

```
for (key2 in cys_x) {  
  =cys_x[key1]-cys_x[key2]  
  =cys_y[key1]-cys_y[key2]  
  =cys_z[key1]-cys_z[key2]  
  stance=sqrt(dx^2+dy^2+dz^2)  
  (distance < 3 && distance != 0  
  i++  
  candidate[i]=key1-"key2": "dis  
  hit[key1]=key1; hit[key2]=key2
```

```
(i!=0){  
  cmd1="cd $(cat FILENAME) 5 1 n  
  while((cmd1 | getline) > 0){  
    close(cmd1)
```

```
txtfile=FILENAME  
gsub(/\./,"_",txtfile)  
print FILENAME "spacefill" > txtfile  
for (keys in candidate) {print  
ORS=  
file=fopen(txtfile,"a")  
print keys > file  
print "spacefill" > file  
print "hide water" > file  
for  
print  
print  
cmd1
```

Röbbe Wünschiers

Computational Biology

A Practical Introduction
to BioData Processing and Analysis
with Linux, MySQL, and R

Second Edition

Computational Biology

Röbbe Wünschiers

Computational Biology

A Practical Introduction to BioData
Processing and Analysis with Linux,
MySQL, and R

Second Edition

Röbbe Wünschiers
Biotechnology/Computational Biology
University of Applied Sciences
Mittweida
Germany

ISBN 978-3-642-34748-1 ISBN 978-3-642-34749-8 (eBook)

DOI 10.1007/978-3-642-34749-8

Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012954526

© Springer-Verlag Berlin Heidelberg 2004, 2013

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

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

Dedicated to ...

*... Károly Nagy, who gave me my first
computer, a Casio PB-100,*

*... the Open-Source Community for providing
fantastic software, and*

*... my offsprings, with the hope that they may
find appropriate filters to manage life's
information avalanche.*

Foreword by an Experimental Biologist

The Data Culture

The preface for the first edition of this book was called “A shift in culture”, since the increasing transition of the work profile of a molecular biologist from the bench to the computer became already more than apparent at that time. Eight years have passed since then and the shift has become even more radical than anyone could have anticipated. Genomics has continued its explosive development, with sequencing costs within genome projects having gone down by a factor of 10,000 in the past 10 years. The generation of huge masses of data, even within an average Ph.D. project, is quickly becoming routine. Benchwork may become reduced to the mere extraction of DNA and RNA, followed by sending the material to sequencing centers. Data are then often returned on hard disks, since even the fastest online connections are too slow to transport them. Working with these data is now the challenge of our days, computational skills are the top key qualification for the molecular biologist.

Bioinformatics has kept up with this challenge and the small helpful “progies” of the past have transformed to increasingly sophisticated software packages that can deal with many of the initial standard problems of data analysis, such as quality control, mapping, and basic statistics. But handling the data remains the task of each individual scientist and being able to work with them in a Unix environment is crucial. This is why this book devotes several chapters to the basic task of mere data processing. The goal of each genomics experiment is of course to make sense out of the data and this requires connecting them with known information stored in databases. Skills in working with relational databases, such as MySQL, understanding their syntax, and being able to do some simple programming within them, is therefore equally indispensable.

But there is also another big problem hidden in the transition from a largely qualitative science, as molecular biology was, to a data-driven quantitative one—the understanding of statistics. An eye-catching outlier may very easily be simply a consequence of random fluctuation in a sea of data. If your chances to win a

jackpot in a lottery are 1 in 50,000,000, you would consider this an extremely unlikely event. But if 50,000,000 players take part in the lottery every week, it is a mere rule of statistics that one of them will take the jackpot almost every week. Sounds trivial, but there have indeed been a number of genomics papers that fell into exactly this trap by reporting a seemingly very novel insight, which was nothing else than statistical fluctuation. To guard you against this trap, statistical analysis and a full understanding of what its results tell you is a must. Hence, the book now includes also a new chapter on the use of R, a powerful statistical package for data analysis and graphics that allows running many controls at every step of data analysis—and can provide also the matching graphical output. I am convinced that this book should be the required reading for every molecular biologist. It will of course be particularly helpful for those dealing with genomics data, but even if genomics is currently not on your experimental agenda, handling large datasets and doing proper statistics is a basic qualification that cannot be underestimated in our discipline today.

Plön, Germany, August 2012

Prof. Dr. Diethard Tautz
Director at the Max-Planck Institute
for Evolutionary Biology

Foreword by a Computer Scientist and Father of AWK

A Must Read for any Computational Scientist

This book is a must read for any scientist interested in computational biology. All experimental scientists today must know how to analyze data and manipulate information by themselves. Because of cost and time, they cannot rely on hired help to do the analyses and manipulations for them. In a few hundred pages, this book teaches the reader to do useful data analysis and processing in the Internet age using the versatile and powerful tools available for all Unix/Linux environments.

The reader need not know anything about programming or Unix/Linux. Professor Dr. Wünschiers begins from scratch explaining the hardware/software architecture of a computer system starting from the hardware level and then going through the operating system kernel and the shell to the languages and packages available at the applications level. He even provides a succinct history of the development of the history behind the various versions of Unix and Linux systems.

The author provides a collection of concise chapters explaining how to get started on a Unix/Linux system, how to create and work with files, how to write programs in data-processing scripting languages such as awk, and how to download useful application packages. He introduces shell programming by showing the reader how to create simple but powerful shell scripts using sed and other shell commands. The author teaches the reader how to write real programs to do useful data-processing tasks. As one of the inventors of awk, I am delighted he has chosen our programming language as his primary vehicle. Since awk is a simple, easy-to-learn, and an easy-to-use data-processing language, he provides a comprehensive description of awk through a graded sequence of awk programs designed to solve illustrative problems. By the end of the section on awk, the reader should be familiar with most of the language and be able to solve useful problems in computational biology by writing his or her own awk programs.

The book also includes sections showing how to use resources available on the net and downloadable application packages such as the relational database system

MySQL, the statistics suite R, and the similarity-sequence detection BLAST to solve representative problems in contemporary computational biology.

There are many things I like about this book. First, the material on Unix/Linux is presented in a no-nonsense manner that would be familiar and appealing to any Unix/Linux programmer. It is clear the author has internalized the powerful Unix/Linux building-block approach to problem solving. Second, the book is written in a lively and engaging style. It is not a turgid user manual. Finally, throughout the book the author admonishes the reader to write programs continuously as he or she reads the material. This cannot be overemphasized—it is well known that the only way to learn how to program effectively is by writing and running programs.

If you want to become a computational biologist proficient in solving real problems in the Unix/Linux environment by yourself, then this book is a must read.

