into the central document.

that make up an entire task, e.g., fitting a model, looking at the residuals, adapting the model, and refitting. We can show the reader an overview of the notebook's contents as a graph and give them an understanding of the overall workflow. Since all the code is also in the document, we can also allow the reader to explore the computations and the conclusions. We can add user interface controls such as sliders, radio buttons, check-boxes, and menus to allow the reader to try different settings. Some of this can be provided by the author, but some can be generated programmatically by analyzing the code, e.g., with the CodeDepends package.

In order to represent this rich content within a notebook document, we need some infrastructure. We could adapt Sweave or knitr for this. However, their use of simple "flat" code chunks makes this problematic. *XML*, on the other hand, gives us the hierarchical structure that we need. Furthermore, querying tasks and elements within a document or paths through the document is relatively easy with *XPath*. As we discuss below, we can also create the different projections using *XSL* or in any programming language with *XML* facilities.

We are continuing to develop the *XML* vocabulary to describe the elements of the notebook (`<task>`, `<path>`, `<thread>`, `<alternative>`). We also have functionality in the CodeDepends package that allows us to manipulate these documents as if they were regular blocks of code. For instance, the xmlSource() function we saw earlier can allow us to evaluate all the code in a particular (sub-) path. We could do this for two alternative paths and plot the different results to compare the approaches. We are also working with Gabriel Becker to develop the interactive dynamic documents that allow the reader to interact with the entire notebook and to interactively adapt the computations with sliders, etc. We are developing this functionality in a Web browser for viewing the notebook, leveraging Web technologies, and also adapting iPython to author and display the contents. Additionally, we are exploring how readers can remotely augment the material in the original document and how authors can incorporate other readers' thoughts. All of this happens without changing the document, but via a network of linked documents with the links applying to specific subparts and elements of the original document.

The important aspect of this section is that we can go a lot further than simple, linear dynamic documents. These linear documents capture the *results* of reproducible computations. They do not capture reproducible research as they do not capture the other activities that are not being reported. The simple structure commonly used these days does not allow for capturing the thought process, dead-ends, and confirmatory activities that we do not want to publish. Regardless of how we represent the content, we want to go beyond simple linear documents to notebooks that capture all of the information from all the tasks from all of the participants.

18.6 Using *XSL* to Transforming *XML* Documents

We can transform *XML* documents to any format using the tools we discussed in other chapters of this book and many different programming languages. However, the eXtensible Stylesheet Language (*XSL*) is a language explicitly designed to transform *XML* content to text or other *XML*/*HTML* documents. We use *XSL* to take a document we author (such as this book) using the *DocBook* format and to generate an *HTML* or *PDF* version from it. We can use *XSL* for processing and transforming any type of *XML* document, e.g., extracting data from a *KML* file or converting an *SVG* graphical display to *JavaScript* code. However, we use it almost exclusively for creating *HTML* and *PDF* versions of articles, books, etc., including the enhanced documents with tasks, threads, and alternatives.

Since we are focusing on transforming *XML*/*DocBook* documents to *HTML* and *PDF* (either via LaTeX or formatting objects (FO)), it makes sense to think of how *XSL* is similar to LaTeX. Basi-

cally, we have mark-up in our documents for structure and content. The structural elements include <section>, <title>, <table>, <figure> and <para> (for paragraph). Content elements include <r:func> to refer to an *R* function, <r:code> to identify verbatim *R* code, <ulink> for a *URL* and <citation>. To render these *XML* elements in LaTeX, we need to transform them to corresponding LaTeX content. For this, we define templates that match a particular *XML* element. The template then acts like a function or rule to operate on the particular instances of that element in the *XML* document. For example, to process an <r:func> element, we might transform it directly to

```
\textsl{\color{rfunc}plot()}
```

Alternatively, we might leave the appearance to a LaTeX command and so create

```
\Rfunc{plot}
```

Our processing adds an extra step in generating our final document. We transform from *XML* to LaTeX and then from LaTeX to *PDF*. This extra step allows us to chose where to perform the different transformations, in addition to allowing us to project to very different views and different formats.

We write our *XSL* template for the first form of output above as

```
<xsl:template match="r:func">
\textsl{\color{rfunc}<xsl:apply-templates/>()}
</xsl:template>
```

We specify for which *XML* elements the template is indented using the *match* attribute. Here we are using the name of the element, qualified by its namespace prefix. We will see we can use much more powerful *XPath*-like expressions to dynamically identify the target *XML* elements based on their content.

The *XSL* content within the <xsl:template> element is what generates the output. Here we have basically specified the content literally, e.g., \textsl{\color{..}...}. However, we need to insert the actual *R* function name into the output. We can access this in various ways. We expect the *XML* on which the template operates to be something like <r:func>plot</r:func>. To get the text in the <r:func> node (i.e. plot), we might use

```
<xsl:value-of select="string(.)"/>
```

However, it is easier and more general to use

```
<xsl:apply-templates/>
```

By default, this recursively processes the children (and attributes) of the current node being processed. The *XSL* processor will find the appropriate templates for each of these children (including text) and apply those to generate the output.

Importantly, and quite differently from LaTeX, the actions in the template can access the element's parent or any ancestor and indeed any part of the *XML* document via *XPath* expressions. We do not need extra arguments as we do in LaTeX macros. For example, suppose we also want to be able to add a link to the help page for the function if the *XML* element has a *pkg* attribute. We can do this with

```
<xsl:template match="r:func">
\Rfunc
<xsl:if test="@pkg">[<xsl:value-of select="@pkg"/>]</xsl:if>
{<xsl:apply-templates/>()}
</xsl:template>
```

to produce

```
\Rfunc[XML]{readHTMLTable}
```

from `<r:func pkg="XML">readHTMLTable</r:func>`

A second important difference is that we can define a template to match elements in specific contexts. In LaTeX, we define a macro or environment based on its name. In *XSL*, we can also do that, but we can use any *XPath* expression. For example, if we want to process a `<title>` element for a `<section>` differently from a `<title>` in a `<figure>` element, we can do so. We can write two different templates as

```
<xsl:template match="section/title">...</xsl:template>

<xsl:template match="figure/title">...</xsl:template>
```

This is very useful and powerful, and greatly facilitates expressing transformations. In our example that adds the *pkg* attribute to the output, we could have avoided the `<xsl:if>` step by using two templates:

```
<xsl:template match="r:func">
\Rfunc{<xsl:apply-templates/>}
</xsl:template>

<xsl:template match="r:func[@pkg]">
\Rfunc[<xsl:value-of select="@pkg"/>]{<xsl:apply-templates/>}
</xsl:template>
```

XSL is a powerful language that makes some tasks very simple. However, it can also be awkward and verbose. Importantly, it represents a different programming model and way of thinking about transforming documents that we might borrow when working with documents generally.

Extensibility

Like *XML*, the X in *XSL* stands for extensibility. What this means is quite simple and powerful. Suppose we introduce a new mark-up element in our *XML* document/vocabulary. For example, suppose we want to use the element `<r:refClass>` to identify the name of an *R* reference class. We want to differentiate the class from an *S3* or *S4* class so we use a new *XML* element. This will allow us to easily query the document for all the `<refClass>` elements and to verify they are spelled correctly and are defined, to add or verify the package name, and to put a link to the help page when we render it.

