# Quantitative Corpus Linguistics with R

As in its first edition, the new edition of *Quantitative Corpus Linguistics with R* demonstrates how to process corpus-linguistic data with the open-source programming language and environment R. Geared in general towards linguists working with observational data, and particularly corpus linguists, it introduces R programming with emphasis on:

- data processing and manipulation in general;
- text processing with and without regular expressions of large bodies of textual and/or literary data, and;
- basic aspects of statistical analysis and visualization.

This book is extremely hands-on and leads the reader through dozens of small applications as well as larger case studies. Along with an array of exercise boxes and separate answer keys, the text features a didactic sequential approach in case studies by way of subsections that zoom in to every programming problem. The companion website to the book contains all relevant R code (amounting to approximately 7,000 lines of heavily commented code), most of the data sets as well as pointers to others, and a dedicated Google newsgroup. This new edition is ideal for both researchers in corpus linguistics and instructors who want to promote hands-on approaches to data in corpus linguistics courses.

**Stefan Th. Gries** is Professor of Linguistics at the University of California, Santa Barbara, USA.

# Quantitative Corpus Linguistics with R

## A Practical Introduction

### Second Edition

**Stefan Th. Gries**

Routledge
Taylor & Francis Group
NEW YORK AND LONDON

Second edition published 2017
by Routledge
711 Third Avenue, New York, NY 10017

and by Routledge
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

*Routledge is an imprint of the Taylor & Francis Group, an informa business*



First edition published by Routledge 2009

*Library of Congress Cataloging in Publication Data*
Names: Gries, Stefan Thomas, 1970- author.
Title: Quantitative corpus linguistics with R : a practical introduction / Stefan Th. Gries.
Description: Second edition. | Milton Park, Abingdon, Oxon ; New York, NY: Routledge, [2016] | Includes bibliographical references and index.
Identifiers: LCCN 2016017842| ISBN 9781138816275 (hardback) | ISBN 9781138816282 (pbk.) | ISBN 9781317597650 (epub) | ISBN 9781317597643 (mobipocket/kindle)
Subjects: LCSH: Linguistics—Statistical methods. | R (Computer program language) | Computational linguistics.
Classification: LCC P138.5 .G75 2016 | DDC 410.1/8802855362—dc23
LC record available at https://lccn.loc.gov/2016017842

ISBN: 978-1-138-81627-5 (hbk)
ISBN: 978-1-138-81628-2 (pbk)
ISBN: 978-1-315-74621-0 (ebk)

Typeset in Sabon
by Swales & Willis Ltd, Exeter, Devon, UK

Visit the companion website: www.linguistics.ucsb.edu/faculty/stgries/research/qclwr/qclwr.html

# Contents

# Figures

# Tables

# Acknowledgments

This book is dedicated to the people who have been so kind as to be part of what I might self-deprecatingly call my 'support network'; they are in alphabetical order of last names: PMC, SCD, MIF, BH, S[LW], MN, H[RW], and DS – I am *very* grateful to all you've done and all your tolerance over the last year or so! I wish to thank the team at Routledge for their interest in, and support of, a second edition of this textbook; also, I am grateful to the members of my corpus linguistics and statistics newsgroups for their questions, suggestions, and feedback on various issues and topics that have now made it into this second edition. Finally, I am grateful to many students and participants of classes, summer schools, and workshops/bootcamps where parts of this book were used.

# 1 Introduction

## 1.1 Why Another Introduction to Corpus Linguistics?

In some sense at least, this book is an introduction to corpus linguistics. If you are a little familiar with the field, this probably immediately triggers the question "Why yet another introduction to corpus linguistics?" This is a valid question because, given the upsurge of studies using corpus data in linguistics, there are also already quite a few very good introductions available. Do we really need another one? Predictably, I think the answer is still "yes" and "yes, even a second edition," and the reason is that this introduction is radically different from every other introduction to corpus linguistics out there. For example, there are a lot of things that are regularly dealt with at length in introductions to corpus linguistics that I will not talk about much:

- the history of corpus linguistics: Kaeding, Fries, early 1m word corpora, up to the contemporary giga corpora and the still lively web-as-corpus discussion;
- how to compile corpora: size, sampling, balancedness, representativity;
- how to create corpus markup and annotation: lemmatization, tagging, parsing;
- kinds and examples of corpora: synchronic vs. diachronic, annotated vs. unannotated;
- what kinds of corpus-linguistic research have been done.

That is to say, rather than telling you about the discipline of corpus linguistics – its history, its place in linguistics, its contributions to different fields, etc. – with this book, I will 'only' teach you *how to do* corpus-linguistic data processing with the programming language R (see McEnery and Hardie 2011 for an excellent recent introduction). In other words, this book presupposes that you know what you would like to explore but gives you tools to do it that go beyond what most commonly used tools can offer and, thus, hopefully also open up your minds about how to approach your corpus-linguistic questions. This is important since, to me, corpus linguistics is a method of analysis, so talking about how to do things should enjoy a high priority (see Gries 2010 and the rest of that special issue, as well as Gries 2011 for my subjective takes on this matter). Therefore, I will mostly be concerned with:

- aspects of how exactly data are retrieved from corpora to be used in linguistically informed analyses, specifically how to obtain from corpora frequency lists, dispersion information, collocation displays, concordances, etc. (see Chapter 2 for explanation and exemplification of these terms);
- aspects of data manipulation and evaluation: how to process and convert corpus data; how to save various kinds of results; how to import them into a spreadsheet program for further annotation; how to analyze results statistically; how to represent the results graphically; and how to report your results.

A second important characteristic of this book is that it only uses freely available software:

- R, the corpus linguist's all-purpose tool (cf. R Core Team 2016): a software which is a calculator, a statistics program, a (statistical) graphics program, and a programming language at the same time. The versions used in this book are R (www.r-project.org) and the freely available Microsoft R Open 3.3.1 (https://mran.revolutionanalytics.com/open, the versions for Ubuntu 16.04 LTS (or Mint 18) and Microsoft Windows 10);
- RStudio 0.99.1294 (www.rstudio.com);
- LibreOffice 5.2.0.4 (www.libreoffice.org).

The choice of these software tools, especially the decision to use R, has a number of important implications, which should be mentioned early on. As I just mentioned, R is a full-fledged multi-purpose programming language and, thus, a very powerful tool. However, this degree of power does come at a cost: In the beginning, it is undoubtedly more difficult to do things with R than with ready-made (free or commercial) concordancing software that has been written specifically for corpus-linguistic applications. For example, if you want to generate a frequency list of a corpus or a concordance of a word in a corpus with R, you must write a small *script* or a little bit of *code* in a programming language, which is the technical way of saying you write lines of text that are instructions to R. If you do not need pretty output, this script may consist of just a few lines, but it will often also be longer than that. On the other hand, if you have a ready-made concordancer, you click a few buttons (and enter a search term) to get the job done. One may therefore ask why go through the trouble of learning R? There is a variety of very good reasons for this, some of them related to corpus linguistics, some more general.

First, let me address this very argument, which is often made against using R (or other programming languages): why use a lot of time and effort to learn a programming language if you can get results from ready-made software within minutes? With regard to the time that goes into learning R, yes, there is a learning curve. However, that time may not be as long as you think: Many participants in my bootcamps and other workshops develop a first good understanding of R that allows them to begin to proceed on their own within just a few days. Plus, being able to program is an extremely useful skill for academic purposes, but also for jobs outside of academia; I would go so far as to say that learning to program is extremely useful in how it develops, or hones, a particular way of analytical and rigorous thinking that is useful in general. With regard to the time that goes into writing a script, much of that usually needs to be undertaken only once. As you will see below, once you have written your first few scripts while going through this book, you can usually reuse (parts of) them for many different tasks and corpora, and the amount of time that is required to perform a particular task becomes very similar to that of using a ready-made program. In fact, nearly all corpus-linguistic tasks in my own research are done with (somewhat adjusted) scripts or small snippets of code from this book. In addition, once you explore how to write your own functions (see Section 3.10), you can easily write your own versatile or specialized functions yourself; I will make several of those available in subsequent chapters. This way, the actual effort of generating a frequency list, a collocate display, a dispersion plot, etc. often reduces to about the time you need with a concordance program. In fact, R may even be faster than competing applications: For example, some concordance programs read in the corpus files once before they are processed and then again for performing the actual task – R requires only one pass and may, therefore, outperform some competitors in terms of processing time.

Another point related to the notion that programming knowledge is useful: The knowledge you will acquire by working through this book is quite general, and I mean that in a

good way. This is because you will not be restricted to just one particular software application (or even one version of one particular software application) and its restricted set of features. Rather, you will acquire knowledge of a programming language and regular expressions which will allow you to use many different utilities and to understand scripts in other programming languages, such as Perl or Python. (At the same time, I think R is simpler than Perl or Python, but can also interface with them via RSPerl and RSPython, respectively; see www.omegahat.org.) For example, if you ever come across scripts by other people or decide to turn to these languages yourself, you will benefit from knowing R in a way that no ready-made concordancing software would allow for. If you are already a bit familiar with corpus-linguistic work, you may now think "but why turn to R and not use Perl or Python (especially since you say Perl and Python are similar anyway and many people already use one of these languages)?" This is a good question, and I myself used Perl for corpus processing before I turned to R. However, I think I also have a good answer to why to use R instead. First, the issue of speed is much less of a problem than one may think. R is fast enough and stable enough for most applications (especially if you heed some of the advice given in Sections 3.6.3 and 3.10). Thus, if a script takes a bit of time, you can simply run it over lunch, while you are in class, or even overnight and collect the results afterwards. Second, R has other advantages. The main one is probably that, in addition text-processing capabilities, R offers a large number of ready-made functions for the statistical evaluation and graphical representation of data, which allows you to perform just about *all* corpus-linguistic tasks within only one programming environment. You can do your data processing, data retrieval, annotation, statistical evaluation, graphical representation . . . *everything* within just one environment, whereas if you wanted to do all these things in Perl or Python, you would require a huge amount of separate programming. Consider a very simple example: R has a function called `table` that generates a frequency table. To perform the same in Perl you would either have to have a small loop counting elements in an array and in a stepwise fashion increment their frequencies in a hash or, later and more cleverly, program a subroutine which you would then always call upon. While this is no problem with a one-dimensional frequency list, this is much harder with multidimensional frequency tables: Perl's arrays of arrays or hashes of arrays etc. are not for the faint-hearted, whereas R's `table` is easy to handle, and additional functions (`table`, `xtabs`, `ftable`, etc.) allow you to handle such tables very easily. I believe learning one environment can be sufficiently hard for beginners, and therefore recommend using the more comprehensive environment with the greater number of simpler functions, which to me clearly is R. And, once you have mastered the fundamentals of R and face situations in which you need maximal computational power, switching to Perl or Python in a limited number of cases will be easier for you anyway, especially since much of the programming languages' syntaxes is similar and the regular expressions used in this book are all Perl compatible. (Let me tell you, though, that in all my years using R, there were a mere two instances where I had to switch to Perl and that was only because I didn't yet know how to solve a particular problem in R.)

Second, by learning to do your analyses with a programming language, you usually have more control over what you are actually doing: Different concordance programs have different settings or different ways of handling searches that are not always obvious to the (inexperienced) user. For instance, ready-made concordance tools often have slightly different settings that specify what 'a word' is, which means you can get different results if you have different programs perform the same search on the same corpus. Yes, those settings can usually be tweaked, but that means that, actually, such a ready-made application requires the same attention to detail as R, and with a programming language all of your methodological choices are right there in the code for everyone to see and replicate.

Third, if you use a particular concordancing software, you are at the mercy of its developer. If the developers change its behavior, its results output, or its default settings, you can only hope that this is documented well and/or does not affect your results. There have been cases where even silent over-the-internet updates have changed the output of such software from one day to the next. Worse, developers might even discontinue the development of a tool altogether – and let us not even consider how sorry the state of the discipline of corpus linguistics would be if a majority of its practitioners was dependent on not even a handful of ready-made corpus tools and websites that allow you to search a corpus online. Somewhat polemically speaking, being able to enter a URL and type in a search word shouldn't make you a corpus linguist.

The fourth and maybe most important reason for learning a programming language such as R is that a programming language is a much more versatile tool than any ready-made software application. For instance, many ready-made corpus tools can only offer the functionality they aim to provide for corpora with particular formats, and then can only provide a small number of kinds of output. R, as a programming language, can handle pretty much any input and can generate pretty much any output you want – in fact, in my bootcamps, I tell participants on day 1 that I don't want to hear any questions that begin with "Can R . . . ?" because the answer is "Yes". For instance, with R you can readily use the CELEX database, CHAT files from language acquisition corpora, the very hierarchically layered annotation of XML corpora, previously generated frequency lists for corpora you no longer have access to, literature files from Project Gutenberg or similar sites, tabular corpus files such as those from the Corpus of Contemporary American English (http://corpus.byu.edu/coca) or the Corpus of Historical American English (http://corpus.byu.edu/coha), and so on and so forth. You can use files of whatever encoding, meaning that data from any language/writing system can be straightforwardly processed, and R's general data-processing capabilities are mostly only limited by your working memory and abilities (rather than, for instance, the number of rows your spreadsheet software can handle). With very few exceptions, R works identically on all three major operating systems: Linux/Unix, Windows, and Mac OS X. In a way, once you have mastered the basic mechanisms, there is basically no limit to what you can do with it, both in terms of linguistic processing and statistical evaluation.

But there are also additional important advantages in the fact that R is an open-source tool/programming language. For instance, there is a large number of functions and packages that are contributed by users all over the world. These often allow effective shortcuts that are not, or hardly, possible with ready-made applications, which you cannot tweak as you wish. Also, contrary to commercial concordance software, bug-fixes are usually available very quickly. And a final, obvious, and very down-to-earth advantage of using open-source software is of course that it comes free of charge. Any student or any department's computer lab can afford it without expensive licenses, temporally limited or functionally restricted licenses, or irritating ads and nag screens. All this makes a strong case for the choice of software made here.

## 1.2 Outline of the Book

This book has changed quite a bit from the first edition; it is now structured as follows. Chapter 2 defines the notion of a corpus and provides a brief overview of what I consider to be the most central corpus-linguistic methods, namely frequency lists, dispersion, collocations, and concordances; in addition, I briefly mention different kinds of annotation. The main change here is the addition of some discussion of the important notion of dispersion.

Chapter 3 introduces the fundamentals of R, covering a variety of functions from different domains, but the area which receives most consideration is that of text processing. There are many small changes in the code and the examples (for instance, I now introduce free-spacing), but the main differences to the first edition consist of: (1) a revision of the section on Unicode, which is now more comprehensive; (2) the addition of a new section specifically discussing how to get the most out of XML data using dedicated packages that can parse the hierarchical structure of XML documents; (3) an improved version of my `exact.matches` function; and (4) a new section on how to write your own functions for text processing and other things – this is taken up a lot in Chapter 5.

Chapter 4 is what used to be Chapter 5 in the first edition. It introduces you to some fundamental aspects of statistical thinking and testing. The questions to be covered in this chapter include: What are hypotheses? How do I check whether my results are noteworthy? How might I visualize results? Given considerations of space and focus, this chapter is informative, I hope, but still short.

The main chapter of this edition, Chapter 5, is brand new and, in a sense, brings it all together: More than 30 case studies in 27 sections illustrate various aspects of how the methods introduced in Chapters 3 and 4 can be applied to corpus data. Using a variety of different kinds of corpora, corpus-derived data, and other data, you will learn in detail how to write your own programs in R for corpus-linguistic analyses, text processing, and some statistical analysis and visualization in detailed step-by-step instructions. Every single analysis is discussed on multiple levels of abstraction and altogether more than 6,000 lines of code, nearly every one of them commented, help you delve deeply into how powerful a tool R can be for your work.

Finally, Chapter 6 is a very brief conclusion that points you to a handful of useful R packages that you might consider exploring next.

Before we begin, a few short comments on the nature of this book are necessary. This book is kind of a sister publication to my introduction to statistics for linguists (Gries 2013), and shares with it multiple characteristics. For instance, and as already mentioned, this introduction to corpus linguistics is different from every other introduction to corpus linguistics I know in how it doesn't even attempt to survey the discipline, but focuses on R programming for corpus linguists. This has two consequences. On the one hand, this book is not a book that requires much previous knowledge: It presupposes only basic (corpus-) linguistic knowledge and no mathematical or any programming knowledge.

On the other hand, this book is an attempt to teach you a lot about how to be a good corpus linguist. As a good corpus linguist, you have to combine many different methodological skills (*and* many equally important analytical skills that I will not be concerned with here). Many of these methodological skills are addressed here, such as some very basic knowledge of computers (operating systems, file types, etc.), data management, regular expressions, some elementary programming skills, some elementary knowledge of statistics, etc. What you must know, therefore, is that (1) nobody has ever learned all of this just by reading – you must *do* things – and (2) this is *not* an easy book that you can read for ten minutes at a time in bed before you fall asleep. What these two things mean is that you really *must* read this book while you are sitting at your computer so you can run the code, see what it does, and work on the examples. This is particularly important because the code file from the companion website contains more than 6,500 lines of code and a huge amount of extra commentary to help you understand the code much better than you can understand it from just reading the book; this is *particularly* relevant for Chapter 5! You will need practice to master all the concepts introduced here, but will be rewarded by acquiring skills that give you access to a variety of data and approaches you may not have considered accessible to you – at least that's what happened to me when I at

one point decided to leave behind the ready-made tools I had become used to. Undergrads in my corpus classes without prior programming experience have quickly learned to write small programs that do things better than many concordance software, and you can do the same.

In order to facilitate your learning process, there are four different ways in which I try to help you get more out of this book. First, there are small Think Breaks. These are small assignments which you should try to complete before you read on; answers to them follow immediately in the text. Second, there are exercise boxes with small assignments. Ideally, you should complete these and check your answers in the answer key before you read any further, but it is not always necessary to complete them right away to understand what follows, so you can also return to them later at your own leisure. Third, there are many boxes with recommendations for further study/exploration, which typically mention functions that you do not need for the section in which they are mentioned the first time, but many are used at a later stage (often this will be Chapter 5), which means I really encourage you to follow-up on those soon after you have encountered them. Fourth, and in addition to the above, I would like to encourage you to go to the companion website for this book at http://tinyurl.com/QuantCorpLingWithR, as well as the Google group "CorpLing with R" which I created and maintain. You need to go to the companion website to get all the files that belong with this book, but if you also become a member of the Google group:

- you can send questions about corpus linguistics with R to the list and, hopefully, get useful responses from some kind soul(s);
- post suggestions for revisions of this book there;
- inform me and the other readers of errors you find and, of course, be informed when other people or I find errata.

Thus, while this is not an easy book, I hope these aids help you to become a good corpus linguist. If you work through the whole book, you will be able to do a large number of things you could not even do with commercial concordancing software; many of the scripts you find here are taken from actual research, and are in fact simplified versions of scripts I have used myself for published papers. In addition, if you also take up the many recommendations for further exploration that are scattered throughout the book, you will probably find ever new and more efficient ways of application.

## References

Gries, Stefan Th. (2010). Corpus linguistics and theoretical linguistics: A love–hate relationship? Not necessarily . . . *International Journal of Corpus Linguistics 15*(3), 327–343.

Gries, Stefan Th. (2011). Methodological and interdisciplinary stance in corpus linguistics. In Geoffrey Barnbrook, Vander Viana, & Sonia Zyngier (Eds.), *Perspectives on corpus linguistics: Connections and controversies* (pp. 81–98). Amsterdam: John Benjamins.

Gries, Stefan Th. (2013). *Statistics for linguistics with R*. 2nd rev. and ext. ed. Berlin: De Gruyter Mouton.

McEnery, Tony, & Andrew Hardie. (2011). *Corpus linguistics: Method, theory, and practice*. Cambridge: Cambridge University Press.

R Core Team. (2016). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from www.R-project.org.

# 2 The Four Central Corpus-Linguistic Methods

> This last point leads me, with some slight trepidation, to make a comment on our field in general, an informal observation based largely on a number of papers I have read as submissions in recent months. In particular, we seem to be witnessing as well a shift in the way some linguists find and utilize data – many papers now use corpora as their primary data, and many use internet data.
>
> (Joseph 2004: 382)

In this chapter you will learn what a corpus is (plural: *corpora*) and what the four methods are to which nearly all aspects of corpus-linguistic work can be reduced in some way.

## 2.1 Corpora

Before we start to actually look at corpus linguistics, we have to clarify our terminology a little. While the actual programming tasks do not differ between them, in this book I will distinguish between a corpus, a text archive, and an example collection.

### *2.1.1 What Is a Corpus?*

In this book, the notion of a *corpus* refers to a machine-readable collection of (spoken or written) texts that were produced in a natural communicative setting, and in which the collection of texts is compiled with the intention (1) to be representative and balanced with respect to a particular linguistic language, variety, register, or genre and (2) to be analyzed linguistically. The parts of this definition need some further clarification themselves:

- "Machine-readable" refers to the fact that nowadays virtually all corpora are stored in the form of plain ASCII or Unicode text files that can be loaded, manipulated, and processed platform-independently. This does not mean, however, that corpus linguists only deal with raw text files – quite the contrary: some corpora are shipped with sophisticated retrieval software that makes it possible to look for precisely defined lexical, syntactic, or other patterns. It does mean, however, that you would have a hard time finding corpora on paper, in the form of punch cards or digitally in HTML or Microsoft Word document formats; the probably most widely used format consists of text files with a Unicode UTF-8 encoding and XML annotation.
- "Produced in a natural communicative setting" means that the texts were spoken or written for some authentic communicative purpose, but not for the purpose of putting them into a corpus. For example, many corpora consist to a large degree of

newspaper articles. These meet the criterion of having been produced in a natural setting because journalists write the article to be published in newspapers and to communicate something to their readers, not because they want to fill a linguist's corpus. Similarly, if I obtained permission to record all of a particular person's conversations in one week, then hopefully, while the person and his interlocutors usually are aware of their conversations being recorded, I will obtain authentic conversations rather than conversations produced only for the sake of my corpus.

- I use "representative [ . . . ] with respect to a particular language, variety . . . " here to refer to the fact that the different parts of the linguistic variety I am interested in should all be manifested in the corpus (at least if you want to generalize much beyond your sample, e.g., to the language in general). For example, if I was interested in phonological reduction patterns of speech of adolescent Californians and recorded only parts of their conversations with several people from their peer group, my corpus would not be representative in the above sense because it would not reflect the fact that some sizable proportion of the speech of adolescent Californians may also consist of dialogs with a parent, a teacher, etc., which would therefore also have to be included.
- I use "balanced with respect to a particular linguistic language, variety . . . " to mean that not only should all parts of which a variety consists be sampled into the corpus, but also that the proportion with which a particular part is represented in a corpus should reflect the proportion the part makes up in this variety and/or the importance of the part in this variety (at least if you want to generalize much beyond your sample, e.g., to the language in general). For example, if I know that dialogs make up 65 percent of the speech of adolescent Californians, approximately 65 percent of my corpus should consist of dialog recordings. This example already shows that this criterion is more of a theoretical ideal: How would one even measure the proportion that dialogs make-up of the speech of adolescent Californians? We can only record a tiny sample of all adolescent Californians, and how would we measure the proportion of dialogs? In terms of time? In terms of sentences? In terms of words? And how would we measure the importance of a particular linguistic variety? The implicit assumption that conversational speech is somehow the primary object of interest in linguistics also prevails in corpus linguistics, which is why corpora often aim at including as much spoken language as possible, but on the other hand a single newspaper headline read by millions of people may have a much larger influence on every reader's linguistic system than 20 hours of dialog. In sum, balanced corpora are a theoretical ideal corpus compilers constantly bear in mind, but the ultimate and exact way of compiling a balanced corpus has remained mysterious so far.

It is useful to point out, however, that the above definition of a corpus is perhaps the prototype, which implies that there are many other corpora that differ from the prototype and other kinds of corpora along a variety of dimensions. For instance, the TIMIT Acoustic-Phonetic Continuous Speech Corpus is made up of audio recordings of 630 speakers of eight major dialects of American English, where each speaker read phonetically rich sentences, a setting which is not exactly a natural communicative setting. Or consider the DCIEM Map Task Corpus, which consists of unscripted dialogs in which one interlocutor describes a route on a map to the other after both interlocutors were subjected to 60 hours of sleep deprivation and one of three drug treatments – again, hardly a normal situation. Even a genre as widely used as newspaper text – journalese – is not necessarily close to being a prototypical corpus, given how newspaper writing is created much more deliberately and consciously than many other texts – plus they often come with linguistically arbitrary restrictions regarding their length, are often not

written by a single person, and are heavily edited, etc. Thus, the notion of corpus is really a rather diverse one.

Many people would prefer to consider newspaper data not corpora, but text archives. Those would be databases of texts which

- may not have been produced in a natural setting;
- have often not been compiled for the purposes of linguistic analysis; and
- have often not been intended to be representative and/or balanced with respect to a particular linguistic variety or speech community.

As the above discussion already indicated, however, the distinction between corpora and text archives is often blurred. It is theoretically easy to make, but in practice often not adhered to very strictly and, again, has very few implications for the kinds of (R) programming they require. For example, if a publisher of a popular computing periodical makes all the issues of the previous year available on their website, then the first criterion is met, but not the last three. However, because of their availability and size, many corpus linguists use them as resources, and as long as one bears their limitations in mind in terms of representativity etc., there is little reason not to.

Finally, an *example collection* is just what the name says it is – a collection of examples that, typically, the person who compiled the examples came across and noted down. For example, much psycholinguistic research in the 1970s was based on collections of speech errors compiled by the researchers themselves and/or their helpers. Occasionally, people refer to such collections as *error corpora*, but we will not use the term *corpus* for these. It is easy to see how such collections compare to corpora. On the one hand, for example, some errors – while occurring frequently in authentic speech – are more difficult to perceive than others and thus hardly ever make it into a collection. This would be an analog to the balancedness problem outlined above. On the other hand, the perception of errors is contingent on the acuity of the researcher while, with corpus research, the corpus compilation would not be contingent on a particular person's perceptual skills. Finally, because of the scarcity of speech errors, usually all speech errors perceived (in a particular amount of time) are included into the corpus, whereas, at least usually and ideally, corpus compilers are more picky and select the material to be included with an eye to the criteria of representativity and balancedness outlined above.[1] Be that as it may, if only for the sake of terminological clarity, it is useful to distinguish the notions of corpora, text archives, and example collections.

### *2.1.2 What Kinds of Corpora Are There?*

Corpora differ in a variety of ways. There are a few distinctions you should be familiar with if only to be able to find the right corpus for what you want to investigate. The most basic distinction is that between *general corpora* and *specific corpora*. The former intend to be representative and balanced for a language as a whole – within the above-mentioned limits, that is – while the latter are by design restricted to a particular variety, register, genre, etc.

Another important distinction is that between *raw corpora* and *annotated corpora*. Raw corpora consist of files only containing the corpus material (see (1) in the example below), while annotated corpora in addition also contain additional information. Annotated corpora are very often annotated according to the standards of the Text Encoding Initiative (TEI, www.tei-c.org/index.xml) or the Corpus Encoding Standard (CES, www.cs.vassar.edu/CES), and have two parts. The first part is called the *header*, which provides information that is typically characterized as *markup*. This is information about (1) the text itself, e.g., where the corpus data come from, which language is represented in the file,

which (part of a) newspaper or book has been included, who recorded whom, where, and when, who has the copyright, what annotation comes with the file; and information about (2) its formatting, printing, processing, etc. Markup refers to objectively codable information – the fact that there is a paragraph in a text or that a particular speaker is female can typically be made without doubt; this is different from *annotation*, which is usually specifically linguistic information – e.g., part-of-speech (POS) tagging, semantic information, pragmatic information, etc. – and which is less objective (for instance, because linguists may disagree about POS tags for specific words). This information helps users to quickly determine, e.g., whether a particular file is part of the register one wishes to investigate or not.

The second part is called the *body* and contains the corpus data proper – i.e., what people actually said or wrote – as well as linguistic information that is usually based on some linguistic theory: Parts of speech or syntactic patterns, for example, can be matters of debate. In what follows I will briefly (and non-exhaustively!) discuss and exemplify a few common annotation schemes (see Wynne 2005; McEnery, Xiao, & Tono 2006: A.3 and A.4; Beal, Corrigan, & Hermann 2007a, 2007b; Gries & Newman 2013 for more discussion).

First, a corpus may be *lemmatized* such that each word in the corpus is followed (or preceded) by its lemma, i.e., the form under which you would look it up in a dictionary (see (2)). A corpus may have so-called *part-of speech tags* so that each word in the corpus is followed by an abbreviation giving the word's POS and sometimes also some morphological information (see (3)). A corpus may also be *phonologically annotated* (see (4)). Then, a corpus may be *syntactically parsed*, i.e., contain information about the syntactic structures of the text/utterances (see (5)). Finally, and as a last example, a corpus may contain several different annotations on different lines (or *tiers*) at the same time, a format especially common in language acquisition corpora (see (6)).

(1) I did get a postcard from him.
(2) I_I did_do get_get a_a postcard_postcard from_from him_he._punct
(3) I<PersPron> did<VerbPast> get<VerbInf> a<Det> postcard<NounSing> from<Prep> him<PersPron>.<punct>
(4) [@:] ·I ·^did ·get ·a ·!p\ostcard ·fr/om ·him# ·– ·–
(5)
```
<Subject, ·NP>
    I<PersPron>
<Predicate, ·VP>
    did<Verb>
    get<Verb>
    <DirObject, ·NP>
        a<Det>
        postcard<NounSing>
    <Adverbial, ·PP>
        from<Prep>
        him<PersPron>.
```
(6)
```
*CHI:    I did get a postcard from him
%mor:    pro|I ·v|do&PAST ·v|get ·det|a ·n|postcard ·prep|from ·
         pro|him ·.
%lex:    get
%syn:    trans
```

Other annotation includes that with regard to semantic characteristics, stylistic aspects, anaphoric relations (co-reference annotation), etc. Nowadays, most corpora come in the

form of XML files, and we will explore many examples involving XML annotation in the chapters to come. As is probably obvious from the above, annotation can sometimes be done completely automatically (possibly with human error-checking), semi-automatically, or must be done completely manually. POS tagging, the probably most frequent kind of annotation, is usually done automatically, and for English taggers are claimed to achieve accuracy rates of 97 percent – a number that I sometimes find hard to believe when I look at corpora, but that is a different story.

Then, there is a difference between *diachronic corpora* and *synchronic corpora*. The former aim at representing how a language/variety changes over time, while the latter provide, so to speak, a snapshot of a language/variety at one particular point in time. Yet another distinction is that between *monolingual corpora* and *parallel corpora*. As you might already guess from the names, the former have been compiled to provide information about one particular language/variety, whereas the latter ideally provide the same text in several different languages. Examples include translations from EU Parliament debates into the 23 languages of the European Union, or the Canadian Hansard corpus, containing Canadian Parliament debates in English and French. Again, ideally, a parallel corpus does not just have the translations in different languages, but has the translations sentence-aligned, such that for every sentence in language $L_1$, you can automatically retrieve its translation in the languages $L_2$ to $L_n$.

The next distinction to be mentioned here is that of *static corpora* vs. *dynamic/monitor corpora*. Static corpora have a fixed size (e.g., the Brown corpus, the LOB corpus, the British National Corpus), whereas dynamic corpora do not since they may be constantly extended with new material (e.g., the Bank of English).

The final distinction I would like to mention at least briefly involves the encoding of the corpus files. Given especially the predominance of work on English in corpus linguistics, until rather recently many corpora came in the so-called ASCII (American Standard Code for Information Interchange) character encoding, an encoding scheme that encodes $2^7 = 128$ characters as numbers and that is largely based on the Western alphabet. With these characters, special characters that were not part of the ASCII character inventory were often paraphrased, e.g., "é" was paraphrased as "é". However, the number of corpora for many more languages has been increasing steadily, and given the large number of characters that writing systems such as Chinese have, this is not a practical approach. As such, language-specific character encodings were developed (e.g., ISO 8859-1 for Western European Languages vs. ISO 2022 for Chinese/Japanese/Korean languages). However, in the interest of overcoming compatibility problems that arose due to how different languages used different character encodings, the field of corpus linguistics has been moving towards using only one unified (i.e., not language-specific) multilingual character encoding in the form of Unicode (most notably UTF-8). This development is in tandem with the move toward XML corpus annotation and, more generally, UTF-8 becoming the most widely used character encoding on the internet.

Now that you know a bit about the kinds of corpora that exist, there is one other really important point to be made. While we will see below that corpus linguistics has a lot to offer to the analyst, it is worth pointing out that, strictly speaking at least, the only thing corpora can provide is information on frequencies. Put differently, there is no meaning in corpora, and no functions, only:

- frequencies of occurrence of items – i.e., how often do morphemes, words, grammatical patterns, etc. occur in (parts of) a corpus?; and
- frequencies of co-occurrence of items – i.e., how often do morphemes occur with particular words? How often do particular words occur in a certain grammatical construction? etc.

It is up to the researcher to interpret these frequencies of occurrence and co-occurrence in meaningful or functional terms. The assumption underlying basically all corpus-based analyses, however, is that formal differences reflect functional differences: Different frequencies of (co-)occurrences of formal elements are supposed to reflect functional regularities, where *functional* is understood here in a very broad sense as anything – be it semantic, discourse-pragmatic, etc. – that is intended to perform a particular communicative function. On a very general level, the frequency information a corpus offers is exploited in four different ways, which will be the subject of this chapter: frequency lists (Section 2.2), dispersion (Section 2.3), lexical co-occurrence lists/collocations (Section 2.4), and concordances (Section 2.5).

## 2.2 Frequency Lists

The most basic corpus-linguistic tool is the frequency list. You generate a frequency list when you want to know how often something – usually words – occur in a corpus. Thus, a frequency list of a corpus is usually a two-column table with all words occurring in the corpus in one column and the frequency with which they occur in the corpus in the other column. Since the notion of *word* is a little ambiguous here, it is useful to introduce a common distinction between (word) type and (word) token. The string "the word and the phrase" contains five (word) *tokens* ("the", "word", "and", "the", and "phrase"), but only four (word) *types* ("the", "word", "and", and "phrase"), of which one ("the") occurs twice. In this parlance, a frequency list lists the types in one column and their token frequencies in the other; often you will find the expression *type frequency* referring to the number of different types attested in a corpus (or in a 'slot' such as a syntactically defined slot in a grammatical construction).

Typically, one out of three different sorting styles is used: frequency order (ascending or, more typically, descending; see the left panel of Table 2.1), alphabetical (ascending or descending), and occurrence (each word occurs in a position reflecting its first occurrence in the corpus).

Apart from this simple form in the leftmost panel, there are other varieties of frequency lists that are sometimes found. First, a frequency list may provide the frequencies of all words together with the words with their letters reversed. This may not seem particularly useful at first, but even a brief look at the second panel of Table 2.1 clarifies that this kind of display can sometimes be very helpful because it groups together words that share a particular suffix – here the adverb marker *-ly*. Second, a frequency list may not list each individual word token and their frequencies, but so-called *n*-grams, i.e., sequences of

*Table 2.1* Examples of differently ordered frequency lists

| *Words* | *Freq.* | *Words* | *Freq.* | *Bigrams* | *Freq.* | *Words* | *Tags* | *Freq.* |
|---|---|---|---|---|---|---|---|---|
| the | 62,580 | yllufdaerd | 80 | of the | 4,892 | the | AT0 | 6,069 |
| of | 35,958 | yllufecaep | 1 | in the | 3,006 | of | PRF | 4,106 |
| and | 27,789 | yllufecarg | 5 | to the | 1,751 | a | AT0 | 2,823 |
| to | 25,600 | yllufecruoser | 8 | on the | 1,228 | and | CJC | 2,602 |
| a | 21,843 | yllufeelg | 1 | and the | 1,114 | in | PRP | 2,449 |
| in | 19,446 | yllufeow | 1 | for the | 906 | to | TO0 | 1,678 |
| that | 10,296 | ylluf | 2 | at the | 832 | is | VBZ | 1,589 |
| is | 9,938 | yllufepoh | 8 | to be | 799 | to | PRP | 1,135 |
| was | 9,740 | ylluferac | 87 | with the | 783 | for | PRP | 916 |
| for | 8,799 | yllufesoprup | 1 | from the | 720 | be | VBI | 874 |

*n* words such as bigrams (where *n* = 2, i.e., word pairs) or trigrams (where *n* = 3, i.e., word triples) and their frequencies; see the third panel of Table 2.1 for an example involving bigrams. Finally, frequency lists may not just be of words or *n*-grams, but also of combinations of units such as words and their POS tags: In the rightmost panel of Table 2.1 you can see that two differently tagged versions of "to" are distinguished – one tagged as the infinitive marker (TO0), the other as a preposition (PRP). With the right kind of annotation, similar lists could be generated of words with particular senses or other interesting combinations of things; in the chapters that follow, you will learn to create all of the lists exemplified in Table 2.1.

Frequency lists are sometimes more problematic than they seem because they presuppose that the linguist (and/or his computer program) has a definition of what a word is and that this definition is shared by other linguists (and their computer programs). This need not be the case, however, as the following exercise will demonstrate.

**Exercise 2.1 What Is a Word, or What Do You Think Your Computer Thinks a Word Is?**

1 Write up a plain English definition of how you would 'tell a computer program' what a word is. When you are finished, do the next exercise.
2 How does your definition handle the expressions *better-suited* and *ill-defined*? How does it handle *armchair-linguist* and *armchair linguist* and *armchair-linguist's*? Or *because of* and *in spite of*? How does it handle *www.linguistics.ucsb.edu*? How does it handle *This* and *this*? Or *O.M.F.G*? How does it handle *1960* and *25-year-old*? And *favor* and *favour*? And *Peter's car*, *Peter's come home*, and *Peter's sick*? And what about позволено (Cyrillic) or 宜蘭童玩節停辦 (Mandarin Chinese) in an English text?

Bear in mind, therefore, that you need to exercise at least some caution in comparing frequency lists from different sources.[2] Another noteworthy aspect is that frequency lists are often compiled so as not to include types from a so-called stop list. For example, one can often exclude a variety of frequent function words such as *the*, *of*, *and*, etc., because these are often not particularly revealing given that they occur nearly equally frequently everywhere.[3]

For what follows below, it is useful to introduce the distinction between a lemma and a word-form: *go*, *goes*, *going*, *went*, and *gone* are all different word-forms but belong to the same lemma, namely *go*; in the remainder of this chapter I will only use *word* where the difference between *lemma* and *word-form/token* is not relevant. Computer programs normally define word-tokens as a sequence of alphabetic (or alphanumeric) characters uninterrupted by whitespace (i.e., spaces, tabs, and newlines) and allow the user to specify what to do with hyphens, apostrophes, and other special characters.

Frequency lists are useful for a variety of purposes. For example, much contemporary work in usage-based linguistics assumes that the type and token frequencies of linguistic expressions are correlated with the degrees of productivity and cognitive entrenchment of these expressions, and studies such as, e.g., Bybee and Scheibman (1999) and Jurafsky, Bell, Gregory, and Raymond (2000) related frequencies of expressions to their readiness to undergo processes of phonological reduction and grammaticalization. For example, Bybee and Scheibman (1999) show that the vowel in *don't* is more likely to be reduced

to schwa in frequent expressions such as *I don't know* (as opposed to, say, *They don't integrate easily*). In psycholinguistics, models of language production have long incorporated frequency effects, but there is now a growing body of work suggesting that information on percentages, conditional probabilities, surprisal, etc. of all kinds is represented in the linguistic systems of speakers and, thus, plays a primary role on all levels of linguistic analysis. On the more practical side of things, word frequency lists are useful for choosing experimental stimuli correctly – for example, the logs of frequencies of words are correlated with reaction times to these words (in priming or naming studies), which often makes it necessary to control for such effects in order not to bias one's results. Also, you may want to do a study on tip-of-the-tongue states and, thus, make sure the words you use are sufficiently infrequent.

A much more applied example, which we will also be concerned with below, involves the identification of words that are (significantly) overrepresented in one (target) corpus as compared to another (typically larger and more balanced reference) corpus. This can be used to identify the words that are most characteristic of a particular domain: A word that is about equally frequent in a particular target corpus and a reference corpus is probably not very revealing in terms of what the smaller corpus is about. However, if a target corpus contains the words *economic*, *financial*, *shares*, *stocks* much more often than a balanced reference corpus, guessing the topics covered in the smaller corpus is straightforward. This kind of approach can be used to generate lists of important, or *key*, vocabulary for language learners.

In the domain of natural language processing or computational linguistics, the frequency of items is relevant to, among other things, speech recognition. For example, imagine a computer gets ambiguous acoustic input in a noisy environment and tries to recognize which word is manifested in the input. If the computer cannot identify the input straightforwardly, one (simplistic) strategy would be for it to assume that the word it hears is the most frequent one of all those that are compatible with the acoustic input. Another area of interest is, for example, spelling error correction, where frequency lists can be useful in two ways: First, for the computer to recognize that a string is perhaps a typo because it neither occurs in a gold-standard frequency list of words of the input language nor in, say, a list of named entities; second, for the computer to rank suggestions for correction such that the computer first determines a set of words that are sufficiently similar to the user's input and then ranks them according to their similarity to the input and their frequency. From a methodological perspective, frequency lists are useful for computing many co-occurrence statistics. Finally, frequency lists may reflect sociocultural differences.

## 2.3 Dispersion Information

A characteristic of words that is often, but not necessarily, correlated with their frequencies is their dispersion: words that are very similar in terms of their overall frequency in a corpus may be very differently distributed in a corpus. For instance, Leech, Rayson, and Wilson (2001) showed that the words *HIV*, *keeper*, and *lively* are about equally frequent in the 100 million-word British National Corpus (about 16 occurrences per million words), but are quite differently distributed in the corpus: *keeper* and *lively* occur in 97 out of 100 equally sized corpus parts, whereas *HIV* only occurs in 62 of the same 100 parts. This indicates that it is used more clumpily: usage is less wide, but where it is used it is frequent. Crucially, while very frequent words will usually be dispersed quite evenly in a corpus and while very rare words will usually be dispersed quite unevenly in a corpus, words in the middle range of frequencies can behave like *HIV* and *lively*: being equally

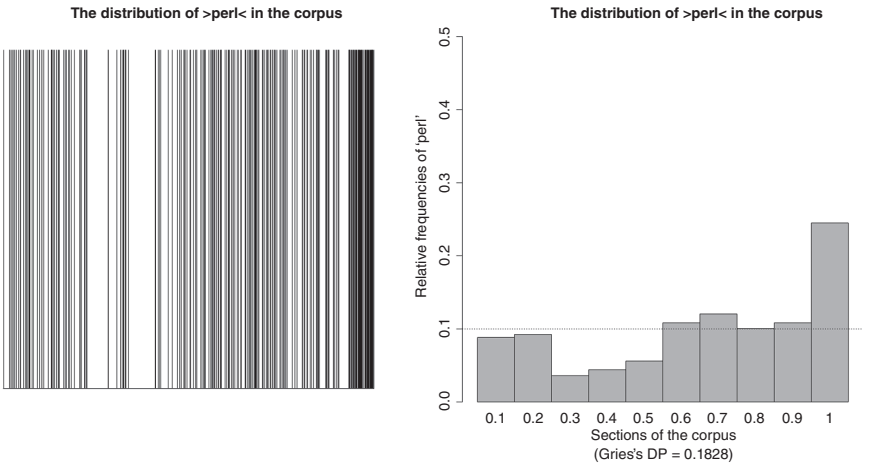


*Figure 2.1* Representational format of corpus files and data frames.

frequent but *very* differently distributed. Thus, it is very important to not rely only on the frequency of words, but also to consider how (un)evenly words are dispersed (see Gries 2008, 2010; Liffijt & Gries 2012).

Dispersion is often explored visually, but can also be quantified. The left panel of Figure 2.1 shows a dispersion plot, in which the *x*-axis represents a corpus from beginning to end – here the 'corpus' is just an older Wikipedia entry on the programming language Perl – and every occurrence of the string "perl" is represented by a vertical line. It is easy to see that there are very many occurrences of that string at the end (i.e., in the reference section), but that there are also longer stretches where it does not occur at all. The right panel represents the same data with a more coarse-grained resolution, namely when the 'corpus' has been binned, i.e., split into equally sized parts (here, ten): We can see that "perl" occurs less often in the part 30–50 percent of the file, and particularly frequently in the last 10 percent of the corpus. Also, the subtext under the *x*-axis label quantifies the dispersion using an index I proposed in Gries (2008) and validated in Gries (2010), as well as Biber and Reppen (forthcoming): The closer the value of *DP* is to 0, the more evenly the expression is distributed; the closer it is to 1, the more unevenly the expression is distributed. You will later learn how to create Figure 2.1 and compute *DP* yourself.

## 2.4 Lexical Co-occurrence: Collocations

One of the most central methodological concepts in corpus linguistics is that of co-occurrence. Corpus linguists have basically been concerned with three different kinds of co-occurrence phenomena:

1 *Collocation*: the probabilistic co-occurrence of word-forms such as *different from* vs. *different to* vs. *different than*, or the absolute frozenness of expressions such as *kith and kin* or *by and large*.
2 *Colligation*: the co-occurrence of words with grammatical phenomena such as parts of speech, grammatical relations, or 'definiteness', such as the preference of *consequence* to occur as a complement (but not an adverbial) and with indefinite articles.

3 (Grammar) *patterns* or *collostructions*: the co-occurrence of words/lemmas with morphosyntactic patterns or constructions (in the construction grammar sense) such as the ditransitive construction or the cleft construction such as the preference of *to hem* to occur in the passive; or the association of the ditransitive to forms of the verb *to give*.

In this section we will restrict ourselves to collocations because they are a natural extension of frequency lists; later in the book, we will of course deal with other kinds of co-occurrence, too. Collocations are co-occurrences of words, which are then referred to as *collocates*. Often, one uses the letters *L* and *R* (for 'left' and 'right') together with a number indicating the position of one collocate with respect to the main/node word to talk about collocations. For example, if you call up a concordance of the word *difference*, then you will most certainly find that the most frequent L1 collocate is *the* while the most frequent R1 collocate is *between*. Thus, a collocate display for a word tells you which other words occur at different positions around the target word and how frequently. In other words, a collocate display is a list of frequency lists for particular positions around a word. To look at a simple example with two adjectives, consider the two adjectives *alphabetic* and *alphabetical*: Table 2.2 is a collocate display of *alphabetic*; the strings in angular brackets are word class tags (see www.natcorp.ox.ac.uk/docs/URG/posguide.html). Table 2.3 is a collocate display of *alphabetical*.

Note that all this means that a collocate display is read *vertically*: Row 1 of Table 2.2 does not tell you anything about a string *of alphabetic literacy* – it tells you that *of* is the most frequent word at L1 (occurring there eight times) and that *literacy* is the most frequent word at R1 (with seven occurrences).

**THINK**

**BREAK**

Now, if we compare the R1 collocates of *alphabetic* and *alphabetical*, can you already discern a tendency of the specific semantic foci of the two adjectives?

The difference between the two adjectives can probably be paraphrased easiest by stating what the opposites of the two adjectives are. My suggestion would be that the opposite

*Table 2.2* A collocate display of *alphabetic* based on the BNC

| *Word at L1* | *Freq. L1* | *Node word* | *Freq. node* | *Word at R1* | *Freq. R1* |
|---|---|---|---|---|---|
| <w prf>of | 8 | <w aj0>alphabetic | 42 | <w nn1>literacy | 7 |
| <w at0>the | 6 | | | <w nn1>writing | 5 |
| <w at0>an | 5 | | | <w nn1>order | 3 |
| <w prp>in | 2 | | | <w nn1>character | 3 |
| <w prp>such as | 2 | | | <w cjc>and | 2 |
| <w dps>our | 2 | | | <w nn1>system | 2 |
| <w cjs>when | 2 | | | <w nn2>characters | 2 |
| <w aj0> widespread | 1 | | | <w nn1>culture | 2 |
| <w nn2>systems | 1 | | | <w prp>in | 1 |
| <w aj0>varying | 1 | | | <c pun>. | 1 |

*Table 2.3* A collocate display of *alphabetical* based on the BNC

| *Word at L1* | *Freq. L1* | *Node word* | *Freq. node* | *Word at R1* | *Freq. R1* |
|---|---|---|---|---|---|
| <w prp>in | 77 | <w aj0>alphabetical | 234 | <w nn1>order | 89 |
| <w at0>an | 36 | | | <w nn1>index | 15 |
| <w at0>the | 23 | | | <w nn1>list | 13 |
| <w prf>of | 6 | | | <w nn1>indexing | 12 |
| <w cjc>and | 6 | | | <w nn1>subject | 12 |
| <c pun>. | 6 | | | <w nn1>sequence | 11 |
| <c pun>, | 6 | | | <w nn1>listing | 9 |
| <w aj0>ascending | 5 | | | <w nn1>guest | 6 |
| <w cjc>or | 5 | | | <w cjc>and | 5 |
| <w aj0>strict | 4 | | | <w nn1>description | 2 |

of *alphabetic* is *numeric*, whereas the opposite of *alphabetical* is *unordered*, but a more refined look at the data may reveal a more precise picture.

Collocate displays are an important tool within semantics and lexicography (cf. Sinclair 1987; Stubbs 1995) as well as language teaching (cf. Lewis 2000); the phenomenon of near synonymy is a particularly fruitful area of application. Consider, for example, the English adjectives *fast*, *quick*, *rapid*, and *swift* (as you will later in Section 5.3.5). While we would be hard-pressed to immediately explain the differences between these semantically extremely similar expressions, inspecting the position R1 in collocate displays for these adjectives might give us a very good clue as to the nouns these adjectives usually modify attributively and invite semantic or other distributional regularities.

Another area of application is what has been referred to as *semantic prosody*, i.e., the fact that collocates of some word *w* may imbue *w* with a particular semantic aura even though this aura is not part of the semantics of *w* proper. One of the standard textbook examples is the English verb *to cause*. As you probably sense intuitively, *to cause* primarily, though not exclusively, collocates with negative things (*problem*, *damage*, *harm*, *havoc*, *distress*, *inconvenience*, etc.), although causing as such need not be negative at all. This is not only a theoretically interesting datum, it also has implications for, say, research on irony and foreign language teaching since, for example, if a foreign language learner uses a word *w* without being aware of *w*'s semantic prosody, this may result in comical situations or, more seriously, communicating unwanted implications.

## 2.5 (Lexico-)Grammatical Co-occurrence: Concordances

However useful collocate displays are, for many kinds of analysis they are still not optimal. On the one hand, it is obvious that collocate displays usually provide information on lexical co-occurrence, but the number of grammatical features that is amenable to an investigation by means of collocates alone is limited. On the other hand, even the investigation of lexical co-occurrence by means of collocate displays can be problematic. If you investigate near-synonymous adjectives such as *big*, *great*, and *large* (or *deadly*, *fatal*, and *lethal*) by looking at R1, you reduce both the precision and the recall of your results:

- *Precision* is defined as the quotient of the number of accurate matches returned by your search divided by the number of all matches returned by your search. The collocate approach may reduce precision because the R1 collocate of *big* in *big shiny*

*tricorder* is *shiny* rather than *tricorder*, and the inclusion of *shiny*, while of course accurate, may not tell you as much about the semantics of *big* as *tricorder*.[4]

- *Recall* is defined as the number of accurate matches returned by your search divided by the number of all possible accurate matches in the data. The R1 collocate approach may reduce recall because, as we have seen above, you miss *tricorder* in *big shiny tricorder*.

The final method to be introduced here addresses this problem, though at a cost. This method, probably the most widespread corpus-linguistic tool, is the *concordance*. You generate a concordance if you want to know in which (larger and more precise) contexts a particular word is used. Thus, a concordance of a word *w* is a display of every occurrence of *w* together with a user-specified context; it is often referred to as KWIC (Key Word In Context) display.[5] This user-specified context is often either the whole sentence in which the word in question occurs (usually with some highlighting or bracketing; see the use of "[[" and "]]" in Table 2.4) or the word in question in a central column together with a user-specified number of words or characters to its left and its right (see Table 2.5 for an example). Especially the display in Table 2.5 is useful because it can be imported into spreadsheet software and then be sorted in various ways, such as according to the words at R1. Obviously, unlike collocate displays, concordances are read horizontally just like normal (parts of) sentences, which is what they in fact show.

While the concordance displays in Table 2.4 and Table 2.5 are not always easily exploitable, they are maximally comprehensive: you can look at your search word and every possible linguistic element and how (frequently) it co-occurs with it (not just words as in the collocate display), but the price you have to pay is that you can usually not extract this information semi-automatically as in the collocate display. In other words,

*Table 2.4* An example display of a concordance of *before* and *after* (sentence display)

| # | *Match* |
|---|---|
| 1 | at that time and erm, we had a lot of German shutters and cameras in museum [[before]] September the third on September the fourth when I got to work they were all out. |
| 2 | It will be easy enough to bleach them with some Milk of Magnesia the night [[before]] he comes home. |
| 3 | And as, as we said [[before]], erm, many of the, erm people who lived in the poorer parts of erm the country, whether in urban or rural En erm England didn't really know about basic nutrition and and health I mean you just |
| 4 | They should be kept in an airtight jar, and rinsed in methylated spirits [[before]] wearing. |
| 5 | Yes if you looked [[after]] a child. |
| 6 | And erm some immediately post-war recipe books and I'm sure you know if you'd like to look at them [[after]] I've finished talking you might even remember some of the er the er My wife still uses the. |
| 7 | This was covered by a biscuit tin lid on top of which was a kettle filled with water for an [[after]] dinner cup of tea and the washing up. |
| 8 | [[After]] the blitz on London in September nineteen forty, the government introduced a scheme whereby payments for damage to the furniture of persons earning less than four hundred pounds a year would be made, up to one hundred percent of th |
| 9 | There was a campaign [[after]] the war to bring back Tottenham puddings they had somebody in Harlow who was Well what was, what, what was Tottenham pudding then. |
| 10 | [[After]] Japan's entry into the war all imports of rubber from the far east were suspended. |

*Table 2.5* An example display of a concordance of *before* and *after* (tabular)

| *L1* | *Node* | *R1* | *R2* |
|---|---|---|---|
| museum | before | September | the |
| night | before | he | comes |
| said | before | erm | many |
| spirits | before | wearing | |
| looked | after | a | child |
| them | after | I | 've |
| an | after | dinner | cup |
| | After | the | blitz |
| campaign | after | the | war |
| | After | Japan | 's |

concordances usually need manual analysis and annotation: In the example of *big shiny tricorder* from above, for example, if your corpus is not syntactically annotated, you must read the concordance line(s) yourself to determine that *big* modifies *tricorder*. Given the maximal explicitness, the utility of concordances is only limited by this latter fact: Inspecting a three-million-line concordance of some word is simply not feasible, and in those cases one normally resorts to statistical techniques to filter out the relevant patterns and/or only investigates a sample of the concordance, hoping it will reflect the overall tendency well enough.

## Notes

1 It is only fair to mention, however, that (1) error collections have proven extremely useful in spite of what, from a strict corpus linguistic perspective, may be considered shortcomings, and that (2) compilers of corpora of lesser-spoken languages such as typologists investigating languages with few written records suffer from just the same data scarcity problems.

2 One important characteristic of language is reflected in nearly all frequency lists. When you count how often each type occurs in a particular corpus, you usually get a skewed distribution such that

   1 a few types – usually short function words – account for the lion's share of all tokens (for example, in the Brown corpus, the ten most frequent types out of approximately all 41,000 different types (i.e., only 0.02 percent of all word types) already account for nearly 24 percent of all tokens); and
   2 most tokens occur rather infrequently (for example, 16,000 of the tokens in the Brown corpus occur only once).

   These observations are a subset of Zipf's laws, the most famous of which states that the frequency of any type is approximately proportional to its rank in a frequency list, and such a distribution is often referred to as a Zipfian distribution.

3 Even if the term *stop list* is new to you, you are probably already familiar with the concept: Search engines often omit function words that are entered into their search fields.

4 Note, however, that this may turn into an interesting finding once you find that some adjectives exhibit a tendency for such kinds of additional premodification, while others don't. Also, note in passing that in the context of regular expressions, *precision* and *recall* are sometimes referred to as *specificity* and *sensitivity*, respectively.

5 Strictly speaking, a concordance does not have to list every occurrence of a word in a corpus. Some programs output only a user-specified number of occurrences, usually either the first *n* occurrences in the corpus or *n* randomly chosen occurrences from the corpus.

## References

Beal, Joan C., Karen P. Corrigan, & Hermann L. Moisl (Eds.). (2007a). *Creating and digitizing language corpora: Synchronic databases vol. 1*. Basingstoke: Palgrave Macmillan.

Beal, Joan C., Karen P. Corrigan, & Hermann L. Moisl (Eds.). (2007b). *Creating and digitizing language corpora: Diachronic databases vol. 2*. Basingstoke: Palgrave Macmillan.

Biber, Douglas & Randi Reppen. (Forthcoming). On the (non)utility of Juilland's D to measure lexical dispersion in large corpora. *International Journal of Corpus Linguistics*.

Bybee, Joan, & Joanne Scheibman. (1999). The effect of usage on degrees of constituency: The reduction of don't in English. *Linguistics, 37*(4), 575–596.

Gries, Stefan Th. (2008). Dispersions and adjusted frequencies in corpora. *International Journal of Corpus Linguistics, 13*(4), 403–437.

Gries, Stefan Th. (2010). Dispersions and adjusted frequencies in corpora: Further explorations. In Stefan Th. Gries, Stefanie Wulff, & Mark Davies (Eds.), *Corpus linguistic applications: Current studies, new directions* (pp. 197–212). Amsterdam: Rodopi.

Gries, Stefan Th., & John Newman. (2013). Creating and using corpora. In Robert J. Podesva & Devyani Sharma (Eds.), *Research methods in linguistics* (pp. 257–287). Cambridge: Cambridge University Press.

Joseph, Brian D. (2004). On change in *Language* and change in language. *Language, 80*(3), 381–383.

Jurafsky, Daniel, Alan Bell, Michelle Gregory, & William D. Raymond. (2000). Probabilistic relations between words: Evidence from reduction in lexical production. In Joan Bybee & Paul Hopper (Eds.), *Frequency and the emergence of linguistic structure* (pp. 229–254). Amsterdam: John Benjamins.

Leech, Geoffrey, Paul Rayson, & Andrew Wilson. (2001). *Word frequencies in written and spoken English: Based on the British National Corpus*. London: Longman

Lewis, Michael (Ed.). (2000). *Teaching collocation: Further developments in the lexical approach*. Hove: Language Teaching Publications.

Lijffijt, Jefrey, & Stefan Th. Gries. (2012). Correction to "Dispersions and adjusted frequencies in corpora". *International Journal of Corpus Linguistics, 17*(1), 147–149.

McEnery, Tony, Richard Xiao, & Yukio Tono. (2006). *Corpus-based language studies: An advanced resource book*. London: Routledge.

Sinclair, John McHardy. (1987). *Looking up: An account of the COBUILD project in lexical computing*. London: Collins.

Stubbs, Michael. (1995). Collocations and semantic profiles: On the cause of the trouble with quantitative studies. *Functions of Language, 2*(1), 23–55.

Wynne, Martin (Ed.). (2005). *Developing linguistic corpora: A guide to good practice*. Oxford: Oxbow Books, http://ahds.ac.uk/linguistic-corpora.

# 3 An Introduction to R

> But writing new scripts requires programming skills that are probably beyond the capabilities of the average corpus linguist.
>
> (Meyer 2002: 114f.)

In this chapter you will learn the foundations of R that will enable you to load, process, and store data, as well as perform simple and more complex operations on text. Thus, this chapter prepares you for the statistical applications in Chapter 4, but especially the linguistic applications in Chapter 5. Let's begin with the installation of the relevant software: R and RStudio. In this book I am primarily using Microsoft R Open (MRO), which is full compatible with default R, but enhanced in a variety of ways (and all code presented here has been tested on Windows 10 and Kubuntu 16.04/Mint 18). These are the relevant steps:

1 Download MRO for your operating system from the MRAN site at https://mran.revolutionanalytics.com/download/#download; if you are not using Mac OS X, also download the MKL Math Library. Click on Next Steps and follow the instructions for your operating system to install both MRO and the Math Library; alternatively, download 'default R' from https://cran.r-project.org.
2 Download the RStudio Desktop version for your operating system from www.rstudio.com/products/rstudio/download.

That's it. You can now start and use R either by starting MRO or, better, by starting RStudio, which will start an R instance, but within a powerful integrated development environment. In this book, with one exception (on page 23), we will always use RStudio. The normal situation of working with RStudio involves that you have either created a new R script (by clicking on "*File: New File: R script*" or opened an existing R script by double-clicking on one (the recommended default) or opening it from within RStudio with "*File: Open File*". If you do that, your RStudio instance will have four panes:

1 A pane called *Untitled1* (if you just created a new empty R script) or that has the name of the R script file you double-clicked on. In this pane, which is essentially a powerful programming editor with syntax-highlighting and many other features, you can write R code that you want R to execute later. I recommend you do *all* your script writing there, *not* in the console, so that you can more easily write, edit, debug, and save your code.
2 A pane called *Console* (followed by your current working directory). This is where R runs the code you typed in the first pane and where R provides statistical/text results; note that when you have written code in the first pane, you do not have to copy it there and paste it into the console pane – any code that is highlighted in the first pane,

or even just the line that the cursor is currently in, can be executed simply by pressing CTRL + ENTER, which will take the code, "put it" into the console and run it there.

3 A pane with tabs called *Environment*, *History*, and maybe others, which lists all data structures and user-defined functions currently in R's working memory (*Environment*) and all code run in the current session (*History*). Another tab that may be shown here is the *Help* tab, where you can find documentation on R and RStudio.

4 A pane with tabs called *Files*, *Plots*, *Packages*, and maybe others. The *File* tab provides a file manager, the *Plots* tab will display all plots generated in the current session, and the *Packages* tab allows you to manage (install, load, etc.) R packages.

The layout of these panes can be customized by going to *Tools: Global Options . . . : Pane Layout*. In addition, I would like to recommend that you also (1) go to "*Tools: Global Options . . . : Code*" and tick the box "Soft-wrap R source files" and (2) go to "*Tools: Global Options . . . : General*" and untick all boxes. These are just my initial recommendations – once you are more familiar with R and RStudio, you can always change these later (as I have for my own set-up).

As a second step, I recommend that you generate a folder <_qclwr2/> somewhere in your home directory on your hard drive (for **Q**uantitative **C**orpus **L**inguistics **W**ith **R**, **2**nd edition). Then, go to the companion website at http://tinyurl.com/QuantCorpLingWithR, download all the relevant files for the second edition from there, and unzip them in the qclwr2 folder:

- <_qclwr2/_scripts>; this directory contains all relevant R code from this book: more than 6,000 lines of code with explanations, examples, exercises, and answer keys. The password you need to unzip these files is (without the double quotes) "_HinteRschinken".
- <_qclwr2/_outputfiles>; this directory contains most output files resulting from R code in this book. The password you need to unzip these files is (without the double quotes) "_KassleR".
- <_qclwr2/_inputfiles>; this directory contains most input files you require; the password you need to unzip these files is (without the double quotes) "_WuRst"; other input files you will need are the British National Corpus (BNC, now freely available from http://ota.ox.ac.uk/desc/2554; files from the CHILDES archive, namely http://childes.psy.cmu.edu/data/Eng-NA-MOR/Brown.zip). Other corpora that I use for just one assignment later each are
  - the Brown corpus (the original version from the ICAME CD-ROM, version 2, see http://clu.uni.no/icame/cd);
  - the Chinese Hong Kong files from the ICLE corpus version 2 (see www.uclouvain.be/en-277586.html); and
  - the fiction part of the Corpus of Contemporary American English and the nineteenth-century part of the Corpus of Historical American English (see http://corpus.byu.edu/full-text).

  (Even if you do not have access to those last four corpora, you will still be able to learn from the code used in the relevant case studies, plus I will actually give you practice assignments that change freely available corpora into the formats of the corpora you might not have – once you've solved those, you can still make full use of these assignments.)

Note that I am using regular slashes here in the paths to the directories because you can use those for paths in R, too, and more easily so than backslashes. Also, check out the file

for errata on the website so that you can make necessary or recommended corrections. Finally, I recommend that you set the settings of your operating system's file manager such that it displays file names *and* their extensions, not just file names.

Note also that R has much more to offer than this base installation: R is open-source software and there is a very lively community of people who have written so-called packages for R. These packages are additions to R that you can download and install (from within R) and then load into R to obtain commands (or functions, as we will later call them) that are not part of the default configuration. I would therefore suggest you do the following:

- start R – not RStudio this one time – with administrator rights: On Linux you would do this by entering `sudo R¶` at a terminal/console and provide your password; on Windows you can right-click on the R icon in your Start Menu and choose "Run as administrator";
- at the console you see, paste the first line from the code file <_qclwr2/_scripts/03-04_allcode.r> there, which will install all packages we will be using in this book.

As mentioned above, I *very strongly* recommend that you read this book while sitting at your computer with the relevant code file open in RStudio – in fact, you should go to <_qclwr2/_scripts> and double-click on <03-04_allcode.r> right now, which will also make <_qclwr2/_scripts> your current working directory in RStudio, which nearly all of the code below presupposes – so that you can follow along more easily. Ideally, you would read the book and run the code I am discussing (by pressing CTRL + ENTER whenever I discuss a line of code so you see how it is executed in RStudio and what it returns); also, you can of course add your own notes to the R code files directly (ideally always preceded by the pound/hash sign # – see below for why).

As was already mentioned in the introduction, R is an extremely versatile piece of software. It can be used as a calculator, a spreadsheet program, a database, a statistics program, a (statistical) graphics program, and a scripting programming language with sophisticated mathematical and character-processing capabilities. In fact, the range of functions R can perform is so vast that I will have to restrict my discussion to a radically reduced subset of its functions. The functions that will be covered in this book are mainly concerned with

- how to load, process, and save various kinds of data structures, with a special emphasis on handling text data from corpora for corpus-linguistic analyses; and
- how to load, process, and evaluate tabular data (for statistical/graphical analysis).

Thus, I will unfortunately have to leave aside many intriguing aspects of R. Also, for didactic and expository reasons I will sometimes not present the shortest or most elegant way of handling a particular problem, but rather present a procedure which is more adequate for one or more reasons. These reasons include the desires to

- keep the number of functions, arguments, and regular expressions manageable;
- highlight similarities between different functions; and
- allow you to recycle parts of the code.

Thus, this book is not a general introduction to R, and while I aim at enabling you to perform a multitude of tasks with R, I advise you to also consult the additional references mentioned below and the comprehensive documentation that comes with R and RStudio;

there are also many instructional videos out there, but as far as I can tell they are very heavily biased in the direction of statistical analysis rather than text processing.

Next, some notational conventions, some of which you have already seen by now. Paths/files and folders will be mentioned like this: <_qclwr2/_inputfiles/some_name.txt>. Input to R is usually given in blocks of code as below, where "·" means a space character and "¶" denotes a line break (i.e., an instruction for you to press ENTER).

```
>·mean(c(1,·2,·3))¶
[1]·2
```

This means: do *not* enter the two characters ">·" – i.e., greater-than and space. These are only provided for you so that (1) you know that this is code you are supposed to enter (i.e., don't enter gray-shaded code that does not begin with ">·") and (2) you can distinguish your *in*put from R's *out*put more easily.

(R code may also be shown inline using the same font and other conventions, like this: `mean(c(1,·2,·3))¶`.) You will also occasionally see lines that begin with "+" or "+·"; these plus signs (and spaces), which you are *not* supposed to enter either, begin lines where R is still expecting further input from you before it begins to execute the function. For example, when you enter "2-" and press ENTER, this is how your R interface will look:

```
>·2-¶
+
```

R is waiting for you to complete the subtraction. When you enter the number you wish to subtract and press ENTER, then the function will be executed properly.

```
+·3¶
[1]·-1
```

Another example: If, for instance, you wish to load a package into R to be able to use the functionality it offers, you can use the function `library` for that. For instance, you could type this to load the package dplyr: `library(dplyr)¶`. (Note: this only works when you have installed the package as explained above.) However, if you forget the closing parenthesis, R will wait for you to provide it, and once you provide the missing closing parenthesis, R will execute the line:

```
>·library(dplyr¶
+·)¶
>
```

```
PartOfSpeech  →  TokenFrequency  → TypeFrequency  →  Class¶
ADJ           →  421            →  271            →  open¶
ADV           →  337            →  103            →  open¶
N             →  1411           →  735            →  open¶
CONJ          →  458            →  18             →  closed¶
PREP          →  455            →  37             →  closed¶
```

*Figure 3.1* Representational format of corpus files and data frames.

If you need to interrupt an R process, you can press ESC.

Corpus files or tables/data frames will be represented as in Figure 3.1, where "→" and "¶" denote tabstops and line breaks respectively.

Menus, submenus, and commands in submenus in applications are given in italics in double quotes, and hierarchical levels within application menus are indicated with colons. So, if you open a document in, say, LibreOffice.org Calc, you do that with what is given here as "*File: Open* . . . ".

Before we delve into R, let me finally mention several ways of using R's own documentation. The simplest way is to just use RStudio: Go the pane that has a "Help" tab, where you have access to a lot of documentation; most helpful at the beginning might be "An introduction to R" as well as "Learning R Online"; later also "R Data Import/Export". Obviously, you can also just Google the name of a function, package, or anything else since there are tons of extremely useful websites, videos, blogs, etc. out there that provide help for every level of expertise. Second, if you know the name of a function or construct and just need some information about it, type a question mark directly followed by the name of the function at the R prompt to access the help file for this function (e.g., `?help¶`, `?sqrt¶`, `?"["¶`). Third, if you know what you would like to do but don't know the function, you can use the search field at the top right of the "Help" tab to search for anything.

## 3.1 Data Structures, Functions, Arguments

The simplest way of using R is using it like you would use a pocket calculator. At the R prompt, you just enter any arithmetic operation and hit ENTER (we'll talk about the `[1]` in a moment):

```
>·2+2¶
[1]·4
>·3^2¶
[1]·9
>·(2+3)^2¶
[1]·25
```

As you may recollect from your math classes in school, however, you often had to work with variables rather than the numbers themselves. R can also operate with variable names, where the names represent the content of so-called data structures. *Data structures* in R can take on a variety of forms. The simplest one, a *vector*, can contain just a number; a more complex one, a *list*, can contain many large tables, texts, numbers, and other data structures. Data structures can be entered into R at the prompt, but the more complex a data structure becomes, the more likely it becomes that you read it from a file,

and this is in fact what you will do most often in this book: reading in text files or tables, processing text files, performing computations on tables, and saving texts or tabular outputs into text files.

One of the most central things to understand about R is how you tell it to do something other than the simple calculations from above. A command in R virtually always consists of two elements: a function and, in parentheses, arguments, where arguments can be null, in which case there could be just opening and closing parentheses. A *function* is an instruction to do something, and the *arguments* to a function represent (1) what the instruction is to be applied to, and (2) how the instruction is to be applied to it. Let us look at two simple arithmetic functions you know from school. If you want to compute the square root of 5 with R – without simply entering `5^0.5`¶, that is – you need to know the name of the function and how many and which arguments it takes. Well, the name of the function is `sqrt`, and it takes just one argument – the number for which you want the square root:

```
>·sqrt(5)¶
[1]·2.236068
```

As another example, let us compute the base-10 logarithm of 150. One possibility would be to use `log10`, which, just like `sqrt`, only takes one argument – the number for which you want the logarithm. However, since one may also want logarithms to other bases, there is a more flexible function in R, `log`, which takes two arguments: The first is the number of which you want the logarithm, the second is the base to the logarithm. Thus, you can enter this:

```
>·log(150,·10)¶
[1]·2.176091
```

One important aspect here is that this way of computing the log is only a short version of the more verbose version:

```
>·log(x=150,·base=10)¶
[1]·2.176091
```

In this longer version, the arguments provided are labeled with the names that R expects the arguments of `log` to have: `x` for the number for which you want the logarithm and `base` for the base. However, if you provide the arguments in *exactly* the order in which R expects them, you can leave the labels out – but if you change the order or want to leave out one of several arguments, then you must label the arguments so that R knows what in your list of arguments means what.

Three other things are worth mentioning: First, outside of quotes, R does not care whether you put a space before the comma or not. Second, R's understanding of its input is case-sensitive: It doesn't know the function `Log`. Third, as you could see when you executed the lines of the code file in RStudio and as you now see again, R ignores everything after a `#`, so you can use this to comment your lines when you write small scripts and want to tell/remind yourself what a particular line is doing.

```
>·Log(150,·10)·#·R·does·not·know·this·function·-·it·only·
  knows·"log"¶
Error:·could·not·find·function·"Log"
```

Note that, in all these cases, R does not store any result – it just outputs it. When you want to assign some content to some data structure for later use, you must use the assignment operator `<-` (a less-than sign and a minus, which together look like an arrow). You can basically choose any name as long as it contains only letters, numbers, underscores, or periods and starts with a letter or a period. However, to avoid confusion, you should not use names for data structures that are R functions (such as `c`, `log`, or `sqrt` or the many other functions you'll get to know below). As a result of assignment, the content resulting from the function is available in the data structure just defined. For example, this is how you store the square root of 5 into the kind of data structure you will get to know as a vector, which is here named `aa`.

```
>·aa<-sqrt(5)¶
```

You can now check whether R actually knows about this vector by instructing R to list all the data structures it knows about (in the current R session):

```
>·ls()¶
[1]·"aa"
```

And you can instruct R to output the contents of the vector (by default to the screen):

```
>·aa¶
[1]·2.236068
```

A nice short way to instruct R to assign a value to a data structure *and* output that data structure at the same time is by putting the assignment into parentheses:

```
>·(aa<-sqrt(5))¶
[1]·2.236068
```

Note that the assignment operator can also be used to change the value of an existing data structure using the name of the data structure again. The following can be read as "create a new version of `aa` by taking the old one and adding 2 to it":

```
>·(aa<-aa+2)¶
[1]·4.23606
```

Note that you can enter more than one command per line by separating commands with a semicolon.

```
>·sqrt(9);·sqrt(16)¶
[1]·3
[1]·4
```

If you ever want to get rid of data structures again, you can also remove them. Either you remove an individual data structure by using `rm` (for remove) and provide the data structure(s) to be deleted as an argument (or as arguments):

```
>·rm(aa)¶
```

or you can just delete all data structures:

```
>·rm(list=ls(all=TRUE))¶
```

Finally, before we look at several data structures in more detail shortly, note that not only may functions not require their arguments to be labeled, many functions even have default settings for arguments which they use when the argument is not defined by the user. A function to exemplify this characteristic that is actually also very useful in a variety of respects is `sample`. This function outputs random or pseudo-random samples, which is useful when, for example, you have a large number of matches or a large table and want to access only a small, randomly chosen sample or when you have a vector of words and want it re-sorted in a random order.

```
sample(x,·size,·replace=FALSE,·prob=NULL)·#·no·">·"·at·the·
  beginning:·don't·enter·this!
```

The function `sample` can take up to four arguments:

- `x`: a data structure, most likely a vector, providing the elements from which you want a sample. If that data structure `x` is a vector of one number only, then R outputs a random ordering of the numbers from 1 to `x`; if `x` is a vector of two or more elements, then R outputs a random ordering of the elements of that vector.
- `size`: an integer number determining the size of the sample; by default, that is the number of elements of `x`, which we will later determine with `length(x)`;
- a logical expression: `replace=FALSE` (if each element of the vector can only be sampled once, the default setting) or `replace=TRUE` (if the elements of the vector can be sampled multiple times, sampling with replacement);

- `prob`: a vector of probabilities with which elements of `x` may be sampled; the default setting is `NULL`, representing equiprobable sampling, i.e., sampling where each element of `x` has the same chance of being sampled.

Let us look at a few examples which make successively more use of label omission and default settings. First we generate a vector `qwe` with the numbers 1 to 10 by using `:` as a range operator:

```
>·(qwe<-c(1:10))¶
[1]··1··2··3··4··5··6··7··8··9·10
```

If you now want to draw five elements randomly and equiprobably from `qwe` with replacement, you enter this:

```
>·sample(qwe,·size=5,·replace=TRUE,·prob=NULL)¶
[1]·2·9·4·4·7
```

However, since you provide the arguments in the default order, you can do away with the labels (although you now of course get different random numbers):

```
>·sample(qwe,·5,·TRUE,·NULL)¶
[1]·7·2·3·6·7
```

But since `prob=NULL` is the default setting, you might as well omit that, too:

```
>·sample(qwe,·5,·TRUE)¶
[1]··6·6·6·6·9
```

And this is how we draw five elements equiprobably and randomly *without* replacement:

```
>·sample(qwe,·5,·FALSE)¶
[1]·9··2··6··7·10
```

But again, since `replace=FALSE` is the default, why not omit it?

```
>·sample(qwe,·5)¶
[1]·3·1·2·9·8
```

You have now seen that the arguments `prob` and `replace` can be omitted because of their default settings. But actually, it is also possible to omit the `size` argument. If the

`size` argument is omitted, its default kicks in and R assumes we want all elements of the vector `qwe` back again and, thus, effectively only provides us with a random order of the elements of `qwe`:

```
>·qwe¶
[1]··1··2··3··4··5··6··7··8··9·10
>·sample(qwe)¶
[1]··8··6·10··4··5··2··9··1··3··7
```

In fact, if the idea is just to get a random ordering of the numbers of 1 to *n*, then you do not even need to provide R with a vector giving all the numbers from 1 to 10 as you did when we defined `qwe` above. You can just give R the number:

```
>·sample(10)¶
[1]··4··9··2··5··8··6·10··1··7··3
```

In fact, default settings sometimes lead to the extreme case of function calls without *any* argument (like `help.start()` above). The function `q` shuts R down and internally processes three arguments:

- A one-element character vector called `save` – i.e., a sequence of characters such as letters, numbers, or other symbols – specifying whether the current workspace, i.e., the data structures you worked with since you last started R, should be saved or whether the user should be prompted regarding whether the workspace should be saved. The latter is the default.
- A one-element numeric vector called `status` specifying what R should 'tell' the operating system when R is shut down. The default is 0, which means "successful shut down".
- A one-element logical vector called `runLast` (`TRUE` or `FALSE`) specifying whether some other user-specified function should be executed before R shuts down. The default is `TRUE`.