New York, USA, September 2012

Alfred V. Aho
Lawrence Gussman Professor
Department of Computer Science
Columbia University

Preface to the Second Edition

AWKology at Its Best

This year was full of innovative achievements in the field of computational biology and bioinformatics. I just like to mention two personal highlights: (a) The publication of a whole-cell computational model of the bacterium *Mycoplasma genitalium* that allows prediction of phenotype from genotype (Karr et al 2012) and (b) the coordinated publication of major results from the international ENCODE (Encyclopedia of DNA Elements) project as a set of 30 papers across three different journals that are digitally cross-linked as so-called threads. Each thread consists of theme-specific paragraphs, figures, and tables from across these papers. Both research projects handle a huge amount of complex data. But at the very basis there are a number of tabulator-delimited text files that needed to be filtered, rearranged, reformatted, statistically analyzed, or transferred to relational databases for improved data handling and number crunching.

This is precisely what this book is about. **My aim is to place you in a better position to handle and analyze data**—lots of data. It arose from my own needs and experiences to do so. Once you learn how to play with your datasets, these in turn may change your mindset (as Hans Rosling once put it). The core is data processing and visualization. Thus, this book is about data piping, not pipetting.

The Book's Title. The title of this book is **Computational Biology**. Some would argue that its content is **bioinformatics**. Why is that? I am a biologist and I complement my experiments with computational methods. **To me, computational biology is the complement to experimental biology.** During my professional career in industries I was head of projects which aimed at different things like gene discovery, data integration and visualization, and statistical analyses. I was employed as a bioinformatics manager. But was it bioinformatics that I was doing? Or was it computational biology? Or something else?

There certainly is a difference between computational biology and bioinformatics. However, it heavily depends on whom you are asking (see [Sect. 1.3](#) on page 6). I often hear that bioinformatics is about the development of software tools while computational biology deals with mathematical modeling. A leading journal, PLOS Computational Biology, states that it publishes work that furthers our understanding of living systems at all scales through the application of computational methods. Computational methods in turn involve information processing—and this is what this book is about. Anyway, both terms are frequently used synonymously and the buzz word clearly is bioinformatics—I am glad that you still found this book.)

New Chapters, Extended Concept and a Dinosaur. What has changed since the publication of the first edition in 2004? A lot! Next-generation sequencing is not the next generation any more—it is presence. This means that there is a lot more data available (Strasser 2012). High throughput methods became the standard in molecular analysis and provide even more data. More data means that there are more and higher possibilities to find correlations between datasets. But it also implies that there are more datasets that have to be processed.

This new edition grew out of my experience of working with biological data in both academia and industries. I saw the need to add chapters on databases (**MySQL**) and statistical data analysis & visualization (**R**). From my courses on computational biology I learned about the importance of having a tangible problem to solve. This motivates to move on in the command line and likewise demonstrates its power. Therefore, I chose to add **worked examples**.

Since 2004, Linux has become much more comfortable—not only to use, but also to install. Gaining access to a USB-stick was a pain in the neck back then: `mount -t vfat /dev/hde1 /home/Freddy/USB/`. Nowadays, everybody can install a free virtual machine and run almost any operating system on any operating system—in parallel. I take advantage of these developments by showing how to set up **Ubuntu Linux in a VirtualBox**.

My dinosaur is **AWK**. Though there is almost no further development, it is still just amazing to see what one can do. I met several experimentalists who are into computational biology that apply AWK. Why? One of its three developers, Alfred Aho, recently said: *If I had to choose a word to describe our centering forces in language design, I'd say Kernighan emphasized ease of learning; Weinberger, soundness of implementation; and I, utility. I think AWK has all three of these properties* (Biancuzzi and Warden 2009). That describes it well. Students without any programming experience usually pickup data processing in the command line with AWK within some days. I love it.

Acknowledgments. I wish to thank all colleagues and students who have read, commented upon, and corrected various chapters of this book. This second edition benefited a lot from the waking eyes of the students attending my computational biology course, especially Sebastian Gustmann and Robin Schwarzer at Cologne University, Germany.

Quedlinburg, Germany, September 2012

Röbbe Wünschiers

References

- Karr et al (2012) A whole-cell computational model predicts phenotype from genotype. *Cell* 150:389. simtk.org.
- Strasser BJ (2012) Data-driven sciences: from wonder cabinets to electronic databases. *Stud Hist Philos Biolo Biomed Sci* 43:85.
- Biancuzzi F, Warden S (eds) (2009) *Masterminds of programming*. O'Reilly, Sebastopol, p 104.

Preface to the First Edition

Welcome on Board!

With this book I would like to invite you, the scientist, to a journey through terminals and program codes. You are welcome to put aside your pipette, culture flask, or rubber boots for a while, make yourself comfortable in front of a computer (do not forget your favourite hot alcohol-free drink), and learn some unixing and programming. *Why?* Because we are living in the information age and there is a huge amount of biological knowledge and databases out there. They contain information about almost everything: genes and genomes, rRNAs, enzymes, protein structures, DNA-microarray experiments, single organisms, ecological data, the tree of life, and endless more. Furthermore, nowadays many research apparatuses are connected to computers. Thus, you have electronic access to your data. However, in order to cope with all this information you need some tools. This book will provide you with the skills to use these tools and to develop your own tools, i.e., it will introduce Unix and its derivatives (Linux, Mac OS X, CygWin, etc.) and programming (shell programming, awk, perl). These tools will make you independent of the way in which other people make you process your data—in the form of application software. What you want is open functionality. You want to decide how to process (e.g., analyze, format, save, correlate) data and you want it now—not waiting for the lab programmer to treat your request; and you know it best—you understand your data and your demands. This is what open functionality stands for, and both Linux and programming languages can provide it to you.

I started programming on a Casio PB-100 hand-held built in 1983. It can store 10 small Basic programs. The accompanying book was entitled “Learn as you go” and, indeed, in my opinion this is the best way to learn programming. My first contact to Unix was triggered by the need to copy data files from a Unix-driven Bruker EPR-Spectrometer onto a floppy disk. The real challenge started when I

tried to import the files to a data-plotting program on the PC. While the first problem could be solved by finding the right page in a Unix manual, the latter required programming skills—Q-Basic at that time. This problem was minor compared to the trouble one encounters today. A common problem is to feed one program with the output of another program: you might have to change lines to columns, commas to dots, tabulators to semicolons, uppercase to lowercase, DNA to RNA, FASTA to GenBank format, and so forth. Then there is that huge amount of information out there on the Web, which you might need to bring into shape for your own analysis.

You and This Book. This book is written for the total beginner. You need not even know what a computer is, though you should have access to one and find the power switch. The book is the result of (a) the way I learned to work with Unix, its derivatives, and its numerous tools and (b) a lecture which I started at the Institute for Genetics at the University of Cologne/Germany. Most programming examples are taken from biology; however, you need not be a biologist. Except for two or three examples, no biological knowledge is necessary. I have tried to illustrate almost everything practically with so-called terminals and examples. You should run these examples. Each chapter closes with some exercises. Brief solutions can be found at the end of the book.