In order to have these `<r:refClass>` elements appear in the rendered document (e.g., *PDF*), we need to add an *XSL* template to process them. For LaTeX, we might want these to appear as

```
\RefClass{name}\sindex{name@\RefClass{name}}
```

We can define the *XSL* template as

```
<xsl:template match="r:refClass">
\RefClass{<xsl:apply-templates}\sindex{
<xsl:call-template name="indexSortString"/>
\RefClass{<xsl:apply-templates/>}}
</xsl:template>
```

Next, we need to tell our *XSL* transformer to use this template when converting the document from *DocBook* to LaTeX. We do this by writing our own *XSL* document that both contains our new template(s) and also merges all of the others that we want to use. This *XSL* document looks something

like the following, consisting of an <xsl:stylesheet> root node, an <xsl:import> instruction to load the other *XSL* documents, and our new templates:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:r="http://www.r-project.org">

<xsl:include
   href="http://www.omegahat.org/XSL/latex/basicLatex.xsl"/>

<xsl:template match="r:refClass">
\RefClass{<xsl:apply-templates}\sindex{
<xsl:call-template name="indexSortString"/>
\RefClass{<xsl:apply-templates/>}}
</xsl:template>
</xsl:stylesheet>
```

When we want to convert our *XML* document to LaTeX, we use this new *XSL* document with our extensions. If we want to convert the *XML* to *HTML*, we create a similar *XSL* document, but change the template to create *HTML* content for <r:refClass> elements. We can also change the <xsl:import> to bring in the templates for creating *HTML* output. Having to write a separate *XSL* file for each target format is slightly irritating, but it makes sense given the different natures of the formats. In some cases, we can share some of the *XSL* code that is format-independent. We can even write templates that act as callable functions in *XSL*, e.g., as in the call to the `indexSortString` template in the example above.

18.6.1 XSL in R

We can use an *XSL* document (and included documents) to transform an *XML* document using any *XSLT* processor such as `xsltproc`. We apply an *XSL* document to an *XML* document and obtain a new document. We can also do this directly in *R* with a little more flexibility and control than calling an external application such as `xsltproc`. The Sxslt package [12] gives us full access to `xsltproc` from within *R*. We can apply a style sheet to an *XML* document using the function xsltApplyStyleSheet(). We can apply the same style sheet to numerous target *XML* documents. We can also load several style sheets and apply each of them in sequence to one or more documents, i.e., the output document from the first transformation becomes the input to the next transformation.

The Sxslt package also allows us to do more advanced things with *XSL*. We can combine both *R* and *XSL* so that *XSL* templates can call *R* functions. This allows us, for example, to extend *XSL*'s string manipulation facilities using *R*'s functionality. For instance, we can use regular expressions by invoking *R*'s grep() and gsub() functions from *XSL*. We can also shift some of the manipulation of *XML* nodes within templates to *R* code and use the functionality of the XML package. From *R* code, we can both query nodes in the input *XML* document, and also create new nodes and insert them into the target/output document. We use this to transform dynamic documents (similar to Sweave and knitr) in a single pass. We can combine the templates for transforming and rendering the text of the document along side the templates to evaluate the *R* code within the document and to insert the results into the output.

The Sxslt package also provides functionality to understand and explore *XSL* documents. Often our main *XSL* document includes/imports numerous other *XSL* documents, which also include other documents, and so on. This gives us the full set of templates. However, this hierarchy of inclusions and overriding of templates with replacements or more specific versions can make it challenging to identify precisely which template will apply to a particular node in the input *XML* document. We can use the function getTemplate() to find the matching template for a given node or node name. We can find the file and line number of the template with getNodeLocation(). We can also get the hierarchy of inclusions with getXSLImports() and even visualize it as a tree.

We also use *XML* and *XSL* for writing to-do files. As we develop code or think of new things to do for an analysis, we add an item to the to-do file. We can arrange them into topics. We add a basic description of the to-do item, perhaps with some additional information about how to reproduce, e.g., a bug in the code. We format this just as we would when writing an article or whatever. In other words, we can write content that will be displayed nicely when we view the to-do file. We can also add additional information that can be hidden from display, but that is still available so that we can track the evolution of how we have dealt with this item. Furthermore, we use a *status* attribute to indicate the priority of the item or whether it is complete, needs to be tested, or has not been dealt with yet.

The *XML* to-do list is convenient as a way of storing the content in an organized manner. However, org-mode in emacs is probably simpler. The key to our *XML* approach is that we can associate an *XSL* file at the top of the `Todo.xml` file. When we open this file in a Web browser and each time we reload the page, the browser processes the *XML* file with the *XSL* file instructions to create an *HTML* document. For our to-do files, the *XSL* transformation creates a dashboard of the different topics with the different priorities, and displays these in an integrated manner. We see how many items there are overall and within each category/topic. We see how many items there are for each status, within each topic. This is a dynamic page that updates itself as we change the underlying `Todo.xml` file. The template format and *XSL* files that dynamically render them are available at `http://www.omegahat.org/XSL/Todo`. I find this approach quite valuable as it allows me to easily collect tasks as I encounter them, to add information about them that is rich and can be rendered properly, and, most importantly, that I can query. Often, after resolving a to-do item, the code and comments migrate to be a test or an example for a package.

Structured, programmatically queryable, and modifiable documents can be valuable in many other circumstances. For example, generating exams, homework assignments, data sets, examples and help files can all benefit from dynamic substitution. We have just hinted at some of the ideas and tools. As publishing documents on the Web and in printed form continues to become richer and more varied, we want to be able to take advantages of technologies to not only display content in new ways, but also to improve the quality and correctness of the content. We want tools to streamline the publishing process and allow us to more rapidly express ourselves accurately.