Thus, if you want to shut down R, you just enter:

```
>·q()¶
```

Since you have not provided any arguments at all, R assumes the default settings. The first requires R to ask you whether the workspace should be saved. Once you answer this question, R will shut down and send 0 (i.e., "successful shut down") to your operating system.

As you can see, label omission and default settings can be very useful ways of minimizing typing effort. However, especially at the beginning, it is probably wise to try to strike a balance between minimizing typing on the one hand and maximizing transparency of the code on the other. While this may ultimately boil down to a matter of personal preferences, I recommend using more explicit code at the beginning in order to be maximally aware of the options your R code uses. The next sections will introduce data structures that are most relevant to linguistic and statistical analysis.

**Recommendation(s) for further study/exploration**

- The functions `?` or `help`, which provide the help file for a function (try `?sample¶` or `help(sample)¶`), and the functions `args` and `formals`, which provide the arguments a function needs, their default settings, and their default order (try `args(sample)¶` or `formals(sample)¶`).
- The function `set.seed`, which I use in the code file to make sure we obtain the same random but replicable numbers: `?set.seed¶`.
- RStudio offers very useful things to help you write code: code-completion (if you type `samp`, it will already suggest `sample` to you) and argument-insertion (if you type `sample(` and press Tab, you get suggestions for arguments). Finally, pressing CTRL-L in RStudio on Windows and Linux clears the console.

## 3.2 Vectors

### *3.2.1 Basics*

The most basic data structure in R is a vector. Vectors are one-dimensional, sequentially ordered sequences of elements (such as numbers or character strings (e.g., words) or logical values). While it may not be obvious why vectors are important here, we must deal with them in some detail since nearly all other data structures in R can ultimately be understood in terms of vectors. As a matter of fact, you have already used vectors when we computed the square root of *5*:

```
>·sqrt(5)¶
[1]·2.236068
```

The "[1]" in front of the result indicates that R internally represents the result as a vector and that the number that is printed is the first (and here also only) number of that vector.[1] You can even test this with R itself. If you assign the data structure resulting from `sqrt(5)¶` to `aa` and then ask R whether `aa` is a vector, we get R's version of "yes" – `TRUE`, a logical vector of length 1:

```
>·aa<-sqrt(5)¶
>·is.vector(aa)¶
[1]·TRUE
```

You can also find out what kind of vector `aa` is by using the function `class`, and you can also determine that the length of `aa` is 1:

```
>·class(aa)¶
[1]·"numeric"
>·length(aa)¶
[1]·1
```

And, you can define vectors without contents but with a particular length, which may seem senseless right now, but is still something you must remember well because it is more memory-efficient than just letting a vector grow dynamically; this will become much clearer and in fact is necessary below.

```
>·(empty<-logical(length=3))·#·logical·vector·of·length·3¶
[1]·FALSE·FALSE·FALSE
>·(empty<-numeric(length=3))·#·numeric·vector·of·length·3¶
[1]·0·0·0
```

For corpus linguists, it is of course important to know that vectors can also contain character strings – the only difference to numbers is that the character strings have to be put either between double or single quotes. You can freely choose which kind of quotes you use, but the opening and the closing quote must be identical. (For the sake of consistency, I will only use double quotes in this book.)

```
>·(a.name<-"James")¶
[1]·"James"
>·class(a.name)¶
[1]·"character"
>·(empty<-character(length=3))¶
[1]·""·""·""
```

Note what happens when you apply `length` to `a.name`. Contrary to what you might have expected, you do *not* get the number of characters of "James" (i.e., 5) – you get the length of the data structure, and since the vector contains one element – "James" – R returns 1:

```
>·length(a.name)¶
[1]·1
```

There are other types of vectors, but we will not distinguish any others than those mentioned above. However, vectors usually only become interesting when they contain more than one item. The function that concatenates (i.e., combines) several elements into a vector is called `c`, and its arguments are the elements to be concatenated into a vector.

```
>·(numbers<-c(1,·2,·3))¶
[1]·1·2·3
>·(names<-c("James",·"Jonathan",·"Jean-Luc"))¶
[1]·"James"····"Jonathan"·"Jean-Luc"
```

Since an individual number such as the square root of 5 is already a vector, it is not surprising that `c` also connects vectors consisting of more than one element (as does `append`):

```
>·(numbers1<-c(1,·2,·3);·numbers2<-c(4,·5,·6))¶
[1]·1·2·3
[1]·4·5·6
>·(numbers1.and.numbers2<-c(numbers1,·numbers2))¶
[1]·1·2·3·4·5·6
```

A characteristic that will be useful further below is that R can handle and apply vectors recursively. For example, adding two equally long numerical vectors yields a vector with all pairwise sums:

```
>·numbers1+numbers2¶
[1]·5·7·9
```

What happens if vectors are not equally long? Two things can happen. First, if the length of the longer vector is divisible without a remainder by the length of the shorter vector – i.e., if the modulus of the two lengths is 0 (try `11%%3¶` in R) – then the shorter vector is recycled as often as necessary to complete the function. The most frequent such case in practice is that the shorter vector has the length 1. In the following line, R multiplies the first element1 of `numbers1`, the 1, with the first element of `bb`, 10. Then, it 'wants' to multiply the second element of `numbers1`, the 2, with a second element of `bb`, but there is none, so R re-uses the first and only element of `bb` for that, and the same again for the 3 of `numbers1`:

```
>·bb<-10¶
>·numbers1*bb¶
[1]·10·20·30
```

Second, if the length of the longer vector is *not* divisible without a remainder by the length of the shorter vector, the operation proceeds as far as possible, but also returns a warning:

```
>·bb<-c(10,·20)¶
>·numbers1*bb¶

[1]·10·40·30
Warning·message:
In·numbers1·*·bb·:
··longer·object·length·is·not·a·multiple·of·shorter·object·
  length
```

Another characteristic you will use a lot later is that elements of vectors can be named. Here, each element of `numbers1` is named according to its position:

```
>·names(numbers1)<-c("first",·"second",·"third");·numbers1¶
·first·second··third
·····1·····2·····3
```

It is important to note that – unlike arrays in Perl – vectors can only store elements of one data type. For example, a vector can contain numbers *or* character strings, but not really both: If you try to force character strings into a vector previously containing only numbers, R will change the data type, and since you can interpret numbers as characters but not vice versa, R changes the numbers into character strings and then concatenates them into a vector of character strings:

```
>·(mixture<-c(1,·2,·"Benjamin"))¶
[1]·"1"·····="2"·····="Benjamin"
```

The double quotes around the 1 and 2 indicate that these are now understood as character strings, which also means you cannot use them for calculations anymore (unless you change their data type back using `as.numeric`). Apart from `class`, you can identify the type of a vector (or the data types of other data structures) with `str` (for "structure"), which takes as an argument the name of a data structure:

```
>·str(numbers1)¶
num·[1:3]·1·2·3
>·str(mixture)¶
chr·[1:3]·"1"·"2"·"Benjamin"
```

Unsurprisingly, the first vector consists of **num**bers, the second one of **ch**a**r**acter strings. In most cases to be discussed here, vectors will not be entered into R at the console but will be read in from files. In the following section, we will be concerned with how to load files containing text.

Even though we will usually load vectors from files, it is still often necessary to create quite long vectors in which (sequences of) elements are repeated. Instead of typing those into R element-by-element, you can use two very useful functions, `rep` and `seq`. In its simplest form, the function `rep` (for ***rep**etition*) takes two arguments: the element(s) to be repeated and their number(s) of repetitions:

```
>·rep(3,·5)¶
[1]·3·3·3·3·3
>·rep(c(1,·3,·5),·2)¶
[1]·1·3·5·1·3·5
```

But `rep` is more powerful than that. You can also use an argument called `each`; here's an example with a manually entered vector and one with a range:

```
>·rep(c(1,·3,·5),·each=2)¶
[1]·1·1·3·3·5·5
>·rep(1:3,·each=2)¶
[1]·1·1·2·2·3·3
```

The function `seq` (for *sequence*) is also very useful. In one form, `seq` takes three arguments: `from` (the starting point of the sequence), `to` (the end point of the sequence), and `by` (the increment of the sequence). Thus, the following is the same as `numbers<-c(1:3)¶` or `numbers<-1:3¶`:

```
>·(numbers<-seq(1,·3,·1))¶
[1]·1·2·3
```

But since 1 is the default increment, the following would suffice:

```
>·(numbers<-seq(1,·3))¶
[1]·1·2·3
```

In fact, you can even just write this:

```
>·(numbers<-seq(3))¶
[1]·1·2·3
```

This works for character vectors (or other data structures such as lists), too, such that `seq` will check their length and then return a sequence of integers from 1 to the length of the vector, something we will use a lot below in so-called `for`-loops:

```
>·seq(c("a",·"bb",·"ccc"))¶
[1]·1·2·3
```

If the numbers in the vector to be created do not increment by 1, you can set the increment to whatever value you need. The following lines generate a vector `qwe` in which the even numbers between 1 and 10 are repeated three times in sequence. Try it out:

```
>·(qwe<-rep(seq(2,·10,·2),·3))¶
[1]··2··4··6··8·10··2··4··6··8·10··2··4··6··8·10
```

Finally, instead of providing the increment, you can also let R figure it out for you, such as when you know how long your sequence needs to be and just want equal increments

everywhere. You can then use the argument `length.out` as the third argument to `sep`, i.e., instead of the `by` argument. The following generates a seven-element sequence from 1 to 10 with equal increments and assigns it to `numbers`:

```
>·(numbers<-seq(1,·10,·length.out=7))¶
[1]·1.0··2.5··4.0··5.5··7.0··8.5·10.0
```

With `c`, `rep`, and `seq`, even long and complex vectors can often be created quite easily.

**Recommendations for further study/exploration**

- The general vector-creation function `vector`, which we later use to create lists: `?vector¶`.
- On how to change the types of vector to numeric and character strings: `?as.numeric¶` and `?as.character¶`.

### *3.2.2 Loading Vectors*

R has a very powerful function to load the contents of text files into vectors: `scan`. Since this function is central to very many corpus loading operations to be discussed below, we will discuss it and a variety of its arguments in some detail. Several useful arguments of `scan`, together with their default settings, are as follows:

```
scan(file="",·what=double(0),·sep="",·quote=if(identical(sep,·
  "\n"))·""·else·"'\"",·dec=".",·skip=0,·quiet=FALSE,·
  comment.char="",·blank.lines.skip=TRUE)¶
```

- The `file` argument is obligatory (for loading vectors from files at least, see below) and specifies the path to the file to be loaded; usually this will look like this `"home/stgries/Corpora/BNCwe/D8Y.txt"`. Instead of providing a path by entering it there directly, you can also use `file.choose()` as the first argument, in which case R will prompt you with an Explorer/File Manager window so you can click your way to the desired file; once you choose a file, R will return the path to that file as a character string and, thus, to `scan`.
- The `what` argument specifies the kind of input `scan` is supposed to read. The most important setting is `what=character()`, which you use if your file contains text – if your file contains numbers, leave out the `what` argument because then R's default setting will load numbers unproblematically.
- The `sep` argument specifies the character that separates individual entries in the file and that, therefore, determines what the elements of your vector will be. The default setting, `sep=""`, means that any whitespace character will separate entries, i.e., spaces, tabs (represented as "\t"), and newlines (represented as "\r" or "\n" or both). Thus, if you want to read a text file into a vector such that each line is

one element of the vector, you write `sep="\n"`, which is what we will do nearly all of the time.
- The `quote` argument specifies which characters surround text quotes; most of the time `sep` will be set as `sep="\n"`, which entails that `quote` is then automatically set to `quote=""`.
- The `dec` argument specifies the decimal point character; if you want to use a comma instead of a period, just enter that here as `dec=","`.
- The `skip` argument specifies the number of lines you wish to skip when reading in a file, which may be useful when, for example, corpus files have a fixed number of header rows at the beginning of the file.
- The `quiet` argument specifies whether R returns the number of entries it has read in (`quiet=FALSE`, which is the default) or not (`quiet=TRUE`).
- The `comment.char` argument specifies which character is used for comments in the file you are loading; the default is an empty character, which means nothing is ever specially treated as comments during loading.
- The `blank.lines.skip` argument is set to `TRUE`, which means that empty lines will not be represented in the vector; this default setting is nearly always useful, but we will encounter a case in Section 5.4.6 where we will need to set this to `FALSE`.

Let us look at a few examples. First, we load into a vector `x` the contents of the text file <_qclwr2/_inputfiles/dat_vector-a.txt>, which looks like Figure 3.2.

You just enter the following line, making use of the fact that R supplies the default settings unless you specify them explicitly; the line loads the file and prints it to the screen immediately – remember, parenthesized assignments print what is assigned.

```
>·(x<-scan(file.choose(),·sep="\n"))¶
Read·3·items
[1]·1·2·3
```

A slightly more complex example. Imagine you have a file, <_qclwr2/_inputfiles/dat_vector-b.txt>, that looks like Figure 3.3.

Here are two ways to load this file into a vector `x`. The first line reads in the file with the default setting of `sep`, so everything separated by whitespace in the original file becomes an element in the vector `x.1`; this already looks nicely like a vector of words, but think about what would happen with punctuation marks. . . . The second line reads in the file such that everything separated by line breaks becomes an element in the vector `x.2`:

```
1¶
2¶
3¶
```

*Figure 3.2* The contents of <_qclwr2/_inputfiles/dat_vector-a.txt>.

```
This·is·the·first·line¶
This·is·the·second·line¶
```

*Figure 3.3* The contents of <_qclwr2/_inputfiles/dat_vector-b.txt>.

```
>·(x.1<-scan(file.choose(),·what=character()))¶
Read·10·items
·[1]·"This"···"is"·····"the"····"first"··"line"···"This"···"is"·····
  "the"····"second"·"line"··
>·(x.2<-scan(file.choose(),·what=character(),·sep="\n"))¶
Read·2·items
[1]·"This·is·the·first·line"··"This·is·the·second·line"
```

Now, how do we load text files that involve Unicode code points? This is an important issue since corpus files these days will typically be in Unicode encoding, specifically UTF-8. Unfortunately, the way to load them and the way they are displayed in RStudio or the default R console depends on your operating system, your system locale, and the specific file. Here's an example of the kind of problems you can run into on, say, a Windows 10 system (American English locale), when you try to load <_qclwr2/_inputfiles/corp_utf8_cyrillic.txt> (I only show a small part of the output here):

```
>·(cyr<-scan(file.choose(),·what=character(),·sep="\n"))·
  #·on·Windows¶
Read·23·items
[1]·"Ð¢Ð¾Ñ€Ð³Ð¾Ð²Ñ‹Ð¹·Ð´Ð¾Ð¼·ÐÐ´ÐµÑ€Ð°·Ñ Ð²Ð»Ñ ÐµÑ‚Ñ Ñ"
```

Not exactly successful. On Linux Mint 18 (same locale) to be precise, however, no problem:

```
>·(cyr<-scan(file.choose(),·what=character(),·sep="\n"))·
  #·on·Kubuntu¶
Read·23·items
[1]·"Торговый·дом·Эдера·является·оптово-розничной"
```

How do we fix this? As many queries on various lists and websites indicate, it's unfortunately not always that straightforward. The function `scan` has two encoding-related arguments – `encoding` and `fileEncoding` – that sometimes do the trick, so for instance on both Windows and Linux, the following works (see `iconvlist()¶` for supported internationalization conversions):

```
>·(cyr<-scan(file.choose(),·what=character(),·sep="\n",·
  encoding="UTF-8"))¶
```

```
Read·23·items
[1]·"Торговый·дом·Эдера·является·оптово-розничной"
```

but the following also works on Linux, but not on Windows:

```
>·(cyr<-scan(file.choose(),·what=character(),·sep="\n",·
  fileEncoding="UTF-8")¶
```

And with yet other files, even some that come with this book, the pattern might be the opposite, and sometimes it's not even clear what encoding a file comes with because that information is not always provided or obvious. The approach that I now usually end up using and that I find works best – across Windows and Linux, with UTF-8 files but also other encodings – is one based on file connections. There is a function `file`, which establishes a connection to a file and allows you to specify an `encoding` argument, and that connection will then be read with a function called `readLines`, which does exactly what its name suggests. This is how we will work most of the time when encodings are an issue:

```
>·(cyr<-readLines(con<-file(file.choose(),·encoding="UTF-8"),·
  warn=FALSE));·close(con)¶
[1]·"Торговый·дом·Эдера·является·оптово-розничной"
```

This means the user chooses a file, to which `file` establishes a connection that it calls `con` and that assumes the input comes in the provided encoding, i.e., UTF-8. R then reads from that connection, i.e., from the file, the content of the file line by line and puts this into `cyr` (it also suppresses warnings about embedded nuls etc.). When the process is done, the connection to the file is closed again and, as you can see, the data import was successful. As mentioned above, this is the strategy that I found works most reliably, and we will use it for the various encodings we will deal with in this book.

Finally, while loading files will be the most frequent way in which you will use `scan`, let me mention the simplest way in which you can use `scan`, namely to enter vectors. If you just write `scan()¶` or `scan(what=character())¶`, you can enter numbers or strings separated by ENTER until you press ENTER twice to end your input:

```
>·x<-scan()¶
1:·1¶
2:·2¶
3:·3¶
4:¶¶
Read·3·items
>·x¶
[1]·1·2·3
```

**Recommendations for further study/exploration**

- Spector (2008: section 2.7).
- On how to customize input even more with `scan`: `?scan¶`.

### *3.2.3 Accessing and Processing (Parts of) Vectors*

Now that we have covered some aspects of how to load vectors, let us turn to how to access parts of vectors and do something with these parts. The simplest ways to get a glimpse of what any data structure looks like are the functions `head` and `tail`. These take the name of a data structure as one argument and return the six first (for `head`) or last (for `tail`) elements of a data structure; if you want a number of elements other than six, you can provide that as a second argument to `head` and `tail`.

```
>·x<-c("a",·"b",·"c",·"d",·"e",·"f",·"g",·"h")¶
>·head(x)¶
[1]·"a"·"b"·"c"·"d"·"e"·"f"
>·tail(x,·2)¶
[1]·"g"·"h"
```

However, one of the most powerful ways to access parts of data structures is by means of subsetting, i.e., indexing with square brackets. In its simplest form, you can access (and of course change) a single element:

```
>·x[3]¶
[1]·"c"
```

Since we have already seen how flexibly R handles variables, the following extensions of this simple principle should not come as a surprise:

```
>·y<-3¶
>·x[y]¶
[1]·"c"
>·z<-c(1,·3)¶
>·x[z]¶
[1]·"a"·"c"
>·z<-c(1:3)¶
>·x[z]¶
[1]·"a"·"b"·"c"
```

With negative numbers, you choose the data structure *without* the designated elements:

```
>·x[-2]¶
[1]·"a"·"c"·"d"·"e"·"f"·"g"·"h"
```

Note also (for much later) that the square bracket can actually be used as a function (just like other function names such as `head` or `tail`), namely when you make it a character string; then its first argument becomes what data structure to extract something from and its second argument becomes what to extract:

```
>·"["(x,·3:4)¶
[1]·"c"·"d"
```

You can use the names of vector elements for subsetting, too:

```
>·qwe<-1:3;·names(qwe)<-c("a",·"b",·"c");·qwe¶
a·b·c
1·2·3
>·qwe["b"]¶
b
2
```

This can be very useful if, for instance, something like `qwe` is a potentially very long frequency list of words and you can just subset it using the word whose frequency you're interested in.

R also offers more useful functions, though. One of the most interesting ones is to let R decide which elements of a vector satisfy a particular condition. The simplest, though not always most elegant, solution is to present R with a logical expression.

```
>·x=="d"¶
[1]·FALSE·FALSE·FALSE··TRUE·FALSE·FALSE·FALSE·FALSE
```

R checks all elements of the argument preceding the logical operator == and outputs for each whether it meets the condition given in the logical expression or not. The one thing you need to bear in mind is that this logical expression – the test for identity – uses == rather than the = we already know from the assignment of values to arguments. Other logical operators we will use are the following:

| | | | |
|---|---|---|---|
| & | and | \| | or |
| > | greater than | >= | greater than or equal to |
| < | less than | <= | less than or equal to |
| ! | not | != | not equal |

The following examples illustrate these logical expressions:

```
>·(x<-c(10:1))¶
[1]·10··9··8··7··6··5··4··3··2··1
>·x==4¶
[1]·FALSE·FALSE·FALSE·FALSE·FALSE·FALSE··TRUE·FALSE·FALSE·FALSE
>·x<=7¶
[1]·FALSE·FALSE·FALSE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE
>·x!=8¶
[1]··TRUE··TRUE·FALSE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE··TRUE
>·(x>8·|·x<3)¶
[1]··TRUE··TRUE·FALSE·FALSE·FALSE·FALSE·FALSE·FALSE··TRUE··TRUE
```

In fact, you can use such logical vectors of TRUEs and FALSEs for subsetting just like you used numbers or names. You have seen that when you subset a vector with a numeric vector in square brackets (as in `x[3]¶` or `x[y]¶` above), you get the elements in the positions indexed by the numbers. The following line shows that when you subset a vector with a logical vector of TRUEs and FALSEs in square brackets, you get the elements in the positions of the TRUEs:

```
>·x[x<=7]¶
[1]·7·6·5·4·3·2·1
```

And since TRUEs and FALSEs correspond to 1s and 0s, you can sum up a logical vector to find out how many TRUEs there are in it, i.e., how many vector elements satisfy a condition:

```
>·sum(x==4)¶
[1]·1
>·sum(x>8·|·x<3)¶
[1]·4
```

Or, for a more detailed breakdown, you can use the function `table`, which in its simplest use looks at one vector (but see below), counts how often each element that occurs in it at least once occurs, and provides an alphabetically ordered frequency list:

```
>·table(x>8·|·x<3)¶
FALSE··TRUE
····6·····4
```

Note (again, for later) that `table` does not automatically also return a count of missing data – to obtain those, too, you need to add the argument `useNA="ifany"`). I hope you can already foresee that we will use `table` a lot to generate frequency lists of corpora: once a corpus is stored in R such that every word is a vector element, using `table` is all it takes.

While this illustrates how logical expressions work, it probably also illustrates that this is not a particularly elegant way to proceed: Regardless of how many elements of `x`

fulfill the condition, you always get ten truth values and must then identify the cases of `TRUE` yourself. In other words, this approach does not scale, meaning it is not practicable once the data you are looking at are 10, 1,000, 100,000 times as large – what would you do, look at 100,000 `TRUE`s and `FALSE`s? Thankfully, there is a more elegant way of doing this using `which`, which takes as an argument a logical vector (e.g., something resulting from a logical expression of the kind above) and outputs the positions of `TRUE`s, i.e., of the element(s) satisfying a particular condition:

```
>·which(x==4)¶
[1]·7
```

Note how much this actually looks like English: you are asking "which x is 4?" and get the answer "the seventh". The following examples correspond to the ones we just looked at but use `which`:

```
>·which(x<=7)¶
[1]··4··5··6··7··8··9·10
>·which(x!=8)¶
[1]··1··2··4··5··6··7··8··9·10
>·which(x>8·|·x<3)¶
[1]··1··2··9·10
```

A central point here is not to mix up the *position of an element* in a vector and the *element* in a vector. If we come back to `which(x==4)¶`, remember that it does not output 4 – the element in `x` – but 7 – the position that 4 occupies in `x`. If, however, you have been following along so far, you may already be able to guess how you get the element itself, so try to write some code that retrieves the elements of `x` that are larger than 8 or smaller than 3 and stores them in a vector `y`.

**THINK**

**BREAK**

Since you can access elements of a vector with square brackets and since the output of `which` is itself a vector, you can simply write:

```
>·(pointer<-which(x>8·|·x<3))¶
[1]··1··2··9·10
>·(y<-x[pointer])¶
[1]·10··9··2··1
```

Or, because you guess that you can combine it all into one expression using the position numbers returned by `which` or even just the logical values returned by the logical expression:

```
>·x[which(x>8·|·x<3)]¶
[1]·10··9··2··1
>·x[x>8·|·x<3]¶
[1]·10··9··2··1
```

To find out how many elements of a vector fulfill the condition, you can apply `length` or, since you remember that `TRUE` and `FALSE` are interpreted as 1 and 0 respectively, `sum`:

```
>·length(which(x>8·|·x<3))¶
[1]·4
>·sum(x>8·|·x<3)¶
[1]·4
```

Thus, the fact that R uses vectors for nearly everything is in fact a great strength and makes it a very versatile language. When you combine the above with subsetting, you can also change elements of vectors very quickly. For example, if you want to replace all elements of `x` which are greater than 8 by 12, this is one way of achieving this:

```
>·x¶
[1]·10··9··8··7··6··5··4··3··2··1
>·y<-which(x>8)¶
>·(x[y]<-12)¶
[1]·12·12··8··7··6··5··4··3··2··1
```

or even directly with the position indices returned by `which`:

```
>·x<-10:1¶
>·(x[which(x>8)]<-12)¶
[1]·12·12··8··7··6··5··4··3··2··1
```

or even just with the `TRUE`s from a logical vector:

```
>·x<-c(10:1)¶
>·(x[x>8]<-12)¶
[1]·12·12··8··7··6··5··4··3··2··1
```

Apart from `which` and subsetting, there are also some more powerful functions available for processing vectors. We will briefly look at `%in%` and `match`, but discuss only simple cases here. The first argument of `%in%` is a data structure (usually a vector) with the elements to be matched; the second is a data structure (also usually a vector) with the elements to be matched against. The output is a logical vector of the length of the first argument with `TRUE`s and `FALSE`s for the elements of the first data structure that are found

and that are not found in the second data structure respectively. Let me clarify this with two examples:

```
>·x<-c(10:1);·y<-c(2,·5,·9)¶
>·x·%in%·y¶
[1]·FALSE··TRUE·FALSE·FALSE·FALSE··TRUE·FALSE·FALSE··TRUE·FALSE
>·y·%in%·x¶
[1]·TRUE·TRUE·TRUE
```

The first example, `x·%in%·y¶`, shows that the first element of `x` – 10 – does not appear in `y`, while the second element – 9 – does, etc. You can combine the logical-vector output of `%in%` with subsetting to get at the matched elements quickly:

```
>·x[x·%in%·y]¶
[1]·9·5·2
```

The function `match` returns a vector of the positions of (first!) matches of its initial argument in its second (again, both arguments are typically vectors):

```
>·match(x,·y)¶
[1]·NA··3·NA·NA·NA··2·NA·NA··1·NA
```

This tells you, as above, that the first element of `x` does not appear in `y`, but that the second element of `x` – 9 – is the third element in `y`. The third, fourth, and fifth element of `x` do not appear in `y`, but the sixth element – 5 – appears in `y`, namely at position 2 of `y`, etc. With this, it should now be clear what the following line does, in which the arguments are reversed:

```
>·match(y,·x)¶
[1]·9·6·2
```

Same thing: the first element of `y` – the 2 – is the ninth element of `x`. The second element of `y` – the 5 – is the sixth element of `x`, and so on. You may ask yourself what this is good for, but you will get to see very useful applications of these two functions, in particular of `match`, below.

**Recommendations for further study/exploration**

- On how to identify elements in a vector that are duplicates of earlier elements in that vector as in `duplicated(c(1,·2,·3,·4,·3,·2,·1))¶`, which can be useful for type-token counts: The number of types is the number of all non-duplicated tokens: see `?duplicated¶`.
- On how to compute cumulative sums as in `cumsum(1:4)¶`: see `?cumsum¶`.

Another related way of processing two vectors uses basic set-theoretic concepts. Let me introduce three such functions – `setdiff`, `intersect`, and `union`, which all take two vectors as arguments – on the basis of the two vectors `x` and `y` we just used. The function `setdiff` returns the elements of the vector given as the first argument that are *not* in the vector given as the second argument:

```
>·setdiff(x,y)¶
[1]·10··8··7··6··4··3··1
>·setdiff(y,x)¶
numeric(0)
```

The function `intersect` provides the elements of the vector of the first argument that *do* also occur in the vector given as the second argument; note that the order in which the elements are returned is determined by the order in which they occur in the vector given as the first argument:

```
>·intersect(x,y)¶
[1]·9·5·2
>·intersect(y,x)¶
[1]·2·5·9
```

The function `union` provides the elements that occur at least once in the combination of the two vectors given as arguments; again, the order depends on which vector is listed first:

```
>·union(x,y)¶
[1]·10··9··8··7··6··5··4··3··2··1
>·union(y,x)¶
[1]··2··5··9·10··8··7··6··4··3··1
```

Let me now briefly also mention the very useful functions `unique`, and then return to the function `table`. The function `unique` takes as its argument a vector or a factor and should be especially easy to explain to linguists since it outputs a vector containing all the types that occur at least once among the tokens (i.e., elements) of said vector/factor:

```
>·g<-c("a",·"b",·"c",·"b",·"c",·"d",·"c",·"d",·"e")¶
>·h<-c("b",·"c",·"a",·"e",·"b",·"f",·"c",·"a",·"b")¶
>·unique(h)¶
[1]·"b"·"c"·"a"·"e"·"f"
```

As you saw above, the function `table` takes as an argument one or more vectors or factors and provides the token frequency of each type or each combination of types.

```
>·table(g)¶
g
a·b·c·d·e
1·2·3·2·1
```

That is, in `g` there is one "a" and there are two "b"s, etc. But now what happens with two vectors?

```
>·table(g,·h)¶
···h
g···a·b·c·e·f
··a·0·1·0·0·0
··b·0·0·1·1·0
··d·1·1·1·0·0
··d·1·0·0·0·1
··e·0·1·0·0·0
```

This is perhaps a little difficult to interpret at first sight, and you will have to remember that vectors are ordered sequences. This output tells you how often all possible combinations of elements in `g` and `h` occur. Beginning in the upper left corner and then moving to the right: There is no occasion at which there is an "a" in `g` and an "a" in `h`. There is one occasion at which there is an "a" in `g` and a "b" in `h` (element 1 of both vectors). There are no occasions where there is an "a" in `g` and a "c" or an "e" or an "f" in `h`. Analogously and in the next row, there are no positions where there's a "b" in `g` and an "a" or a "b" in `h`, but there is one position where `g` is "b" and `h` is "c" (element 2), and so on.

As mentioned above, `table` will prove very useful later on. It is also worth mentioning here that there is a related function called `prop.table`, which provides a table of percentages. This function takes as its first argument a table generated by `table` and then as a second argument

- nothing, if you want percentages to add up to 100 (or 1, for that matter) in the whole table;
- a "1" if you want percentages that add up to 1 row-wise;
- a "2" if you want percentages that add up to 1 column-wise.

Try it out by entering `prop.table(table(g,·h))¶` or `prop.table(table(g,·h),·1)¶` or `prop.table(table(g,·h),·2)¶`.

Let me finally mention two useful functions concerned with the order of elements in vectors. The first function is used to sort the elements of a vector into alphabetical or numerical order. It is appropriately called `sort` and the main two arguments we will discuss here are the vector to be sorted and an argument `decreasing=FALSE` (the default setting) or `decreasing=TRUE`:

```
>·h¶
[1]·"b"·"c"·"a"·"e"·"b"·"f"·"c"·"a"·"b"
```

```
>·sort(h,·decreasing=TRUE)¶
[1]·"f"·"e"·"c"·"c"·"b"·"b"·"b"·"a"·"a"
```

The other function is called `order`. It takes as arguments one or more vectors and `decreasing=...` – but provides a very different output. Can you recognize what `order` does?

```
>·z<-c("a",·"c",·"e",·"d",·"b")¶
>·order(z,·decreasing=FALSE)¶
[1]·1·5·2·4·3
```

**THINK**

**BREAK**

The output of order when applied to a vector `z` is a vector which tells you in which order to put the elements of `z` to sort them as specified. Let us clarify this rather opaque characterization: If you want to sort the values of `z` in increasing order, you first have to take `z`'s first value (the "a"). Thus, the first value of `order(z,·decreasing=FALSE)¶` is 1. The next/second value you have to take is the fifth value of `z` (the "b"), the next/third value you take is the second value of `z` (the "c"), etc. (If you provide `order` with more than one vector, additional vectors are used to break ties.) In other words, `order` produces a vector that, if you use it to subset the original element (here, `z`), orders that original element as specified:

```
>·z[order(z,·decreasing=FALSE)]¶
[1]·1·2·3·4·5
```

As we will see below, this function will turn out to be useful when applied to data frames.

### *3.2.4 Saving Vectors*

The function to output vectors we will use most is `cat`, which can also take a variety of arguments, some of which we will discuss here:

```
cat(...,·file="",·sep="·",·append=FALSE)¶
```

- The first, obligatory, argument `...` is the vector you want to output.
- The second argument is the `file` argument. If you don't provide it, the output is directed to the console. If you want to save the output into a file, you can either specify a path to a file directly or enter `file.choose()` as the argument. In general,

make sure you provide an extension to the file name; in the context of this book, the files you create with `cat` or `write.table` below should either have the extension .txt or.csv (if what you are printing as a vector line by line amounts to a table/data frame).

- The `sep` argument works in just the same way as it does in `scan`. If you want each vector element to get its own line, which is probably the most useful strategy for nearly all our cases, enter `sep="\n"`, but the default separator is just a space.
- The `append` argument specifies whether (1) the output is appended to the end of an already existing file or whether (2) a new file is created for the output or an existing file is overwritten (the default).

You should now do "Exercise box 3.1: Handling vectors."

**Recommendations for further study/exploration**

- On how to test whether any one or all elements of a vector satisfy a logical expression: `?any¶` and `?all¶`.
- The `fill`-argument of `cat`.
- Spector (2008: sections 6.1–6.4).

## 3.3 Factors

The second data structure we will look at are factors. Factors are superficially similar to vectors, and in this book they are of less importance since we mostly deal with character vectors; thus, we will deal with them here only cursorily. The most straightforward way to generate a factor is by first generating a vector as introduced above and then turning it into a factor using the command `factor` with the vector to be factorized as the first argument:

```
>·f<-c("open",·"open",·"open",·"closed",·"closed")¶
>·(f<-factor(f))¶
[1]·open···open···open···closed·closed
Levels:·closed·open
```

The function `factor` can also take a second argument, `levels`, which specifies the levels the factor has. If you print out a factor, all levels of the factor that occur at least once are outputted in the same way that `unique` outputs all values of a vector, but, crucially and unlike `unique`, a factor can have levels that are not attested in the factor.

If you want to change an element of a factor into something else, you will face one of two scenarios. First, the simple one: you want to change a factor element into something for which the factor already has a level (most likely because it is already attested in the factor or was in the past). In that case, you can treat factors like vectors:

```
>·f[2]<-"closed";·f¶
[1]·open···closed·open···closed·closed
Levels:·closed·open
```

Second, the more complex one: You want to change a factor element into something that is not already attested in the factor and never has been. In that case, you must first redefine the factor by adding the level that you want to change something into, and then you can treat that new factor like a vector again, as above:

```
>·f<-factor(f,·level=c(levels(f),·"half-open"))¶
>·f[3]<-"half-open";·f¶
[1]·open·····closed····half-open·closed····closed···
Levels:·closed·open·half-open
```

Note again that, unlike with `unique`, a factor can have levels that are not attested among its elements, as when you change the third element of `f` back into what it was:

```
>·f[3]<-"open";·f¶
[1]·open···closed·open···closed·closed
Levels:·closed·open·half-open
```

The simplest way to get rid of those is to redefine the factor using `droplevels`:

```
>·(f<-droplevels(f))¶
[1]·open···closed·open···closed·closed
Levels:·closed·open
```

Finally, note that there is a nice function called `cut`, which takes as its first argument a numeric vector and, in its simplest form, a number of levels as its second. It returns a factor that replaces each numeric value by the factor level that it has been categorized into. For example, the following creates a factor `f` from the number from 1 to 12 by creating four levels and replacing each number by the group of values it belongs in. The notation (0.989,3.75] means 'numbers greater than 0.989 and less than or equal to 3.75', which is R's way saying '1 to 3', etc.

```
>·(f<-cut(1:12,·4))¶
·[1]·(0.989,3.75]·(0.989,3.75]·(0.989,3.75]·(3.75,6.5]··
·[5]·(3.75,6.5]···(3.75,6.5]···(6.5,9.25]···(6.5,9.25]··
·[9]·(6.5,9.25]···(9.25,12]····(9.25,12]····(9.25,12]···
Levels:·(0.989,3.75]·(3.75,6.5]·(6.5,9.25]·(9.25,12]
```

You can check that R formed the right four groups for the numbers from 1 to 12 – obviously 1:3, 4:6, 7:9, 10:12 – by cross-tabulating the original numeric vector and the new factor f:

```
> table(1:12, f)
    f
     (0.989,3.75] (3.75,6.5] (6.5,9.25] (9.25,12]
  1             1          0         0          0
  2             1          0         0          0
  3             1          0         0          0
  4             0          1         0          0
  5             0          1         0          0
  6             0          1         0          0
  7             0          0         1          0
  8             0          0         1          0
  9             0          0         1          0
  10            0          0         0          1
  11            0          0         0          1
  12            0          0         0          1
```

**Recommendations for further study/exploration**

- Spector (2008: chapter 5).

## 3.4 Data Frames

While vectors are the most relevant data structure for retrieving data from corpora, data frames are most relevant, first, as a tabular output form of corpus data (recall the discussion of frequency lists, collocate displays, concordances, etc. above), and, second, as the main input data structure for statistical analysis (often created/modified in a spreadsheet software and then imported into R). This data structure, which basically corresponds to a two-dimensional matrix, will be illustrated in this section.

### *3.4.1 Generating Data Frames in R*

While data frames are usually loaded from text files generated with other programs, you can also generate data frames within R by combining several equally long vectors and/or factors. Let us assume you have characterized five parts of speech, as captured in a variable PARTOFSPEECH, in terms of three other variables (or parameters or characteristics):

- a variable "TOKENFREQUENCY" – i.e., the frequency of all individual words of this part of speech in a very small corpus *C*;
- a variable "TYPEFREQUENCY" – i.e., the number of all different words of this part of speech in *C*;
- a variable "CLASS" – i.e., whether the part of speech is an open class or a closed class.

| PARTOFSPEECH | TOKENFREQUENCY | TYPEFREQUENCY | CLASS |
|---|---|---|---|
| ADJ | 421 | 271 | open |
| ADV | 337 | 103 | open |
| N | 1411 | 735 | open |
| CONJ | 458 | 18 | closed |
| PREP | 455 | 37 | closed |

*Figure 3.4* An example data frame.

Let us further assume the variables and the data frame you wish to generate should look like Figure 3.4.

In a first step, you generate the four vectors, one for each column of the data frame:

```
> rm(list=ls(all=TRUE))
> PARTOFSPEECH<-c("adj", "adv", "n", "conj", "prep")
> TOKENFREQUENCY<-c(421, 337, 1411, 458, 455)
> TYPEFREQUENCY<-c(271, 103, 735, 18, 37)
> CLASS<-c("open", "open", "open", "closed", "closed")
```

Note that the first row of the data frame you want to generate does not contain data points, but the names of the columns. You must now also decide whether you would like the first column to contain case numbers from 1 to *n* or names of rows. If you prefer the former, then you can simply generate the data frame using `data.frame`, which in this case takes as arguments only the vectors/factors you want to combine. Note that the order of the vectors is only important in that it determines the order of the columns in the data frame. Let us now look at the data frame and its properties using the functions `str` (for ***str****ucture*) and `summary`:

```
> (x<-data.frame(PARTOFSPEECH, TOKENFREQUENCY, TYPEFREQUENCY,
  CLASS)
  PARTOFSPEECH TOKENFREQUENCY TYPEFREQUENCY  CLASS
1          adj           421           271   open
2          adv           337           103   open
3            n          1411           735   open
4         conj           458            18 closed
5         prep           455            37 closed

> str(x)
'data.frame':     5 obs. of  4 variables:
 $ PARTOFSPEECH  : Factor w/ 5 levels "adj","adv","conj",..:
  1 2 4 3 5
 $ TOKENFREQUENCY: num  421 337 1411 458 455
 $ TYPEFREQUENCY : num  271 103 735 18 37
 $ CLASS         : Factor w/ 2 levels "closed","open": 2 2 2 1 1

> summary(x)
 PARTOFSPEECH TOKENFREQUENCY   TYPEFREQUENCY      CLASS
 adj  :1      Min.   :337.0   Min.   : 18.0   closed:2
```

```
·adv·:1·····1st·Qu.:·421.0···1st·Qu.:·37.0···open··:3
·conj:1·····Median·:·455.0···Median·:103.0···········
·n···:1·····Mean···:·616.4···Mean···:232.8···········
·prep:1·····3rd·Qu.:·458.0···3rd·Qu.:271.0···········
·············Max.···:1411.0···Max.···:735.0···········
```

We can see several things from this output. First, R has generated the data frame as desired. Second, R has automatically converted those vectors which contained character strings into factors (namely, PARTOFSPEECH and CLASS), and we can see that factors are internally represented as numbers (e.g., for CLASS, "closed" is 1 and "open" is 2). Third, since we have not specified any row names, R automatically numbers the rows. Finally, the column names in the output of `str` are preceded by a `$` sign. This means that when you have stored a data frame in R, you can access columns by using the name of the data frame, a `$` sign, and a column name (if the column name includes spaces, you need to put the column name in double quotes):

```
>·x$PARTOFSPEECH¶
[1]·adj··adv··n····conj·prep
Levels:·adj·adv·conj·n·prep
```

The above way of generating data frames is the default that we will use in most cases.

### *3.4.2 Loading and Saving Data Frames in R*

The more common way of getting R to recognize a data frame is to load a file that was generated with spreadsheet software. Let us assume you have generated a spreadsheet file <_qclwr2/_inputfiles/dat_dataframe-a.ods> or a tab- or comma-delimited file <_qclwr2/_inputfiles/dat_dataframe-a.csv> containing the above data frame `x` (using a text editor or a spreadsheet software such as LibreOffice Calc). Ideally your spreadsheet would only contain alphanumeric characters and no whitespace characters (to facilitate later handling in R or other software). The first step is to save that file as a raw text file. In LibreOffice Calc, you choose the Menu "*File: Save As . . .* " and choose "Text CSV (.csv)" from the "*Save as type*"/Formats menu. Then, you enter a file name, confirm "Automatic file name extension" and confirm you want to save the data into a text CSV format file, if prompted to do so. After that, ideally you choose "Unicode (UTF-8)" as the character set, "{Tab}" as the field delimiter and, usually, delete the text delimiter.

The second step is to load <_qclwr2/_inputfiles/dat_dataframe-a.csv> into R with `read.table`. These are the most frequently used arguments of `read.table` with their default settings:

```
read.table(file=...,·header=FALSE,·sep="",·quote="\"'",·dec=".",·
  row.names,·na.strings="NA",·comment.char="#")
```

- The argument `file` is obligatory and specifies a data frame as saved above (as before with a one-element character vector, you can also just enter `file.choose()` to be prompted to choose a file interactively).

- The argument `header` specifies whether the first row contains the labels for the columns (`header=TRUE`) – as would normally be the case – or not (`header=FALSE`).
- The argument `sep`, as mentioned above, specifies the character that separates data fields, and given the file we wish to import – and in fact most of the time – our setting should be `sep="\t"` for a tab.
- We also know the `dec` argument, which is used as introduced above; in an English locale, the setting would not have to be changed.
- The argument `quote`, which provides the characters used for quoted strings. On most, if not all, occasions you should set this to `quote=""` to avoid input problems.
- The argument `row.names` can either be a vector containing names for the rows or, more typically, the number of the columns containing the row names (typically 1). If you do not specify a `row.names` argument, the rows will be numbered automatically.
- The argument `na.strings` takes a character vector of strings which are to be considered as unavailable/missing data.[2]
- Finally, the argument `comment.char`, which provides R with the character that separates comments from the rest of the line. Just like with `quote`, you will most likely wish to set this to `comment.char=""`.

Thus, in order to import the table we may just have saved from LibreOffice Calc into a.csv file into a data frame called `x`, we enter the following at the R console (the first line is just to clear memory again):

```
>·rm(list=ls(all=TRUE))¶
>·x<-read.table(file.choose(),·header=TRUE,·sep="\t",·
  comment.char="")¶
```

If, by contrast, you wish to save a data frame from R into a text file, you need `write.table`. Here are the most important arguments and their default settings:

```
write.table(x,·file=...,·append=FALSE,·sep="·",·eol="\n",·na="NA",·
  dec=".",·row.names=TRUE,·col.names=TRUE)
```

- The argument `x` is the data frame to be saved.
- The arguments `file`, `append`, `sep`, and `dec` are used in the ways introduced above.
- The argument `quote` specifies whether you want factor levels within double quotes, which is usually not particularly useful for editing data in a spreadsheet software.
- The argument `eol` provides the character that marks the end of lines; this will normally be `eol="\n"`, which is also the default.
- The arguments `row.names` and `col.names` specify whether you would like to include the names (or numbers) of the rows and the names of the columns in the file.

Given the above default settings and under the assumption that your operating system uses an English locale, this would probably be the 'normal' way to save such data frames:

```
write.table(x,·path=file.choose(),·quote=FALSE,·sep="\t",·row.
  names=FALSE)¶
```

**Recommendations for further study/exploration**

- On how to handle and detect missing or non-numeric data: `?NA¶` and `?is.na¶`, as well as `?NaN¶` and `?is.nan¶`, `?na.action¶`, `?na.omit¶`, and `?na.fail¶`.
- On how to test whether cases (e.g., rows in data frames) are complete (i.e., don't have a single `NA` in them): `?complete.cases¶`.

### *3.4.3 Accessing and Processing (Parts of) Data Frames in R*

There are several R functions that are useful for accessing parts of data frames that are to be used for subsequent analysis. To look at these, let us clear memory and then load the data frame from <_qclwr2/_inputfiles/dat_dataframe-a.csv> above into `x` again:

```
>·rm(list=ls(all=TRUE))¶
>·x<-read.table(file.choose(),·header=TRUE,·sep="\t",·comment.
  char="")¶
```

You have seen above that you can use the name of a data frame followed by a `$` and a column name (potentially in quotes) to access columns in a data frame as in `x$TYPEFREQUENCY`. An alternative approach is the use of `with`, where you provide a data structure as the first argument and the part of that data structure that you want to access as the second:

```
>·with(x,·TYPEFREQUENCY)¶
[1]·271·103·735··18··37
```

But that's even more typing than just `x$TYPEFREQUENCY`. The probably simplest way to achieve the same objective is to use `attach` to make all columns of the data frame available without having to type the name of the data frame. This way, while R does not know the column `TYPEFREQUENCY` when the data frame was only loaded (see first line and its output), once it has been attached you can use its column names to access its columns without prefixing the name of the data frame:

```
>·TYPEFREQUENCY¶
Error:·object·'TYPEFREQUENCY'·not·found
>·attach(x)¶
>·TYPEFREQUENCY¶
[1]·271·103·735··18··37
```

To undo an `attach(...)`¶, use `detach(...)`¶. Also, note that if you attach a data frame that has the same column names as a previously attached data frame, R will do it, no problem, but will give you a warning that informs you that the columns of the previously attached data frame are no longer available by just typing the variable name. Also note that if you want to make changes to data that come from a data frame, it is safest to first detach the data frame, make your changes in the original data frame, and then attach it again: This way you make sure that you change the actual data, not the 'copy' that `attach` makes available.

The perhaps most general and versatile approach to accessing data from a data frame is again subsetting. We saw above that we can use square brackets to select parts of unidimensional vectors – since data frames are two-dimensional data structures, we now need two (sets of) figures, one for the rows and one for the columns:

```
>·x[2,3]¶
[1]·103
>·x[2,]¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
2··········adv·····------337·····------103···open
>·x[,3]¶
[1]·271·103·735··18··37
>·x[2:3,4]¶
[1]·open·open
Levels:·closed·open
>·x[c(1,3),·c(2,4)]¶
··TOKENFREQUENCY·CLASS
1·····------421··open
3·····------1411··open
```

As you can see, row and column names are not counted. Also, recall the versatile ways of juxtaposing different functions:

```
>·which(x[,2]>450)¶
[1]·3·4·5
>·x[,3][which(x[,3]>100)]¶
[1]·271·103·735
>·x[,3][x[,3]>100]¶
[1]·271·103·735
>·TYPEFREQUENCY[TYPEFREQUENCY>100]¶
[1]·271·103·735
```

Sometimes, you want to investigate only one part of a data frame, namely a part where one variable has a particular value. One way to get at a subset of the data follows logically from what we have done already. Let us assume you want to define a data frame `y` that contains only those rows of `x` referring to open-class words. If you have already made the variables of a data frame available using `attach`, this would be one possibility to do that:

```
>·(y<-x[which(CLASS=="open"),])¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open
3···········n··········1411··········735···open
```

If you have not made the variable names available, you can still do this:

```
>·(y<-x[which(x[,4]=="open"),])¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open
3···········n··········1411··········735···open
```

The maybe more readable way of achieving the same is provided by `subset`. This function takes as its first argument the data frame to subset and as additional arguments a logical expression specifying the selection conditions:

```
>·(y<-subset(x,·CLASS=="open"))¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open
3···········n··········1411··········735···open

>·(y<-subset(x,·CLASS=="open"·&·TOKENFREQUENCY<1000))¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open

>·(y<-subset(x,·PARTOFSPEECH·%in%·c("adj",·"adv")))¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open
```

This way, data frames can be conveniently customized for many different forms of analysis. If, for whatever reason, you wish to edit data frames in R rather than in a spreadsheet program – for example, because Microsoft Excel or LibreOffice Calc are limited to data sets with maximally about one million rows, you can also do this in R. (One million rows may seem larger than you would ever need to you, but even a frequency list of the BNC is already at that limit.)

Finally, let me mention a very practical way of reordering data frames by means of `order`, introduced above. Recall that `order` returns a vector of positions of elements and recall that subsetting can be used to access parts of vectors, factors, and data frames (and later lists). As a matter of fact, you may already guess how you can reorder

data frames. For example, imagine you want to sort the rows in the data frame `x` according to `CLASS` (ascending in alphabetical order) and, within `CLASS`, according to values of the column `TOKENFREQUENCY` (in descending order). Of course we can use `order` to achieve this goal, but there is one tricky issue involved here: The default ordering style of `order` is `decreasing=FALSE`, i.e., in ascending order, but you actually want to apply two different styles: ascending for `CLASS` and descending for `TOKENFREQUENCY`, so changing the default alone will not help. What you can do, though, is this:

```
> ordering.index<-order(CLASS, -TOKENFREQUENCY); ordering.index
[1] 4 5 3 1 2
```

That is, you apply `order` not to `TOKENFREQUENCY`, but to the negative values of `TOKENFREQUENCY`, effectively generating the desired descending sorting style. Then, you can use this vector to reorder the rows of the data frame by subsetting:

```
> x[ordering.index,]
  PARTOFSPEECH TOKENFREQUENCY TYPEFREQUENCY  CLASS
4         conj           458           18 closed
5         prep           455           37 closed
3            n          1411          735   open
1          adj           421          271   open
2          adv           337          103   open
```

Of course, this could have been done in one line:

```
> x[order(CLASS, -TOKENFREQUENCY),]
```

You can also apply a function we got to know above, `sample`, if you want to reorder a data frame randomly. This may be useful, for example, to randomize the order of stimulus presentation in an experiment or to be able to randomly select only a part of the data in a data frame for analysis. One way of doing this is to first retrieve the number of rows that need to be reordered using `dim`, whose first result value will be the number of rows and whose second result value will be the number of columns (just as in subsetting or `prop.table`):

```
> no.of.rows<-dim(x)[1]
```

Then, you reorder the row numbers randomly using `sample` and then you use subsetting to change the order; of course, you may have a different (random) order:

```
> (ordering.index<-sample(no.of.rows))
[1] 1 4 2 3 5
```

```
>·x[ordering.index,]¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1··········adj·····------421·····------271···open
4·········conj·····------458·····------18·closed
2··········adv·····------337·····------103···open
3···········n·····------1411·····------735···open
5·········prep·····------455·····------37·closed
```

Alternatively, you do it all in one line:

```
>·x[sample(dim(x)[1]),]¶
```

Finally, on some occasions you may want to sort a data frame according to several columns and several sorts (increasing and decreasing). However, you cannot apply a minus sign to *factors* to force a particular sorting style as we did above with TOKENFREQUENCY, which is why in such cases you need to use `rank`, which rank-orders a column with a factor first into numbers, to which then the minus sign can apply:

```
>·ordering.index<-order(-rank(CLASS),·-rank(PARTOFSPEECH))¶
>·x[ordering.index,]¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
3···········n·····------1411·····------735···open
2··········adv·····------337·····------103···open
1··········adj·····------421·····------271···open
5·········prep·····------455·····------37·closed
4·········conj·····------458·····------18·closed
```

You should now do "Exercise box 3.2: Handling data frames."

**Recommendations for further study/exploration**

- On how to edit data frames in a spreadsheet interface within R: `?fix¶` (note also that you can also view them in RStudio by clicking on the *Environment* tab and then on the small spreadsheet symbol next to the name of the data frame you want to view).
- On how to merge different data frames: `?merge¶`.
- On how to merge different columns and rows into matrices (which can in turn be changed into data frames easily): `?cbind¶` and `?rbind¶`.
- See the package dplyr for many useful data frame operations.
- Spector (2008: chapter 2, section 6.8).

## 3.5 Lists

While the data frame is probably the most central data structure for statistical evaluation in R and has thus received much attention here, data frames are actually just a special kind of data structure, namely lists. More specifically, data frames are lists that contain vectors and factors which all have the same length. Lists are a much more versatile data structure which can in turn contain various different data structures within them. For example, a list can contain different kinds of vectors, data frames, and other lists, as well as other data structures we will not discuss explicitly in this book (e.g., arrays or matrices):

```
>·rm(list=ls(all=TRUE))¶
>·a.vector<-c(1:10)¶
>·a.dataframe<-read.table(file.choose(),·header=TRUE,·sep="\t",·
  comment.char="")¶
>·another.vector<-c("This",·"may",·"be",·"a",·"sentence",·
  "from",·"a",·"corpus","file",".")¶
>·(a.list<-list(a.vector,·a.dataframe,·another.vector))¶
[[1]]
·[1]··1··2··3··4··5··6··7··8··9·10
[[2]]
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·········adj···········421··········271···open
2·········adv···········337··········103···open
3···········n··········1411··········735···open
4·······conj···········458··········18·closed
5·······prep···········455··········37·closed
[[3]]
·[1]·"This"·····"may"·····"be"·····"a"·····"sentence"·"from"
·[7]·"a"·····"corpus"···"file"·····"."
```

As you can see, the three different elements – a vector of numbers, a data frame with different columns, and a vector of character strings – are now stored in a single data structure:

```
>·str(a.list)¶
List·of·3
·$·:·int·[1:10]·1·2·3·4·5·6·7·8·9·10
·$·:'data.frame':·····5·obs.·of··4·variables:
····$·PARTOFSPEECH··:·Factor·w/·5·levels·"adj","adv","conj",..:·
  1·2·4·3·5
····$·TOKENFREQUENCY:·int·[1:5]·421·337·1411·458·455
····$·TYPEFREQUENCY·:·int·[1:5]·271·103·735·18·37
····$·CLASS·····:·Factor·w/·2·levels·"closed","open":·2·2·2·1·1
·$·:·chr·[1:10]·"This"·"may"·"be"·"a"·...
```

An alternative way to generate such a list involves labeling the elements of the list, i.e., giving them names just like above when we gave names to elements of vectors. You can do that either when you generate the list by assignment:

```
>·a.list<-list(Part1=a.vector,·Part2=a.dataframe,·Part3=another.
  vector)¶
```

or you can do the labeling with `names` as you did with vectors above:

```
>·names(a.list)<-c("Part1",·"Part2",·"Part3")¶
```

But for now let's stick to the unlabeled format (and thus redefine `a.list` as before):

```
>·a.list<-list(a.vector,·a.dataframe,·another.vector)¶
```

This kind of data structure is interesting because it can contain many different things and because you can access individual parts of it in ways similar to those you use for other data structures. As the above output indicates, the elements in lists are numbered and the numbers are put between double square brackets. Thus:

```
>·a.list[[1]]¶
[1]··1··2··3··4··5··6··7··8··9·10
>·a.list[[2]]¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1··········adj·····------421·····------271···open
2··········adv·····------337·····------103···open
3··········n·····------1411·····------735···open
4·········conj·····------458·····------18·closed
5·········prep·····------455·····------37·closed
>·a.list[[3]]¶
[1]·"This"·····"may"·····"be"·····"a"·····"sentence"·"from"···
  ··"a"·····"corpus"···"file"·····"."
```

It is important to bear in mind that there are two ways of accessing a list's elements with subsetting. The one with double square brackets is the one suggested by how a list is shown in the console. This way, you get each element as the kind of data structure you entered into the list, namely as a vector, a data frame, and a vector respectively. What, however, happens when you use the general subsetting approach, which you know uses single square brackets? Can you see the difference in the notation with double square brackets? What is happening here?

```
>·a.list[1]¶
[[1]]
·[1]··1··2··3··4··5··6··7··8··9·10

>·a.list[2]¶
```

```
[[1]]
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1··········adj·····------421·····------271···open
2··········adv·····------337·····------103···open
3··········n·····------1411·····------735···open
4·········conj·····------458·····------18·closed
5·········prep·····------455·····------37·closed

>·a.list[3]¶
[[1]]
·[1]·"This"·····"may"·····"be"·····"a"·····"sentence"·"from"
  ·····"a"·····"corpus"···"file"·····"."
```

**THINK**

**BREAK**

The difference is this: If you access a component of a list with *double* square brackets, you get that component as an instance of the kind of data structure it was entered into the list as (see the first output). If you access a component of a list with *single* square brackets, you get that component as an instance of the data structure from which you access it, i.e., as a list. This may seem surprising or confusing, but it is actually only consistent with what you already know: If you use single square-bracket subsetting on a vector, you get a vector; if you use single square-bracket subsetting (with more than one row and column chosen) on a data frame, you get a data frame, etc. So, once you think about it, no big surprise.

Finally, if you named the elements of a list, you could also use those (but we did away with the names again, which is why you get `NULL` here):

```
>·a.list$Part1·#·or·a.list[["Part1"]]¶
NULL
```

Of course, you can also access several elements of a list at the same time. Since you will get a list as output, you use the by now familiar strategy of single square brackets:

```
>·a.list[c(1,3)]¶
[[1]]
[1]··1··2··3··4··5··6··7··8··9·10
[[2]]
[1]·"This"·····"may"·····"be"·····"a"·····"sentence"·"from"
  ·····"a"·"corpus"···"file"·····"."
```

So far we have looked at how you access *elements* of lists, but how do you access *elements of elements* of lists? The answer of course uses the notation with double square brackets

(because you now do not want a list, but, say, a vector) and either of the two following options works – the latter is more intuitive to me:

```
>·a.list[[c(1,3)]]¶
[1]·3
>·a.list[[1]][3]¶
[1]·3
```

And this is how you access more than part of a list at the same time:

```
>·a.list[[1]][3:5]¶
[1]·3·4·5
>·a.list[[1]][c(3,·5)]¶
[1]·3·5
```

Thus, this is how to handle data frames in lists such as, for instance, accessing the second element of the list, and then the value of its third row and second column:

```
>·a.list[[2]][3,2]¶
[1]·1411
>·a.list[[2]][3,2:4]¶
··TOKENFREQUENCY·TYPEFREQUENCY·CLASS
3···········1411···········735··open
```

And this is how you take the second element of the list, and then the elements in the third row and the second and fourth column:

```
>·a.list[[2]][3,c(2,·4)]¶
··TOKENFREQUENCY·CLASS
3···········1411··open
```

And this is how you delete a list part:

```
>·a.list[3]<-NULL;·a.list¶
[[1]]
·[1]··1··2··3··4··5··6··7··8··9·10
[[2]]
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1··········adj···········421···········271···open
2··········adv···········337···········103···open
3···········n···········1411··········735···open
```

```
4·····conj·····458·····18·closed
5·····prep·····455·····37·closed
```

As you can see, once one has figured out which bracketing to use, lists can be an extremely powerful data structure. They can be used for very many different things. One is that many statistical functions output their results in the form of a list, so if you want to be able to access your results in the most economical way, you need to know how lists work. A second one is that sometimes lists facilitate the handling of your data. For example, we have used the data frame `a.dataframe` in this section, and above we introduced `subset` to access parts of a data frame that share a set of variable values or levels. However, for larger or more complex analyses, it may also become useful to be able to split up the data frame into smaller parts, depending on the values some variable takes. As you saw in the previous exercise box, you use `split`, which takes as its first argument the data frame to be split up and as its second argument the name of the variable (i.e., column name) according to which the data frame is to be split. Why do I mention this here (again)? Because the result of `split` is a list:

```
>·(y<-split(a.dataframe,·a.dataframe$CLASS))¶
$closed
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
4·····conj·····458·····18·closed
5·····prep·····455·····37·closed
$open
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·····adj·····421·····271···open
2·····adv·····337·····103···open
3·····n·····1411·····735···open
```

This then even allows you to access the parts of the list using the above `$` notation:

```
>·y$open¶
··PARTOFSPEECH·TOKENFREQUENCY·TYPEFREQUENCY··CLASS
1·····adj·····421·····271···open
2·····adv·····337·····103···open
3·····n·····1411·····735···open
```

Note that you can also split up a data frame according to all combinations of two variables (i.e., more than one variable); the second argument of `split` must be a list of these variables, which of course is not the most useful thing ever with the present tiny example, given that many combinations of `a.dataframe$CLASS` and `a.dataframe$PARTOFSPEECH` are not even attested in `a.dataframe`:

```
>·split(a.dataframe,·list(a.dataframe$CLASS,·a.dataframe$
  PARTOFSPEECH))¶
```

## 3.6 Elementary Programming Issues

In order to really appreciate the advantages R has to offer over all available corpus-linguistic software, we will now introduce a few immensely important functions, control statements, and control-flow structures that will allow you to apply individual functions as well as sets of functions to data only in certain conditions and/or more often than just the single time we have so far been covering. To that end, this section will first introduce conditional expressions; then we will look at the notion of loops; finally, we will turn very briefly to a few functions which can sometimes replace loops and are often more useful because they are faster and consume less memory.

### *3.6.1 Conditional Expressions*

You will often be in the situation that you want to execute a particular (set of) function(s) if and only if some condition *C* is met, and that you would like to execute some other (set of) function(s) if *C* is not met. The most fundamental way of handling conditions in R involves a control-flow structure that can be expressed as follows:

```
if·(some logical expression testing a condition)·{¶
   what to do if this 1st logical expression evaluates to TRUE
   (this can be more than one line)¶
}·else·if·(other condition){¶
   what to do if this 2nd logical expression evaluates to TRUE
   (this can be more than one line)¶
}·else·{¶
   what to do if all logical expressions above evaluate to FALSE
   (this can be more than one line)¶
}¶
```

(Note that the structures `else·if·{·...·}` and `else·{·...·}` are optional.) The `logical expression testing a condition` represents something we have already encountered above (e.g., in the context of `which` but also elsewhere), namely an expression with logical operators that is evaluated and returns either `TRUE` or `FALSE`. If it returns `TRUE`, R will execute the first (set of) function(s); if the expression condition returns `FALSE`, R will test whether the second condition is true. If it is, it will execute the next (set of) function(s), otherwise it will execute the last (set of) function(s). Two simple examples will clarify this; recall from above the fact that the plus sign at the beginning of lines is not something you enter but the version of the prompt that R displays when it expects further input before executing, here, a series of commands:

```
>·a<-2¶
>·if·(a>2)·{¶
+····b<-TRUE¶
}·else·{¶
+····b<-FALSE¶
+·}¶
b¶
[1]·FALSE
```

and

```
>·b<-"car"¶
>·if·(b=="automobile")·{¶
+····a<-"b·is·'automobile'"¶
}·else·if·(b=="car")·{¶
+····a<-"b·is·'car'"¶
+·}·else·{¶
+····a<-"b·is·neither·'automobile'·nor·'car'"¶
+·}¶
a¶
[1]·"b·is·'car'"
```

You may wonder what the series of leading spaces in some lines are for, especially since I said above that R only cares about space within quotes. This is still correct: R does not care about these spaces, but *you* will. The only function of the spaces here is to enhance the legibility and interpretability of your small script. In the remainder of the book, every indentation by three spaces represents one further level of embedding within the script. The third line of the second example above is indented by three spaces, which shows you that it is a line within one conditional expression (or loop, as we will see shortly). When you write your own scripts, this convention will make it easier for you to recognize coherent blocks of code which all belong to one conditional expression. Of course, you could also use just two spaces, a tabstop, etc.

When do you need conditional expressions? Well, while not all of the following examples are actually best treated with `if`, they clearly indicate a need for being able to perform a particular course of action only when particular conditions are met. You may want to

- include a corpus file in your analysis only if the corpus header reveals it is a file containing spoken language;
- search a line of a corpus file only if it belongs to a particular utterance;
- use a certain part of the code only if your R instance is running on Windows, use another part of the code if your R instance is running on Linux;
- include a word in your frequency list only if its frequency exceeds a particular value; or
- want to use one search expression if a file has SGML annotation and another if it has XML annotation, etc.

You should now do "Exercise box 3.3: Conditional expressions."

**Recommendations for further study/exploration**

- On how to test an expression and specify what to do when the expression is true or false in one line: `?ifelse¶`.
- On how to choose one of several alternatives and perform a corresponding action: `?switch¶`.

### *3.6.2 Loops*

This section introduces loops, which are one useful way to execute one or more functions several times. While R offers several different ways to use loops, I will only introduce one of them here in a bit more detail, namely `for`-loops, and leave `while` and `repeat`-loops for you to explore on your own (I do provide one example for each in the code file, though).

A `for`-loop has the following structure:

```
for·(name·in·seq)·{¶
   what to do as often often as seq has elements
   (this can be more than one line)¶
}¶
```

This requires some explanation. The expression `name` stands for any name you choose to assign, and `seq` stands for anything that can be interpreted as a sequence of values (where one value is actually enough to constitute a sequence); typically, such a sequence is a vector, and very often it is a vector with integers from 1 to *n* (either generated using the range operator `:` or `seq`). Let us look at the probably simplest conceivable example, that of printing out numbers (which one would of course not really do with a loop but with `cat(1:3,·sep="\n")¶` – this is just a didactic example):

```
>·for·(i·in·1:3)·{¶
+····cat(i,·"\n")¶
+·}¶
1
2
3
```

This can be paraphrased as follows: Generate a variable called `i` as an index for looping. The value `i` should take on at the beginning is 1 (the first value in the sequence after the `in`), and then R executes the function(s) within the loop – i.e., those enclosed in curly brackets. Here, this just means printing `i` and a newline. When the closing curly bracket is reached, `i` should take on the next value in the user-defined sequence – i.e., 2 – and again perform the functions within the loop, etc. until the functions within the loop have been executed, with `i` having assumed the last value of the sequence, i.e., 3. Then, the `for`-loop is completed/exited. Again, the above kind of sequence is certainly the most frequently used one – the index `i` starts with 1 and proceeds through a series of consecutive integers – but it is by no means the only one: In the next example, `i` takes on all values of `letters[4:6]` (`letters` is a built-in constant containing the letters from a to z):

```
>·for·(i·in·letters[4:6])·{¶
+····cat(i,·"\n")¶
+·}¶
```

```
d
e
f
```

Of course, you can also use nested loops, that is, you can execute one loop within another one, but be aware of the fact that you must use different names for the looping indices (`i` and `j` in this example):

```
>·for·(i·in·1:2)·{¶
+····for·(j·in·6:7)·{¶
+·····cat(i,·"times",·j,·"is",·i*j,·"\n")¶
+····}¶
+·}¶
1·times·6·is·6
1·times·7·is·7
2·times·6·is·12
2·times·7·is·14
```

While these examples only illustrate the most basic kinds of loops conceivable (in the sense of what is done within each iteration), we will later use `for`-loops in much more useful ways.

Since you specify a sequence of iterations, `for`-loops are most useful when you must perform a particular operation a known number of times. However, sometimes you do not know the number of times an operation has to be repeated. For example, sometimes the number of times something has to be done may depend on a particular criterion, e.g., when your corpus has 200 files but you only want to search those files that contain written data, but you want R to find that out for you from the loaded corpus file's header, meaning you know what will be in the header of a file with written data, but you don't even know which and how many files those are. R offers several easy ways of handling these situations. One possibility of, so to speak, making such `for`-loops more flexible is by means of the control expression `next`. If you use `next` within a loop, R skips the remaining instructions in the loop and advances to the next element in the sequence/counter of the loop. The following is a (rather boring) example where you can see that, when the logical expression returns `TRUE`, R never reaches the line in which `i` is printed so the 2 does not get printed. In the above example of written corpus data, you would check whether the file contains written data or not – if yes, you do what you want to do with it; if not, you immediately go to the next file:

```
>·for·(i·in·1:3)·{¶
+····if·(i==2)·{¶
+·····next¶
+····}¶
+····cat(i,·"\n")¶
+·}¶
1
3
```

Another way to make loops more flexible is `break`. If you use `break` within a loop, the whole loop is exited and R proceeds with the first statement *outside* of the inner-most loop in which `break` was processed. This is how it works: When the logical expression returns `TRUE`, R processes `break` and breaks out of the `for`-loop, and never gets to print 3, 4, and 5:

```
>·for·(i·in·1:5)·{¶
+····if·(i==3)·{¶
+·······break¶
+····}¶
+····cat(i,·"\n")¶
+·}¶
1
2
```

Before we proceed, let me mention that loops slightly complicate your replicating scripts directly in the console. This is because once you enter a loop, R will not begin with the execution before the final "}" – hence the "+·" prompts – so you may not be able to easily recapitulate the inner working of the iterations. My recommendation to handle this is the following: When a loop begins like this `for·(i·in·1:3)·{¶`, but you want to proceed through the loop stepwise (for instance, in order to find and fix an error), then do *not* enter this line into R immediately. Rather, to see what is happening inside the loop, set the counter variable, `i` in this case, to its first value *manually* by writing `i<-1¶`. This way, `i` has been defined and whatever in the loop is depending on `i` having been defined as if the loop was actually happening can proceed. If you then want to go through the loop once more, but again do not yet want to do all iterations, just increment `i` by 1 manually (`i<-i+1¶`) and 'iterate' again. Once you have understood and maybe fixed the instructions in the loop and want to execute it all in one go, then just type the real beginning of the loop (`for·(i·in·1:3)·{¶`) or copy and paste from the script file. This way you will be able to understand more complex scripts more easily.

You should now do "Exercise box 3.4: Loops."

### *3.6.3 Rules of Programming*

This book cannot serve as a fully fledged introduction to good programming style (in R or in general). In fact, I have not even always shown the shortest or most elegant way of achieving a particular goal because I wanted to avoid making matters more complicated or compact than absolutely necessary (and will continue to do so). However, given that there are usually many different ways to achieve a particular goal, there are two aspects of reasonable R programming that I would like to emphasize because they will help to

- increase the speed with which some of the more comprehensive tasks can be completed;
- reduce memory requirements during processing; and
- reduce the risk of losing data when you perform more comprehensive tasks.

The first of the two aspects is concerned with capitalizing on the fact that R's most fundamental data structure is the vector, and that R can apply functions to all elements of a vector at once without having to explicitly loop over all elements of a vector individually. Thus, while conditional expressions and `for`-loops are extremely powerful control structures and you can of course make R do many things using these structures, there are sometimes more elegant ways to achieve the same objectives. Why may some ways be more elegant? First, they may be more elegant because they achieve the same objective with (many) fewer lines of code. Second, they may be more elegant because the code is easier to read. Third, they may be more elegant because they are more compatible with how R processes things and manages memory. Let us look at a few examples using the vectors from above again:

```
> PARTOFSPEECH<-c("adj", "adv", "n", "conj", "prep")
> TOKENFREQUENCY<-c(421, 337, 1411, 458, 455)
> TYPEFREQUENCY<-c(271, 103, 735, 18, 37)
> CLASS<-c("open", "open", "open", "closed", "closed")
```

Let us assume we would like to determine the overall token frequencies of open class items and closed class items in our data set. There are several ways of doing this. One is terribly clumsy and involves using a `for`-loop to determine for each element of `CLASS` whether it is "open" or "closed". Note one important thing: We know we will collect results from/during each iteration of a loop – that means the loop will contain some line in which each iteration's result is assigned to a data structure, but *that* in turn means that that data structure has to have been defined before the loop presupposes it, which is why the first line of the code below creates two what I will call *collector* data structures, numeric vectors containing 0, that will, after all iterations of the `for`-loop have been completed, contain the results:

```
> sum.clos<-0; sum.open<-0
> for (i in 1:5) {
+     current.class<-CLASS[i]
+     if (current.class=="closed") {
+         sum.clos<-sum.clos+TOKENFREQUENCY[i]
+     } else {
+         sum.open<-sum.open+TOKENFREQUENCY[i]
+     }
+ }
> sum.clos; sum.open
[1] 913
[1] 2169
```

The following is already a much better way: It requires only two lines of code and avoids a loop to access all elements of `CLASS`:

```
> sum(TOKENFREQUENCY[CLASS=="closed"])
[1] 913
> sum(TOKENFREQUENCY[CLASS=="open"])
[1] 2169
```

But the best way involves a function called `tapply`, which you have already encountered briefly in an exercise box on vectors. For now, we will discuss how this function behaves when given three arguments. The first is usually a vector to (parts of) which you would like to apply a particular function. The second is usually a vector or factor (or a list of vectors or factors) with as many elements as the first-argument vector to which you would like to apply a function. The third argument is the function you wish to apply, which can be one of many, many functions available in R (`min`, `max`, `mean`, `sum`, `sd`, `var`, etc.) as well as ones you define yourself:

```
> tapply(TOKENFREQUENCY, CLASS, sum)
closed   open
   913   2169
```

This is the way to tell R "for every level of `CLASS`, determine the `sum` of the values in `TOKENFREQUENCY`". Note again that the second argument can be a list of vectors/factors to look at more than one classifying factor at the same time.

You should now do "Exercise box 3.5: `tapply`."

Another area where the `apply`-family of functions becomes particularly useful is the handling of lists. We have seen above that lists are a very versatile data structure. We have also seen that subsetting by means of double square brackets allows us to access various parts of lists. However, there are other ways of getting information from lists that we have not dealt with so far. Let us first generate a list as an example:

```
> (another.list<-list(1, 2:3, 4:6, 7:10))
```

One useful thing to be able to do is to determine how long all the parts of a list are. With the knowledge we have so far, we would probably proceed as follows:

```
> lengths<-numeric()
> for (i in seq(another.list)) {
+     lengths[i]<-length(another.list[[i]])
+ }
> lengths
[1] 1 2 3 4
```

However, with `sapply` this is much simpler. This function takes several arguments, some of which we will discuss here. The first argument is usually a list or a vector to whose elements you want to apply a function. The second argument is the function you wish to apply. The "s" in `sapply` stands for "simplify," which means that, if possible, R coerces the output into a vector format (or matrix) format. From this you may already guess what you can write:

```
>·sapply(another.list,·length)¶
[1]·1·2·3·4
```

This means "to every element of `another.list`, apply `length`". Another great use of `sapply` allows you to access several parts of data structures (vectors, lists, etc.). For example, you may want to retrieve the first element of each of the four vectors of the list `another.list`. This is how you would do it with a loop:

```
>·first.elements<-vector()¶
>·for·(i·in·1:length(another.list))·{¶
····first.elements[i]<-another.list[[i]][1]¶
+·}¶
>·first.elements¶
[1]·1·2·4·7
```

The more elegant way uses subsetting as a function. That is to say, in the previous example, the function we applied to `another.list` was `length`. Now, we want to access a subset of elements, and we have seen above that subsetting is done with single square brackets. We therefore only need to use a single opening square bracket in place of `length` (remember this from Section 3.2.3) and then provide as a third argument the positional index, 1 in this case because we want the first element:

```
>·sapply(another.list,·"[",·1)¶
[1]·1·2·4·7
```

Note that this works even when the elements of the list are not all of the same kind:

```
>·sapply(a.list,·"[",·1)·#·use·the·list·we·generated·in·Section·3.5¶
[[1]]
[1]·1
$PartOfSpeech
[1]·ADJ··ADV··N····CONJ·PREP
Levels:·ADJ·ADV·CONJ·N·PREP
[[3]]
[1]·"This"
```

These last two examples hint at something interesting. The function `sapply` (and its sister function `lapply`) apply to the data structure given in the first argument the function called in the second argument; okay, so far nothing new. Interestingly, the third argument is provided to `sapply` (or `lapply`) but those two just 'pass it on' to the function mentioned as the second argument. Let me give a simple example:

```
>·a<-c(1,·5,·3);·b<-c(2,·6,·4);·(ab<-list(a,·b))¶
[[1]]
[1]·1·5·3
[[2]]
[1]·2·6·4

>·lapply(ab,·sort,·decreasing=FALSE)¶
[[1]]
[1]·1·3·5
[[2]]
[1]·2·4·6
```

In a way, `lapply` takes the argument `decreasing=FALSE` and 'hands it' to `sort`, which then uses this specification to sort the elements of the list `ab` in the specified way. These functions provide very elegant and powerful ways of processing lists and other data structures. We will see many applications of such and similar uses of `tapply`, `sapply`, and `lapply` below.