Why Linux? This book is not limited to Linux! All examples are valid for Unix or any Unix derivative like *Mac OS X*, *Knoppix* or the free Windows-based *CygWin* package, too. I chose Linux because it is open source software: you need not invest money except for the book itself. Furthermore, Linux provides all the great tools Unix provides. With Linux (as with all other Unix derivatives) you are close to your data. Via the command line you have immediate access to your files and can use either publicly available or your own designed tools to process these. With the aid of *pipes* you can construct your own data-processing pipeline. It is great.

Why awk and perl? **awk** is a great language for both learning programming and treating large text-based data files (contrary to binary files). For 99 % you will work with text-based files, be it data tables, genomes, or species lists. Apart from being simple to learn and having a clear syntax, **awk** provides you with the possibility to construct your own commands. Thus, the language can grow with you as you grow with the language. I know bioinformatic professionals entirely focusing on **awk**. **perl** is much more powerful but also more unclear in its syntax (or flexible, to put it positively), but, since **awk** was one basis for developing **perl**, it is only a small step to go once you have learned **awk** – but a giant leap for your possibilities. You should take this step. By the way, both **awk** and **perl** run on all common operating systems.

Acknowledgments. Special thanks to Kristina Auerswald, Till Bayer, Benedikt Bosbach, and Chris Voolstra for proofreading, and all the other students for encouraging me to bring these lines together.

Hürth, Germany, January 2004

Röbbe Wünschiers

Contents

Part I Whetting Your Appetite

1	Introduction	3
1.1	A Very Short History of Bioinformatics and Computational Biology	3
1.2	Contemporary Biology	5
1.3	Computational Biology or Bioinformatics?	6
1.4	Computers in Biology	7
1.5	The Future: Digital Lab Benches and Designing Organisms	8
2	Content of This Book	11
2.1	The Main Chapters	11
2.1.1	Linux	11
2.1.2	Shell Programming	12
2.1.3	Sed	12
2.1.4	AWK	13
2.1.5	Perl	13
2.1.6	MySQL	14
2.1.7	R	14
2.1.8	Worked Examples	15
2.2	Prerequisites	15
2.3	Conventions	16
2.4	Additional Resources	17

Part II Computer and Operating Systems

3	Unix/Linux	21
3.1	What is a Computer?	21
3.2	Some History	22

3.2.1	Versions of Unix.	23
3.2.2	The Rise of Linux.	24
3.2.3	Why a Penguin?	26
3.2.4	Linux Distributions	26
3.3	X-Windows	26
3.3.1	How Does it Work?	27
3.4	Linux Architecture	28
3.5	What is the Difference Between Unix and Linux?	29
3.6	What is the Difference Between Linux and Windows?	29
3.7	What is the Difference Between Linux and Mac OS X?	29
3.8	One Computer, Two Operating Systems	30
3.8.1	VMware.	30
3.8.2	CygWin	30
3.8.3	Wine	31
3.8.4	Others	31
3.9	Knoppix.	31
3.9.1	Vigyaan Knoppix for Biosciences	32
3.10	Software Running Under Linux	33
3.10.1	Bioscience Software for Linux	33
3.10.2	Office Software and Co.	34
3.10.3	Graphical Desktops	35
3.10.4	Others	35

Part III Working with Linux

4	The First Touch	39
4.1	Just Before the First Touch	39
4.1.1	Running Linux from USB or CDROM	39
4.1.2	Running Linux as a Virtual Machine.	40
4.2	Login	43
4.2.1	Working with CygWin or Mac OS X	44
4.2.2	Working Directly on a Linux Computer.	44
4.2.3	Working on a Remote Linux Computer.	44
4.3	Using Commands	46
4.3.1	History and Autocompletion.	47
4.3.2	Syntax of Commands	47
4.3.3	Editing the Command Line	48
4.3.4	Change Password	49
4.3.5	Help: I'm Stuck	49
4.3.6	How to Find out More.	49
4.4	Logout.	50

5	Working with Files	53
5.1	Browsing Files and Directories	53
5.2	Moving, Copying and Renaming Files and Directories	56
5.3	File Attributes	56
5.4	Special Files: <code>.</code> and <code>..</code>	57
5.5	Protecting Files and Directories	59
5.5.1	Directories	59
5.5.2	Files	59
5.5.3	Examples	59
5.5.4	Changing File Attributes	60
5.5.5	Extended File Attributes	62
5.6	File Archives	62
5.7	File Compression	65
5.8	Searching for Files	66
5.8.1	Selecting Files for Backups	68
5.9	Display Disk Usage	68
6	Remote Connections	71
6.1	Downloading Data from the Web	71
6.1.1	<code>wget</code>	71
6.1.2	<code>curl</code>	72
6.2	Secure Copy: <code>scp</code>	72
6.3	Secure Shell: <code>ssh</code>	73
6.4	Backups with <code>rsync</code>	73
6.5	<code>ssh</code> , <code>scp</code> & <code>rsync</code> without Password	74
7	Playing with Text and Data Files	77
7.1	Viewing and Analyzing Files	78
7.1.1	A Quick Start: <code>cat</code>	78
7.1.2	Text Sorting	79
7.1.3	Extract Unique Lines	81
7.1.4	Viewing File Beginning or End	81
7.1.5	Scrolling Through Files	81
7.1.6	Character, Word, and Line Counting	82
7.1.7	Splitting Files into Pieces	82
7.1.8	Cut and Paste Columns	83
7.1.9	Finding Text: <code>grep</code>	84
7.1.10	Text File Comparisons	85
7.2	Editing Text File	86
7.2.1	The Sparse Editor <code>Pico</code>	87
7.2.2	The Rich Editor <code>Vim</code>	88
7.2.3	Installing <code>Vim</code>	89
7.2.4	Immediate Takeoff	89
7.2.5	Starting <code>Vim</code>	89

7.2.6	Modes	90
7.2.7	Moving the Cursor	91
7.2.8	Doing Corrections	92
7.2.9	Save and Quit	93
7.2.10	Copy and Paste	93
7.2.11	Search and Replace	94
7.3	Text File Conversion (Unix ↔ DOS)	95
7.3.1	Batch Editing	95
8	Using the Shell	97
8.1	What is the Shell?	97
8.2	Different Shells	98
8.3	Setting the Default Shell	99
8.4	Useful Shortcuts	99
8.5	Redirections	101
8.6	Pipes	103
8.7	Lists of Commands	104
8.8	Aliases	104
8.9	Pimping the Prompt	105
8.10	Batch Jobs	106
8.11	Scheduling Commands	107
8.12	Wildcards	108
8.13	Processes	109
8.13.1	Checking Processes	110
8.13.2	Realtime Overview	111
8.13.3	Background Processes	111
8.13.4	Killing Processes	113
9	Installing BLAST and ClustalW	115
9.1	Downloading the Programs via FTP	115
9.1.1	Downloading BLAST	116
9.1.2	Downloading ClustalW	117
9.2	Installing BLAST	118
9.3	Running BLAST	119
9.4	Installing ClustalW	120
9.5	Running ClustalW	121
9.6	A Wrapper for ClustalW	122
10	Shell Programming	125
10.1	Script Files	125
10.2	Modifying the Path	127
10.3	Variables	128