18.7 Further Reading

In addition to the documentation for the XDocTools, XDynDocs, CodeDepends and Sxslt packages [13, 7, 14, 12], a few articles [4, 6, 5] describe our approach.

There are several books that introduce both *XSL* and also *XSL* for converting *DocBook* documents to *HTML*, *FO*, and LaTeX: [1] is a succinct but comprehensive overview of *XSL* 1.0; [15] covers both versions 1.0 and 2.0 of *XSL*; [3] contains many practical examples, tips, and strategies for performing useful computations in *XSL*. For converting *DocBook* documents to various different formats, [10]

is a very useful overview and reference. While those of us who use mathematical content in our documents may prefer to use LaTeX as the typesetting system, *FO* [8] can be useful.

References

[1] Neil Bradley. *The XSL Companion*. Addison-Wesley, London, 2000.

[2] Friedrich Leisch. Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Roenz, editors, *Compstat 2002, Proceedings in Computational Statistics*, pages 575–580. Physika Verlag, Heidelberg, 2002.

[3] Sal Mangano. *XSLT Cookbook: Solutions and Examples for XML and XSLT Developers*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[4] Deborah Nolan, Roger Peng, and Duncan Temple Lang. Enhanced dynamic documents for reproducible research. In M.F. Ochs, J.T. Casagrande, and R.V. Davuluri, editors, *Biomedical Informatics for Cancer Research*, pages 335–346. Springer-Verlag, New York, 2009.

[5] Deborah Nolan and Duncan Temple Lang. Dynamic, interactive documents for teaching statistical practice. *International Statistical Review*, 75:295–321, 2007.

[6] Deborah Nolan and Duncan Temple Lang. Learning from the statistician's lab notebook. In *Data and Context in Statistics Education: Towards an Evidence-based Society. Proceedings of the Eighth International Conference on Teaching Statistics (ICOTS8, July, 2010), Ljubljana, Slovenia.* Voorburg, 2010.

[7] Deborah Nolan and Duncan Temple Lang. XDynDocs: Dynamic documents with *XML* and *XSL*. `http://www.omegahat.org/XDynDocs`, 2011. *R* package version 0.3-1.

[8] Dave Pawson. *XSL-FO: Making XML Look Good in Print*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[9] Karthik Ram and Duncan Temple Lang. `rProv`: Provenance tracking in *R*. `https://github.com/karthikram/RProvenance`, 2012. *R* package version 0.2-0.

[10] Bob Stayton. *DocBook XSL: The Complete Guide*. Sagehill Enterprises, Santa Cruz, CA, fourth edition, 2007.

[11] Duncan Temple Lang. RCSS: Facilities for reading and working with *CSS* files in *R*. `http://www.omegahat.org/RCSS`, 2011. *R* package version 0.2-0.

[12] Duncan Temple Lang. Sxslt: *R* extension for `liblibxslt`. `http://www.omegahat.org/Sxslt`, 2011. *R* package version 0.91-1.

[13] Duncan Temple Lang. XDocTools: Tools for working with *XML* and *XSL* documents. `http://www.omegahat.org/XDocTools`, 2011. *R* package version 0.1-0.

[14] Duncan Temple Lang, Roger Peng, and Deborah Nolan. CodeDepends: Analysis of *R* code for reproducible research and code comprehension. `http://www.omegahat.org/CodeDepends`, 2011. *R* package version 0.2-1.

[15] Jeni Tennison. *Beginning XSLT 2.0: From Novice to Professional*. Apress, Berkeley, CA, 2005.

[16] Hadley Wickham, Peter Danenberg, and Manuel Eugster. `roxygen2`: A Doxygen-like in-source documentation system for Rd, collation, and NAMESPACE. `http://cran.r-project.org/web/packages/roxygen2/`, 2011. *R* package version 2.2.2.

[17] Yihui Xie. `knitr`: A general-purpose package for dynamic report generation in *R*. `http://cran.r-project.org/web/packages/knitr/index.html`, 2013. *R* package version 1.0.

Bibliography

[1] 10gen, Inc. The MongoDB NoSQL database. http://www.mongodb.org, 2011.

[2] Daniel Adler. rdyncall: Improved foreign function interface (FFI) and dynamic bindings to *C* libraries (e.g., *OpenGL*). http://cran.r-project.org/package=rdyncall, 2012. *R* package version 0.7.5.

[3] Adobe. Flash player software, version 10.3. http://get.adobe.com/flashplayer/, 2011.

[4] Advogato. The Advogato Community resource site for developers of free software. http://www.Advogato.org, 2011.

[5] Amazon Web Services, Inc. Amazon simple storage service (Amazon S3). http://aws.amazon.com/s3/, 2012.

[6] J. Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[7] Apache Software Foundation. The Apache Batik project: A *Java*-based toolkit for applications or applets that want to use images in scalable vector graphics (*SVG*), version 1.7. http://xmlgraphics.apache.org/batik/, 2008.

[8] Apache Software Foundation. Apache Lucene: Open-source search software. http://lucene.apache.org, 2011.

[9] Apache Software Foundation. OpenOffice: The free and open productivity suite; 3.0 New Features. http://www.openoffice.org/dev_docs/features/3.0/, 2011.

[10] Apache Software Foundation. Spam Assassin: A spam filter that can be used on a wide variety of email systems. http://spamassassin.apache.org, 2011.

[11] Apple, Inc. Numbers for iOS: Supported file formats. http://support.apple.com/kb/HT4642, 2011.

[12] ASA Sections on Statistical Computing and Graphics. Data Expo 09: Airline on-time performance. http://stat-computing.org/dataexpo/2009/, 2009.

[13] Thomas Baier. rcom: *R* COM client interface and internal COM server. http://cran.r-project.org/web/packages/rcom/index.html, 2011. *R* package version 2.2-5.

[14] Shay Banon. Elasticsearch: An open source, distributed, *REST*ful search engine. http://www.elasticsearch.org, 2011.

[15] Gabriel Becker and Duncan Temple Lang. RBrowserPlugin: *R* in the Web browser. https://github.com/gmbecker/RFirefox, 2012. *R* package version 0.1-5.

[16] Richard Becker, Allan Wilks, Ray Brownrigg, and Thomas Minka. maps: Draw geographical maps. http://cran.r-project.org/web/packages/maps/, 2011. *R* package version 2.1.

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014

[17] Alex Berger, Alex Pucher, Alexandra Medwedeff, Andreas Neumann, Andre Winter, Christian Furpass, Christian Resch, Florent Chuffart, Florian Jurgeit, Georg Held, Greg Sepesi, Iris Fibinger, Klaus Forster, Martin Galanda, Nedjo Rogers, Nicole Ueberschar, Peter Sykora, Sudhir Kumar Reddy Maddirala, Thomas Mailander, Till Voswinckel, Tobias Bruehlmeier, Torsten Ullrich, and Yvonne Barth. Carto.net software. http://www.carto.net, 2010.

[18] Anders Berglund. Extensible Stylesheet Language (*XSL*) Version 1.1. Worldwide Web Consortium, 2006. http://www.w3.org/TR/xsl.

[19] Arif Bilgin, Don Caldwell, John Ellson, Emden Gansner, Yifan Hu, and Stephen North. Graph visualization software: Drawing graphs since 1988. http://www.graphviz.org/, 2012.

[20] Thomas Bock. R4CouchDB: Collection of *R* functions for CouchDB access. https://github.com/wactbprot/R4CouchDB, 2011. *R* package version 0.08.