The second useful programming guideline is that it is often useful to split up what you want to do into (many) smaller parts and, if you do so, consider regularly saving interim results into new data structures or even into files on your hard drive that you then combine later (see Chapter 5 for several examples). While this may occasionally counteract your desire to let programs operate as quickly and memory-economically as possible – generating new data structures costs memory, saving interim results into files costs output time (hard drive access), loading them again later costs corresponding input time – it is often worth the additional effort: First, if your program crashes, then having split up your program into many smaller parts will make it easier for you to determine where and why the error occurred simply because your script/code is easier to read. This is true even for yourself: If you submit a paper and wait the customary 9–19 months that so many outlets need for reviewing, by that time you will have forgotten much of what you did and why, so you will need to decipher your code in a way that is frustratingly similar to how you would read someone else's code that you have never seen before. Second, if your program crashes, then having saved interim results into other data structures or files regularly will also decrease the amount of work you have to do again.

It is vital to realize that the way R works often strongly encourages such a modular approach. For instance, the way R handles memory is not always ideal. For instance, R is really not good at handling memory of data structures that grow (say, on each iteration of a loop). For instance, recall the last two loop examples, where we created one or two collector vectors before the loop. We did that in a way that is actually very bad practice: We made the collector vectors empty vectors and then filled, at iteration step `i`, their `i`-th slot. That means that before the first iteration the collector vector has length 0, after the first iteration (when `i` was 1), that vector has a length of 1, after the second iteration (when `i` was 2), that vector has a length of 2, and so on. That is, the vector grew a bit on each iteration, and that is something that can slow R down *very* much once the vector grows much more than just the four items we dealt with here. Here's an example: Save all your currently open files in all applications you may have open and then run the following three lines. You will see that even the simplest possible operation, just putting one number at a time into a vector, can really take a long time if you let the vector `qwe` grow dynamically:

```
>·qwe<-c();·for·(i·in·1:500000)·{¶
+····qwe[i]<-i¶
+·}¶
```

Thus, whenever you know before a loop how long the output will be, you should define the collector structure – usually a vector or a list – such that you already reserve all required output slots in advance. So try this, where we immediately define the output vector `qwe` to have a length of 500,000 elements; you can see that the exact same operation takes *much* less time:

```
>·qwe<-numeric(500000);·for·(i·in·1:500000)·{¶
+····qwe[i]<-i¶
+·}¶
```

On my laptop the first script needed three minutes and three seconds, the second needed one second. But you might wonder now "When would I ever know the length of the result in advance?" That kind of situation is actually more frequent than you might assume: yes, when you do a concordance of a word in a corpus you probably do not already know the number of necessary results slots in advance, but here are some applications where you would:

- You are interested in the frequencies of three words in a corpus: Thus, you create a collector vector with three slots – one for each word – and then loop over the corpus files, load each one, retrieve the frequency of each word in each file, and successively add the frequency of each word in the current corpus file to the summed frequency from all previous files; that is, even before you start, you know your output vector will have three elements (this is comparable to the loop of the first `tapply` example above; also see Section 5.3.1 for a somewhat similar application).
- You are interested in the lengths of corpus files both in words and in sentences. Your corpus has 4,049 files, so you know you will have one results vector for all files' lengths in words (with 4,049 slots), and another results vector for all files' lengths in sentences (with 4,049 slots) (see Section 5.2.4 for a similar application).
- You are interested in the dispersion of three words in a corpus of 4,049 files, which means you need to collect, for each of the three words, its frequency in all 4,049 files. Thus, you know that your output structure could be a list with three elements, where each of the elements is a numeric vector of length 4,049 (see Section 5.1.1).

So what do you do when you are not in a situation like this? For example, what if you wanted to generate a frequency list of a reasonably large corpus such as the 100 million-word BNC, where you do not know the exact number of word tokens and types in advance? Two possibilities: First and as briefly mentioned above, you can loop over each file, identify all word tokens in it, use `table` to generate a frequency table of it, and then save that frequency list into a separate file, which means you end up with 4,049 very small frequency list files, none of which will take up a lot of memory on its own. Then, in a second step, you do a second loop in which you load and amalgamate all 4,049 frequency list files. This strategy will cost *much* less time and memory than trying to do it all in one loop, and it is a strategy we will employ a number of times in Section 5.2.8.

A second possibility is to define a collector for the output that is so large that it will definitely be able to accommodate all results, and then fill the slots of this vector 'from left to right'. Let's pretend your collector vector is called `coll` (and it has 30 slots) and the first file generated four output figures: These four output figures then go into `coll[1:4]`, meaning `coll[5:30]` will still be empty or 0. If the second file generated three output figures, then those go into `coll[(4+1):(4+3)]`, i.e., `coll[5:7]`, and so on and so forth. Here is a small script that exemplifies this, and I again recommend that you look at this in the code file with RStudio because I have added much commentary there to explain the process in great detail, because this, too, is a strategy we will use a few times to avoid forcing R to grow vectors:

```
>·(coll<-numeric(30))¶
>·new.results.begin.here<-1¶
>·for·(i·in·1:3)·{¶
+····sample.size<-sample(10,·1)¶
+····sample.data<-sample(100,·sample.size)¶
+····coll[new.results.begin.here:(new.results.begin.here-1+
  length(sample.data))]<-sample.data¶
+····new.results.begin.here<-new.results.begin.here+
  length(sample.data)¶
+····cat("I·have·added:",·sample.size,·"elements,·namely",·sample.
  data,·";\ncoll·is·now:",·coll,·"\n\n")¶
}¶
>·coll¶
>·(coll<-coll[coll>0])¶
```

In this context, if your program involves one or more loops, it is often useful to build in a line that outputs a 'progress report' to the screen so that you can always see whether your program is still working and, if it crashes, where the crash occurred; a first example of this you see in the above script in the line beginning with `cat`.

One final recommendation in this section is concerned with naming practices. In much of the code below – in particular in Chapter 5 – I will often use quite long names for data structures etc., which is really only in the interest of recoverability or ease of parsing and recognizing things: Sometimes, names such as `qwe` or `asd` will be used because they are superfast and virtually error-free to type on an American QWERTY keyboard, but in the long run, i.e., for your real research projects, names such as `sentences.without.tags` or `cleaned.corpus.file` are infinitely more revealing than the kind of names (e.g., `qwe` or `aa`) I have mostly used so far.

**Recommendations for further study/exploration**

- On how to apply a function to user-specified margins of data structures: `?apply¶`.
- On how to apply a function to multiple list or vector elements: `?mapply¶`.
- On how to perform operations similar to `tapply`, but with a different output format and/or more flexibly: `?aggregate¶`, `?by¶`, and `?rowsum¶`.

## 3.7 Character/String Processing

So far, we have only been concerned with generating and accessing (parts of) different data structures in R. However, for corpus linguists it is more important to know that R also offers a variety of sophisticated pattern-matching tools for the processing of character strings. In this section I will introduce and explore many of these tools.

There is one extremely important point to be made right here at the beginning: KNOW YOUR CORPORA! As you hopefully recall from the first exercise box in Section 2.1, a computer's understanding of what a word is may not coincide with yours, and that of any other linguist may be different yet again. It is absolutely imperative that you know exactly what your corpus files look like – both in terms of the corpus data themselves and their annotation – which may require going over corpus documentation or, if you use unannotated texts, over parts of the corpus files themselves, in order to get to know spelling, formatting, annotation, etc.

A little example of something I came across in an undergrad corpus linguistics course was that we generated a concordance of *in* and found that in one of the SGML-annotated files of the BNC that we looked at, the word *in* was tagged as VBZ, which means "third person singular form of to be". All of us were stunned for a second, but then we noticed what was going on – can you guess?

**THINK**

**BREAK**

The answer is that there were instances of *innit* as a reduction of *isn't it* in the files, and *innit* is tagged in the BNC in SGML annotation as "<w·VBZ>in<w·XX0>n<w·PNP>it" – thus, you really must know what's in your data.

You may *now* think "But can't this be done more easily? Surely, ready-made programs wouldn't require me to do that?" Well, the answer is "No! And wrong!" There is no alternative to knowing your corpora, this cannot be done more easily, and any concordance programs that come with more refined search options also require you to thoroughly consider the format of the corpus files even if their interface 'hides' such decisions behind clickable buttons with smiling corpus linguists on them, in settings, or in .ini files. This issue will surface repeatedly below in this section and throughout all of Chapter 5.

### *3.7.1 Getting Information From and Accessing Character Vectors*

The probably most basic character operation in R gives the length of all character strings within a vector. The function `nchar` takes as its most important argument the name of a character vector containing one or more elements and outputs the number(s) of characters of all elements of the vector:

```
>·example<-c("I",·"do",·"not",·"know")¶
>·nchar(example)¶
[1]·1·2·3·4
```

**Recommendations for further study/exploration**

- On how to determine whether elements of a character vector have more than zero characters: `?nzchar¶`.

If you want to access a part of a character string, you can use `substr`, which we will always use with three arguments. The first is a character vector of which you want substrings; the second one is a numeric vector with the position(s) of the first character(s) you want to access; the third one is a numeric vector with the position(s) of the last character(s) you want to access:

```
>·substr("internationalization",·6,·13)¶
[1]·"national"
```

or

```
>·substr(example,·2,·3)¶
[1]·""···"o"··"ot"·"no"
```

From the plurals above, you probably already inferred that `substr` also handles longer vectors efficiently in the same way:

```
>·some.first.vector<-c("abcd",·"efgh")¶
>·some.other.vector<-c("ijkl",·"mnop")¶
>·substr(c(some.first.vector,·some.other.vector),·c(1,·2,·3,·4),·
  c(2,·3,·4,·4))¶
[1]·"ab"·"fg"·"kl"·"p"
```

As you can see, first `some.first.vector` and `some.other.vector` are combined into one vector with four elements, and then the four different start and end positions of characters are applied to this character vector. This can be extremely useful, for instance, when you know where in a line of annotation, say, the three-character identifier of a speaker is located (as may be the case in CHAT files from the CHILDES database) or when you want to access the line numbers in some version of the Brown corpus, which are always the first eight characters of each line, etc. (see, for example, Sections 5.4.5 and 5.4.9).

### *3.7.2 Elementary Ways to Change Character Vectors*

The most basic ways in which (vectors of) character strings can be changed involves changing all characters to lower or upper case. The names of the relevant functions are `tolower` and `toupper` respectively, which both just take the relevant character vectors as their arguments; we will use this for case-insensitive frequency lists, for example:

```
>·tolower(example)¶
[1]·"i"····"do"···"not"··"know"
>·toupper(example)¶
[1]·"I"····"DO"···"NOT"··"KNOW"
```

Another elementary way to change (vectors of) character strings involves replacing *x* characters in (elements of) character vectors by *x* other characters. The name of the function is `chartr`, and it takes three arguments. First, the character(s) to be replaced; second, the character(s) that are substituted (this needs to have as many characters as the first argument); third, the character vector to which the operation is applied. This can be very useful, for example, to perform comprehensive transliteration operations in a single line of code or, as we will do in Section 5.4.3 below, recode a phonemic transcription into a segmental one distinguishing just vowels and consonants:

```
>·chartr("no",·"pq",·example)¶
[1]·"I"····"dq"···"pqt"··"kpqw"
```

### *3.7.3 Merging/Splitting Character Vectors Without Regular Expressions*

If you want to merge several character strings into one, you can use `paste`. These are the default arguments `paste` takes and their settings (where applicable):

- one or more character vectors;
- `sep="·"`: the character string(s) used to separate the different character strings when they are merged – the space is the default;
- `collapse=NULL`: the character string used to separate the different character strings when they are merged into just a single character string.

The use of `paste` can sometimes be a bit confusing to beginners, but once you think through the examples and explanations provided below, the way paste works *does* make sense, and the good news is that we will only use one of the six examples really often. It is useful to distinguish three different scenarios, which differ with regard to what you are pasting together: (1) multiple one-element vectors, (2) one multiple-element vector, or (3) multiple multiple-element vectors.

In scenario (1), `sep` and `collapse` don't make a difference; here, we paste together four one-element vectors; we will occasionally use the first of the two examples below to paste together a file name into which we save results:

```
>·paste("I",·"do",·"not",·"know",·sep="·")¶
[1]·"I·do·not·know"
>·paste("I",·"do",·"not",·"know",·collapse="·")¶
[1]·"I·do·not·know"
```

In scenario (2), we paste together one vector that has four elements and here `collapse` makes sure that all the vectors' elements get merged into a single character string; the

second one of these two examples is something we will do frequently (when we load corpus files line by line and then paste them together into one very long string so that we can then break them up where *we* want them to be split up):

```
>·qwe<-c("I",·"do",·"not",·"know")¶
>·paste(qwe,·sep="·")¶
[1]·"I"····"do"···"not"··"know"
>·paste(qwe,·collapse="·")¶
[1]·"I·do·not·know"
```

Finally, in scenario (3), we paste together two multiple-element vectors, and again what `collapse` does is make sure everything is merged into a single output string:

```
>·paste(qwe,·qwe,·sep="·")¶
[1]·"I·I"·····do·do"·····"not·not"···"know·know"
>·paste(qwe,·qwe,·collapse="·")¶
[1]·"I·I·do·do·not·not·know·know"
```

The 'opposite' function of paste is `strsplit`, with which you can split up a character string into several character strings. Its first argument is a character vector to be split up; the second argument is the character string that `strsplit` will delete to split up the first argument:

```
>·asd<-"I·do·not·know"¶
>·strsplit(asd,·"·")¶
[[1]]
[1]·"I"····"do"···"not"··"know"
```

Note that if you use an empty character string as the second argument, R will split up the first argument character by character, which can be very useful to immediately see all characters that are attested in a file (especially when combined with `unlist` and `table`, see below):

```
>·strsplit(asd,·"")¶
[[1]]
·[1]·"I"·"·"·"d"·"o"·"·"·"n"·"o"·"t"·"·"·"k"·"n"·"o"·"w"
```

It is important to note that the output of `strsplit` is *not* a vector – it is a *list* of, here, one element, which contains one vector (as you can infer from the double square brackets). The same works for larger character vectors: The list will have as many parts as `strsplit`'s first argument has parts, which can be useful if you want to split up `strsplit`'s first argument but nevertheless retain the information regarding where in the input each element of the output came from:

```
>·zxc<-c("This·is·the·first·character·string",·
  "This·is·the·second·character·string")¶
>·strsplit(zxc,·"·")¶
[[1]]
[1]·"This"·····"is"·····"the"·····"first"····"character"·
  "string"
[[2]]
[1]·"This"·····"is"·····"the"·····"second"···"character"·
  "string"
```

If you do *not* need to know where each element in the output came from, i.e., you just want one vector of all resulting items, just add `unlist` to change the list into one long(er) vector:

```
>·unlist(strsplit(zxc,·"·"))¶
[1]·"This"·····"is"·····"the"·····"first"····"character"·
  "string"···"This"·····"is"·····"the"·····"second"····
  "character"·"string"
```

This will probably make you realize that this is close to how we will create frequency lists. If `zxc` was loaded from a corpus file, then this is how you create an alphabetically sorted or a frequency-sorted frequency list respectively:

```
>·table(unlist(strsplit(zxc,·"·")))¶
character····first·····is···second···string·····the·····This
·····2·····1·····2·····1·····2·····2·····2
>·sort(table(unlist(strsplit(zxc,·"·"))),·decreasing=TRUE)¶
character·····is···string·····the·····This····first···second
·····2·····2·····2·····2·····2·····1·····1
```

While these examples, in which character strings are split up on the basis of, say, just one space, are straightforward, `strsplit` is much more powerful because it can use regular expressions. However, before we introduce regular expressions, we will first treat some other functions used for searching and replacing character strings.

### *3.7.4 Searching and Replacing Without Regular Expressions*

In this section we will introduce several pattern-matching functions in a very basic manner, i.e., without using the powerful regular expression techniques involving character classes and placeholders/wildcards. To that end, let us first generate a vector of character strings:

```
>·txt<-c("This·is·a·first·example·sentence.",·"And·this·is·a·second·
  example·sentence.")¶
```

The first function is called `grep`. If it is executed without invoking its regular expression capabilities, its most basic form requires only two arguments: a one-element character vector to search for and a character vector (or something that can be coerced into a character vector, such as a factor) in which the first argument is to be found. The function returns the *positions of the elements* in which the string searched for occurs *at least one time*:

```
>·grep("second",·txt)¶
[1]·2
```

This output illustrates two properties of `grep` that beginners sometimes do not expect because (1) when you use "*Find:* . . . " in a word processor you don't get, say, the page number where something is attested, but the exact thing you were looking for, and (2) beginners have the nasty habit to think too much like humans, a problem we will encounter with regular expressions a lot. As for (1), in this most basic form, `grep` does not return the match like a word processor does, but the position of the match. However, if you still remember what we said about subsetting in Section 3.2.3, you should be able to figure out a way to arrive at the complete elements in which matches were found (which are still not the exact matches!).

**THINK**

**BREAK**

```
>·txt[grep("second",·txt)]¶
[1]·"And·this·is·a·second·example·sentence."
```

However, this result will usually be produced slightly differently. Unless specified otherwise, `grep` assumes a default setting for a third argument, `value=FALSE`, which means that only the positions are returned. But if you set `value=TRUE`, you will get the elements of the character vectors in which the matches occurred instead of just their position numbers/indices:

```
>·grep("second",·txt,·value=TRUE)¶
[1]·"And·this·is·a·second·example·sentence."
```

Second, R does of course not know what a word is: It searches for character strings and returns the positions of the matches or the matches themselves, irrespective of whether what is searched for corresponds to what a user considers a word or not:

```
>·grep("eco",·txt,·value=TRUE)¶
[1]·"And·this·is·a·second·example·sentence."
```

Third, remember that `grep` returns the position of matches in the vector that was searched maximally once (just like `match` returns only the position of the first match, not all of them). The following line returns "1" and "2" only once each, although "is" occurs twice in both first lines, once in "This" or "this" and once in "is". Thus, `grep` just answers the question "Does something occur somewhere at all (i.e., at least once)?", not "Does something occur somewhere at all and how often?"

```
>·grep("is",·txt)¶
[1]·1·2
```

If you set `grep`'s argument `invert` from its default of `FALSE` to `TRUE`, then you get the positions/elements where what you searched for does *not* occur:

```
>·grep("second",·txt,·value=TRUE,·invert=TRUE)¶
·[1]·"This·is·a·first·example·sentence."
```

Also there is a useful function called `grepl`, which returns a `TRUE` or `FALSE` for every element of the vector searched depending on whether the search expression is attested in the vector searched or not, which can be useful for later subsetting or conditional expressions; thus, the following shows that the first element of `txt` does not contain "eco" but the second one does:

```
>·grepl("eco",·txt)¶
[1]·FALSE··TRUE
```

Note that in all these cases, `grep` returns the positions of matches in the vector that was searched – e.g., the character string "second" occurs in the second character string of the vector `txt` – it also does *not* return the positions of the matches in the elements of the vector, i.e., the fact that the character string "second" begins at character position 15 in the second element of the vector `txt`. However, this information can be retrieved with a different function: `gregexpr`. This function is one of the most powerful search functions in the default installation of R. It takes the same two arguments as the most basic form of `grep`, but its output is different: It returns a list with the starting positions and lengths of *all* matches at the same time (the companion function `regexpr` only finds the first matches). This list has as many vectors as the data structure that was searched has vectors (recall that behavior from `strsplit`?), and each of these vectors contains positive integers which correspond to the starting positions of all matches. In addition, each of these vectors comes with a `match.length` attribute that provides the lengths of the matches (and a

`useBytes` attribute that is not relevant right now). It is important that you understand this hierarchical arrangement: Again, the list returned by `gregexpr` has vectors (one for each element of the vector that was searched), and each of the vectors in the list has attributes; the fact that the attributes are attributes of the vector, not of the list, are represented here by the indentation:

```
>·gregexpr("e",·txt)¶
[[1]]
[1]·17·23·26·29·32          ←—— Vector with starting positions of matches
attr(,"match.length")            for the first element of text
[1]·1·1·1·1·1               ←—— Attribute of that vector with the lengths of
                                 matches
attr(,"useBytes")           ←—— Attribute of that vector indicating byte-by-
[1]·TRUE                         byte matching

[[2]]
[1]·16·22·28·31·34·37       ←—— Vector with starting positions of matches
attr(,"match.length")            for the second element of text
[1]·1·1·1·1·1·1             ←—— Attribute of that vector with the lengths of
                                 matches
attr(,"useBytes")           ←—— Attribute of that vector indicating byte-by-
[1]·TRUE                         byte matching
```

If there is no match of the search expression in an element of the vector being searched, `gregexpr` returns vectors of –1 for those:

```
>·gregexpr("y",·txt)¶·#·output·not·shown·here
```

But now how do we access the components of the output of `gregexpr`? Let's first assign the output of `gregexpr` to a data structure:

```
>·ges.output<-gregexpr("e",·txt)¶
```

To extract the numeric vector that contains the starting positions of all matches for the first element of `txt`, you can just use the familiar double square-brackets notation, and even if the output is voluminous, the result is really still just a simple numeric vector, just one with attributes:

```
>·ges.output[[1]]¶
[1]·17·23·26·29·32
attr(,"match.length")
```

```
[1]·1·1·1·1·1
attr(,"useBytes")
[1]·TRUE
>·is.numeric(ges.output[[1]])
[1]·TRUE
```

The above suggests two more things: you can use `length` to find out how many matches (of "e") are in `txt[1]`:

```
>·length(ges.output[[1]])¶
[1]·5
```

and you can `unlist` the list from `gregexpr` to find out how many matches there are in all of `txt`:

```
>·length(unlist(ges.output))¶
[1]·11
```

Now, how do you get at the attributes (because you want to know how long the matches are)? There is a function, `attributes`, to access attributes of objects, so you might think the following will do the trick – but why doesn't it?

```
>·attributes(ges.output)¶
NULL
```

**THINK**

**BREAK**

This doesn't work because the attributes are attributes *not* of the list but of the elements of the list. Therefore, this is how you do it:

```
>·attributes(ges.output[[1]])¶
$match.length
[1]·1·1·1·1·1
$useBytes
[1]·TRUE
```

Which in turn means you can now access the attributes of this vector separately using either bracketing or their names:

```
>·attributes(ges.output[[1]])[1]¶
$match.length
[1]·1·1·1·1·1
>·attributes(ges.output[[1]])$match.length¶
[1]·1·1·1·1·1
```

An alternative to `attributes` is the function `attr`, whose first argument is the data structure whose attributes you want to access and whose second argument is a character string with the desired attribute name:

```
>·attr(ges.output[[1]],·"match.length")¶
[1]·1·1·1·1·1
```

Thinking back to Section 3.6, can you write code that accesses the lengths of all 11 matches in `txt` in one go?

**THINK**

**BREAK**

You can use `sapply` as follows, either to get a list with all lengths per match separately (the first line of code) or immediately unlist this to get all lengths of matches in a single vector (the second line of code):

```
>·sapply(ges.output,·attr,·"match.length")¶
[[1]]
[1]·1·1·1·1·1

[[2]]
[1]·1·1·1·1·1·1

>·unlist(sapply(ges.output,·attr,·"match.length"))¶
·[1]·1·1·1·1·1·1·1·1·1·1·1
```

Thus, you can also use `sapply` to count how many matches there are in each element of `txt`:

```
>·sapply(ges.output,·length)¶
[1]·5·6
```

Now everything is in place to get not just the `TRUE`s or `FALSE`s indicating whether there is at least one match (as with `grepl`), not just the positions where at least one match was found (as with `grep(...,·value=FALSE)`), not just the whole elements where at least one match was found (as with `grep(...,·value=TRUE)`), and not just the starting positions

(and lengths, as with `gregexpr`), but – finally – the exact matches. I will first show you the painful way (so that you learn about its logic) and then two much easier ways.

The painful way uses `gregexpr`'s output to retrieve the starting points of all matches in all elements of `txt` and then also the lengths of all matches to compute their end points, because then, once we have starting points and end points, we can use `substr` to extract the relevant matches from the original input vector `txt`. However, the pain doesn't end here: When we do that – use `substr` and get the starting and end points from `gregexpr`, we also have to make sure that we get the first argument of `substr` right, the input vector: we cannot give `substr` just `txt` as its first argument because in the present case we need the first element of `txt` five times (for the five matches in it and, therefore, the five starting and end points we will get from `gregexpr`), but we also need the second element of `txt` six times (for the six matches in it). Thus, this is what we need to do (make sure you look at this in the code file, which provides a lot of commentary and visually helpful indentation):

```
>·substr(rep(txt,·sapply(ges.output,·length)),¶
+····unlist(ges.output),¶
+····unlist(ges.output)+¶
  unlist(sapply(ges.output,·attr,·"match.length"))¶
  -1)¶
·[1]·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"
```

If that isn't painful, what is? While it is useful to understand this code well, let us now turn to the less painful ways of getting the exact matches. First, there is a function in R called `regmatches`, which can take three arguments: first, a character vector with the input data; second, an object created by `gregexpr` with match data (usually from the same input data of course); third, `invert=FALSE` (the default) or `TRUE`. Thus, we can easily do this now:

```
>·ges.output<-gregexpr("e",·txt)·#·reminder·for·you¶
>·regmatches(txt,·ges.output)¶
[[1]]
[1]·"e"·"e"·"e"·"e"·"e"

[[2]]
[1]·"e"·"e"·"e"·"e"·"e"·"e"
>·unlist(regmatches(txt,·ges.output))¶
[1]·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"
```

In fact, we will later create our own function `just.matches`, which makes this even easier.

The second less painful way involves a function that I wrote for this book – a revision of the corresponding function in the first edition of this book. This function is called `exact.matches.2` and you can use it once you have sourced it into R by running the following line and then choosing the file <_qclwr2/_scripts/exact.matches.2.r>:

```
>·source(file.choose())¶
```

The function `exact.matches.2` requires minimally two arguments, but can take many more, and those first two arguments are the same that `grep` and `gregexpr` require: (1) a search expression and (2) a (corpus) character vector in which to find the search expression. If that's all you provide, this is how it works; I only show the most important first four components of the list that `exact.matches.2` generates as output (if you run this without `[1:4]`, you will see the final component as well):

```
>·exact.matches.2("is",·txt)[1:4]¶
$`Exact·matches`
[1]·"is"·"is"·"is"·"is"

$`Locations·of·matches·(lines)`
[1]·1·1·2·2

$`Proportion·of·non-empty·corpus·parts·with·matches`
[1]·1

$`Lines·with·delimited·matches`
[1]·"Th\tis\tis·a·first·example·sentence."·····"This\tis\ta·first·
  example·sentence."·····
[3]·"And·th\tis\tis·a·second·example·sentence."·"And·this\tis\
  ta·second·example·sentence."
```

With all default settings, the first component contains the exact matches as if we had generated them with `unlist(regmatches(gregexpr(...),·...))` as above – if you add the argument `vectorize=FALSE` to the `exact.matches.2` call, then this first component will be a list, i.e., the same as `regmatches(gregexpr(...),·...))`.

The second component is a numeric vector which states in which parts of the input vector there were matches (and how many): Above we can see that both the first and the second part of `txt` contain two instances if "is". The third component lists the percentage of corpus parts containing at least one match, which here amounts to 100 percent.

The fourth component was the main reason to write this function: It returns for each match the corpus element in which it was found but also separates the match from its preceding and subsequent contexts with a tabstop (so that, if you print the content of `exact.matches.2(...)[[4]]` into a. txt or. csv file, you get a nice three-column output with the context preceding a match in a first column, the match in a second, and the context following a match in a third. If you set the argument `lines.around` to a number greater than zero, then you increase the preceding and subsequent context by that number of corpus elements. If you set the argument `characters.around` to a number greater than zero, then the preceding and subsequent contexts will be as many characters (as opposed to corpus elements/lines). You can also suppress the generation of this fourth component of `exact.matches.2` (if you're only interested in the first component, for example, and want to avoid the processing burden of computing potentially very many tab-delimited matches) by setting an argument `gen.conc.output=FALSE`. Also, note that by default this function deletes spaces around the column-delimiting tabstops – if you do not want that, set `clean.up.spaces=FALSE`; check out `formals(exact.matches.2)¶` for additional arguments.

Thus, with all this, we can do our searches for "e" in `txt` with `regmatches` etc. as above, or with `exact.matches.2` as follows; either is a huge improvement over the manual and superlong `substr(...)` approach:

```
>·exact.matches.2("e",·txt)[[1]]¶
·[1]·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"·"e"
>·exact.matches.2("e",·txt,·vectorize=FALSE)[[1]]¶
[[1]]
[1]·"e"·"e"·"e"·"e"·"e"

[[2]]
[1]·"e"·"e"·"e"·"e"·"e"·"e"·"e"
```

All of these functions so far have only accessed (parts of) character vectors, but we have not changed the character vectors themselves yet. However, for many cases this is of course exactly what we want to do. For example, we might want to delete all tags from a corpus file, we might want to change all numbers in a corpus file into one overall code (like "NUM"), etc. In R, the function for substitutions is called `sub`, but just like `regexpr`, `sub` only performs substitutions on the first match it finds per searched element, which is why we will immediately restrict our attention to the global substitution function `gsub`. The following are the main arguments of `gsub` and their default settings (again first without regular expressions, which will be introduced below):

```
gsub(pattern,·replacement,·x,·ignore.case=FALSE)¶
```

The first argument is the expression that you want to replace, the second is the one you want to substitute instead. The third argument is the character vector you want to search and modify. The fourth argument specifies whether the search to be conducted will be case-sensitive (`ignore.case=FALSE`) or not (`ignore.case=TRUE`). Let us apply this function to our small character vector `txt`. For example, we might want to change the indefinite determiner "a" to the definite determiner "the". By now, you should already know that the following will not really work well and why – don't run it, just think about it first:

```
>·gsub("a",·"the",·txt)¶
```

**THINK**

**BREAK**

This will not work because with this function call we would assume that R 'knows' that we are only interested in changing "a" to "the" when "a" is a word, the indefinite determiner. But of course R does not know that:

```
>·gsub("a",·"the",·txt)¶
[1]·"This·is·the·first·exthemple·sentence."
[2]·"And·this·is·the·second·exthemple·sentence."
```

Thus, we need to be more explicit. What is the most primitive way to do this?

```
>·gsub("·a·",·"·the·",·txt)¶
[1]·"This·is·the·first·example·sentence."
[2]·"And·this·is·the·second·example·sentence."
```

Note, however, that R just outputs the result of `gsub` – R has not changed `txt`:

```
>·txt¶
[1]·"This·is·a·first·example·sentence."
[2]·"And·this·is·a·second·example·sentence."
```

Only if we assign the result of `gsub` to `txt` again or some other vector will we be able to access the vector with its substitutions:

```
>·(txt.2<-gsub("·a·",·"·the·",·txt))¶
[1]·"This·is·the·first·example·sentence."
[2]·"And·this·is·the·second·example·sentence."
```

While all these examples are probably straightforward to the point of being self-evident, the real potential of the functions discussed above only emerges when they are coupled with regular expressions – i.e., placeholders/wildcards – to which we will now turn.

### *3.7.5 Searching and Replacing With Regular Expressions*

In this section, we introduce the most important aspects of regular expressions, which will be the most central tool in most of the applications to be discussed below. R offers two types of regular expressions, extended and Perl-compatible regular expressions – we will only be concerned with the latter because Perl is already widely used so chances are that you can apply what you learn here in different contexts. Thus, in order to maximally homogenize the explanations below, all regular expressions from now on will be written with the argument `perl=TRUE` even if that may not be necessary in all cases (and may sometimes slow code down a bit, but typically only with fairly complex regular expressions).

Let us begin with expressions that specify (or 'anchor') the position of character strings within character strings. The caret "`^`" and the "`$`" specify the beginning and the end of a character string. Thus, the following function only matches the first character string in the vector `txt` since only this one begins with a "t" or, `ignore.case=TRUE`, a "T":

```
>·grep("^t",·txt,·ignore.case=TRUE,·perl=TRUE,·value=TRUE)¶
[1]·"This·is·a·first·example·sentence."
```

Note that these expressions do not match any particular character in the string that is being searched: "`^`" does not match the first character of the line – it, so to speak, matches

the beginning of the line *before* the first character: You can see that we still provided the "t" in the search string, which *is*, and maps onto, the first character. Thus, these expressions are also referred to as zero-width tests and we say that the caret does not *consume* a character, which means, for instance, that it is not a character that the regex engine would replace:

```
>·gsub("^t",·"_T_",·txt,·ignore.case=TRUE,·perl=TRUE)¶
[1]·"_T_his·is·a·first·example·sentence."
[2]·"And·this·is·a·second·example·sentence."
```

The next expressions we deal with are not zero-width tests – they do match and consume characters. The period ".", for example, means "any single character except for the newline".[3] Thus, for example, if you want to find all occurrences of "s" which are followed by one other character and then a "c", you could enter the following into R:

```
>·grep("s.c",·txt,·ignore.case=TRUE,·perl=TRUE,·value=TRUE)¶
[1]·"And·this·is·a·second·example·sentence."
```

And of course, if you want a larger but fixed number of intervening characters, you just use as many periods as necessary:

```
>·grep("f...t",·txt,·ignore.case=TRUE,·perl=TRUE,·value=TRUE)¶
[1]·"This·is·a·first·example·sentence."
```

But now this raises the question of what to do when we want to look for a period, i.e., when we do not want the period as a metacharacter but literally. Obviously, we cannot take the period as such because the period stands for "any one character but the newline", which means we would get many more matches than we want. Thus, we need to know two things: First, a character that tells R "do not use a particular character's wildcard behavior but rather the literal character as such". These are called *escape characters* and in R this escape character is the backslash "\". Thus, you write:

```
>·gsub("\\.",·"!",·txt,·perl=TRUE)¶
[1]·"This·is·a·first·example·sentence!"
[2]·"And·this·is·a·second·example·sentence!"
```

Now, why do we need two backslashes? The answer is that the two backslashes are actually just one character, namely the escape character followed by an actual backslash (as you can easily verify by entering `nchar("\\");·cat("\\")¶` into R). Thus, the two backslashes are in fact just one character – to the R interpreter, the first one instructs R to treat the second as a literal backslash (rather than part of some other sequence of characters), and the second one then says "I really want a period literally, don't use it as a regex!" but when you print them, you really only see one.

Second, we need to know which characters need to be escaped. The following is a list of such characters and their non-literal meanings, all of which will be used further below.

Line anchors

| | |
|---|---|
| `^` | the beginning of a character string (but see below) |
| `$` | marks the end of a character string |

Quantifying expressions

| | |
|---|---|
| `*` | zero or more occurrences of the preceding regular expression |
| `+` | one or more occurrences of the preceding regular expression |
| `?` | zero or one occurrences of the preceding regular expression |
| `{` and `}` | ranges, mark the beginning and the end of a quantifying expression |

Other expressions

| | |
|---|---|
| `.` | any one character but the newline |
| `\` | the escape character |
| `(` and `)` | mark the beginning and the end of a (part of a) regular expression to be treated as a single element |
| `\|` | or |
| `[` and `]` | mark the beginning and the end of a character class |
| `"` | a double quote needs to be escaped if you use double quotes to delimit your character vector that makes up the regular expression |
| `'` | a single quote needs to be escaped if you use single quotes to delimit your character vector that makes up the regular expression (which we don't do) |

While the period stands for "any one character apart from the newline", you may want to do more than retrieve exactly one kind of match. The above list already shows some of the quantifying expressions you can use. For example, if you have a vector in which both the British and the American spellings of *colour* and *color* occur and you would like to retrieve both spelling variants, you can make use of the fact that a question mark means "zero or one occurrence of the immediately preceding expression". Thus, you can proceed as shown here:

```
>·colors<-c("color",·"colour")¶
>·grep("colou?r",·colors,·perl=TRUE,·value=TRUE)¶
[1]·"color"··"colour"
```

And with what search expression would you find both the singular and the plural of both spellings? With this one: "colou?rs?" of course.

Another domain of application of such quantifying expressions would be to clean up text files by, for example, changing all sequences of more than one space to just one space. How would you do this?

**THINK**

**BREAK**

Well, you could write:

```
>·some.text.line<-"This··is····just·one····example."¶
>·gsub("··+",·"·",·some.text.line,·perl=TRUE)¶
[1]·"This·is·just·one·example."
```

As a matter of fact, using the question mark and the plus sign are just two special shortcuts for a more general notation which allows you to specify, for example:

- the exact number of matches *m* of an expression you would like to match: just add `{m}` to an expression;
- the minimum number of matches *m* of an expression you would like to match (by leaving open the maximum): just add `{m,}` to an expression;
- the minimum number of matches *m* and the maximum number of matches *n* of an expression you would like to match: just add `{m,n}` to an expression.

Thus, the application of the question mark above is actually just a shorthand for an expression involving ranges. So, how could you rewrite the above line `grep("colou?r",·colors,·perl=TRUE,·value=TRUE)¶` using a (in this case clumsy) range expression?

**THINK**

**BREAK**

This would be it:

```
>·grep("colou{0,1}r",·colors,·perl=TRUE,·value=TRUE)¶
[1]·"color"··"colour"
```

It is important here to clarify the range to which these quantifying expressions apply. From the previous examples, it emerges that the quantifiers always apply only to the immediately preceding character – you have to resist the human impulse to apply the quantifier to, say, "colou" – the `{0,1}` really only applies to the "u". However, sometimes you may want to quantify something that consists of more than one element. The way to do this in R has already been hinted at above: You use opening and closing parentheses to group elements (characters and/or metacharacters). Thus, for example, if you want to match two or three occurrences of "st", you could do this:

```
>·some.vector<-c("a·st·b",·"a·ststst·b",·"a·st·st·b")¶
>·grep("(st){2,3}",·some.vector,·perl=TRUE,·value=TRUE)¶
[1]·"a·ststst·b"
```

As you can see, the expression in parentheses is treated as a single element, which is then quantified as specified. Of course, such an expression within parentheses can itself contain metacharacters, further parentheses, etc.:

```
>·grep("(.t){2,3}",·some.vector,·perl=TRUE,·value=TRUE)¶
[1]·"a·stst·b"
```

Note that what the period matches the first time around does not have to be the same character for the second (and additional matches). You can see here that, in the second element, the period first matches an "s" and then an "f":

```
>·grep("(.t){2,3}",·c("q·stst·w",·"a·stft·a"),·perl=TRUE,·
  value=TRUE)¶
[1]·"q·stst·w"
[2]·"a·stft·a"
```

In fact, one of the probably more frequent domains in which the notation with parentheses is applied involves providing R with several alternative expressions to be matched. To do that, you simply put all alternatives in parentheses and separate them with a vertical bar (or pipe) |:

```
>·gsub("a·(first|second)",·"another",·txt,·
  ignore.case=TRUE,·perl=TRUE)¶
[1]·"This·is·another·example·sentence."
[2]·"And·this·is·another·example·sentence."
```

As another example, imagine that you want to replace all occurrences of "a", "e", "i", "o", and "u" by a "V" (for vowel). You could do this as follows:

```
>·gsub("(a|e|i|o|u)",·"V",·txt,·ignore.case=TRUE,·perl=TRUE)¶
[1]·"ThVs·Vs·V·fVrst·VxVmplV·sVntVncV."
[2]·"Vnd·thVs·Vs·V·sVcVnd·VxVmplV·sVntVncV."
```

But since this kind of application – the situation where one wishes to replace one out of several individual characters with something else – is a particularly frequent one, there is actually a shorter alternative notation which uses square brackets without "|" to define a class of individual characters and replace them all in one go:

```
>·gsub("[aeiou]",·"V",·txt,·ignore.case=TRUE,·perl=TRUE)¶
[1]·"ThVs·Vs·V·fVrst·VxVmplV·sVntVncV."
[2]·"Vnd·thVs·Vs·V·sVcVnd·VxVmplV·sVntVncV."
```

(I am of course hoping that you are thinking, wait a second, [cw]ouldn't I do this with `chartr`? The answer is yes, see the code file.) It is also possible to define such character classes in square brackets using the minus sign as a range operator. For example, if you want to replace any occurrence of the letters "a" to "h" and "t" to "z" by an "X", you can do this as follows:

```
>·gsub("[a-ht-z]",·"X",·txt,·ignore.case=TRUE,·perl=TRUE)¶
[1]·"XXis·is·X·XirsX·XXXmplX·sXnXXnXX."
[2]·"XnX·XXis·is·X·sXXonX·XXXmplX·sXnXXnXX."
```

Note that the minus sign is now not used as part of a character class: "[a-z]" does not mean "match either 'a' or '-' or 'z'" – it means "match any of the letters from 'a' to 'z'." If you want to include the minus sign in your character class, you can put it at the first position within the square brackets. Thus, to match either "a", "-", or "z", you write "[-az]" or "[-za]".

The same pattern works for numbers. For example, you could replace the numbers from 1 to 5 in the string "0123456789" with "3" as follows:

```
>·gsub("[1-5]",·"3",·"0123456789",·perl=TRUE)¶
[1]·"0333336789"
```

You should now do "Exercise box 3.6: A few regular expressions."

For cases in which you would like to match every character that does *not* belong to a character class, you can use the caret "^" as the first character within square brackets.

```
>·gsub("[^1-5]",·"3",·"0123456789",·perl=TRUE)¶
[1]·"3123453333"
```

Note how the caret within the definition of a character class in square brackets means something else (namely, "not") than the caret outside of a character class (where it means "at the beginning of a string"); this is one of two regular expression characters that has a different meaning in a certain environment (here, in square brackets) than in another (at the beginning of a string).

While it is of course possible to define all alphanumeric characters or all numbers as character classes in this way ("[a-zA-Z0-9]" and "[0-9]" respectively), R offers a few predefined character classes. One such character class, written as "\\d" in R (see above on double backslashes), is equivalent to "[0-9]", i.e., all digits:

```
>·gsub("\\d",·"3",·"a1b2c3d4e5",·perl=TRUE)¶
[1]·"a3b3c3d3e3"
```

The opposite of this character class – i.e., what one might want to write as "[^\\d]" – can be expressed as "\\D":

```
>·gsub("\\D",·"3",·"a1b2c3d4e5",·perl=TRUE)¶
[1]·"_1_2_3_4_5"
```

Another predefined character class which will turn out to be very useful is that of word characters, written as "\\w" in R. This character class is the short form of what you could define more lengthily as "[a-zA-Z0-9]" (but many Unicode characters from other scripts can also be included), and again "\\W" is its opposite. From what you know by now, it should be obvious that you can combine this character class with a plus sign to look for what will, in the case of the Latin alphabet, correspond to most words, at least words without apostrophes and hyphens in them:

```
>·gsub("\\w+",·"WRD",·txt,·perl=TRUE)¶
[1]·"WRD·WRD·WRD·WRD·WRD·WRD."·····"WRD·WRD·WRD·WRD·WRD·WRD·WRD."
```

We will use this character class frequently below. Then, there is the class written as "\\s", which stands for all whitespace characters: tabstops (otherwise written as "\t"), spaces (otherwise written as " ", or here "·"), newlines (otherwise written as "\n"), and carriage returns (otherwise written as "\r"). As before, changing "\\s" into "\\S" gives you the opposite character class.

The final such metacharacter to be mentioned here is again an anchor, a zero-width test (just like the line anchors "^" and "$" mentioned above). The expression "\\b" refers to a word boundary, which is the position between a word character and a non-word character (as defined above and in either order); again "\\B" is the opposite. That is to say, "\\b" is the position between the two characters of "\\w\\W" or "\\W\\w", but includes, or consumes, neither "\\w" nor "\\W"! Note the differences between the first two searches, where what you search for is consumed and thus replaced, and the third one using the zero-width pattern, which does not consume, and thus not replace, anything:

```
>·gsub("\\w\\W",·"<WB>",·txt,·perl=TRUE)¶
[1]·"Thi<WB>i<WB><WB>firs<WB>exampl<WB>sentenc<WB>"
[2]·"An<WB>thi<WB>i<WB><WB>secon<WB>exampl<WB>sentenc<WB>"

>·gsub("\\W\\w",·"<WB>",·txt,·perl=TRUE)¶
[1]·"This<WB>s<WB><WB>irst<WB>xample<WB>entence."
[2]·"And<WB>his<WB>s<WB><WB>econd<WB>xample<WB>entence."

>·gsub("\\b",·"<WB>",·txt,·perl=TRUE)¶
[1]·"<WB>This<WB>·<WB>is<WB>·<WB>a<WB>·<WB>first<WB>·<WB>example
  <WB>·<WB>sentence<WB>."
[2]·"<WB>And<WB>·<WB>this<WB>·<WB>is<WB>·<WB>a<WB>·<WB>second
  <WB>·<WB>example<WB>·<WB>sentence<WB>."
```

There are two important potential problems that we have so far simply glossed over. The first of these is concerned with what R does when there are several ways to match a pattern. Let's try it out:

```
>·gregexpr("s.*s",·txt[1],·perl=TRUE)¶
[[1]]
[1]·4
attr(,"match.length")
[1]·22
```

The answer is clear. We ask R to match any "s" followed by zero or more occurrences of any character followed by another "s". There would be many ways to match this in `txt[1]`. In the following lines, the possible matches for the first string of the vector `txt` are underlined, and the first two lines give the number of the position within the character string:

```
position   0000000001111111111222222222233333
position   1234567890123456789012345678901234
           This·is·a·first·example·sentence.
           This·is·a·first·example·sentence.
           This·is·a·first·example·sentence.
           This·is·a·first·example·sentence.
           This·is·a·first·example·sentence.
           This·is·a·first·example·sentence.
```

Obviously, R returns the third solution: It begins to match at the first occurrence of "s" – at position 4 – and then chooses the longest possible match, as we can also see in the following line of code:

```
>·substr(txt[1],·4,·4+22-1)¶
[1]·"s·is·a·first·example·s"
```

In regular expressions lingo, this is called *greedy matching*. However, this may not always be what we want. If, for example, we have a corpus line which matches a particular search pattern more than once, then, as corpus linguists, we often would not just want one long match (with unwanted material in the middle):

```
>·some.corpus.line<-"he·said,·you·are·lazy·and·you·are·stupid."¶
>·gregexpr("you.*are·\\w+",·some.corpus.line,·perl=TRUE)¶
[[1]]
[1]·10
attr(,"match.length")
[1]·31
```

```
>·substr(some.corpus.line,·10,·10+31-1)¶
[1]·"you·are·lazy·and·you·are·stupid"
```

Why don't we want this behavior? Imagine you are interested in the most interesting and important linguistic phenomenon that has ever existed, i.e., the constituent order alternation of verb-particle constructions in English, i.e., the alternation between *John picked up the book* and *John picked the book up*. If you have a tagged corpus in which verbs and particles are tagged – e.g., the corpus would look like this: *John*_N *picked*_V *the*_D *book*_N *up*_P – then you don't want R to do greedy matching because of how that would handle sentences with two verb-particle constructions such as *John*_N *picked*_V *up*_P *the*_D *book*_N *and*_CJ *brought*_V *it*_PN *back*_P. If we look for a verb followed later by a particle, greedy matching would return a match that, at first sight, suggests the first verb (*picked*) goes with the second particle (*back*):

```
>·vpc.example<-"John_N·picked_V·up_PRT·the_D·book_N·and_CJ·
  brought_V·it_PN·back_PRT"¶
>·gregexpr("[^_·]+_V.*[^_]+_PRT",·vpc.example,·
  ignore.case=TRUE,·perl=TRUE)¶
[[1]]
[1]·8
attr(,"match.length")
[1]·56
attr(,"useBytes")
[1]·TRUE

>·substr(vpc.example,·8,·6+60-1)¶
[1]·"picked_V·up_P·the_D·book_N·and_CJ·brought_V·it_PN·back_P"
```

Pay attention to the regex used above with `gregexpr`: Note in particular how the regex above operationalizes the notion of a word – one or more characters that are not spaces (because spaces separate words from each other, at least according to a simplistic definition adopted here) and that are not underscores (because those separate words and their tags from each other).

Thus, what we are more likely to be interested in is getting to know that there are two matches and what these are. There are two ways of setting R's matching strategy to *lazy*, i.e., non-greedy. The first one applies to a complete search expression, setting all quantifiers in the regular expression to non-greedy matching. You just write "(?U)" as the first four characters of your regular expression:

```
>·gregexpr("(?U)you.*are·\\w+",·some.corpus.line,·perl=TRUE)¶
[[1]]
[1]·10·27
attr(,"match.length")
[1]·9·9
```

As you can see, now both consecutive matches are found. However, this is still bad – why?

**THINK**

**BREAK**

The matches are just nine characters long, two more than before, which means they only added a space and "l"/"s". And why? Because the (?u) makes every quantifier lazy, i.e., also the "+" after "\\w" so R stops after one word character. What we want, however, is for the first quantifier to match lazily (the "*") but for the second to match greedily so that the "\\w+" gets the whole word. Thus, we would like a more flexible way to set lazy matching: Rather than using non-greedy matching for the complete regular expression, you can use greedy *and* non-greedy matching in one expression by adding a question mark to those quantifiers which you would like to use non-greedily. As you can see, we now get the desired result – both matches and their correct lengths are matched:

```
>·gregexpr("you.*?are·\\w+",·some.corpus.line,·perl=TRUE)¶
[[1]]
[1]·10·27
attr(,"match.length")
[1]·12·14
```

There we go. Let's apply this to `vpc.example`:

```
>·gregexpr("[^_·]+_V.*?[^_]+_PRT",·vpc.example,·
  ignore.case=TRUE,·perl=TRUE)¶
[[1]]
[1]··8·44
attr(,"match.length")
[1]·15·24
attr(,"useBytes")
[1]·TRUE

>·substr(rep(vpc.example,·2),·c(8,·44),·c(8+15-1,·44+24-1))¶
[1]·"picked_V·up_PRT"·····"brought_V·it_PN·back_PRT"
```

Not bad at all. But now note what happens if we apply this solution to the vector `txt`:

```
>·gregexpr("s.*?s",·txt[1],·perl=TRUE)¶
[[1]]
[1]··4·14
attr(,"match.length")
[1]··4·12
```

We still don't get all three consecutive occurrences we would probably want, namely "s·is", "s·a·firs", and "st·example·s", let alone these plus the additional three that are theoretically also possible from above ("s is a firs", "s a first example s", and "s is a first example s"). Why is that?

**THINK**

**BREAK**

This is because when the regex engine has identified the first match – "s·is" – then the second "s" in `txt[1]` has already been consumed as being the second "s" in the match and is, thus, 'not available' anymore to be the first match next time: The regex engine will start to look for new matches only after it. We will therefore revisit this case below to get at least at the third useful match.

The second issue we need to address is that we have so far only been concerned with the most elementary replacement operations. That is to say, while you have used simple matches as well as more complex matches in terms of what you were searching for, your replacement has so far always been a specific element (a character or a character string). However, there are of course applications where, for example, you may want to add something to your match. Imagine, for example, you would like to tag the vector `txt` such that every word is followed by "<w>" and every punctuation mark is followed by "<p>". Now, obviously you cannot simply replace the pattern match as such with the tag because then your words and punctuation marks would be replaced rather than have tags added to them:

```
>·(txt.2<-gsub("\\w+",·"<w>",·txt,·perl=TRUE))¶
[1]·"<w>·<w>·<w>·<w>·<w>·<w>."
[2]·"<w>·<w>·<w>·<w>·<w>·<w>·<w>."
```

Obviously, this might not be the world's best tagger ever. Second, you cannot simply replace word boundaries "\\b" with tags because then you get tags before *and* after the words:

```
>·(txt.2<-gsub("\\b",·"<w>",·txt,·perl=TRUE))¶
[1]·"<w>This<w>·<w>is<w>·<w>a<w>·<w>first<w>·<w>example<w>·
  <w>sentence<w>."
[2]·"<w>And<w>·<w>this<w>·<w>is<w>·<w>a<w>·<w>second<w>·
  <w>example<w>·<w>sentence<w>."
```

Thus, what you need is a way to tell R to (1) match something, (2) bear in mind what was matched, and (3) replace the match by what was matched *together with* the additional tag you want to insert. Fortunately, regular expressions offer the possibility of what is referred to as "back-referencing". Regular expressions or parts of regular expressions which are grouped by parentheses are internally numbered (in the order of opening parentheses) so that they are available for future matching and/or replacing in the same function call.

Let us first look at the simple tagging example from above. In two steps, you can tag the 'corpus' in the above way:

```
>·(txt.2<-gsub("(\\w+)",·"\\1<w>",·txt,·perl=TRUE))¶
[1]·"This<w>·is<w>·a<w>·first<w>·example<w>·sentence<w>."
[2]·"And<w>·this<w>·is<w>·a<w>·second<w>·example<w>·sentence<w>."

>·(txt.3<-gsub("([!:,.?])",·"\\1<p>",·txt.2,·perl=TRUE))¶
[1]·"This<w>·is<w>·a<w>·first<w>·example<w>·sentence<w>.<p>"·
[2]·"And<w>·this<w>·is<w>·a<w>·second<w>·example<w>·sentence<w>.<p>"
```

That is to say, in the first step, the pattern you are searching for is defined as one or more consecutive occurrences of a word character, and the parentheses instruct R to treat the complete sequence as a single element. Since there is only one set of opening and closing parentheses, the number under which the match is available is of course 1. The replacement part, "\\1<w>", then instructs R to insert the remembered match of the first parenthesized expression in the search pattern ("\\1") – which in this case amounts to the whole search pattern – and insert "<w>" afterwards. The second step works in the same way except that now the parenthesized subexpression is a character class of multiple punctuation marks, which is treated as a single item, to which a punctuation mark tag "<p>" is added; note the fact that punctuation marks being in a character class obviates the need for writing "\\." and "\\?".

Of course, this approach can be extended to handle more complex expressions, too. Imagine, for example, your data contain dates in the American-English format, with the month preceding the day (i.e., Christmas would be written as "12/25/2016") and you want to reverse the order of the month and the day so that your data could be merged with corpus files that already use this ordering. Imagine also that some of the American dates do not use slashes to separate the month, the date, and the year, but periods. And, you must take into consideration that some people write April 1 as "04/01" whereas others write "4/1". How could you change American dates into British dates using only slashes as separators (regardless of what separators are used in the American dates)?

```
>·(American.dates<-c("7/31/1976",·"02.15.1970",·"11.31.1986"))¶
```

**THINK**

**BREAK**

```
>·(British.dates<-gsub("(\\d{1,2})\\D(\\d{1,2})\\D",·"\\2/\\1/",·
  American.dates,·perl=TRUE))¶
[1]·"31/7/1976"·"15/02/1970"·"31/11/1986"
```

This regular expression may look pretty daunting at first, but in fact it is a mere concatenation of many simple things we have already covered. The first parenthesized

expression, "(\\d{1,2})", matches one or two digits and, if matched, stores them as the first matched regular expression, "\\1". "\\D" then matches any non-digit character and, thus, both the period or the slash. Of course, if you are sure that the only characters in the data used to separate the parts of the date are a period and slash, a character class such as "[./]" would have worked just as well as "\\D". If you are not, then "\\D" may be a better choice because it would also match if the parts of the date were separated by hyphens as in "7-31-1976". The second parenthesized expression, again "(\\d{1,2})", matches one or two digits but, if matched, stores the match as the second matched regular expression, "\\2". You might actually want to think about how you can also preserve the original separators instead of changing them into slashes.

**THINK**

**BREAK**

```
>·(British.dates<-gsub("(\\d{1,2})(\\D)(\\d{1,2})
  (\\D)",·"\\3\\4\\1\\2",·
  American.dates,·perl=TRUE))¶
[1]·"31/7/1976"·"15.02.1970"·"31.11.1986"
```

Let me use this regex to introduce one very nice way in which search expressions in complex regular expressions can be made more readable. This way is what is referred to as *free-spacing mode* and it is defined by a flag "(?x)" that is put first in a regular expression (just like "(?U)" was above for lazy matching). If your regular expression begins with "(?x)", then – simplifying a bit – spaces " " (written here as "·") and newlines are ignored and you can insert comments into regular expressions (prefixed with "#") to make them easier to parse; note that this means that if you want to find a space in free-spacing mode, you need to use "\\s" for the space. Here is the above line in free-spacing mode; it's obvious how, for longer or more complex regular expressions, this makes parsing regexes much simpler, because now every component of the regex can be explained properly right in the code, which is why I will sometimes make use of this in the code file:

```
>·(British.dates<-gsub(¶
+····"(?x)·····#·set·free-spacing¶
+····(\\d{1,2})·#·1·or·2·digits,·memorize·as·\\1¶
+····(\\D)·····#·1·non-digit··,·memorize·as·\\2¶
+····(\\d{1,2})·#·1·or·2·digits,·memorize·as·\\3¶
+····(\\D)·····#·1·non-digit··,·memorize·as·\\4¶
+····",¶
+····"\\3\\4\\1\\2",¶
+····American.dates,·#·in·American.dates¶
+····perl=TRUE))·····#·using·Perl-compatible·regular·expressions¶
[1]·"31/7/1976"·"15.02.1970"·"31.11.1986"
```

So far we have used back-referencing in such a way that the match of a search expression or parts of the match of a search expression were used again in the *replacement pattern*. Note, however, that back-referencing also includes the possibility of re-using a match *within the search pattern* itself. Let us look at a simple example first. Imagine you are interested in the rhyming of adjacent words and would, therefore, like to find in a character vector just like `txt` cases where adjacent words end with the same character sequences, obviously because these are likely candidates for rhymes.[4] More specifically, imagine you would like to add "<r>" to every potential rhyme sequence such that "This is not my dog" gets changed into "This<r> is<r> not my dog", but that you only want cases of adjacent words so that the string "This not is my dog" would not match. In such a case, you would want to find a sequence of at least two characters at the end of a word, which we can translate as "\\w{2,}?\\W+". However, it is more complex than that. You must instruct R to bear the sequence of characters in mind, which is why you would need to write "(\\w{2,}?)". Then, this character sequence in question must be followed by at least one non-word character (e.g., a space or a comma and a space) plus the word characters of the beginning of the next word: "\\W+\\w*?". Note, however, that we write "\\w*?" because the beginning of the word may already rhyme so the beginning is optional, hence the asterisk. Since you will have to put those back into the string, you will need to memorize and, thus, parenthesize them, too. Then, you only want matches if the same character sequence you found at the beginning is matched again: "\\1". However, you only want it to match if this is again at the end of a word (i.e., followed by "\\W") and, what is more, you will have to memorize that non-word character to put it back in: "\\1\\W". As you can see, the back-reference is back to a pattern within the search.

When this is all done, the replacement is easy: You want to replace all this by the potentially rhyming characters followed by "<r>", then the characters after the potentially rhyming characters, then the potentially rhyming characters, and the next non-word character. All in all, this is how it looks:

```
>·gsub("(\\w{2,}?)(\\W+\\w*?)\\1(\\W)",·"\\1<r>\\2\\1<r>\\3",·txt,·
  perl=TRUE)¶
[1]·"This<r>·is<r>·a·first·example·sentence."
[2]·"And·this<r>·is<r>·a·second·example·sentence."
```

You can now also test that this regular expression does return a match on "This is not my dog" but does not match "This not is my dog", because in the latter the *This* and the *is* are not adjacent anymore; check out the code file for the (free-space commented) code. When you do so, hopefully, you look at this and immediately see a way to improve it. If not, think about (or try out) what happens if the elements of the vector `txt` did not have final periods. How could you improve the script in this regard and simplify it at the same time?

**THINK**

**BREAK**

The solution and simpler strategy would be this:

```
>·gsub("(\\w{2,}?)(\\W+\\w*?)\\1\\b",·"\\1<r>\\2\\1<r>",·txt,·
  perl=TRUE)¶
```

As you can see, instead of the final non-word character, this uses a word boundary at the end. Why (is this better)? This is because the expression "\\1\\b" only consumes the rhyming characters, which has two consequences. First, this expression also matches the end of the string if there is no period because it does not require a separate non-word character to match after the rhyming character(s). Second, because the expression does not consume anything after the rhyming character(s), you need not capture anything after their second occurrence because you do not have to put anything back in; this is why this expression works with only two, not three, back-references. Thus, the second version is actually preferable.

This example and its discussion gloss over the additional issue of how the approach would have to be changed to find rhyming words that are not adjacent, and you will meet this issue in the next exercise box.

The final kind of expression to be discussed here is referred to as *lookaround*. Lookaround is a cover term for four different constructs:

Lookahead

| | |
|---|---|
| `(?=...)` | positive lookahead: match if you see . . . on the right |
| `(?!...)` | negative lookahead: match if you do not see . . . on the right |

Lookbehind

| | |
|---|---|
| `(?<=...)` | positive lookbehind: match if you see . . . on the left |
| `(?<!...)` | negative lookbehind: match if you do not see . . . on the left |

Lookaround is sometimes a bit difficult to understand for beginners, but often exceptionally useful. The key characteristics of lookaround expressions are that

- like most regular expressions we have seen before, they match characters (rather than, say, positions);
- unlike most regular expressions we have seen before, they give up the match – i.e., they do not consume characters and are therefore, just like line anchors, zero-width assertions – and only return whether they matched or did not match.

Let us look at a simple example for positive lookahead. You use positive lookahead to match something that is followed by something else (which, however, you do *not* wish to consume with that expression). For example, imagine you have the following vector `example`:

```
>·example<-c("abcd",·"abcde",·"abcdf")¶
```

If you now want to match "abc" but only if "abc" is followed by "de", you can use positive lookahead like this:

```
>·grep("abc(?=de)",·example,·perl=TRUE)¶
[1]·2
```

You may now ask "So what, I could just write this . . . ":

```
>·grep("abcde",·example,·perl=TRUE)¶
[1]·2
```

True, but one area where the difference is particularly relevant is when you want to not just find, but also replace. If you want to replace "abc" by "xyz", but only if that "abc" is immediately followed by "de", you can't just write the following because it will also consume and thus replace the "de":

```
>·gsub("abcde",·"xyz",·example,·perl=TRUE)¶
[1]·"abcd"··"xyz"···"abcdf"
```

With positive lookahead, no problem:

```
>·gsub("abc(?=de)",·"xyz",·example,·perl=TRUE)¶
[1]·"abcd"··"xyzde"·"abcdf"
```

As you can see, the string "de" is not consumed, and thus not replaced. Of course, you can combine lookaround expressions with nearly all the regular expressions you already know. For example, you could replace every character in front of an "e" by two occurrences of that character:

```
>·gsub("(.)(?=e)",·"\\1\\1",·example,·perl=TRUE)¶
[1]·"abcd"···"abcdde"·"abcdf"
```

Maybe somewhat more realistically, you could tag every "vowel letter" ("[aeiou]") that is between two "consonant letters" ("[bcdfghjklmnpqrstvwxyz]"):

```
>·gsub("(?<=[bcdfghjklmnpqrstvwxyz])([aeiou])(?=[bcdfghjklmnpqr
  stvwxyz])",·
  "<\\1>",·txt,·perl=TRUE)¶
[1]·"Th<i>s·is·a·f<i>rst·ex<a>mple·s<e>nt<e>nce."
[2]·"And·th<i>s·is·a·s<e>c<o>nd·ex<a>mple·s<e>nt<e>nce."
```

However, positive lookahead is not only useful for replacements. Let us return again to the problem of finding the three instances of one "s" followed by some characters up to

another "s" that we faced with the vector `txt`. The last time we managed to get two out of the three desired matches (when we introduced lazy matching), but we still had the problem that a consumed second "s" was not available as a first "s" in the next match. As you may already guess, the answer is to not consume the second "s", and you now know how to do that: with positive lookahead:

```
>·gregexpr("s.*?(?=s)",·txt[1],·perl=TRUE)¶
[[1]]
[1]··4··7·14
attr(,"match.length")
[1]··3··7·11
```

Finally, all three matches are identified: Check out the code for how you can extract them (with a crude approach that introduces the function `paste0` and only works well in simple cases like the one here).

Negative lookahead allows you to match something if it is *not* followed by something else. In this example, you capitalize only those "d" characters that are not followed by "e":

```
>·gsub("d(?!e)",·"D",·example,·perl=TRUE)¶
[1]·"abcD"··"abcde"·"abcDf"
```

Note the difference from what you may think is the same, but which in fact is *not* the same, and it is again different in how the lookahead does not consume the "e", whereas "[^e]" does:

```
>·gsub("d[^e]",·"D",·example,·perl=TRUE)¶
[1]·"abcd"··"abcde"·"abcD"
```

Negative lookahead is especially useful when you want to replace unwanted strings, and here is an example we need to discuss in detail. Imagine you have a sentence with an SGML word-class annotation as in the BNC. In this format, every word is preceded by a tag in angular brackets (see www.natcorp.ox.ac.uk/docs/URG/posguide.html). The first character after the opening angular bracket of the tag is the tag's name "w" (for "word"), which is mostly followed by a space and a three character POS tag. However, the BNC also has so-called portmanteau tags, which indicate that the automatic tagger could not really decide which of two tags to apply. For example, a portmanteau tag "<w·AJ0-VVN>" would mean that the tagger was undecided between the base form of a regular adjective ("AJ0") or the past participle of a lexical verb ("VVN"). Apart from POS tags, other information is also included. For example, tags named "s" contain a name–value pair providing the number of each sentence-unit in a file; tags named "ptr" are used to indicate overlapping speech and contain a name–value pair identifying the location and the speaker of the overlapping speech. This is an example:

```
>·example1<-"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<ptr·target=KB0LC003><w·AV0>so
  <c·PUN>,·<w·PNP>you·<w·VVB>know·<ptr·target=KB0LC004></u>"¶
```

Imagine now you want to delete all tags that are not word tags or punctuation-mark tags. In other words, you want to keep all tags that look like this "<[wc]·...(-...)?>". Unfortunately, you cannot do this with negated character classes: Something involving "[^wc·]" does not work because R will understand this to mean that the "w", the "c", and the space are one-character alternatives, but not as meaning "not 'w·' and also not 'c·'". Now you might say, okay, but I can do this:

```
>·gsub("<[^wc][^·].*?>",·"",·example1,·perl=TRUE)¶
[1]·"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<w·AV0>so<c·PUN>,·<w·PNP>you·
  <w·VVB>know·"
```

And yes, this looks nice. But . . . you are just lucky because `example1` does not have a tag like "<wtr·target=KB0LC004>", which begins with "<w" and which you would also want to delete. Look what would happen:

```
>·example2<-"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<ptr·target=KB0LC003><w·AV0>so
  <c·PUN>,·<w·PNP>you·<w·VVB>know·<wtr·target=KB0LC004></u>"¶
>·gsub("<[^wc][^·].*?>",·"",·example2,·perl=TRUE)¶
[1]·"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<w·AV0>so<c·PUN>,·<w·PNP>you·
  <w·VVB>know·<wtr·target=KB0LC004>"
```

The last tag, which you wanted to delete, was not deleted. Why? Because it does not match: You were looking for something that is not "w" or "c" followed by something that is not a space, but this tag *has* a "w" followed by a non-space. The same problem arises when there is no "w" at the beginning of the tag in question, but a space in its second position:

```
>·example3<-"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<ptr·target=KB0LC003><w·AV0>so
  <c·PUN>,·<w·PNP>you·<w·VVB>know·<p·tr·target=KB0LC004>xyz
  </u>"¶
>·gsub("<[^wc][^·].*?>",·"",·example3,·perl=TRUE)¶
[1]·"<w·UNC>er<c·PUN>,·<w·AV0>anyway·<w·PNP>we<w·VBB>'re·
  <w·AJ0>alright·<w·AV0>now·<w·AV0>so<c·PUN>,·<w·PNP>you·
  <w·VVB>know·<p·tr·target=KB0LC004>"
```

One answer to our problem is negative lookahead. Here's how you do it, and you can try it out with `example2` and `example3` as well to make sure it works:

```
>·gsub("<(?![wc]·...(-...)?>).*?>",·"",·example1,·perl=TRUE)¶
```

It is important that you understand how this works, so let me go over it in detail. First, the expression matches an opening angular bracket "<", no problem here. Then, negative lookahead says, match only if the parenthesized lookahead expression, i.e., the characters after the "?!" after the opening angular bracket, does *not* (*negative* lookahead) match "<[wc]· . . . (- . . . )?>". Now comes the tricky part. You must recall that lookahead does not consume: thus, after having seen that the characters after the angular bracket are not "[wc]· . . . (- . . . )?>", R's regular expression engine is still at the position after the first consumed character, the angular bracket; that's what zero-width assertion was all about. Therefore, after having said what you *don't* want to see after the "<" – which is "[wc]· . . . (- . . . )?>" – you must now say what you *do* want to see, which is anything, hence the ".*?", but only until the next angular bracket closing the tag. Another way to understand this is as follows: negative lookahead says "match the stuff after the negative lookahead ('.*?'), but not if it (also) looks like the stuff in the negative lookahead ('[wc]· . . . (- . . . )?>')."

So how does it apply to the `example1`? The first tag "<w·UNC>" is not matched because the expression cannot match the negative lookahead: "<w·UNC>" is exactly what "?!" says *not* to match. The same holds for the next few tags. Once the regex engine arrives at "<ptr·target=KB0LC003>", though, something different happens. The regex engine sees the "<" and notices that the characters that follow the "<" are not "[wc]· . . . (- . . . )?>". Thus, it matches, but does not consume; the engine is still after the "<". The string "ptr target=KB0LC003" is then matched by ".*?", and the closing ">" is matched by ">". Thus, everything matches, and R can replace this by "", effectively deleting it.

This is a very useful approach to know and some of the scripts below that handle BNC files will make use of this expression to "clean up" the file before further searches take place (so that overlap tags do not interrupt sequences of words we are interested in, for instance). An extension of this not only deletes unwanted tags, but also the material that follows these unwanted tags up until the next tag:

```
>·gsub("<(?![wc]·...(-...)?>).*?>[^<]*",·"",·example3,·perl=TRUE)¶
```

**Recommendations for further study/exploration**

- On how to match with positive and negative lookbehind: explore `(?<=)` and `(?<!)`. Note that Perl-compatible regular expressions do unfortunately not allow variable-length lookbehind.

You should now do "Exercise box 3.7: A few more regular expressions."

### *3.7.6 Merging/Splitting Character Vectors With Regular Expressions*

We saw in Section 3.7.3 that we can split up character strings using `strsplit`. While the above treatment of `strsplit` did not involve regular expressions, this function can handle regular expressions in just about the way that `grep` and the other functions can.

Let us exemplify this by means of a character string that may be found like this in the BNC with SGML annotation: Every word is preceded by a POS tag in angular brackets, just as in the above examples.

```
>·(tagtext<-"<w·DPS>my·<w·NN1>mum·<w·CJC>and·<w·DPS>my·
  <w·NN1>aunt·<w·VVD>went·<w·PRP>into·<w·NN1>service")¶
```

Imagine we want to get the words and get rid of the tags. Looking at the above, there seem to be two alternatives to do that: First, we could simply delete the tags (with `gsub`) and then `strsplit` on spaces. Second, we could `strsplit` on tags. Which one is better and why?

**THINK**

**BREAK**

This was a bit of a trick question. Obviously, the second is better because it's just one operation. But there is a potentially more important reason. Look at the excerpt from a BNC file shown in Figure 3.5.

If you pursue the first of the two strategies, you end up with the following word tokens: "now", "because", "of", "obvious", and "reasons" – if you pursue the second, you get "now", "because·of", "obvious", and "reasons" because splitting up on the tag doesn't lose the (potentially very useful) information that the BNC compilers considered *because of* a multi-word unit. Think how this seemingly small decision has potentially big implications: If you generate a frequency list of the BNC using the first approach, then you lose the counts of all multi-word units but every *because of*, *in spite of*, *out of*, etc. increases the frequencies of *of* as well as *because*, *in*, *spite*, and *out*. A frequency list generated with the second approach doesn't have that problem, which is why we will go with the second approach here.

Back to `tagtext`: If you want to split up this character string at each tag such that every word is one element of an overall vector, this is what you could do:

```
>·unlist(strsplit(tagtext,·"<w·...>",·perl=TRUE))¶
[1]·""·.......·"my·"·....·"mum·"·...·"and·"·...·"my·"·....·"aunt·"···
  "went·"···"into·"···"service"
```

As you can see, the function looks for occurrences of "<w·" followed by three characters and a ">" and splits the string up at these points. Recall again that `unlist` is used here to let R return a vector, not a list. While this works fine here, however, life is usually not that

```
<w·AV0>now·<w·PRP>because·of·<w·AJ0>obvious·<w·NN2>reasons
```

*Figure 3.5* A few words from the BNC World Edition (SGML format).

simple. For one thing, not all tags might correspond to the pattern "<w· . . . >", and in fact not all do. Punctuation mark tags, as you already saw above, look different:

```
>·tagtext<-"<w·DPS>my·<w·NN1>mum·<c·PUN>,·<w·DPS>my·
  <w·NN1>aunt·<w·VVD>went·<w·PRP>into·<w·NN1>service"¶
```

However, you should be able to find out quickly what to do. This is a solution that would work when the tag beginnings are either "<w·" or "<c·" (result not shown here):

```
>·unlist(strsplit(tagtext,·"<[wc]·...>",·perl=TRUE))¶
```

Again, however, life is more complex than this: recall also that we have to be able to accommodate portmanteau tags:

```
>·tagtext<-"<w·DPS>my·<w·NN1>mum·<c·PUN>,·<w·DPS>my·
  <w·AJ0-VVN>beloved·<w·NN1>aunt·<w·VVD>went·<w·PRP>into·
  <w·NN1>service"¶
```

Among the ways of handling this, this is one with alternatives, but without ranges:

```
>·unlist(strsplit(tagtext,·"<[wc]·(...|...-...)>",·perl=TRUE))¶
```

As you can see, the expression either looks for any three characters or for any three characters followed by a hyphen and another three characters. The other possibility involves an expression with a range such that we are either looking for three or seven (three + "-" + three) characters; strictly speaking, the former alternative is better since it is more restrictive (by specifying the nature of the up to seven characters in more detail).

```
>·unlist(strsplit(tagtext,·"<[wc]·.{3,7}>",·perl=TRUE))¶
[1]·""·.........."my·"......"mum·"....."·"........"my·"......
  "beloved·"·"aunt·"····"went·"····"into·"····"service"
```

One final thing is less nice. As you can see, `strsplit` splits up at the tags, but it does not remove the spaces after the words. This may not seem like a great nuisance to you but, before you go on reading, look at the last word and think about in what way this may be problematic for further corpus-linguistic application.

**THINK**

**BREAK**

If you could not think of something, consider what would happen if the vector `tagtext` looked like this:

```
>·tagtext<-"<w·DPS>my·<w·NN1>service·<w·DPS>my·<w·AJ0-VVN>beloved·
  <w·NN1>aunt·<w·VVD>went·<w·PRP>into·<w·NN1>service"¶
```

**THINK**

**BREAK**

The problem arises when you apply the above regular expression to this version of `tagtext`:

```
>·unlist(strsplit(tagtext,·"<[wc]·(...|...-...)>",·perl=TRUE))¶
[1]·""·····"my·"·····"service·"·"my·"·····"beloved·"·
  "aunt·"····"went·"····"into·"····"service"
```

As you can see, there is one occurrence of "service" with a following space – the first one – and one without a following space – the second one. The second one is not followed by a space because its characters were the last ones in the vector `tagtext` and, more often, they might not be followed by a space because they are immediately followed by a punctuation mark. The problem is that if you now wanted to generate a frequency list using `table`, which you already know, R would of course treat "service" and "service·" differently, which is presumably not what anyone would want:

```
>·table(unlist(strsplit(tagtext,·"<[wc]·(...|...-...)>",·
  perl=TRUE)))¶
·····aunt···beloved···into···my···service···service····went
··1·····1·····1·····1····2·····1·····1·····1
```

Thus, what would a better solution look like, i.e., a solution that deletes the spaces?

**THINK**

**BREAK**

This is what you could do:

```
>·unlist(strsplit(tagtext,·"·*<[wc]·(...|...-...)>·*",·perl=TRUE))¶
[1]·""·····"my"·····"service"·"my"·····"beloved"·"aunt"····
  "went"····"into"····"service"
```

This was just a simple example, but I hope it has shown how important it is to know what you're doing and to what you're doing it.

**Recommendations for further study/exploration**

- On how to do approximate pattern matching: `?agrep¶` and also check out `?adist¶`.
- Spector (2008: chapter 7).
- On regular expressions in general: Stubblebine (2007), Goyvaerts and Levithan (2012), the "regular expression bible", Friedl (2006), and in particular the absolutely wonderful website www.regular-expressions.info.

## 3.8 Two Particularly Relevant Areas: Unicode and XML

### *3.8.1 Some Notes on Handling Unicode*

This section looks at non-English non-ASCII data. Just as most corpus-linguistic work has been done on English, this chapter has so far also been rather Anglo/ASCII-centric. In this section we are going to deal with data involving Unicode by briefly looking at data involving Cyrillic characters. There is good news and bad news. The good news is that nearly everything that you have read so far applies to Unicode files with Cyrillic characters, too. There are a few small changes, such as that you cannot really use regular expressions such as "\\w", "\\W", or "\\b" to work (because they are defined with regard to a Latin alphabet), but otherwise nothing major. The bad news for Windows users is that, with R for Windows at least, it is often not as straightforward to handle Unicode as it is with R for MacOS X or R for Linux, which is mainly due to the fact that Mac OS X and Linux are UTF-8 operating systems so they play well with input files that have the same encoding, which is one of many reasons (see Section 3.8.2 for another one) why I recommend doing corpus processing with R on a Linux machine (and it takes very little time to install a long-term support version of, for instance, Kubuntu or Linux Mint on a desktop computer).