10.4	Input and Output	130
10.4.1	echo	130
10.4.2	Here Documents: <<.	131
10.4.3	read and line.	132
10.4.4	Script Parameters	133
10.5	Substitutions and Expansions	134
10.5.1	Variable Substitution.	134
10.5.2	Command Expansion.	136
10.6	Quoting	136
10.6.1	Escape Character	137
10.6.2	Single Quotes.	137
10.6.3	Double Quotes	137
10.7	Decisions: Flow Control	138
10.7.1	if...then...elif...else...fi.	138
10.7.2	test	139
10.7.3	while...do...done	141
10.7.4	until...do...done	143
10.7.5	for...in...do...done	144
10.7.6	case...in...esac	145
10.7.7	select...in...do	146
10.8	Desktop Notifications	147
10.8.1	MacOSX	147
10.8.2	Linux.	148
10.8.3	Windows	148
10.9	Debugging	148
10.9.1	bash -xv.	148
10.9.2	trap	150
10.10	Remote Control Interactive Programs	150
10.11	Examples.	152
10.11.1	Check for DNA as File Content	152
10.11.2	Time Signal	153
10.11.3	Select Files to Archive	153
10.11.4	Remove Spaces.	155
11	Regular Expressions	157
11.1	Regular Expression and Neurons	157
11.2	Get Started	158
11.3	Using Regular Expressions.	160
11.4	Search Pattern and Examples	161
11.4.1	Single-Character Meta Characters	162
11.4.2	Quantifiers	164
11.4.3	Grouping	165
11.4.4	Anchors	166
11.4.5	Escape Sequences	167

11.4.6	Alternation	167
11.4.7	Back References	168
11.4.8	Character Classes	169
11.4.9	Priorities	169
11.4.10	egrep Options	170
11.5	RE Summary	170
11.6	Regular Expression Training	172
11.6.1	Regular Expressions and vim.	172
11.7	Regular Expression in Genome Research.	172
12	Sed	175
12.1	When to Use Sed?	176
12.2	Getting Started	176
12.3	How Sed Works	178
12.3.1	Pattern Space	178
12.3.2	Hold Space.	179
12.4	Sed Syntax.	179
12.4.1	Addresses.	180
12.4.2	Sed and Regular Expressions	181
12.5	Commands.	181
12.5.1	Substitutions.	182
12.5.2	Transliterations	184
12.5.3	Deletions	185
12.5.4	Insertions and Changes	186
12.5.5	Sed Script Files	188
12.5.6	Printing	189
12.5.7	Reading and Writing Files	189
12.5.8	Advanced Sed.	190
12.6	Examples.	190
12.6.1	Gene Tree	191
12.6.2	File Tree	191
12.6.3	Reversing Line Order	192

Part IV Programming

13	AWK.	197
13.1	Getting Started	198
13.2	AWK's Syntax	199
13.3	Example File	200
13.4	Patterns	201
13.4.1	Regular Expressions	201
13.4.2	Pattern-Matching Expressions.	202
13.4.3	Relational Character Expressions	203

13.4.4	Relational Number Expressions	205
13.4.5	Mixing and Conversion of Numbers and Characters	206
13.4.6	Ranges.	206
13.4.7	BEGIN and END.	207
13.5	Variables	208
13.5.1	Assignment Operators	208
13.5.2	Increment and Decrement	209
13.5.3	Predefined Variables	210
13.5.4	Arrays	215
13.5.5	Shell Versus AWK Variables	219
13.6	Scripts and Executables	219
13.7	Decisions: Flow Control	220
13.7.1	if...else...	220
13.7.2	while...	221
13.7.3	do...while...	222
13.7.4	for...	223
13.7.5	Leaving Loops	224
13.8	Actions	225
13.8.1	Printing	225
13.8.2	Numerical Calculations	228
13.8.3	String Manipulation.	229
13.8.4	System Commands	234
13.8.5	User-Defined Functions	234
13.9	Input with <code>getline</code>	238
13.9.1	Reading from a File	239
13.9.2	Reading from the Shell	241
13.10	Produce Nice Output with $\text{\LaTeX}$	242
13.11	Examples.	243
13.11.1	Sort an Array by Indices	243
13.11.2	Sum up Atom Positions	244
13.11.3	Convert FASTA $\leftrightarrow$ Table Formats	244
13.11.4	Mutate DNA Sequences.	245
13.11.5	Translate DNA to Protein	246
13.11.6	Calculate Atomic Composition of Proteins.	247
13.11.7	Dynamic Programming: Levenshtein Distance of Sequences	250
14	Perl	255
14.1	Intention of This Chapter.	255
14.2	Running Perl Scripts	256
14.3	Variables	256
14.3.1	Scalars.	257
14.3.2	Arrays	258

14.3.3	Hashes	262
14.3.4	Built-in Variables	266
14.4	Decisions: Flow Control	266
14.4.1	if...elsif...else	267
14.4.2	unless, die and warn.	267
14.4.3	while...	268
14.4.4	do...while...	268
14.4.5	until...	268
14.4.6	do...until...	268
14.4.7	for...	269
14.4.8	foreach...	269
14.4.9	Controlling Loops	269
14.5	Data Input	271
14.5.1	Command Line	271
14.5.2	Internal	272
14.5.3	Files	272
14.6	Data Output	274
14.6.1	print, printf and sprintf	274
14.6.2	Here Documents:<<	274
14.6.3	Files	275
14.7	Hash Databases	276
14.8	Regular Expressions	277
14.8.1	Special Escape Sequences	277
14.8.2	Matching: m/.../.../	278
14.9	String Manipulations	279
14.9.1	Substitute: s/.../.../	279
14.9.2	Transliterate: tr/.../.../	279
14.9.3	Common Commands	280
14.10	Calculations	281
14.11	Subroutines	282
14.12	Packages and Modules	283
14.13	Bioperl	285
14.14	You Want More?	286
14.15	Examples	286
14.15.1	Reverse Complement DNA	286
14.15.2	Calculate GC Content	287
14.15.3	Restriction Enzyme Digestion	288
15	Other Programming Languages	291