[21] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. Cascading style sheets, level 2, revision 1 (CSS 2.1) specification. Worldwide Web Consortium, 2011. http://www.w3.org/TR/CSS2/.

[22] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP), version 1.1. Worldwide Web Consortium, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[23] Neil Bradley. *The XSL Companion*. Addison-Wesley, London, 2000.

[24] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry Thompson. Namespaces in *XML* 1.0. Worldwide Web Consortium, 2009. http://www.w3.org/TR/REC-xml-names/.

[25] David Brillinger and Brent Stewart. Elephant-seal movements: Modelling migration. *Canadian Journal of Statistics*, 26:431–443, 1998.

[26] Michael Brundage. *XQuery: The XML Query Language*. Addison Wesley, Boston, MA, 2004.

[27] Cairo Graphics. Cairo: A 2D graphics library with support for multiple output devices, version 1.1. http://www.cairographics.org, 2006.

[28] Dan Carr, Nicholas Lewin-Koh, and Martin Maechler. hexbin: Hexagonal binning routines. http://www.bioconductor.org/packages/2.6/bioc/html/hexbin.html, 2009. *R* package version 1.22.

[29] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[30] John M Chambers. *Software for Data Analysis: Programming with R*. Springer, New York, 2008.

[31] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weeraearana. Web Service Description Language (WSDL) 1.1. Worldwide Web Consortium, 2001. http://www.w3.org/TR/wsdl.

[32] Civic Impulse, LLC. GovTrack.us developer documentation. http://www.govtrack.us/developers, 2012.

[33] James Clark. *XSL* transformations (*XSLT*). Worldwide Web Consortium, 1999. http://www.w3.org/TR/xslt.

[34] James Clark. nXML mode: An addon for GNU Emacs. http://www.thaiopensource.com/nxml-mode/, 2004.

[35] Alex Couture-Beil. rjson: Converts *R* object into *JSON* and vice-versa. http://cran.r-project.org/web/packages/rjson/, 2011. *R* package version 0.2.6.

[36] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[37] Gabor Csardi. igraph: Network analysis and visualization. http://cran.r-project.org/web/packages/igraph/index.html, 2012. *R* package version 0.6.4.

[38] Data Mining Group. Predictive Model Markup Language. http://www.dmg.org/pmml-v3-2.html, 2011.

[39] Adrian Dragulescu. xlsx: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files. http://cran.r-project.org/package=xlsx, 2011. *R* package version 0.5.0.

[40] Sandrine Dudoit, Sunduz Keles, and Duncan Temple Lang. RHTMLForms: Programmatically create *R* functions corresponding to Web/*HTML* forms. http://www.omegahat.org/RHTMLForms, 2012. *R* package version 0.6-0.

[41] ECMA International. Ecma Office Open *XML* file formats standard, Part 3: Primer. http://www.ecma-international.org/news/TC45_current_work/TC45_available_docs.htm, 2011.

[42] Economic Commission for Europe. Common open standards for the exchange and sharing of socio-economic data and metadata: The SDMX initiative. http://sdmx.org/docs/2002/wp11.pdf, 2002.

[43] Dirk Eddelbuettel and Romain Francois. Rcpp: Seamless *R* and *C++* integration. http://cran.r-project.org/package=Rcpp, 2011. *R* package version 0.9.15.

[44] J David Eisenberg. *SVG Essentials*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[45] Bryan English. JPath: A *JavaScript* class which provides an *XPath*-like querying ability to *JSON* objects. http://bluelinecity.com/software/jpath, 2011.

[46] European Central Bank. Euro foreign exchange reference rates. http://www.ecb.int/stats/exchange/eurofxref/html/index.en.html, 2011.

[47] European Central Bank. SDMX-ML and SDMX-EDI (GESMES/TS): The ECB statistical representation standards. http://www.ecb.int/stats/services/sdmx/html/index.en.html, 2011.

[48] David Fallside and Priscilla Walmsley. *XML* schema, Part 0: Primer. Worldwide Web Consortium, 2004. http://www.w3.org/TR/xmlschema-0/.

[49] Federal Election Commission. Top 50 house incumbents by contributions from PACs and other committees, January 1, 2011 – June 30, 2011. http://www.fec.gov/press/summaries/2012/PAC/6mnth/1pac6mosummary11.xlsx, 2011.

[50] R. Fielding, J. Gettys, J. Mugul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol: *HTTP*/1.1. Worldwide Web Consortium, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[51] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[52] FLOWR Foundation. *Zorba*: The *XQuery* processor. http://www.zorba-xquery.com, 2012.

[53] A. Frank and A. Asuncion. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2010.

[54] Robin Gareus. liboauth: *OAuth* Library functions, version 1.0.0. http://liboauth.sourceforge.net, 2012.

[55] R. Gentleman, Elizabeth Whalen, W. Huber, and S. Falcon. graph: A package to handle graph data structures. http://cran.r-project.org/package=graph, 2011. *R* package version 1.33.0.

[56] Robert Gentleman. *R Programming for Bioinformatics*. Chapman & Hall/CRC, Boca Raton, FL, 2009.

[57] Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

[58] Jeff Gentry, Robert Gentleman, and Wolfgang Huber. How to plot a graph using Rgraphviz. http://bioconductor.org/packages/2.6/bioc/vignettes/

Rgraphviz/inst/doc/Rgraphviz.pdf, 2010.

[59] Jeff Gentry, Li Long, Robert Gentleman, Seth Falcon, Florian Hahne, Deepayan Sarkar, and Kaspar Hansen. Rgraphviz: Provides plotting capabilities for *R* graph objects. http://www.bioconductor.org/packages/2.11/bioc/html/Rgraphviz.html, 2011. *R* package version 2.2.1.

[60] Jeff Gentry and Duncan Temple Lang. ROAuth: *R* interface for *OAuth*. http://cran.r-project.org/web/packages/ROAuth/index.html, 2012. *R* package version 0.9.2.

[61] Markus Gesmann and Diego de Castillo. googleVis: Interface between *R* and the Google Visualisation API. http://cran.r-project.org/package=googleVis, 2011. *R* package version 0.2.13.

[62] GlobalGiving Foundation. Globalgiving: Donate to projects around the world supporting disaster relief, education, health, women and children, and more. http://www.globalgiving.org/, 2012.

[63] Stefan Goessner. *JSON* Path – *XPath* for *JSON*. http://goessner.net/articles/JsonPath/, 2011.

[64] Google, Inc. Keyhole markup language (*KML*) reference. https://developers.google.com/kml/documentation/kmlreference, 2010.

[65] Google, Inc. Google cloud storage: A *REST*ful service for storing and accessing data on Google's networking infrastructure. https://developers.google.com/storage/, 2011.

[66] Google, Inc. Google Earth: A 3D virtual earth browser, version 6. http://www.google.com/earth/, 2011.

[67] Google, Inc. Google Maps: A Web mapping service application. http://maps.google.com/, 2011.

[68] Google, Inc. Google News: A news aggregator service. http://news.google.com, 2011.

[69] Google, Inc. Google APIs console. https://code.google.com/apis/console/, 2012.

[70] Google, Inc. Google documents list API: Allows developers to create, retrieve, update, and delete Google Docs. http://code.google.com/apis/documents/, 2012.