Let us load a file that has UTF-8 encoding – like actually nearly all others before – but now also non-Western European characters in it, namely the file with Cyrillic characters we looked at above, <_qclwr2/_inputfiles/corp_utf8_cyrillic.txt>:

```
>·rm(list=ls(all=TRUE))¶
>·corpus.file<-readLines(con<-file(file.choose(),·encoding="UTF-8"),·
  warn=FALSE);·close(con)¶
>·tail(corpus.file)¶
[1]·"*·Короткий·срок·изготовления·двери·на·заказ·(10·дней)."
[2]·"*·Качественная·установка."
[3]·"*·Гарантия·1·год·на·все·двери."
[4]·"*·Возможность·скидок·для·оптовых·клиентов."
[5]·"Мы·работаем·для·Вас!"
[6]·""
```

Let us first briefly look at generating frequency lists. I will treat three ways of creating case-insensitive frequency lists of the words in this file, all of which involve the strategy of splitting up the corpus file using `strsplit`, but they differ in how we define how you are splitting.

The first of these is basically no different from how we did it above, which is why I reduce most of the exposition to just the code, which you will understand without problems. To make it a little more interesting, however, we will first determine which characters are in the file and which of these we might want to use for `strsplit`, so, how do you find out all characters that are used in the corpus file?

**THINK**

**BREAK**

You need to remember that `strsplit` splits up (elements of) a vector characterwise if no splitting character is provided. Thus:

```
>·table(unlist(strsplit(corpus.file,·"")))¶
```

This tells you pretty much exactly which characters to use for splitting (and which not to worry about here, such as semicolons and single quotes), and you can proceed as above:

```
>·words.1<-unlist(strsplit(corpus.file,·"[--!:,\"\\?\\.\\*\\
  (\\)\\s\\d]+",·
  perl=TRUE))¶
>·words.1<-tolower(words.1[nzchar(words.1)])¶
>·sorted.freq.list.1<-sort(table(words.1),·decreasing=TRUE)¶
>·head(sorted.freq.list.1,·10)¶
words.1¶
·····и·····в·····для·daminnad·····к·····—····белья
·····12·····9·····5·····4·····4·····3·····3
····двери·····мы·····на
·····3·····3·····3
```

By the way, note that we could apply `tolower` and `toupper` in the familiar way without problems.

As mentioned above, one difference to the above treatment of ASCII data is that in an American-English locale, you cannot use characters such as "\\b" or "\\W" with `perl=TRUE` here because what is a word character depends on the locale, and the Cyrillic characters simply are not part of the ASCII character range [a-zA-Z]. However, there is a nice alternative worth knowing, which brings us to the second way to split up the corpus file into words, which is using character points and, more usefully, character ranges. Specifically, you can access each Unicode character in R individually by the sequence "\u" followed by a four-digit hexadecimal number.[5] This has two important consequences: First, once you know a Unicode character's number, you can use that number to call the character, which is particularly handy if, for example, you want to define a search expression in Cyrillic characters but don't have a Cyrillic keyboard and don't want to fiddle around with system keyboard settings, etc. Here are some examples:

```
>·"\u0401"·#·Ë
[1]·Ë
>·"\u0410"·#·A¶
[1]·A
>·"\u044F"·#·я¶
[1]·я
>·"\u0451"·#·ë¶
[1]·ë
```

Thus, you can look up the characters for the Russian word meaning "but" from the Unicode website or sites at Wikipedia, 'construct' it at the console, and even assign it to a character vector, which could then be the search expression for `grep`, `gregexpr`, `exact.matches.2`, etc.:

```
>·(x<-"\u043d\u043e")¶
[1]·"но"
```

Second, when the characters of the alphabet you are concerned with come in ranges, you can easily define a character class using hexadecimal Unicode ranges. For instance, the small and capital characters of the Russian Cyrillic alphabet can be defined as follows:

```
>·(russ.char.yes.capit<-"[\u0410-\u042F\u0401]")¶
>·(russ.char.yes.small<-"[\u0430-\u044F\u0451]")¶
```

Note that the square brackets with which we define this character class are part of the character strings – this is so we can use these two character vectors as search expressions in `grep` etc. This of course then also means that you can easily define non-word characters in nearly the same way:

```
>·(russ.char.yes<-"[\u0401\u0410-\u044F\u0451]")¶
>·(russ.char.no<-"[^\u0401\u0410-\u044F\u0451]")¶
```

(Several regular-expression engines actually have a variety of predefined so-called *Unicode blocks* – i.e., contiguous ranges of code points – but the Perl-compatible regular expressions R uses do not support predefined Unicode blocks, so the above way of defining ranges manually is probably often your best choice.)

Thus, the second way to create the above kind of frequency list is this: You split on one or more occurrences of characters that are not Cyrillic letters:

```
>·words.2<-unlist(strsplit(corpus.file,·paste0(russ.char.no,·"+"),·
  perl=TRUE))¶
>·words.2<-tolower(words.2[nzchar(words.2)])¶
```

```
>·sorted.freq.list.2<-sort(table(words.2),·decreasing=TRUE)¶
>·head(sorted.freq.list.2,·10)¶
····и·····в·····для·····к···белья···двери·····мы·····на
···12·····9·····5·····4·····3·····3·····3·····3
·нижнего·····все
·····3·····2
```

If you compare this output to the one above, you should learn another important lesson: The word "daminnad" shows up four times in the first list, but not in the second. This is of course because the characters that make up this word are not Cyrillic characters as defined by the above Unicode character ranges, which is why they get used by `strsplit` and, thus, effectively get deleted. Also, note that just because the "a" in "daminnad" and the ASCII "a" look the same to the human eye does not mean they are the same Unicode character. For example, what may look like ASCII spaces or commas in the file <_qclwr2/_inputfiles/corp_utf8_mandarin.txt> are not ASCII characters! Thus, you must be very careful about how to define your splitting characters.

The third and final way is based on the fact that there are several predefined *Unicode categories* which you can use with a regular expression that defines so-called *properties*: the regex for that involves the syntax "\\p{ . . . }", where you replace . . . with the name of the property you wish to use. The code file provides a few examples of Unicode categories you can set with properties (and www.regular-expressions.info/unicode.html provides many more), but here I will only show how you use a property to define a so-called *Unicode script* for the task of creating our frequency list of Russian words with the Cyrillic alphabet:

```
>·words.3<-exact.matches.2("\\p{Cyrillic}+",·corpus.file,·
  gen.conc.output=FALSE)[[1]]¶
>·words.3<-tolower(words.3[nzchar(words.3)])¶
>·sorted.freq.list.3<-sort(table(words.3),·decreasing=TRUE)¶
>·head(sorted.freq.list.3,·10)¶
words.3
····и·····в·····для·····к···белья···двери·····мы····
  ··на·нижнего·····все
···12·····9·····5·····4·····3·····3·····3····
  ···3·····3·····2
```

As you can see, this approach returns the same results as the second one, but this one of course presupposes that the writing system you are studying is predefined as a Unicode script.

Let us now turn to concordancing with Unicode data, which is also nearly the same as with ASCII data; the main difference being again that, with `perl=TRUE`, you cannot use, say, "\\b". Let us begin with a simple example in which we look for a Russian word meaning "but", which is "но". You first define it as a search word (and often it is useful to immediately put parentheses around it so you can use back-referencing later when, for instance, you want to put tabstops around the match):

```
>·search.word<-"(но)"¶
```

The simplest way to look for this is of course to just use the above `search.word` as the search expression with `grep`. However, if you look for the search expression you will find many examples where "но" is part of adjectival inflection (e.g., "ное" or "ного") or of the derivational morpheme "ность":

```
>·exact.matches.2(search.word,·corpus.file,·case.sens=FALSE)[[4]]
  [6:8]¶
[1]·"Торговый·дом··Эдера·–·компания,·развивающая·стратегические·
  от\tно\tшения·с·партнерами,·находя·индивидуальный·подход·
  и·учитывая·не·только·свои·интересы,·но·и·интересы·
  каждого·из·партнеров,·для·обеспечения·взаимовыгодного·
  сотрудничества."
[2]·"Торговый·дом··Эдера·–·компания,·развивающая·стратегические·
  отношения·с·партнерами,·находя·индивидуальный·подход·и·
  учитывая·не·только·свои·интересы,\tно\ти·интересы·каждого·из·
  партнеров,·для·обеспечения·взаимовыгодного·сотрудничества."
[3]·"Торговый·дом··Эдера·–·компания,·развивающая·стратегические·
  отношения·с·партнерами,·находя·индивидуальный·подход·и·
  учитывая·не·только·свои·интересы,·но·и·интересы·каждого·
  из·партнеров,·для·обеспечения·взаимовыгод\tно\tго·
  сотрудничества."
```

To get rid of false hits, you might decide to find "но" only when it is used with non-word characters around it (because you can't use "\\b") and define a search expression such this one:

```
>·(search.expression<-paste0(russ.char.no,·search.word,·
  russ.char.no))¶
[1]·"[^ЁА-яё](но)[^ЁА-яё]"
```

But you already know that would cause problems with matches that are at the beginnings or ends of lines because then there are no preceding or subsequent characters respectively. You have therefore two choices: Either you define a better word boundary or you use negative lookaround. I will illustrate both options using the function `exact.matches.2` from above.

For a better word boundary, you can add beginnings and ends of character strings to the non-Cyrillic-character range you defined earlier:

```
>·(word.boundary<-paste0("(^|$|",·russ.char.no,·")"))¶
[1]·"(^|$|[^ЁА-яё])"
```

and then create and use a new search expression:

```
>·(search.expression<-paste0(word.boundary,·search.word,·
  word.boundary))¶

[1]·"(^|$|[^ËA-яё])(но)(^|$|[^ËA-яё])"
>·exact.matches.2(search.expression,·corpus.file,·case.sens=FALSE)
  [[4]]¶
[1]·"Торговый·дом··Эдера·–·компания,·развивающая·стратегические·
  отношения·с·партнерами,·находя·индивидуальный·подход·и·
  учитывая·не·только·свои·интересы,\tно\ti·интересы·каждого·
  из·партнеров,·для·обеспечения·взаимовыгодного·
  сотрудничества."
[2]·"Молодежное·нижнее·белье·эконом·класса·DAMINNAD·(Россия):·
  трусы,·майки,·комплекты.·Эта·молодежная·дерзкая·серия·
  нижнего·белья·обеспечит·Вам·хорошее·настроение.·В·целом,·
  для·DAMINNAD·характерна·приверженность·к·комфортным·и·
  удобным·в·носке·вещам,\tно\tесть·и·стремление·к·
  оригинальности,·экстравагантности.·Ведь·DAMINNAD·–·это·
  важнейшее·средство·самовыражения,·а·молодость·–·именно·то·
  время,·когда·позволено·все·и·когда·любая·одежда·тебе·к·
  лицу.·В·ОБЩЕМ,·МОЛОДО·И·ПОЗИТИВНО!!!"
```

As you can see, now only the one desired match in the third line is found plus another one in the fifth line.

The alternative involves negative lookaround: You look for the search word, but only if there is no character from the Cyrillic character range in front of it and no character from the Cyrillic character range after it. As you can see, the same matches are returned:

```
>·(search.expression<-paste0("(?<!",·russ.char.yes,·")",·
  search.word,·"(?!",·russ.char.yes,·")"))¶
>·exact.matches(search.expression,·corpus.file,·
  case.sens=FALSE)[[4]]¶
```

However, since none of these matches actually involves a match at the beginning or end of a string, let's just make sure that the former approach, the one involving the well-defined word boundaries, works by looking for the Russian translation of *in*, "в":

```
>·search.word<-"(в)"¶
```

If you define the search expressions just like above and then perform both searches with `exact.matches.2`, you will see that matches of "в" within words are not returned – as desired – but that the line-initial match in the eighth line is found:

```
>·(search.expression<-paste0(word.boundary,·search.word,·
  word.boundary))¶

>·(search.expression<-paste0("(?<!",·russ.char.yes,·")",·
  search.word,·"(?!",·russ.char.yes,·")"))¶
```

This concludes our brief excursus on Unicode character searches in this section. However, since this is an important issue, I would like to encourage you to not only get more familiar with this kind of encoding, but also do Exercise box 3.8 now.

You should now do "Exercise box 3.8: Unicode."

### *3.8.2 Some Notes on Handling XML Data*

XML, the e**X**tensible **M**arkup **L**anguage, is one of the most widely used formats that corpora come in; it is typically the format of choice in which information in the form of markup and annotation (recall Section 2.1.2) is added to the transcripts or written texts that corpora contain. It is therefore useful to at least briefly discuss how such data can be processed in R. For my discussion of XML below, I provide an input file that is a very small subpart of the file <D94.xml> from the BNC World Edition, but I will show the code and results you would obtain if you run the relevant code on the complete file; since the BNC World Edition in XML format is now freely available (from http://ota.ox.ac.uk/desc/2554), I reiterate my strong recommendation that you download this corpus: You can then use the complete version D94.xml here and, more importantly, for the many case studies in the next chapter that use it. If for some reason you cannot get your hands on the BNC, you can access a random sample from the BNC XML World Edition at the companion website of this book, but because of the sampling process, you can only do everything with it that treats it as a flat text file (as many case studies in Chapter 5 do), but applications that presuppose the hierarchical nature of the actual XML annotation (e.g., anything using the packages XML and xml2) will not work.

Space precludes an exhaustive discussion of the XML format, but a few comments about it are probably in order. XML is a hierarchical format (that lends itself well to representation as a tree that does not allow cross-nesting/overlapping) in which you add to data markup and annotation in the form of either start and end tags (which may contain attribute–value pairs), or just start tags (with attribute–value pairs), and in fact you have seen examples above that are similar to that annotation already. For instance, consider Figure 3.6, which shows how the word "It" is annotated in the BNC World Edition:

```
<w·c5="PNP"·hw="it"·pos="PRON">It</w>
```

*Figure 3.6* The XML representation of "It" in the BNC World Edition.

Everything between the first set of angular brackets is the start tag, the word "It" is the so-called data value, and everything between the second set of angular brackets that begins with "/" is the end tag. The start tag contains three attribute–value pairs:

- c5 = "PNP" (which means "the CLAWS 5 tag of the word annotated here is PNP", which means "personal pronoun");
- hw = "it" (which means "the lemma/headword of the word annotated here is *it*");
- pos = "PRON" (which means "the more coarse-grained POS tag of the word annotated here is PRON", which means "pronoun").

An example where no end tag is used – these are sometimes referred to as empty elements – can be seen in Figure 3.7, which is an opening tag (providing the information that there are 37 sentence tags (<s . . . >) in this file, and then the tag 'closes itself' with the "/>" at the end:

```
<tagUsage·gi="s"·occurs="37"/>
```

*Figure 3.7* The XML representation of the number of sentence tags in the BNC World Edition.

Note that one can theoretically store information in either format: the information shown in Figure 3.6 might theoretically just as well be represented as in Figure 3.8 – you have to know what the corpus you want to work with looks like:

```
<w·c5="PNP"·hw="it"·pos="PRON"·word="It"/>
```

*Figure 3.8* A hypothetical XML representation of "It" in the BNC World Edition.

If you want to process XML files with R, there are two ways in which you can do that. One is simply what we did in the previous sections of this chapter: We treat the XML file as a regular flat text file (essentially not paying much attention to the hierarchical structure of the file) and use regular expressions to find exactly what we want. Often this is fast and unproblematic and I must admit that the vast majority of my work with XML files is really just that. The other way is using (one of) two R packages that are dedicated to XML processing: You can use the package XML (www.omegahat.net/RSXML; I used version 3.98-1.4) and/or the package xml2 (https://github.com/hadley/xml2/issues; I used version 1.0.0). In general, the former is more powerful than the latter, which is why I will focus on it here, but there is one problem: On the Windows operating system the XML package has a memory leak which makes it impossible to handle larger amounts of data with it – the code that runs fine on Linux will make your computer crash very quickly on Windows. Thus, for some of the case studies below, I will provide a solution that uses the more powerful XML package, but also provide a solution using the xml2 package, which does not have that problem.

XML files in the BNC are stored as UTF-8 text files, which you can either load with `scan` and then convert using the function `xmlInternalTreeParse` from the XML package, or you can even load the file directly from the hard drive with `xmlInternalTreeParse`; I am showing the former here because it allows you to have both versions of the file in memory with only one hard drive access:

```
>·library(XML)¶
>·D94.flatfile<-scan(file.choose(),·what=character(),sep="\n")¶
>·D94.file<-xmlInternalTreeParse(D94.flatfile)¶
```

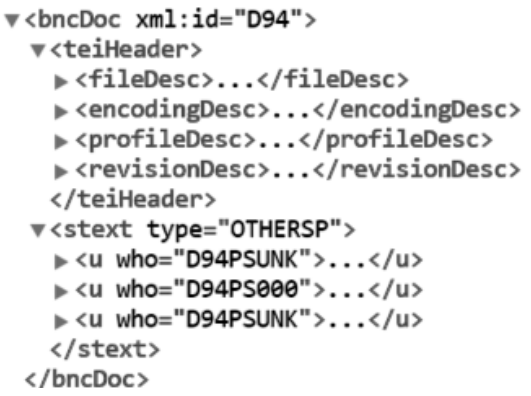


*Figure 3.9* The topmost hierarchical parts of BNC World Edition: <corp_D94_part.xml>.

A summary of `D94.file` will provide you with frequency tables of attribute names and nodes (output not shown here), which is already interesting: If all you wanted was the number of words, you could already get it from there:

```
>·summary(D94.file)¶
>·summary(D94.file)[["nameCounts"]]["w"]¶
```

However, you will usually want to define a structure containing the root of the XML file:

```
>·D94.root<-xmlRoot(D94.file)¶
```

Let us now look at the structure of an XML file. If you open the file <D94.xml> (or, if necessary, <_qclwr2/_inputfiles/corp_D94_part.xml>) in a browser, you will see a hierarchical representation (by means of indentation) of the file contents. As Figure 3.9 shows for <corp_D94_part.xml>, the file has a so-called root element called bncDoc, which has two parts: the TEI header and a part called "stext" that contains the (spoken) corpus text. The header, in turn, has four parts, and the stext part of <corp_D94_part.xml> contains three utterances.

You can explore the root a bit by, for instance, running `xmlName(D94.root)¶` or `xmlAttrs(D94.root)¶`, but for our purposes this does not buy us much – it is more interesting to see how we can access the information we see in Figure 3.9 about the children of the root; note the list-like notation with double square brackets:

```
>·xmlSize(D94.root)¶
[1]·2
>·names(D94.root)¶
··teiHeader·····stext·
"teiHeader"·····"stext"·
```

```
>·xmlName(D94.root[[1]])¶
[1]·"teiHeader"
>·xmlName(D94.root[[2]])¶
[1]·"stext"
```

You can also check out the contents of both of the root's parts, either as a hierarchically represented object quite similar to what you see in a browser or as a one-element character vector (output not shown):

```
>·D94.root[[1]]¶
>·xmlValue(D94.root[[1]])¶
>·D94.root[[2]]¶
>·xmlValue(D94.root[[2]])¶
```

Now, how do you explore the header more? The header of <corp_D94_part.xml> is shown in Figure 3.10, and the R code that gives you access to the number of elements the header consists of and their names is this:

```
>·xmlSize(D94.root[[1]])¶
>·names(D94.root[[1]])¶
>·D94.root[[1]][3:4]¶
```

What about the main corpus part, stext? As you can see in Figure 3.11, the immediate constituents of stext are the utterances, so you can use the same code you used above to

```
▼<bncDoc xml:id="D94">
  ▼<teiHeader>
    ▼<fileDesc>
      ▶<titleStmt>...</titleStmt>
      ▶<editionStmt>...</editionStmt>
       <extent>14 w-units; 3 s-units</extent>
      ▶<publicationStmt>...</publicationStmt>
      ▶<sourceDesc>...</sourceDesc>
     </fileDesc>
    ▼<encodingDesc>
      ▶<tagsDecl>...</tagsDecl>
     </encodingDesc>
    ▼<profileDesc>
       <creation date="1991">1991-09-04</creation>
      ▶<settingDesc>...</settingDesc>
      ▶<textClass>...</textClass>
     </profileDesc>
    ▼<revisionDesc>
       <change date="2006-10-21" who="#OUCS">Tag usage updated for BNC-XML</change>
     </revisionDesc>
   </teiHeader>
  ▶<stext type="OTHERSP">...</stext>
 </bncDoc>
```

*Figure 3.10* The header of <corp_D94_part.xml>.

```
▼<bncDoc xml:id="D94">
  ▶<teiHeader>...</teiHeader>
  ▼<stext type="OTHERSP">
    ▼<u who="D94PSUNK">
      ▼<s n="1">
          <w c5="PNP" hw="it" pos="PRON">It</w>
          <w c5="VM0" hw="would" pos="VERB">would</w>
          <w c5="XX0" hw="not" pos="ADV">n't</w>
          <w c5="VBI" hw="be" pos="VERB">be</w>
          <w c5="DT0" hw="any" pos="ADJ">any</w>
          <w c5="NN1-VVB" hw="bother" pos="SUBST">bother</w>
          <w c5="PRP" hw="to" pos="PREP">to</w>
          <w c5="PNP" hw="i" pos="PRON">me</w>
          <c c5="PUN">,</c>
          <w c5="CJS" hw="if" pos="CONJ">if</w>
          <w c5="PNP" hw="you" pos="PRON">you</w>
        </s>
      </u>
    ▼<u who="D94PS000">
      ▼<s n="2">
          <w c5="ITJ" hw="mm" pos="INTERJ">Mm</w>
          <c c5="PUN">.</c>
        </s>
      </u>
    ▼<u who="D94PSUNK">
      ▼<s n="3">
          <w c5="VVD" hw="want" pos="VERB">Wanted</w>
          <w c5="PNP" hw="i" pos="PRON">me</w>
          <w c5="TO0" hw="to" pos="PREP">to</w>
          <c c5="PUN">.</c>
        </s>
      </u>
      <u who="D94PS000"/>
    </stext>
  </bncDoc>
```

*Figure 3.11* The stext part of <corp_D94_part.xml>.

get at the parts of the header to now get at the parts of stext: just access `D94.root[[2]]`, and the subsetting allows you to target specific utterances:

```
>·xmlSize(D94.root[[2]])¶
>·names(D94.root[[2]])¶
>·D94.root[[2]][1]¶
>·D94.root[[2]][1:3]¶
```

But that still doesn't give us easy access to a specific utterance and its characteristics and parts, or content. For brevity's sake, let's define an object containing the first utterance and look at it in more detail: we can retrieve the code of the speaker who said it ("D94PSUNK", i.e., the speaker is unknown), the name of the utterance tag ("u", from "<u who= . . . "), the size of the utterance in terms of its immediate constituents (i.e., one sentence), and what the actual data value of the utterance is:

```
>·utt1<-D94.root[[2]][[1]]¶
>·xmlAttrs(utt1)¶
·······who
"D94PSUNK"
>·xmlName(utt1)¶
[1]·"u"
>·xmlSize(utt1)¶
```

```
·[1]·1
>·xmlValue(utt1)¶
[1]·"It·wouldn't·be·any·bother·to·me,·if·you"
```

However, the most useful ways of accessing corpus data from such XML files are either of the following:

- The function `xpathSApply` with a first argument being the object returned by `xmlInternalTreeParse` (above that was the object `D94.file`), the second argument defining the hierarchical level at which you want to operate, and an optional third and fourth argument being a function that you want to apply there and maybe additional arguments for that function.
- The function `xmlSApply` with a first argument being the root of an object returned by `xmlInternalTreeParse` (above that was the object `D94.root`) and the second argument being a function that you want to apply to `D94.root`'s parts.

For instance, both approaches can be used to tell you the two parts that <D94.xml> has:

```
>·xpathSApply(D94.file,·"/bncDoc/*",·xmlName)¶
[1]·"teiHeader"·"stext"
>·xmlSApply(D94.root,·xmlName)¶
··teiHeader·····stext
"teiHeader"·····"stext"
```

The former line of code can be read as "apply XPath syntax to all elements of `D94.file` at the level right below the bncDoc level (i.e., the topmost one), and for each of the elements at that level, return its name." In the remainder of this section I will focus on how to use `xpathSApply` because I find it more straightforward and useful to work with; in particular, we will mostly use the following kind of command (don't enter this, there's no ">·":

```
xpathSApply(xmlInternalTreeParse(some.file),
  "location.in.hierarchy",
  FUN,·...)
# FUN will usually be one of these 3 functions:
# xmlValue
# xmlAttrs
# xmlGetAttr
```

(Often, the code file will sometimes also provide code alternatives using `xmlSApply` and another function called `getNodeSet`, but if you want to you can skip those; the focus really is on the above approach with `xpathSApply`.) Most of the time now, I will not show the output because it is often too voluminous.

Given the above, you will not find it surprising that you can target the next lower levels like this: all that changes is the location description, which essentially uses a slash notation that is reminiscent of folder structures on your computer:

```
>·xpathSApply(D94.file,·"/bncDoc/teiHeader/*",·xmlName)¶
>·xpathSApply(D94.file,·"/bncDoc/teiHeader/*",·xmlSize)¶
```

You can see the parts of the header and how many parts each of them has. More interestingly, we can apply the same logic to the other part of the root, namely stext, and use `xmlValue` to immediately get the content of the corpus file without markup/annotation and without ever having had to use any regex:

```
>·xpathSApply(D94.file,·"/bncDoc/stext",·xmlValue)¶
```

Instead of an absolute path, where you move down the XML tree via the single slash-separated parts, you can also use relative paths indicated by two slashes to skip multiple intervening levels. The following jumps right down to the utterance level in stext and accesses each utterance separately, which means you can apply all sorts of text processing functions from above to it (such as `nchar` to get utterance lengths in characters):

```
>·xpathSApply(D94.file,·"//u",·xmlValue)¶
>·nchar(xpathSApply(D94.file,·"//u",·xmlValue))¶
```

This also means you can access any other level right away: sentences, words, punctuation marks, or even info from the header about the nature of the data, which here is spoken conversation (check out Figure 3.11 to verify the output):

```
>·xpathSApply(D94.file,·"//s",·xmlValue)¶
>·xpathSApply(D94.file,·"//w",·xmlValue)¶
>·xpathSApply(D94.file,·"//c",·xmlValue)¶
>·xpathSApply(D94.file,·"//classCode",·xmlValue)¶
```

You can check the code for a few more examples of such relative paths, which also exemplify the use of `xpathSApply` without a third argument. Also, note that you can move up one or more levels from the one you specified and you can provide alternatives, as exemplified in the following two examples. The first uses the ".." notation from terminal/command prompts to access the level above fileDesc (which is teiHeader, see Figure 3.10); the second uses the "|" notation we know from logical expressions above to access both listed alternative parts:

```
>·xpathSApply(D94.file,·"//fileDesc/..")¶
>·xpathSApply(D94.file,·"//profileDesc·|·//revisionDesc")¶
```

You can also immediately identify the lengths of all utterances (in terms of their components, i.e., sentences) and the speakers of all utterances, which are extracted from the who=" . . . " attribute–value pair of the utterance tag (see Figure 3.11 again), hence `xmlAttrs`:

```
>·xpathSApply(D94.file,·"//u",·xmlSize)¶
>·xpathSApply(D94.file,·"//u",·xmlAttrs)¶
```

The last example was a first step toward getting at the attribute–value pairs of the XML tags. Above, we mostly looked at retrieving the data values of our XML data, but of course we want to also use the often detailed annotation that is within the tags. The above approach with `xpathSApply(...,·...,·xmlAttrs)` works best if tags have just one attribute, as the utterance tags do, but it is more cumbersome when elements have more attributes; see the code file for the matrix output you get from this:

```
>·xpathSApply(D94.file,·"//w",·xmlAttrs)¶
```

For instance, we have seen that words have much more annotation and provide three attribute–value pairs: two with differently detailed POS tags and one with the word's lemma. How do we retrieve those easily? By using the function `xmlGetAttr` with a specification of which attribute we want (which I hope is reminiscent of the function `attr` discussed above):

```
>·xpathSApply(D94.file,·"//w",·xmlGetAttr,·"hw")¶
>·xpathSApply(D94.file,·"//w",·xmlGetAttr,·"pos")¶
>·xpathSApply(D94.file,·"//w",·xmlGetAttr,·"c5")¶
>·xpathSApply(D94.file,·"//u",·xmlGetAttr,·"who")¶
```

Finally, let us do some more advanced searches, searches that tap into different kinds and levels of annotation at the same time. A feature that they use is checking whether an attribute at some level has a certain value or not. The first of the following two lines should be read as "apply to those utterance elements of `D94.file` whose who-value (i.e., speaker) is 'D94PS000' the function `xmlValue` (which extracts the data value)", which returns a nice character value of D94PS000's utterances. The second line retrieves all words whose c5 tag says they are modal verbs:

```
>·xpathSApply(D94.file,·"//u[@who·=·'D94PS000']",·xmlValue)¶
>·xpathSApply(D94.file,·"//w[@c5·=·'VM0']",·xmlValue)¶
```

Note how the logical expression that checks whether the c5 tag is "VM0" uses single quotes to keep them separate from the opening double quote before `//w`. The following

examples work the same way but add a layer of complexity in that they do not just return the data value of the elements that meet a certain condition, but their attribute values (POS tags in the first, lemmas in the second):

```
>·xpathSApply(D94.file,·"//w[@c5·=·'VM0']",·xmlGetAttr,·"pos")¶
>·xpathSApply(D94.file,·"//w[@c5·=·'ITJ']",·xmlGetAttr,·"hw")¶
```

Finally, let's recognize that we can combine logical conditions both on one level of hierarchical organization and on different levels. The first example determines how often *will* is used as a modal verb by combining two conditions on the level of the word tag, namely the c5 and the hw attribute–value pairs; the second one determines how often a certain speaker uses the lemma *will* as a modal verb by combining one condition on the utterance level with two additional ones on the word level; the third adds yet another condition on the sentence level to that:

```
>·length(xpathSApply(D94.file,·"/bncDoc/stext/u/s/w[@c5·=·'VM0'·and·
  @hw·=·'will']",·xmlValue))¶
>·xpathSApply(D94.file,·"/bncDoc/stext/u[@who·=·'D94PS000']/s/w
  [@c5·=·'VM0'·and·@hw·=·'will']",·xmlValue)¶
>·xpathSApply(D94.file,·"/bncDoc/stext/u[@who·=·'D94PS000']/
  s[@n·=·'31']/w[@c5·=·'VM0'·and·@hw·=·'will']",·xmlAttrs)¶
```

As you can see, this package allows us to do very powerful searches of the kind we'd expect from a database. Many of them, for instance those just targeting words and their tags, could be done with regular expressions, but the kinds of searches exemplified here at the end show how useful some knowledge of how to use R for XML with XPath searches really is because they allow you to combine information from different levels of annotation relatively easily – all of the above is possible even when you treat an XML file as a text file – but trust me, it's painful. In what follows, I provide a very brief overview of a few additional ways in which the XML package can help you search XML corpus data in sophisticated ways. I will focus in particular on three things: (1) how speaker-specific information can be retrieved; (2) how you can search and retrieve information from different levels of the XML tree; and (3) some XPath syntax aspects that allow for simple character processing. The examples in this part will be based on the BNC Word Edition file <_qclwr2/_inputfiles/corp_H00.xml>, part of whose header is shown in Figure 3.12; note how this file contains data from three speakers – PS2AB, PS2AC, and PS2AD – for whom different kinds of information are available.

Given the previous discussion, the following should be mostly obvious ways of retrieving all information regarding the speakers as lists (lines 1 and 2), retrieving all the names of all the attributes provided for each speaker (line 3, which previews the notion of an anonymous function you will learn more about in Section 3.10), and retrieving two specific characteristics for each speaker (lines 4 and 5):

```
▼<bncDoc xml:id="H00">
  ▼<teiHeader>
    ▶<fileDesc>...</fileDesc>
    ▶<encodingDesc>...</encodingDesc>
    ▼<profileDesc>
       <creation date="0000">0000-00-00 Origination/creation date not
       known</creation>
      ▼<particDesc n="C268">
         <person ageGroup="X" xml:id="PS2AB" role="unspecified" sex="f"
         soc="UU" dialect="NONE" educ="X"/>
        ▼<person ageGroup="X" xml:id="PS2AC" role="unspecified" sex="f"
         soc="UU" dialect="XSD" firstLang="EN-GBR" educ="X">
           <occupation>carpet factory worker</occupation>
         </person>
        ▼<person ageGroup="X" xml:id="PS2AD" role="unspecified" sex="m"
         soc="UU" dialect="NONE" educ="X">
           <persName>Ken</persName>
         </person>
      </particDesc>
    ▶<settingDesc>...</settingDesc>
    ▶<textClass>...</textClass>
    </profileDesc>
  ▶<revisionDesc>...</revisionDesc>
  </teiHeader>
▶<stext type="OTHERSP">...</stext>
</bncDoc>
```

*Figure 3.12* The teiHeader/profileDesc part of <corp_H00.xml>.

```
>·xpathSApply(H00.file,·"//person")¶
>·xpathSApply(H00.file,·"//person",·xmlAttrs)¶
>·xpathSApply(H00.file,·"//person",·function(x)·names(xmlAttrs(x)))¶
>·xpathSApply(H00.file,·"//person",·xmlGetAttr,·"sex")¶
>·xpathSApply(H00.file,·"//person",·xmlGetAttr,·"dialect")¶
```

The next few lines should be similarly obvious: They apply the logic of logical expressions in `xpathSApply` applied above to sentences, words, etc. now to the person tag in the header: the following lines retrieve all the attributes of the (one and only) male speaker (line 1), his "id" attribute which is the code all his utterance tags will use in the who-attribute (line 2), and the data value of this tag, which here turns out to be his name, "Ken" (line 3):

```
>·xpathSApply(H00.file,·"//person[@sex·=·'m']",·xmlAttrs)¶
>·xpathSApply(H00.file,·"//person[@sex·=·'m']",·xmlGetAttr,·"id")¶
>·xpathSApply(H00.file,·"//person[@sex·=·'m']",·xmlValue)¶
```

Note that you can unfortunately not expect complete consistency in the data all the time: If you tweak the above line to retrieve data for the female speakers (by replacing ='m' with ='f'), then you will see that `xmlValue` does not return the female speakers' names but their occupation.

Once you know the male speaker's id, it is easy to retrieve what he said in sentences (line 1) or in words (line 2):

```
>·xpathSApply(H00.file,·"/bncDoc/stext/u[@who·=·'PS2AD']/s",·
  xmlValue)¶
>·xpathSApply(H00.file,·"/bncDoc/stext/u[@who·=·'PS2AD']//w",·
  xmlValue)¶
```

and we can combine this kind of query with everything we discussed above to, for instance, find out which inflectional forms of the verb lemma *work* the male speaker uses:

```
>·xpathSApply(H00.file,·"//u[@who·=·'PS2AD']//w[@pos·=·'VERB'·and·
  @hw·=·'work']",·xmlValue)¶
[1]·"working·"·"working·"
```

Let me finally mention that XPath also allows you to exploit more features of the hierarchical tree that an XML file represents. First, you can use so-called *XPath axes* which define the relationships between different nodes in an XML tree using kinship and ordering terms such as *ancestor*, *child*, *sibling*, or *parent* on the one hand and *preceding* and *following* on the other. Thus, the following line retrieves from `H00.file` the utterances by the speaker with the id "PS2AD", specifically sentence 162, looks for words whose POS tag is "VERB" and whose lemma is "work", and then recovers the data values of the preceding words (which are siblings of *work* by virtue of being in the same sentence):

```
>·xpathSApply(H00.file,·"//u[@who·=·'PS2AD']/s[@n·=·'162']/
  w[@pos·=·'VERB'·and·@hw·=·'work']/
  preceding-sibling::w",·xmlValue)¶
```

See the code file for another similar example. Along the same lines, the next example does something similar but does not retrieve data values, but values of an attribute, namely lemma annotation. Specifically, it retrieves from `H00.file` the utterances by the speaker with the id "PS2AD", looks for words whose POS tag is "VERB" and whose lemma is "work", and then recovers the hw attributes (i.e., lemmas) of the next preposition(s) (which, too, are siblings of *work* by virtue of being in the same sentence):

```
>·xpathSApply(H00.file,·"//u[@who·=·'PS2AD']//
  w[@pos·=·'VERB'·
  and·@hw·=·'work']/following-sibling::w[@pos·=·'PREP']",·
  xmlGetAttr,·"hw")¶
```

Second, XPath allows you to use so-called *predicates* to utilize numerical and textual characteristics of the data for your searches. Such predicates work similarly to the logical expressions you already know from above (e.g., "u[@who = 'PS2AD']") by stating a

condition and selecting those data values or attributes etc. for which the condition is true. Space does not permit an exhaustive discussion of everything XPath has to offer, so a few examples of predicates, whose names make it very obvious what they do, shall suffice. The next two lines, for example, return all words that have more than six characters, which may include spaces (line 1) and all words beginning with the character "q" (line 2):

```
>·xpathSApply(H00.file,·"//w[string-length()>6]",·xmlValue)¶
>·xpathSApply(H00.file,·"//w[starts-with(text(),·'q')]",·
  xmlValue)¶
```

The next two lines return all sentences that contain a word beginning with a "q" (line 1) and their sentence numbers in the file (line 2):

```
>·xpathSApply(H00.file,·"//w[starts-with(text(),·'q')]/
  parent::s",·xmlValue)¶
>·xpathSApply(H00.file,·"//w[starts-with(text(),·'q')]/
  parent::s",·xmlGetAttr,·"n")¶
```

Finally, the next two lines retrieve all sentences beginning with "Was" (line 1) and all words beginning with "w" in sentences beginning with "Was" (line 2):

```
>·xpathSApply(H00.file,·"//w[starts-with(text(),·'Was')]/
  parent::s",·xmlValue)¶
>·xpathSApply(H00.file,·"//w[starts-with(text(),·'Was')]/
  parent::s/w[starts-with(text(),·'w')]",·xmlValue)¶
```

Again, I hope it becomes obvious why having even only the most basic knowledge of XPath and how to use it in R is a good thing for any corpus linguist. Let me finally very briefly mention the package xml2, which is useful to avoid the XML package's memory leak on Windows. With that package, you can load an XML file as follows:

```
>·library(xml2)¶
>·(D94.file<-read_xml(file.choose()))¶
```

You can then access some information regarding the structure and parts of this object using the following set of functions:

```
>·xml_structure(D94.file)¶
>·xml_children(D94.file)¶
>·xml_name(D94.file)¶
>·xml_text(D94.file,·trim=TRUE)¶
```

but for our purposes the most important function is `xml_find_all`, which does what you might expect: It finds all elements of a particular type/hierarchy level, which means we can easily retrieve all sentences (line 1) or all words (line 2):

```
>·xml_find_all(D94.file,·".//s")¶
>·xml_find_all(D94.file,·".//w")¶
```

We can also very easily convert these results into character vectors that we can then handle with regular expressions. The following line retrieves all adjectives:

```
>·exact.matches.2("(?<=\"ADJ\">).*?(?=·?<)",·
  as.character(xml_find_all(D94.file,·".//w")),
  gen.conc.output=FALSE)[[1]]¶
```

**Recommendations for further study/exploration**

- On how to convert XML data structures into lists and data frames: `?xmlToDataFrame` and `?xmlToList¶`.
- On using R to process XML data: Nolan & Temple Lang (2014) and Munzert, Rubba, Meißner, & Nyhuis (2015, in particular chs. 3–4).

The next section introduces a few functions to handle files and folders; some of these will do things you usually perform using parts of your operating system such as Explorer in MS Windows.

## 3.9 File and Directory Operations

You have already encountered one of the most important functions. On all three major kinds of operating systems, Windows, Mac OS X, and Linux, you can use the function `file.choose` to interactively choose one file in a file manager window. On Windows, the additional function `choose.files` allows you to interactively choose one or more files for either reading or writing (or to create one new one); on Mac OS X and Linux, this function is unfortunately not available, but there is a bit of a workaround based on the package rChoiceDialogs, which requires the package rJava and which you can load as follows:

```
>·library(rChoiceDialogs)¶
```

This package provides a useful function called `rchoose.files`, which allows you to interactively choose multiple (already existing!) files and returns the paths of the files you chose as a character vector (which you can then loop over, for instance, to load each file with `scan`).

Other important functions have to do with directories. First, the package rChoiceDialogs again provides a useful function called `rchoose.dir`, which allows you to interactively choose an existing directory (and on Windows even to create a new one). Then, you need to know a bit about R's *working directory*, i.e., the directory that is accessed if no other directory is provided in a given function. By default, i.e., if you haven't changed R's default settings, then this is either R's installation directory, the logged-in user's main data directory, or, if you use RStudio and double-clicked on an R script file (<. . . .r>), then it's the directory that file is located in; most of the time while you're working through this book, it should therefore be <_qclwr2/_scripts>, which is what some output operations in Chapter 5 assume. You can easily find out what the current working directory is by using `getwd` without any arguments at the prompt:

```
>·getwd()¶
[1]·"home/stgries/_qclwr2/_scripts"
```

If you want to change the working directory at the console, you use `setwd` and provide as the only argument a character string with the new working directory (either manually or again with `rchoose.dir()`). Note that when you do this in RStudio on Windows, the Explorer window sometimes doesn't come to the foreground so you may have to Alt-Tab your way to it (or find it in the taskbar).

```
>·setwd(rchoose.dir())¶
>·getwd()¶
```

If you now want to find out which files are in your working directory, you can use `dir`. The default settings of `dir` are:

```
dir(path=".",·pattern=NULL,·all.files=FALSE,·full.names=FALSE,·
  recursive=FALSE)
```

The first one specifies the directory whose contents are to be listed; if no argument is provided, the current working directory is used. The second argument is an optional regular expression with which you can restrict the output. If, for example, you have all 4,049 files of the BNC World Edition (the XML version) in one directory, but you only want those file names starting with "D", this is how you could proceed:

```
>·dir(rchoose.dir(),·pattern="^D")¶
```

The default setting of the third argument returns only visible files; if you change it to `TRUE`, you would also get to see all hidden files. The default setting of the fourth argument only outputs the names of the files and folder without the path of the directory you are searching. If you set this argument to `TRUE`, you will get the complete path for every file and

folder, which is usually recommended. By the way, the function `basename` does exactly the opposite: It cuts off all the path information.

```
>·basename("/_qclwr2/bestbookever.txt")¶
[1]·"bestbookever.txt"
```

Finally, the default setting `recursive=FALSE` only returns the content of the current directory, but if you set this to `TRUE`, the contents of all subdirectories will also be returned. This is of course useful if your corpus files come in a particular directory structure. For example, you can use the following line in a script to prompt the user to input a directory and then store the names of all files in this directory and all its subdirectories into one vector called `corpus.files`, and we will do just that very often in Chapter 5:

```
>·corpus.files<-dir(rchoose.dir(),·recursive=TRUE,·full.
  names=TRUE)¶
```

Apart from the number of files in a directory and their names, however, you can also retrieve much more useful information using R. The function `file.info` takes as input a vector of file names and outputs whether something is a file or a directory, the sizes of the files, the file permissions, and dates/times of file modification, creation, and last access. Thus, the easiest and most comprehensive way to take a detailed look at your working directory is therefore:

```
>·file.info(dir())¶
```

There are some other file management functions, which are all rather self-explanatory; I will only provide some of the options here and advise you to explore the documentation for more details:

- `file.create(x)` creates the file(s) in the character vector `x`;
- `dir.create(x)` creates the directory that is provided as a character string either as a directory in your working directory or as a relative or rooted directory; thus, `dir.create("temp")` will create a directory called "temp" in your current working directory and `dir.create("../temp")` will create a directory called "temp" one directory above your current working directory;
- `unlink(x)` deletes the files/directories in the character vector `x`; `file.remove(x)` can only delete files;
- `file.exists(x)` checks whether the file(s) in the character vector `x` exist(s) or not;
- `download.file(x,·...)` downloads the website at the URL `x` (e.g., <http:// . . . >) into the file . . . specified).

Sometimes, you need to use a data structure, such as a data frame or a list, again and again; as such, you want to save it into a file. While you can always save data frames as

tab-delimited text files with `write.table`, which are easy to use with other software, sometimes your files may be too large to be opened with other software (in particular spreadsheet software such as Microsoft Excel or LibreOffice Calc). The option of using raw text files may also not be easily available with lists. Finally, you may want to save disk space and not only save your data, but also compress them at the same time. In such cases, you may want to use `save`. In its simplest syntax, which is the only one we will deal with here (see the documentation for more detailed coverage), you can save any object into a binary compressed file, which will usually be much smaller than the corresponding text file and which, if you gave it the extension ".RData", makes your data available to a new R/RStudio session upon a simple double-click. As an example, consider a data frame `aa` with the following structure (details such as factor levels etc. are omitted):

```
'data.frame':···57437·obs.·of··12·variables:
·$·PERSON···:·Factor·w/·12·levels·...
·$·PERSON2··:·Factor·w/·2·levels·...
·$·FILE_ID··:·num·...
·$·FILE_LINE:·int·...
·$·VERB·····:·Factor·w/·6754·levels·...
·$·LEMMAS···:·Factor·w/·2451·levels·...
·$·VERBTAG··:·Factor·w/·40·levels·...
·$·ASPECT···:·Factor·w/·2·levels·...
·$·ASPECT2··:·Factor·w/·2·levels·...
·$·TENSE····:·Factor·w/·2·levels·...
·$·TENSES1··:·Factor·w/·2·levels·...
·$·TENSES2··:·Factor·w/·3·levels·...
```

This data frame was loaded from a tab-delimited text file with a size of 5.45 MB (5,721,842 bytes). The following function call would save this file into a user-specified file, which should have the extension. RData (e.g., <aa.RData>) with a size of 579 KB (593,822 bytes) that, if double-clicked, opens a new instance of R and makes the data available instantly:

```
save(aa,·file="aa.RData")¶
```

If you have saved the data structure as an .RData file and want to open it in a new R session, you can just as well use `load` to access your data again:

```
load(file=file.choose())¶
```

You will be prompted to choose a file, and then the data structure(s) will be available (type `ls()¶` at the console to get a listing of all objects in your current workspace). Also, you can just save your whole R workspace, not just one data structure, into a user-definable file with `save.image`, which is useful to store intermediate results for later processing into an. RData file. This file can either be loaded from within R, or you can just double-click on them to open R and restore your workspace.

Let me finally mention how you would go about saving graphs generated in R. There are two possibilities. The first is to generate the graph in RStudio with whatever code is needed and save it by clicking the *Export* button in the relevant RStudio Plots pane, then click *Save as Image . . .* , choose a file format (I usually use .png), define the width and height of the plot in pixels or by drag-and-drop graph boundaries, and save it into the desired output file. The second is more flexible and does this with code: First, open a graphics device for the desired output format with the indicated dimensions in pixels (line 1), plot the file into that graphics device (line 2), and then close that device (line 3), as in this example:

```
>·png(filename="03-9_graph.png",·width=600,·height=600)¶
>····plot(1:10,·1:10,·type="b")¶
>·dev.off()¶
```

You can now open the file with an image viewer (e.g., GIMP) and look at it.

**Recommendations for further study/exploration**

- On how to read tab-delimited text files: `?read.csv¶`.
- On how to produce text representations of objects into files: `?dump¶` and `?dput¶`.
- On how to divert output to a file instead of the screen/console: `?sink¶`.
- On how to handle compressed files: `?bzfile¶` and `?gzfile¶`.

## 3.10 Writing Your Own Functions and Some Final Recommendations

The fact that R is not just a statistics software but a full-fledged programming language means you are not limited by the functions that already exist in R or in any package that you may install – you can fairly easily write your own functions to facilitate and/or automate tedious and/or frequent tasks. In this section I will give a few very small examples of the logic of how to write your own functions, which is something that will become extremely useful when we turn to the case studies in Chapter 5.

Let's imagine a scenario in which we want to generate a frequency table of a vector, which we would normally do with `table`, but we do not want the table sorted alphabetically (`table`'s default) or by frequency (which we would do with `sort(table(...))` – we want our table to be sorted by the order of occurrence in the input vector. Let's quickly generate a vector that contains a random sample of the first ten letters of the alphabet (see again `?letters¶`):

```
>·rm(list=ls(all=TRUE));·set.seed(1)¶
>·qwe<-sample(letters[1:10],·100,·replace=TRUE)¶
>·head(qwe,·20)¶
[1]·"c"·"d"·"f"·"j"·"c"·"i"·"j"·"g"·"g"·"a"·"c"·"b"·"g"·"d"·"h"·
  "e"·"h"·"j"·"d"·"h"
```

That means we want our frequency list to first give the frequency of the letter "c", then that of "d", then "f", then "j", then "i" (because "c" is already covered), etc. The easiest way to write a function consists of the following three steps:

1 You write the code that you would use if you didn't write a function (i.e., as we have always done so far).
2 You determine which of the data structures you are using in your code are required for the code/function to work.
3 You wrap a function definition around your code and (1) make sure that the function definition requests or defines the required data structures (and any others you may want to add) and (2) use names that are general and useful enough to cover all applications you have in mind for this function (this is only for readability/interpretability later).

We begin with step 1. If we have a vector `qwe` of which we want a frequency list, but one sorted in a particular way, then we should probably first generate a default frequency list:

```
>·(asd<-table(qwe))¶
qwe
·a··b··c··d··e··f··g··h··i··j
·7··6·11·14·14··5·11·15·11··6
```

But now we want to have this output ordered as mentioned above. That means we need to find for each letter that is actually attested in `qwe` – i.e., the names of the frequencies in `asd` – at what position it shows up in `qwe` for the first time. This formulation should remind you of the function `match`, which, to quote from above, "`match` returns a vector of the positions of (first!) matches of its initial argument in its second":

```
>·match(names(asd),·qwe)¶
·[1]·10·12··1··2·16··3··8·15··6··4
```

That means the letter "a" (the first name of `asd`) shows up in position 10 of `qwe`; the letter "b" (the second name of `asd`) shows up in position 12 of `qwe`, and, crucially, the letter "c" (the third name of `asd`) shows up in position 1 of `qwe`, etc. That in turn means two things: (1) we want the above output reordered so that the 1 (for "c") comes first, the 2 (for "d") comes second etc.; and (2) once we have that ordering of these numbers, we can then apply these numbers to re-order `asd`. Hence for (1):

```
>·order(match(names(asd),·qwe))¶
#·[1]··3··4··6·10··9··7··1··2··8··5
```

And from that, we can do (2), which is our desired outcome:

```
>·asd[order(match(names(asd),·qwe))]¶
qwe
·c··d··f··j··i··g··a··b··h··e
11·14··5··6·11·11··7··6·15·14
```

Now that we have completed step 1 of writing a function, we now turn to step 2 and look at all the code we wrote and determine which data structures are minimally required for this whole thing to work. It turns out the only data structure required is the input vector, which above was called `qwe`, and this is because there is only one other data structure we use, which is `asd`, but `asd` can be derived from `qwe`. Thus, we now can turn to step 3: We take all the code we wrote, but wrap it into a function definition (let's assume we want to call our function `table.1stocc`, for table sorted by first occurrence), ensure that the user is forced to provide an input vector, and use more revealing names than `qwe`, `asd`, and the like. In the code below, I assign what we would like the function to output – the sorted frequency table – to an object called `output`, which then, by ending the function definition with `return(output)`, is returned as the result of the function:

```
>·table.1stocc<-function(input.vector)·{¶
+····freq.table<-table(input.vector)¶
+····output<-freq.table[order(match(names(freq.table),·
  input.vector))]¶
+····return(output)¶
+·}¶
```

Now we can test our new function by giving it `qwe` as an argument. The function then maps `qwe` onto the function-internal more usefully named `input.vector` and executes everything without problems. If you save this little bit of code now, the function definition, then you never have to worry about how to do this again: If this task ever comes up (again), just remember you have this function, problem solved:

```
>·table.1stocc(qwe)¶
input.vector
·c··d··f··j··i··g··a··b··h··e
11·14··5··6·11·11··7··6·15·14
```

Let us briefly discuss one other interesting application, namely a function that we will use a lot in the scripts in Chapter 5, a function we will call `just.matches`. Let us first re-generate the example vector `txt` from above:

```
>·txt<-c("This·is·a·first·example·sentence.",
·········"And·this·is·a·second·example·sentence.")¶
```

Let's assume we want to find all words that end in "is"; this is how we would use `regmatches` and `gregexpr` for this when we want the result in a vector:

```
>·unlist(regmatches(txt,·gregexpr("\\w*is\\b",·txt,·perl=TRUE)))¶
[1]·"This"·"is"···"this"·"is"
```

Since finding exact matches like this (and maybe vectorizing them) is definitely a *very* frequent task for a corpus linguist, it would be nice if this could be done shorter and without having to repeat the argument `txt` twice. Thus, we could write a function `just.matches` that does this for us, and here's one way to go about this:

```
>·just.matches<-function(search.expression,·corpus,·pcre=TRUE,·
  vectorize=TRUE,·...)·{¶
+····output<-regmatches(corpus,¶
+·······gregexpr(search.expression,·corpus,·perl=pcre),...)¶
+····if(vectorize)·{·output<-unlist(output)·}¶
+····return(output)¶
+·}¶
```

The first line defines a function `just.matches`, but also specifies that it requires at least two arguments – the search expression and the input/corpus vector. However, the function definition also states that an additional argument `pcre` is by default – i.e, unless otherwise specified by the user – set to `TRUE`, as is another additional argument `vectorize`. That is, if the user does not change these settings, these are used by R when this function is run. Finally, there is an ellipsis at the end of the function definition, which stands for "whatever other arguments a user might want to provide to a certain location in the body of the function, where the ellipsis is taken up again".

The body of the function then defines an object `output` using nearly the exact same code we used manually before, but (1) it sets `gregepxr`'s `perl` argument to the value of `pcre` that is defined when `just.matches` is called (i.e., by default to `TRUE`); (2) it uses the value of `vectorize` to decide whether the user receives a vector (the default) or a list; and (3) it passes any arguments the user may have provided to `just.matches` on to the `regmatches` function, which means we can, for example, use `invert` in our function:

```
>·just.matches("\\w*is\\b",·txt)¶
[1]·"This"·"is"···"this"·"is"
>·just.matches("\\bexample\\b",·txt,·invert=TRUE)¶
[1]·"This·is·a·first·"·····"·sentence."···········"And·this·is·a·
  second·"·"·sentence."
```

(Of course, the former at least could also be done with the function I provide, `exact.matches.2`, which could also be tweaked to accept `invert` as well – if I ever needed that functionality, I might update `exact.matches.2` accordingly.) I hope you can see

by now how useful it can be to recognize that some task will be repeated a lot so it may be more efficient to define a function for it, which usually not only speeds up the process considerably, but also makes your code easier to read for you; we will use this function a lot of times in the case studies below – sometimes I will use `exact.matches.2`, sometimes `just.matches`, and sometimes both (to show how they would yield the same results).

You should now do "Exercise box 3.10: Writing your own functions."

The penultimate recommendation of this chapter is concerned with the very useful notion of so-called anonymous, or inline, functions. These are functions that are (1) defined as above but not given a name and (2) used as they are being defined, i.e., they are defined in the very same line in which they are applied. This can be extremely useful for doing small tasks for which no ready-made function is available. Let's consider an example: We want to find out for each sentence in a corpus how many unique words, how many types, it contains. The following two lines create a 'corpus' called `qwe` and split it up into 'words':

```
>·qwe<-c("a·b·c·d·e·f·g·f·e·d·c·b·a",·"m·n·o·m·n·o·m·n·o")¶
>·words<-strsplit(txt,·"\\W+",·perl=TRUE)¶
```

We would now want to use `sapply` to access each element of `words` and count the number of word types. But there is no function to count the number of unique word *types* – there is only a function to count the number of *tokens*, which is `length`. Thus, we can do the following but it doesn't give us what we want:

```
>·sapply(words,·length)¶
[1]·13··9
```

In other words, we are applying `length` to each element of `words`, but what we need is to apply `length` to the result of applying `unique` to each element of `words`, but there is no function for `length(unique(...))`. We could just solve this ad hoc by using the following bit of code:

```
>·sapply(sapply(words,·unique),·length)¶
[1]·7·3
```

It works, but for didactic purposes we want to talk about how to solve this with a function or, more importantly, an anonymous/inline function. Thus, we could define a function for that like we did above (`return` is not used here because it's not strictly speaking necessary – the function will automatically return the last-assigned object):

```
>·types<-function(some.vector)·{¶
+····length(unique(some.vector))¶
+·}¶
>·sapply(words,·types)¶
[1]·7·3
```

Or, because this may seem like overkill, we do the following, namely put the body of our definition of `types` into the second argument slot of `sapply` directly. In the following line, we apply to the elements of `words` a nameless (hence *anonymous*) function that requires one argument that is internally called `x` and to which R applies `length(unique(x))`; note that you of course don't have to use `x` – you can use any name, but just like with regular functions above, whatever name you use must be the one that is in the `...` slot of `length(unique(...))`. This way, we avoid having to define a function and just do the definition and application of the function in one line (hence *inline*). Note that the `x` in the function definition is not available outside of the function, which means it doesn't matter if there is or isn't an object called `x` before you run the line below, and it means that after you run the line below there is still no object called `x`. `x` is used for the duration of the anonymous function only and does not get retained in your general workspace.

```
>·sapply(words,·function(x)·length(unique(x)))¶
[1]·7·3
```

Anonymous functions can sometimes be very useful and we will use them on occasion.

Finally, most of this book uses only the functionality from base R so as to minimize users' dependence on functions in additional packages, which may introduce changes that are not backwards-compatible more often than base R will. However, one of the few exceptions I want to make to this practice involves an extremely useful operator from the package dplyr (https://github.com/hadley/dplyr), which is written as %>%. This operator can make code *much* more readable, in particular when your code involves a lot of nested functions.

```
>·library(dplyr)¶
```

Remember how we generated just the exact matches with our 'normal' R code? We did it like this (or see the more heavily commented version using several lines and indentation in the code file):

```
>·unlist(regmatches(txt,·gregexpr("\\w*is\\b",·txt,·perl=TRUE)))¶
[1]·"This"·"is"···"this"·"is"
```

The 'annoying' thing about this is that, if you want to understand what the code does, you have to process it from the inside out: The first thing that is done is the `gregexpr`

search, the result of which is then passed outside to `regmatches`, the result of which is then passed outside to `unlist` – not exactly intuitive, and this example involves just one short line and two nested function calls. The operator `%>%` can simplify this by making the order in which you write functions in the code the same as the order in which they are executed. Here is a very simple example, which you can read as 'take `txt` and to it apply `nchar`':

```
>·txt·%>%·nchar¶
[1]·33·38
```

This is the same as `nchar(txt)¶`: The function `nchar` requires one argument and the `%>%` operator takes its left-hand side (lhs, the vector `txt`) and makes it the first argument of its right-hand side (rhs, the function `nchar`). Now what if the lhs is not the first argument of the rhs? Then you can use a period to indicate where in the rhs the lhs should be, as in the following example, where we make `txt` the second argument in the call of `just.matches`. So this should be read as "take `txt` and give it to `just.matches` (as the second argument) to look for "\\w*is\\b" in it":

```
>·txt·%>%·just.matches("\\w*is\\b",·.)¶
[1]·"This"·"is"···"this"·"is"
```

So how does this apply to the above line returning just the matches? The following is read as "take `txt`, make it the thing to be searched by `gregexpr` (i.e., `gregexpr`'s second argument), take the result from `gregexpr` and make it the second (match data) argument of `regmatches`, take the result from that and `unlist` it":

```
>·txt·····································%>%·#·take·txt¶
+····gregexpr("\\w*is\\b",·.,·perl=TRUE)·%>%·#·find·"\\w*is\\b"¶
+····regmatches(txt,·.)··················%>%·#·give·to·regmatches¶
+····unlist·····························#·and·unlist¶
[1]·"This"·"is"···"this"·"is"
```

While this is of course not as reader-friendly and concise as if you already have a function `just.matches`, it is much more reader-friendly than the previous code version with the multiple embedding, because you read it from top to bottom, which is the order in which things are done, rather than having to read code from the inside out (rather than the usual left-to-right). Thus, check the code file for how sleek a definition of this `table.1stocc` this operator allows you to write. This operator takes a bit of getting used to and to make good use of it you have to remember that operators such as subsetting or subtracting can be written as functions (namely "`[`" and "`-`") etc., but it is such a useful alternative to multiple embeddings that several of the case studies in Chapter 5 will provide the traditional code, but also the `%>%` alternative.

## Notes

1 If you let R output vectors to the screen that unlike the above are so long that they cannot be shown in one row anymore, R will use as many lines as necessary to represent all elements of the vector, and each line will start with the number (in square brackets) of the first element of the vector that is given in each line. Try it out: enter `1:100¶` at the prompt.

2 This is actually a generally important concept: R shows missing or unavailable data as `NA`. Since this may happen for many different reasons, let me mention two aspects relevant in the present context. First, you can check whether (some elements of) something is/are not available using the function `is.na` which takes the element to be tested as its argument. For example, if the vector `a` consists only of the number 2, then `is.na(a)` gives `FALSE` (because there is something, the number 2), but `is.na(a[2])` ·gives `TRUE`, because there is no second element in `a`. Second, many functions have default settings for how to deal with `NA`, which is why the unexpected behavior of a function can often be explained by checking whether `NA`'s distort the results. To determine the default behavior of functions regarding `NA`, check their documentation.

3 The issue of how paragraph breaks are represented in files is actually more problematic than it might appear at first sight. Different operating systems mark paragraph breaks in text-processing software differently: Microsoft Windows uses a carriage return (CR) and a line feed (LF). In many programming languages, text editors, and also in R, the former is represented as "\r" while the latter – often also referred to as newline – is represented as "\n". Unfortunately, other operating systems use different characters to represent what for the user is a paragraph break: Linux/Unix systems, for instance, use only LF. In order to avoid errors with regular expressions, it is often useful to look at the file in a text editor that displays all control characters to determine which paragraph mark is actually used.

4 In a language with many-to-many correspondence of spelling and pronunciation such as English, this can of course only be a heuristic. However, this regular expression would certainly narrow down the search space considerably, and when applied to languages such as Spanish or to phonetic transcription, which could in fact be assigned to a text given a database such as CELEX, the regular expression below could do the trick.

5 The hexadecimal system is a numerical system that has 16 as its base (unlike the familiar decimal system, which of course has base 10). Instead of the ten numbers 0 to 9 that the decimal system uses, the hexadecimal system uses 16 distinct characters, 0–9 and a–f (or A–F). Thus, the decimal number 8 is also 8 in the hexadecimal system, the decimal number 15 is hexadecimal F, the decimal number 23 is hexadecimal 17 (namely $1 \times 16 + 7 \times 1$), and the decimal number 1970 is hexadecimal 7B2 (namely $7 \times 256 + 11 \times 16 + 2 \times 1$).

## References

Friedl, Jeffrey E.F. (2006). *Mastering regular expressions*. 3rd ed. Cambridge, MA: O'Reilly Media.

Goyvaerts, Jan, & Steven Levithan. (2012). *Regular expressions cookbook*. 2nd ed. Sebastopol, CA: O'Reilly Media.

Meyer, Charles F. (2002). *English corpus linguistics: An introduction*. Cambridge: Cambridge University Press.

Munzert, Simon, Christian Rubba, Peter Meißner, & Dominic Nyhuis. (2015). *Automated data collection with R: A practical guide to web scraping and text mining*. Chichester: John Wiley.

Nolan, Deborah, & Duncan Temple Lang. (2014). *XML and web technologies for data sciences with R*. New York: Springer.

Spector, Phil. (2008). *Data manipulation with R*. New York: Springer.

Stubblebine, Tony. (2007). *Regular expression pocket reference*. 2nd ed. Cambridge, MA: O'Reilly Media.

# 4 Some Basic Statistical Notions and Tests

> [I]t seems to me that future research should deal with frequencies in a much more empirically sound and statistical professional way. . . . In the long run, linguistic theorising without a firm grip on statistical methods will lead corpus linguistics into a dead-end street.
>
> (Mukherjee 2007:140f.)

Corpus linguistics is a discipline which is based on distributional data: 'things' – words, morphemes, semantic features – occur in corpora or not, they co-occur with other things or they do not, their frequencies of occurrence or co-occurrence are proportional to those of other 'things' or not, etc. Thus and as mentioned in Section 2.1, all one really obtains from corpora are frequencies of occurrence and co-occurrence. This has two important corollaries, the first of which was also mentioned above: Whatever a corpus linguist is interested in needs to be operationalized and interpreted in terms of frequencies. The second is just as important: If you have frequencies and other distributional data, then you need to employ the tools of the discipline that is concerned with frequencies and distributions, which is statistics. Thankfully, over the past ten or so years, corpus linguistics has evolved considerably in terms of the statistical tools that are being used, but (1) the obvious fact that a corpus linguist needs statistical expertise is still not as widely accepted as it should be – scholars in many other disciplines that deal with much more well-behaved data (i.e., smaller and more balanced data sets) have accepted their statistical needs much longer ago; and (2) while more statistical methods are used these days, they are often applied and/or reported on incorrectly. While I cannot provide a full-fledged introduction to statistics for (corpus) linguists in this book – for that, see Gries (2013), which, as mentioned above, is to some extent a companion volume of this one – in this chapter I will introduce some of the absolute basics of statistical thinking, analysis, and visualization that can aid corpus-linguistic research.

## 4.1 Introduction to Statistical Thinking

Before we begin to actually explore ways corpus data can be analyzed statistically with R, we need to cover a few basic statistical concepts. The first concept is that of a variable. A *variable* is a symbol of a set of values or levels characterizing an entity figuring in an investigation; in the remainder of this chapter, variables will be printed in small caps. For example, if you investigate the lengths of subjects (by counting their lengths in words), then we can say the subject of *The man went to work* gets a value of 2 for the variable LENGTH. There are two ways in which variables must be distinguished: in terms of the role they play in an analysis and in terms of their information value, or level of measurement.

### *4.1.1 Variables and Their Roles in an Analysis*

You will need to distinguish between dependent and independent variables. *Dependent variables* are the variables whose behavior/distribution you investigate or wish to explain; *independent variables* are the variables whose behavior/distribution you think correlates with what happens in the dependent variables. Thus, independent variables are often, but not necessarily, the causes for what's happening with dependent variables.[1]

### *4.1.2 Variables and Their Information Value*

In this brief chapter we will distinguish only two classes of variables. The variable class with the lower information value of the two is that of *categorical variables*. If two entities A and B have different values/levels on a categorical variable, this means that A and B belong to different classes of entities. For example, you can code the two NPs *the book* and *a table* with respect to the variable Definiteness as *definite* and *indefinite* respectively, and the fact that the two NPs receive different levels on this variable means that, with regard to Definiteness, they are different. Other examples for categorical variables (and their possible levels) are

- phonological variables: Stress (*stressed* vs. *unstressed*), StressPosition (*stress on first syllable* vs. *stress on second syllable* vs. *stress elsewhere*);
- syntactic variables: NP-type (*lexical* vs. *pronominal*), Construction (*V NP PP* vs. *V NP NP*);
- semantic variables: Animacy (*animate* vs. *inanimate*), Concreteness (*concrete* vs. *abstract*), LexicalAspect (*activity* vs. *accomplishment* vs. *achievement* vs. *state*).

Given what you read in Section 3.3, you will not be surprised to read that categorical variables of this kind are usually stored as factors in R, and maybe sometimes as character vectors.

The other variable class to be distinguished here is that of *numeric variables*. For example, the syllabic length of an NP is a numeric variable; other examples are pitch frequencies in hertz, word frequencies in a corpus, number of clauses between two successive occurrences of two ditransitives in a corpus file, and the reaction time toward a stimulus in milliseconds. Again, given what you have read in Chapter 3, you will correctly expect that such variables are stored as numeric vectors in R.

This classification of variables in terms of their role in an analysis and their information value will be important to choose the right statistical technique for the evaluation of data because not every statistical technique can be applied to every kind of variable.

### *4.1.3 Hypotheses: Formulation and Operationalization*

One of the most central notions in statistical analysis is that of a hypothesis. The term *hypothesis* is used here in a somewhat stricter sense than in the everyday sense of "assumption". Following Bortz (2005: 1–14), in what follows, hypothesis refers to

- a statement characterizing more than a singular state of affairs;
- a statement that has at least implicitly the structure of a conditional sentence (with *if . . . , then . . .* or *the more/less . . . , the more/less . . .* );
- the hypothesis must be potentially falsifiable.

While the first criterion needs no additional explanation, the second criterion requires a little elaboration. The logic behind this criterion is that, while hypotheses typically have one of the above syntactic forms, they need not. All that is required is that the statement

can be transformed into a conditional sentence. For example, the statement *On average, English subjects are shorter than English objects* does not explicitly instantiate a conditional sentence, but it can be transformed into one: *If a constituent is an English subject, it is on average shorter than a constituent that is an English object*. Or, for example, the statement *More frequent words have more senses than less frequent words* does not explicitly instantiate a conditional sentence, but it can be transformed into one: *The more frequent a word is, the more senses it has*.

The third criterion should also briefly be commented on. It means that there must be conceivable states of affairs that show the hypothesis to be false. Thus, *Drinking alcohol may influence the reaction times of subjects in an experiment* is not a hypothesis in this sense because *may influence* basically means "may influence or may not", so every result of an experiment – an influence or a lack of it – is compatible with the hypothesis. Note that this criterion does not mean that the falsifying state of affairs is ever observed – it really only means that one can imagine it so that, if it ever were to happen, one could recognize it (by virtue of the similarity to the imagined outcome).

Above, hypotheses were characterized as invoking the notion of conditionality. This characterization reiterates the first central distinction of variables, that between independent and dependent variables. Simplistically speaking, independent variables are variables that are mentioned in the *if*-clause or the first *the more/less* clause, while dependent variables are variables mentioned in the *then*-clause or the second *the more/less* clause. It is important to understand, however, that there are two parameters according to which hypotheses are classified.

The first parameter refers to the contrast between the so-called *alternative hypothesis* (abbreviated as $H_1$) and the so-called *null hypothesis* (abbreviated as $H_0$). Usually, the former is a hypothesis that,

- if you look at just one dependent variable, states that the values/levels of the dependent variable do not follow a random distribution of the investigated entities (or some other expected distribution such as the normal distribution or a uniform distribution);
- if you look at dependent and independent variables, states that the values/levels of the dependent variable vary non-trivially as a function of the values/levels of the independent variable(s).

For example, *On average, English subjects are shorter than English objects* would be an alternative hypothesis because it states that some non-trivial part of the variation you find in the lengths of XPs is due to the fact that the XPs instantiate different grammatical relations, namely subjects and direct objects. $H_0$, by contrast, is the logical counterpart of $H_1$ that,

- if you look at just one dependent variable, states that the values/levels of the dependent variable are randomly distributed or do not vary quasi-systematically from some specified distribution (such as the normal or the binomial distribution);
- if you look at dependent and independent variables, states that the values/levels of the dependent variable do not vary non-trivially as a function of the values/levels of the independent variable(s).

You can often form a $H_0$ by inserting *not* into $H_1$ (that is, *On average, English subjects are* not *shorter than English objects*), but typically $H_0$ states there is no effect (that is, *On average, English subjects and English objects do not differ in their lengths* or *On average, English subjects and English objects are equally long*).

The second parameter is concerned with the language of the hypothesis, so to speak. Both the $H_1$ and $H_0$ come in two forms. The first of these is the one you have already seen, a text form in natural human language. The other form is its 'translation' into a statistical

form in mathematical language, which brings me to the issue of *operationalization*, the process of deciding how the variables in your text hypotheses shall be investigated. In other words, you answer two interrelated questions. First, "What will I perceive when we perform our study and observe the values/levels of the variables involved?" Second, "Which mathematical concept will I use – counts/frequencies, averages, dispersions, or correlations – when I couch my hypothesis into numeric terms?" This is necessary because, often, the formulation of the hypotheses in text form leaves open what exactly will be counted, measured, etc. and how. In the above case, for example, it is not yet clear how you will know whether a particular subject is longer than a particular direct object:

(1) The younger bachelors ate the nice little parrot.

With regard to the question of subject and object lengths, this sentence can support either $H_1$ or $H_0$, depending on how you operationalize the variable LENGTH. If you operationalize LENGTH as "number of morphemes", the subject gets a value of 5 (*The*, *young*, comparative *-er*, *bachelor*, plural *s*) and the direct object gets a value of 4 (*the*, *nice*, *little*, *parrot*). On the other hand, if you operationalize LENGTH as number of words, the subject and the object get values of 3 and 4 respectively. Finally, if you operationalize LENGTH as "number of letters (without spaces)", both subject and object get a value of 19.

With regard to the latter question, you also must decide on a mathematical concept. For example, if you investigate, say, 99 sentences and choose to operationalize lengths as numbers of morphemes, your hypotheses could involve averages:

$H_0$: The average length of the subjects in morphemes is the same as the average length of the direct objects in morphemes; mean length of subjects = mean length of direct objects.

$H_1$: The average length of the subjects in morphemes is different from the average length of the direct objects in morphemes; mean length of subjects ≠ mean length of direct objects.

or frequencies:

$H_0$: The number of cases where the subject is longer (in morphemes) than the direct object is the same as the number of cases where the subject is not longer (in morphemes) than the direct object.

$H_0$: The number of cases where the subject is longer (in morphemes) than the direct object is different from the number of cases where the subject is not longer (in morphemes) than the direct object.

Thus, you must exercise care in operationalizing your variables: Your operationalization of variables determines the validity of the investigation, i.e., whether you measure/investigate what you intend to measure/investigate.

You should now do "Exercise box 4.1: How would you operationalize these variables? Why?" and "Exercise box 4.2: Formulate alternative and null hypotheses in text form and in statistical form."

*Table 4.1* Fictitious data set for a study on constituent lengths

| | GramRelation: *subject* | GramRelation: *object* | *Totals* |
|---|---|---|---|
| ClauseType: *main* | 30 | 30 | 60 |
| ClauseType: *subordinate* | 30 | 30 | 60 |
| Totals | 60 | 60 | 120 |

While the discussion so far has focused on the situation in which we have maximally two variables, a dependent and an independent one, of course life is usually not that simple – there is virtually always more than one determinant for some phenomenon. While we will not deal with statistical techniques to deal with several independent variables here (but see Gries 2013: ch. 5), it is important for you to at least understand how such phenomena would be approached conceptually. We have been talking about the alternative hypothesis that, on average, English subjects are shorter than English objects, which involves one dependent variable (Length) and one independent variable (GramRelation: *subject* vs. *object*). For the sake of the argument, let us imagine you now suspect that the lengths of some constituents do not only vary as a function of their being subjects and objects, but also in terms of whether they occur in main clauses or subordinate clauses. Technically speaking, you are introducing a second independent variable, ClauseType, with two levels, *main clause* and *subordinate clause*. Let us also assume you do a pilot study – without having a hypothesis yet – in which you look at 120 subjects and objects (from a corpus) as shown in Table 4.1.

You then count for each of the 120 cases the length of the subject or object in syllables and compute the mean lengths for all four conditions, subjects in main clauses, objects in main clauses, subjects in subordinate clauses, and objects in subordinate clauses. Let us assume these are the main effects you find, where a *main effect* is the effect of one variable in isolation:

- Constituents that are subjects are shorter than constituents that are objects.
- Constituents in main clauses are shorter than in subordinate clauses.

There are now two possibilities for what these results can look like (or three, depending on how you want to look at it). On the one hand, the two independent variables may work together *additively*. That means the simultaneous combination of levels of independent variables has the effect that we would expect on the basis of their individual effects. In this case, you would therefore expect that

- the shortest constituents are subjects in main clauses while the longest constituents are objects in subordinate clauses;
- the length difference between main and subordinate clause subjects is the same as the length difference between main and subordinate clause objects; and
- the length difference between main clause subjects and objects is the same as the length difference between subordinate clause subjects and objects.

This perfectly additive result is represented in a so-called interaction plot in Figure 4.1.

However, the independent variables may not work together additively, but *interact*. Two variables are said to interact if their joint influence on the dependent variable is not additive, i.e., if their joint effect cannot be inferred on the basis of their individual effects alone. In our example: If direct objects are longer than subjects and if elements in

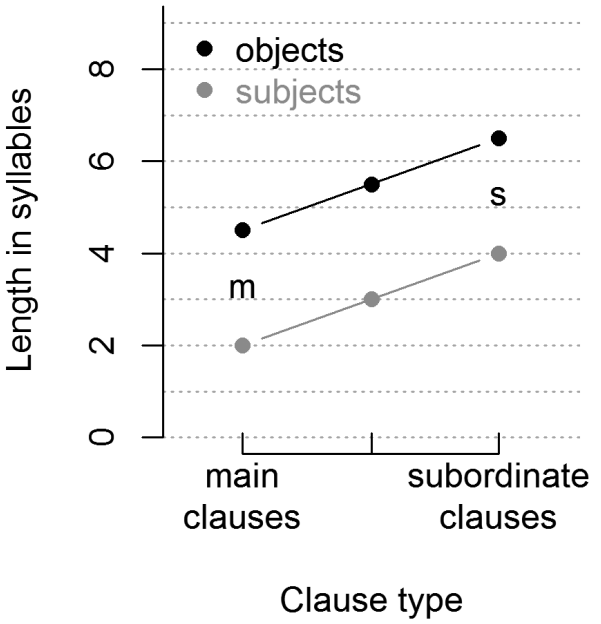


*Figure 4.1* Interaction plot for GramRelation × ClauseType 1.

subordinate clauses are longer than elements in main clauses, but you also find that objects in subordinate clauses are in fact very short, then this would be called an interaction of GramRelation and ClauseType. One such scenario is represented in Figure 4.2.