Part V Advanced Data Analysis

16	Relational Databases with MySQL	295
16.1	What is MySQL	295
16.1.1	Relational Databases	296
16.2	Administration	297
16.2.1	Get in Touch with the MySQL Server	297
16.2.2	Starting and Stopping the Server	298
16.2.3	Setting a Root Password	299
16.2.4	Set Up an User Account and Database	301
16.3	Utilization	303
16.3.1	Remote Access	303
16.3.2	MySQL Syntax	303
16.3.3	Creating Tables	304
16.3.4	Filling and Editing Tables	306
16.3.5	Querying Tables	309
16.3.6	Joining Tables	312
16.3.7	Access from the Shell	313
16.4	Backups and Transfers	314
16.5	How to Move on?	315
17	The Statistics Suite R	317
17.1	Getting Started	317
17.1.1	Getting Help	320
17.2	Reading and Writing Data Files	320
17.3	Data Structures	322
17.3.1	Vectors	322
17.3.2	Arrays and Matrices	322
17.3.3	Lists and Data Frames	324
17.4	Programming Structures	325
17.5	Data Exploration	327
17.5.1	Saving Graphic	330
17.5.2	Regression	332
17.5.3	t-Test	336
17.5.4	Chi-Square Test	337
17.6	R Scripts	338
17.7	Extensions: Installing and Using Packages	339
17.7.1	Connect to MySQL	339
17.7.2	Life-Science Packages	341

Part VI Worked Examples

18	Genomic Analysis of the Pathogenicity Factors from <i>E. coli</i> Strain O157:H7 and EHEC Strain O104:H4	345
18.1	The Project	345
18.2	Bioinformatic Tools and Resources	346
18.2.1	BLAST+	347
18.2.2	Genome Databases	347
18.3	Detailed Instructions	348
18.3.1	Downloading and Installing BLAST+	349
18.3.2	Downloading the Proteomes	351
18.3.3	Comparing the Genomes	353
18.3.4	Processing the BLAST+ Result File	357
18.3.5	Playing with the E-Value	360
18.3.6	Organize Results with MySQL	361
18.3.7	Extract Unique ORFs	366
18.3.8	Visualize and Analyze Results with R	367
19	Limits of BLAST and Homology Modeling	375
19.1	The Project	375
19.2	Bioinformatic Tools	376
19.2.1	genomicBLAST	376
19.2.2	ClustalW	376
19.2.3	Jpred	377
19.2.4	SWISS-MODEL	378
19.2.5	Jmol	378
19.3	Detailed Instructions	378
19.3.1	Download <i>E. coli</i> HybD Sequences	378
19.3.2	Cyanobacterial BLAST I	379
19.3.3	Cyanobacterial BLAST II	380
19.3.4	Compute Alignment and Tree Diagram	382
19.3.5	Secondary Structure Examination	384
19.3.6	Tertiary Structure Alignment (Homology Modeling)	386
19.3.7	Download and Visualize <i>E. coli</i> HybD Structure	387
19.3.8	View and Compare Cyanobacterial Structures	389
20	Virtual Sequencing of pUC18c	393
20.1	The Project	393
20.2	Bioinformatic Tools	393
20.2.1	Sequencer	393
20.2.2	Assembler	393
20.2.3	Dotter	394
20.2.4	GenBank	394

20.3	Detailed Instructions	394
20.3.1	Download the pUC18c Sequence	395
20.3.2	Download and Setup the Virtual Sequencer	395
20.3.3	Download and Install the TIGR Assembler	396
20.3.4	Download and Setup Dotter	397
20.3.5	Virtual Sequencing	399
20.3.6	Sequence Assembly.	402
20.3.7	Testing Different Parameters	403
20.3.8	Visualizing Results with R.	406
21	Querying for Potential Redox-Regulated Enzymes.	409
21.1	The Project	409
21.2	Bioinformatic Tools and Resources.	409
21.2.1	RCSB PDB	409
21.2.2	Jmol	410
21.2.3	Surface Racer.	410
21.3	Detailed Instructions	411
21.3.1	Download Crystal Structure Files	411
21.3.2	Analyze Structures Visually	412
21.3.3	Analyze Structures Computationally	413
21.3.4	Expand the Analysis to Many Proteins	417
21.3.5	Automatically Test for Solvent Accessibility	419
	Appendix A: Supplementary Information.	425
	References	433
	Solutions.	437
	Index	443

Part I
Whetting Your Appetite

Chapter 1

Introduction

1.1 A Very Short History of Bioinformatics and Computational Biology

Sequencing of the first proteins by Frederick Sanger in 1953 (1958 Nobel Prize in chemistry for his work on the structure of proteins, especially that of insulin) and the first protein crystal structure analysis by Max Perutz and John Kendrew (1962 Nobel Prize in chemistry for determining the first atomic structures of proteins using X-ray crystallography) in 1960 were among the first important datasets that gave rise to computational biology and bioinformatics. John Kendrew took advantage of the first European computer named ESDAC (*electronic delay storage automatic calculator*, setup in 1949 in Cambridge/UK) to calculate the structure of myoglobin from X-ray diffraction data. Thus, John Kendrew can be considered as one of the founders of chemoinformatics. At the same time, John William Mauchly and John Presper Eckert developed the first commercially available computer UNIVAC I (*universal automatic computer*) in 1951 (Fig. 1.1 on the next page) and, in 1956, IBM employee John Backus developed with Fortran the first compiler programming language.

Fortran was the programming language used by Margaret Oakley Dayhoff from the US (Fig. 1.2 after the next page). She was a pioneer in bioinformatics. One of her first projects was to create a Fortran program to determine the full protein sequence from sequence fragments in the early 1960s (Dayhoff 1964). The same task, but to an incomparably greater extent and with DNA sequences, was solved by the team led by Craig Venter when they established whole-genome shotgun sequencing. With this method, complete genomes are assembled from “shredded” genomic DNA. It was used when sequencing the first genome of a free-living bacterium in 1995 (Fleischmann et al. 1995), and in the sequencing of the human genome in 2001 by Celera Genomics (Venter et al. 2001).