[71] Google, Inc. Google Sky: An online outer-space viewer. http://www.google.com/sky/, 2012.

[72] Google, Inc. Google Visualization API reference. https://developers.google.com/chart/interactive/docs/reference, 2012.

[73] Google, Inc. Using *OAuth* 2.0 to access Google APIs. https://developers.google.com/accounts/docs/OAuth2, 2012.

[74] David Gourley and Brian Totty. *HTTP: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[75] John Gruber. Markdown: A text-to-*HTML* conversion tool for Web writers. http://daringfireball.net/projects/markdown/, 2004.

[76] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework*. Worldwide Web Consortium, 2007.

[77] Florian Hahne. Rgraphviz: A new interface to render graphs using Rgraphviz. http://www.bioconductor.org/packages/2.11/bioc/vignettes/Rgraphviz/inst/doc/newRgraphvizInterface.pdf, 2012.

[78] Eran Hammer. The *OAuth* 1.0 guide. http://hueniverse.com/oauth/guide/, 2011.

[79] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5849.

[80] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.

[81] Tomislav Hengl, Pierre Roudier, Dylan Beaudette, and Edzer Pebesma. plotKML: Visualization of spatial and spatio-temporal objects in Google Earth. http://cran.r-project.org/web/packages/plotKML/, 2012. *R* package version 0.3-2.

[82] Ian Hickson. *HTML5*: A vocabulary and associated APIs for *HTML* and *XHTML*. Worldwide Web Consortium, 2011. http://www.w3.org/TR/html5/.

[83] Robert Hijmans and Jacob van Etten. raster: Geographic data analysis and modeling. http://cran.r-project.org/web/packages/raster/, 2012. *R* package version 2.0-41.

[84] Anthony T. Holdener. *HTML5 Geolocation*. O'Reilly Media, Inc., Sebastopol, CA, 2011.

[85] Jeffrey Horner. rApache: Web application development with *R* and Apache. http://rapache.net/, 2011. *R* package version 1.2.1.

[86] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.

[87] Lars Magne Ingebrigtsen. Gmane: A public mailing list archive. http://gmane.org, 2011.

[88] *JSON* Advocate Group. Introducing *JSON*: A lightweight data-interchange format. http://www.json.org/, 2006.

[89] Minoru Kanehisa. KEGG: Kyoto Encyclopedia of Genes and Genomes. http://www.genome.jp/kegg/, 2012.

[90] Chafic Kazoun and Joey Lott. *Programming Flex 3: The Comprehensive Guide to Creating Rich Internet Applications with Adobe Flex*. O'Reilly Media, Inc., Sebastopol, CA, 2008.

[91] KDE e.V. KOffice: Standards-compliant office and productivity applications. http://userbase.kde.org/KOffice, 2011.

[92] Bill Kennedy and Chuck Musciano. *HTML and XHTML: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.

[93] Eric Kidd. XMLRPC how to. http://tldp.org/HOWTO/XML-RPC-HOWTO, 2001.

[94] Kiva Organization. Kiva: Loans that change lives. http://www.kiva.org/, 2011.

[95] Murat Koksalan, Jyrki Wallenius, and Stanley Zionts. *Multiple Criteria Decision Making: From Early History to the 21st Century*. World Scientific Publishing Co. Pte. Ltd., Singapore, 2011.

[96] B. N. Lawrence, R. Lowry, P. Miller, H. Snaith, and A. Woolf. Information in environmental data grids. *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences*, 367:1003–1014, 2009.

[97] Michael Lawrence and Deepayan Sarkar. qtbase: Interface between *R* and Qt. http://cran.r-project.org/package=qtbase, 2011. *R* package version 1.0.4.

[98] Johnathan LeBlanc. *Programming Social Applications: Building Viral Experiences with OpenSocial, OAuth, OpenID, and Distributed Web Frameworks*. O'Reilly Media / Yahoo Press, Sebastopol, CA, 2011.

[99] Eric Lecoutre. R2HTML: *HTML* exportation for *R* objects. http://cran.r-project.org/package=R2HTML, 2011. *R* package version 2.2.

[100] Friedrich Leisch. Dynamic generation of statistical reports using literate data analysis. In W. Härdle and B. Roenz, editors, *Compstat 2002, Proceedings in Computational Statistics*, pages 575–580. Physika Verlag, Heidelberg, 2002.

[101] Library of Congress. MODS: Metadata Object Description Schema. http://www.loc.gov/standards/mods/mods.xsd, 2010.
[102] LibreOffice; The Document Foundation. Calc: The LibreOffice spreadsheet program. http://www.libreoffice.org/features/calc/, 2011.
[103] Gerald Lindsly. rmongodb: *R*-MongoDB driver. http://cran.r-project.org/package=rmongodb, 2011. *R* package version 1.0.3.
[104] Markus Loecher. RgoogleMaps: Overlays on Google map tiles in *R*. http://cran.r-project.org/package=RgoogleMaps, 2011. *R* package version 1.2.0.
[105] Jake Luciani. RSvgDevice: An *RSVG* graphics device. http://cran.r-project.org/web/packages/RSvgDevice/, 2009. *R* package version 0.6.
[106] Sal Mangano. *XSLT Cookbook: Solutions and Examples for XML and XSLT Developers*. O'Reilly Media, Inc., Sebastopol, CA, 2006.
[107] R.G. Mann, R.M. Baxter, R. Carroll, Q. Wen, O.P. Buneman, B. Choi, W. Fan, R.W.O. Hutchison, and S.D. Viglas. *XML* Data in the virtual observatory. *Astronomical Data Analysis Software and Systems XIV*, 347:223, 2005.
[108] Norman Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. No Starch Press, San Francisco, 2011.
[109] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2000 and Excel XP. *Computational Statistics & Data Analysis*, 40:713–721, 2002.
[110] B.D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2007. *Computational Statistics & Data Analysis*, 52:4570–4578, 2008.
[111] Eric A Meyer. *CSS Pocket Reference*. O'Reilly Media, Inc., Sebastopol, CA, 2004.
[112] Patrick Meyer and Sebastien Bigaret. RXMCDA. http://cran.r-project.org/package=RXMCDA, 2012. *R* package version 1.4.2.
[113] Paul Murrell. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL, 2005.
[114] Paul Murrell. gridBase: Integration of base and grid graphics. http://www.stat.auckland.ac.nz/~paul/grid/grid.html, 2006. *R* package version 0.4.
[115] Paul Murrell. gridSVG: Export grid graphics as *SVG*. http://www.stat.auckland.ac.nz/~paul/R/gridSVG/gridSVG_0.5-10.tar.gz, 2010. *R* package version 0.5.
[116] Paul Murrell. grid: The grid graphics package. http://cran.r-project.org/package=grid, 2011. *R* package version 2.16.0.
[117] National Aeronautics and Space Administration. Moderate resolution imaging spectroradiometer: MODIS Website. http://modis.gsfc.nasa.gov/, 2012.
[118] National Center for Biotechnology Information. Entrez, the life sciences search engine. http://www.ncbi.nlm.nih.gov/gquery, 2011.
[119] National Center for Integrative Biomedical Informatics. Michigan molecular interactions. http:/mimi.ncibi.org, 2010.
[120] Andreas Neumann and Andre Winter. Vector-based Web cartography: Enable *SVG*. http://www.carto.net/papers/svg/index_e.shtml, 2003.
[121] Eric Neuwirth. RExcel: Interface between *R* and Excel. http://cran.r-project.org/package=RExcel, 2011. *R* package version 3.2.6.
[122] Erich Neuwirth. RColorBrewer: ColorBrewer palettes. http://CRAN.R-project.org/package=RColorBrewer, 2011. *R* package version 1.0-5.
[123] Tom Ngo. Office Open *XML* overview. http://www.ecma-international.org/news/TC45_current_work/OpenXMLWhitePaper.pdf, 2005.