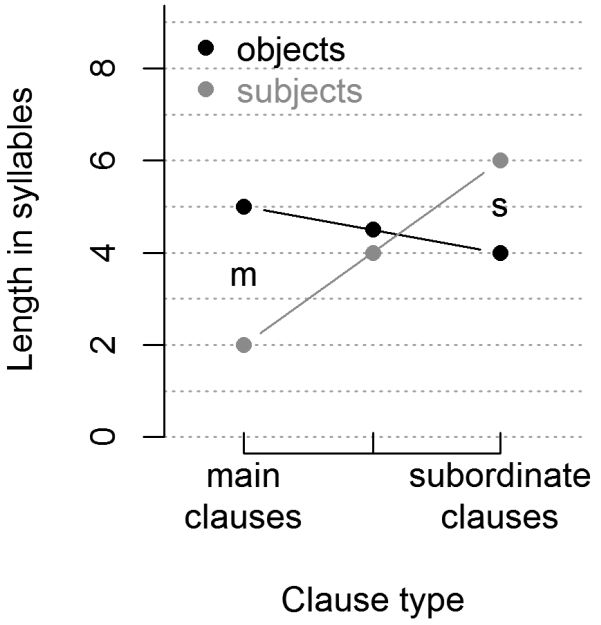


*Figure 4.2* Interaction plot for GramRelation × ClauseType 2.

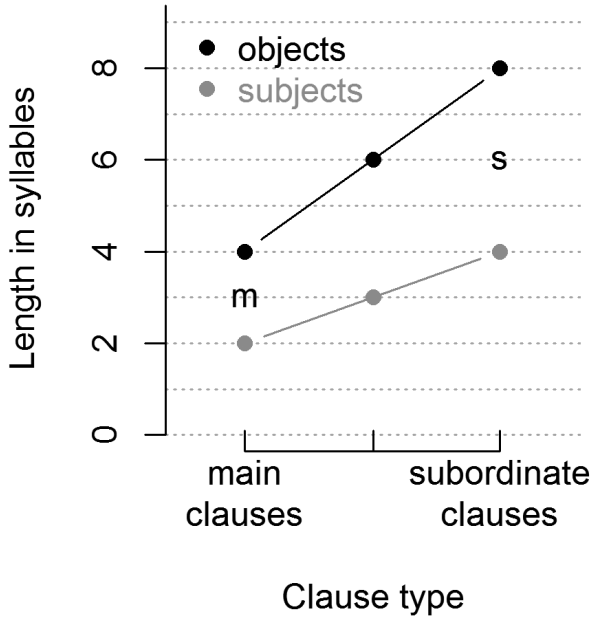


*Figure 4.3* Interaction plot for GramRelation × ClauseType 3.

As you can see, the mean length of subjects is still smaller than that of objects. Also, the mean length of main clause constituents is still smaller than that of the subordinate clause constituents. However, the combinations of these levels does not result in the same predictable effect as it did in Figure 4.1. In other words, while objects are overall longer than subjects – the main effect – not *all* objects are: The objects in subordinate clauses are not only shorter than objects in main clauses, but in fact also shorter than subjects in subordinate clauses. Thus, if there is an interaction of this kind – which is often easy to recognize because of the crossing lines in such an interaction plot, then you should really not talk that much about the main effects of the variables participating in a significant interaction because the interaction indicates that the main effects are only true in parts of the data.

There is yet another kind of interaction, which is represented in Figure 4.3.

This may strike you as strange. Again, the main effects are the same: Subjects are shorter than objects and main clause constituents are shorter than subordinate clause constituents. But as opposed to Figure 4.2, even the prediction that objects in subordinate clauses are the longest elements and subjects in main clauses are the shortest elements is borne out. So why is this still also an interaction?

**THINK**

**BREAK**

It is still an interaction because while the lines in the interaction plot do not intersect, the slope of the black line is much steeper than that of the gray line. Put differently, while the difference between the means of subjects and objects in main clauses is only two

syllables, it is four syllables in subordinate clauses, and because of this unexpected element, this is still an interaction.

Interactions are a very important concept, and whenever you consider or investigate designs with multiple independent variables, you must be careful about how to formulate your hypotheses – sometimes you may be more interested in an interaction than a main effect – and how to evaluate and interpret your data.

One final aspect is noteworthy here. Obviously, when you formulate hypotheses, you have to make many decisions: which variables to include, which variable levels to include, which kinds of relations between variables to consider (additive vs. interaction), etc. One important guideline in formulating hypotheses you should always adhere to is called Occam's razor. Occam's razor, named after William of Ockham (1285–1349), is the name for the following Latin principle: *entia non sunt multiplicanda praeter necessitatem*, which means "entities should not be multiplied beyond necessity" and, generally boils down to "try to formulate the simplest explanation or model possible". I use *model* here in the way defined by Johnson (2008: 106), referring to a "proposed mathematical description of the data with no assumptions about the possible mechanisms that cause the data to be the way they are". More specifically, this principle – also referred to as the *principle of parsimony* or succinctness – boils down to the following guidelines (that statistically all amount to not making the numbers of coefficients to be estimated from the data any larger than is warranted):

- Don't invoke more variables than needed: If you can explain the same amount of variability of a dependent variable with two models with different numbers of independent variables, choose the model with the smaller number of independent variables.
- Do not invoke more variable levels than necessary: If you can explain the same amount of variability of the dependent variable with two models that have different numbers of levels of independent variables, choose the model with the smaller number of levels of independent variables. For example, if you can explain 60 percent of the variation in the data with a variable that distinguishes two levels of animacy (*animate* vs. *inanimate*), then prefer this variable over one that distinguishes three levels (e.g., *human* vs. *animate* vs. *inanimate*) but also only explains 60 percent of the variation in the data.
- Choose additive relationships over relationships involving interactions: If you can explain 60 percent of the variation of the dependent variable with a model without interactions, then prefer this model over one that involves interactions and also only explains 60 percent of the variation.

One motivation for this principle is of course simplicity: simpler models are easier to explain and test. Also, this approach allows you to reject the unprincipled inclusion of ever more variables and variable levels if these do not explain a sufficiently substantial share of the data. Thus, you should always bear this principle in mind when you formulate your own hypotheses.

### *4.1.4 Data Analysis*

When you have formulated your hypotheses, and only then, you begin with the analysis of the data. (In other words, it is problematic to collect data, pour over them for hours in search of any pattern that might be interesting, see some pattern, and then declare you have a hypothesis, namely, *tadaah!*, the pattern and then proceed with statistical testing.) For example, your analysis might begin with you writing a script to retrieve corpus data.

The best way of handling data is to store your data in a data frame that you either edit in R or, more conveniently, in a spreadsheet software such as LibreOffice Calc. There are a few crucial aspects to your handling of the data, which I cannot emphasize enough. First, every row represents *one and only one* analyzed case or observation of the dependent variable. Second, every column but the first represents either the data in question (e.g., (parts of) a concordance line to be investigated) or *one and only one* variable with respect to which the data are coded. Third, the first column just gets a counter from 1 to *n* so that you can always restore the data frame to one particular (the original?) order, and the first row should contain the names of all columns (i.e., variables). Fourth, missing data points are entered as NA (not as empty cells). In corpus studies, it is often useful to also have columns for variables that you might call "source of the data"; these would contain information regarding the corpus file and the line of the file where the match was obtained etc.

Let us assume you investigated the alternative hypothesis *Subjects are longer than objects* (operationalizing length as "length in words"). First, a think break: What is the independent variable, and what is the dependent variable?

**THINK**

**BREAK**

The independent variable is the categorical variable GramRelation, which has the levels *subject* and *object*, and the dependent variable is the numeric variable Length. If you now formulated all four hypotheses – $H_0$ and $H_1$ in text and statistical forms – and decided to analyze the (ridiculously small) sample in (2),

(2) a The younger bachelors ate the nice little parrot.
b He was locking the door.
c The quick brown fox hit the lazy dog.

then your data frame should *not* look as shown in Table 4.2, an arrangement students often use.

The data frame in Table 4.2 violates all of the above rules. First, every row contains two data points, not just one; for instance, row 1 contains two measurements of the dependent variable, 3 and 4. Second, the data frame in Table 4.2, leaving aside the first column, which was only included for expository purposes, does not have a column for every variable. Rather, it has two columns, each of which represents one *level* of the independent variable GramRelation. Before you look below, think about how you would have to reorganize Table 4.2 so that it conforms to the above rules.

*Table 4.2* A bad data frame

| Sentence | Subject | Object |
|---|---|---|
| The younger bachelors ate the nice little parrot. | 3 | 4 |
| He was locking the door. | 1 | 2 |
| The quick brown fox hit the lazy dog. | 4 | 3 |

*Table 4.3* A better data frame

| Case | Sentence | Relation | Length |
|---|---|---|---|
| 1 | The younger bachelors ate the nice little parrot. | Subject | 3 |
| 2 | The younger bachelors ate the nice little parrot. | Object | 4 |
| 3 | He was locking the door. | Subject | 1 |
| 4 | He was locking the door. | Oobject | 2 |
| 5 | The quick brown fox hit the lazy dog. | Subject | 4 |
| 6 | The quick brown fox hit the lazy dog. | Object | 3 |

**THINK**

**BREAK**

Table 4.3 is how your data frame should look. Now every observation has one and only one row, and every variable – independent and dependent – has its own column.

An even more precise variant may have an additional column listing only the subject or object that is coded in each row, i.e., listing *The younger bachelors* in the first row. As you can see, every data point – every subject and every object – gets its own row and is in turn described in terms of its variable levels in the two rightmost columns. This is how you should virtually always store your data. Ideally, you save data from text processing with R into a format that already looks like that of Table 4.3, then you may add additional annotation either in/with R or in some spreadsheet software, and then you save the data (1) in the native format of that software (to preserve colors or other formatting that would not be preserved in a text file) and (2) as a .txt file or, better, a tab-separated .csv file.

### *4.1.5 Hypothesis (and Significance) Testing*

Once you have created a data frame such as Table 4.3 containing all your data, you must evaluate the data. As a result of that evaluation, you obtain frequencies/counts, averages, dispersions, or correlations. The most essential part of the statistical approach of hypothesis testing is that, contrary to what you might expect, you do *not* try to prove that your $H_1$ is correct – you try to show that $H_0$ is most likely (!) not true, and since $H_0$ is the logical counterpart of your $H_1$, this in turn lends credence to $H_1$. In other words, in most cases you wish to be able to show that $H_0$ can *not* account for the data well enough so that you can adopt $H_1$. But what does "well enough" mean? The conventional interpretation of "well enough" is the following: If the probability to get your results when $H_0$ is true is less than 0.05 (i.e., 5 percent), then it seems as if $H_0$ doesn't really account very well for the results that you have; thus, you reject $H_0$ as too bad an account of your results and instead assume $H_1$.

This may seem a little confusing because there are suddenly two probabilities: one that says how likely you are to get your result when $H_0$ is true, another one which is typically set to 0.05. This latter probability $p$ is the probability not to be exceeded to still accept $H_1$. It is called *significance level* and is a threshold value *defined before* the analysis. The former probability $p$ to get the obtained result when $H_0$ is true is the probability to err when accepting $H_1$. It is called the *probability of error* or *the* p*-value* – this $p$ is computed on the basis of your data, i.e., *computed after* the analysis. You may now wonder how this probability is computed, but this is a question we will not be concerned with in this book – see Gries (2013: 26–45) for detailed discussion. Suffice it here to say that the probabilities of error are typically computed in one of two ways: either *parametrically*, i.e., on the basis of

approximations, which work very well if the data meet certain distributional criteria, or *non-parametrically*, e.g., on the basis of counting frequencies of different outcomes out of all possible outcomes. In the remainder of this chapter, we will use R functions to make either of those two computations.

Now that the groundwork has been laid, we will look at a few statistical techniques that allow you to compute *p*-values to determine whether you are allowed to reject the $H_0$ and, therefore, accept your $H_1$. For each of the tests, we will also look at all relevant assumptions. Unfortunately, considerations of space do not allow me to discuss more than some of the most elementary monofactorial statistical techniques and skip the treatment of all the techniques that would also be useful, in particular the multifactorial statistics that one ultimately always requires. Since the graphical representation and exploration of distributional data often facilitates the understanding of the statistical results, I will also briefly show how simple graphs are produced. However, again considerations of space do not allow for a detailed exposition here in the book, so do study the code files, which provide a lot of information, review the documentation of the graphics functions I mention, and refer to the documentation for `par`, Johnson (2008) and Gries (2013), and, for more advanced readers, Baayen (2008) and Murrell (2011).

## 4.2 Categorical Dependent Variables

One of the most frequent corpus-linguistic scenarios involves categorical dependent variables in the sense that one records the frequencies of several mutually exclusive outcomes or categories. We will only distinguish two cases, one in which one just records frequencies of a dependent variable as such without any additional information (such as which independent variable is observed at the same time) and one in which you also include a categorical independent variable.

### *4.2.1 No Independent Variables*

Let us look at a syntactic example, namely the word order alternation of English verb-particle constructions as exemplified in (3):

(3) a. He brought back the book. Verb Particle DirectObject
b. He brought the book back. Verb DirectObject Particle

Among other things, one might be interested in finding out whether the two semantically so similar constructions are used equally frequently. To that end, one could decide to look at corpus data for the two constructions. The first step of the analysis consists of formulating the hypotheses to be tested. In this case, this is fairly simple: Since $H_0$ usually states that data are not systematically distributed, the most unsystematic distribution would be a random one, and if sentences are randomly distributed across two construction categories, then both constructions should be equally frequent, just like tossing a coin will in the long run yield nearly identical frequencies of heads and tails. Thus:

$H_0$: The frequencies of the two constructions (*V Part DO* vs. *V DO Part*) in the population are not different; $n_{\text{V Part DO}} = n_{\text{V DO Part}}$, and variation in the sample is just random noise.

$H_1$: The frequencies of the two constructions (*V Part DO* vs. *V DO Part*) in the population are different; $n_{\text{V Part DO}} \neq n_{\text{V DO Part}}$, and variation in the sample is not just random noise.

*Table 4.4* Observed distribution of verb-particle constructions in Gries (2003a)

| *Verb Particle DirectObject* | *Verb DirectObject Particle* |
|---|---|
| 194 | 209 |

Gries (2003) used a small data sample from the BNC Edition 1 and counted how often each of these two constructions occurred. If the data frame is set up as in Table 4.3, the simplest descriptive approach would be to use `table` on the column with the dependent variable so as to arrive at something like Table 4.4.

For this simple example, we will not load a data frame but immediately tell R what the data look like:

```
>·Gries.2003<-c(194,·209)¶
>·names(Gries.2003)<-c("V-P-DO",·"V-DO-P")¶
```

As a first step, we look at the distribution of the data. One simple way of representing this distribution graphically would be to use a bar plot:

```
>·barplot(Gries.2003)¶
```

The question that now arises is of course whether this is a result that one may expect by chance or whether the result is unlikely to have arisen by chance and, therefore, probably reflects a regularity to be uncovered. The actual computation of the test is in this case very simple, but for further discussion below it is useful to briefly comment on the underlying logic. The test that is used to investigate whether an observed frequency distribution deviates from what might be expected on the basis of chance is called the *chi-squared test for goodness of fit* (because it tests how good the fit of the observed data is to some expected distribution). In this case, $H_0$ states the frequencies are the same so, since we have two categories, the expected frequency of each construction is the overall number of sentences divided by two (Table 4.5).

Like with most statistical tests, however, we must first determine whether the test we want to use can in fact be used. The chi-squared test should only be applied if:

- all observations are independent of each other (i.e., the value of one data point does not influence that of another); and
- all expected frequencies are larger than or equal to 5.[2]

The latter condition is obviously met, and we assume for now that the data points are independent such that the constructional choice in any one corpus example is

*Table 4.5* Expected distribution of verb-particle constructions in Gries (2003a)

| *Verb particle DirectObject* | *Verb DirectObject particle* |
|---|---|
| 201.5 | 201.5 |

independent of other constructional choices in the sample – note that this assumption is often not correct (because, often, one speaker contributes multiple data points, which means all the data points he contributes could be affected by that speaker's idiosyncratic behavior) and would therefore need to be carefully defended in each case. The actual computation of the chi-squared value is summarized in the equation in (4):

$$(4)\quad \chi^2 = \sum_{i=1}^{n} \frac{(\text{observed} - \text{expected})^2}{\text{expected}} = \frac{(194 - 201.5)^2}{201.5} + \frac{(209 - 201.5)^2}{201.5} \cong 0.558$$

You take the observed value of each cell in the table, subtract from it the expected value for that cell, square the obtained difference, divide it by the expected value for that cell again, and sum up all values (see the code file for a quick manual computation of this value). Each of these two summands is sometimes referred to as a *contribution to chi-squared*.

However, we just do the whole test in R, which is extremely simple. Since R already knows what the observed data look like (from the vector `Gries.2003`), we can immediately use the function `chisq.test` to compute the chi-squared value and the *p*-value at the same time by providing `chisq.test` with three arguments: a vector with the observed data, a vector with the probabilities resulting from $H_0$ (i.e., two times 0.5, because according to $H_0$ we have two equally likely constructions), and `correct=TRUE` or `correct=FALSE`: If the size of the data set *n* is small ($15 \leq n \leq 60$), it is sometimes recommended to perform a so-called continuity correction; by calling `correct=TRUE` you can perform this correction. We immediately assign the result of this test to an object `G2003.test` and also check out its structure:

```
>·(G2003.test<-chisq.test(Gries.2003,·p=c(0.5,·0.5),·
  correct=FALSE))¶
  Chi-squared·test·for·given·probabilities
data:··Gries.2003
X-squared·=·0.55831,·df·=·1,·p-value·=·0.4549

>·str(G2003.test)¶
List·of·9
·$·statistic:·Named·num·0.558
·..·-·attr(*,·"names")=·chr·"X-squared"
·$·parameter:·Named·num·1
·..·-·attr(*,·"names")=·chr·"df"
·$·p.value··:·num·0.455
·$·method···:·chr·"Chi-squared·test·for·given·probabilities"
·$·data.name:·chr·"Gries.2003"
·$·observed·:·Named·num·[1:2]·194·209
·..·-·attr(*,·"names")=·chr·[1:2]·"V-P-DO"·"V-DO-P"
·$·expected·:·Named·num·[1:2]·202·202
·..·-·attr(*,·"names")=·chr·[1:2]·"V-P-DO"·"V-DO-P"
·$·residuals:·Named·num·[1:2]·-0.528·0.528
·..·-·attr(*,·"names")=·chr·[1:2]·"V-P-DO"·"V-DO-P"
·$·stdres···:·Named·num·[1:2]·-0.747·0.747
·..·-·attr(*,·"names")=·chr·[1:2]·"V-P-DO"·"V-DO-P"
·-·attr(*,·"class")=·chr·"htest"
```

The result is unambiguous: The *p*-value is much larger than the usual threshold value of 0.05 so we conclude that the frequencies of the two constructions in Gries's sample do not differ from a random distribution. Another way of summarizing this result would be to say that Gries's data can be assumed to come from a population in which both constructions are equally frequent.

You may now wonder what the *df*-value means. The abbreviation "df" stands for "degrees of freedom" and has to do with the number of values that were entered into the analysis (two in this case) and the number of statistical parameters estimated from the data (one in this case); it is customary to provide the *df*-value when you report all summary statistics, as I will exemplify below for each test to be discussed. For reasons of space, I cannot discuss the notion of *df* here in any detail but refer you to full-fledged introductions to statistics instead (e.g., Crawley 2012).

As you can see from the structure output, `chisq.test` also computes the expected frequencies as well as a variety of other results, which are all stored in a list. If we are interested in the expected frequencies, we just need to call the part of the result/list we are interested in:

```
>·G2003.test$expected¶
[1]·201.5·201.5
```

The usual statistical lingo to summarize this result would be something like this: "The verb-particle construction where the particle directly follows the verb occurs 194 times although it was expected 201.5 (i.e., 202) times. On the other hand, the construction with the particle following the direct object was produced 209 times although it was expected 201.5 (i.e., 202) times. According to a chi-squared test for goodness of fit, this difference, however, is statistically insignificant ($\chi^2 = 0.56$; $df = 1$; $p = 0.455$): We must assume the two constructions are equally frequent in the population for which the sample is representative."

You should now do "Exercise box 4.3: Unidimensional frequency distributions."

**Recommendations for further study/exploration**

- On another way to test whether an observed frequency differs from an expected percentage: `?prop.test¶`.

### *4.2.2 One Independent Categorical Variable*

The probably more frequent research scenario with dependent categorical variables, however, is that one records not just the frequency of some dependent variable but also that of an independent categorical variable that one might suspect is correlated with the dependent one. Fortunately, the required method is very similar to that of the previous section: It is called the *chi-squared test for independence* and has the same two requirements

as the chi-squared test in the previous section. In order to explore how this test works when an additional variable is added, let us revisit the example of Gries (2003) from above. The above discussion actually provided only a part of the data. In fact, the above characterization left out one crucial factor, namely one of many independent variables whose influence on the choice of construction Gries wanted to investigate. This independent variable to be looked at here was CONCRETENESS, i.e., whether the referent of the direct object was abstract (such as, e.g., *peace*) or concrete (such as, e.g., *the book*). Again, we first formulate the hypotheses. Since we have seen in the previous section how the chi-squared statistic is computed – namely with the absolute differences between observed and expected values in the numerator of (4), we immediately add the statistical hypotheses as well:

$H_0$: The frequencies of the two constructions (*V Part DO* vs. *V DO Part*, the dependent variable) do not vary depending on whether the referent of the direct object is abstract or concrete (the independent variable); chi-squared = 0.

$H_1$: The frequencies of the two constructions (*V Part DO* vs. *V DO Part*, the dependent variable) vary depending on whether the referent of the direct object is abstract or concrete (the independent variable); chi-squared > 0.

We first read the data from <_qclwr2/_inputfiles/stat_vpc.csv> into R:

```
>·Gries.2003<-read.table(file.choose(),·header=TRUE,·sep="\t",·
  comment.char="")¶
>·str(Gries.2003);·attach(Gries.2003)¶
'data.frame':···403·obs.·of··2·variables:
·$·CONSTRUCTION:·Factor·w/·2·levels·"V_DO_PART","V_PART_DO":·
  2·1·1·1·2·1·...
·$·CONCRETENESS:·Factor·w/·2·levels·"abstract","concrete":·
  1·1·2·2·2·2·1·...
```

Since we now know the column names etc., we tabulate (as usual, the first-named variable goes into the rows) and use the generic `plot` command to get a so-called mosaic plot of a table of our two variables (we use `t(table(...))` to transpose the table so that the plot has the same row–column arrangement as the table):

```
>·table(CONSTRUCTION,·CONCRETENESS)¶
···········CONCRETENESS
CONSTRUCTION·abstract·concrete
···V_DO_PART·····64·····145
···V_PART_DO·····125·····69
>·mosaicplot(t(table(CONSTRUCTION,·CONCRETENESS)),·
  main="The·relation·between·CONCRETENESS·and·
  CONSTRUCTION")¶
```

In this kind of plot, the frequencies of the two levels of CONCRETENESS are reflected in the widths of the bars, while the frequencies of the two constructions within each level of

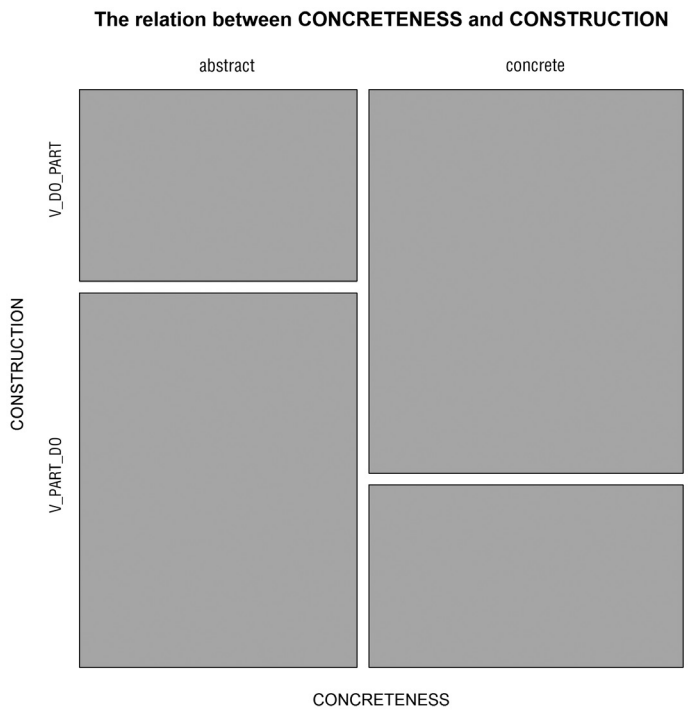


*Figure 4.4* Mosaic plot of the distribution of verb-particle constructions in Gries (2003).

CONCRETENESS are indicated by how the two bars are split. It immediately emerges that the two constructional choices are characterized by very different proportions of concrete and abstract DO referents: When CONCRETENESS is *concrete*, the construction *V DO Part* is more frequent, when CONCRETENESS is *abstract*, *V Part DO* is.

Does this distribution differ significantly from one expected by chance? The function to test whether there is a correlation between two categorical variables or not is again `chisq.test`, and the most important arguments it takes are:

- a two-dimensional table for which you want to compute a chi-squared test;
- `correct=TRUE` or `correct=FALSE`; see above.

We have already loaded the data so we can immediately execute the function:

```
>·(G2003.test<-chisq.test(table(CONSTRUCTION,·CONCRETENESS),·
  correct=FALSE))¶
·····Pearson's·Chi-squared·test
data:··table(CONSTRUCTION,·CONCRETENESS)
X-squared·=·46.184,·df·=·1,·p-value·=·1.076e-11
```

For the sake of completeness, we also compute the expected frequencies by calling up a part of the output that is normally not provided (by adding `$expected` to the name of the list):

```
>·G2003.test$expected¶
···········CONCRETENESS
CONSTRUCTION·abstract·concrete
···V_DO_Part·98.01737·110.9826
···V_Part_DO·90.98263·103.0174
```

Note that you can compute the expected frequency for any of the four cells of the table of the observed frequencies by multiplying the row total of the row of a cell by the column total of the column of that cell and dividing that by the overall total of the table, as is here shown for the top left cell:

```
>·209*189/403¶
[1]·98.01737
```

The results show that the variable Concreteness is highly significantly correlated with the choice of construction. However, there are two things we still don't really know. One is how strong the effect is: It is important to note that one cannot use the chi-squared value as a measure of effect size, i.e., as an indication of how strong the correlation between the two investigated variables is. This is due to the fact that the chi-squared value is dependent on the effect size, but also on the sample size. We can test this in R very easily:

```
>·chisq.test(table(CONSTRUCTION,·CONCRETENESS)*10,·correct=FALSE)¶
·····Pearson's·Chi-squared·test
data:··table(CONSTRUCTION,·CONCRETENESS)·*·10
X-squared·=·461.84,·df·=·1,·p-value·<·2.2e-16
```

As is obvious, when the sample is increased by one order of magnitude, so is the chi-squared value. This is of course a disadvantage: While the sample size of this increased table is ten times as large, the relations of the values in the table have of course not changed. In order to obtain a measure of effect size that is not influenced by the sample size, one can transform the chi-squared value into a measure of correlation. This is the formula for the measure $\phi$ (read: phi, for $k \times 2$ / $m \times 2$ tables, where $k$ and $m$ are the numbers of rows and columns respectively) or Cramer's $V$ (for $k \times m$ tables with $k, m > 2$):

(5) $\phi$ or Cramer's $V = \sqrt{\dfrac{\chi^2}{n^* \left(\min[k, m] - 1\right)}}$

The theoretically extreme values of this correlation coefficient are 0 (no correlation) and 1 (perfect correlation). With R, this computation can be done in one easy step, but for expository reasons we break it down into smaller steps:

```
>·numerator<-G2003.test$statistic¶
>·denominator<-sum(table(CONSTRUCTION,·CONCRETENESS))*
  (min(dim(table(CONSTRUCTION,·CONCRETENESS)))-1)¶
>·fraction<-numerator/denominator¶
>·(phi<-sqrt(fraction))¶
X-squared
0.3385275
```

Thus, the correlation is not particularly strong but highly significant. But where does it come from and how can the results be interpreted? In this simple case, the mosaic plot above already gave it all away, but in tables with more rows and columns, interpreting the result from such a plot may be more difficult. Thus, the most straightforward way to answer these questions involves inspecting (1) the so-called Pearson residuals and/or (2) an association plot. The Pearson residuals indicate the degree to which observed and expected frequencies differ: The more they deviate from 0, the more the observed frequencies deviate from the expected ones. Each residual of a cell is $^{obs\text{-}exp}/_{\mathrm{sqrt}(exp)}$, so if you square them you get the above-mentioned contributions to chi-squared, whose sum in turn corresponds to the overall chi-squared value. They are generated just like the expected frequencies, by calling up the result of the chi-squared test again, this time requesting another part of the output normally not provided:

```
>·G2003.test$residuals¶
···········CONCRETENESS
CONSTRUCTION··abstract··concrete
···V_DO_Part·-3.435969··3.229039
···V_Part_DO··3.566330·-3.351548
```

This table is interpreted as follows: Positive values mark observed frequencies which are larger than expected and negative values mark observed frequencies which are smaller than expected, and you already know that the more the values deviate from 0, the stronger the effect. Thus, the strongest effect in the data is the strong preference of abstract objects to occur in *V DO Part*; the second strongest effect is the dispreference of abstract objects to occur in *V Part DO* etc. (In this case, all residuals are similarly high, but in tables where this is not so, one could distinguish the cells that matter from those that do not.)

A visual approach to the same issue involves a so-called association plot (Figure 4.5). The function `assocplot` requires only one argument, namely a two-dimensional table, and again we usually want to transpose it so its dimensions line up with those of the input table:

```
>·assocplot(t(table(CONSTRUCTION,·CONCRETENESS)),·col=c
  ("black",·"grey"))¶
```

In this representation, black and gray boxes represent table cells whose observed frequencies are greater and smaller than the expected ones respectively (i.e., what corresponds to positive and negative Pearson residuals), and the area of the box is

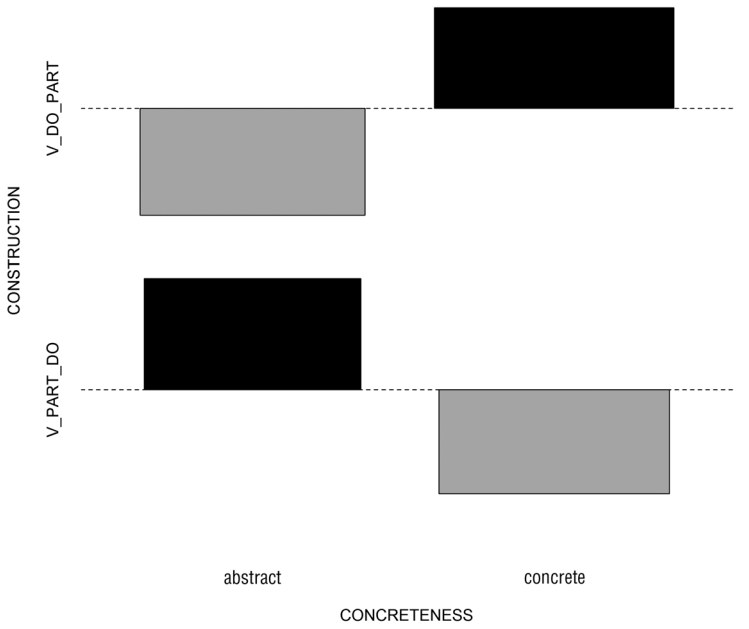


*Figure 4.5* Association plot of the distribution of verb-particle constructions in Gries (2003).

proportional to the difference in observed and expected frequencies (in a way you can read up on at `?assocplot¶`).

In sum, the data and their evaluations would be summarized as follows: "According to a chi-squared test for independence, there is a statistically highly significant albeit moderate correlation between the choice of a verb-particle construction and the abstractness/concreteness ($\chi^2 = 46.18$; $df = 1$; $p < 0.001$; $\phi = 0.34$). The significant result is due to the fact that the construction where the particle follows the verb directly is preferred with abstract objects while the construction where the particle follows the direct object is preferred with concrete objects."

Let me briefly mention one important area of application of the chi-squared test and other related statistics: measures of collocational strength. Collocational statistics quantify the strength of association or repulsion between a node word and its collocates. Node words tend to attract some words – such that these words occur close to the node word with *greater* than chance probability – while they repel others – such that these words occur close to the node word with *less* than chance probability – while they occur at chance-level frequency with yet others. Let us look at one example from the case study chapter below, the collocation of *alphabetical order*. In the BNC, there are 6,024,359 numbered sentences (occurrences of "<s·n") and 225 and 28,662 such sentences that

*Table 4.6* Observed distribution of *alphabetical* and *order* in the BNC

| | *Order* | *Other words* | *Totals* |
|---|---|---|---|
| *Alphabetical* | **96** | 129 | **225** |
| Other words | 28,566 | 5,995,568 | 6,024,134 |
| Totals | **28,662** | 5,995,697 | **6,024,359** |

contain *alphabetical* tagged as an adjective and *order* tagged as a singular noun, and there are 96 sentences in which both of these words occur. This distribution is summarized in Table 4.6.

Of course, you can evaluate this distribution with a chi-squared test; you use the `matrix` function to create a table like Table 4.6:

```
> example <- matrix(c(96, 28566, 129, 5995568), ncol=2,
  dimnames=
  list(ALPHABETICAL=c("yes", "no"), ORDER=c("yes", "no")))
> example
           ORDER
ALPHABETICAL   yes    no
         yes    96   129
         no  28566 5995568
> addmargins(example)
           ORDER
ALPHABETICAL   yes    no   Sum
         yes    96   129   225
         no  28566 5995568 6024134
         Sum 28662 5995697 6024359
> (example.test <- chisq.test(example, correct=FALSE))
  Pearson's Chi-squared test
data:  example
X-squared = 8458.9, df = 1, p-value < 2.2e-16
```

As you can see, this distribution is highly significant because the observed co-occurrence of *alphabetical* and *order* (96) is much larger than the expected one (1.07, check `example.test$expected`). Two issues have to be mentioned, however. First, the chi-squared test is actually not the best test to be applied here given the small expected co-occurrence frequency, which is much smaller than five. As a matter of fact, there is a huge number of other statistics to quantify the reliability and the strength of the co-occurrence relationship between two (or more) words (see Evert 2004). Second, testing a single collocation like *alphabetical order* is of course not very revealing, so, ideally, one would test all collocates of *alphabetical* and all collocates of *alphabetic* to identify semantic or other patterns in the distributional tendencies. We will return to collocational/collostructional statistics in the next chapter.

You should now do "Exercise box 4.4: Two-dimensional frequency distributions."

## 4.3 Numeric Dependent Variables

Another frequent kind of scenarios involves numeric dependent variables. In Section 4.3.1 I will briefly review a few useful descriptive statistics for such dependent variables, but will not discuss significance tests for this scenario (cf. Gries 2013 for much discussion and worked examples). In Sections 4.3.2 and 4.3.3 we will then distinguish two cases, one in which you have a two-level categorical independent variable and one in which you have a numeric independent variable.

### *4.3.1 No Independent Variables*

First, let us briefly review some useful descriptive statistics with which you can summarize numeric data. Usually, one distinguishes measures of central tendency – what you probably know as averages – and measures of dispersion.

Measures of central tendency serve to summarize the central tendency of a numeric variable in a single statistic. The most widely known measure of central tendency is the *mean*. You compute the mean by summing all observations/cases and dividing that sum by the number of observations. In R you use `mean`, which takes as its only argument a vector of values:

```
>·mean(c(1,·2,·3,·2,·1))¶
[1]·1.8
```

Despite its apparent simplicity, there are two important things to bear in mind. First, the mean is extremely sensitive to outliers: A single very high or very low value can influence the mean so strongly that it stops being a useful statistic. In the example below, the mean value of 168.1667 neither represents the first five nor the sixth value particularly well:

```
>·mean(c(1,·2,·3,·2,·1,·1000))¶
[1]·168.1667
```

In such cases, you should better either trim the data of extremes (using `mean`'s argument `trim`) or choose the median as a less sensitive measure of central tendency. If you sort the values in ascending order, the *median* is the middle value (or the mean of the two middle values). As you can see below, the median of this vector is 2, which represents the distribution of the first five values very well. Thus, whenever you look at a distribution as extreme as the above example or whenever your data are better conceived of as ordinal (i.e., rank data), you should use the median, not the mean:

```
>·sort(c(1,·2,·3,·2,·1,·1000))¶
·[1]····1····1····2····2····3·1000
>·median(c(1,·2,·3,·2,·1·1000))¶
[1]·2
```

The second important thing to bear in mind is to never, ever (!) report a measure of central tendency without a measure of dispersion because without a measure of *dispersion*, i.e., the variability of the data around its central tendency, one can never know how well the measure of central tendency does in fact summarize all of the data: Obviously, if in the case of the above extreme distribution you only report a mean in 168.17 without reporting the huge dispersion of the distribution, you're really not providing anything informative at all.[3] Let us look at a less contrived example by comparing two cities' average temperature in one year.

```
>·City1<-c(-5,·-12,·5,·12,·15,·18,·22,·23,·20,·16,·8,·1)¶
>·City2<-c(6,·7,·8,·9,·10,·12,·16,·15,·11,·9,·8,·7)¶
>·mean(City1)¶
[1]·10.25
>·mean(City2)¶
[1]·9.833333
```

From the means alone, it seems as if the two cities have very similar climates – the difference between the means is very small. However, if you look at the data in more detail, especially when you do so graphically as in Figure 4.6, you can see immediately that the cities have very different climates – I know where I prefer to be in February – but just happen to have similar averages; make sure you go through the code that generates Figure 4.6, which contains a lot of important functions and settings.

Obviously, the mean of `City2` summarizes the central tendency in `City2` much better than the mean of `City1` because `City1` exhibits a much greater degree of dispersion of the data throughout the year. Measures of dispersion quantify this and summarize it into one statistic. One widely used dispersion measure for numeric data is the so-called *standard deviation*, which is computed as represented in (6):

$$(6)\quad sd = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

While this may look daunting at first, the 'translation' into R (as applied to `City1`) should clarify this:

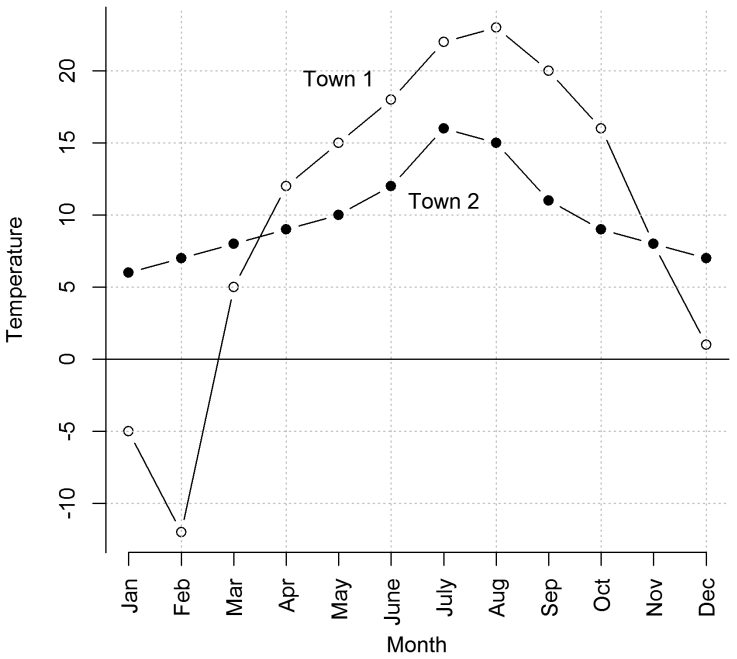


*Figure 4.6* Average fictitious temperatures of two cities.

```
>·numerator<-sum((City1-mean(City1))^2)¶
>·denominator<-length(City1)-1¶
>·sqrt(numerator/denominator)¶
[1]·11.12021
```

But of course R has a function for this:

```
>·sd(City1)¶
[1]·11.12021
>·sd(City2)¶
[1]·3.157483
```

Note in passing that the standard deviation is the square root of another well-known measure of central tendency, the so-called *variance* (the R function is `var`). As is obvious, the measure of dispersion confirms what was already obvious from the graph: the values of `City2` are much more homogeneous than the values of `City1`. It is important, however, to note here that you can only make this kind of statement on the basis of standard deviations or variances when the means of the two vectors to be compared are very similar. This is because the standard deviation is dependent on the size of the mean:

```
>·sd(City1)¶
[1]·11.12021
>·sd(City1*10)¶
[1]·111.2021
```

Thus, if the means of two distributions are markedly dissimilar, then you must not compare their dispersions by means of the standard deviation – rather, you should compare the dispersions on the basis of the *variation coefficient*, which you get by dividing the standard deviation by the mean:

```
>·sd(City1)/mean(City1)¶
[1]·1.084899
>·sd(City1*10)/mean(City1*10)¶
[1]·1.084899
>·sd(City2)/mean(City2)¶
[1]·0.3210999
```

As you can see, now the measures of dispersion for `City1` and its derivative are the same, and they are still considerably higher than that of `City2`, which is to be expected, given the similarities of the means.

Another way of summarizing the dispersion of a numeric variable is again using a measure that helps in cases when the data are decidedly non-normal (i.e., not likely bell-shaped) and/or when you have outliers etc. (just like the median may replace the mean

for measures of central tendencies in such cases). This measure of dispersion for ordinal variables is the so-called *interquartile range* (`IQR`), which gives you the length of the interval around the median that includes about half of the data:

```
>·IQR(City1)¶
[1]·14.5
>·IQR(City2)¶
[1]·3.5
```

Again, the conclusions are the same: `City1` is more heterogeneous than `City2`.

**Recommendations for further study/exploration**

- On how to get quartiles or deciles for a numeric vector: `quantile(0:100)¶` and `?quantile¶`.
- On how to get the range of a vector (i.e., a shortcut for `max(City1)-min(City1)`): `?range¶`.
- On how to get the median absolute deviation, another robust measure of dispersion: `?mad¶`.

A good way to look at such data is the *boxplot*, a graph that provides a lot of information about the distribution of a vector; see the top panel of Figure 4.7, which is generated as follows:

```
>·boxplot(City1,·City2,·notch=TRUE,·xlab="Cities",·
  ylab="Temperature",·main="Box·plot·of·temperatures:\nCity·1·vs.·
  City·2",·names=c("City·1","City·2"));·grid()¶
```

This plot tells you something about the central tendencies of both cities because the bold horizontal lines represent the medians of each distribution. Second, it also tells you something about dispersion because the horizontal lines delimiting the boxes at the top and at the bottom extend from the upper to the lower hinge (roughly, the two data points delimiting the highest 25 percent and the lowest 25 percent of all the data). Third, the whiskers – the dashed vertical lines with the horizontal limits – extend to the most extreme data points which are no more than 1.5 times the interquartile range from the box (the default value of 1.5 can be changed; enter `?boxplot¶` at the R prompt). Fourth, outliers beyond the whiskers would be represented by small circles. Finally, the notches extend to `±1.58*IQR/sqrt(n)` (enter `?boxplot.stats¶` at the R prompt for information on hinges and whiskers). If the notches of two boxplots do not overlap, this is strong *prima facie* evidence that the medians are significantly different from each other (but of course you would still have to test this properly). The function `grid()` draws the dashed gray line grid into the coordinate system.

If we apply this to our example, we can see all of what we have observed so far separately in one glance: The central tendencies of both cities are very similar (because the

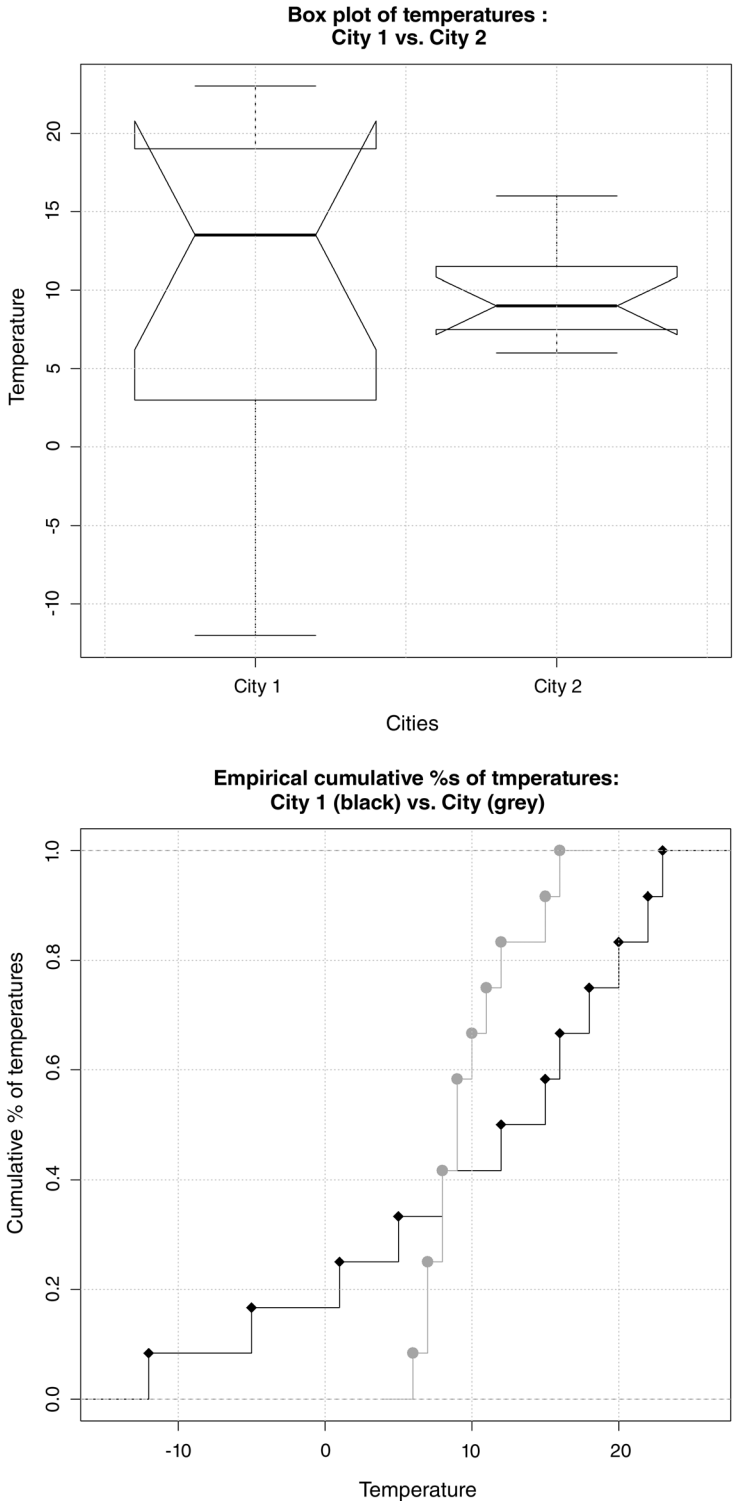


*Figure 4.7* Plots of the temperatures of the two cities.

medians are close to each other and the notches overlap). The first city exhibits much more heterogeneous values than the second city (because the boxes and the whiskers of the first city cover a larger range of the *y*-axis; also, the notches of `City1` are huge). Note finally the difference between the top and the bottom of the box in the top plot. While the bottom poses no problems, the top is sometimes difficult to understand because the horizontal line, so to speak, folds back-/downwards. This is R's way of giving you both the correct top end of the box and the full extension of the notch. In this case, the upper end of the box does not go up as far as the notch, which is why the line for the notch extends higher than the upper limit of the box.

The lower panel of Figure 4.7 shows what is an even more informative but often less intuitive plot, the *empirical cumulative distribution* function (`ecdf`) of the temperatures in each city, which is generated as follows:

```
>·plot(ecdf(City1),·verticals=TRUE,·pch=18,·cex=1.5,·
  xlab="Temperature",·
  ylab="Cumulative·%·of·temperatures",·main="Empirical·
  cumulative·%s·of·temperatures:\nCity·1·(black)·vs.·
  City·2·(grey)")
>·lines(ecdf(City2),·verticals=TRUE,·pch=19,·cex=1.5,·col="grey")
>·grid()
```

Each point indicates how much in percent of the data of one city (on the *y*-axis) are covered by the corresponding temperature on the *x*-axis and all smaller ones. For instance, we can estimate from the plot that 25 percent of all temperature values of `City1` are 1 or less, that 33 percent of all temperature values of `City1` are 5 or less, etc. This plot is often more informative than a boxplot because it doesn't bin data points (into a box covering the middle 50 percent of the data): Rather, every unique observed data point (here a temperature) is represented with a point and, for instance, the fact that the temperatures of `City2` are much less variable is revealed by the fact that the gray curve for `City2` covers a much smaller range of the *x*-axis limits than the black curve for `City1`.

**Recommendations for further study/exploration**

- On how to generate histograms of numeric data: `?hist¶`, which we will use in Chapter 5.
- On how to compute one-sample *t*-tests and one-sample Wilcoxon signed-rank tests to test whether the central tendency of one dependent numeric variable differs from an expected value (see below for the corresponding two-sample test): `?t.test¶` and `?wilcox.test¶`; Gries (2013: sections 4.3.1.1 and 4.3.1.2).
- On how to test whether one dependent numeric variable is distributed differently from what is expected (see below for the corresponding two-sample test): `?ks.test¶` and Gries (2013: section 4.1.2.2).

### *4.3.2 One Independent Categorical Variable*

Another very frequent scenario involves the situation in which you have a numeric dependent variable and a categorical independent variable; the more detailed discussion here will be restricted to the case where the independent categorical variable has just two levels. As an example for this scenario, we can return to the example we discussed above: the different lengths of subjects and (direct) objects. Let us first formulate our hypotheses, assuming we operationalize the lengths by the numbers of syllables:

$H_0$: On average, English subjects are as long as English direct objects; mean length of English subjects = mean length of English direct objects.

$H_1$: On average, English subjects not as long as English direct objects; mean length of English subjects ≠ mean length of English direct objects.

(One might in fact hypothesize that subjects are not just of different length, but also *how* they differ from objects, namely by being shorter, because referents of subjects are often assumed to be given or accessible information, and given/accessible information is usually encoded with less linguistic material than new(er) information. In the interest of keeping matters simple, we will stick with $H_1$ above, which only postulates a difference in length, but not also the direction of the difference.) Let us further assume we have investigated so many randomly drawn subjects and direct objects from a corpus until we had 152 instances of each and stored them in a data frame that conforms to the specifications in Section 4.1.4. We first load the data from <_qclwr2/_inputfiles/stat_subjectobject.csv>:

```
>·subj.obj<-read.table(file.choose(),·header=TRUE,·sep="\t",·
  comment.char="")¶
>·str(subj.obj);·attach(subj.obj)¶
'data.frame':···304·obs.·of··3·variables:
·$·CASE····:·int··1·2·3·4·5·6·7·8·9·10·...
·$·LENGTH··:·int··5·1·7·1·1·4·10·1·3·1·...
·$·RELATION:·Factor·w/·2·levels·"object","subject":·
  2·2·2·2·2·2·2·2·2·2·...
```

Next, we look at the data graphically by again using `boxplot`. Although this was not mentioned above, there are two different ways of using `boxplot`. One is to provide the function with vectors as arguments, as we did above for the two cities. We could do the same here like this:

```
>·boxplot(LENGTH[RELATION=="subject"],·LENGTH[RELATION=="object"],·
  notch=TRUE)¶
```

but if the data frame looks as I told you it should, then R also offers the possibility to use a *formula notation* in which the dependent variable is followed by a tilde (~) and the independent variable. We also immediately add the ecdf plot for objects and subjects (and to practice, you should also generate histograms for these data):

```
>·boxplot(LENGTH~RELATION,·notch=TRUE,·xlab="Grammatical·
  relations",·ylab=
  "Length",·main="Box·plot·of·lengths·of·grammatical·relations:
  \nobjects·vs.·subjects");·grid()¶
```

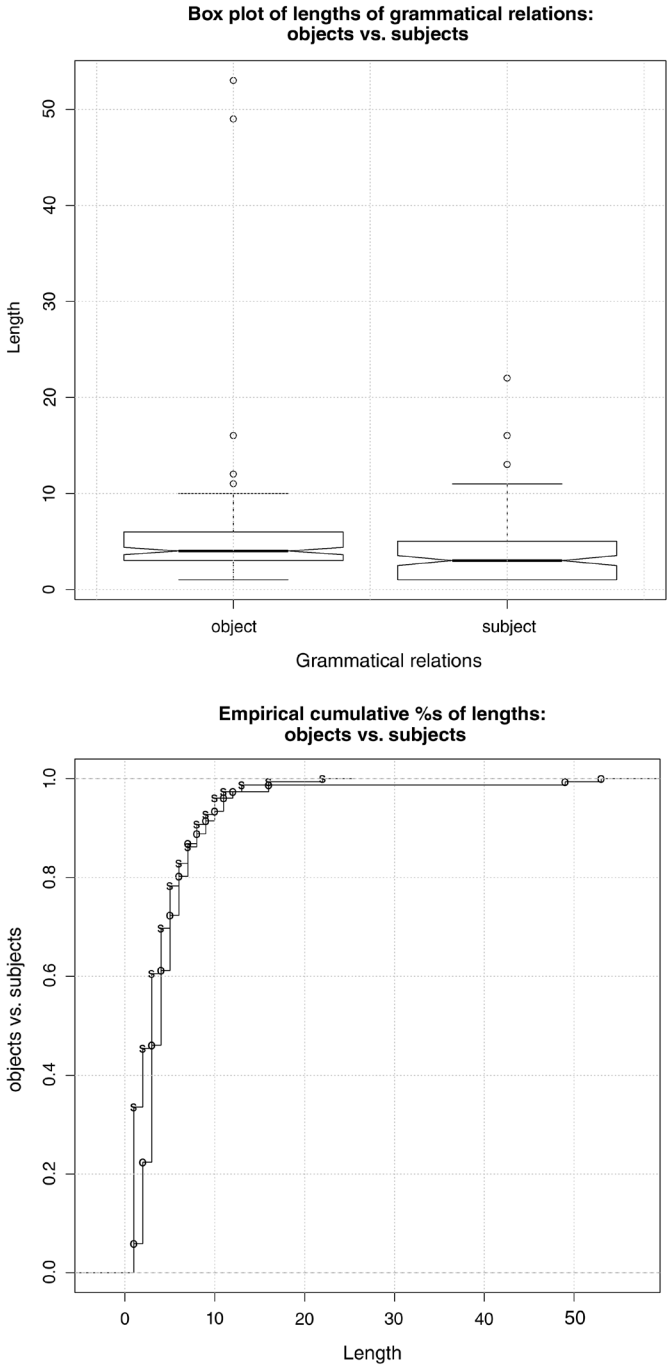


*Figure 4.8* Plots of the lengths of subjects and objects.

```
>·plot(ecdf(LENGTH[RELATION=="object"]),·verticals=TRUE,·pch="o",·
  xlab="Length",·ylab="Cumulative·%·of·lengths:\nobsjects·vs.·
  subjects",·main="Empirical·cumulative·%s·of·lengths:
  \nobjects·vs.·subjects")¶
>·lines(ecdf(LENGTH[RELATION=="subject"]),·verticals=TRUE,·pch="s");
  ·grid()¶
```

As we can see, the median of the objects is slightly larger than that of the subjects, but we can also immediately see that there is a lot of variability and that the ecdf curves are very similar. (Given the nature of the data, a logged *y*-axis for the boxplot might make the graph easier to grasp; also, see the code file for how to generate side-by-side histograms of the object and subject lengths.)

Let us now also compute the means and standard deviations as well as medians and interquartile ranges:

```
>·tapply(LENGTH,·RELATION,·mean)¶
··object··subject
5.118421·3.769737
>·tapply(LENGTH,·RELATION,·sd)¶
··object··subject
6.021962·3.381818
>·tapply(LENGTH,·RELATION,·median)¶
·object·subject
·····4·····3
>·tapply(LENGTH,·RELATION,·IQR)¶
·object·subject
·····3·····4
```

The usual test for such a case – a dependent numeric variable and an independent categorical variable – is either the so-called *t*-test for independent samples (if the independent categorical variable has two levels, as it does here) or a one-way ANOVA (if the independent categorical variable has three or more levels); if you use a one-way ANOVA with a binary independent variable, the results will be identical to those of the *t*-test. The next step again consists of determining whether we are in fact allowed to perform a *t*-test here. The *t*-test for independent samples is a parametric test and may only be used if:

- the observations of the samples are independent of each other such that there is no meaningful relation between, say, pairs of data points;
- the populations from which the samples are drawn are normally distributed (especially with sample sizes smaller than 30); and
- the variances of the two samples are homogeneous.

As to the former assumption, the subjects and objects were randomly drawn from a corpus so there is no relation between the data points. As to the latter two assumptions, Figure 4.8 suggests that at least the assumption of normality of the object and subject lengths will not be met: If the data were perfectly normally distributed, the ecdf curves should be *s*-shaped. There are formal statistical tests for these conditions (`?shapiro.test¶` and `?var.test¶`;

see Gries 2013: sections 4.1.1.1 and 4.2.1), but we will skip those here for lack of space and because corpus data tend to violate at least one if not both of these assumptions anyway; if you run the relevant tests on this data set, you would indeed see that you wouldn't be allowed to do a *t*-test.

Thus, we compute the non-parametric alternatives to the *t*-test/the one-way ANOVA, which are the *U*-test (if the independent categorical variable has two levels, as it does here) or the Kruskal–Wallis test (if the independent categorical variable has three or more levels). The function for the *U*-test is called `wilcox.test`, the function for the Kruskal–Wallis test is `kruskal.test`, and both accept the formula notation we have seen above:

```
>·wilcox.test(LENGTH~RELATION)¶
  Wilcoxon·rank·sum·test·with·continuity·correction
data:··LENGTH·by·RELATION
W·=·14453,·p-value·=·0.0001305
alternative·hypothesis:·true·location·shift·is·not·equal·to·0

>·kruskal.test(LENGTH~RELATION)¶
  Kruskal-Wallis·rank·sum·test
data:··LENGTH·by·RELATION
Kruskal-Wallis·chi-squared·=·14.639,·df·=·1,·p-value·=·0.0001302
```

Note how, since the independent variable has only two levels, the *p*-values of both tests are virtually completely identical so you could have used either test, but once your independent variable has more than two levels, you *have* to use `kruskal.test`. Here, both *p*-values are much smaller than 0.05, which is why we conclude that the difference in the median lengths is most likely not due to chance. Thus, this result would be summarized as follows: "The median length of direct objects was four syllables (interquartile range: three) while the median length of subjects was three syllables (interquartile range: four). Since the data violated the assumption of normality, a *U*-test/Kruskal–Wallis test was computed, which showed that the difference between the two lengths is highly significant (for the *U*-test: $W = 14{,}453$, $p < 0.001$): In the population of English for which our sample is representative, direct objects are longer than subjects."

You should now do "Exercise box 4.5: Averages."

### *4.3.3 One Independent Numeric Variable*

The final statistical method we look at is one where both the dependent and the independent variables are numeric. As an example, let us assume that we are again interested in the lengths of XPs. Let us also assume that we generally believe that the best way of operationalizing the length of an element is by counting its number of syllables. However, we may be facing a data set that is so large that we don't think we have

the time to really count the number of syllables, something that could require a lot of manual counting (but see Section 5.4.3). Since you already know how to use R to count words automatically, however, we are now considering having R assess the XPs lengths by counting the words and using that as an approximation for the XPs' lengths in syllables. However, we would first want to establish that this is a valid strategy so we decide to take a small sample of our XPs and count their lengths in syllables and lengths in words and see whether they correlate strongly enough to use the latter as a proxy towards the former.

This kind of question can be addressed using a linear correlational measure. Linear correlation coefficients such as $r$ or $\tau$ (see below) usually range from –1 to +1:

- negative values indicate a negative correlation which can be paraphrased by sentences of the form "the more . . . , the less . . . " or "the less . . . , the more . . . ";
- values near 0 indicate a lack of a correlation between the two variables;
- positive values indicate a positive correlation which can be paraphrased by sentences of the form "the more . . . , the more . . . " or "the less . . . , the less . . . ".

The absolute size of the correlation coefficient, on the other hand, indicates the strength of the correlation. Our hypotheses are therefore as follows:

$H_0$: The lengths in syllables do not correlate with the lengths in words; $r / \tau = 0$.

$H_1$: The lengths in syllables correlate positively with the lengths in words such that the more words the XP has, the more syllables it has; $r / \tau > 0$.

Let us now load the data from the file <_qclwr2/_inputfiles/stat_lengths.csv> into R:

```
>·lengths<-read.table(file.choose(),·header=TRUE,·sep="\t",·
  comment.char="")¶
>·attach(lengths);·str(lengths)¶
'data.frame':···302·obs.·of··2·variables:
·$·LENGTH_SYLL:·int··5·1·7·1·1·4·10·1·3·1·...
·$·LENGTH_WRD·:·int··3·1·3·1·1·3·6·1·2·1·...
```

As usual, we begin by inspecting the data visually. Conveniently, we can again use the generic `plot` function to produce the very simple scatterplot in the top panel of Figure 4.9: As before, the first-named variable is used for the $x$-axis:

```
>·plot(LENGTH_SYLL,·LENGTH_WRD);·grid()¶
```

However, the plot in the top panel of Figure 4.9a is not particularly nice, and not only because of the labeling of the axes. The much bigger problem is that one cannot see in it that the circle shown at coordinates (1, 1), for instance, actually represents 60 (!) circles all plotted on top of each other, whereas the identical-looking circle at (22, 10) is just one circle. Thus, we create the much better plot in the lower panel of Figure 4.9b:

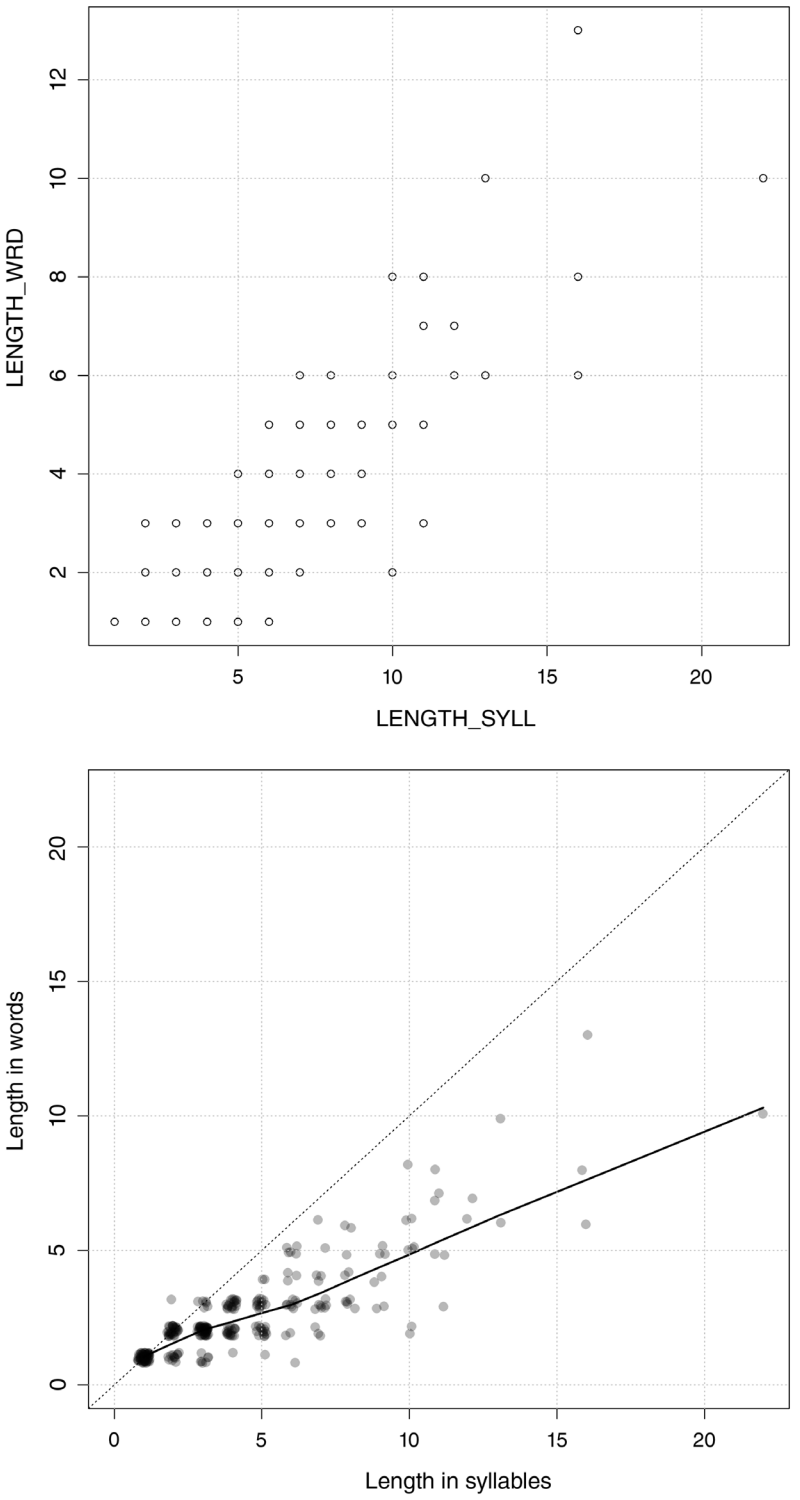


*Figure 4.9* Scatterplots of the lengths of words in syllables and words.

```
>·plot(jitter(LENGTH_WRD)·~·jitter(LENGTH_SYLL),·xlim=c(0,
  22),·ylim=c(0,22),·
  xlab="Length·in·syllables",·ylab="Length·in·words",·
  pch=16,·cex=1.25,·
  col=rgb(0,·0,·0,·50,·maxColorValue=255))¶
>·lines(lowess(LENGTH_WRD~LENGTH_SYLL),·lwd=2)¶
>·abline(0,·1,·lty=3);·grid()¶
```

This plot (1) jitters the data points by adding a small amount of noise to each coordinate using `jitter` (highlighting cases where many points would otherwise be identical); (2) uses the `rgb` function to create transparent grayscale colors (with overplotting leading to darker colors); (3) adds a thicker, smoother (`lowess`) line (`lines`) that summarizes the trend we see in the data; and (4) adds a main diagonal marking the cases where the word length is the same as the syllabic length: `abline` can take an intercept and a slope so if you set them values as above, you get a line representing the situations where $y = x$.

It is immediately obvious that there is a positive correlation between the two variables: The larger the syllabic length, the larger the lexical length. The correlation between two numeric variables is referred to as Pearson's product-moment correlation $r$ (for *regression*), but its significance test, too, comes with an assumption that corpus data usually do not meet, namely that the population from which your samples were taken is bivariately normally distributed, which in practice is often approximated by testing whether each variable is distributed normally. Figure 4.9 clearly shows that the data are not normally distributed at all, but rather Zipfian: A few very small lengths (e.g., 1–4) cover the vast majority of all data (about 66 percent). The solution is again to use a non-parametric alternative to the test we originally wanted to do. Instead of $r$, we now compute another correlation coefficient, Kendall's $\tau$ (read: tau). Fortunately for us, the function for both correlations is the same, `cor.test`. It minimally takes the two vectors to be correlated (the order doesn't matter) and the `method` argument, in which you specify the desired correlation test; in this particular case, we have a so-called directed $H_1$: We do not just expect $\tau$ to be different from 0 in whatever direction, we expect it to be positive, i.e., greater than 0 (which is called a *one-tailed test*); thus, we add another argument called alternative and set it to "greater":

```
>·cor.test(LENGTH_SYLL,·LENGTH_WRD,·method="kendall",
  ·alternative="greater")¶
·····Kendall's·rank·correlation·tau
data:··LENGTH_SYLL·and·LENGTH_WRD
z·=·16.814,·p-value·<·2.2e-16
alternative·hypothesis:·true·tau·is·greater·than·0
sample·estimates:
·····tau
0.7657678
```

It turns out that the correlation is rather high and highly significant so that we might want to be confident that the length in syllables can be reasonably well approximated by the length in words that is computationally easier to obtain. We can thus say: "There is a

highly significant positive correlation between the lengths of XPs in words and the lengths of XPs in syllables (Kendall's $\tau = 0.766$, $z = 16.8139$; $p_{\text{one-tailed}} < 0.001$)." Just so that you know in case you have data that are normally distributed: The computation of $r$ is just as easy because you only need to change "kendall" into "pearson".

**Recommendations for further study/exploration**

- To do simple regressions in R you can use `lm` or `ols` from the package `rms`. Caution: do not use these methods before having consulted some additional references!

## 4.4 Reporting Results

The previous sections have introduced you to a few elementary statistical methods that are widely applied to corpus data. For each of the methods, I have also shown you how the data would be summarized both in terms of graphic representation and in terms of statistical testing. However, now that you know something about each aspect of the empirical analysis of language using corpus data, I would also like to at least briefly mention the overall structure of an empirical paper that I myself find most suitable, which corresponds to the structure of quantitative/empirical data in many disciplines. This structure looks as represented in Table 4.7.[4]

This structure is very rigid, but given its wide distribution you will see it very often and, thus, get used to it, and other people will find your papers well structured. Also, the structure may seem excessively precise in that it requires you to specify even the retrieval algorithm or syntax and the software that you used. The reason for this is one I already mentioned a few times: Different applications incorporate different search parameters,

*Table 4.7* Structure of a quantitative corpus-linguistic paper

| *Part* | *Content* |
|---|---|
| Introduction | What is the question?<br>Motivation of the question<br>Overview of previous work<br>Formulation of hypotheses |
| Methods | Operationalization of variables<br>Choice of method (e.g., diachronic vs. synchronic corpus data; tagged vs. untagged corpora; etc.)<br>Source of data (which corpus/corpora?)<br>Retrieval algorithm or syntax<br>Software that was used<br>Data filtering/annotation (e.g., how were false hits identified? What did you do to guarantee objective coding procedures? How did you annotate your data? etc.)<br>Choice of statistical test(s) and how they are implemented |
| Results | Summary statistics<br>Graphic representation<br>Significance test: test statistic, degrees of freedom (where available), and $p$<br>effect size: the difference in means, the correlation, etc. |
| Discussion | Implications of the results for your hypotheses<br>Implications of the results for the research area |

and definitions of what is a word will differ across software applications. Thus, in order for others to be able to understand and replicate your results, you must outline your approach as precisely as possible.

Of course, if you use R, all of this is much less problematic because you can just provide your regular expression(s) or even your complete program as pseudocode. Pseudocode is a kind of structured English that allows you to describe what a program/script does without having to pay attention to details of the programming language's syntax (see the very useful page by John Dalbey at www.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html for more details), and Chapter 5 will make use of something very similar to pseudocode all the time. Here is an example of pseudocode for a script that generates a concordance of *right* when tagged as an adjective:

```
01 clear memory
02 choose corpus files to be searched
03 for each corpus file
04 load the file
05 if the corpus file is from the right register
06 downsize the file to all lines with sentence numbers
07 retrieve all matches (i.e., lines containing right as an adjective)
   from the file
08 store all matches with the names of the files successively
09 end if
10 end for
11 insert tab stops for better output
12 delete unwanted tags and unnecessary spaces
13 output result into a file
```

If you provide this information together with the actual regular expressions you used to

- check the right register in line 5 ("<teiHeader type.*S conv</classCode>");
- downsize the file in line 6 ("<s·n=");
- retrieve the matches in line 7 ("<w·AJ0(-AV0)?>right\\b");
- delete unwanted tags and unnecessary spaces in line 12 ("<.*?>" and "·*\t·*");

then you will be more explicit than many corpus linguists are at present, and than I myself have been on too many occasions in the past.

Now we are at the point where we have covered nearly all the programming knowledge you need, we can finally get down to business and deal with the case studies.

## Notes

1 I will not deal with other kinds of variables such as confounding or moderator variables; see Gries (2013b: 12) for a discussion.

2 As a matter of fact, five is the most widely cited threshold value. There is, however, a variety of studies that indicate that the chi-squared test is fairly robust even if some expected frequencies are smaller than five, especially when the null hypothesis is a uniform distribution (as in the present case). In order not to complicate matters here any further, I stick to the conservative threshold and advise the reader to consult statistics textbooks for more comprehensive coverage. If whatever threshold value adopted on the basis of the pertinent literature is not met, you should either use the binomial test or the multinomial test (see Gries 2013b).

3 The same is actually true of corpus frequencies. Too many studies content themselves by reporting frequencies of occurrence or co-occurrence only, without also reporting a measure of corpus dispersion, which can sometimes hide a considerable bias in the data; see Gries (2008, 2010), Lijffijt and Gries (2012) for discussion, and Section 5.1 for case studies.
4 If you have more than one case study, then each case study gets its own methods, results, and discussion sequence, and then you pull together the results of all case studies in one section entitled "General discussion" at the end of the paper.

## References

Baayen, R. Harald. 2008. *Analyzing linguistic data: A practical introduction to statistics using R*. Cambridge: Cambridge University Press.

Bortz, Jürgen. 2005. *Statistik für Human- und Sozialwissenschaftler*. 6th ed. Heidelberg: Springer.

Crawley, Michael J. 2012. *The R book*. 2nd ed. Chichester: Wiley and Sons.

Dalbey, John. 2003. Pseudocode standard. Retrieved November 26, 2006, from www.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html.

Evert, Stefan. 2004. The statistics of word cooccurrences: Word pairs and collocations. Ph.D. diss., University of Stuttgart.

Gries, Stefan Th. 2003. *Multifactorial analysis in corpus linguistics: A study of particle placement*. London: Continuum Press.

Gries, Stefan Th. 2008. Dispersions and adjusted frequencies in corpora. *International Journal of Corpus Linguistics, 13*(4), 403–437.

Gries, Stefan Th. 2010. Dispersions and adjusted frequencies in corpora: Further explorations. In Stefan Th. Gries, Stefanie Wulff, & Mark Davies (Eds.), *Corpus linguistic applications: Current studies, new directions* (pp. 197–212). Amsterdam: Rodopi.

Gries, Stefan Th. 2013. *Statistics for linguistics with R*. 2nd rev. and ext. ed. Berlin: De Gruyter Mouton.

Johnson, Keith. 2008. *Quantitative methods in linguistics*. Malden, MA: Blackwell.

Lijffijt, Jefrey, & Stefan Th. Gries. 2012. Correction to "Dispersions and adjusted frequencies in corpora". *International Journal of Corpus Linguistics, 17*(1), 147–149.

Mukherjee, Joybrato. 2007. Corpus linguistics and linguistic theory: General nouns and general issues. *International Journal of Corpus Linguistics, 12*(1), 131–147.

Murrell, Paul. 2011. *R graphics*. 2nd ed. Boca Raton, FL: Chapman and Hall.

# 5 Using R in Corpus Linguistics
## Case Studies

> Now that corpus linguistics is turning more and more into an integral part of mainstream linguistics, . . . we have to face the challenge of complementing the amazing efforts in the compilation, annotation and analysis of corpora with a powerful statistical toolkit and of making intelligent use of the quantitative methods that are available.
>
> (Mukherjee 2007: 141)

In this chapter you will learn how the functions introduced in Chapter 3 can be applied to the central tasks of a corpus linguist, namely to retrieve and process linguistic data to produce frequency lists and concordances, to work with collocations, to compute dispersion statistics, and generally work with a variety of different corpora. Obviously, the number of different formats of corpora as well as the number of different tasks you might wish to perform are actually so large that we will not be able to look at all possible combinations. Rather, I will exemplify how to perform many different tasks on the basis of several frequently used corpora and corpus formats. (A few of the case studies were part of the additional assignments of the first edition, but you will see that there are many new ones and even those that were posted before have been rewritten (and are often much faster now) and are now heavily commented.) The tasks we will perform in the case studies below are roughly grouped according to the kind of corpus-linguistic task they involve, but this grouping is only a heuristic because (1) several scripts involve tasks that defy an easy characterization into frequency lists, dispersion, collocation, and concordancing, and (2) several scripts involve more than one of these aspects; they cover a wide range of things in the hope that you can generalize from them to your own applications. Within each group the tasks or case studies are ordered according to difficulty, but this, too, is only approximate because difficulty is hard to operationalize objectively.

In some sense, you have arrived in a 'good news, bad news' kind of situation. The good news is that you actually already know nearly all the functions we will use: You will have to learn only a very small number of new functions for this chapter and in fact you already know virtually all functions I use for my own research. The bad news is that it's still not an easy chapter because it is all about putting these functions together in the right ways to get stuff done, and in that sense, R *is* like a natural language, where it's often easy to learn vocabulary and their local syntax (functions and their arguments), but much less easy to employ them properly in sentences and paragraphs (scripts). But fear not, the chapter is written in a way that I hope will be useful and also in a way that will make it easy for you to recycle (parts of) scripts for your own work: Wherever possible I wrote the code such that I didn't hardcode particular numbers of files etc. into the script but have R determine the lengths of vectors and lists (so that you don't have to fiddle with details as you recycle the script for your own work).