In the 1970s, Margaret Dayhoff examined the frequency of amino acid exchanges on the basis of a few hundred related protein sequences using statistical methods. From this data she developed the so-called substitution matrices, which are of great practical importance in sequence analysis and sequence queries in databases to this day. In this

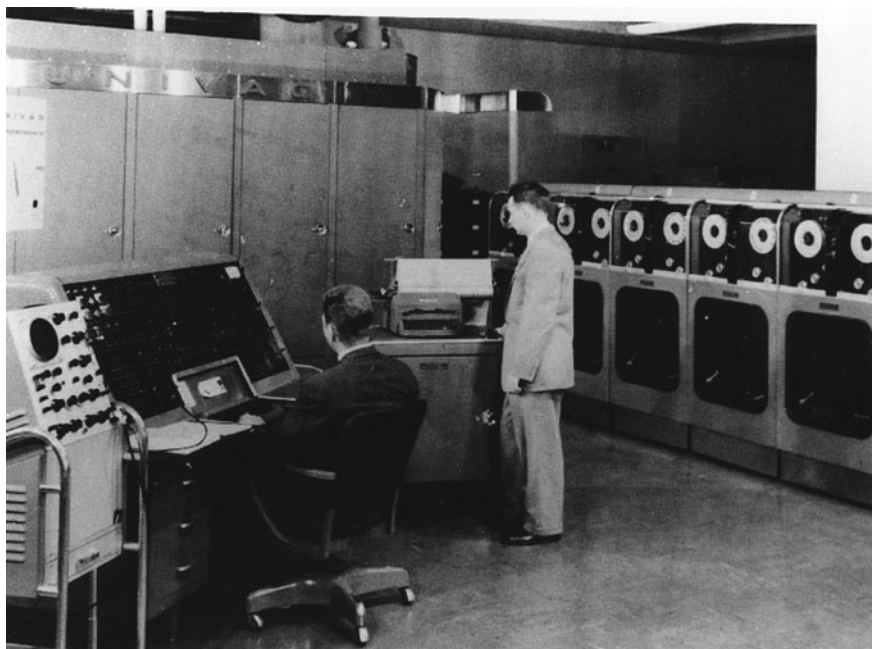


Fig. 1.1 UNIVAC I. The first commercially available computer was composed of 5,200 electron, weighed 13 tons and consumed 125 kW. At a frequency of 2.25 MHz, it could execute 1,905 calculations per second. The entire computer occupied 36 m². The program was input by punched cards

context, the first algorithms to reconstruct the evolution of protein sequences of organisms were developed (Fitch and Margoliash 1967). Prerequisite was access to powerful computers, such as the IBM 7090 that was also used in the Mercury and Gemini space program. In 1977, Alan Maxam and Walter Gilbert and, independently, Frederick Sanger published methods, which laid the foundation for automated DNA sequencing.

Two years later, the company Oracle introduced the first commercial database software and, in 1979, Walter Goad developed the prototype of GenBank, the first public nucleic acid and protein sequence database. To find sequences in this database that are similar to a query sequence, a sequence alignment algorithm originally developed by Saul Needleman and Christian Wunsch (1970) and modified by Temple Smith and Michael Waterman (1981) was employed. This development led to BLAST (*basic local alignment software tool*), a software known by every molecular biologist today (Altschul et al. 1990, 1997).

Molecular Design Ltd. (chemical databases, founded in 1978), Health Design Inc. (toxicological predictions, founded in 1978), Tripos Associates Inc. (molecular modelling and drug design, founded in 1979) and IntelliGenetics (DNA and protein sequence analysis, founded in 1980) were pioneers in market-based use of computers in the field of chemistry and biochemistry. Astonishingly, the basis for what today is celebrated as bioinformatics revolution has been laid out about 30 years ago.



Fig. 1.2 Margaret Dayhoff (1925–1983). A pioneering bioinformatician. Photo: Ruth E. Dayhoff, M.D.; U.S. National Library of Medicine

In the early days of bioinformatics, access to a computer for bioscientists was anything but a matter of course. In the early 1960s, computers were the size of closets, cost at least \$50,000, could only edit one program at a time and could only be used by one single user at a time. Initially, only universities and other large research facilities could afford a computer that had to be maintained and operated by qualified personnel. Users supplied their programs in the form of punched cards and received their result a few days later. In 1969, Ken Thompson and Ritchie Dennes from the US company AT&T started to develop a multi-user operating system, allowing program execution by multiple users in parallel: Unix. They published their idea in 1974 (Ritchie and Thompson 1974), and thus set in motion the distribution of Unix. Today, access to computers is available to all students. In addition, the development of the World Wide Web allows to exchange large amounts of data easily and biological databases can be accessed through user-friendly interfaces (front-ends). Therefore, we saw a boom in computational biology.

1.2 Contemporary Biology

For many scientists, the computer has become more important than the actual experiment. This is true for both bench and field scientists. The development of automated, computer-controlled instrumentation has led to the generation of lots of tools which

aid data retrieval, analysis, and visualization. Automation of data collection affects all disciplines; it does not make any difference if the data source is a photoelectric barrier at the entry of a beehive, a meteorological station, an animal trap, or a DNA sequencing machine. Usually, data analysis is supported by the software programmed by specialist scientists. The focus of such software is to solve problems, but not user-friendliness. Like MUSCLE, a powerful software for sequence alignments (Edgar 2004), these tools often use the command line of the operating system, i.e., either the DOS or PowerShell window in Microsoft Windows or the Unix-like terminal (console, shell) in Apple's MacOSX and Linux.

Currently, data processing with office software packages like Microsoft Excel is common in sciences like biology. But there are limitations. Zeeberg et al. (2004) showed that Excel could irreversibly change gene names to non-gene names: this even affected one of the most important resources for molecular biology information, GenBank. Due to default date format conversions and floating-point format conversions in Excel, the gene name SEPT1 (encoding a protein of the *septin* family) becomes 1-Sep and the gene identifier 2310009E13 gets converted into the floating-point number $2.31E + 13$, respectively. These changes are irreversible; the original gene names or identifiers cannot be recovered. Furthermore, the organization and format of data frequently need to be adapted to certain analysis software requirements; e.g., columns need to become rows or colons should be commas. These few examples illustrate that, with commercial software, one is usually bound to standard functionality. If scientists want to escape from these limits, they should learn something about data processing, which in turn requires knowledge about command line tools. In this book, I introduce you to some of the possibilities from an experimental biologist's perspective.

1.3 Computational Biology or Bioinformatics?

The title of this book is *Computational Biology*; however, the title of many similar books is *Bioinformatics* (see Table 1.1 on the facing page). This immediately raises the question: what is the difference? Does bioinformatics sell better? Is bioinformatics what most people do?

I spent quite a while to find out what is buried behind both terms. The result is not only Table 1.1 but also the impression that there is no clear distinction. In May 2012, Google found 34,100,000 hits for bioinformatics and 9,920,000 hits for computational biology. Google Trends does even teach me, that the term *bioinformatics* has been queried 20 times more frequently than *computational biology* over the period from 2004 to 2012. However, in a recent review, Christos Ouzounis shows that Google Trends hits for the query term *bioinformatics* has diminished by almost sixfold over the past 7 years (Ouzounis 2012). He extrapolates that the term will be irrelevant in 651 weeks, i.e. in 2024. Apart from this humorous conclusion, the review use both terms synonymously.