[124] Michael Nielsen. *Reinventing Discovery: The New Era of Networked Science*. Princeton University Press, Princeton, NJ, 2012.

[125] Deborah Nolan, Roger Peng, and Duncan Temple Lang. Enhanced dynamic documents for reproducible research. In M.F. Ochs, J.T. Casagrande, and R.V. Davuluri, editors, *Biomedical Informatics for Cancer Research*, pages 335–346. Springer-Verlag, New York, 2009.

[126] Deborah Nolan and Duncan Temple Lang. Dynamic, interactive documents for teaching statistical practice. *International Statistical Review*, 75:295–321, 2007.

[127] Deborah Nolan and Duncan Temple Lang. Learning from the statistician's lab notebook. In *Data and Context in Statistics Education: Towards an Evidence-based Society. Proceedings of the Eighth International Conference on Teaching Statistics (ICOTS8, July, 2010), Ljubljana, Slovenia.* Voorburg, 2010.

[128] Deborah Nolan and Duncan Temple Lang. RKML: Simple tools for creating *KML* displays from *R*. http://www.omegahat.org/RKML/, 2011. *R* package version 0.7.

[129] Deborah Nolan and Duncan Temple Lang. SVGAnnotation: Tools for post-processing *SVG* plots created in *R*. http://www.omegahat.org/SVGAnnotation, 2011. *R* package version 0.9.

[130] Deborah Nolan and Duncan Temple Lang. XDynDocs: Dynamic documents with *XML* and *XSL*. http://www.omegahat.org/XDynDocs, 2011. *R* package version 0.3-1.

[131] Deborah Nolan and Duncan Temple Lang. Interactive and animated scalable vector graphics and *R* data displays. *Journal of Statistical Software*, 46:1–88, 2012.

[132] Jeroen Ooms. opencpu.encode: Encodes *R* objects to a standardized *JSON* format. http://cran.r-project.org/web/packages/opencpu.encode/, 2012. *R* package version 0.22.

[133] Open Geospatial Consortium, Inc. OGC *KML* standards. http://www.opengeospatial.org/standards/kml/, 2010.

[134] Open *SSL* Project. Open *SSL*: Cryptography and *SSL*/TLS toolkit. http://openssl.org, 2011.

[135] Dave Pawson. *XSL-FO: Making XML Look Good in Print*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[136] Edzer Pebesma, Roger Bivand, Barry Rowlingson, and Virgilio Gomez-Rubio. sp: Classes and methods for spatial data. http://cran.r-project.org/web/packages/sp/, 2012. *R* package version 1.0-5.

[137] Mark Pilgrim. *HTML5: Up and Running*. O'Reilly Media, Inc., Sebastopol, CA, 2010.

[138] Tony Plate. RSVGTipsDevice: An *RSVG* graphics device with dynamic tips and hyperlink. http://cran.r-project.org/web/packages/RSVGTipsDevice/, 2011. *R* package version 1.0.

[139] David Raggett. *HTML* 4.01 specification. Worldwide Web Consortium, 1999. http://www.w3.org/TR/html401.

[140] Karthik Ram and Duncan Temple Lang. rDrop: Dropbox *R* interface. https://github.com/karthikram/rDrop/, 2012. *R* package version 0.3.

[141] Karthik Ram and Duncan Temple Lang. rProv: Provenance tracking in *R*. https://github.com/karthikram/RProvenance, 2012. *R* package version 0.2-0.

[142] Eric Raymond. *DocBook* demystification HOWTO, revision v1.3. The Linux Documentation Project, 2004. http://en.tldp.org/HOWTO/DocBook-Demystification-HOWTO/.

[143] *R* Core Team. *Condition Handling and Recovery*, 2012. http://stat.ethz.ch/R-manual/R-patched/library/base/html/conditions.html.

[144] *R* Core Team. *R Data Import/Export*, 2012. http://cran.r-project.org/doc/manuals/R-data.html.

[145] *R* Core Team. *Writing R Extensions*. Vienna, Austria, 2012. http://cran.r-project.org/doc/manuals/r-release/R-exts.html.

[146] *R* Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. http://www.r-project.org.

[147] Frank Rice. Introducing the Office (2007) Open *XML* file formats. http://msdn.microsoft.com/en-us/library/aa338205(v=office.12).aspx, 2006.

[148] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., Sebastopol, CA, 2007.

[149] Brian Ripley. RODBC: ODBC database access. http://cran.r-project.org/package=RODBC, 2011. *R* package version 1.3-3.

[150] Michel Rodriguez. *XML*::Twig: A *PERL* module for processing huge *XML* documents in tree mode. http://search.cpan.org/dist/XML-Twig/, 2012.

[151] Hans Rosling. Gapminder: World. http://www.gapminder.org/world, 2008.

[152] Barry Rowlingson. imagemap: Create *HTML* imagemaps. http://www.maths.lancs.ac.uk/Software/Imagemap/, 2004. *R* package version 0.9.

[153] Royal Society of Chemistry. ChemSpider: The free chemical database. http://www.chemspider.com/, 2012.

[154] Wendell Santos. 195 science APIs: Springer, EPA, and NCBI. http://blog.programmableweb.com/2012/03/28/195-science-apis-springer-epa-and-ncbi/, 2012.

[155] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. http://lmdvr.r-forge.r-project.org/figures/figures.html.

[156] Deepayan Sarkar. lattice: Lattice graphics. http://cran.r-project.org/web/packages/lattice/, 2011. *R* package version 0.19.

[157] Christopher Schmitt and Kyle Simpson. *HTML5 Cookbook*. O'Reilly Media, Inc., Sebastopol, CA, 2011.

[158] Marc Schwartz. WriteXLS: Cross-platform *PERL*-based *R* function to create Excel 2003 (XLS) files . http://cran.r-project.org/package=WriteXLS, 2011. *R* package version 2.3.0.

[159] Yakov Shafranovich. Common format and MIME type for comma-separated values (CSV) files. http://tools.ietf.org/html/rfc4180, 2011.

[160] Paul Shannon. RCytoscape: Interactvive viewing and exploration of graphs, connecting *R* to Cytoscape. http://rcytoscape.systemsbiology.net, 2011. *R* package version 1.8.1.

[161] John Simpson. *XPath and XPointer: Locating Content in XML Documents*. O'Reilly Media, Inc., Sebastopol, CA, 2002.

[162] Michael Smoot, Keiichiro Ono, Johannes Ruscheinski, Peng-Liang Wang, and Trey Ideker. Cytoscape 2.8: New features for data integration and network visualization. *Bioinformatics*, 27:431–432, 2011.

[163] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly Media, Inc., Sebastopol, CA, 2001.

[164] Miria Solutions. XLConnect: Manipulate Excel files from *R*. http://cran.r-project.org/package=XLConnect, 2011. *R* package version 0.2-3.

[165] Richard Stallman. GNU Emacs: An extensible, customizable text editor. http://www.gnu.org/software/emacs/, 2008.

[166] Statistical Data and Metadata Exchange Initiative. SDMX information model: UML conceptual design (version 2.0). http://www.sdmx.org/docs/2_0/SDMX_2_0SECTION_02_InformationModel.pdf, 2005.

[167] Bob Stayton. *DocBook XSL: The Complete Guide*. Sagehill Enterprises, Santa Cruz, CA, fourth edition, 2007.

[168] Daniel Steinberg. curl_easy_setopt — Set options for a curl easy handle . http://curl.haxx.se/libcurl/c/curl_easy_setopt.html, 2012.

[169] Daniel Steinberg. libcurl: The multiprotocol file transfer library. http://curl.haxx.se, 2012.

[170] Hans-Peter Suter. xlsReadWrite: Natively read and write Excel files. http://cran.r-project.org/package=xlsReadWrite, 2011. *R* package version 1.5-4.

[171] Debby Swayne, Dianne Cook, Duncan Temple Lang, and Andreas Buja. GGobi software, version 2.1. http://www.ggobi.org, 2010.