With very few exceptions, the case study sections all have the same structure: They begin with a short characterization of what the goal of the task/script is; sometimes, this will be just a prose explanation, sometimes I will also show you the results of the script – a results table or a plot, for instance. Then, each case study has four sections, which are designed to help you understand the programming process in a stepwise fashion. This sequence is important because it is very easy to show someone the solution to a programming problem, but very hard to teach someone to see why that solution is a good one and even harder to teach someone to develop a way of thinking that enables them to find such solutions themselves, because programming requires an analytical and to some extent taxonomic and modular kind of thinking. In this context, a small half-humorous side-remark is in order: One thing that you should do right now is drop your human way of thinking and your human expectations, because one other important aspect of writing scripts is to understand that the 'thing' implementing your instructions is a 'stupid computer,' i.e., an entity that will cover up your gaps in scripts with well-intentioned base understandings of what words are, which words you might be interested in etc. – no, you must try to write scripts such that all eventualities are covered. That means you may *think* that every corpus file contains matches of an expression, but your script still needs to be able to handle cases when a corpus file does not contain a match. That also means you may think a certain website may be downloadable, but your script still needs to be able to handle the situation that it's not. And so on.

Back to the sequencing of each section: The sequencing I am using here is designed to help you zoom in from a general description of the task into the more specific aspects of the code such as (1) the R functions you will need to use and (2) the structure of the script, i.e., how and in which order the functions are used. These are the four parts, or subsection headings:

- **What are the things we will need to do?** This section explains in plain English which steps the relevant task involves; it uses hardly any R code but already introduces the kinds and names of a few data structures that the script will contain.
- **What are the functions we will need for that?** This section lists all the main functions one will need to use to perform the things implied by the description formulated in the previous step.
- **Thus, this is the overall structure of the script**. This section provides a skeleton of the R code we will use in what is called pseudocode: a description of the algorithm and structure of a program that performs a particular task. It is imperative that you read this part with the relevant script open in RStudio because in the pseudocode I will provide the line numbers of the relevant R script from the companion website so you can see exactly which lines of code in the script do which part of the pseudocode, or how the pseudocode is 'translated' into actual R code. The files with all the scripts are in the folder <_qclwr2/_scripts/> and all begin with "05_"; this part will greatly help you understand the logic of the scripts.
- **Which aspects of the script are worth additional comment?** If you look at the script files, you will see that they are *very* heavily commented: Often even a single function call is broken up into several lines so that each argument can be explained, and sometimes I will break down regular expressions into multiple lines (remember free-spacing from above?) to explain everything in detail. However, sometimes scripts involve something that I think merits additional explanation here, or they involve something you haven't seen yet at all or in the form in which something is used. If there are such situations – and not every case study has such parts – then this section provides additional discussion of these aspects of the scripts.

Note that sometimes scripts provide more than one solution to a problem, and note that even if scripts do not provide alternative solutions, there are usually many ways to solve a problem. The solutions I provide here are probably *relatively* good ones, but there will be others and probably better ones, so if, after thinking about a task or reading my solution you come up with a better one, great – post it on the newsgroup for the Google Group for this book!

Ok, let's get started.

## 5.1 Dispersion

### *5.1.1 Dispersion 1:* HIV, Keeper, *and* Lively *in the BNC*

As mentioned in Section 2.3, it is often extremely important to not just consider the frequency of morphemes, words, constructions, etc. in a corpus, but also their dispersion, because elements may have very similar overall frequencies in a corpus but nevertheless be very differently dispersed in that corpus. Above, I mentioned Leech, Rayson, and Wilson (2001), who demonstrated this fact by showing how the words *HIV*, *keeper*, and *lively* are similarly frequent, but differently dispersed in the BNC, and in this case study we will try to replicate their findings while at the same time write our script such that it can be extended easily to more different words than just those three. In addition, we also want to compute Gries's (2008; 2010) measure of dispersion *DP*, which is computed by computing for each corpus file (1) the size of that file as a percentage of the whole corpus and (2) the relative frequency of occurrences of the word in question in that file as a percentage, and then take half of the sum of the absolute differences of these two percentages per file, as shown in (1):

(1) $0.5\Sigma_{i=1}^{n}|\%\ of\ corpus\ part_i - \%\ of\ word\ occ.s\ in\ corpus\ part_i|$

Here's a little made-up example of a corpus with three files: the files make up 50 percent, 30 percent, and 20 percent of the corpus, and of all occurrences of the word in question, 70 percent, 20 percent and 10 percent are in the corresponding files. Then, *DP* is computed like this:

```
> file.sizes.ie.exp.percs<-c(0.5, 0.3, 0.2)¶
> word.hits.ie.obs.percs<-c(0.7, 0.2, 0.1)¶
> 0.5*sum(abs(word.hits.ie.obs.percs-file.sizes.ie.exp.percs)) # 0.2¶
```

Simplifying a bit, *DP* ranges from 0 (a word is perfectly evenly distributed in the corpus, i.e., in accordance with the sizes of the corpus files) to 1 (a word is completely unevenly distributed in the corpus). If we want to compute *DP* for a word *w*, we therefore need to determine (1) the frequency of *w* in every single corpus file (which may be 0 or higher) and (2) the size of every single corpus file (which will be 1 or higher). Thus, if we have three words for which we want to compute *DP*s and we have $n$ = 4,049 corpus files, we want two forms of output: First, for the frequencies of each word *w* in each corpus file $c_{1-n}$, we can use a list `freqs.of.words` that has three components (one for each word $w_{1-3}$ whose *DP* value we want to compute), and each of these three components is a numeric vector of length 4,049 that states how frequent each word is in each corpus file $c_{1-n}$. Second, for the

corpus file sizes, we need a numeric vector `sizes.of.files.in.words` of length 4,049, one for each corpus file $c_{1-n}$.

**What are the things we will need to do?**

- We need to define a vector `corpus.files` that contains the paths to all corpus files, which should all be in one directory (potentially with sub-directories).
- We need to define a vector `words` that contains the words for which we want to compute *DP* values and then also a vector `search.expressions` that changes `words` into regular expressions we can search for in the corpus files (using the annotation of the corpus). Recall from Section 3.6.3, this is a script in which we know the dimensions of the output in advance: If we have three words and 4,049 corpus files, we know, for instance, that the vector corresponding to `sizes.of.files.in.words` above will need to have 4,049 slots, and we know that the list that collects the three words' frequencies will need three components each with 4,049 slots. Thus, we make sure that we define data structures with these dimensions before we enter any of the loops.
- We will access each of these files, i.e., use a first `for`-loop (using `i` as a counter) where on each iteration we load (with `scan`) one corpus file; once we have that file in memory, we need to determine its size in words, which we save into a slot in `sizes.of.files.in.words`.
- Also, while we have that corpus file in memory, we need a second `for`-loop (using `j` as a counter) that iterates over each `search.expression` to determine how frequent each word $w$ in `words` is in each corpus file $c_{1-n}$, once we have the frequency of word $w_j$ in corpus file $c_i$, we store its frequency in the corresponding slot of `freqs.of.words[[j]][i]`.
- We complete the inner `for`-loop for the words and then the outer `for`-loop for the corpus files; we do that in this order so that we load every corpus file just once to look for the three words in it rather than loading each corpus files three times.
- Finally, once we have the results of this loop, we can compute *DP* for each word along the lines discussed above and then generate a plot with, say, the frequencies of the words on the *x*-axis and the *DP* values of the same words on the *y*-axis.

**What are the functions we will need for that?** Obviously, we need to be able to define the corpus files we want to search, which means it will be useful to use `rchoose.dir` to define the directory containing the corpus files; also, we will need to be able to retrieve all the file names from that directory using `dir`. We will need to define vectors and a list (with `integer` and `list`), and we will need to use one `for`-loop nested into another. We will use `scan` to load each corpus file in the outer loop, use `tolower` to make everything case-insensitive, and we will use `grep` to make sure we only search for words in the part of the corpus file that's not the header. To find the words we are interested in, we can use `just.matches` (which we would then define at the beginning of the script again) or `exact.matches.2` (which we would then have to source at the beginning of the script).

Once we're done with the loops, we will need to compute the file sizes in percent, which means we need `sum` to compute the overall corpus size from `sizes.of.files.in.words` so we can compute each file's size as a fraction of that, and we will need `lapply` to access each element of `freqs.of.words` to do the same. Then, we loop over each word $w_{1-3}$ to compute *DP* as defined in (1) above and then we plot it using `plot`. In order to make the plot easier to interpret, we might use the first letters of the three words in question as point characters rather than a circle, which would then have to be labeled anyway; for that we need `substr`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_01_dispersion1.r>; read this side by side with the actual script in RStudio:

clear memory and define and/or load all relevant packages and/or functions (18–33)
define the words to search for (and corresponding regular expressions): `words` and `search.expressions` (36–38)
define the corpus files to search: `corpus.files` (40–41)
define the data structures that will collect the results: `sizes.of.files.in.words` (43–44) and `freqs.of.words` (46–53)
outer `for`-loop (counter = `i`, over `seq(corpus.files)`) (57, –95)
  output a progress report (% of files processed) (58–60)
  load the `i`-th file from `corpus.files` and switch it to lower case (62–66)
  discard the header/find the sentences (67–70)
  inner `for`-loop (counter = `j`, over `seq(search.expressions)`): (72, –83)
    store in `freqs.of.words[[j]][i]` the frequency of the j-th word in the corpus file (73–76 or 77–82)
  compute the corpus file size in words (85–87)
compute the corpus file sizes in percent and store it in `sizes.of.files.in.words[i]` (99–102)
compute the words' frequencies per file in percent (104–108)
compute and show *DP*s (110–120)
visualize (122–126)

**Which aspects of the script are worth additional comment?** One aspect worth commenting on in more detail is the creation of `search.expressions` from `words`: We take each word and paste the closing angular bracket from the tag in front of it, and an optional space ("·?") before the closing word tag "</w>" after it (and we use `paste0` to paste without a separator) so that we can use each element of this as a regular expression. Creating search expressions like this, i.e., by including closing tags etc., makes discarding the header less necessary since it is extremely unlikely that the search expressions we're pasting together here would match anything in the header – we're just being extremely careful here:

```
>·words<-tolower(c("HIV",·"keeper",·"lively"))¶
>·search.expressions<-paste0(">",·words,·"·?</w>")¶
```

Another thing that's useful to talk about is the creation of the data structures that will collect the results. Creating `sizes.of.files.in.words` is easy: The following line (44 in the script) creates an integer vector that has as many slots as you have corpus files:

```
>·sizes.of.files.in.words<-integer(length=length(corpus.files))¶
```

But creating the list to collect the words' frequencies in each corpus file is more involved. We proceed in two steps: First, we create a temporary vector that has as many slots as there are corpus files and make sure that each slot has the name of one corpus file:

```
>·temp<-rep(NA,·length(corpus.files))¶
>·names(temp)<-basename(corpus.files)¶
```

But then we also need our list with as many elements as there are words for which to compute *DP*. Thus, we create a list with that many empty components (line 51), give each list element the name of a word we are computing *DP* for (line 52), and then use the subsetting function "[" to put `temp` into each list element:

```
>·freqs.of.words<-vector(mode="list",·length=length(words))¶
>·names(freqs.of.words)<-words¶
>·"["(freqs.of.words)<-list(temp)¶
```

The rest of the text-processing loops should be clear from the script file. But then what? Computing the sizes of the corpus files in percent is unproblematic, but what about computing the relative frequencies of the words in the corpus files?

```
>·freqs.of.words.perc<-lapply(freqs.of.words,·function(x)·{·x/
  sum(x)·})¶
```

Remember that `freqs.of.words` is a three-element list. We use `lapply` to access each element of the list and then apply an anonymous/inline function to it that takes the list element – which you know is a numeric vector – temporarily refers to it as `x`, and divides each number of that vector `x` (i.e., each frequency of a word in a corpus file) by the sum of all numbers in `x` (i.e., the overall frequency of a word in the whole corpus). That way, we now have the two vectors that at the beginning of this section were exemplified as `sizes.of.files.in.words` and `freqs.of.words`, which means we can use a small `for`-loop to compute *DP* for each word and plot the results.

### *5.1.2 Dispersion 2:* Perl *in a Wikipedia Entry*

Let us now consider another application of dispersion: This time around we want to visualize the dispersion of the string "Perl" (case-insensitively) in an older version of its Wikipedia entry (already cleaned and available in <_qclwr2/_inputfiles/corp_perl.txt>). Specifically we want to create the results shown in Figure 5.1.

The left panel is a dispersion plot whose *x*-axis represents all the words in the 'corpus' which shows a vertical line when a word is "Perl" (case-insensitively) and no line otherwise. We can easily see how, for instance, there are a lot of occurrences at the very end – presumably that's the section with references and links. The right panel is a similar plot but it bins the words in the corpus (here into ten equally large parts), and again we can see that there are a lot of occurrences of "Perl" in the last 10 percent slice of the corpus.

**What are the things we will need to do?**

- We need to load and switch to lower case our single corpus file and split it up into words (say, into a character vector `word.tokens`); we may have to get rid of empty character strings resulting from the splitting.

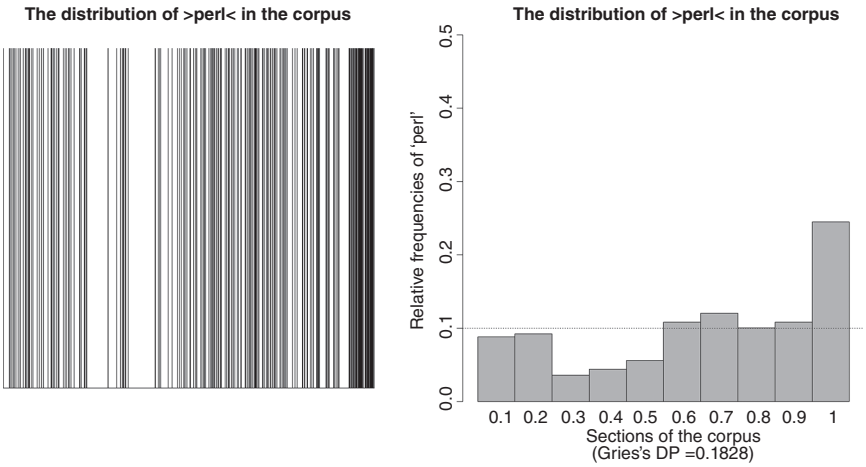


*Figure 5.1* Dispersion results for *perl* in its Wikipedia entry.

- We need to check for every word token in `word.tokens` whether it is "perl" or not.
- For the left panel, we plot a vertical line whenever a word token is "perl".
- For the right panel, we need to create a vector `corpus.parts` that lists for every element of the vector `word.tokens` which of ten equally sized parts it's in and then count how often "perl" shows up in each of those parts; from that we can compute *DP* as above.
- Then, since we then already have the observed percentages of how often "perl" shows up in each of the ten parts, we generate a bar plot that represents these ten percentages.

**What are the functions we will need for that?** We need `scan` and `tolower` to load and prepare the corpus file, and we will need `strsplit` to retrieve the word tokens as well as `unlist` to turn the list returned by `strsplit` into a vector `word.tokens`. We will discard empty character strings with `nzchar` and test whether each word token is "perl" with a simple logical expression testing for equality (i.e., `==`) and then use `plot` (with `type="h"`, for *histogram*) to plot the left panel (because `FALSE` = 0 and `TRUE` = 1). For the right panel, we will define a number of corpus parts we want (here ten) so that the script can easily be changed to accommodate different divisions of the corpus into parts. Then, we create a vector of word positions in `word.tokens`, which will run from 1 (for the first word in `word.tokens`) to $n$ = 6,065 (for the last word in `word.tokens`) with `seq` and we then split that vector up into ten equally long parts using `cut`. The factor that `cut` returns will have ten levels and can be cross-tabulated (using `table`) with the `TRUE`s and `FALSE`s from the logical vector created to plot the left panel.

The two final steps are computing *DP* and plotting the bar plot. For the former, our table contains the frequencies of "perl" in each corpus part (in the column for `TRUE`s), which we can divide by the overall frequency of "perl" in the file to get percentages (to be stored in a vector called `obs.percs`), and we can use the function `rowSums` to compute the corpus part sizes in percentage in a vector `exp.percs` (which should all be really close to 10 percent, given how we split the corpus up into ten parts above), from which we can compute *DP*. Finally, we use the function `barplot` to plot the observed percentages of "perl" in the corpus parts and customize the plot.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_02_dispersion2.r>; read this side by side with the actual script in RStudio:

clear memory (1), load the corpus file into a vector `text.file`, and switch it to lower case (3–4)

split `text.file` into words (simplistically at non-word characters) and `unlist` the result to obtain a vector `word.tokens` (6–10), from that, discard empty character strings (11)

use a logical expression to check for each element of `word.tokens` whether it is "perl"; save the logical values into a vector `is.perl` (13–14, note the alternative syntax in 16)

create the left panel by plotting the logical values (18–20)

define the number of parts you want to divide the corpus into (22–23)

define a `se`quence of all 6,065 positions of words and `cut` it up into ten parts, store that in a factor called `corpus.parts` (25–27)

cross-tabulate `corpus.parts` and `is.perl` (use `addmargins` to get row and column totals) (29–34)

compute the percentages of uses of "perl" per corpus part from the relevant column of that table (36–37) as well as the corpus part sizes (38) and compute *DP* (39)

visualize (41–50, note the suggestion in 52–53)

**Which aspects of the script are worth additional comment?** There shouldn't be any here, the above plus the comments in the code file should make things fairly clear.

## 5.2 Frequencies, Frequency Lists, and Key Words

### *5.2.1 Character* N*-Grams*

Sometimes, it might be useful to generate *n*-grams, i.e., chains of multiple elements of the same resolution. Typically, these would be word *n*-grams, but one can also find applications of character *n*-grams. For instance, the three-grams on the level of the word of the preceding sentence would be "Typically these would", "these would be", "would be word", etc., and the three-grams on the level of the character would be "Typ", "ypi", "pic", etc. In this section, we will write a small script that can generate character three-grams (and will also write a new function for this in R); in the next section, we will then write a script that can generate word three-grams; for simplicity's sake, both applications will work with a simple unannotated text file, <_qclwr2/_inputfiles/corp_gpl_long.txt>.

**What are the things we will need to do?**

- We need to load and switch to lower case our single input file.
- We will get rid of everything that's not a letter from *a* to *z* and a space (for simplicity's sake – this [cw]ould need to be tweaked for other kinds of input files of course) – and merge all lines of the input file into one long string, which we also clean up and prepare for output (for instance, replacing spaces with underscores so one can see word boundaries better).
- We will then determine how many character three-grams there will be (so as to avoid having a vector of three-grams grow all the time, which is bad practice;

recall Section 3.6.3) and create a vector `all.3grams1` of that length to collect the results.
- We will then go through the one long character string that is now our input file with a loop and extract all character three-grams, storing each one of them in `all.3grams1`.
- Finally, we will create a sorted frequency list of all character three-grams, explore it briefly statistically, and plot it – can you guess what the most frequent three-gram will be?

In addition to the above, I will present a version of this script that does not use a loop but a function from the family of `apply` functions (see `?mapply¶`), and I will show how you can use that kind of code to write and apply a function called `character.ngrams` for your future work.

**What are the functions we will need for that?** We need `scan` and `tolower` to load and prepare the corpus file, and we will need `gsub` to clean up characters we do not want to consider, and we will need `paste` and `gsub` again to create a clean version of a character vector with one element that contains the whole text.

We then need `nchar` to determine how many three-grams we will have to create/retrieve and then store, and a `for`-loop that applies `substr` to the cleaned text vector to extract the three-grams. Finally, we will use `sort` and `table` to create our frequency list of the three-grams, `quantile` to explore their distribution, and `plot` with `type="h"` again to visualize the results.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_03_char-ngrams.r>; read this side by side with the actual script in RStudio:

clear memory (1)

load the corpus file into a vector `textfile` and switch it to lower case (3–8)

delete all characters you do not want to include to obtain `textfile.cleaned` (10–14)

use `paste` to merge all the elements of `textfile.cleaned` into a one-element vector `textfile.oneline.1` (16–19)

use `gsub` to delete excess whitespace that results from the cleaning and merging to create `textfile.oneline.2` (21–25)

create `textfile.oneline.3` by `gsub`bing spaces in `textfile.oneline.2` with underscores so that three-grams involving word boundaries are easier to see in the output (27–31)

determine how many characters `textfile.oneline.3` has with `nchar`, that number minus 3 – 1 = 2 (because you are generating three-grams) will be the number of three-grams; reserve a vector `all.3grams1` that has that many slots (35–44)

use a `for`-loop (counter = `i`, over `seq(all.3grams1)`) to retrieve every three-gram from `textfile.oneline.3` and store each in the `i`-th slot of `all.3grams1` (46–52)

create a sorted frequency list of `all.3grams1`; call it `sorted.ngram.table` (54–56)

compute `sorted.ngram.table`'s deciles to see if the frequency distribution is (very) skewed/Zipfian or not (58–61)

visualize the sorted frequency table (63–68)

**Which aspects of the script are worth additional comment?** One function call that may be useful to mention here is the one in lines 38–44 simply because it is something you

have not seen before: It uses the function `length` to count the number of elements that an object has, but the interesting (space-saving) thing about it is that the argument to `length` is not just a data structure but actually an assignment of a name to a data structure:

```
>·length(all.3grams1<-character(length=nchar(textfile.
  oneline.3)-2))¶
```

That is, the above is the short version of this:

```
>·all.3grams1<-character(length=nchar(textfile.oneline.3)-2)¶
>·length(all.3grams1)¶
```

Now what about the alternative solution that does not require a `for`-loop? This one is shown in lines 72–77 and is a bit more tricky and involves a function I haven't discussed, but only recommended to you for further exploration before, `mapply`. As the help says, it's a multivariate version of `sapply` that takes as its first argument a function and as additional arguments it can take arguments to the first-named function. In the following line, this means, use the function `substr` with `textfile.oneline.3` as `substr`'s first argument (i.e., as the thing from which to extract substrings), with the positions 1, 2, . . . 17,093, 17,094 as the second argument to `substr` (i.e., the starting points of the substrings to extract), and with the positions 3, 4, . . . , 17,095, 17,096 as the third argument to `substr` (i.e., the end points of the substrings to extract), and we do not allow `mapply` to name the output elements, because that makes the output extremely unwieldy:

```
>·all.3grams2<-mapply(substr,·textfile.oneline.3,·¶
+····1:(nchar(textfile.oneline.3)-2),·¶
+····3:nchar(textfile.oneline.3),·¶
+····USE.NAMES=FALSE)¶
```

A third alternative to create all character three-grams also avoiding a loop is shown in lines 81–86, which involves just using `substr`, but using `rep` to make its first argument – the string from which to extract substrings – as long as its arguments `start` and `stop`:

```
>·all.3grams3<-substr(¶
+·····rep(textfile.oneline.3,·(nchar(textfile.oneline.3)-2)),·¶
+·····1:(nchar(textfile.oneline.3)-2),·¶
+·····3:nchar(textfile.oneline.3))¶
```

Explore the rest of the code file, lines 90–110, to see how the second approach – the one using `mapply` – is then used to define a function `character.ngrams`, where you or any other user provides a character vector of length 1 as the first argument

and the desired gram length as the second to get the job done very quickly. As a 'homework assignment' you may want to think about how you could create a version of `character.ngrams` that works on input vectors of length 1 or longer ones. Also, check out `?stringdist::qgrams¶`, which is the R notation to say "the help file for the function `qqgrams` from the package stringdist".

### *5.2.2 Word* N*-Grams*

The next case study is very similar to the preceding one: we are again interested in *n*-grams – three-grams, to be precise – but this time in three-grams of words, not characters; for the sake of simplicity, we will work with the same input file, <_qclwr2/_inputfiles/corp_gpl_long.txt>.

**What are the things we will need to do?**

- We need to load and switch to lower case our single input file.
- We will retrieve all words of the file by splitting on everything that's not a letter from *a* to *z* and a space (again, this [cw]ould need to be tweaked for other kinds of input files).
- We will then determine how many word three-grams there will be (so as to avoid having a vector of three-grams grow all the time – same as above) and create a vector `all.3grams1` of that length to collect the results.
- We will then go through the vector of all words that now is our input file with a loop and extract all word three-grams, storing each one of them in `all.3grams1`.
- Finally, we will create a sorted frequency list of all word three-grams, explore it briefly statistically, and plot it.

As before, I will also present a version of this script that does not use a loop, but `mapply`, and I will also show how you can use that kind of code to write and apply a function called `word.ngrams` for your future work.

**What are the functions we will need for that?** We need `scan` and `tolower` to load and prepare the corpus file, and we will need `strsplit` and `unlist` to split up the corpus file into words (and `nzchar` to eliminate empty strings) and put them into a vector called `textfile.words`. We then need `length` to determine how many three-grams we will have to create/retrieve and then store, and a `for`-loop that applies `paste` (with `collapse`) to the text vector to create the three-grams. Finally, we will use `sort` and `table` to create our frequency list of the three-grams, `quantile` to explore their distribution, and `plot` with `type="h"` again to visualize the results.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_04_word-ngrams.r>; read this side by side with the actual script in RStudio:

clear memory (1)

load the corpus file into a vector `textfile` and switch it to lower case (3–8)

use `strsplit` to split up at every occurrence of 1+ characters you do not want (11–13), `unlist` the result into a vector `textfile.words` (14), and remove empty strings from it (16–17)

*(continued)*

*(continued)*

determine how many words `textfile.words` has with `length`, that number minus 3 – 1 = 2 (because you are generating three-grams) will be the number of three-grams; reserve a vector `all.3grams1` that has that many slots) (21–30)

use a `for`-loop to retrieve every three-gram from `textfile.words` and store each in the relevant slot of `all.3grams1` (32–39)

create a sorted frequency list of `all.3grams1`; call it `sorted.ngram.table` (41–43)

compute `sorted.ngram.table`'s deciles to see if the frequency distribution is (very) skewed/Zipfian or not (45–48)

visualize the sorted frequency table (50–55)

**Which aspects of the script are worth additional comment?** Not much really, because we've dealt with pretty much all of it in the previous section on character *n*-grams. However, if you look at lines 59–71 of the code file, you can see an alternative way of generating the word three-grams without a `for`-loop, but this time the solution is quite complicated and involves several nested steps:

- We are again using `mapply`, but this time we do it to generate a matrix with 2,950 columns (one for each three-gram to create) and three rows (with positions of the three words in each three-gram for subsetting. Here are the first six columns of this matrix:

```
·····[,1]·[,2]·[,3]·[,4]·[,5]·[,6]
[1,]····1····2····3····4····5····6
[2,]····2····3····4····5····6····7
[3,]····3····4····5····6····7····8
```

- Then we use the function `apply`, which takes three arguments: First, a two-dimensional data structure such as the above matrix; second, a 1 or a 2 depending on whether you want to do something to each row (1) or column (2) of that matrix; third, a function that says what you want to do with these row or column elements. In this case, we want to process the matrix columnwise because the positions of the words of each three-gram are in the columns.
- The function we pass on to `apply` is an application of `paste` with `collapse="·"`: We want each vector of numbers from each column to be used in subsetting from `textfile.words` so that we can paste those three words together with spaces between them.

This is a pretty complicated application and it doesn't matter if you do not fully understand it right away: Some of the `apply` functions are difficult at first – revisit this once you have more practice. However, do note that lines 75–90 provide you with a function `word.ngrams` that uses that `apply(mapply(...))` approach and that you can use whenever you need to create *n*-grams from a character vector each element of which is a word.

### *5.2.3 Zero-Derivation of* Run *and* Walk *in the BNC*

In this example we look at how the word forms *run*, *runs*, *walk*, and *walks* are used in the BNC. Specifically, we explore whether the two lemmas differ with regard to how frequently they are used as nouns or as verbs. We will do so in the BNC but, to add a slight twist to things, we will write our script such that it prompts the user to interactively choose what annotation the files to search have (XML or SGML), and then it proceeds with the required search expressions; (2) and (3) remind you of the form of the two annotation formats, XML and SGML, respectively:

```
(2) a   <w·c5="NN1"·hw="walk"·pos="SUBST">walk·</w>
    b   <w·c5="VVG"·hw="walk"·pos="VERB">walking·</w>
(3) a   <w·NN2>runs
    b   <w·VVG>running
```

The desired output of this script <05_05_run-walk.r> is a data frame `result` like this (shown for XML results), which lists for every occurrence of *run*, *runs*, *walk*, and *walks*:

- a coarse-grained tag (the pos-value in the XML annotation and the first letter of the tag in the SGML annotation) and a fine-grained tag (the c5 tag in the XML annotation and the full tag in the SGML annotation);
- the form that was found and the lemma of the form that was found:

```
>·head(result)¶
··TAGSCOARSE·TAGSFINE·FORMS·LEMMAS
1·····subst·····nn1···run····run
2·····verb·····vvn···run····run
3·····verb·····vvn···run····run
4·····verb·····vvn···run····run
5·····verb·····vvb···run····run
6·····verb·····vvn···run····run
>·tail(result)¶
·····TAGSCOARSE·TAGSFINE·FORMS·LEMMAS
38565·····verb·····vvi··walk···walk
38566·····verb·····vvz·walks···walk
38567·····verb··vvz-nn2·walks···walk
38568·····verb·····vvi··walk···walk
38569·····verb·····vvi··walk···walk
38570·····verb·····vvi··walk···walk
```

**What are the things we will need to do?**

- We need to define a vector `corpus.files` that contains the paths to all corpus files, which should all be in one directory (potentially with sub-directories).
- We will let the user make a choice of which kind of corpus annotation the script will deal with, and depending on what the user chooses, the script will define all search expressions accordingly. Also, the script generates empty vectors to collect search results for each verb.

- We will load each corpus file and use our search expressions to find the four forms in question with the right tags and store all their matches.
- We will process the matches so as to retrieve the coarse-grained and the fine-grained tags as well as the lemmas of the matches and save all of the results in a data frame `result`.
- Finally, we will split up that data frame by the lemma and determine whether, within each lemma, there is a significant correlation between the two forms and whether they are used as nouns or verbs: We will perform chi-squared tests and create some plots.

**What are the functions we will need for that?** Obviously, we need to be able to define the corpus files we want to search, which means we use `rchoose.dir` to define the directory containing the corpus files and `dir` to retrieve all the file names from that directory. We will need the functions `switch` and `menu` (which you do not know yet so you may want to briefly look at their help pages – they are not difficult and the script will show you how they are used anyway) to prompt the user to choose the annotation format that will be processed, and we need a conditional with `if` to then define regular expressions for either choice. We will need to define empty character vectors to collect results using `character`, and we will need to use a `for`-loop to load each corpus file; during each iteration, we will use `grep` to find all corpus sentences and then use `exact.matches.2` to find all matches of *run*, *runs*, *walk*, and *walks*.

After the loop, we will again use `exact.matches.2` to retrieve the tags from the matches, we will use `gsub` to retrieve the forms only (i.e., discard all annotation), and we will use `substr` to define the lemma of each match (by retrieving the first letter, which gives us enough information to distinguish *run* and *walk*) and using `ifelse` to define a lemma column). All those results will then be put into the data frame `result` using `data.frame`.[1] That data frame will then be split up using `split` and cleaned using `droplevels`, and then we use `chisq.test`, `mosaicplot`, and `assocplot` to explore the distributions of nouns and verbs for all inflectional forms of each lemma; in that process, I will also introduce you to the functions `with`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_05_run-walk.r>; read this side by side with the actual script in RStudio:

clear memory and define and/or load all relevant packages and/or functions (1–3)

define the vector with corpus files to search: `corpus.files` (5–6)

have the user define a logical vector `xml.version` by choosing from a menu with two options which annotation the corpus files have, XML or SGML (8–16)

define the search expressions conditional on which annotation will be processed; the search expressions must cover the retrieval of the overall form with its tag, how to retrieve just the tags, and just the forms (18–30)

define empty character vectors to collect matches: `all.runs` and `all.walks` (32–33)

`for`-loop (counter = `i`, over `corpus.files`): (35, -61)

- output a progress report: the name of the file currently being processed (36) and load that corpus file `i` (37)
- discard the header/find the sentences and switch them to lower case (39–44)
- find and store in `all.runs` the matches for *run* and *runs* when tagged as a verb or a noun (46–53)

find and store in `all.walks` the matches for *walk* and *walks* when tagged as a verb or a noun (54–60)

retrieve from `all.runs` and `all.walks` all coarse-grained tags with `exact.matches.2` (63–70)

retrieve from `all.runs` and `all.walks` all fine-grained tags with `exact.matches.2` (71–78)

retrieve from `all.runs` and `all.walks` all forms by `gsub`bing away everything else (79–88)

combine all results in a data frame `result`; while doing that, retrieve the first letter from the forms to decide on the lemma; summarize the result (90–99)

create a list `results` by `split`ting up `result` by its lemma column (101–103)

clean away unused levels from all data frames in `results` by `lapply`ing `droplevels` (105–107)

cross-tabulate forms and coarse tags for "runs?", compute a chi-squared test and Cramer's *V*, and visualize the results (111–128)

cross-tabulate forms and coarse tags for "walks?", compute a chi-squared test and Cramer's *V*, and visualize the results (132–149)

**Which aspects of the script are worth additional comment?** It is maybe useful to briefly comment on how the user is prompted to choose the annotation format: The function `menu` takes as its first argument a character vector of options to choose from and can be further customized with `graphics` (do you want a window pop-up or a prompt in the console?) and `title`; it returns the number of the option the user chose, i.e., in our case 1 or 2. In the code here, this 1 or 2 is then the input to `switch`, which takes that number and picks the corresponding item from its arguments. Thus, if the user chose "XML" then `menu` made that a 1 and `switch` then assigns `TRUE` to `xml.version`.

A second aspect worth mentioning is the definition of the search expressions; here's one example:

```
>·search.expression.run·<-"<w·c5=\\\"[vn][^>]+>runs?(?=·?<)"¶
```

This looks for the beginning of an XML tag, with the c5 tag beginning either with "v" or with "n", followed by one or more characters that are not already the closing angular bracket of that tag till you reach that closing angular bracket, followed by "run" or "runs" but only if, looking to the right, you can see a space (optional) and the next opening angular bracket. Okay, but why are there three backslashes? To understand this, it's useful to compare the difference between how R represents those strings as a vector in the console and how they are actually printed as a string:

```
>·search.expression.run¶
[1]·"<w·c5=\\\"[vn][^>]+>runs?(?=·?<)"
>·cat(search.expression.run)¶
```

```
<w·c5=\"[vn][^>]+>runs?(?=·?<)
>·nchar(search.expression.run)¶
[1]·30
```

That is, the first line is how we defined it, but when R, so to speak, 'uses' it then the third backslash is one that makes R recognize the " as not being the " that closes the opening " of the search expression, and the first backslash is the one that marks the second one as an actual backslash, not a backslash of a special character. A little confusing, but if you ever struggle with this – as I admittedly do all the time myself – it helps to just `cat` the expression to the screen to see what you have really defined.

Finally, a brief note on the function `with` as used in the following line:

```
>·(results.run.coarse<-with(results$run,·table(TAGSCOARSE,·FORMS)))¶
```

The purpose of using `with` as above is that R does not know the two objects that are the arguments of `table`: If you tried to retrieve `TAGSCOARSE`, you will get an error message:

```
>·TAGSCOARSE¶
Error:·object·'TAGSCOARSE'·not·found
```

Thus, recall from Section 3.4.3 that `with` tells R where `TAGSCOURSE` and `FORMS` can be found, namely in the data frame `results`.

The rest of the script should be more or less self-explanatory but why don't you, as a final task, try to take the two main search expressions, the ones used in the loop in lines 50 and 57, and devise a new one that finds all four forms at the same time? The solution to this little exercise is in the code file with free-spacing in lines 157–165. Also, why don't you think about (for later) how this script could be changed so as to avoid growing results vectors in the loop.

### *5.2.4 Word and Sentence Lengths in the BNC*

In this small case study, we will go over all files in the BNC and determine for each sentence how long it is (in words) and how long the words in it are (in characters) so we can generate a plot that checks whether there is a correlation between the two.

**What are the things we will need to do?**

- We need to define a vector `corpus.files` that contains the paths to all corpus files, which should all be in one directory (potentially with sub-directories).
- We need to generate two lists to store for each file (1) the lengths of all sentences in it and (2) the lengths of all words in it. Thus, each of these lists needs to have as many empty elements as there are files in `corpus.files`; let's call those lists `all.sent.lengths` and `all.word.lengths`.
- We will load each corpus file, discard the header and keep only the sentences.

- To determine sentence lengths, we count all the word tags we find in each sentence; we save the sentence lengths for a file in the relevant element of `all.sent.lengths` for this file.
- To determine word lengths, we first use the word tags to find words, but then delete the tags and count the number of characters of what's left; we save the word lengths for a file in the relevant element of `all.word.lengths` for this file.
- After the loop, we compute the median sentence and word lengths for each file (very crude approach!) and summarize the distribution of these medians.
- Finally, we generate a scatterplot with word lengths on the *x*-axis and (logged) sentence lengths on the *y*-axis and try to determine whether there's a correlation between the two.

**What are the functions we will need for that?** As usual, we use `rchoose.dir` to define the directory containing the corpus files and `dir` to retrieve all the file names from that directory. We define two lists with as many empty elements as there are files using the function `vector` with `mode="list"`. We then use the usual `for`-loop to load each corpus file during each iteration and use `grep` to find all corpus sentences. We then use `exact.matches.2` to find all word tags but only retain the second component of `exact.matches.2`, namely the locations of the matches, i.e., the numbers of the sentences they occur in (because we can tabulate those with `table`). After that, we use `exact.matches.2` again to find all words and their tags as well as `gsub` to delete the tags and `nchar` to determine the numbers of characters of what's left. At the end of each iteration of our `for`-loop, we save all sentence lengths of a file and all word lengths of a file into the relevant components of `all.sent.lengths` and `all.word.lengths`.

After the loop, we use `sapply` together with `median` and `summary` to compute the median sentence lengths and median word lengths per file and use `plot` to create a scatterplot as defined above; to avoid overplotting of many points onto the same coordinates, we use `jitter` and `rgb` (for grayscale and transparency effects) just as we did in Section 4.3.3 when we discussed correlations as well as `log` to log the sentence lengths. Finally, we use `abline` and `text` to add some annotation into the plot; we use `lines(lowess(...))` to add a trendline, and I introduce `mtext` to add colored axis labels.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_06_wrd-sent-lengths.r>; read this side by side with the actual script in RStudio:

clear memory and define and/or load all relevant packages and/or functions (1–3)

define the corpus files to search: `corpus.files` (5–6)

define empty lists to collect matches: `all.sent.lengths` and `all.word.lengths` (8–9)

`for`-loop (counter = `i`, over `seq(corpus.files)`): (11, -45)

  output a progress report: the name of the file currently being processed and the percentage of files covered after the current iteration (12–17)

  load the `i`-th corpus file (19)

  discard the header / find the sentences (21–25)

  find the numbers of word tags for each sentence and store them all in `all.sent.lengths[[i]]` (27–31)

*(continued)*

*(continued)*

find the lengths of all words (identified by their tags (33–36), but with the tags deleted by `gsub`) and store them all in `all.word.lengths` (38–43)

use `sapply` to compute `median` sentence and words lengths for all files and store the 4,049 medians in `average.sent.lengths` and `average.word.lengths` (47–49)

use `plot` and other visualization functions to create the scatterplot described above (51–88)

**Which aspects of the script are worth additional comment?** This script has no particularly complicated components – the only thing I recommend is that you study the code for the plot and all its commentary in great detail because it involves many useful functions.

### *5.2.5 Approximating Syntactic Complexity: Fichtner's* C

In the context of an applied linguistics study, you want to compute for every file in the BNC World Edition a measure of syntactic complexity. A particularly easy one to compute uses information that our last case study made us collect: Fichtner's *C*, which is computed as shown in (4):

$$\text{(4)}\quad \text{Fichtner's } C = \frac{\text{number of verbs}}{\text{number of sentences}} \times \frac{\text{number of words}}{\text{number of sentences}}$$

**What are the things we will need to do?**

- We need to define a vector `corpus.files` that contains the paths to all corpus files, which should all be in one directory (potentially with sub-directories): thus, as usual, we need `rchoose.dir` and `dir`.
- We need to generate three vectors to store for each file (1) the number of verbs in it; (2) the number of words in it; and (3) the number of sentences in it. Thus, each of these vectors needs to have as many empty elements as there are files in `corpus.files`. Let's call those vectors `nos.of.verbs`, `nos.of.words`, and `nos.of.sentences`.
- We will load each corpus file, discard the header and keep only the sentences.
- To determine the number of sentences, we just determine the length of the vector remaining after the previous step, which was precisely aiming at only keeping sentences; we save the number of sentences in the relevant element of `nos.of.sentences`.
- To determine the number of words, we count how many closing word tags ("</w>") we find in the sentences; we save that number into the relevant element of `nos.of.words`.
- To determine the number of verbs, we count how many instances of the pos-tag "VERB" we find in the sentences; we save the number of verbs in the relevant element of `nos.of.verbs`.
- After the loop, we compute Fichtner's *C* values for all files as defined above in (4) and, just for housekeeping, give the values the names of the corpus files; also, we compile all the results in a data frame `result` that looks like this:

```
>·head(result)¶
·····SENTENCES·VERBS·WORDS·VERBSperSENT·WORDSperSENT·FichtnersC
A00.xml·····423··1143··6708·····2.7·····15.9···42.85076
A01.xml·····588··1569··7898·····2.7·····13.4···35.84144
A02.xml·····223···564··3376·····2.5·····15.1···38.28881
A03.xml····1051··3277·19433·····3.1·····18.5···57.65153
A04.xml····1621··6294·39163·····3.9·····24.2···93.80731
A05.xml····1706··7733·41363·····4.5·····24.2··109.90109
```

- Finally, we explore the data visually by (1) generating histograms of the first five columns of that above data frame as well as their logs and (2) scatterplots of the logs of Fichtner's *C* as a function of the first five columns (with trendlines).

**What are the functions we will need for that?** Obviously, we need `rchoose.dir` to define the directory containing the corpus files and `dir` to retrieve all the file names from that directory. We will need to define three vectors to collect numeric results using `integer`, and we will need to use a `for`-loop to load each corpus file. During each iteration, we will use `grep` to find all corpus sentences. We then use `length` to find the number of sentences, and use `length` applied to searches for closing word tags and verb tags using `exact.matches.2`. to find the numbers of words and verbs.

After the loop, we use simple arithmetic to compute Fichtner's *C* values and `data.frame` to compile the results all into one data frame. We then use `par(mfrow=c(rows,·columns))` to define the number of rows and columns we want to have in the plotting windows and then use `apply` (see above) and `hist` (and `log`) to create histograms of the data in the first five columns of `result`. Finally, we use an alternative to `par`, `layout`, to divide up the plotting window into five regions and then plot five scatterplots into them with `plot`, `log`, `lines(lowess(...))`, and `abline`; we also do a correlation test to see how strongly Fichtner's *C* is related to the number of verbs/sentence and the number of words/sentence.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_07a_Fichtners-C.r>; read this side by side with the actual script in RStudio:

clear memory and define and/or load all relevant packages and/or functions (1–3)

define the vector of corpus files to search: `corpus.files` (5–6)

define empty `integer` vectors to collect results: `nos.of.verbs`, `nos.of.words`, and `nos.of.sentences` (8–9)

`for`-loop (counter = `i`, over `seq(corpus.files)`): (11, –35)

- output a progress report: the percentage of files covered after the current iteration (12–14)
- load the `i`-th corpus file (16)
- discard the header/find the sentences (18–21)

*(continued)*

*(continued)*

find the numbers of sentences and store it in `nos.of.sentences[i]` (23)

find the numbers of closing word tags and store it in `nos.of.words[i]` (25–28)

find the numbers of verb tags and store it in `nos.of.verbs[i]` (30–34)

compute Fichtner's *C* for all files (37–39)

compile the results into one data frame (41–49), summarize each column (50), and save the result (52)

set up a plotting window with two rows and five columns (line 57) and use `apply` to plot `hist`ograms of the raw and the `log`ged values of all variables but Fichtner's *C* (add the argument `breaks="FD"` to `hist` for more fine-grained bins in the histrograms (lines 58–69), then reset the plotting window to its default (line 70)

(learn about the `layout` function and how it defines panels in the plotting window (74))

set up a plotting window for five plots (line 76) and plot scatterplots of Fichtner's *C* against every other variable in isolation (77–83, 84–90, 91–97, 98–104, 105–111), then reset the plotting window (112)

compute correlation tests for Fichtner's *C* against the numbers of words/sentence and verbs/sentence (114–115)

**Which aspects of the script are worth additional comment?** This script also has no particularly complicated components but let me again recommend that you study the code for the plots in detail, in particular (1) how `apply` is used to do something (namely `hist`) to the first five columns of `result` (check Section 5.2.2 again), and (2) how `layout` works: It takes as input a matrix whose numbering shows where different plots go: equal numbers other than 0 cover space for one panel, and 0s reserve space that R doesn't plot into.

As you probably noticed, there is a second case study script file on Fichtner's *C*, namely <_qclwr2/_scripts/05_07b_Fichtners-C_XML.r>. This script does the exact same thing as the one just discussed and most of the code is in fact identical, but it does it not by essentially treating the corpus file as a simple flat sequence of character strings as we did above, but by utilizing the hierarchical XML annotation in ways discussed in Section 3.8.2. Thus, most of the things discussed above do not change – the only changes occur in the loop. Remember that the most powerful package to handle XML data in R is XML, but also remember that it has a bad memory leak when used on Windows systems. The script therefore does the following within the loop: It checks whether it is currently running on a Windows computer or not (lines 18, 31, -42) and then,

- if the script detects it is running on a Windows computer, it uses functions from the xml2 package to get the job done – it loads the corpus file with `read_xml` and retrieves the frequencies of elements using `xml_find_all` (lines 19–30);
- if the script detects it is not running on a Windows computer, it uses functions from the XML package – it loads the corpus file with `xmlInternalTreeParse` and retrieves the frequencies of elements using either the summary of the parsed XML object or using `xpathApply` with `xmlGetAttr` (lines 32–41).

*Table 5.1* The frequency of *w* (="perl") and all other words in two 'corpus files'

| | *Target corpus* | *Reference corpus* | *Totals* |
|---|---|---|---|
| Word *w* from target corpus | ***a* = 249** | ***b* = 8** | ***a* + *b* = 257** |
| All other words from target corpus | *c* = 5,816 | *d* = 5,588 | 11,404 |
| Totals | **6,065** | **5,596** | 11,661 |

Thus, check out how you can query your R instance with regard to what system it's running on (line 18) and, more importantly, how I use both of these XML processing packages to get the job done.

### *5.2.6 Key Words*

In this section, we will discuss a script that computes different measures of keyness for words occurring in a target corpus by comparing their frequencies in that target corpus to that of a reference corpus and quantifying the relative attraction of each word to the target corpus by means of a keyness statistic. The most frequently used keyness statistics are based on a 2 × 2 occurrence table such as the one shown in Table 5.1, which represents a case where the target corpus of interest has 6,065 words, the reference corpus to which the target corpus is compared has 5,596 words, and the word *w* of interest is attested 257 times in both corpora: 249 times in the target corpus and 8 times in the reference corpus.

The most widely used statistic is the log-likelihood ratio $G^2$ (see Dunning 1993) and it is computed as shown in (5), where *ln* refers to the natural log (R's default setting for `log`), *obs* refers to the observed frequencies in cells *a*–*d* (i.e., those shown in Table 5.1), and *exp* refers to the expected frequencies as one would obtain them from a chi-squared test (see the discussion in Section 4.2.2); according to the logic discussed there, the expected frequency of cell *a* – i.e., the frequency with which word *w* should occur in the target corpus if its occurrences in the two corpora were random – is $^{257 \times 6{,}065}/_{11{,}661} = 133.6682$.

$$(5) \quad G^2 = 2\Sigma_{i=1}^{4} obs \times \ln \frac{obs}{exp}$$

Note that this computation can run into problems: logs of 0 return –Inf (negative infinity) in R, which then makes the result of the whole equation –Inf. This is commonly handled by changing the relevant summand from –Inf to 0.

Another useful measure one might use is the *difference coefficient* used by Leech and Fallon (1992), which is computed as in (6) and yields $^{241}/_{257} = 0.9377$ when applied to Table 5.1.

$$(6) \quad difference\,coefficient = \frac{a-b}{a+b}$$

Finally, one can use Damerau's relative frequency ratio *rfr* (Damerau 1993; Manning and Schütze 2000: 175f.), which is computed as in (7) and yields $^{(249 \div 8)}/_{(6{,}065 \div 5{,}596)} = 28.7181$ when applied to Table 5.1:

$$(7) \quad rfr = \frac{a \div b}{(a+c) \div (b+d)}$$

In this section we will compare an older Wikipedia entry on the programming language Perl (the target corpus) to an older Wikipedia entry on the programming language Python (the reference corpus) and generate a data frame `results` that, in the rows, lists all the words from the target corpus and, in its columns, contains

- the four observed cell frequencies *a*, *b*, *c*, and *d*;
- the expected cell frequencies for cells *a* and *b*;
- the three keyness statistics discussed above; and
- a column that states which of the two corpora the word is preferred in, or key for.

**What are the things we will need to do?**

- We will define a function we will call `log.0`, which returns the log of a number it is given (just like default `log` does), but that returns 0 when the number it is given is 0.
- We define the paths for the two corpus files we will use (<_qclwr2/_inputfiles/corp_perl.txt> and <_qclwr2/_inputfiles/corp_python.txt>).
- We define two vectors: one that will collect all word tokens that are attested in both corpora (let's call it `all.word.tokens`) and one that states for every occurrence of every word token which corpus it's from (let's call it `all.sources`) (recall this kind of approach from Exercise box 3.1).
- We then load each corpus file, split it up into words, remove any empty character strings that might result from that, replace any numbers by the placeholder "_NUM_", store the word tokens in `all.word.tokens`, and note in `all.sources` the corpus file of each token.
- Then we cross-tabulate `all.word.tokens` and `all.sources` to obtain a table that lists for every word type how often it occurs in each corpus; from those, we compute the three keyness statistics for each word type.
- We collect all results in the data frame `results` described above.
- We visualize the results in a plot that plots the words at the coordinates of the (logged?) frequencies of the word types on the *x*-axis and the difference coefficients on the *y*-axis.

**What are the functions we will need for that?** We need `function`, `ifelse`, and `log` to define our function `log.0` and, of course, `rchoose.files` as well as `vector` to choose the corpus files and define the vectors in which we collect word tokens and corpus names respectively. In a `for`-loop, we then use `scan` to load the corpus files as well as `strsplit` and `unlist` to identify word tokens and `nzchar` to discard empty character strings; also we use `grepl` and subsetting to identify and replace all numbers; finally (still within the loop), we use `rep` and `basename` to fill `all.sources`.

After the loop, we use `table` to cross-tabulate `all.word.tokens` and `all.sources`, before we use `length` as well as mathematical functions such as `sum`, `rowSums`, and `colSums`, and our `log.0` to compute our keyness statistics. Finally, we use `data.frame` to create `results` and `plot` with `log` as well as `text` to visualize the results.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_08_keywords.r>; read this side by side with the actual script in RStudio:

clear memory and load all relevant packages (1–2)

define the function `log.0` so that it can take a number to log and a user-defined base of the log (4–8)

define the corpus files to search: `text.files` (10–11)

define empty vectors to collect results: `all.word.tokens` and `all.sources` (13)

`for`-loop (counter = `i`, over `text.files`): (15, -36)

- load the corpus/text file `i` (16)
- split it up into words at the occurrence of 1+ non-word characters and `unlist` (18–22)
- remove empty character strings from the resulting vector (23)
- replace all 'words' that only consist of numbers by "_NUM_" (24–28)
- add to `all.word.tokens` the current word tokens (29–30)
- add to `all.sources` the name of the current corpus file as often as that file has words (32–35)

to compute all keyness statistics, create a word-types-by-corpus table (`word.by.corp.table`) and inspect its first 20 rows (40–41)

use the columns of that table (note that they already contain the observed frequencies of the cells *a* and *b*) and its sums to compute four vectors that provide for each word type the frequencies shown in Table 5.1, call them `obs.as`, `obs.bs`, `obs.cs`, and `obs.ds` (43–47)

using the logic from Section 4.2.2 of how expected frequencies are computed, create four analogous vectors `exp.as`, `exp.bs`, `exp.cs`, and `exp.ds` (49–52)

then compute the relevant keyness statistics from those vectors: $G^2$ (lines 54–60, use `log.0`), difference coefficients (62–63), and *rfr* (65–66)

compile all the results into the data frame `results` (68–77), reorder its rows by which corpus each word prefers and then by the difference coefficients (78–80), and save it (81–83)

visualize the results in a scatterplot either simply (87–90, see `?rug¶`) or more nicely (92–102)

**Which aspects of the script are worth additional comment?** Most aspects of this script should be straightforward, given that everything happening in the loop is relatively simple text and data processing. In my experience, the only difficult part here may be the generation of the observed frequencies and the subsequent computations. For that you have to appreciate that R uses vectors even if they are very long. For instance, here's a part of `word.by.corp.table`:

```
>·word.by.corp.table[1542:1547,]¶
···············all.sources
all.word.tokens·corp_perl.txt·corp_python.txt
·······performs·····------1·------------0
·······perhaps·····------1·------------0
·······period·····------1·------------0
·······perl·····------249·------------8
·······perl1·····------1·------------0
·······perl5·····------1·------------0
```

You can see the frequencies of "perl" in both corpora, which are the numbers you know from Table 5.1. Since each column of `word.by.corp.table` covers all word types attested in that corpus, you can infer that the cells *c* and *d* of Table 5.1 are the sum totals of each column (i.e., all *a*s and all *b*s) minus the *a* and *b* for "perl", i.e., 249 and 8. And once you remember that in R you can do vector1 – vector2 even if the two vectors are differently long (the shorter element will be reused as often as necessary), you can understand how lines 46–47 do what they do (because `sum(obs.as)` gets reused as many times as there are word types), and lines 49–52 work in the same way. In fact, this is one aspect of R that is very elegant: You can perform thousands of additions, subtractions, etc. with just one line, where in other programming languages you might have to write a `for`-loop for that.

Finally, as before, do spend some time on understanding the code that creates the plot. The threshold value for significant results that you see highlighted in the plot is the minimum chi-squared value that is significant at one degree of freedom (i.e., for a 2 × 2 table), as you could check with `qchisq(0.05,·1,·lower.tail=FALSE)¶`, but for the script per se, that's not important. Also note how the labeling of the *x*-axis is done: As *x*-coordinates of each word, I am using their logs (so as to stretch the display), but I am then not labeling the *x*-axis tickmarks with their logs but with their antilogged values so that a reader doesn't have to make that computation himself; thus, perl is plotted at the *x*-coordinate of log(257) ≈ 5.55, i.e., right between *x* = 5 and *x* = 6, but the tickmarks at 5 and 6 are labeled with the actual unlogged frequencies they correspond to: 148.4 and 403.4 respectively.

### *5.2.7 Frequencies of* -ic *and* -ical *Adjectives*

In this case study, we are studying English adjectives ending in *-ic* and *-ical* on the basis of an already stored frequency list of the BNC Version 1; this frequency list was compiled by Adam Kilgarriff and is provided here in the file <_qclwr2/_inputfiles/corp_bnc_sgml_freql.txt>, which looks like this:

```
>·head(freqs)¶
··FREQUENCY···WORD·POS·FILES
1·········1···!*?*·unc·····1
2·······602·····%·nn0···113
3·········1··%/100·unc·····1
4·········3··%/day·unc·····1
5·········1···%295·unc·····1
6·········1·%5,000·unc·····1
```

Specifically, we want to do two things: First, we want to generate a table that states for every adjective that is attested at least once with either *-ic* (e.g., *public*) or *-ical* (e.g., *physical*) – or of course with both suffixes (e.g., *electric(al)*) – how often it is attested with either suffix (both in absolute and relative frequencies. This is supposed to look like this:

```
>·head(result,·4)¶
···············their.suffixes
both.adjectives····ic··ical
```

```
·····politic·····80·30366
·····public···30384·····0
·····economic·23484···498
·····specific·11313·····0
>·head(result.perc,·4)¶
·············their.suffixes
both.adjectives·····ic···ical
·····politic··0.0026·0.9974
·····public···1.0000·0.0000
·····economic·0.9792·0.0208
·····specific·1.0000·0.0000
```

Second, we want to explore whether there is a correlation between the overall frequency of an adjective stem (with either suffix) and the percentage of these adjectives ending in *-ical*: Is it the case that *-ical* adjectives are more frequent (as has been claimed by Marchand 1969: 242)?

**What are the things we will need to do?**

- We load the BNC frequency list file into a data frame called `freqs`.
- So as to not have to deal with such a large date structure all the time (nearly 940,000 rows), we then extract from it two data frames: one with all rows that contain adjectives ending in "ic" (`freqs.ic.adj`), and one with all rows that contain adjectives ending in "ical" (`freqs.ical.adj`). Note that we are interested in being memory-efficient: We will therefore first trim down all of `freqs` to a data frame that just contains adjectives (`freqs.adj`); we then remove `freqs` from memory and look for *-ic* and *-ical* adjectives, but only in `freqs.adj`.
- Noting that some adjectives may be represented in that file with more than one tag (remember the existence of portmanteau tags, check out *physical* for instance), we then make sure we sum up the frequencies of all tags for each adjective in either data frame, which will give us one separate vector with these frequencies for each suffix. Let's call those `ic.adjectives.freqs` and `ical.adjectives.freqs` (for mild amusement, check out `freqs.ical.adj` for spelling mistakes, e.g., the different spelling of *hierarchical* attested in the file).
- Next, we need to, so to speak, join those vectors such that we can see for each adjective stem how often it occurs with *-ic* and how often with *-ical*. To that end, we apply the logic of comparing frequency lists (from the first exercise box on vectors) and from the previous case study: First, we create a vector with all *-ic* adjective tokens (called `ic.adjectives`) and another one with all *-ical* adjective tokens (called `ical.adjectives`). Second, we create a vector `their.suffixes` that lists "ic" and "ical" as many times as there are *-ic* and *-ical* adjectives (which we know from the lengths of `ic.adjectives` and `ical.adjectives` respectively). Third, we combine `ic.adjectives` with a version of `ical.adjectives` from which we deleted "al" into a vector called `both.adjectives`. As a result, we have one vector with only *-ic* tokens (`both.adjectives`), but another one that says which suffix each token was attested with originally (`their.suffixes`), which we can then tabulate for both raw frequencies and percentages.
- Finally, we can then `plot` the row sums of the frequency table `result` (the overall frequency of an adjective stem) against the second column of `results.perc` (the relative frequency of the *-ical* form) to see whether there is a correlation between

the two, and we can explore whether the *-ic* adjectives differ in frequency from that of *-ical* adjectives. Let me recommend here to do that once for all the data and once for adjectives that occur with a certain minimum frequency (such as three) – you will be amazed at the difference in results!

**What are the functions we will need for that?** We need `read.table` and `rchoose.files` to load the frequency list file, and we need `grepl`, subsetting, and `droplevels` to extract the sub-data frames with *-ic* and with *-ical* adjectives from it; using `droplevels` is important: It makes sure that all the adjectives that are not *-ic* and *-ical* adjectives don't "stick around" as factor levels and in frequency tables! Next, we need `tapply` and `sum` to merge the frequencies of adjectives that are attested with more than one tag; after that we use `rep` and `c` to merge the vectors and `gsub` to reduce *-ical* adjectives to *-ic* forms. Finally, we need `table` to compute raw adjective frequencies, `prop.table` to change that table into one of percentages, and `order` to re-order the rows of that table by the combined adjective stem frequencies.

For the statistical and visual exploration of the correlation between stem frequency and the percentage of *-ical*, we will need `plot`, `log`, and `rowSums` as well as `lines(lowess(...))` again. For the exploration of potential differences between *-ic* and *-ical* adjective frequencies, we can use `boxplot` and `wilcox.test` as well as `plot(ecdf(...))`. To create a downsized version of the frequency table, use `apply` and `min` with subsetting.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_09_ic-ical.r>; read this side by side with the actual script in RStudio:

clear memory (1) and the package rChoiceDialogs (2)

load Kilgarriff's frequency list file and check whether the loading worked (3–9)

create `freqs.adj` by subsetting the rows of `freqs` that have "aj" in the tag column (11–15)

remove the original large data frame to free up memory (using `rm`, 17)

create `freqs.ic.adj` and `freqs.ical.adj` by subsetting the rows of `freqs.adj` that end with "ic" or "ical" in the word column (19–24, 25–29) and check the results (31–32)

use `tapply` and `sum` to add up all frequencies of each adjective type regardless of how many tags it is observed with (34–39, 40–43)

from these frequencies, create vectors with `rep` with all *-ic* adjective tokens (`ic.adjectives`) and all *-ical* adjective tokens (`ical.adjectives`) (49–52)

create `both.adjectives` by merging `ic.adjectives` and a version of `ical.adjectives` with string-final "al" removed (with `gsub`) (54–57)

from `ic.adjectives` and `ical.adjectives`, use `rep` to create a vector with all suffixes of all adjectives (`their.suffixes`) (58–59)

create a `table` with the adjective stems in the rows and the suffixes in the columns; also, convert that into a table of percentages, and then use `order` to order that table in descending order of adjective frequency (61–66)

visualize the correlation of frequency and *-ical* percentage with a scatterplot and a trend line (70–81)

generate a boxplot (83–87) and compute a Wilcoxon-test (88–91) to compare *-ic* and *-ical* frequencies

generate an ecdf plot (93–105) and compute a Kolmogorov-Smirnov-test (106–107) to compare *-ic* and *-ical* frequencies

create a version of the table that requires a minimum frequency threshold of three for any adjective form by checking whether the smallest number per row exceeds three and use the resulting TRUEs and FALSEs for subsetting rows (112–123)

re-run all the above statistics on this smaller table (125–161)

**Which aspects of the script are worth additional comment?** Most of the script should be clear by now, but I want to briefly revisit the trimming down of the frequency table, which is done in the following lines:

```
>·threshold.to.be.met<-3¶
result.trimmed<-result[apply(result,·1,·min)>=threshold.to.be.met,]
```

The `apply` part goes through the frequency table `result` row-by-row and returns the `min`imum number per row, i.e., per adjective stem. The vector with these numbers is then used in a logical expression to return for each of these minimum numbers whether it is greater than or equal to three, which returns a vector of TRUEs and FALSEs. That logical vector is then used to subset only the rows of `result` (note, it's before the comma) where that condition evaluated to TRUE, i.e., the rows with large enough frequencies.

As before, go through the code for the plots: they don't introduce much that is new – check `?legend¶` though – but serve to entrench what you have learned so far. For the statistical tests, look at the statistics chapter again and maybe at Gries (2013b) for more details; for more on *-ic* and *-ical* adjectives, check out Gries (2001; 2003).

### *5.2.8 Frequencies of All Word-Tag Combinations in the BNC*

Now we're getting serious. The next script does something seemingly elementary – we will create a frequency list of word-tag combinations (so as to be able to distinguish *run* as a noun from *run* as a verb) – but we will do it on a relatively large data set, the complete BNC World Edition with XML annotation, and we will use the annotation well by utilizing information provided by the BNC's multi-word tags, i.e., tags that mark multi-word units such as *because of*, *in spite of*, *on behalf of*, etc. That is, this is output we want:

```
>·result.df[533:537,]¶
····TAGS·····FORMS·FREQS
533··nn1·department·17464
534··nn0·····data·17463
535··prp·because·of·17440
536··nn1·····town·17424
537··nn2····minutes·17410
```

```
<mw·c5="CJS">
      <w·c5="AV0"·hw="even"·pos="ADV">even·</w>
      <w·c5="CJS"·hw="when"·pos="CONJ">when·</w>
</mw>
```

*Figure 5.2* The annotation of *even when* as a multi-word unit.

To make this happen, you need to know what the multi-word tags look like; Figure 5.2 shows an example (broken up into multiple lines for expository reasons only).

Crucially, what that means is that you need to be careful to not count things twice: Once *even when* as a subordinating conjunction (CJS) has been counted, you must not count *even* as an adverb (AV0) and *when* as another subordinating conjunction again – otherwise you imply that Figure 5.2 does not contain one, but three words.

Also, we will create and use a function called `whitespace`, which can clean excess whitespace – leading and trailing spaces, sequences of more than one space – as well as empty character strings. Finally, and this is maybe the biggest new thing here: Given the size of the data and R's memory handling, we will most likely not be able to have two vectors – one for about 100 million words and multi-word units, one for all their tags – just grow dynamically in memory. Your computer session might not survive that and either take forever or even crash (each of my two computers has 16 GB of RAM and I stopped the scripts because they were taking forever). Therefore, we will create a sub-directory into which we will save interim output after each corpus file, and after having gone through all 4,049 corpus files we will then merge the 4049 output files into the one data structure we want, `results.df`.

**What are the things we will need to do?**

- We set the warnings option so that any problems will be indicated during the iteration in which they arise, not just later at the end (see `?options¶`).
- We define a function `whitespace`, which takes as a first argument a character vector, from which it deletes leading and trailing spaces, in which it reduces sequences of 2+ spaces to 1, and from which it can delete empty character strings.
- We define a vector `corpus.files` with all paths to all 4,049 BNC corpus files; also, we create a directory <05_10_freqoutput> into which to save our 4,049 interim results.
- We load each file, switch it to lower case, and keep only the corpus sentences.
- Then, we retrieve the multi-word units by looking for things beginning with a "<mw·" and ending with "</mw>".
- `if` there are such units, we retrieve their c5 attribute and, by splitting on all tags within the multi-word unit, the multi-word unit itself; we `paste` the tags and the multi-word units together and, for memory management reasons, do not store all instances but just a frequency list of the instances. Finally, before exiting this conditional expression, we delete the multi-word units from the corpus sentences so that their constituent words are not counted again.
- Then, we use nearly the same approach to retrieve the tags and the words of the 'normal,' meaning 'one-word words,' which we also `paste` together and tabulate.
- Then we merge the results for the 'normal' words and the multi-word units and save them into the directory for the interim results under the name of the current corpus file.

- After the loop, there's the next hurdle: it is again likely that just growing all 4,049 frequency tables together into one super-long one will be too much for a 'normal computer'. Thus, we reserve memory space in advance by creating two very long vectors, one for word-tag combinations (`all.word.tags`) and one for their frequencies (`all.word.tag.freqs`), each with, say, 25 million elements (more than enough for a 100 million word corpus). Then, we use a loop to load each of the 4,049 interim frequency tables and put the word-tag combinations and their frequencies into the relevant slots in `all.word.tags` and `all.word.tag.freqs` respectively. We do this by having a `counter` that always specifies the first position of new input and then inserting the next *n* elements, where *n* is the number of word-tag combinations from the current interim-results file – this is like having a large number of small empty containers in front of us and we're filling them from left to right.
- The last step is then to sum up all frequencies that belong to the same word-tag combinations so that we can save the output as a .csv file and an .Rdata file.

**What are the functions we will need for that?** One piece of good news is that, in terms of the functions this script requires, it is not challenging. We need `function`, several `if`-conditionals, and `gsub` as well as `nchar` to define `whitespace`. Next, the usual `rchoose.dir` and `dir`, but now also `dir.create` to create the folder to collect all interim-results files. Then, we do the usual `for`-loop to load each file with `scan` and get the sentences with `grep`. We then use `exact.matches.2` to find potential multi-word units. We then use `if` to check, with `is.null`, whether there are any matches for multi-word units at all, and if there are, we use `exact.matches.2` again to get the tags, and then use `strsplit` to split on tags, access the resulting list elements with `sapply` and use `paste` to glue the words together, and then `paste` again to glue them together with the tags so that we can tabulate them with `table`. Then, before proceeding with the words, we delete all multi-word units from the sentences.

We then use pretty much the same logic to extract the one-element words and `paste` them together with their tags so we can again tabulate. At the end of the loop, we use `c` to combine the multi-word unit table and the one-element word table and then `save` the result into the interim results folder.

To merge the results, we need to create long empty `character` and `integer` vectors `all.word.tags` and `all.word.tag.freqs`, and we define a position `counter`, which is initially set to 1 (because on the first iteration, we begin to put things into the first position of these two long vectors). With another `for`-loop, we load each interim result and add them into the first empty slots – the one indicated by `counter` of the two long collector vectors. Once we're done with all 4,049 frequency list files, we use `nzchar` and subsetting to delete all empty character strings that might remain in our long collector vectors and use `tapply` to `sum` up all frequencies of each word-tag combination and save the results as a .csv file with `cat` and as a data frame (created by `strsplit`ting the words and tags) into an .RData file with `save`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_10_bncxml-freqs.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs package, and source the function `exact.matches.2` (1–3)

set `options(warn)` to return warnings immediately when they arise, not later (4)

*(continued)*

*(continued)*

define the function `whitespace` (6–18)
define the vector `corpus.files` and create a directory for interim results files (20–22)
`for`-loop (counter = `i`, over `corpus.files`): (26, –89)
  output a progress report: the name of the current corpus file (27–29)
  load corpus file `i` (31)
  discard the header/find the sentences (33–37)
  look for multi-word unit tags with `exact.matches.2` (39–43)
  `if` there were any . . . (use the fact that no matches would return `NULL`) (45, –68)
    extract the tags from the multi-word unit tags (46–48)
    split the multi-word units up into parts with `strsplit` using the tags (50–52)
    merge the parts of a multi-word unit by `sapply`ing `paste` to list elements (53–57)
    `paste` together the multi-word unit tags and the units with tabstops, then tabulate into a frequency list (58–61)
    delete the multi-word unit tags and their values from the corpus (63–67)
  look for word tags with `exact.matches.2` (70–72)
  extract the words from the tags with `exact.matches.2` (74–76)
  `paste` together the word tags and the words with tabstops, then tabulate into a frequency list (78–81)
  combine the frequency tables for the multi-word units and the words and save the combined table into a file that gets the name of the current corpus file `i` (83–88)
set the working directory to the interim-results folder (93)
define two large vectors `all.word.tags` and `all.word.tag.freqs` to collect word-tag combinations and word-tag frequencies (95–97)
define a `counter` that provides where into these vectors new material gets inserted (99–100)
`for`-loop (counter = `i`, over `dir()`): (102, -117)
  output a progress report: the name of the frequency list file (103–105)
  load frequency list file `i` (106)
  insert into `all.word.tags` and `all.word.tag.freqs` the names of the frequencies and the frequencies in frequency list file `i` respectively (108–111, 112–115)
  increment `counter` to the new first unfilled positions of `all.word.tags` and `all.word.tag.freqs` (116)
discard empty character strings and their frequencies from `all.word.tags` and `all.word.tag.freqs` (119–120)
use `tapply` to sum up the values of `all.word.tag.freqs` grouped by the elements of `all.word.tags` (122–126)
`sort` the table (127–128)
save the results into one directory higher (i) as a .csv file (133–139) and (ii) as an .RData file (141–146)

**Which aspects of the script are worth additional comment?** This script is quite long and involves multiple steps, but I think everything within the first loop is probably straightforward; make sure you understand (1) how the regexes use lookaround to find things the right way and not consume too much; and how (2), in lines 85–88, `paste0` is used to paste together a path for each interim-result file that begins with the sub-directory created in line 22.

The only more difficult thing might be how we amalgamate all the frequency list files, specifically this part in the overall second loop of the script:

```
+····all.word.tags[counter:¶
+·····················(counter-1+length(current.file.freqs))¶
+···················]<-names(current.file.freqs)¶
+····all.word.tag.freqs[counter:¶
+·····························(counter-1+length(current.file.freqs))¶
+·····················]<-current.file.freqs¶
+····counter<-counter+length(current.file.freqs)¶
```

and this part after it:

```
>·result<-tapply(all.word.tag.freqs,·all.word.tags,·sum)¶
```

As is often the case, a complex procedure like this is best understood if it is applied to a very small simulated case (repeating and paraphrasing some discussion from Section 3.6.3 here simply because it's my experience that this part is challenging to beginners). In the code file, you find that, beginning in line 150:

- I define two frequency lists `freqs.1` and `freqs.2` (152–154), which simulate the interim-results files `i` we load from the sub-directory (106).
- I define two collector vectors `lett.coll` and `freq.coll` (156–158), which simulate `all.word.tags` and `all.word.tag.freqs` respectively (96–97).
- I define `counter` as 1 (line 160–161) as before (line 100).
- Then, to simulate our `for`-loop (102–117), I manually go through two iterations of it: Lines 163 to 175 show how the data from `freqs.1` are inserted into `lett.coll` and `freq.coll` and how `counter` gets incremented; lines 177 to 186 show how the data from `freqs.1` are inserted into `lett.coll` and `freq.coll` and how `counter` gets incremented again.
- We remove unused slots from `lett.coll` and `freq.coll` (188–190), just like we did before with the real data (119–120).
- Finally, we sum up the frequencies for the letters (192–196) just like before for the real data (122–126).

This manual execution of a really small example and going through a simulated loop should clarify this way of making use of predefined results vectors, and it is generally a good way to understand more complex parts of scripts, which I encourage you to use whenever you run into problems with loops (recall the end of Section 3.6.2).

As an additional practice assignment, now that this is all done, why don't you try to revise the script so that it uses the corpus files' XML annotation, i.e., try to use the packages XML and/or xml2.

## 5.3 Co-Occurrence Data: Collocation/Colligation/Collostruction

We are now turning to a variety of case studies that involve the exploration of co-occurrence data.

### *5.3.1 The Collocation* Alphabetical Order *in the BNC*

We begin with a very simple example: We want to determine the collocation strength of the collocation *alphabetical order* in the BNC but, for now, in a very simplistic way. We will just count how many sentences there are with *alphabetical* in it, how many there are with *order* in it, how many there are with both, and how many sentences there are altogether. This characterization maybe already allows you to guess what we will do with this information, namely compile it in a 2 × 2 table like Table 5.2 (bold cells are the ones whose frequencies we compute with the script, the others follow from subtraction). From this kind of table, various association measures (of collocational or collostructional attraction) can be computed (see Church and Hanks 1990 as well as Church, Gale, Hanks, and Hindle 1991 and Church, Gale, Hanks, Hindle, and Mood 1994 for early classics, and see Evert 2004, Pecina 2009, and Gries 2013a for recent overviews), and typically one would make such computations for many collocations to compare them.

Once we have this table, we will compute two kinds of association measures. First, a very simple one – namely pointwise mutual information *MI*, which is computed as shown in (8):

$$(8)\quad MI = \log_2 \frac{observed\ freq\ in\ cell\ a}{expected\ freq\ in\ cell\ a} = 6.4867$$

Positive values of *MI* reflect that the two words are more frequently observed in the same sentence than expected by chance, negative values reflect the opposite, and values close to 0 reflect a random/chance distribution.

The other measure we will compute is potentially more revealing because, unlike most association measures, it is directional. That means while *MI* quantifies how much *alphabetical* and *order* are attracted to *each other*, $\Delta P$ (read "delta P") can be computed in either direction to quantify (1) how much *alphabetical* attracts *order* and (2) how much *order* attracts *alphabetical* – as we will see, direction is not necessarily bidirectional/symmetric (see Gries 2013a for much discussion and the reference to Nick Ellis's first mention of this measure for corpus linguistics). $\Delta P$ values range from –1 (for complete repulsion) to +1 (for complete attraction) via 0 (no attraction/repulsion) and are computed as follows (for Table 5.2):

$$(9)\quad \Delta P_{row\ attracting\ colum} = \frac{a}{a+b} - \frac{c}{c+d} = 0.4219$$

*Table 5.2* Frequencies of sentences with and without *alphabetical* and *order*

| | *Sentences with order* | *All other sentences* | *Totals* |
|---|---|---|---|
| Sentences with *alphabetical* | ***a* = 96** | *b* = 129 | ***a* + *b* = 225** |
| All other sentences | *c* = 28,566 | *d* = 5,995,568 | *c*+*d* = 6,024,134 |
| Totals | ***a*+*c* = 28,662** | *b*+*d* = 5,995,697 | ***a*+*b*+*c*+*d* = 6,024,359** |

$$(10)\quad \Delta P_{column\ attracting\ row} = \frac{a}{a+c} - \frac{b}{b+d} = 0.0033$$

The result is what one might already have expected intuitively but that is ignored in >90 percent of all collocation studies: The association is not symmetric because *alphabetical* somewhat strongly predicts *order* (*order* is probably the most frequent collocate of *alphabetical*), but *order* does not predict *alphabetical* at all (because it can and does occur with so many other things).

**What are the things we will need to do?**

- We need to define the corpus files and we will again define search expressions to find *alphabetical* tagged as an adjective (AJ0) and *order* tagged as singular noun (NN1). We will again play around with the fact that the BNC is available in an XML and an SGML version, but rather than, as in Section 5.2.3, have the user state which version is being used, we will have R load the file and discover it on its own and then pick the right search expressions. We will use the fact that the SGML version has no closing word tags.
- We create four vectors that we set to 0 each: `freq.both` will be cell *a* of Table 5.2, `freq.alph` will be cell *a* + *b*, `freq.ord` will be cell *a* + *c*, and `freq.sent` will be *a* + *b* + *c* + *d*, the number of sentences.
- We load each corpus file, keep only its sentences, and add the numbers of sentences to `freq.sent`. We look for *alphabetical* and *order* and add the numbers of sentences they occur in to `freq.alph` and `freq.ord` respectively.
- In addition, we compute the number of sentences in which both are attested and add that number to `freq.both`.
- After the loop, we generate a matrix looking like Table 5.2 and compute our association measures.