Table 1.1 Co-occurrence of the terms *Computational Biology* and *Bioinformatics* with other terms as of May 2012

Search term	Amazon books	Google scholar	Google books	PubMed	PubMed titles
“Computational biology”	1,189	232,000	77,100	35,198	168
“Computational biology” algorithm (%)	29	26	44	25	<0.01
“Computational biology” tool (%)	8	30	19	9	<0.01
“Computational biology” molecular (%)	28	59	40	48	0.05
Bioinformatics	2,703	1,210,000	370,000	99,595	2008
Bioinformatics algorithm (%)	41	32	22	17	<0.1
Bioinformatics tool (%)	52	46	23	8	0.02
Bioinformatics molecular (%)	41	57	41	42	0.03
“Computational biology” bioinformatics	1,074	–	27,300	35,198	0

The percentages are with respect to the hits for the single query term

During my professional career in industries I was head of projects which aimed at different things like gene discovery, data integration and visualization, and statistical analyzes. I was employed as a bioinformatics manager. But was it bioinformatics what I was doing? Or was it computational biology? Or something else? There are many opinions on what bioinformatics or computational biology are. **To me, computational biology is the complement to experimental biology.** The aim of both disciplines is to learn more about biology, i.e. an ecosystem, a synthetic cell, the stoichiometry of a genome, the regulation of a gene . . . Bioinformatics on the other hand provides algorithms and tools for the treatment of biological data. Here, the focus lays more at the side of the algorithm or the statistical method applied. Anyway, both terms are frequently used in synonym and the buzz word clearly is bioinformatics—I am glad that you still found this book.

1.4 Computers in Biology

The first use of a computer for sequence analysis was in the 1960s by Margaret Dayhoff (Dayhoff and Ledley 1962; Dayhoff 1964) (see also Sect. 1.1 on p. 3). Since then the field of computational biology has developed immensely, as has the power of computers. Recent years have seen an exponential growth in biological data, which are usually no longer published in a conventional sense, but deposited in databases. Every year, the journal Nucleic Acid Research devotes an issue to present new databases. All these data need to be processed, analyzed, and incorporated into one’s own research focus. The buzz word is bioinformatics, used by almost everybody who once “blasted” (i.e. queried) a sequence against Genbank or calculated a distance tree. Still, we have to discriminate between bioinformatics as a tool and bioinformatics as a research field. What most biologists—and I—mean by bioinformatics, is that they perform computational biology, in contrary to experimental

biology. Whatever you call it, biologists have to treat increasingly large amounts of data and often depend on command-line-based software tools in order to perform front-line research.

Let me give you one vivid example: *No, a GUI has not been implemented yet. Are you a biologist?* Replies like this are not uncommon when bioinformatics software developers are asked whether there is a graphical user interface (GUI) for their software. I encountered this answer at the ISMB 2004 conference (Intelligent Systems for Molecular Biology) in Glasgow. Robert Edgar demonstrated his new sequence alignment software MUSCLE (Edgar 2004), which was awarded the best paper price. As in many other cases, this software was originally developed for Unix-like operating systems (e.g. Linux) and had no nice graphical user interface—it only runs in the command line. Today, ports to Windows and MacOSX exist and a web interface has been set up. Yet, the Linux, Windows, and MacOSX versions still have no GUI and the web interface is functionally limited; most program options are only available via the command line. Why do programmers commonly omit the GUI? One reason is that the bioinformaticians are interested in solving an algorithmic problem. In order to convince everybody that he or she can do better than others, the programmer has to implement his or her algorithm, i.e. write a program. He or she definitely does not want to spend a lot of time on implementing a GUI. This is commonly done by others. And then it depends on the competence, focus and will of the GUI programmer if he or she makes all the features of the background command line program accessible via the graphical interface. Now, without the skills to work in the command line, you might quickly be put off the peak of research tools.

Automation of high-throughput data retrieval from increasingly sophisticated devices puts masses of raw data on hard disks. As mentioned above, commercial applications often show practical limitations in processing these data. In order not to be trapped by the given functionality of a software application, modern biologists have to learn new skills. The setup of data processing channels is often determined by the file formats that the installed software can handle. These are often proprietary formats that bind the user to that particular software. Working in the field of functional genomics, I frequently observe the need for file format conversions. The task is often easy, like changing decimal delimiters, permuting columns, or fusing columns from different files, but the number of processed files is huge. There is a clear advantage for those scientists who are capable of automating such transformations. It all boils down to this: **Modern biologists require high-throughput data processing and analysis skills and this book shall assist you to gain them.**

1.5 The Future: Digital Lab Benches and Designing Organisms

Ironically, while language extensions like BioPython and BioPerl are good tools for experienced programmers who need to make software for biologists, they are usually too complicated to be useful to biologists themselves. Consequently, the

newest trend is the development of programming languages designed for biologists. A recent approach is BioBike (formerly BioLingua), a programmable knowledge environment for biologists (Massar et al. 2005, Elhai et al. 2009). This Lisp-based programming language and working environment currently integrates knowledge about 13 cyanobacterial strains. BioBike is not installed locally. Instead, every user gets his own working space online and can connect from wherever and whenever he or she wants. Darwin (Data analysis and retrieval with indexed nucleotide/peptide sequences), on the other hand, is an interpreted programming language that has been created from scratch (Gonnet et al. 2000). While BioBike currently focuses on genome analysis, Darwin is a biochemist's workbench primarily for peptide and nucleotide sequence analysis. These two examples clearly show the direction of future developments. The goal is to create digital benches that provide the researcher with easy to handle computational tools. The pipetting robot, the cDNA synthesis kit, as well as probe design or SNP (single nucleotide polymorphism) analysis will be intuitively usable. The biologist still needs to provide the data though.

Currently, at the front end of biotechnology and genetic engineering stands synthetic biology. This rather new field incorporates many research fields and disciplines all focussing at one final goal: the targeted design of organisms that fulfil a specific task. Bioinformatics and systems biology are key disciplines in this endeavor. Microsoft Research even developed a programming environment, VisualGEC (Visual Genetic Engineering of Living Cells), specifically designed for the envisioned workflow in synthetic biology (Pedersen and Phillips 2009). The future has just begun.

This boom poses new requirements for the training of biologists. About 20 years ago a number of molecular biological methods such as PCR (polymerase chain reaction) were included in the curricula and departments dealing with molecular biological aspects in different biological disciplines were erected. Around 10 years ago the same happened for bioinformatics and today we see a similar development in the emerging field of synthetic biology. Often these departments are anchored outside of biology, e.g., in mathematics, computer science or physics. Consequently, computational biology is taught to the specialist instead of all students. Like scientific data presentation or writing skills, knowledge of computational biology is assumed to be common knowledge. It remains to the initiative or the fortune of the student to acquire this knowledge as part of his training. But what skills have to be learned? What is needed by a biologist in order to process and analyze large sets of experimental data? In my opinion the basic skills are how (a) to use local relational databases, (b) to make best use of the Linux operating system and (c) to write little programs for data processing, analysis and visualization.