[172] Alex Szalay, Jim Gray, Ani Thakar, Bill Boroski, Roy Gai, Nolan Li, Peter Kunszt, Tanu Malik, Wil O'Mullane, Maria Nieto-Santisteban, Jordan Raddick, Chris Stoughton, and Jan van den Berg. The SDSS DR1 SkyServer: Public access to a terabyte of astronomical data. http://cas.sdss.org/dr6/en/skyserver/paper/, 2002.

[173] Duncan Temple Lang. RDCOMEvents: Responding to *DCOM* events with *R* functions . http://www.omegahat.org/RDCOMEvents/, 2005. *R* package version 0.3-1.

[174] Duncan Temple Lang. RGCCTranslationUnit: *R* interface to GCC source code information. http://www.omegahat.org/RGCCTranslationUnit, 2009. *R* package version 0.4-0.

[175] Duncan Temple Lang. RGoogleTrends: Download Google Trends data. http://www.omegahat.org/RGoogleTrends, 2009. *R* package version 0.2-1.

[176] Duncan Temple Lang. RCIndex: *R* interface to the clang parser's *C* API. http://www.omegahat.org/RCIndex, 2010. *R* package version 0.2-0.

[177] Duncan Temple Lang. RKMLDevice: Creating *R* plots in *KML*. http://www.omegahat.org/RKMLDevice/, 2010. *R* package version 0.1.

[178] Duncan Temple Lang. FlashMXML: A simple Flash graphics device for *R*. http://www.omegahat.org/FlashMXML, 2011. *R* package version 0.2-0.

[179] Duncan Temple Lang. R2GoogleMaps: Create *HTML*/*JavaScript* documents to display data using Google Maps. http://www.omegahat.org/R2GoogleMaps, 2011. *R* package version 0.2-0.

[180] Duncan Temple Lang. RAmazonDBREST: *REST*-based interface to Amazon's SimpleDB. http://www.omegahat.org/RAmazonDBREST, 2011. *R* package version 0.1-1.

[181] Duncan Temple Lang. RCSS: Facilities for reading and working with *CSS* files in *R*. http://www.omegahat.org/RCSS, 2011. *R* package version 0.2-0.

[182] Duncan Temple Lang. RDCOMClient: Accessing *DCOM* services within *R*. http://www.omegahat.org/RDCOMClient/, 2011. *R* package version 0.93-0.

[183] Duncan Temple Lang. RExcelXML: Tools for working with Excel *XML* documents. http://www.omegahat.org/RExcelXML, 2011. *R* package version 0.5-0.

[184] Duncan Temple Lang. Rffi: Interface to libffi to dynamically invoke arbitrary compiled routines at run-time without compiled bindings. http://www.omegahat.org/Rffi, 2011. *R* package version 0.3-0.

[185] Duncan Temple Lang. RJSONIO: Serialize *R* objects to *JSON* (*JavaScript* Object Notation). http://www.omegahat.org/RJSONIO, 2011. *R* package version 0.95.

[186] Duncan Temple Lang. ROOXML: Simple tools for Open Office *XML* documents. http://www.omegahat.org/ROOXML, 2011.

[187] Duncan Temple Lang. ROpenOffice: Basic reading of Open Office spreadsheets and workbooks. http://www.omegahat.org/ROpenOffice, 2011. *R* package version 0.4-1.

[188] Duncan Temple Lang. RTidyHTML: Tidy *HTML* documents. http://www.omegahat.org/RTidyHTML, 2011. *R* package version 0.2-1.

[189] Duncan Temple Lang. RXMLHelp: *XML* format and tools for *R* documentation. http://www.omegahat.org/RXMLHelp, 2011. *R* package version 0.1-0.

[190] Duncan Temple Lang. RXQuery: Bi-directional interface to an *XQuery* engine. http://www.omegahat.org/RXQuery, 2011. *R* package version 0.3-0.

[191] Duncan Temple Lang. SJava. http://cran.r-project.org/package=SJava, 2011.

[192] Duncan Temple Lang. SWinTypeLibs: Type library information reader. http://www.omegahat.org/SWinTypeLibs, 2011. *R* package version 0.6-0.

[193] Duncan Temple Lang. Sxslt: *R* extension for liblibxslt. http://www.omegahat.org/Sxslt, 2011. *R* package version 0.91-1.

[194] Duncan Temple Lang. WADL: Programmatically process Web Application Description Language documents. http://www.omegahat.org/WADL, 2011. *R* package version 0.2-0.

[195] Duncan Temple Lang. XDocTools: Tools for working with *XML* and *XSL* documents. http://www.omegahat.org/XDocTools, 2011. *R* package version 0.1-0.

[196] Duncan Temple Lang. XML: Tools for parsing and generating *XML* within *R* and *S-PLUS*. http://www.omegahat.org/RSXML, 2011. *R* package version 3.4.

[197] Duncan Temple Lang. Zillow: Simple interface to Zillow.com's house price estimate API. http://www.omegahat.org/Zillow, 2011. *R* package version 0.1-1.

[198] Duncan Temple Lang. Rcompression: In-memory decompression for GNU **zip** and bzip2 formats. http://www.omegahat.org/Rcompression, 2012. *R* package version 0.94-0.

[199] Duncan Temple Lang. RCurl: General network (HTTP, FTP, etc.) client interface for *R*. http://www.omegahat.org/RCurl, 2012. *R* package version 1.95-3.

[200] Duncan Temple Lang. RGoogleDocs: Primitive interface to Google Documents from *R*. http://www.omegahat.org/RGoogleDocs, 2012. *R* package version 0.7-0.

[201] Duncan Temple Lang. RGoogleStorage: Accessing the Google storage API from *R*. http://www.omegahat.org/RGoogleStorage, 2012. *R* package version 0.1-0.

[202] Duncan Temple Lang. RUbigraph: Interface to Ubigraph server via XML-RPC. http://www.omegahat.org/RUbigraph/, 2012. *R* package version 0.1-0.

[203] Duncan Temple Lang. RWordPress: Interface to WordPress blogs. http://www.omegahat.org/RWordPress, 2012. *R* package version 0.2-3.

[204] Duncan Temple Lang. SSOAP: Client-side SOAP access for *R*. http://www.omegahat.org/SSOAP, 2012. *R* package version 0.9-0.

[205] Duncan Temple Lang. XMLRPC: Remote procedure call (RPC) via *XML* in *R*. http://www.omegahat.org/XMLRPC, 2012. *R* package version 0.2-5.

[206] Duncan Temple Lang. XMLSchema: *R* facilities to read *XML* schema. http://www.omegahat.org/XMLSchema, 2012. *R* package version 0.7-0.

[207] Duncan Temple Lang and Gabriel Becker. RWordXML: Tools for Open Office word processing *XML* documents. http://www.omegahat.org/RWordXML, 2010. *R* package version 0.1-0.

[208] Duncan Temple Lang and Roger Peng. RAmazonS3: *R* interface to Amazon's S3 storage. http://www.omegahat.org/RAmazonS3, 2011. *R* package version 0.1-5.

[209] Duncan Temple Lang, Roger Peng, and Deborah Nolan. CodeDepends: Analysis of *R* code for reproducible research and code comprehension. http://www.omegahat.org/CodeDepends, 2011. *R* package version 0.2-1.

[210] Jeni Tennison. *Beginning XSLT 2.0: From Novice to Professional*. Apress, Berkeley, CA, 2005.

[211] Jenni Tennison. *XSLT and XPath On the Edge*. M & T Books, New York, NY, 2001.

[212] The *JSON* Schema Community. *JSON* schema: A *JSON*-based format for describing *JSON* data. http://json-schema.org/, 2011.
[213] The *New York Times* Company. The Times Developer Network: An API clearinghouse and community. http://developer.nytimes.com/docs/campaign_finance_api/campaign_finance_api_examples, 2012.
[214] Martin Theus. Interactive data visualization using Mondrian. *Journal of Statistical Software*, 7:1–9, 2002.
[215] Doug Tidwell. *XSLT*. O'Reilly Media, Inc., Sebastopol, CA, 2008.
[216] Twitter, Inc. Twitter: A real-time information network. http://twitter.com/about, 2012.
[217] United Nations Statistical Commission. Report on the thirty-ninth session. (Supplement No. 4, E/2008/24). http://unstats.un.org/unsd/statcom/doc08/DraftReport-English.pdf, 2008.