**What are the functions we will need for that?** We use `rchoose.dir` and `dir` as nearly always. We use a `for`-loop with `scan` and `grep` to load each file and retain the sentences. Then we use `if` and `any` (see the very simple definition at `?any¶`) to let R check whether the file has XML or SGML annotation; depending on that, `search.expression.alph` and `search.expression.ord` will be defined in the required way. We use `length` to count the number of sentences and `grep` to find our search expressions (and `length` again for the numbers of their matches); the number of sentences with both search words we obtain with `length` and `intersect` (on the results of the `grep`s). After the loop, we create Table 5.2 with `matrix` and use `chisq.test` to compute *MI* and simple arithmetic for the Δ*P*s.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_11_alphabetical-order.r>; read this side by side with the actual script in RStudio:

clear memory and load the rChoiceDialogs package (1–2)
define the vector `corpus.files` (4–5)
set `freq.sent`, `freq.alph`, `freq.ord`, and `freq.both` to 0 (7–8)

*(continued)*

*(continued)*

`for`-loop (counter = `i`, over `corpus.files`): (10, –50)
  output a progress report: the name of the current corpus file (11)
  load corpus file `i` (12)
  discard the header/find the sentences (13–16)
  `if` R finds "</w>" in the corpus sentences (18, –26)
    define the search expressions to work with XML annotation (19–21)
  `else` (22)
    define the search expressions to work with SGML annotation (23–25)
  determine the number of sentences and add it to `freq.sent` (28–31)
  determine the number of sentences with *alphabetical* and add it to `freq.alph` (33–39)
  determine the number of sentences with *order* and add it to `freq.ord` (41–47)
  determine the number of sentences with both and add it to `freq.both` (49)
create Table 5.2 with `matrix` (58–74) and compute *MI* (76–82) and the Δ*P*s (84–89)

**Which aspects of the script are worth additional comment?** None, this one is simple.

### *5.3.2 Frequencies of Collocates of* -ic *and* -ical *Adjectives*

Our next co-occurrence case study brings us back to an earlier section: In Section 5.2.7, we explored the frequencies of *-ic* and *-ical* adjectives to determine whether *-ical* forms are more frequent than *-ic* forms. This time, we will do the same with the noun collocates following *-ic*/*-ical* adjectives; specifically, we want to determine whether the collocates of *-ical* adjectives are more frequent than those of *-ic* adjectives; to keep scripting simpler this time, we will for now only compare the overall frequencies of all collocates of all adjectives ending in *-ic* and all ending in *-ical*, not the pairwise frequencies of adjective stems that are attested at least once with both *-ic* and *-ical*, which of course would be a nice follow-up exercise for you.

**What are the things we will need to do?**

- We define the functions `just.matches` (from Sections 3.10 and 5.1.1) and `whitespace` (from Section 5.2.8) and define the corpus files.
- With a loop, we load and `tolower` each file, print a progress report to the screen, retain only the sentences, and delete all tags that are not word-tags (as well as what they annotate, so that, for instance, overlap markers don't interfere with our retrieval of the adjective–noun sequence).
- We then retrieve *-ic* adjectives followed by nouns, extract the noun collocates, and add them to a vector collecting all noun types that ever follow an *-ic* adjective; then we do the same for *-ical* adjectives.
- We put the results of the loop in a list and save it into an output file.
- Then, we load Adam Kilgarriff's BNC frequency-list data frame again (as in Section 5.2.7) and trim it down to a data frame containing only lexical nouns; as before, we make sure that, for each noun type, we add up all frequencies of all tags it occurs with (for instance, check how many different tags *kiss* comes with).

- Using the names of the frequencies, from that list we then retrieve the frequencies of all nouns ever occurring after an *-ic* adjective and all nouns ever occurring after an *-ical* adjective and compare them with `boxplot` (and a *U*-test) as well as `edcf` plots (and a Kolmogorov–Smirnov test).

**What are the functions we will need for that?** After defining `just.matches` and `whitespace` as above, we of course use `rchoose.dir` and `dir` as nearly always. We use a `for`-loop with `scan` and `grep` to load each file and retain the sentences; we use `gsub` to delete all non-word tags and their values (using negative lookahead). We then find our adjectives with `just.matches`, identify the noun collocate strings with `gsub`, and clean the results with `whitespace` before adding them to a list of types (for which we need `unique`). The result from the loop will be a `list` of the two noun-collocate vectors, which we `save`.

We then need `read.table` to load the frequency list file, and we will need `grepl`, subsetting, and `droplevels` to extract a sub-data frame with only lexical nouns from it. Next and as before, we need `tapply` and `sum` to merge the frequencies of nouns that are attested with more than one tag. Since the result of `tapply` is a set of frequencies with the names of the counted nouns, we can then use subsetting to retrieve the frequencies of all collocates of *-ic* adjectives, and then the same for *-ical* adjectives. Finally, we use `boxplot` and `wilcox.test` for plotting and testing the difference in medians of *-ic* and *-ical*-adjective collocates for significance, and `plot(ecdf(...))` and `ks.test` to do the same for the empirical cumulative distributions/percentages.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_12_ic-ical-collocates.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs package, and source `exact.matches.2` (just in case) (1–3)

define `just.matches` (5–16) and `whitespace` (18–30) as before

define the vector `corpus.files` (32–33)

define vectors `all.ic.N.types` and `all.ical.N.types` to collect noun collocates (35–36)

`for`-loop (counter = `i`, over `corpus.files`): (38, -95)

- progress report: the name of `i` and the number of collocate types found so far (39–43)
- load and `tolower` corpus file `i` (45)
- discard the header/find the sentences (47–51)
- delete all non-word annotation and their values (53–57)
- find all sequences of an *-ic* adjective followed by a noun (59–61, or 66–70)
- find all sequences of an *-ical* adjective followed by a noun (62–64, or 71–74)
- delete the tags and clean `whitespace` to get just the noun collocates (76–80 and 81–85)
- store the results in vectors that collect only `unique` collocate types (87–90 and 91–94)

store both collocate vectors in a `list` and save it into an .RData file (96–97)

*(continued)*

*(continued)*

load Kilgarriff's frequency list file and check whether the loading worked (101–106)

downsize it to one that only contains data on lexical nouns by subsetting the rows whose tag column begins with "nn" (108–116)

remove the original large data frame to free up memory (using `rm`, 118–119)

use `tapply` and `sum` to add up all frequencies of each lexical noun regardless of how many tags it is observed with (121–126)

retrieve and summarize the frequencies of the *-ic* and *-ical*-adjectives collocates by using them to retrieve the frequencies (generated by `tapply`) that have those words as names (128–130)

create a `boxplot` (134–137) and compute a *U*-test (138–140)

create an `ecdf` plot (142–147) and compute a Kolmogorov–Smirnov test (149–151)

**Which aspects of the script are worth additional comment?** None, because this script borrows heavily from the one in Section 5.2.7, but remember the potential follow-up to test the frequencies of collocates per adjective pair.

### *5.3.3 The Reduction of* to be *Before Verbs*

In this case study, we will explore corpus data in a way inspired by Bybee and Scheibman (1999): We will check whether the reduction of forms of *to be* before gerunds in only those files of the BNC that contain spoken data is correlated with (1) the frequencies with which these gerunds appear after *to be* and (2) the frequencies with which these gerunds appear in general. For that, we will need to generate two tables of which I show the first three rows here, namely tables that have gerunds in the rows, whether the form of *to be* was reduced or not in the columns, and frequencies as well as row percentages in the cells:

```
>·head(reductions,·3)¶
···········no··yes
··going···4888·7641
··gon·····1902·6160
··saying··1504·1356
>·head(reductions.perc,·6)¶
···············no·····yes
··going···0.3901349·0.6098651
··gon·····0.2359216·0.7640784
··saying··0.5258741·0.4741259
```

With these data, we can test the first hypothesis as well as generate a nice visualization, but to test the second hypothesis we will again use Adam Kilgarriff's frequency list data frame so that we can correlate the percentages of *be*-reductions with the frequencies of the gerunds in the BNC as a whole.

**What are the things we will need to do?**

- We define the functions `just.matches` and `whitespace` and define the corpus files.
- After defining collector vectors, we load each corpus file within a loop, output a progress report, but then immediately check whether the corpus file contains spoken data or not, and if not we immediately iterate to load the next file.
- If the file contains spoken data, we discard everything that's not a sentence and everything that's not a word and its tag.
- We then retrieve sequences of *to be* (using the BNC's lemma annotation) followed by a gerund (using the BNC's c5 tags) (note that there are a few tagging errors where forms of *to be* are tagged as the lemma *was*).
- We extract the forms of *to be* and the gerunds and store them in separate vectors; after the loop, all the results go into a data frame whose rows we order by row sums.
- We then cross-tabulate the gerunds with whether their form of *to be* was reduced or not both using raw frequencies and percentages of reduction (see above).
- We then plot reduction percentages against gerund frequencies and compute Kendall's *tau*.
- Then, for the second exploration, we load Kilgarriff's data frame again, trim it down to only gerunds, and extract our gerunds' overall corpus frequencies from it so that we can generate an analogous plot and compute an analogous Kendall's *tau* for those frequencies, too.

**What are the functions we will need for that?** After defining `just.matches` and `whitespace` as above, we of course use `rchoose.dir` and `dir` as nearly always. We use a `for`-loop with `scan` to load each file and `if` as well as `grepl` to check for whether the corpus file contains spoken data. If it does, we use `grep` to retain the sentences and `gsub` to delete all non-word-tags and their values (using negative lookahead). To retrieve all sequences of *to be* plus gerund we use `just.matches`, and we extract forms of *to be* and the following gerunds using `just.matches` and `gsub` (plus `whitespace` to clean them up).

After the loop, we compile all data into a data frame and use `table` as well as `prop.table` to create `reductions` and `reductions.perc` as well as `order` and `rowSums` to re-order their rows. Then, we use `plot`, `text`, `abline`, and `lines(lowess(...))` for the scatterplot and `cor.test` for the correlation. For the second study, we load the data frame with `read.table`, trim it down with `subsetting` and `grepl`, and `tapply` with `sum` to sum up all frequencies per gerund. With those data, we can then create a new plot and do a new correlation test with the same code as before.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_13_reduction-be.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs package, and source `exact.matches.2` (just in case) (1–3)

define `just.matches` (5–16) and `whitespace` (18–30) as before

*(continued)*

*(continued)*

define the vector `corpus.files` (32–33)

define vectors `all.bes` and `all.verbs` to collect those forms (35–36)

`for`-loop (counter = `i`, over `corpus.files`): (38, –104)

- progress report: the name of `i` and the number of *be*'s + gerund found so far (39–42)
- load and `tolower` corpus file `i` (44)
- `if` R does not find "<stext" in corpus file `i` (47–51)
  - go to the next iteration (52)
- discard the header/find the sentences (54–58)
- delete all non-word annotation and their values (60–64)
- find all sequences of *to be* followed by a gerund (66–68, or 70–74)
- extract the forms of *to be* from those sequences and store them (76–83, or 85–94)
- extract the verbs after *to be* from those sequences and store them (96–103)

compile the results of the loop into a data frame `result.of.loop`; include a column indicating whether the form of *to be* is reduced, and save it into an .RData file (106–114)

generate the frequency and proportion tables `reductions` and `reductions.perc` shown above (116–118) and order them by the frequencies of the verbs after *to be* (120–128)

create a scatterplot with the frequencies of the gerunds after *to be* on the *x*-axis and the proportion of *be*-reductions on the *y*-axis (133–137), `text` the gerunds in there (138–141), customize (142–143, 144–145), and compute Kendall's *tau* (146–149)

load Kilgarriff's frequency list file and check whether the loading worked (154–159)

downsize it to one that only contains data on gerunds by subsetting the rows whose tag column contains "vvg" (161–169)

remove the original large data frame to free up memory (using `rm`, 171–172)

use `tapply` and `sum` to add up all frequencies of each gerund regardless of how many tags it is observed with (174–177)

check whether the frequency data frame covers all your gerunds and set the frequencies of missing gerunds to 1 (so that, when you log them for the plot, they become 0) (179–185)

create the same scatterplot as above now with the overall frequencies of the gerunds (179–199) and compute another Kendall's *tau* on it (200–203)

**Which aspects of the script are worth additional comment?** Not many additional comments are needed, I think, because this script borrows heavily from several previous ones, but I want to comment on one aspect, namely the check of whether the frequency list file made from the BNC Version 1 covers all gerunds in your concordance of the newer BNC World Edition:

```
>·(missing.vs<-setdiff(rownames(reductions),·names(freqs.vvg)))¶
>·(missing.where<-match(missing.vs,·rownames(reductions)))¶

>·freqs.vvg<-freqs.vvg[rownames(reductions.perc)]¶
>·freqs.vvg[missing.where]<-1¶
>·names(freqs.vvg[missing.where])<-missing.vs¶
```

The first line uses `setdiff` to check which of the gerunds you have in your concordance data are not in the frequency list and returns those verb forms; the second then uses `match` to find where in the tables `reductions` and `reductions.perc` these show up.

The third line retrieves the frequencies with which all your gerunds are attested in Kilgarriff's data frame just using subsetting of names again, but of course we now know that two of those will not be found, which is why lines 4 and 5 set those frequencies to 1 (for the later logging) and the names of these frequencies to the two missing verbs.

### *5.3.4 Verb Collexemes After* Must

At the very beginning of this section, in Section 5.3.1, we dealt with computing association measures for the collocation *alphabetical order* – but even then I already mentioned that, typically, one doesn't just compute an association measure for one collocation – rather, one computes them for potentially many collocations and then ranks those by their strength of association. This is what we will do here: We will explore which verbs are most strongly attracted to the position after the modal verb *must* in the folder K of the BNC World Edition. In other words, now we need a table like Table 5.3 for every verb attested after *must*, i.e., potentially thousands of them or more.

But that raises two interesting questions: First, as you know and can see here again, this kind of association measure computation requires us to know not only how often, say, *must admit* occurs – we also need to know how often *must* occurs altogether ($a + b$), how often *admit* occurs altogether ($a + c$), and a corpus size ($a + b + c + d$) – and actually we need all that for all potentially hundreds or thousands of verb types after *must*. But we can't determine those frequencies all in one loop: Imagine we write a loop that looks for *must* as a modal verb and then extracts all verbs $v_{1-n}$ after it so that it can then also determine those verbs' frequencies. Okay, but what if *must admit* is not attested in the first file but only in the second? Then we will already have missed the frequency of *admit* in the first file. Thus, we can't do it like this – rather, we must have a second independent loop. In the first loop, we identify all cases of *must* + V so that, at the end of it, we know all verb types ever occurring after *must*, and then we can look for all occurrences of all of them in the whole corpus within the second loop.

This brings us to the second question, namely the tricky and often misunderstood question of granularity: What level of resolution are we adopting for the counts of our table(s)? Above, we only had one table and since we were counting occurrences in sentences, the level of resolution was clear: sentences. But here we are looking at something much

*Table 5.3* Frequencies of *must*/other *admit*/other in BNC K

| | $Verb_2 = admit$ | $Verb_2 \neq admit$ | *Totals* |
|---|---|---|---|
| $Verb_1$ = *must* | $\boldsymbol{a}$ = **164** | $b$ = 1,951 | $\boldsymbol{a + b}$ = **2,115** |
| $Verb_1$ = other modal | $c$ = 14 | $d$ = 68,128 | $c + d$ = 68,142 |
| Totals | $\boldsymbol{a + c}$ = **178** | $b + d$ = 70,079 | $\boldsymbol{a + b + c + d}$ = **70,257** |

more specific: verbs after the modal verb *must*. The solution, which weirdly enough seems surprising to some, is in fact fairly obvious: We choose a level of resolution that is as similar to the phenomenon being studied as possible (see Gries 2012; 2015). Thus, we are of course not using sentences and also not words (and of course not characters or files), but something close to our question: If our study is on verbs after the modal verb *must*, which means that the left column will always be "$verb_2$ is some verb" and the right column will always be "$verb_2$ is all-other-verbs", then one reasonable possibility is making the rows work the same: The upper row will always be "$verb_1$ = modal verb *must*" and the lower row can correspondingly be "$verb_1$ = any-other-modal-verb", which is what Table 5.3 already shows. There are 70,257 occurrences of modal verbs followed by infinitives in BNC K; 2,115 of those are cases where *must* is the modal verb, *must admit* occurs 164 times, and *admit* occurs after modals 178 times.

Finally, it will turn out that there are 440 verbs occurring after *must* as a modal verb, so how do we compute all these tables and then also an association measure for each? Thankfully, since this kind of question was one I frequently used in works using collostructional analysis (Gries & Stefanowitsch 2004a; 2004b; Stefanowitsch & Gries 2003; 2005), I wrote an interactive R script (<coll.analysis.3.r>) that you can use by sourcing it directly from a website of mine (http://tinyurl.com/collostructions): This script will ask you a few questions interactively at the console and, after you have answered them and provided an input file in a particular format, it will do everything for you (see <_qclwr2/_outputfiles/14_must_out.csv>).

**What are the things we will need to do?**

- We define the function `just.matches`, the corpus files, and two collector vectors: one character vector `infs.after.must` that will collect all infinitives of verbs occurring after *must* as a modal verb (that is, from this we can compute all cells *a* and *a* + *b*), the other is the numeric vector `modals.plus.inf`, which is initially set to 0 but will then determine the sample size *a* + *b* + *c* + *d*.
- We load each corpus file, trim it down to just the sentences, and then look for all sequences of any modal verb followed by an infinitive; we store that in `current.mpis`; we add the number of matches to `modals.plus.inf`.
- Then we extract from `current.mpis` the cases where the modal is *must* and extract from those cases the lemma of the verb after *must*, which we add to `infs.after.must`. This way, not only does `infs.after.must` collect those infinitives, but its length will ultimately determine cell *a* + *b*.
- After this first loop, we can determine for each table the frequency of *must* + V (*a* + *b*) and corpus size (*a* + *b* + *c* + *d*), but also all *must* + $V_1$, *must* + $V_2$, etc., i.e., all *a*s. Also, we know how many different verbs occur after *must* (440) and what those are, so after creating a collector vector for the cells *a* + *c*, we do a second loop over the corpus files where now we determine the frequencies of these verbs after modals in general, not just after *must*) by looking into this loop's `current.mpis`.
- We then save the results in the format that <coll.analysis.r> needs for a simple collexeme analysis (see readme.txt at http://tinyurl.com/collostructions) and then source the script from my website from within RStudio and run it, providing the answers listed in the script.

**What are the functions we will need for that?** Nothing challenging here: `rchoose.files` and `dir` as nearly always. A `for`-loop will load each corpus file with `scan`, we use `tolower` and `grep` to get lower-case sentences, and we use `just.matches` to find our modal verbs plus infinitives as well as `grep` and `gsub` to extract the infinitives after *must*. We then

tabulate the infinitives after *must*, create a vector `freqs.overall` (using `rep`) with a 0 for each infinitive, and enter into a second loop that does everything as before but now has another `for`-loop in it in which we look for each of the infinitives using a search expression created with `paste0`; we add the frequency of each infinitive to the relevant slot of `freqs.overall` and then, after the loop, use `data.frame` to save the results in the relevant format. After that, we start <coll.analysis.r> by sourcing it from www.linguistics.ucsb.edu/faculty/stgries/teaching/groningen/coll.analysis.r, answer its questions, and let it do the rest.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_14_must-V.r>; read this side by side with the actual script in RStudio:

clear memory, load the tcltk package, and source `exact.matches.2` (just in case) (1–3)
define `just.matches` as before (5–16)
define the vector `corpus.files` (18–19)
define vectors `infs.after.must` and `modals.plus.inf` to collect results in the loop (21–23)
`for`-loop (counter = `i`, over `corpus.files`): (25, –49)
  output a progress report (26) and load the current corpus file `i` (28)
  extract the sentences from it (29–32)
  look for *must* (as a modal verb) plus an infinitive (34–37)
  add the length of the result to `modals.plus.inf` (38–39)
  extract the cases where the relevant modal verb is *must* (41)
  extract the infinitives from there and add them to `infs.after.must` (42–48)
find all infinitive types after *must* and their frequencies (`freqs.overall`) (51–52)
compute and look at $a + b$ and $a + b + c + d$ from the results (54–55, 57–58)
create `freqs.overall` to store the $a + c$ frequencies (62–65)
`for`-loop (counter = `i`, over `corpus.files`): (67, –105)
  output a progress report (68) and load and prepare the current corpus file `i` (70–74)
  look for *must* (as a modal verb) plus an infinitive (76–79)
  `for`-loop (counter = `j`, over `names(freqs.overall)`): (81, –92)
    `paste0` together a search expression to find the current infinitive (82–83)
    find that current infinitive as a lemma in the modal verb matches, add the number of matches to the slot for that infinitive of `freqs.overall` (85–91)
  (check out the code alternative to the inner loop, which is not really much easier (94–104)
compile all results in a data frame and save it into a .csv file for the collocation/collexeme analysis (107–112, 114–116)
run the <coll.analysis> script from my website and answer the questions it asks you according to the instructions in the script (120–134), then explore the output file

**Which aspects of the script are worth additional comment?** None really: This script uses only things we have already discussed at length.

### *5.3.5 Noun Collocates After Speed Adjectives in COCA (Fiction)*

After having worked through so many examples using the BNC with XML annotation – because the BNC is such a widely used corpus and XML such a widely used form of annotation – we will now explore two corpora in a different format. In this case study specifically, we will look at four near-synonymous speed adjectives – *fast*, *quick*, *rapid*, and *swift* – and the nouns they premodify in the fiction component of the Corpus of Contemporary American English (COCA), which is available for download from http://corpus.byu.edu/full-text. It comes in several formats but we will use the one shown in Figure 5.3.

As you can guess, the first line contains an identifier code, but then the results is a tab-delimited table with a form, a lemma, and a POS-tag column (using the CLAWS7 tag-set shown at http://ucrel.lancs.ac.uk/claws7tags.html). What we want to create is two things: First, we want a data frame that lists for every occurrence of any of the four adjectives (excluding for now comparative and superlative forms) the noun that it premodifies and the year in which that co-occurrence was observed, as shown in Table 5.4.

Second, from that we want to generate a list `top.10.collocates` that provides for each year for which we have corpus data the top ten most attracted noun collocates of each adjective. We will then save that into a spreadsheet.

One question may arise here: What if you don't have access to COCA (and COHA, which we'll deal with in the next section)? If that's the case, jump ahead to the end of this section for a moment, do the assignment mentioned there, and then come back here.

```
##1001741
"     "     y
If    if    cs
you   you   ppy
do    do    vd0
n't   n't   xx
sit   sit   vvi
your  your  appge
stinkin     stinkin     nn1
'     '     ge
,     ,     y
useI  usei  nn2
less  less  rrr_rgr
butt  butt  vv0
back  back  rp
down  down  rp
```

*Figure 5.3* The first few lines of <wlp_fic_1990.txt>.

*Table 5.4* Desired co-occurrence results to be extracted from COC5.3A: fiction

| *Adjective* | *Noun* | *Year* |
|---|---|---|
| Quick | Success | 1990 |
| Fast | Action | 1990 |
| Rapid | Motion | 1990 |

But assuming you have COCA for now, **what are the things we will need to do?** This script is more complex than most others:

- We define `just.matches` and a vector `corpus.files` with all paths to the 23 COCA fiction files; also, we establish empty vectors to collect adjective lemmas, noun lemmas, and years.
- In a loop, we print a progress report, load each file (using the file connection approach to assign the required ISO-8859-1 encoding during the input), and discard the first line; then, we split up the corpus file at tabstops and extract the lemmas and their tags from the resulting list.
- We then look for (1) the positions of the adjective lemmas we're interested in and (2) the positions of adjective tags (jj); the desired rows – where these lemmas are attested as adjectives – are the intersection of these two positions.
- However, we only want the four adjectives when they occur before nouns, so we look for the positions of noun tags and see whether these overlap with the positions of our adjective lemmas when tagged as adjectives plus 1.
- We then retrieve the four adjective lemmas (when they precede nouns) and the noun lemmas (when they follow the four adjective lemmas) and store them in our collector vectors.
- We then also add the year covered in the current file (as many times as there are matches) to the vector collecting the years.
- After the loop, we store everything in a data frame looking like Table 5.4 and save that into a .csv file.
- To compute association measures for each year, split up that data frame by its years column (obtaining a list); we also create a list `top.10.collocates` with as many elements as there are years in the corpus to collect collocates and their association strengths.
- We loop over this list (i.e., access data from each year) and, for each year, create a frequency table with the nouns in the rows, the four adjectives in the columns, and their co-occurrence frequencies in the cells, to which we apply `chisq.test` to retrieve the Pearson residuals of that table, which we store in `current.residuals`.
- In a second loop, we take each of the four columns of the current year's `current.residuals`, sort them by amount, and put the largest ten values into the adjective-specific element of `top.10.collocates` for this year.
- Finally, we convert that into a data frame and save it into a second .csv file.

**What are the functions we will need for that?** We use `rchoose.files` to define the corpus files and `c` to create vectors collecting results. We use a `for`-loop to load each file with `readLines(file(...))` and `tolower`, and then we use `strsplit` to split up the file by column; from the columns, we use `sapply` and `"["` to extract lemmas and tags. We find adjectives and adjective tags with `grep` and `which` plus `"=="` and we find noun tags with `grep`; intersections of positions we find with `intersect`, and we use subsetting to retrieve the relevant words; the years are just added with `c` and `rep` (of the `basename` of the corpus file) plus `length`.

After the loop, we create Table 5.4 with `data.frame` and save it with `write.table`. We `split` it up by year and use `vector(...,·mode="list",·length=...)` to create an empty list with the relevant number of elements. We then `for`-loop over this list, creating the frequency table of nouns and adjectives with `table` for each year and computing the residuals with `chisq.test(...)$residuals`. In a second, inner `for`-loop, we then put the `sort`ed top ten noun collocates for each adjective (using `colnames`) into the list.

Finally, we use `unlist` and `data.frame` to convert this into a nice data frame and save that with `write.table`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_15a_speed-adj-in-coca.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs package, and set options to show warning right away (1–3)
define `just.matches` (5–16) and the vector with the corpus files (18)
define collector vectors for adjectives, nouns, and years (20–21)
`for`-loop (counter = `i`, over `seq(coca.fiction.files)`): (23, –65)
  output a progress report (24–27), load the file, `tolower` it, and discard the first line of the current corpus file (29–34)
  split the corpus file up into a list by column, i.e., at tabstops (35)
  retrieve the lemmas from the second positions of the list elements (37)
  replace zero-character lemmas, if any, with "NA" (38)
  retrieve the tags from the third positions of the list elements (39)
  find the four adjectives but only when they are tagged as such (41–44)
  find all noun tags (45)
  determine where the four adjectives precede nouns (48)
  determine where nouns follow the four adjectives (49)
  retrieve the relevant adjectives and nouns and store them in collector vectors (51–54)
  add the relevant number of occurrences of the year (which you get from the corpus file name) to the collector vector for years (56–64)
compile the results into a data frame (67–71) and save it (72–74)
`split` up that data frame by year into a list `result1.list` (76–77)
create a list `top.10.collocates` to collect the top ten noun collocates/adjective/year (79–81)
`for`-loop (counter = `i`, over `seq(top.10.collocates)`) (83, –95)
  extract the current year's data frame from `result1.list`, add `droplevels` (84)
  create a `table` of nouns and adjectives, compute a chi-squared test on it, and extract the `residuals` (85–88)
  `for`-loop (counter = `j`, over 1:4 (90, –94)
    put into `top.10.collocates` for the relevant year (= `i`) and the relevant adjective (for which you use `colnames` and `j`) the highest ten residuals for that combination of year and adjective (91–93)
take that list, convert it into a data frame (98–104) and save it into a file (105–107)

**Which aspects of the script are worth additional comment?** One small clarification may be necessary regarding line 38: The "NA" there is *not* the special logical constant `NA` indicating missing values but just a normal character string, just so that we avoid empty strings; I made it "NA" so that (1) it's similar to `NA` and (2) because the mere

fact that it's two capital letters in an otherwise `tolower`-ed corpus will indicate that it's not a real word.

The trickiest part here is probably the two loops that compute and then collect as well as sort the residuals of the adjective–noun co-occurrences in lines 83–95. One important thing to note is again the use of `droplevels` to make R not show zero frequencies for all sorts of words that are not even attested in the data. Lines 85 to 88 are then just a compact representation where a table of nouns and adjectives is created, but never stored or used other than as immediate input to `chisq.test`, and even the result of that main significance test is never used but we immediately jump to the residuals part of that test's output, which we essentially use as an association measure – this may strike you as strange, but recall (1) from Section 5.3.1 that, for instance, an association measure such as *MI* is based on the ratio of observed and expected frequencies, and (2) from Section 4.2.2 that the residuals of a chi-squared test are, too – a simulation with 1,000 random 2 × 2 tables shows that *MI* values and the residuals are in fact very highly correlated with each other (adj. $R^2$ of a polynomial regression (second degree) of residuals against *MI*s was 0.87).

Now, what if you don't have COCA and COHA? These corpora are 'available' but not 'freely available' so not everyone will have them. If you do in fact not have them, here's a very nice solution to the problem in the form of an extra assignment: Write a corpus-conversion script that changes the whole BNC into the COCA format exemplified in Figure 5.3 so that you can then run nearly all of the script I discuss here for COCA on the transformed version of the BNC. To make it more interesting, write that corpus-conversion script in such a way that it handles multi-word units properly where by *properly* I mean that the multi-word unit in Figure 5.2 gets represented as in Figure 5.4.

(In the above script for COCA, we also use the information of which year corpus data are from – feel free to ignore that or, alternatively, use the creation date of the BNC file for that; I would recommend that you then conflate the years into decades and find a way to handle rare decades or other little problems.) Once you're done with your script, (1) run it of course and compare it to my version in <_qclwr2/_scripts/05_15b_BNCasCOCA.r> and (2) tweak the above script so that it can run on the BNC-like-COCA corpus.

### *5.3.6 Collocates of* Will *and* Shall *in COHA (1810–90)*

In this case study we will again look at collocates, this time at collocates in a window of three words to the left and three words to the right of the verbs *will* and *shall* in the nineteenth-century data from the Corpus of Historical American English (COHA). Like COCA, COHA is available for download from http://corpus.byu.edu/full-text, in the same format as COCA shown above. Again, if you don't have COHA yourself, use the script <_qclwr2/_scripts/05_15b_BNCasCOCA.r> or, better, your own script, to convert the BNC into the tabular COCA/COHA format we are working with here and then, after reading this section, tweak the script minimally so that it works on the BNC-as-COHA corpus.

Given this similarity in format, several aspects of the code will be quite similar to the previous case study. However, this time we want two kinds of different output. The first one is essentially just a small concordance display and looks like this:

```
even when          even when          CJS
```

*Figure 5.4* Desired result of transforming Figure 5.2 into the COCA format of Figure 5.3.

```
>·head(results)¶
··YEAR·DECADE·····L3·····L2·····L1·NODE·····R1···R2··R3
1·1810···181_·dramatic·literature·····,·will·perceive··that·the
2·1810···181_·····lord·····'s·merry-making·will·····sit·sadly··on
3·1810···181_·····;·····but·····he·will····never···be·the
4·1810···181_·····one·····,·····i·will····swear·····,·for
5·1810···181_···madona·····,·····i·will·····tell···you·now
6·1810···181_·····lady·····.·····i·will·····but··read·and
```

For this, we will retrieve positions of matches, i.e., where *will* and *shall* will occur tagged as modal verbs, which will be numbers such as 6, 29, etc., and that means we will need to retrieve the words in positions 3, 4, 5, **6** (the match), 7, 8, 9 for the first match, and 26, 27, 28, **29** (the match), 30, 31, 32 for the second match, etc. That could be achieved using something like `(matchposition-3):(matchposition+3)` – but what if the match is the first or second word in the corpus, or the penultimate or the last, meaning the subtraction and addition of 3 will result in nonsensical position values? In order to handle such cases, it is useful to define a function that I will here call `ranger` and that is provided in the code file, which you should explore (in general and for its use of `lapply` with an anonymous function). This function takes as input:

- a vector of numbers called `positions`, which typically will be positions of matches in a vector of words (i.e., in the above example 6, 29, etc.);
- an argument `before.and.after`, which says how many collocate, i.e., slots, to the left and right of the matches one wants (the default is set to five but we want only three here). As a programming exercise, you might want to tweak the function such that it can have different numbers of collocates on the left and on the right;
- an argument `desired.min`, which indicates the earliest vector position you might want as collocate slots (the default is of course 1, the first word slot in the corpus (file));
- an argument `desired.max`, which indicates the last vector position you might want as collocate slots (the default is the maximum of `positions`, but you should set it to the length of the vector of words that you will subset so that the last word in the corpus (file) could be shown as a collocate));
- two more arguments `padded` and `with.center`, which you usually shouldn't need to change from their default setting of `TRUE`, which is why I will not explain them here – play around with them if you want to get to know them.

Check out the small example in the code file to see how `ranger` gives you all possible collocate positions for a corpus vector of a certain length (and pads the remaining positions with NAs so all output is reliably equally long).

The other display we want is that exemplified in Table 2.2 or Table 2.3, i.e., one where for every time period and modal verb (i.e., *will* vs. *shall*), we can see the, say, 15 most frequent collocates per slot as shown in Figure 5.5.

**What are the things we will need to do?**

- We define `ranger` and a vector `coha.files` with all paths to the 10,000 or so COHA files from the nineteenth century (or your BNC surrogate); also, we define the search terms and the window span.

| L3 | L2 | L1 | NODE | R1 | R2 | R3 |
|---|---|---|---|---|---|---|
| , : 217 | , : 322 | i : 365 | shall : 2224 | be : 386 | the : 107 | to : 117 |
| . : 152 | . : 145 | we : 163 | will : 0 | i : 170 | be : 103 | , : 113 |
| the : 65 | and : 124 | , : 125 | | not : 140 | , : 80 | the : 112 |
| NA : 60 | the : 91 | you : 123 | | have : 87 | . : 54 | . : 111 |
| and : 59 | -- : 84 | he : 67 | | never : 66 | to : 51 | NA : 60 |
| of : 54 | ! : 73 | and : 59 | | we : 63 | you : 44 | in : 48 |
| ; : 50 | NA : 63 | thou : 59 | | , : 28 | my : 43 | of : 46 |
| ! : 37 | that : 56 | it : 54 | | find : 27 | do : 41 | ? : 38 |
| -- : 36 | ; : 54 | what : 52 | | see : 27 | it : 37 | with : 36 |
| " : 26 | my : 48 | how : 40 | | hear : 23 | NA : 36 | and : 30 |
| my : 23 | of : 48 | who : 37 | | know : 23 | a : 35 | you : 30 |
| me : 22 | but : 40 | they : 35 | | make : 23 | in : 35 | be : 28 |
| to : 20 | " : 31 | which : 33 | | the : 23 | me : 26 | for : 27 |
| that : 18 | when : 23 | NA : 24 | | soon : 21 | with : 26 | my : 27 |
| (Other) : 1385 | (Other) : 1022 | (Other) : 988 | | (Other) : 1117 | (Other) : 1506 | (Other) : 1401 |

*Figure 5.5* A 3L-3R collocate display of *shall* as a modal verb (1810–1819 in COHA).

- We create a matrix with seven columns (see Figure 5.5) for the results.
- We go through an outer loop to load each corpus file (in the same way as in the previous section); we split it up into columns, from which we access the word lemmas and the tags; also, we determine where the modal verb tags are.
- We go through an inner loop to search for each search word by looking for the intersections of the positions of the word in the lemmas and the positions of the modal verb tags (this wouldn't have to be a loop – you could just use the same code twice, once for *will*, once for *shall* – but putting in a loop there right away makes the script easy to reuse if you have more than two search expressions: You just change the vector with the search terms and all else stays the same).
- For the positions of each search word, we use `ranger` to recover the relevant 3L to 3R collocate slots around the matches and put them into a matrix, which we add to all collocates from the previous search words and files.
- We then change the results into a data frame that we can nicely save and, to get results like Figure 5.5, we split it up by decade–verb combination and generate a summary output for each decade–verb combination.

**What are the functions we will need for that?** We use the function `ranger`, which uses `lapply` to apply `seq` to all numbers in its first argument `positions`, with `seq`'s starting point being the maximum of `desired.min` and the smallest starting point of collocates and its end point the minimum of `desired.max` and the end point of the input vector. We use `rchoose.dir` and `dir` as nearly always but, since the COHA files are in separate directories, may have to use `sapply` and `dir` to look into all user-specified directories to retrieve the files. We use `matrix` to create the beginning of the matrix `results` into which we will dump all collocates.

We `for`-loop over the files, which we load with `readLines(file(...))` and process with `tolower` and then `strsplit` (as in the previous section); thus, we can use `sapply` and `"["` to extract lemmas and tags. We determine modal-verb positions with `grep` and, in a second `for`-loop, find the intersections of the search words' positions with those of the modal tags. Once we have the position of the intersecting matches, it's time for `ranger`

to return collocate slots 3L to 3R around them. The collocate slots are then `unlist`ed and used with subsetting on the vector with all lemmas in the file, which is in turn passed on to `matrix`, which generates a representation of the type of Figure 5.5. (This wouldn't have to happen in the loop, but it seems to make little difference in processing time.) After the loop, we use `data.frame` to compile all info in a data frame, which we `save` into an .RData file. Finally, we `split` up the data frame by decade (which we obtained by using `sub` to delete the last character of the four-digit year) and search word, and we use `lapply` as well as `summary(...,·maxsum=15)` to get the 15 most frequent collocates for each slot.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_16_modals-in-coha.r>; read this side by side with the actual script in RStudio:

clear memory and load the rChoiceDialogs package (1–2)
define the function `ranger` (4–25)
define the vector `coha.files` (assuming you have to pick the folder with all of COHA installed in it (41–48)
define the search terms (50) and the window span (51)
set up a matrix for the results (53–58)
`for`-loop (counter = `i`, over `seq(coha.files)`): (62, –105)
  output a progress report (63–66), load, and trim the current corpus file (68–72)
  split the corpus file up into a list by column, i.e., at tabstops (73)
  retrieve the lemmas from the second positions of the list elements (75)
  replace zero-character lemmas, if any, with "NA" (76)
  retrieve the tags from the third positions of the list elements (77)
  find modal verb tags (79–81)
  `for`-loop (counter = `j`, over `search.terms` (83, –104)
    look for the current search word (84) and determine where the matches `intersect` with the modal verb tags (85–86)
    retrieve the 3L–3R collocate slots around the matches (88–91)
    retrieve the lemmas in those collocate slots, organize them into a `matrix` (93–102)
    add these results to the previously collected ones (103)
delete the first row of the matrix (107), change it into a data frame (108–114), and save it (115)
split the results up by decade (recall line 103 above) and verb (117–121), `lapply` `summary` to each vector of collocates (122–123), and explore the top 15 collocates (125–129)

**Which aspects of the script are worth additional comment?** The most difficult part of this script, I think, is the retrieval of the collocates: first the slots, then the actual words. The slots are retrieved in lines 88–91 with `ranger` and they return an object `collocate.slots`, which, if one interrupted or simulated the loops, would look like this:

```
>·collocate.slots[1:3]¶
$`26554`
[1]·26551·26552·26553·26554·26555·26556·26557
$`26777`
[1]·26774·26775·26776·26777·26778·26779·26780
$`33148`
[1]·33145·33146·33147·33148·33149·33150·33151
```

That means there are matches (here of *shall*) in positions 26,554, 26,777, and 33,148 of the file, and the list elements have these position indices as names (see these numbers after the $ signs), and then each list element is a vector with the position indices beginning three words earlier and ending three words later. But how is that used in the subsequent lines 93–101, which are compact, to put it mildly? If you used just lines 93–95, the `matrix` that these lines generate would look like this: The positions in `collocate.slots` are `unlist`ed and then used to subset `current.lemmas`, i.e., retrieve the lemmas from those positions, which are then organized into a `matrix` that has twice the window span plus the match, which makes seven columns:

```
·····[,1]··[,2]·····[,3]·····[,4]····[,5]·····[,6]·····[,7]
[1,]·"and"·"beautiful"·"cecilia"·"shall"·"appear"·"in"·····"the"
[2,]·"."···"the"·····"picture"·"shall"·"be"·····"return"·"to"
[3,]·"and"·"land"·····"we"·····"shall"·"have"···"see"····"by"
```

The remaining lines, 96–101, just create nice dimension names: The name of the corpus file is `rep`eated as many times as there are matches (with collocates around them). From that, everything till the first underscore is deleted with `sub`, which makes the remainder of the string begin with the four-digit year of the file name. That year is then extracted using `substr(...,·1,·4)`, and those become the row names of the matrix. The column names are just numbers from –3 to +3, which represent 3L, 2L, 1L, 0 (for the match, here *shall*), 1R, 2R, and 3R. The final result:

```
·····-3····-2·····-1·····0·····1·····2·····3
1899·"and"·"beautiful"·"cecilia"·"shall"·"appear"·"in"·····"the"
1899·"."···"the"·····"picture"·"shall"·"be"·····"return"·"to"
1899·"and"·"land"·····"we"·····"shall"·"have"···"see"····"by"
```

### *5.3.7 Split Infinitives*

The final case study in this section on co-occurrence deals with split infinitives, i.e., examples such as *to boldly go*, which contrast with *to go boldly*. We will return to the BNC XML (just folder A this time) to answer the following questions:

- Which adverbs and which verbs are most often observed in the split construction compared to the overall frequency of split vs. non-split constructions?
- Which combinations of adverbs and verbs are most often used in the split construction?
- To get an idea of how frequent both constructions are, we'd like to know which words are about as frequent as split and non-split infinitives in the same corpus? Saying that a certain construction occurs *x* many times per million words is not intuitive to understand, whereas saying that a construction is as frequent as words *a*, *b*, *c* is more straightforward.

**What are the things we will need to do?**

- We define the function `whitespace` and a vector `corpus.files`; also, we define two search expressions – one to find split infinitives, the other to find non-split ones.
- We create two collector vectors for split (`all.splits`) and non-split infinitives (`all.nonsplits`) as well as an interim-results directory (<_qclwr2/_outputfiles/05_17_freqoutput>) in which to store frequency lists of all words per file (which we later will amalgamate into one frequency list as in Section 5.2.8).
- We load each corpus file, trim it down to what we need, and look for split and non-split infinitives and add them to their respective collector vectors.
- We also create a frequency list of each file and `save` it as an .RData into the interim-results folder.
- After the loop, we extract all adverbs and verbs from both the split and the non-split infinitives, which means we can look at the most frequent adverbs and verbs in either construction and also count the numbers of split and non-split infinitives.
- We then identify the adverbs that are attested in both constructions, how often they are attested in each construction (raw frequencies), how often in percent they are used in the split construction, and how much higher that number is than the overall (baseline) percentage of split infinitives out of split *and* non-split infinitives (e.g., if split infinitives made up 10 percent of all infinitives and *actually* is attested in split constructions 25 percent of the time, we'd get 15 percent as a result); we then plot these 'split-preference' values.
- Then, we do the same for the verbs.
- We also create a frequency table of all split constructions to see which combinations of adverbs and verbs are most frequent in split infinitives.
- Finally, we amalgamate all frequency lists from <05_17_freqoutput> and list all words that are as frequent as both kinds of infinitives plus/minus 5 percent.

**What are the functions we will need for that?** We use `whitespace`, which mostly uses `gsub`, and as always we use `rchoose.dir` and `dir`. We use the usual `for`-loop with `scan` and `grep` to choose and process the files, and then we use `exact.matches.2` to retrieve all the infinitives as well as all lemmas for the overall frequency list. After the loop, we use `strsplit` as well as `sapply` with `whitespace` to extract adverbs and verbs; we use `length` to count how many of each infinitive construction there are, and we use `intersect` to determine which adverbs are used in both constructions. We summarize those in a `matrix`, to which we apply `prop.table` to get the percentages of split infinitives, and then `plot` with `text` and `abline` to visualize the adverbs' preference for the split construction. Then we do the same for the verbs. For the frequencies of verb–adverb combinations, we use `paste` and then `table`.

The last step consists of merging the frequency list files: We generate an empty `table` first, and then use another `for`-loop to `load` each frequency list file and merge them

into one long table (with `c`). Then we group the frequencies by the lemmas with `tapply` and `sum` up all frequencies of each lemma (we don't do the insertion into a long vector with counter here because, since we're only looking at BNC folder A, the data set is small enough for this to work even on a computer that's not state-of-the-art). Finally, we compute the `abs`olute difference of all frequencies from the construction frequencies and `sort` and display those that do not differ from the construction frequencies by 5 percent or more.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_17_split-infs.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs package, and source `exact.matches.2` (1–3)

define the function `whitespace` (5–17)

define the vector `corpus.files` (19–20)

define two search expressions (22–24), the collector vectors `all.splits` and `all.nonsplits` (27), and create the interim-results output directory (28)

`for`-loop (counter = `i`, over `corpus.files`): (30, –70)

- output a progress report: the name of `i` (31–33)
- load and `tolower` corpus file `i` (35)
- retrieve only sentences (37–41) and delete all non-word annotation and its values (43–47)
- retrieve and collect all split (50–53) and non-split infinitives (54–57)
- retrieve all lemmas in the file (59–62), tabulate them (63), and `save` them in the dedicated temporary folder (64–68)

split up the matches of both constructions and extract the verbs and adverbs (72–74, 75–77)

count verbs and adverbs per construction and display the most frequent ones (79–81, 82–84)

identify the adverbs attested with both constructions (88–92)

generate a frequency table/`matrix` of the shared adverbs and their frequencies in both constructions (94–100, alternative: 102–106)

from that, compute the percentages of split constructions for each adverb (108–113)

visualize those percentages for each adverb (115–124)

identify the verbs attested with both constructions (128–132)

generate a frequency table/`matrix` of the shared verbs and their frequencies in both constructions (134–140, alternative: 142–146)

from that, compute the percentages of split constructions for each verb (148–153)

visualize those percentages for each adverb (155–164)

create a co-occurrence `table` of verbs and adverbs in the split construction with the most frequent adverbs on the left and the most frequent verbs on the right (168–175, useless)

create a `sort`ed frequency `table` of adverb–verb combinations (177–178)

*(continued)*

*(continued)*

go into the interim-results directory and create an empty frequency `table` called `all.lemma.freqs` for amalgamating all frequency tables (182–184)

`for`-loop (counter = `i`, over `dir()`): (186, –194)

  output a progress report (187–189) and `load` each frequency list file (190)

  add the current frequency list to `all.lemma.freqs` (191–193)

use `tapply` to `sum` up all frequencies grouped by lemma (196–200) and `sort` the result (201)

output lemmas whose observed frequency does not differ more from the frequency of the split construction by more than 5 percent (203–209)

output lemmas whose observed frequency does not differ more from the frequency of the non-split construction by more than 5 percent (211–217)

**Which aspects of the script are worth additional comment?** The only part of this script that differs from what we've already done before is the very last step. We have a frequency table of all lemmas (about 176,000 elements long) and we have a frequency of the split constructions, say 414. We first compute the absolute differences of the former and the latter:

```
abs(all.lemma.freqs-length(all.splits))
```

This vector of numbers is then compared to 5 percent of the number of split constructions (414, of which 5 percent is 20.7) with a logical expression, which will return `TRUE`s if the absolute difference between the lemma frequency and 414 is <20.7:

```
abs(all.lemma.freqs-length(all.nonsplits))·<·(length(all.
  nonsplits)*0.05)
```

That logical vector is used to subset `all.lemma.freqs`, the result of which is then `sort`ed.

## 5.4 Other Applications

We are now turning to a variety of case studies that involve applications that are not so easily grouped into (just one of) the previous sections on dispersion, frequencies, and concordancing or co-occurrence; thus, the following scripts are, I hope, good examples to showcase the variety and potential of corpus-linguistic methods and the different kinds of data you can use and/or explore.

### *5.4.1 Corpus Conversion: the ICE-GB*

You may have already engaged in some corpus conversion activities above – if you didn't have COCA and COHA that is. Here's another one: In this tiny assignment, we are going

```
[<#3:1:A> <sent>]
PU,CL(main,inter,intr,past)
 DISMK,FRM {Sorry}
 INTOP,AUX(modal,past) {could}
 SU,NP
  NPHD,PRON(pers) {you}
 VB,VP(intr,infin,modal)
  MVB,V(intr,infin) {start}
 A,AVP(ge)
  AVHD,ADV(ge) {again}
[<$B>]

[<#4:1:B> <sent>]
PU,NONCL
[<O> <EXTM(begin)> cry-of-outrage </O> <EXTM(end)>]
 DISMK,REACT {OK}
[<$A>]

[<#5:1:A> <sent>]
PU,CL(main,inter,intr,past,incomp,supersede)
 INTOP,AUX(do,past) {Did}
 SU,NP
  NPHD,PRON(pers) {you}
 A,AVP(ge)
  AVHD,ADV(ge) {not}
[<unclear>]
 INDET,?(ignore) {<unclear-words>}
[</unclear> <$?>]
```

*Figure 5.6* Three sentences from ICE-GB Release 2.

to process the files from the one-million-word British Component of the International Corpus of English (Release 2), a POS-tagged and syntactically parsed corpus, of which a very small section is exemplified in Figure 5.6.

The task is simple: We want to strip the file of all its annotation and obtain a one-sentence-per-line format as exemplified in Figure 5.7; we do that by recognizing two things: (1) sentences are tagged with "<sent>", and words are between curly brackets.

**What are the things we will need to do?**

- We load the rChoiceDialogs package, `exact.matches.2`, and set options so that we get warnings as soon as they arise (e.g., when we load a file whose encoding differs from what we expect to handle).
- We define a vector `corpus.files`; and an output directory (<_qclwr2/_output-files/05_18_icegb-output>) for the output files, which will look like Figure 5.7.
- In a `for`-loop, we load each file, `paste` it together into one long string with line break characters "\n" between vector elements, and split it up again at every occurrence of "<sent>" (discarding anything that `strsplit` might leave before the first "<sent>") into a vector `current.sentences`.

```
Sorry could you start again

OK

Did you not <unclear-words>
```

*Figure 5.7* The same three sentences after processing.

- We then create `current.words`, which contains the first two components of `exact.matches.2` when looking for all words in `current.sentences`, where words are defined as sequences of characters that have a "{" to their left, do not contain another opening curly bracket or a line break in them, and have a "}\n" to their right. You may wonder why we can't just define a word as something that has a curly bracket to the left and is not the closing curly bracket as in "(?<={)[^}]+". That's because there are things in the corpus like that shown in Figure 5.8.
- It of course all depends on how *exactly* you paste all lines together in the third step above and how *exactly* you define the regex to find all words, but such cases of opening angular brackets (not even closed in the same line) caused problems in my own first draft of the script. Once you're done findings words though, `current.words[1]` contains the words, `current.words[2]` their locations in `current.sentences`.
- We `paste` all words coming from the same sentence together with spaces in between them and output them into a file in the output directory that has the same name as the input corpus file.

**What are the functions we will need for that?** We use `rchoose.dir` and `dir` to define the corpus files and `dir.create` to create an output directory. We `for`-loop to load

```
   CJ,NP

    NPPR,AJP(attru)

     AJPR,AVP(inten)

      AVHD,ADV(inten) {really}

     AJHD,ADJ(ge) {blond}

    NPHD,N(com,sing) {hair}

 INTOP,V(intr,pres) {is}

 EXOP,EXTHERE {there}

 SU,NP

[<{> <[>]                                          ← ← ←

  NPHD,PRON(ass,sing) {something}

  NPPO,AJP(attrd)

   AJHD,ADJ(ge) {odd}

 A,AVP(ge)

  AVHD,ADV(ge) {there}
```

*Figure 5.8* Problematic (for us) uses of brackets in the ICE-GB Release 2

each file with `scan`, `paste` it together with `collapse="\n"`, and `unlist` the result of `stringsplitting` it on the occurrence of sentence tags. We then use `exact.matches.2` to find the words and their locations, and then we `tapply paste` to all the words in the same sentence (now with `collapse="·"`). Then we `cat` the results into a file in the output directory named with `paste0`, done.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_18_icegb.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs and dplyr packages, and source `exact.matches.2` (1–4)
set the warnings option to immediate warning (5) for the reason discussed in lines 6–9
define the vector `corpus.files` (11–12) and create the output directory (14–15)
`for`-loop (counter = `i`, over `corpus.files`): (17, –42)
  output a progress report: the name of `i` (18)
  load the corpus file `i` (20) and paste it together into one string (21–22)
  split the file up into sentences and discard stuff before the first sentence tag (24–27)
  find the words between curly brackets and their sentence locations (29–32)
  paste the sentences back together from the words and their (sentence) locations (34–37)
  output the sentences into an output file in the output directory (39–41)

**Which aspects of the script are worth additional comment?** None, because this script is very short and simple, but do check out the several excurses at the end: The first one shows how we could change part of the loop by using the argument `vectorize=FALSE` in `exact.matches.2`; the second one proposes an alternative to parts of the loop using dplyr's %>%; the third combines those two approaches to replace everything in the loop by some nicer-looking %>% syntax.

### *5.4.2 Three Indexing Applications*

In this section we will play around with something that doesn't really have much corpus-linguistic value per se, but it has a lot of practical value and, more importantly, is a good programming exercise: We will use R for indexing. Two applications will involve a small part of the first edition of QCLWR, the third will involve a paper of mine. Specifically, the first case study consists of (1) finding function names in a text version of a few pages of the proofs of QCLWR (first edition) and (2) creating and outputting an index for them.

**What are the things we will need to do?**

- We `source` `exact.matches.2` and load a text file (that is essentially just a few pages copied from the PDF proofs of the first edition of QCLWR and pasted into a text file).
- We `paste` them together in one long string, but then split that up again into a character vector at each occurrence of a page marker that's repeated on every page of the PDF proofs; this way, that character vector has as many elements as the (part of the) proofs had pages.

- Then we find lines with code in that character vector, namely lines that contain the R prompts ">" or "+" followed later by the line break indicator "¶".
- From those lines with code, we extract strings that are likely to be function names, namely sequences of letters, periods, and/or underscores before opening parentheses.
- Then we look for all the potential function names in the character vector with the manuscript text and collect their locations as the page numbers (of this manuscript excerpt).
- Finally, we output each function name followed by a tabstop followed by the page numbers (separated with ",·").

**What are the functions we will need for that?** We use `source` to load `exact.matches.2` and `scan` to load the text file <_qclwr2/_inputfiles/corp_indexing-1.txt> into R. We use `paste` to merge all elements into one long character vector and `unlist` plus `strsplit` to break it up into a character vector of pages. We then use `exact.matches.2` to retrieve lines with code and again to extract strings that might be function names. We then `for`-loop over the list of function names and use `grep` to find them in the character vector with the book pages and store them in a list – note how it helps here that `grep` only returns one position even if a function name is attested on one page multiple times (because an index entry lists each page on which a word occurs just once, too). In a separate second `for`-loop we then use `paste` to put page numbers together (separated by commas and spaces) and `cat` to output all results.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_19a_indexing.r>; read this side by side with the actual script in RStudio:

clear memory and source `exact.matches.2` (1–2)

load the text file with the part of the manuscript (5–8)

`paste` it together into one string (10–11 and then break it up again at the occurrence of the page marker (deleting things before the first page marker (13–17)

extract lines with R code (19–21) and extract strings that could be functions (23–29)

create a list to collect index entries (34)

`for`-loop (counter = `current.function`, over `function.candidates`): (35, –46)

  `grep` for the current function candidate (surrounded by word boundaries) in the pages of the book and store the locations of matches (i.e., page numbers) (36–45)

`for`-loop (counter = `i`, over `seq(index1)`): (48, –58)

  paste together the name of the index entry with a tabstop before the page numbers, which are in turn pasted together with commas and spaces between them and print them with line breaks between index entries (50–57)

**Which aspects of the script are worth additional comment?** The script as discussed so far is pretty simple, so no additional comments are required. However, in lines 62–72 and 76–84, I provide alternative solutions to what above is done very R-unlike with `for`-loops in lines 33–46 and 48–58: These alternatives use faster and more elegant functions from the `apply` family and don't require loops at all. These alternatives are heavily commented so check them out to better understand how one does things 'the R way' and to prepare for the next case study. Also, note that the above way of finding function names (stuff before opening parentheses) is not perfect because: it wouldn't find `head` in

`sapply(some.list,·head)` or `sum` in `tapply(x,·y,·sum)`), so before checking out my first attempt at how to do this in lines 89–103, why don't you try your hand at this?

The second case study is very similar to the previous one – the only differences are that (1) we now don't just index function names but essentially all word types in the manuscript, and (2) we now use the loopless ways to create and output the index. I therefore do not discuss this script – <_qclwr2/_scripts/05_19b_indexing.r> – here in much detail but only draw your attention to how the word types we use in the index are defined: A word type for the purposes of this script is defined as one or more characters of the following: hyphens, letters from *a* to *z* (small and capital), digits from 0 to 9, underscores, and periods (lines 22–26). These word types are then cleaned/pruned such that word types that do not begin with letters or numbers are deleted (lines 27–30). In addition, any word types that are integers or decimals are replaced by "NUM" to avoid index entries for many different numbers (lines 31–34). Finally, all unique non-empty character strings remaining are used for indexing (line 35), and the rest is as in the previous case study; check the detailed comments for how `lapply` and `sapply` are used to prepare the index and ready it for output (lines 40–49 and 51–54) respectively. Also, note a small excursus at the end where I show you some code to quickly do somewhat interactive index queries.

The third case study is again similar to the previous ones, but adds two little twists. One is that we're now loading a file with Windows's version of Unicode UCS-2LE and so it is particularly useful to load this file with `readLines(file(...))` and the encoding argument set to "UCS-2LE".

The other twist is more complex and addresses the fact that if an index term is mentioned on several successive pages, say 1, 2, 3, and 4, then indices usually don't print all of those but sum them up as "1–4". Thus, what we want to do this time is not output index entries like this: "1, 2, 3, 4, 6, 7, 9, 11", but like this: "1–4, 6–7, 9, 11".

Much of this script works pretty much as before, which is why I again abbreviate the discussion here. We use a different page marker now (namely the name of the publisher that shows up on every page) (lines 13–17), we change the paper to lower case (line 18) and get rid of the header and the footer of the paper (lines 19–22), and we define a few index terms manually (lines 24–27). We then create the index in the same way as before (with `lapply`), but now the second twist: merging successive page numbers.

Let's approach this by looking at the index term "(association measure|\\bam\\b)". We assign that to `qwe` to keep code shorter as I am discussing lines 40–53 – noting that of course (sigh) Linux and Windows use different alphabetical sorting so that index term is the first list element on Windows but the second on Linux . . . :

```
>·(qwe<-index[[ifelse(.Platform$OS.type=="windows",·1,·2)]])¶
·[1]··2··3··4··6··8··9·10·12·13·14·16·18·20·21·22·24·25·
  26·28·30·32
```

We want to change that to this:

```
2-4,·6,·8-10,·12-14,·16,·18,·20-22,·24-26,·28,·30,·32
```

But how? The first step uses the function `diff`, which computes the differences between neighboring values in a vector:

```
>·diff(qwe)¶
·[1]·1·1·2·2·1·1·2·1·1·2·2·2·1·1·2·1·1·2·2·2
```

That is, 3 – 2 = 1, 4 – 3 = 1, 6 – 4 = 2, etc., which is already a nice first step given how this indicates occurrences of adjacent page numbers (with the 1s). The next step is to define an object called `ranges`, which contains hyphens when the difference between two page numbers was 1, and commas otherwise:

```
>·(ranges<-ifelse(diff(qwe)==1,·"-",·","))¶
·[1]·"-"·"-"·","·","·"-"·"-"·","·","·"-"·"-"·","·","·","·","·"-"·"-
  ","·","·"-"·"-"·","·","·","·","
```

We then paste the numbers from `qwe` and the characters from `ranges` together into a character vector `all.in.one` of only one string, of which we then delete the final character:

```
>·(all.in.one<-paste(qwe,·ranges,·sep="",·collapse="·"))¶
[1]·"2-·3-·4,·6,·8-·9-·10,·12-·13-·14,·16,·18,·20-·21-·22,·24-
  ·25-·26,·28,·30,·32-"
>·(all.in.one<-substr(all.in.one,·1,·nchar(all.in.one)-1))¶
[1]·"2-·3-·4,·6,·8-·9-·10,·12-·13-·14,·16,·18,·20-·21-·22,·24-
  ·25-·26,·28,·30,·32"
```

Last step: We look for numbers (memorized as `\\1`) that are followed by a hyphen and a space followed by, and this is the crucial point, potentially more numbers or more numbers followed by hyphens (memorized as `\\2`) till the last sequence of numbers (memorizes as `\\3`) – i.e., we're looking for sequences of numbers with hyphens – and we replace them only by the starting and the end point of the sequence:

```
>·gsub("(\\d+)-·(\\d+-·)*(\\d+)",·"\\1-\\3",·all.in.one,·
  perl=TRUE)¶
[1]·"2-4,·6,·8-10,·12-14,·16,·18,·20-22,·24-26,·28,·30,·32"
```

How does this work? It finds the "2-·" (with the "2" memorized as `\\1`), followed by "3-·" (memorized as `\\2`), followed by the "4" (memorized as `\\3`), but then it takes all that out and puts \\1 (the "2") back in followed by a hyphen, followed by \\3 (the "4"), effectively conflating all intermediate ranges by retaining only the starting and the end point of a range. This logic is the applied within a loop to all index entries (lines 57–70).

If you are now thinking "How the h . . . would I ever figure that out myself?", then you're in good company: It took me quite a bit of time and playing around with stuff before I came up with this, where the main recognition of how to do it started with the realization that `diff` would make it possible to find adjacent numbers. But you're really

not expected to immediately be able to come up with such exotic solutions and I don't dare claim that my solution is good, elegant, or anything other than functional.

### *5.4.3 Playing With CELEX*

One very useful database for many research purposes is the CELEX database (Baayen, Piepenbrock, & Gulikers 1995; https://catalog.ldc.upenn.edu/LDC96L14), which provides phonological, morphological, syntactic information on many words in Dutch, English, and German. Figures 5.9 and 5.10 show you three lines for three different lemmas in English (here highlighted in bold); as you can see, the two files are aligned: row 26,198 refers to the same lemma both in the file containing phonological information (<EPL.CD>) and the file containing syntactic information (<ESL.CD>):

In this section we will load samples of these two files (<_qclwr2/_inputfiles/EPL_qclwr2.CD> and <_qclwr2/_inputfiles/ESL_qclwr2.CD>) into two text vectors `epl` and `esl` (lines 4–5 and 17–18 respectively) and split them up with `strsplit` at backslashes. Specifically, we will create a vector `words` with all word lemmas and a vector `pronuns` with all their pronunciations from `epl` (lines 12–14 and 15 respectively); then we create a logical vector `adjectives` that states for each lemma whether it is an adjective or not (lines 20–27). Note that `strsplit` requires four backslashes to split on one backslash: the third one says the fourth one is a literal backslash, the first one says the second one is a literal backslash (try `cat("\\\\")¶`), so that, of those two literal ones, the first one escapes the escape meaning of the second . . . (don't get me started!). Note also that the use of CELEX files in this section requires you to know which column in these files contains what kind of information; thus, look at the lines with `sapply` in the code file (14, 15, and 27), which tell you where you find words (column 2 of <EPL.CD>), their pronunciations in a format where every phoneme is represented by one and only one character (column 6 of <EPL.CD>), and parts of speech (column 4 of <ESL.CD>); for that as well as for the phonemic transcription or any other info, you would need

```
26198\linguistic\97\1\P\lIN-'gwI-stIk\[CVC][CCV][CCVC]\[lIN][gwI][stIk]
26199\linguistically\12\2\P\lIN-'gwI-stI-k@-
      lI\[CVC][CCV][CCV][CV][CV]\[lIN][gwI][stI][k@][lI]\P\lIN-'gwI-stI-
      klI\[CVC][CCV][CCV][CCV]\[lIN][gwI][stI][klI]
26200\linguistics\78\1\P\lIN-'gwI-stIks\[CVC][CCV][CCVCC]\[lIN][gwI][stIks]
```

*Figure 5.9* Three lines from CELEX, <EPL.CD>.

```
26198\linguistic\97\2\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\Y\N\N\N\N\N\N\N
      \N\N\N\N\N\N\N\N\N\N\N\N\N
26199\linguistically\12\7\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\Y\N\N
      \N\N\N\N\N\N\N\N\N\N\N\N\N\N
26200\linguistics\78\1\N\Y\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\N\
      N\N\N\N\N\N\N\N\N\N\N\N\N\N
```

*Figure 5.10* Three lines from CELEX, <ESL.CD>.

to consult the CELEX documentation, the phonemic transcription I make available in <_qclwr2/_inputfiles/CELEX_pronuncode.pdf>.

With these vectors in place, we now do some small searches just to explore the kinds of things that one can now do (in the code file, I also always provide the results you get if you run this code on the real CELEX database, not just the tiny excerpts here). To make this section worth your time, though, most code in this particular code file, <_qclwr2/_scripts/05_20_celex.r>, will not use the traditional R syntax with nesting of functions, but the %>% operator from the package dplyr (which we then obviously need to load). Here in the book, I will discuss just a few of them, but study the code file to see what else there is for you to explore.

The first example to be discussed involves the number of adjectives that have four syllables (lines 38–47). The traditional nested code is shown in line 40 as well as here, and is ugly and counterintuitive to read because you have to go from the inside out (starting with `pronuns[adjectives]`):

```
>·table(sapply(strsplit(pronuns[adjectives],·"-"),·length))["4"]¶
```

The nicer way using %>% can be read as "take the pronunciations of the adjectives, `strsplit` them up at the syllabification code (see Figure 5.9), to the result of that `sapply length`, count the resulting elements of that with `table`, and from that table extract the frequency of 4 (for 4 syllables)" – much nicer. The next two examples use similar approaches: One uses just a very simple `grep` to find pronunciations of adjectives beginning ("^") with schwa ("@"); the other also adds a `gsub` before that to delete syllabification markers first.

For the last example, we will pretend for a moment that the CELEX database doesn't have segmental information for its lemmas (which it does, in column 7 of <EPL.CD>). We want to find out how frequent different consonant cluster lengths are in English adjectives (ignoring syllabification for now). Since we pretend to not have segmental information, we need to create it ourselves: We create a vector with all consonant phonemes (`C.codes`) and one with all vowel phonemes (`V.codes`) and then create equally long (in number of characters) vectors with *C*s and *V*s, so that we can use `chartr` for transliteration like here:

```
>·chartr(V.codes,·Vs,·chartr(C.codes,·Cs,·"dIsplizIN"))¶
[1]·"CVCCCVCVC"
```

If we want to have lengths of consonant clusters, then all we need to do is `strsplit` on one or more occurrences of "V" (because that will leave only the sequences of "C"s, measure the lengths of each of the remaining list elements (by `sapply`ing `nchar` to them), and then `unlist` the consonant cluster length and count them (with `table`), and that is what lines 83–90 do. Two follow-ups on this: First, lines 98–104 and 106–110 then show you how you find the adjectives with the maximal cluster lengths. Second, recall that the CELEX database *has* segmental information and we just ignored it: therefore, lines 120–122, 124–127, and 129–132 give you the more elegant and more realistic results; side-remark: I call the result of lines 120–122 `seggies` and not `segments` not to be cute but because `segments` is a function in R.

### *5.4.4 Match All Numbers*

This section is another one of the very few that doesn't have the same structure. This is because this exercise is really only one (complicated) regular expression: The goal is to match all kinds of formats of numbers, which is something that can easily come up when you generate frequency lists of (large) corpora and want to avoid having potentially tens of thousands of frequency list entries that are really just different numbers – in such a situation, you would probably want just one entry "_NUM_" or something similar; thus, this is a realistic situation. Figure 5.11 gives you 40 character strings, of which 39 are different (versions of) numbers as character strings that you are supposed to be able to define with a single regular expression, and one character string that you must not match (obviously, that would be "not this").

Thus, I encourage you to not immediately look at the solution but go to a website such as http://regexr.com, enter all the strings above as test cases, and then try to craft your regular expression and see how well you can match all numbers. Bear in mind that such regex-testing websites usually do not require double backslashes, so if in R you'd write `\\d`, then there `\d` will be enough. Also think about all you need to cover: positive numbers (with and without a +) and negative numbers, numbers with and without decimal points, numbers with (multiple?) and without grouping separators, positive and negative exponentiation, values between 0 and nearly 1 that begin with a leading 0 or with a decimal point right away, and combinations of these things with different lengths, etc. When you think you have a good (enough) solution, or if you really hit a roadblock, then check out the answer key in <_qclwr2/_scripts/05_21_numbers.r>.

### *5.4.5 Retrieving Adjective Sequences From Untagged Corpora*

In this section we want to do something that may sound trivial given the previous sections: We want to retrieve sequences of two adjectives in the base form, which may not seem like a big deal given, for instance, that we already retrieved sequences of more than two words when we looked for split and non-split infinitives. However, this section adds an additional challenge: The corpus from which we want to retrieve sequences of two adjectives is not tagged. We will do two case studies; one will be based on the Chinese-Hong Kong data from the International Corpus of Learner English (ICLE), the other on the Brown corpus of the ICAME CD-ROM version 2 (as before, see the end of this section if you do not have access to either corpus). Thus, we will pursue the following three-step strategy: We will first retrieve all adjective tokens from the BNC World Edition; because

| | | | | |
|---|---|---|---|---|
| +1.234567e-4 | -0.12 | -1.234567e+4 | -123 | 12,345.67 |
| +1.234567E-4 | +.12 | -1.234567E+4 | +123.456 | -12,345.67 |
| 1.234567e-4 | .12 | not this | 123.456 | +12345.67 |
| 1.234567E-4 | -.12 | +1 | -123.456 | 12345.67 |
| -1.234567e-4 | +1.234567e+4 | 1 | +12,345 | -12345.67 |
| -1.234567E-4 | +1.234567E+4 | -1 | 12,345 | +12,345,678.9 |
| +0.12 | 1.234567e+4 | +123 | -12,345 | 12,345,678.9 |
| 0.12 | 1.234567E+4 | 123 | +12,345.67 | -12,345,678.9 |

*Figure 5.11* Thirty-nine character strings representing numbers to match and one not to match.

this number is going to be very high, we do not collect them in a vector that's growing in the iterations but use the logic from Section 5.2.8: We insert them into a very large vector defined before the loop using a counter.

After the loop, we'll pick the most frequent $n$ adjectives (something like $n$ = 2,000 for the learner data case study and $n$ = 5,000 for the Brown corpus case study) occurring in it and tag all occurrences of these forms in the untagged corpus files, and then we will retrieve sequences of two adjective tags and whatever they tag from these corpus files; with the ICLE corpus, we will actually save the tagged corpus files before we search them, with the Brown corpus example, we'll tag the files and immediately search them while they are still in memory. One comment here: We could just use Adam Kilgarriff's frequency list file or the results from the word-tag combination exercise in Section 5.2.8 to get adjectives to tag, and one could just use a tagged version of the ICLE or Brown corpus, but we will pretend we don't have access to any such resources and write a script from scratch just so you get some more practice that'll help you do these things when those additional resources are really not available; unlike several scripts above, we will write this one again such that it uses the BNC's XML annotation and, thus, the packages XML and xml2.

**What are the things we will need to do?**

- We load all required packages (rChoiceDialogs, XML, xml2), source `exact.matches.2`, and define `whitespace`.
- Following Section 5.2.8, we define a vector with corpus files as well as a long character vector for all adjective tokens with ten million slots (called `all.adjectives`), and we set a vector `counter` to 1.
- Within the usual kind of `for`-loop, we use the logic from Section 5.2.5 and let R detect whether it's running on a Windows system or not; depending on that, we load (and process) the file with functions from the package XML (if not on Windows) or xml2 (if on Windows).
- Either way, we retrieve all adjective lemmas and insert them into the relevant slots of `all.adjectives`; note that, while we are only interested in adjective types, we still need their frequencies because we want to pick the $n$ most frequent types – that's why we won't already get the unique types (although we *could* save a (growing) frequency table).
- After the loop, we clean up that vector and create as well as save a sorted frequency list of all adjective lemmas.
- We then find the $n$ = 2,000th highest frequency of all adjective frequencies and retain all adjectives with at least that frequency (to tag them in CH-HK ICLE).
- We define the CN-HK ICLE corpus files to be tagged and use an outer `for`-loop to load them, `paste` them into one string, and, if necessary, clean them up.
- With each file, we then use an inner loop over the ≈2,000 adjectives to look for each of them (surrounded by word boundaries and parenthesized for back-referencing) and put them back in with an adjective tag in front of them.
- After every one of our ≈2,000 adjectives has been tagged, we save the file in the same directory from which it was loaded (but with a different name!).
- We then define those annotated files as the new corpus files and load each of them within a loop.
- We look for sequences of two words tagged as adjectives; if we find any, we paste the file name in front of them and save them with 100 characters on each side into a vector.
- We output that vector into a .csv file.

**What are the functions we will need for that?** The script is one of our longest, but the good thing is most of it is familiar. The function `whitespace` still uses `gsub`, and we of course use `rchoose.dir` and `dir` to define the corpus files as well as `character` to define our long results vector `all.adjectives`. Our test for which operating system R is running on is done with `if` on `.Platform$OS.type`, and for Windows we use `read_xml`, `xml_find_all`, `tolower`, and `exact.matches.2`, whereas for other operating systems we use `xmlInternalTreeParse` and `xpathSApply`, and again `tolower`. We check whether there are any matches with `if` and use subsetting and `length` to insert adjective tokens into `all.adjectives`. We create our sorted frequency list of all adjectives with `sort(table(...))` and `save` it into an .RData file; we also then remove `all.adjectives` from memory with `rm` and, since that's a pretty large object, we then also initiate a process called garbage collection with `gc`, which can make R return unused memory back to the operating system.

We use `rchoose.files` to define the to-be-tagged corpus files, and `for`-loop over them with `scan`; each file gets `paste`d into one long string and cleaned, and in a second `for`-loop we `paste` word boundaries around the adjectives to tag, and replace any occurrences of them by them with a tag (using of course `gsub`) so that we can `cat` them with a different file name (using `gsub`).

Finally, we define the new tagged corpus files again with `rchoose.files` and `for`-loop over them again with `scan`, so that we can use `exact.matches.2` to look for adjective doubles. We check whether there are any matches with `is.null` and `if` there are matches, we `paste` the file name in front of them (with `sep="\t"`) and collect them. Once we're done with all corpus files, we add case numbers using `paste` to add a `se`quence from 1 to *n* to each case, and then we `cat` that into our final overall output file.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_22a_adjorder.r>; read this side by side with the actual script in RStudio:

clear memory, load the rChoiceDialogs, XML, and xml2 packages, and source `exact.matches.2` (1–5)

define `whitespace` as before (7–19)

define a vector `corpus.files` (21–22) and define a long collector vector `all.adjectives` to be filled and set its first position `counter` to 1 (24–26)

`for`-loop (counter = `i`, over `corpus.files`): (28, –65)

- output a progress report (29)
- if R is running on Windows (31, –44)
  - use functions from xml2 to load the file and find all adjective tokens (32–43)
- if R is running elsewhere (44, –52)
  - use functions from XML to load the file and find all adjective tokens (45–51)
- keep only adjectives consisting of letters and hyphens only and clean whitespace (54–57)
- if there were no matches, go do the next loop (59), otherwise insert the matches into `all.adjectives` (60–63) and set `counter` to the new position for new tokens to be added (64)

*(continued)*

*(continued)*

clean out the unused slots of `all.adjectives` (67), create a sorted frequency list of the adjective tokens (`all.advectives.freq`) (68), and save it (69)

put all adjectives as frequent or more frequent than the 2,000th most frequent adjective into a vector `adj.2.annotate` (71–72) and free up your workspace a bit with `rm` and `gc` (74–75)

define the to-be-tagged CNHK-ICLE files, which ideally would all be in one folder (79–80)

`for`-loop (counter = `i`, over `corpus.files`): (82, –109)

  output a progress report (83), load the file (85), `paste` it into one long string (86)

  clean the string up by replacing tabs, trimming `whitespace` (87–93)

  `for`-loop (counter = `j`, over `adj.2.annotate`): (95, –101)

    create a search expression by pasting word boundaries around each of the approx. 2,000 adjectives to annotate (96)

    replace all occurrences of that search expression by itself with a tag (97–100)

  after all replacements, `cat` the tagged corpus files into a new results .txt file (with a different name) (103–108)

delete the old corpus files (probably in a file explorer, but you can actually also (1) use `unlink` for that from within R or (2) in fact ignore them for now because we will know to look for tags and the regular corpus files don't have them so, while it would be inelegant and slow your script down (loading twice as many files as necessary), it would not affect the returned results (113)

define the to-be-tagged annotated CNHK-ICLE files (114–115) and prepare a collector vector `all.adj.doubles` for all adjective doubles (117–118)

`for`-loop (counter = `i`, over `corpus.files`): (120, –134)

  output a progress report (121), load and `tolower` the file (123), and look for sequences of two adjectives using the just inserted tags (124–126)

  `if` there are no matches, go to the `next` file . . . (128)

  otherwise `paste` the `basename` of the corpus file in front of the matches (129–131) and store the results of the current file with all others (132–133)

`paste` a `se`quence number in front of all matches to number them (135–137)

`cat` the results into a .csv file (139–142)

**Which aspects of the script are worth additional comment?** Not much, because, as mentioned above, this script is long but recycles so many things we've done before that it's not that hard. I do want to emphasize one seemingly small thing, however: Maybe you wondered in the final loop, when we were looking for adjective doubles, why there would be an `if`-conditional to test whether there are any results (in line 128). The reason is something I pointed out at the beginning of this chapter: Your script must be prepared to handle any outcome. Sure, if there are matches, you want to paste the file name in front of them (line 129–131), and sure, it's likely there will be matches in most files – but what if there are none and, worse even, what if you were looking for something really rare? Then, without this `if`-conditional, your output file would contain tons of rows with just the file names but no matches after

```
A01 0010 ... The Fulton County Grand Jury said Friday an investigation
A01 0020 of Atlanta's recent primary election produced "no evidence" that
A01 0030 any irregularities took place. ... The jury further said in term-end
A01 0040 presentments that the City Executive Committee, which had over-all
A01 0050 charge of the election, "deserves the praise and thanks of the
A01 0060 City of Atlanta" for the manner in which the election was conducted.
```

*Figure 5.12* The first six lines of <BROWN1_A.txt>.

them. Yes, you would still get all the matches – recall would be perfect – but precision would be bad because your few matches would be buried in many useless rows with file names. So, it is important to think through every step of the process: "What can happen here? A, B, and C. What do I want the script to do when A, what if B, what if C?", etc.