[218] Simon Urbanek. rJava: Low-level *R* to *Java* interface. http://cran.r-project.org/package=rJava, 2011. *R* package version 0.9-3.
[219] Simon Urbanek and Tobias Wichtrey. iplots: Interactive graphics for *R*. http://cran.r-project.org/web/packages/iplots/, 2011. *R* package version 1.1.
[220] US Department of Transportation. Research and innovative technology administration. http://www.rita.dot.gov/, 2011.
[221] US Food and Drug Administration. Structured product labeling resources. http://www.fda.gov/ForIndustry/DataStandards/StructuredProductLabeling/default.htm, 2012.
[222] USGS Earthquakes Hazards Program. Latest earthquakes: feeds and data. http://earthquake.usgs.gov/earthquakes/catalogs/, 2010.
[223] Eric van der Vlist. *XML Schema*. O'Reilly Media, Inc., Sebastopol, CA, 2002.
[224] Guido van Steen. dataframes2xls: Write data frames to xls files. http://cran.r-project.org/package=dataframes2xls, 2011. *R* package version 0.4.5.
[225] Wouter van Vugt. Open *XML*: The markup explained. http://openxmldeveloper.org/blog/b/openxmldeveloper/archive/2007/08/13/1970.aspx, 2007.
[226] Daniel Veillard. The *XML C* parser and toolkit of Gnome. http://www.xmlsoft.org, 2011.
[227] Todd Veldhuizen. Dynamic multilevel graph visualization. http://arxiv.org/abs/0712.1549, 2007.
[228] Todd Veldhuizen. UbiGraph: Free dynamic graph visualization software. http://ubietylab.net/ubigraph, 2007.
[229] W3Schools, Inc. *JavaScript* tutorial. http://www.w3schools.com/JS/default.asp, 2011.
[230] W3Schools, Inc. *XML* tutorial. http://www.w3schools.com/xml/default.asp, 2011.
[231] W3Schools, Inc. *XPath* tutorial. http://www.w3schools.com/XPath/default.asp, 2011.
[232] W3Schools, Inc. *DTD* tutorial. http://www.w3schools.com/dtd/default.asp, 2012.
[233] W3Schools, Inc. *JSON* tutorial. http://www.w3schools.com/json/default.asp, 2012.
[234] W3Schools.com. *SVG* tutorial. http://www.w3schools.com/svg/default.asp, 2011.

[235] Jonathan Wallace. The libjson project: A *JSON* reader and writer. http://sourceforge.net/projects/libjson/, 2012.
[236] Priscilla Walmsley. *Definitive XML Schema*. Prentice Hall PTR, Upper Saddle River, NJ, 2001.
[237] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., Sebastopol, CA, 2007.
[238] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, first edition, 1999. http://www.docbook.org/tdg5/.
[239] Gregory Warnes. gdata: Various *R* programming tools for data manipulation. http://cran.r-project.org/package=gdata, 2011. *R* package version 2.12.0.
[240] Hadley Wickham. ggplot2: An implementation of the Grammar of Graphics. http://cran.r-project.org/web/packages/ggplot2/, 2010. *R* package version 0.8.
[241] Hadley Wickham, Peter Danenberg, and Manuel Eugster. roxygen2: A Doxygen-like in-source documentation system for Rd, collation, and NAMESPACE. http://cran.r-project.org/web/packages/roxygen2/, 2011. *R* package version 2.2.2.
[242] Hadley Wickham, Deborah Swayne, and David Poole. Bay Area blues: The Effect of the housing crisis. In Toby Segaran and Jeff Hammerbacher, editors, *Beautiful Data: The Stories Behind Elegant Data Solutions*, pages 303–322. O'Reilly Media, Inc., Sebastopol, CA, 2009.
[243] Wikimedia Foundation. Wikipedia: The free encyclopedia. http://en.wikipedia.org/wiki/Main_Page, 2011.
[244] Wikipedia contributors. Mass spectrometry. http://en.wikipedia.org/wiki/Mass_spectrometry, 2012.
[245] Graham Williams, Michael Hahsler, Hemant Ishwaran, Udaya Kogalur, and Rajarshi Guha. pmml: Generate PMML for various models. http://cran.r-project.org/web/packages/pmml/index.html, 2012. *R* package version 1.2.30.
[246] Dave Winer. XML-RPC for newbies. http://scripting.com/davenet/1998/07/14/xmlRpcForNewbies.html, 1998.
[247] Dave Winer. XML-RPC specification. http://xmlrpc.scripting.com/spec.html, 1999.
[248] WordPress Community. Blog tool, publishing platform, and CMS. http://wordpress.org/, 2012.
[249] World Bank Group. WDR2011 dataset. http://databank.worldbank.org/databank/download/WDR2011Dataset.xlsx, 2011.
[250] World Bank Group. World development report 2011 on conflict, security and development. http://data.worldbank.org/data-catalog/wdr2011, 2011.
[251] Worldwide Web Consortium. Extensible Markup Language (*XML*) 1.0. http://www.w3.org/TR/REC-xml/, 2008.
[252] Carl Worth and Keith Packard. Cairo: Cross-device rendering for vector graphics. http://cworth.org/~cworth/papers/xr_ols2003/, 2003.
[253] Yihui Xie. animation: A gallery of animations in statistics and utilities to create animations. http://cran.r-project.org/web/packages/animation/, 2011. *R* package version 2.0.
[254] Yihui Xie. knitr: A general-purpose package for dynamic report generation in *R*. http://cran.r-project.org/web/packages/knitr/index.html, 2013. *R* package version 1.0.
[255] Zillow, Inc. Zillow: A free online real estate marketplace. http://www.zillow.com/, 2012.

General Index

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014

Z

R Function and Parameter Index

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014

H

I

K

L

Z

R Package Index

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014

S

W

X

Z

R Class Index

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014

Colophon

The content of this book was authored using *DocBook*, an *XML* vocabulary for technical documents. We introduced numerous extensions to the vocabulary for *R* code and plots, other languages and concepts such *XSL*, *XPath*, and shell. The content was edited primarily using the Emacs text editor, using our extended version of nxml-mode (which is available on the book's Web site).

The typesetting for the book was performed by transforming the *XML* to LaTeX using *xsltproc*. We used the *dblatex XSL* style sheets that build on top of the docbook-xsl distribution (version 1.74-0). Again, we implemented extensions to these style sheets to process our extensions to the *DocBook* vocabulary and also to customize the appearance to conform with the Springer format. The resulting LaTeX markup is processed by *pdflatex* to create the final *PDF* file.

Draft versions of the book were created by transforming the *DocBook* to *FO*, using the *DocBook XSL* style sheets, again with our own additions and customizations. The resulting *FO* documents were converted to *PDF* using `FOP`, an Apache project. We also created *HTML* versions of the book with customized *XSL* files based on *DocBook*'s *XSL* distribution.

The authoring tools for Emacs and the *XSL* stylesheets are available on the book's Web site and also in the XDynDocs package [130] available via Github and the Omegahat Web site. Similarly, tools for programmatically querying, validating and updating the document are available in the XDocTools package [195]. Code from the book and supplementary material are available at `http://rxmlwebtech.org`

D. Nolan and D. Temple Lang, *XML and Web Technologies for Data Sciences with R*, Use R!,
DOI 10.1007/978-1-4614-7900-0, © Springer Science+Business Media New York 2014