Let us now turn to the second case study in this section, in which we modify this script such that

- the corpus from which adjective doubles are extracted is the Brown corpus of written American English from the 1960s (using the folder BROWN1 from the ICAME2 CD);
- the adjectives we are looking for are now many more, namely approximately 5,000 rather than the 2,000 for the learner data;
- we are not going to save the tagged Brown corpus files, but perform the search right after the tagging – i.e., immediately in the same loop;
- we will add the sentence numbers to the matches and delete all tags before saving the final results file.

The Brown corpus files look like the example shown in Figure 5.12.

The good news is that the first part of the script doesn't change at all: With a single exception, lines 1–75 of <_qclwr2/_scripts/05_22a_adjorder.r> stay the same (so you can actually recycle the output file <_qclwr2/_outputfiles/05_22_adjorder1.RData> rather than generate the list of adjectives from the BNC from scratch again!). That one exception is that we increase the number of adjectives to annotate in line 72 – only after that do we have to make some other changes (to be found in <_qclwr2/_scripts/05_22b_adjorder_brown.r>:

define the to-be-tagged Brown corpus files, which ideally would all be in one folder (79–80)

since now we tag *and* immediately retrieve adjective doubles, we already define a collector vector `all.adj.doubles` for all adjective doubles (82–83)

`for`-loop (counter = `i`, over `corpus.files`): (85, –125)

output a progress report (86), load the file (88), delete line initial annotation (89–92), `paste` it together into one long string (94–95), `strsplit` it again at three spaces, and clean it with `whitespace` (97–100)

*(continued)*

*(continued)*

retain only non-empty character strings (102)

`for`-loop (counter = `j`, over `adj.2.annotate`): (104, –110)

create a search expression by pasting word boundaries around each of the approx. 5,000 adjectives to annotate (105)

replace all occurrences of that search expression by itself with a tag (106–109)

after all replacements, we use `exact.matches.2` to find sequences of two adjectives using the just inserted tags (112–115) – note, for once we store not just [[4]] (or [[1]]) but everything because we will need the locations of the matches later

`if` there are no matches, go to the `next` file – otherwise . . . (117)

`paste` the `basename` of the corpus file and the line numbers of the matches in front of the matches (118–121) and store the results of the current file with all others (122–123)

`paste` a `seq`uence number in front of all matches to number them and delete the tags (127–129)

`cat` the results into a .csv file (131–134); pay attention to how the heading is defined

I provide two output files: the regular .csv file that you can open with LibreOffice Calc or even a text editor (<_qclwr2/_outputfiles/22b_adjorder3_brown.csv>), but also a more nicely formatted OpenDocument spreadsheet (<_qclwr2/_outputfiles/22b_adjorder3_brown.ods>), in which I use some formatting that I find useful for subsequent annotation: font and color highlighting, frozen rows and columns (at C2), and I added the column F in which one can enter whether a match is really a match of two adjectives; for that, I use LibreOffice Calc's Validation tool: You can only enter "no", "yes", and "?" there, which is great if you have collaborators that do part of the annotation.

Now what if you don't have access to either the CNHK-files of ICLE or the version of the Brown corpus we used here? You may already guess the answer is another assignment for you: Write a corpus-conversion script that reads in the BNC file <HHV.XML> and makes it look like a file of the Brown corpus, i.e., as exemplified in Figure 5.12, and then (1) check your script against my relatively crude suggestion in <_qclwr2/_outputfiles/05_22c_BNC-HHVasBROWN.r> and (2) tweak the above script so that it would run on that HHV-as-Brown corpus file.

### *5.4.6 Type-Token Ratios/Vocabulary Growth:* Hamlet *vs.* Macbeth

The next application is concerned with type-token ratios (as a crude measure of lexical richness/repetitiveness) and vocabulary growth (see Youmans 1990; 1991), and we will use an example from literary linguistics to play with. Specifically, we will compare two plays by William Shakespeare – *Hamlet* and *Macbeth* – with regard to their type-token ratios. This section is relatively simple, which is why we will make it worth your time by (1) writing a function `ttr` that helps us compute type-token ratios and vocabulary-growth curves in the future, and (2) we will again explore a bit how we can change code from the traditional approach to the chaining approach with dplyr's great %>%.

Let us begin by discussing how we compute type-token ratios and vocabulary-growth curves using a small vector `tokens` as a 'corpus', something that I always recommend to get started on a new project: Create a data set realistic enough in its make-up but small enough to be seen on one screen, and start developing your code with that. Here we'll do that together, but that means you should already follow along with the script now. With the functions you already know well, it's very easy to compute a type-token ratio:

```
>·(tokens<-c("a",·"b",·"c",·"a",·"d",·"e",·"f",·"a",·"c",·"g"))¶
>·length(unique(tokens))·/·length(tokens)¶
[1]·0.7
```

But what is a vocabulary-growth curve and how do we compute it? It is a line plot that shows positions of all tokens of a vector on the *x*-axis (as in Section 5.1.2) and all type frequencies on the *y*-axis; in other words, each point's *y*-coordinate divided by each point's *x*-coordinate provides the point's type token ratio. For the above `tokens`, it would look like Figure 5.13.

The *x*-coordinates of each point are obvious: It's just the numbers from 1 to 10, one for every element of `tokens`, which means that, given what we said above, the *y*-coordinates are then the type-token ratios of each token slot multiplied by the number of tokens of each slot. Alternatively, the *y*-coordinates of all ten points are the number of unique types you have seen at each position if you've read all tokens from slot 1 to that position, i.e., what is here called `type.freqs`:

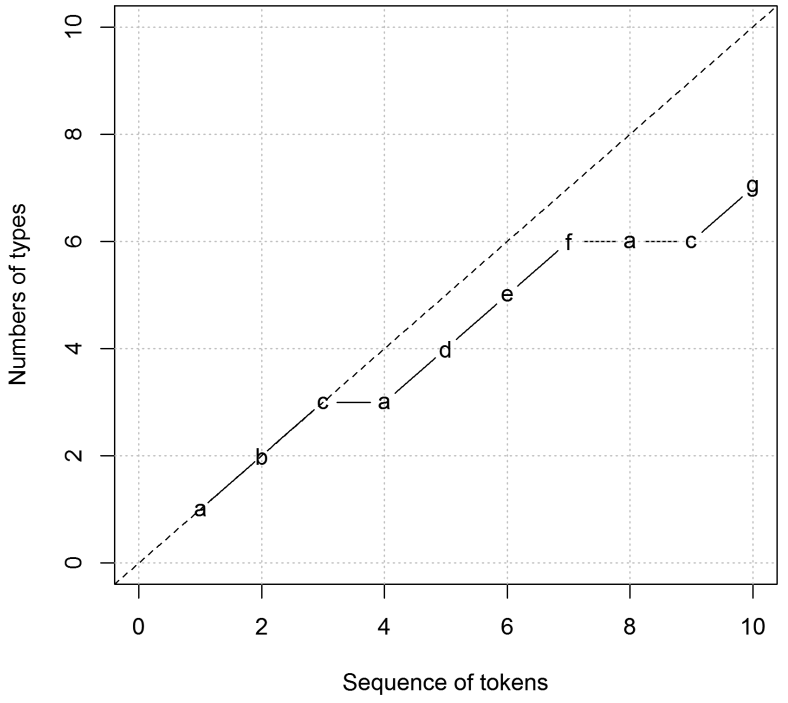


*Figure 5.13* The vocabulary-growth curve of `tokens`.

```
(type.freqs_y.coords<-c(1,·2,·3,·3,·4,·5,·6,·6,·7))¶
```

For instance, note how the rightmost value is 7, i.e., the type-token ratio of the whole vector (0.7) times the length of the vector (10). But how do we get that info? One approach that does the trick is this:

```
>·ttrs<-type.freqs_y.coords<-rep(0,·length(tokens))¶
>·for·(i·in·seq(tokens))·{¶
+····type.freqs_y.coords[i]<-length(unique(tokens[1:i]))¶
+····ttrs[i]<-y.coords[i]·/·i¶
+·}¶
>·type.freqs_y.coords¶
·[1]·1·2·3·3·4·5·6·6·6·7
>·round(ttrs,·3)¶
·[1]·1.000·1.000·1.000·0.750·0.800·0.833·0.857·0.750·0.667·0.700
```

For each position `i` in tokens, you essentially create a vector that includes all elements of `tokens` from 1 to `i`, then store the number of types (in `y.coords[i]`) and then divide by the `length` of this part of `tokens` and store that in `ttrs[1]`. It works, yes, but it is terribly slow: When I tried this approach with a vector `tokens` that contains 50,000 randomly sampled occurrences of the letters a–z on my laptop, it took 58 seconds – when I did the same with the function `ttr` we will write below, it took less than one second. Thus, as you have already seen above, it often pays off to find approaches that do things better when applied to large(r) data, i.e., approaches that scale. When I dealt with this problem the first time, I think I came up with five different solutions, but of the ones I tested, this one's the fastest. This is what we'll do: We first reserve a vector `type.freqs` that will change into the one we manually defined above:

```
>·(type.freqs<-rep(0,·length(tokens)))¶
·[1]·0·0·0·0·0·0·0·0·0·0
```

Then we determine all unique `types` . . .

```
>·(types<-unique(tokens))¶
[1]·"a"·"b"·"c"·"d"·"e"·"f"·"g"
```

as well as their first occurrences in the vector `tokens` using `match`:

```
>·(first.occs<-match(types,·tokens))¶
[1]··1··2··3··5··6··7·10
```

That is, "f" shows up in `tokens` for the first time in the seventh position. So far so good, but now the two lines that do the magic of creating the vector `type.freqs` we want: Remember that `type.freqs` only contains 0s so far; the first line now inserts a 1 into each slot that belongs to a new type in `tokens`, as you can see in the output. Second, if now every new type is represented with a 1 while every re-used, or `duplicated` type is represented with a 0, we can use `cumsum` to create a vector that contains, for each position, the cumulative sum of all 1s (i.e., all types that are new till there) and all earlier vector values:

```
>·type.freqs[first.occs]<-1¶
>·type.freqs¶
·[1]·1·1·1·0·1·1·1·0·0·1
>·(type.freqs<-cumsum(type.freqs))¶
·[1]·1·2·3·3·4·5·6·6·6·7
```

Why, then, is this approach so much faster when applied to a vector `tokens` with 50,000 sampled (with replacement) letters? Because this approach doesn't have to create 50,000 character vectors and check each of them for the number of `unique` types – it only determines the number of types (26), does a simple `match` for them, and assigns and then `sum`s up a bunch of 1s. (I admit that the speed benefit will vary as a function of the length and the type frequency of `tokens`, but it should never be slower than the loop-based approach.)

If you check out the script now, you will see how, after a small excursus, the above code is integrated into a function `ttr` (defined and heavily commented in lines 67–92); note that its most important output component is the fifth one called `TypesCounts`, which corresponds to the *y*-coordinates of a vocabulary-growth curve. This function minimally requires a vector of (word) tokens as its first argument, but if you want a plot such as Figure 5.13, too, you can set the additional argument `plot` to `TRUE` as well as provide `plot` with further arguments (in place of the ellipsis): lines 94–99 provide you with examples of that.

The remainder of the discussion below uses this function. But we want to add one little twist to the discussion: A vocabulary-growth curve is dependent – to some degree at least – on the exact order of the words in the corpus. This is unproblematic if you compute vocabulary-growth values on a play or a novel because what other orders of the words that are still the same play or novel would there be?! But if you compute vocabulary-growth values on a corpus consisting of many files, then the order of the files is typically arbitrary and makes the vocabulary-growth curve you plot a bit dependent on the usually unmotivated order of files that the corpus comes in. Thus an interesting approach to deal with this is to plot not just one vocabulary-growth curve for the corpus one is interested in but, say, 100 vocabulary-growth curves, one for each of 100 versions of the corpus in which the words have been randomly reshuffled, which is what we will add here.

**What are the things we will need to do?** With the function `ttr` now already defined, this script is relatively straightforward:

- We load the text file containing a *Hamlet* version from Project Gutenberg (<_qclwr2/_inputfiles/corp_hamlet.txt>); crucially, we need to load it with blank lines left intact (to make sure we can use them to find character names – *then* we discard them, check out the input file!); after cleaning *Hamlet*, we split it up into words.

- Then we do the exact same things with *Macbeth* (<_qclwr2/_inputfiles/corp_macbeth.txt>).
- After that, we apply our new function `ttr` to the vector of all words from *Hamlet* and add to it the `points` from the fifth component of the output of `ttr` applied to the words from *Macbeth*.
- For the resampling approach, create a list `collector` that has two components – one called `collector$HAMLET`, the other called `collector$MACBETH` – each of which is itself a list with 100 elements; each of these 100 elements in both `HAMLET` and `MACBETH` will collect the *y*-axis coordinates for one of the 100 randomly sampled word orders (i.e., what above was called `type.freqs`).
- We then do a `for`-loop in which we (1) randomly reshuffle the words of the original play *Hamlet*, (2) apply `ttr` to it, and (3) save the fifth component of the output into the `i`-th list element of `collector$HAMLET`; on each iteration, we do the same with *Macbeth*.
- We obtain the *y*-axis values for our plot from the elements of `collector`.
- We generate the *x*-axis values for our plot by `rep`eating, for each play separately, the number `seq`uence from 1 to the length of the play in words 100 times.
- Then we `plot` all resampled vocabulary-growth curves into one coordinate system, using different colors (and transparency effects) for the two plays.

**What are the functions we will need for that?** We use `rchoose.files` together with `tolower` and `scan` (note the argument `blank.lines.skip=FALSE`) to load each play. We then identify empty lines to delete with `which` and `nchar` as well as the immediately following lines with speaker names; also, we use `grep` to find lines with stage directions. Then, we apply negative subsetting to discard those lines from each play, before we use `unlist(strsplit(...))` to split each play up into words. After that, all we need to do is input the vectors of words into `ttr` to get our plot (we begin with the plot for *Hamlet*, then use `points` to add the one for *Macbeth*).

For the resampling, we use the functions `list` and `vector(...,·mode="list")` to create our list `collector`. In a `for`-loop, we randomly reorder each play with `sample`, apply `ttr` to it, and collect the *y*-coordinates from its fifth component. We then `unlist` the *y*-coordinates from the two parts of `collector`, and we use `rep(seq(...),·...)` to create *x*-coordinates. Finally, we plot the vocabulary-growth curves using periods as point characters (to minimize overplotting), define the color with `rgb(...)`, and add a `grid` and a main diagonal `abline`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_23_ttrs-vg.r>; read this side by side with the actual script in RStudio. (I am skipping (1) the part of the code that I used above to explain the basics of how to compute vocabulary-growth curves and (2) the definition of `ttr` because it basically just recycles that code in a function definition and is explained in detail in the code file; we therefore begin around line 100.)

load *Hamlet* (103–104)
identify empty lines in *Hamlet* (107)
identify character names in *Hamlet* (108)
identify lines with stage directions (110–112)
delete all those lines from the play (114–120) and split it up into words (122–125)

do the same sequence of steps for *Macbeth* (127–133)

excursus in lines 137–156: note how you can proceed from the loaded file to the words using the much simpler notation with %>%; note in particular how operators such as "==" and "-" are used as functions: run `4·%>%·"-"(0,·.)¶`, which is the same as `0-4¶` (i.e., `-4`) to understand how I create the negative indices

apply `ttr` to the words of *Hamlet* with `plot=TRUE` (160–163); then apply `ttr` to the words of *Macbeth*, too, and plot the `points` of the fifth component as well (164–166)

create the list `collector` for all the results from the 100 cases of resampling (171–177)

`for`-loop (counter = `i`, over `seq(no.of.resamples)`): (179, –185)

reorder the words of *Hamlet* randomly (182), apply `ttr` to them (181–183), and save the resulting *y*-coordinates into the `i`-th slot of `collector$HAMLET` (183)

do the same for *Macbeth* (184)

retrieve all *y*-coordinates (i.e., type frequencies) from `collector` with `unlist` (187–189)

generate all *x*-coordinates (i.e., token indices) (191–193)

open a graphics device for outputting .png graphs (195, –205)

plot the vocabulary-growth curves into the plot (196–204)

close the graphics device (205)

As you can see, *Hamlet* is much longer than *Macbeth*, but the vocabulary-growth curves overlap considerably for as long as *Macbeth* has words, suggesting that in terms of lexical diversity, the two plays are not that different.

**Which aspects of the script are worth additional comment?** Most of this is not particularly challenging programming-wise, so the only thing maybe worth mentioning is lines 164–166:

```
>·points(seq(mac.words),¶
+····(mac.ttr<-ttr(mac.words))[[5]],¶
+····col="blue")¶
```

You may wonder about the second line above: Why am I not just writing this?

```
>·points(seq(mac.words),¶
+····ttr(mac.words)[[5]],¶
+····col="blue")¶
```

And sure enough, this will create the same plot. The reason I use the first variant is essentially pedantry: In line 160, we did not just apply `ttr` to `ham.words` to plot; no, we also assigned the result of `ttr` to an object because we might do more with those results for *Hamlet* later. In line 165 I just make sure that we have a corresponding object `mac.ttr`

with the results for *Macbeth*. Thus, I apply `ttr` to `mac.words`, but make that an assignment. However, I want to put all of the results of `ttr` into `mac.ttr`, but plot only the fifth component, which is why I parenthesize the assignment (so as to not make `mac.ttr` just the fifth component).

### *5.4.7 Hyphenated Forms and Their Alternative Spellings*

This assignment was inspired by Kuperman and Bertram (2013): We will explore spelling alternations of forms that are hyphenated (e.g., *part-time*) with their potential alternants that involve separation by spaces (*part time*) or concatenated forms (*parttime*). Somewhat tongue-in-cheek: We're exploring the different frequency distributions of what in R could be paraphrased as in the following three lines:

```
>·paste("word1",·"word2",·sep="-")¶
>·paste("word1",·"word2",·sep="·")¶
>·paste("word1",·"word2",·sep="")¶
```

Specifically, we will want to determine how frequently the most frequent hyphenated forms in the BNC World Edition are also used in the other two spelling variants; in addition, we will make heavy use of dplyr's %>% operator to make parts of the script easier to parse. Overall, this script involves similar steps to the ones in Sections 5.3.4 and 5.2.8: In Section 5.3.4 we did a first pass through the BNC to find instance of *must* + V, and then we did a second pass to find all instances/frequencies of all the verbs after *must*. Here, we will do a first pass through the BNC to find all hyphenated forms, then we'll identify the most frequent ones (using the arbitrary and relatively high threshold of 1,000 occurrences in the corpus), and then we will do a second pass through the corpus looking for the spelling alternants of these frequent hyphenated forms. In Section 5.2.8, our going through the data involved creating a directory to collect frequency tables that then got merged in a second loop; here we will do the same.

**What are the things we will need to do?**

- We load packages dplyr and rChoiceDialogs and define a vector with the paths of the corpus files.
- We create a directory for interim-results files, namely the frequencies of all hyphenated forms.
- We load each corpus file, `tolower` it, trim it down to just the sentences, and look for words that contain hyphenated forms (but only forms with one hyphen such as *co-operation*, but not ones with two, such as *day-to-day*); we generate a frequency table of all the words we find and save it into the interim-results directory.
- Following Section 5.2.8, we define two long vectors for hyphenated forms and their frequencies (with one million slots) and we set a vector `counter` to 1.
- We then loop over all frequency tables in the interim-results directory and insert the words and their frequencies into the long vectors created in the previous step.
- We then discard all unused elements of the long collector vectors and `tapply` over them to sum up all frequencies of all hyphenated forms; we then pick all those forms with a frequency of 1,000 or more and their frequencies (in a vector `freq.hyphens`).
- We then create two frequency vectors `freq.spaces` and `freq.nothings` (1) whose contents are as many zeros as `freq.hyphens` has elements and (2) whose names are

the hyphenated forms with (a) spaces instead of hyphens and with (b) nothing instead of hyphens respectively.

- We then load each corpus file again, `tolower` it, trim it down to just the sentences, and then look for all alternate spellings of the hyphenated forms and add their frequencies to `freq.spaces` and `freq.nothings`.
- We compile the results in a data frame, `save` it, and explore a table of percentages of the spelling variants.

**What are the functions we will need for that?** Not much new here: We use `rchoose.dir` and `dir` as nearly always and `dir.create` to create the interim-results folder. We use a `for`-loop to `scan` and `tolower` the corpus files, and `grep` to get the sentences. We use `exact.matches.2` to find hyphenated forms and `sub` to clean away the tags and spaces; then we apply `table` to create a frequency list and then `save` it.

In the second part, we use `character` and `numeric` to create empty collector vectors to merge the hyphenated forms and their frequencies from all over the corpus, a `for`-loop to `load` each frequency list file, and, `if` there are hyphenated forms in the file, we merge them with subsetting, incrementing the vector `counter` on each iteration (as in Section 5.2.8). We use `nzchar` to discard all unused slots of the collector vectors and use `tapply` and `sum` up all forms' frequencies so we can use subsetting to pick all those with frequencies of 1,000 and more. We create collector vectors for the frequencies of non-hyphenated forms with `rep` and `length` and set them all to 0. Then we do a second `for`-loop over the corpus files, this time using `lapply` with `exact.matches.2` to look for all non-hyphenated equivalents to the most frequent hyphenated forms – see how nicely that avoids another loop? Then, we count their frequencies (with `sapply`, "`[`", and `length`) and add up their frequencies. Finally, we compile all results in a data frame and generate a `table` of `proportions` that lists for each form which spelling variant it prefers.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_24_hyphenation.r>; read this side by side with the actual script in RStudio:

clear memory and load the packages rChoiceDialogs and dplyr; also, source `exact.matches.2` (1–4)

define the vector `corpus.files` (6–7)

create a directory for interim-results files (9–10)

`for`-loop (counter = `i`, over `corpus.files`): (12, –30)

  output a progress report (13)

  load the current corpus file `i` (16), set it `tolower` case (17), extract the sentences from it (18), find hyphenated words in them (19–21), but just the exact matches (22), delete the opening tag (23) as well as any final spaces (24), and tabulate (25)

  `save` that `table` into an interim results file (26–29)

move into the interim-results directory (34–35) and define long collector vectors `all.hyphs` and `all.hyphs.freqs` for all forms and their frequencies respectively; set a vector `counter` to 1 (37–41)

`for`-loop (counter = `i`, over `dir()`): (43, –55)

*(continued)*

*(continued)*

- output a progress report (44) and load the `i`-th frequency list file (45)
- if there were no hyphenated forms in that file, go to the `next` one (47–49)
- otherwise, insert the observed forms and their frequencies into the next available slots of `all.hyphs` and `all.hyphs.freqs` (50–51, 52–53)
- increment the value of `counter` accordingly (54)

discard the unused slots in `all.hyphs` and `all.hyphs.freqs` (57–59)

sum up all forms' frequencies with `tapply(...,·...,·sum)` and make sure the result is a `table` (61–65)

`sort` that table and move up one directory into <_qclwr2/_outputfiles> (67–69)

store the forms that occur 1,000 times or more and their frequencies as `freq.hyphens` (73) and create analogous vectors for forms with spaces (`freq.spaces`) and with nothing between the parts (`freqs.nothings`) (74–81)

create search expressions that make sure you only find the relevant forms when they are words (and, for instance, not part of corr-tags (83–84)

`for`-loop (counter = `i`, over `corpus.files`): (86, –128)

- output a progress report (87)
- put into `current.sentences` (89) the result of scanning `i` (90), setting it to lower case (91), and extracting the sentences from it (92)
- add to the values of `freq.nothings` the frequencies of `search.expressions.nothings` in file `i` (94–101)
- add to the values of `freq.spaces` the frequencies of `search.expressions.spaces` in file `i` (103–110)
- excursus in lines 112–126: compare the traditional code to the one using %>%

compile all the results into a data frame, explore its top, and `save` it into an .RData file (130–136)

generate a `table` of percentages of the three spelling variants (138–140)

**Which aspects of the script are worth additional comment?** The only part of the code that may be worth a brief explanation is how we look for, say, all non-hyphenated forms with no spaces in lines 94–101:

```
>·freq.nothings<-freq.nothings·+¶
+····lapply(search.expressions.nothings,·exact.matches.2,·¶
+··········corpus.vector=current.sentences,·¶
+··········gen.conc.output=FALSE)·%>%·¶
+····sapply("[",·1)·%>%·sapply(length)¶
```

First, remind yourself that `freq.nothings` is a vector that already has many values (initially 0s) as there are forms to look for. We then use `lapply` to avoid a loop by taking

the search expressions for the forms without spaces or hyphens and make each one of them the first argument of `exact.matches.2` one time, while the next arguments that `exact.matches.2` receives are always the same ones: `current.sentences` becomes the vector to be searched, and the concordance output always gets suppressed. The result of that `lapply` execution is a list with as many elements as `search.expression.nothings` has elements. But we are not interested in all the info that `exact.matches.2` provides – we are only interested in knowing how many instances were found of each search expression, which is retrievable from the output components 1 and 4, which is why we pass that list on to (%>%) `sapply`, which extracts (`"["`) from each element of that list the first component (`1`). But as before, then we don't need the exact matches per se – we just need to know how many there are, which is why we pass that output on to (%>%) another `sapply`, which measures the `length` of each of these vectors containing exact matches. This returns a vector with as many elements as we looked for and, crucially, in the same order as in `freq.nothings`, which means we can just add the new vector to the current values of `freq.nothings` and use the fact that R will do pairwise additions of vector elements.

Apart from that, everything else should be clear because the main points of this script were dealt with in separate examples above and because the code file provides a detailed comparison of the use of %>% versus the nested-functions approach. There is one natural extension of this script I do want to tempt you with, though: If you look at the results, you will see that the spelling variants with spaces are, let's say, rare. That is for two reasons: First, because we picked only those hyphenated forms that occurred very frequently and so their spellings with hyphens may not exhibit much variation (anymore). The more important reason is the second: We constructed our search expression such that the versions with spaces still needed to be tagged as one word (check line 84 again), which is of course possible – after all, *in spite of* is tagged as a multi-word unit – but it really wasn't very likely to begin with that the kind of words we're looking for here would simply be tagged as one word. Thus, the natural extension that you might consider doing as an exercise (because it requires very few changes) would be to change the script such that it finds versions of hyphenated words written with spaces when these are tagged as two separate words. That is, we found *long-term* as a hyphenated form and we looked for something like this (not a regular expression "<w . . . >long term</w>" and didn't really find much, so now we try again, but this time looking for something like this "<w . . . >long</w><w . . . >term</w>". Why don't you see what happens when you change things like this?

### *5.4.8 Lexical Frequency Profiles*

As we are approaching the end of this chapter, it is only fitting that this case study is one of the most complex ones. It involves the notion of *lexical frequency profiles*, which is a corpus-based approach in applied linguistics concerned with the amount and the kind of vocabulary second/foreign-language learners use in their speaking and writing (see Laufer & Nation 1995 for an introduction; Meara 2005 for a critique; and Laufer 2005 for a response). It involves retrieving all words from one or more texts produced by learners and classifying all words as belonging to one of several word families and word frequency bins. The bins that have frequently been used come in the form of base word files that establish word families as well as frequency-based groupings of these word families; consider Figure 5.14, which represents the beginning of the first base word list (where the small arrow represents a tabstop and "·" and "¶", as usual, are a space and a line break).

```
A·1894¶
   →  AN·1¶
ABLE·4¶
   →  ABILITY·1¶
   →  ABLER·0¶
   →  ABLEST·0¶
   →  ABLY·0¶
   →  ABILITIES·0¶
   →  UNABLE·1¶
   →  INABILITY·1¶
```

*Figure 5.14* The first ten lines of <baseword1.txt>.

Nation's Range program (see www.victoria.ac.nz/lals/about/staff/paul-nation) is a Windows program, one packaging of which comes with currently 16 bins and can be used to identify vocabulary that is shared across texts, vocabulary that is used in a text but that is not part of a vocabulary list, etc.; see the instructions that come with Range and Paul Nation's website, and check out <_qclwr2/_inputfiles/dat_range-perl.txt>, which is the result of applying Range to the file <_qclwr2/_inputfiles/corp_perl.txt> (using only five base word lists). For instance, lines 14–22 provide the absolute and relative frequencies of types and tokens of the Perl file in each of the frequency bins, and lines 43–1,262 provide the word types found in each base word list (types found in base list 1, 2, . . . , 5): their range (in how many input files do they occur? Since we only have one input file, this is always 1) and the frequency with which they occur in the input file(s). In this case study we will write a script that provides some of the functionality of Range; first, we will load and process the base word lists to convert them into a format that is more R-like (we will use the version from www.victoria.ac.nz/lals/about/staff/publications/paul-nation/BNC-14000-and-programs-and-instructions.zip); second, we will compute lexical frequency profiles for the two Wikipedia entry files on Perl and Python so that we can determine whether the Perl file is different from the Python file in terms of its lexical difficulty (as operationalized by the proportions of words in the different bins); third, we will compute two (related) measures of lexical richness for each file, Yule's *K* and Yule's *I*, which are computed as shown in (11) and (12), where $M_1$ is the number of all word tokens a text consists of and $M_2$ is the sum of the products of (1) each observed frequency to the power of two and (2) the number of word types observed with that frequency (see Oakes 1998: 204): If one word occurs three times and four words occur five times, $M_2 = 3^2 + 5^2 + 5^2 + 5^2 + 5^2 = 109$.

(11) $$Yule's\,K = 10{,}000 \times \frac{M_2 - M_1}{{M_1}^2}$$

(12) $$Yule's\,I = \frac{{M_1}^2}{M_2 - M_1}$$

**What are the things we will need to do?**

- We load the package rChoiceDialogs, and we define our function `just.matches` as well as a function `yules.measures`, which takes as input a vector of word tokens and returns as output Yule's *K* and Yule's *I*.
- We define the paths to all 16 base word list files and create four vectors `BASWEWORDLIST` (for the bin), `WORD`, `FAMILY`, and `NUMBER`, which will later be merged into a data frame.
- In a loop, we load each base word list file (note: their encoding is ISO-8859-1), strip the empty lines that some of them have, and output a progress report.
- We add to `BASEWORDLIST` the number of the current file as many times as it has words.
- We add to `WORD` the current words.
- We add to `FAMILY` the current family words in several steps; this is easy for the cases where the word is also the family word: We determine which words are at the beginnings of lines and insert them into a temporary vector `FAMILY.curr`.
- For the words nested into a family word (that is not provided in the same line), it's not that easy: After the step above for the first file, this is what we have, and we need to make `FAMILY.curr[2]` "A" and `FAMILY.curr[4:10]` "ABLE" etc.:

```
> head(FAMILY.curr, 12)
 [1] "A"       NA        "ABLE"    NA        NA        NA
       NA        NA
 [8] NA        NA        NA        NA        "ABOUT"   "ABSOLUTE"
```

- Thus, we compute a matrix `position.differences` with every position of a word that is not a family word minus the position of a word that is. In the following excerpt, the family words are in the rows, the positions of words that are not family words (and for which we still need a family word are in the columns, and the cells give the differences. The circled 1 says that the fourth word ("ABILITY") is 1 row away from the last mentioned word that is also a family word, namely ("ABLE"):

```
> position.differences[1:12, 1:12]
          2   4   5   6   7   8   9  10  13  14  15  17
A         1   3   4   5   6   7   8   9  12  13  14  16
ABLE     -1  (1)  2   3   4   5   6   7  10  11  12  14
ABOUT    -9  -7  -6  -5  -4  -3  -2  -1   2   3   4   6
ABSOLUTE -10 -8  -7  -6  -5  -4  -3  -2   1   2   3   5
ACCEPT   -14 -12 -11 -10 -9  -8  -7  -6  -3  -2  -1   1
ACCOUNT  -24 -22 -21 -20 -19 -18 -17 -16 -13 -12 -11  -9
ACHIEVE  -28 -26 -25 -24 -23 -22 -21 -20 -17 -16 -15 -13
ACROSS   -38 -36 -35 -34 -33 -32 -31 -30 -27 -26 -25 -23
ACT      -39 -37 -36 -35 -34 -33 -32 -31 -28 -27 -26 -24
ACTIVE   -51 -49 -48 -47 -46 -45 -44 -43 -40 -39 -38 -36
ACTUAL   -60 -58 -57 -56 -55 -54 -53 -52 -49 -48 -47 -45
ADD      -63 -61 -60 -59 -58 -57 -56 -55 -52 -51 -50 -48
```

- That means we now only need to find the smallest non-negative number per column to get the position of the family word for the word in the column. For instance,

column 9 represents "UNABLE" (check <_qclwr2/_inputfiles/dat_basewrd1.txt>), the smallest non-negative number in that column is 6, which points to "ABLE" as the family word; similarly, column 13 represents "ABSOLUTELY" and the smallest non-negative number in that column is 1, which points to "ABSOLUTE" as the family word, etc. With these, we can now extract the `rownames` of this matrix to get the missing family words, which we insert into `FAMILY.curr`, which we then add to `FAMILY`.

- We extract the numbers from each line and add them to `NUMBERS`.
- After our loop, we compile all this into a data frame that now has a structure that allows us to process things more nicely (which is a technical term meaning 'more R-like').
- We define the paths to the two Wikipedia entry text files and create a list `lists.of.tables` to collect results for each file.
- In a loop, load each file, split it up into words, and create a sorted frequency list of the words so that you have all types and all their frequencies; from that you already compute Yule's lexical richness measures.
- For each word type, find its base word list file number; for instance, the word "an" is in base word list 1.
- For each word type, finds its family word; for example, the word "an" has as its family word the word "a".
- For the family word of each word token in the file, we `sum` up the frequencies of all its members; for instance, the word "an" is a member of the family 'headed' by "a" that comprises only "a" and "an". The words "a" and "an" occur 151 and 21 times respectively, which means we need to get 172 here for the family frequency of *both* "a" and "an".
- While we're still in the loop, we compile the results into a data frame that we save into an output file that has the name of the programming language studied and part of which looks like this; note rows 5 and 33:

```
>·result[c(1:5,·33),]¶
···WORDTYPE·WORDTYPEFREQ·WORDTYPEFAMILY·WORDTYPEFAMILYFREQ·WORDTYPELIST
1·····THE·····346·····THE·····346·····1
2·····PERL·····249·····<NA>·····NA·····NA
3·····OF·····166·····OF·····166·····1
4·····AND·····159·····AND·····159·····1
5·····A·····151·····A·····172·····1
33·····AN·····21·····A·····172·····1
```

- Also, we store in `lists.of.tables` for each programming language the frequencies with which types from different base word lists are attested (for easier plotting later).
- Finally, we generate and save in a file for each programming language a bar plot that provides the percentage of types in each Wikipedia entry that are in each base word list file; also, afterwards, we compute a chi-squared test to see whether the lexical frequency profile of the article on Perl is significantly different from the one on Python.

**What are the functions we will need for that?** The definition of `just.matches` is the same as multiple times above, and our function `yules.measures` requires `sum`, `table(table(...))`, "`*`", and `list`. After that we use `rchoose.files` to define the base word list files (and later the Wikipedia entries), which we read in with the right encoding. We use `gsub`, `grep`, and `just.matches` to define the list file numbers, words, and numbers, whereas, to define the family words, we use `grep` (to see which lines contain family words and which don't) and then `sapply` and `"-"` (yes, the regular minus) as well as `max` and `which.min` to compute the matrix `position.differences` excerpted above; at the end, we use `data.frame` and `write.table` to merge all results and save them.

As we turn to the Wikipedia entries, we `scan` them, `strsplit` them into words, and create a `sort`ed frequency list. We then use `match` to assign base word list numbers and family words to each word type of each Wikipedia entry; the more challenging part of summing the family frequencies and assigning them to each relevant type is done with `tapply(...,·...,·sum)` as so often before, and then we again use `match` and subsetting for the assignment part. We use `strsplit` and `unlist` to `paste` together file names for results, save all results per programming language into a data frame with `write.table` and then generate a `barplot`, which we annotate with `text` and save into a file with `png` and `dev.off`. At the very end, we use `chisq.test` for a chi-squared test for goodness of fit.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_25_lfp.r>; read this side by side with the actual script in RStudio:

clear memory, load the package rChoiceDialogs (1–2)

define `just.matches` (4–15)

define a function `yules.measures` that takes as input a vector of words and returns Yule's *K* and *I* as defined above (17–34)

define the vector `baseword.files` (36) and create collector vectors (38)

`for`-loop (counter = `i`, over `seq(baseword.files)`): (40, –103)

- output a progress report (41)
- load the current base word list file and discard empty lines (42–43)
- add the number of the base word list file to the required collector vector (45–47)
- add the words (which are either at the beginning of the line or after a tabstop) to the required collector vector (49–53)
- define a vector to collect family words (57–58)
- find the lines in the base word file that start with a family word and insert those into the collector vector (60–65)
- find the lines in the base word file that do *not* start with a family word and use them with `sapply(...,·"-",·...)` to subtract from them the lines that *do* start with a family word because you will always need the last family word for the current non-family word); this will create the matrix `position.differences` (67–84)
- set all values smaller than 1 in `position.differences` to the largest value in the matrix (86–88) so that it can never be the smallest value, and then collect

*(continued)*

*(continued)*

- the smallest value per column and its corresponding row name (90–96), which we then insert into the collector vector (97–98)
- use `just.matches` to find the line-final numbers per line (100–101) and store them in the relevant collector vector (102)

compile all collector vectors in a data frame and save it into a .csv file (105–108)

define a vector with the paths to the Wikipedia entry text files (112–113)

create a list to store the lexical frequency profiles, the frequencies of types per base word list (115–116)

`for`-loop (counter = `i`, over `text.files`): (118, –205)

- load the current Wikipedia entry and `toupper` it (like the base word lists) (119)
- split it up into words, discard empty elements, and create a sorted frequency list (121–127); for readability's sake, create short telling names for all word types and their frequencies (129–131)
- compute measures of lexical richness (133–134)
- for each word in the entry, look up its position in `WORD` and retrieve the number of its base word list file (which is in the same position in `BASEWORDLIST` as the word is in `WORD`) (136–139)
- for each word in the entry, look up its position in `WORD` and retrieve its family word (which is in the same position in `FAMILY` as the word is in `WORD`) (141–144)
- use `tapply` to `sum` up all words' frequencies by their family word (146–151), then order the summed frequencies by the order of occurrences in the vector of family words with `match` (153–157, read this carefully!)
- put everything into a data frame `result` (159–164)
- extract the name of the programming language from the file name (166–168), `paste` together an output file name (169), and save the data frame into a .csv file withthat name (170–173)
- store the lexical frequency profile in the list element for the current programming language (175–179)
- `paste` together an output file name for a `barplot` (182) and a table `list.percentages` stating how many word types are from which base word list (183–185), and generate as well as annotate that bar plot and save it into a `png` file (186–204)

compute a chi-squared test for goodness of fit to test whether the lexical frequency profile for the entry on Perl is significantly different from that on Python (209–214); explore how the observed frequencies compare to the expected ones and the residuals (215–218)

**Which aspects of the script are worth additional comment?** This script has some complex parts but most of them were already dealt with in detail – make sure you understand the explanation of lines 72–97 above (on how to identify family words) and the explanation in lines 155–157 on how we obtain and order the summed family frequencies for each word in the family. However, I do want to mention two small things in the plotting and

the statistical analysis. First, note how in the generation of the plots the function call `par("usr")` is helpful (lines 196, 200): That function call provides us with the actually plotted *x*- and *y*-axis limits in the order `xmin`, `xmax`, `ymin`, and `ymax`, which, as you see, allows you, for instance, to center plotting elements regardless of how the limits of axes change.

Second, I want to make it very clear what the chi-squared test tests. It does not test whether the frequency distributions of the words in the Perl and Python entries differ from each other (i.e., bidirectionally) – it tests whether the frequency distribution of the words in the Perl entry is different from the one in the Python entry (i.e., unidirectionally). In classes, students at first often do not differentiate between the following two analogous scenarios:

- the frequencies of words in corpus A and corpus B do not differ from each other;
- the frequencies of words in corpus A differ from those in corpus B.

But these two things are not the same! Here's the example I use to teach that difference: made-up frequencies of the words *horrible*, *horrifying*, and *horrid* in the Brown and the LOB corpus (lines 222–230 in the code file):

```
>·Brown<-c(13,·3,·1);·LOB<-c(9,·4,·6)¶
>·names(Brown)<-names(LOB)<-c("horrible",·"horrifying",·"horrid")¶
>·chisq.test(rbind(Brown,·LOB),·correct=TRUE)¶
#·X-squared·=·4.3439,·df·=·2,·p-value·=·0.114
>·chisq.test(Brown,·p=LOB/sum(LOB),·correct=TRUE)¶
#·X-squared·=·6.6879,·df·=·2,·p-value·=·0.0353
```

The first chi-squared test tests the first scenario, the second chi-squared test tests the second, because it determines whether the frequencies in the Brown corpus differ from those expected from LOB, but *not* also vice versa. Note how the two tests return very different *p*-values so it's important you understand that these two are different hypothesis tests.

### *5.4.9 CHAT Files 1: Eve's MLUs and ttrs*

In this assignment and the next, we turn to a completely different corpus format. After dealing with unannotated files, lots of XML examples, a bit of SGML, tabular versions of the COCA/COHA corpora, the Brown and the ICE-GB formats, we are now turning to the CHAT format that is widely used in language acquisition corpora, but also others. These case studies are interesting because they showcase once more how R allows you to handle data and perform analyses that most normal software cannot handle or do.

Most widespread corpus-linguistic software applications require that all information concerning, say, one particular sentence is on one line. In its most trivial form, this is reflected in the BNC by the fact that an expression such as "Wanted me to." would be annotated in one line as in (13) and not in, say, three as in (14). The latter format can be just as informative, but you may already guess that if you wanted to add further annotation to the expression "Wanted me to.", reading and processing all the information in the different lines may quickly become unwieldy.

(13) <w c5="VVD" hw="want" pos="VERB">wanted </w> <w c5="PNP" hw="i" pos="PRON">me </w>· <w c5="TO0" hw="to" pos="PREP">To </w> <c c5="PUN">.</c>

(14) Wanted·me·to.

VVD PNP TO0 PUN

want me to.

The following is an excerpt from the file <eve01.cha> from the Brown data of the CHILDES database (Brown 1973; http://childes.psy.cmu.edu/data/Eng-NA-MOR/Brown.zip). I only give a few lines of the header as well as the first two and one later utterance. The lines with header information begin with "@". The lines with utterances – the so-called utterance tier – begin with "*" followed by a three-character person identifier ("CHI" for the target child, "MOT" for the mother of the target child, etc.) plus a colon and a space. Finally, the lines with annotation begin with "%" and a three-character annotation identifier ("mor" for "morphosyntax/lemmatization", "spa" for "speech act", "int" for "interaction", etc.). Words and annotations are separated by spaces, and the annotation for one word on the mor tier consists of

```
@UTF8¶
@PID:   →   11312/c-00034743-1¶
@Begin¶
@Languages:   →   eng¶
@Participants:   →   CHI·Eve·Target_Child·,·MOT·Sue·Mother·,·
COL·Colin·Investigator·,·RIC·Richard·Investigator¶
@ID:   →   eng|Brown|CHI|1;6.|female|||Target_Child|||¶
[…]¶
*CHI:   →   more·cookie·.·[+·IMP]¶
%mor:   →   qn|more·n|cookie·.¶
%gra:   →   1|2|QUANT·2|0|INCROOT·3|2|PUNCT¶
%trn:   →   qn|more·n|cookie·.¶
%grt:   →   1|2|QUANT·2|0|INCROOT·3|2|PUNCT¶
%int:   →   distinctive·,·loud¶
*MOT:   →   you·have·another·cookie·right·on·the·table·.¶
%mor:   →   pro|you·v|have·qn|another·n|cookie·adv|right·
            prep|on·art|the¶
        →   n|table·.¶
%gra:   →   1|2|SUBJ·2|0|ROOT·3|4|QUANT·4|2|OBJ·
            5|2|JCT·6|2|JCT·7|8|MOD·8|6|POBJ¶
        →   9|2|PUNCT¶
%trn:   →   pro|you·v|have·qn|another·n|cookie·
            adv|right·prep|on·det|the·n|table·.¶
%grt:   →   1|2|SUBJ·2|0|ROOT·3|4|QUANT·4|2|OBJ·
            5|2|JCT·6|2|JCT·7|8|DET·8|6|POBJ¶
        →   9|2|PUNCT¶
```

*Figure 5.15* Excerpt of a file from the CHILDES database annotated in the CHAT format.

a POS tag, followed by "|", the lemma of the word and, if applicable, morphological annotation; see Figure 5.15.

Not only is the analysis of such corpora more complicated because the information concerning one utterance is distributed across several lines/tiers, but in this case it is also more complex to load the file. If you look at the above excerpt, you will see that the compilers/annotators did not complete every utterance/annotation within only one line: The morphological annotation of the utterance "you have another cookie right on the table". stretches across two lines with the second line being indented with tabstops, so if we just read in the file as usual, we cannot be sure that each element of the resulting vector is a complete utterance. Also, we cannot `scan` the file and specify a regular expression as a separator character (e.g., "[@%*]"): `sep` only allows single-byte separators. Thus, we have to be a little more creative this time and will proceed in a way inspired by our treatment of the ICE-GB corpus in Section 5.4.1.

What is the current task? We want to explore data of one of the most widely studied children acquiring English, Eve from the Brown (1973) corpus. In particular, we want to do two things: (1) compute MLU values (mean lengths of utterance values) for each file in Eve's data, and (2) compute type-token ratios of every utterance of each file in Eve's data and then determine whether, as one would expect, both increase over time. That in turn raises another issue: Developmental data often get plotted such that the variable representing time goes on the *x*-axis, but the way in which ages in these corpus files are represented is as follows: "1;6." (see the @ID row in Figure 5.15) means "1 year, 6 months, and 0 days" and "2;3.4" would mean "2 years, 3 months, and 4 days". That means we would ideally have some kind of function that can take the corpus data format as input and turn it into a decimal that we can then easily plot. That is, "1;6." or "1;6.0" should return 1.5, and "2;3.4" should return something very close to 2.25. Thus, we begin this assignment with a function called `age.converter`, which takes a CHILDES date string as an argument and then:

- splits the argument up at anything that's not a digit, unlists the result, and turns it into a numeric vector;
- if that numeric vector has only two parts, it adds a 0 as a third one; and
- returns the first part plus the second part divided by 365.

```
>·age.converter("1;6.")¶
[1]·1.5
>·age.converter("2;3.4")¶
[1]·2.260959
```

It works – in fact, the function `age.converter` I provide for you in the script can do more: It can also do the reverse, i.e., take a number as an argument and then compute a CHILDES date from it, and the function decides what to do automatically. If it gets a character string, it does the above, but if it gets a number, it does the reverse:

```
>·age.converter(1.5)¶
·Year·Month···Day
```

```
····1·····6·····0
>·age.converter(2.260959)¶
·Year·Month···Day
····2·····3·····4
```

So, look at the definition of the function and see how it does it. Armed with `just.matches` and `age.converter`, let's get started. **What are the things we will need to do?**

- We load the packages dplyr and rChoiceDialogs, and we define the functions `just.matches` and `age.converter`.
- We define the paths to all 20 files for Eve and we create (1) two lists `lus` and `ttrs` for utterance lengths and type-token ratios for every utterance for every file; and (2) two vectors `mlus` and `mttrs` for *mean* lengths of utterances and *mean* type-token ratios per file.
- We load each CHAT file and, to handle the fact that utterances and/or their annotation tiers may use up more than one line, `paste` it together into one long string (with a `collapse` argument that is not attested in the file so it marks where there were line breaks before the merging), delete whitespace after these previous line breaks, and then split the file up again at the collapse argument.
- Then we retrieve only the child's utterances, delete the line-initial annotation (check Figure 5.15 again), and replace annotation between square brackets with a space.
- We delete everything else that's not a letter or a space and then split the remaining vector up at any occurrence of 1+ spaces (obtaining a list), and retain only those utterances that have at least one word character in them.
- We then determine the lengths of all utterances and save them into the relevant slot of `lus`, plus we determine the type-token ratios of all utterances and save them into the relevant slot of `ttrs`.
- We compute the mean length of all utterances per file and store it in the relevant slot of `mlus`, plus we compute the mean type-token ratio of all utterances per file and store it in the relevant slot of `mttrs`.
- We then extract the name of the child and her age from the header and use it as the name for the current parts of `lus`, `ttrs`, `mlus`, and `mttrs` (so that we have it when we need it).
- After the loop, we inspect summaries of our collector lists and vectors.
- We compute a `shapiro.test` on each element of `lus` to determine whether the utterance lengths are normally distributed because that could affect our choice of statistical tests (even though it is beyond obvious a priori that they won't be normally distributed).
- We then compare each file's utterance lengths to every other file's utterance lengths using a *U*-test; since this involves making $20 \times 20 - 20 = 380$ tests, we do that with two nested loops and store the resulting *p*-values in a matrix (we take the negative log of the *p*-values for ease of representation).
- Finally, some visualization: First, we plot both mean lengths of utterances and mean type-token ratios against time (here we need `age.converter`!) into one plot; note first that it is really only going to be a nice plot if we have two *y*-axes: a left one for `mlus`, which ranges, say, from 1 to 5, and a right one for `mttrs` which ranges from 0 to 1. Note, second, that R will only use the *y*-axis values from the left *y*-axis for plotting, which means you must rescale the values you want to display with regard to the right *y*-axis to the range of the left *y*-axis (and a real nerd would of course write a function for that for later applications).

- As a second plot, one might be interested in exploring the finer resolution of `lus` and `ttrs`; for instance, we can plot type-token ratios against utterance lengths in Eve's first and last recording (to see whether things changed), or we could plot ecdf curves of the lengths of utterances.

**What are the functions we will need for that?** The definition of `just.matches` is the same as above, and the only new function we need to define `age.converter` is the very simple function floor: `?floor¶` (plus, at this point, we only need one of the two capabilities of `age.converter`, from string to number). After that we use `rchoose.files` to define Eve's corpus files and the function `vector` (with and without `mode="list"`) to define collector structures. We then load each corpus file with `scan`, merge it with `paste(...,·collapse=...)`, `gsub` line-initial spaces away and replace them by spaces, `strsplit` the file apart again, `unlist`, and do further cleaning with multiple `gsub`s, before we `strsplit` up all utterances into words and retain (with `grep`) only those with at least one word character.

After that we store the results we obtain by `sapply`ing functions to the list to count `length`s and compute type-token ratios (with an anonymous/inline function); we compute a mean type-token ratio using all types and tokens, and a mean of all lengths of utterances with `mean`, but to avoid the effect that outliers might have we use the argument `trim=0.05` to discard both the smallest and the largest 5 percent of the data. We use `just.matches` to extract from the file's header the name of the child and her age and `paste` together `names` for all vector and list elements. We summarize our collector vectors with `sapply` and `summary`.

We then `sapply` the function `shapiro.test` to all utterance lengths of each file and subset their $p$-values to see that, surprise, surprise . . . none are normally distributed. We then use two nested `for`-loops to compute `wilcox.test`s on every combination of elements of `lus`, and store their negative $\log_{10}$ $p$-values in a large 20 × 20 `matrix` called `tests` (20 because we have 20 corpus files).

Finally, we visualize the data. We apply `age.converter` to Eve's age strings, use `par(mar=...)` to make more room for our right $y$-axis label, and use `plot` to create a scatterplot of `mlus` against `ages`. We define each `axis` manually and use `mtext` to create axis labels. We use `lines` to add the type-token ratio plot and add smoothers with `lines(lowess(...))`. For the comparison of the first and last recording, we just generate regular scatterplots of type-token ratios against `mlus`, but we `jitter` the points to avoid overplotting and add the usual `grid`. Finally, we use `plot(ecdf)` for the ecdf comparison plot.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_26_CHILDES-mlus.r>; read this side by side with the actual script in RStudio:

clear memory, load the packages dplyr and rChoiceDialogs (1–3)
define `just.matches` (5–16) and `age.converter` (18–33)
define a vector `chat.files` with the corpus files (35–36)
create collector lists (38–39) and vectors (41–42)
`for`-loop (counter = `i`, over `seq(chat.files)`): (44, –122)
  output a progress report (45) and `scan` the current chat file (47)

*(continued)*

*(continued)*

- `paste` it together in one string with a `collapse` argument unattested in the string (49–50)
- replace whitespace before that `collapse` argument (i.e., after line breaks) with a single space (52–54), then split the string apart again at the `collapse` argument and `unlist` (56–57)
- find the child's utterances (59–62), delete its line-initial annotation (64–66), and replace its square-bracketed annotation with spaces (68–70); also delete everything that's not letters or spaces (72–74)
- split the file up at 1 or more spaces (76–78), and from that list retain only elements that contain at least one word character/letter (80–84)
- excursus: how to do all this more elegantly with dplyr's %>% (86–97)
- compute all utterance lengths by `sapply`ing `length` to each element of `current.chat.file.10` (99–100)
- compute all utterances type-token ratios by `sapply`ing an anonymous function (recall the example in Section 3.10) to each element of `current.chat.file.10` (101–104)
- compute the trimmed mean of the utterance lengths and one overall type-token ratio for the whole file (106–109)
- extract the name of the child (111–112) and her age (113–114, 115–117) and make the concatenation of the two `names` of all current elements of collector structures (119–121)

inspect the results (124–125, 127–128)

compute separate Shapiro–Wilk tests on all utterance lengths of each file (134–135) and check how many are significant (by extracting the *p*-values that are always in the second of four slots of the list that `shapiro.test` returns, check `str(shapiro.test(1:5))¶`) (136–141)

generate a `matrix` called `tests` with `NA`s that will collect all *p*-values from all pairwise tests of files' utterance lengths (143–148)

`for`-loop (counter = `i`, over `seq(lus)`): (149, –161)

- `for`-loop (counter = `j`, over `seq(lus)`): (150, –160)
  - if a file would be compared to itself (151), iterate immediately (152–153)
  - otherwise store in slot `tests[i,j]` the $-\log_{10}$ (154) of the *p*-value of a *U*-test comparing file `i` to `j` (155–158)

inspect the results (162–165)

generate the scatterplot of MLUs and type-token ratios as a function of time: compute numeric age values from the age strings (167) and tweak the margins of the plotting area by increasing the right margin (168)

plot *m*s to represent the MLUs (169–171), define the bottom *x*-axis (172–173), the left *y*-axis (174–175), and the right *y*-axis, where the tickmarks' numbers are rescaled (176–177)

add plotted *t*s to represent the type-token ratios but rescale them to the values of the left *y*-axis that was used to create the coordinate system in the first place: we

- don't plot `mttrs`, but 1 + 4 × `mttrs` (178–180), then we add smoothers and a `grid` (181–183)

restore the normal plotting margins (184)

set up a plotting window with one row and three columns (189)

- into the first panel, plot a scatterplot of jittered type-token ratios against jittered MLU values of the first recording (190–195)
- into the second panel, plot a scatterplot of jittered type-token ratios against jittered MLU values of the last recording (196–201)
- into the third panel, plot ecdf curves of utterance lengths in the first and last recording reusing the colors from the first two panels to facilitate interpretation (202–205)

restore the standard plotting window arrangement (206)

**Which aspects of the script are worth additional comment?** Most aspects of this script in terms of loops and text processing were not particularly difficult – there were many operations in the loop, but simple ones. The position of this script here at the end is largely due to the more complex statistical and visualization issues such as the 380 significance tests (lines 146–162) and the code for some of the plots, which involves many different tweaks; make sure you go over all this thoroughly line-by-line so you see what each one does.

Two suggestions at the end: First, you should try to write a function `right.y.axis`; it might take as arguments the tickmarks of the left *y*-axis (because you want to have corresponding tickmarks on the right side) and the minimum and maximum right *y*-axis values (because that will determine the range of values you might want to label on the right), and it might then return the right *y*-axis tickmarks and/or *y*-axis values that are rescaled from the right *y*-axis to the left for plotting. Second, we have proceeded in a slightly simplistic manner here because the only way we tried to make sure that 24-word utterances in which Eve repeated two words 3 and 21 times do not distort our data too much is by using a trimmed mean. One way of being more careful about such things could be to include a check in the right place that would either flag or omit such cases – why don't you make that a practice assignment?

### *5.4.10 CHAT Files 2: Merging Multiple Files*

The final case study is again on CHAT files, but this one involves more of a data management task, which then makes it possible to do linguistically interesting searches. Specifically, in order to deal with multiple files at the same time, we want to write a script that reads in multiple CHAT files and outputs a list or a data frame with all utterances in the rows and all annotation tiers in the columns; this will allow us to perform nice searches with regular expressions on multiple tiers to identify the rows where different

| CASE | FILE | NAMES | DATE | UTTERANCE | ADD | COM | EXP | EXP_2 |
|---|---|---|---|---|---|---|---|---|
| 18 | ifamdl02.txt | LID | 00/09/1979 | che / è diventato famoso //$ bello <que' gattino / prop> | NA | NA | famoso is a reconstructed word | NA |
| 20 | ifamdl02.txt | LID | 00/09/1979 | quande / e gli dico / Duna / dico / se tu ti metti seduta> | NA | NA | straordinario is a reconstructed word | they both laugh |
| 51 | ifamdl03.txt | LOR | 00/12/2000 | [<] <vero ? telefonica> //$ | NA | LOR laughs | NA | NA |
| 60 | ifamdl03.txt | NIC | 00/12/2000 | un vaso //$ | NA | NIC doesn' t answer to LOR | NA | NA |
| 61 | ifamdl03.txt | LOR | 00/12/2000 | si / e ci vo' un' ora xxx / porto una pianta di fiori //$ | NA | NA | NA | NA |
| 72 | ifamdl04.txt | ART | 00/01/1998 | mentre / altre borse / le chiamano a filetto //$ a borda> | turning to one of his workers | NA | NA | NA |
| 76 | ifamdl04.txt | DAN | 00/01/1998 | hhh$ | NA | NA | laugh | NA |

*Figure 5.16* Intended output of the case study.

search results intersect. In other words, the input will be CHAT files, the output will be what is shown (with some nicer formatting) in Figure 5.16.

We will use three excerpts from files from an early version of a part of the CORAL-ROM corpus, which are not in Unicode format (so we'll have to make sure we import them correctly); in addition, these files can have multiple instances of the same tier all of which we need the script to recognize and retain (note that the second row of the data in Figure 5.16 has two EXP entries and, thus, needs to have two EXP columns); finally, these files do not have an @ID line with an age in them, so we will use the @Date line instead as an identifier.

**What are the things we will need to do?**

- We load the package rChoiceDialogs, and we define the paths to the three example files; also, we set up a list, `chat.files.list`, that will collect all data from the files (in different parts that will later become columns of the data frame in Figure 5.16), and we set a counter `utt.counter` to 0, which will increase with each corpus line that contains a new utterance.
- We load each chat file (using `readLines` to set the encoding to "ISO-8559-1") and, to handle the fact that utterances may use up more than one line, `paste` it together into one long string (with a `collapse` argument that is not attested in the file so it marks previous line breaks), delete whitespace after these previous line breaks, and then split the file up again at the collapse argument (you will recognize this part from the previous case study).
- We then extract from the corpus file all lines with utterances (those beginning with an asterisk) or annotation (those beginning with a percent character); also, we extract the recording date from the header.
- With a loop, we then access each of these lines of the corpus file and extract its first character ("*" or "%") as well as the three characters after that (which will either contain the code for the speaker or the name of the annotation tier), and the rest of the line, which will be the speaker's utterance or one aspect of its annotation.
- If the current line is a new utterance, then we increment `utt.counter` by 1 and set a vector `tier.counter` to 1, which will help us distinguish columns with the same names (like EXP in the output above); also, we store in parts of `chat.files.list` the file name, the name of the speaker, the date retrieved above, and the utterance itself.
- If the current line is not a new utterance, then it is annotation of an utterance that has already been dealt as discussed above, so we check whether there has already been some annotation of this type for this utterance – if not, we just add the current line content to the component of `chat.files.list` that collects the kind of annotation we found here. If this type of annotation *has* already been processed for this utterance, then we have a situation like the one with EXP in Figure 5.16 so we increment the tier counter by 1 and paste that new number together with the annotation name so that it can become a new element of `chat.files.list`, where we now store that second (or third, etc.) instance of this annotation for this utterance – hence the "EXP_2" in Figure 5.16.
- After the loop, we reorder the elements of `chat.files.list` alphabetically (so that all cases with numbered annotations are listed adjacently, but we still make sure that the components with the file name, the date, the speaker, and the utterance come first.
- We convert `chat.files.list` into a data frame that we then save into a .csv file.

**What are the functions we will need for that?** We use `rchoose.files` to define the three corpus files and the function `list` to define the one collector structure. We then load each corpus file in a `for`-loop with `readLines(file(...))`, merge it with `paste(...,·collapse=...)`, `gsub` line-initial spaces away and replace them by spaces,

`strsplit` the file apart again, `unlist`, and retain (with `grep`) only those with utterances or their annotation.

In a second `for`-loop for each line, we use `substr` to extract speaker/annotation names etc. and then use an `if`-conditional to distinguish utterances from annotation. If we have an utterance, we just fill relevant slots of our collector list, but if we have annotation, we use a second `if`-conditional with `length` to check whether that annotation has already been used – if not, we store the annotation; if it has, we `paste` together a newly numbered annotation name and then store the annotation.

After both loops, we use `order` to re-order the components of `chat.files.list` and as`.data.frame` to convert it into a data frame; to make sure all columns have the same length, we apply `max` to the results of checking all columns' `length`s with `sapply`. Finally, we store the data frame with `write.table`.

**Thus, this is the overall structure of the script** <_qclwr2/_scripts/05_27_merging-chat.r>; read this side by side with the actual script in RStudio:

clear memory, load rChoiceDialogs (1–2)

define a vector `chat.files` with the corpus files (4–5)

create a collector list and set the utterance counter to 0 (7–9)

`for`-loop (counter = `i`, over `seq(chat.files)`): (11, -68)

- output a progress report (12) and read the lines of the current chat file (14–15)
- `paste` it together in one string with a `collapse` argument unattested in the string (17–18)
- replace whitespace before that `collapse` argument (i.e., after line breaks) with a single space (20–22), then split the string apart again at the `collapse` argument and `unlist` (24–25)
- find utterances and annotation lines and store them in a vector `current.chat.file.05` to be processed (27–30) and extract the date from the header (32–37)
- `for`-loop (counter = `j`, over `seq(current.chat.file.05)`): (40, -67)
  - extract the line type (utterance or annotation) from the current line (42), the identifier (speaker name of annotation type) (43), and the line content (44–45)
  - if the line type is "utterance" (47), increment the utterance counter by 1 (48) and set the tier counter to 1
  - also, store the file name, the speaker name, the date, and the line content (51–54)
  - if the line type is "annotation" (56), check whether that kind of annotation has not been found yet for this utterance (this is where we handle the two EXPs) (57)
  - if there isn't, save this line's annotation into `chat.files.list` (58–59)
  - if there is already that kind of annotation (60), increment the tier counter by 1 (61) and `paste` the new value of `tier.counter` together with the annotation name and insert the annotation into the `utt.counter`'s part of a thusly named component of `chat.files.list` (62–64)

`order` all but the first four component names of `chat.files.list` alphabetically (70–71)

*(continued)*

*(continued)*

create a data frame from the list by using `sapply` to extract from every list component all elements from 1 till that largest number of elements that any component of `chat.files.list` has (most likely the utterance tier) (73–79) and save it into a .csv file (81–83)

**Which aspects of the script are worth additional comment?** The only part that might be challenging is lines 57–65, but the difficulty of this part is not so much understanding the code, which I hope I explain in enough detail (especially in the code file), but being able to come up with some such approach. The key is to realize that annotation is saved in slots of the relevant list component that are tied to the utterance counter. Thus, if you're in utterance 20 and encounter EXP annotation, but `chat.files.list$exp` already has a twentieth element, then you know there has already been a first instance of an exp tier and you need to create another one. That in turn makes you realize you need a tier counter so that different exp (or other) tiers can be distinguished (rather than overwrite each other), but since the annotation is utterance-specific, you also realize that the tier counter has to be reset to 1 whenever you encounter a new utterance, and so on. Try to go through this stepwise and don't hesitate, for instance, to create versions of the corpus files that have more repeated tiers so you can go through the loop manually and check every step (recall p. 69), which will make things much clearer.

This concludes the long case study chapter. As with the first edition, I will make additional case studies or assignments available on the companion website as time goes by, so check it regularly for new stuff and/or look out for announcements on the newsgroup. However, there are already also some exercises for you in an exercises code file <_qclwr2/_scripts/exerciseboxes/05_exercises.r>.

## Note

1 We could have done this already on each iteration – i.e., within the loop – but that would have meant we have many more vectors that dynamically grow in the loop than necessary, which puts a considerable strain on memory. The version of the script you see here took a bit more than 11 minutes to run on my desktop, but the version which let the vectors for the forms and the tags grow in the loop took more than 71 minutes. Thus, these considerations – where/when do we create data structures, in or after a loop? – are not just nerdy technicalities but make a real difference in how fast scripts can run or, if the data set is *really* large, whether they can run at all.

## References

Baayen, R. Harald, Richard Piepenbrock, & Leon Gulikers. 1995. *The CELEX lexical database* (Release 2). Philadelphia, PA: Linguistic Data Consortium.

Brown, Roger. 1973. *A first language: The early stages*. Cambridge, MA: Harvard University Press.

Bybee, Joan, & Joanne Scheibman. 1999. The effect of usage on degrees of constituency: The reduction of don't in English. *Linguistics, 37*(4), 575–596.

Church, Kenneth W., & Patrick Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational Linguistics, 16*(1), 22–29.

Church, Kenneth Ward, William Gale, Patrick Hanks, & Donald Hindle. 1991. Using statistics in lexical analysis. In Uri Zernik (Ed.), *Lexical acquisition: Exploiting on-line resources to build a lexicon* (pp. 115–164). Hillsdale, NJ: Lawrence Erlbaum.

Church, Kenneth Ward, William Gale, Patrick Hanks, Donald Hindle, & Rosamund Mood. 1994. Lexical substitutability. In Beryl T. Sue Atkins & Antonio Zampolli (Eds.), *Computational approaches to the lexicon* (pp. 153–177). Oxford: Oxford University Press.

Damerau, F.J. 1993. Generating and evaluating domain-oriented multi-word terms from texts. *Information Processing and Management, 29*, 433–447.

Dunning, Ted. 1993. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics, 19*(1), 61–74.

Evert, Stefan. 2004. The statistics of word cooccurrences: Word pairs and collocations. Ph.D. diss., University of Stuttgart.

Gries, Stefan Th. 2001. A corpus-linguistic analysis of *-ic* and *-ical* adjectives. *ICAME Journal, 25*, 65–108.

Gries, Stefan Th. 2003. Testing the sub-test: A collocational-overlap analysis of English *-ic* and *-ical* adjectives. *International Journal of Corpus Linguistics, 8*(1), 31–61.

Gries, Stefan Th. 2008. Dispersions and adjusted frequencies in corpora. *International Journal of Corpus Linguistics, 13*(4), 403–437.

Gries, Stefan Th. 2010. Dispersions and adjusted frequencies in corpora: further explorations. In Stefan Th. Gries, Stefanie Wulff, & Mark Davies (Eds.), *Corpus linguistic applications: Current studies, new directions* (pp. 197–212). Amsterdam: Rodopi.

Gries, Stefan Th. 2012. Frequencies, probabilities, association measures in usage-/exemplar-based linguistics: Some necessary clarifications. *Studies in Language, 36*(3), 477–510.

Gries, Stefan Th. 2013a. 50-something years of work on collocations: What is or should be next . . . *International Journal of Corpus Linguistics, 18*(1), 137–165.

Gries, Stefan Th. 2013b. *Statistics for linguistics with R*. 2nd rev. and ext. ed. Berlin: De Gruyter Mouton.

Gries, Stefan Th. 2014. Coll.analysis 3.5. A program for R.

Gries, Stefan Th. 2015. More (old and new) misunderstandings of collostructional analysis: On Schmid & Küchenhoff (2013). *Cognitive Linguistics, 26*(3), 505–536.

Gries, Stefan Th., & Anatol Stefanowitsch. 2004a. Extending collostructional analysis: A corpus-based perspective on "alternations". *International Journal of Corpus Linguistics, 9*(1), 97–129.

Gries, Stefan Th., & Anatol Stefanowitsch. 2004b. Co-varying collexemes in the *into*-causative. In Michel Achard & Suzanne Kemmer (Eds.), *Language, culture, and mind* (pp. 225–236). Stanford, CA: CSLI.

Kuperman, Victor, & Raymond Bertram. 2013. Moving spaces: Spelling alternation in English noun–noun compounds. *Language and Cognitive Processes, 28*(7), 939–966.

Laufer, Batia. 2005. Lexical frequency profiles: From Monte Carlo to the real world. A response to Meara. *Applied Linguistics, 26*(4), 581–587.

Laufer, Batia, & Paul Nation. 1995. Vocabulary size and use: Lexical richness in L2 written production. *Applied Linguistics, 16*(3), 307–322.

Leech, Geoffrey and Roger Fallon. 1992. Computer corpora: What do they tell us about culture? *ICAME Journal, 16*, 29–50.

Leech, Geoffrey, Paul Rayson, & Andrew Wilson. 2001. *Word frequencies in written and spoken English: Based on the British National Corpus*. London: Longman.

Manning, Christopher D., & Hinrich Schütze. 2000. *Foundations of statistical natural language processing*. 2nd printing with corrections. Cambridge, MA: MIT Press.

Meara, Paul. 2005. Lexical frequency profiles: A Monte Carlo analysis. *Applied Linguistics, 26*(1), 32–47.

Mukherjee, Joybrato. 2007. Corpus linguistics and linguistic theory: General nouns and general issues. *International Journal of Corpus Linguistics, 12*(1), 131–147.

Oakes, Michael P. 1998. *Statistics for corpus linguistics*. Edinburgh: Edinburgh University Press.

Pecina, Pavel. 2009. Lexical association measures and collocation extraction. *Language Resources and Evaluation, 44*(1–2), 137–158.

Stefanowitsch, Anatol, & Stefan Th. Gries. 2003. Collostructions: Investigating the interaction between words and constructions. *International Journal of Corpus Linguistics, 8*(2), 209–243.

Stefanowitsch, Anatol, & Stefan Th. Gries. 2005. Covarying collexemes. *Corpus Linguistics and Linguistic Theory, 1*(1), 1–43.

Youmans, Gilbert. 1990. Measuring lexical style and competence: The type-token vocabulary curve. *Style, 24*(4), 584–599.

Youmans, Gilbert. 1991. A new tool for discourse analysis: The Vocabulary Management Profile. *Language, 67*(4), 763–789.

# 6 Next Steps . . .

> I think computer searching of corpora is the most useful tool that has been provided to the grammarian since the invention of writing.
>
> (Pullum 2006: 39)

As you will have inferred from this book so far, I am very much in favor of the idea that corpus linguists should not be dependent on a limited set of tools such as ready-made corpus software, which, by definition, comes with limitations: Not everything a (corpus) linguist may want to do can be readily implemented in them. But my reluctance against being dependent on 'things' goes further than that, which is why this book has only used freely available software and in fact mostly open-source software: All the code was primarily developed on Linux (though tested on Windows as well), the spreadsheet software used was from the LibreOffice suite etc. A final implication of this worth mentioning here briefly is that, with very few exceptions, this book has taught you how to do things relying as much as possible on just base R, i.e., relying on extra packages as little as possible – the exceptions are the packages for processing XML data and dplyr for the %>% operator, but even there I often show how things can be done without them. This is, again, just so that researchers don't become overly reliant on a particular package's functionality, which may change, be discontinued, etc. – I prefer researchers in the driving seat, as when they are when they are able to develop the functionality they need with the functions in 'base R' because then they, as do we all, know best what they're doing.

That being said, at the very end of this book I would like nevertheless to point out a few packages that may be useful for your research. The list is very brief and very selective, and new functions or even packages are being developed all the time, so I encourage you to regularly check the website of the CRAN Task View on Natural Language Processing (https://cran.r-project.org/web/views/NaturalLanguageProcessing.html), where many interesting packages are listed; some of these are more computational-linguistic than corpus-linguistic (at least to my mind), but not only is that definition subjective (see Gries 2011 for my take on this), but in practice may also not be a necessary one to make. Here are packages I recommend you check out (I do not provide code here, but you will find some examples of functions and their uses in the code file):

- stringi (https://cran.r-project.org/web/packages/stringi/index.html and www.rexamine.com/resources/stringi), which not only provides a unified set of string-processing functions with homogeneous and informative function names, but also encoding detection and conversion capabilities that can be very useful if your work involves frequently handling different encodings.

- stringr (https://cran.r-project.org/web/packages/stringr/index.html), which is a set of convenient functions that facilitate using stringi, which it in fact imports.
- gsubfn (https://cran.r-project.org/web/packages/gsubfn/index.html and https://code.google.com/archive/p/gsubfn), which played an important role in the first edition of this book and which allows you to find matches in a string and apply functions to them (`strapply`) and/or even insert the result of the function back into the original input vector (`gsubfn`).
- stringdist (https://cran.r-project.org/web/packages/stringdist/index.html), which provides a variety of approximate string matching functions that complement R's `%in%` and `match`; also, it offers many different measures of similarity between strings and even a function from which you can obtain character *n*-grams (recall Section 5.2.1).

In addition to these packages, I think the following ones may also be useful to learn about: tm, RcmdrPlugin.temis, openNLP, koRpus, hunspell, and wordnet – but again, explore the Task View for other useful tools that may help you in your research. A not so much corpus-linguistically interesting package, but one that provides a lot of useful functionality, is readr (https://cran.r-project.org/web/packages/readr/index.html), which offers functions to read large flat and tabular data files from connections. Finally, let me recommend this website for many new updates and ideas (e.g., on how to use R for webscraping, downloading Twitter feeds, etc.): www.r-bloggers.com

This concludes this book. I hope you have made it till here both in the book and in the code file; I hope you have learned a lot about all the possibilities R offers to linguistics in general and corpus linguistics in particular, and that the book has whetted your appetite to pursue your research with this amazing tool. In that spirit, run this:

```
>·rm(list=ls(all=TRUE));·set.seed(1)¶
>·paste(paste0(sample(letters)[c(17,·24,·19,·19,·13)],·
  collapse=""),¶
+·····"R",¶
+·····paste0(sample(letters)[c(20,·26,·9,·25,·26,·1,·6,·6,·19,·1
  7,·25)],¶
+·····collapse=""),¶
+·····"!",·collapse="")¶
```

## References

Gries, Stefan Th. 2011. Methodological and interdisciplinary stance in corpus linguistics. In Geoffrey Barnbrook, Vander Viana, & Sonia Zyngier (Eds.), *Perspectives on corpus linguistics: Connections and controversies* (pp. 81–98). Amsterdam: John Benjamins.

Pullum, Geoffrey K. 2006. Ungrammaticality, rarity, and corpus use. *Corpus Linguistics and Linguistic Theory, 3*(1): 33–47.

# Appendix

### Websites

The companion website for this book: http://tinyurl.com/QuantCorpLingWithR
The newsgroup on corpus linguistics with R: http://groups.google.com/group/corpling-with-r

### R and R Add-ons (Open Source)

R Project: www.r-project.org
R Project download page CRAN: https://cran.r-project.org
Microsoft R Open: https://mran.revolutionanalytics.com/open
RStudio: www.rstudio.com

### Statistics for Linguistics With R

The companion website for this book: http://tinyurl.com/StatForLingWithR
The newsgroup on statistics for linguistics with R: http://groups.google.com/group/statforling-with-r
R-lang: https://ling.ucsd.edu/mailman/listinfo.cgi/r-lang

### Office Software and Other Useful Software (Mostly Open Source)

Libreoffice: www.libreoffice.org
Unicode consortium: http://unicode.org

### Regular Expression Testers and Websites

www.regular-expressions.info
http://regexr.com
https://regexper.com
http://regexlib.com

# Index