This book covers all these topics and shall provide you with a solid introduction.

Chapter 2

Content of This Book

This book aims at the total beginner. However, if you know something about computers but not about programming, the book will still be useful for you. After introducing the basics of how to work in the Linux environment, some great tools will be presented. Among these are the stream line editor `sed`, the script-oriented programming languages `awk` and `perl`, the data analysis and visualization tool `R`, and the relational database system `MySQL`. These utilities are extremely helpful when it comes to formatting and analyzing data files. After you have worked through all the chapters, you can use this book as a reference. The learning approach is absolutely practically oriented. Thus, you are invited to run all examples, printed in so-called Terminals, on your own! *If you face any problems: contact me!* Of course, I cannot help you if your non-unix-like-operating-system driven computer crashes continuously. However, if things connected to this book confuse you—or you even find errors—please let me know (Email: rw@biowasserstoff.de). Further information about this book, including lists with internet links and known errors, can be found at my homepage (<http://www.hs-mittweida.de/wuenschi>). You are very much welcome to supply me with good ideas for examples!

2.1 The Main Chapters

This book contains several chapters that focus on one particular concept, e.g. a programming language. Though, there is a progression—chapters usually depend on previous chapters. However, if you are already a command line geek, you might skip early chapters.

2.1.1 *Linux*

Linux is a multi-user multi-task operating system, originally based on *Minix*, which is an operating system similar to Unix. Linux was initially developed by Linus Torvalds

in 1991. It is an *open source* operating system. This means everybody who has programming knowledge can modify and improve the system; but it also means that everybody can download and install it. This is a main reason to choose Linux: you need invest no money except for the book itself. Still, the content of this book is valid for any Unix-like operating system like Mac OS X or CygWin, the free Unix emulator for Windows.

2.1.2 Shell Programming

The shell, also called terminal or console, will be our playground. Everything we do in this book is done in the shell. The shell can be seen as a command interpreter: we enter a command and the shell takes care of its execution; but we can also combine a number of commands and programs, including programming structures like decisions, in order to generate new functionality. Typical shell programs handle files and directories rather than file contents. A common task would be to convert all file extensions from *.txt* to *.seq*, make specific files executable or archive all recently changed files. Shell programming resembles DOS's programming language for *batch files*.

2.1.3 Sed

No, this is not the about the Socialist Unity Party of Germany (German: Sozialistische Einheitspartei Deutschlands, SED) that was governing the German Democratic Republic from October 1949 until March 1990. *sed* (stream **e**ditor) is used to perform basic text editing on an input text file (or data stream) and was written by Lee E. McMahon in 1973. *sed* does not allow for any interactions.

Figure 2.1 shows a sketch for an example of what *sed* can do. Here, the stop codon of a DNA sequence is replaced by the text “!STOP!”. *sed* is well suited to perform small formatting tasks like converting RNA to DNA, commas to points, tabs to semicolons and the like.

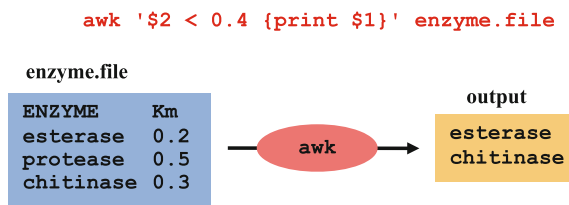
Fig. 2.1 Stream EDitor. What does Sed do?



2.1.4 AWK

AWK is a programming language for handling common data manipulation tasks with only a few lines of code. It was initially developed by Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan in 1977. This is where the name comes from. AWK is really a great tool when it comes to analyzing the content of data files. With AWK you can perform calculations, draw decisions, read and write multiple files. What is best is that AWK can be extended with your own designed functions. A typical task would be to fuse the content of files having one common field. Another typical task would be to extract data matching certain criteria. AWK forms the kernel of this book. After you have finished the chapter on AWK you should be able to (a) program basically anything you need and (b) learn any other programming language. The example shown in Fig. 2.2 shows one basic function of AWK. All enzyme names in the file *enzymes.file* are printed, if the corresponding Km value (do you remember enzyme kinetics and Michaelis-Menten?) is smaller than 0.4.

Fig. 2.2 AWK. What does AWK do?



2.1.5 Perl

Practical extraction and report language (Perl) arose from a project started in 1987 by Larry Wall. It combines some of the best features of the programming language C, Sed, AWK and shell programming and was optimized for scanning arbitrary text files, extracting information from those text files and printing reports based on that information. Perl is intended to be practical rather than being a beautiful language. It offers you everything a programming language can offer. Perl is often used to program web-based applications (known as CGI scripts), it provides database connectivity and there is even a Bioperl project which provides many tools biologists need. This book can introduce you to only the very basics of Perl. Still, you will get to the point where you learn how to write your own modules and generate small database files.

2.1.6 MySQL

When you happen to have several, possibly cross-linked, Excel tables, then you should consider to save them in a MySQL database. Imagine you have one table (or tab-delimited file) containing gene expression data. Another table contains gene annotations (Fig. 2.3).

Thus, there is a relationship between data in these tables. The power of MySQL (and other relational database systems) is to query over such relations. Furthermore, MySQL is a good storage place for such data because everything is available in one database. Many programming languages and software packages—e.g. R—can connect to MySQL.

Table: annotation

gene	function	metabolism
alr2938	iron superoxide dismutase	Detoxification
alr4392	nitrogen-responsive regulator	Nitrogen assimilation
alr4851	preprotein translocase subunit	Protein and peptide secretion
alr3395	adenylosuccinate lyase	Purine biosynthesis
alr1207	uridylate kinase	Pyrimidine biosynthesis
alr5000	CTP synthetase	Pyrimidine biosynthesis
all3556	succinate-dehydrogenase	TCA cycle

Table: expression

gene	expr_level
alr1207	8303
alr2938	10323
alr3395	1432
all3556	8043
alr4392	729
alr4851	633
alr5000	5732

```
mysql> SELECT a.gene, a.function, e.expr_value
-> FROM annotation as a, expression as e
-> WHERE a.gene = e.gene AND
-> a.metabolism REGEXP
-> "(Purine|Pyrimidine) biosynthesis" AND
-> e.expr_value < 5000;
+-----+-----+-----+
| gene | function | expr_value |
+-----+-----+-----+
| alr3395 | adenylosuccinate lyase | 1432 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql>
```

Fig. 2.3 MySQL. The query shown at the bottom right displays gene names and gene functions for all genes, which are involved in either purine or pyrimidine metabolism and which have an expression value less than 5000

2.1.7 R

There are a broad range of software tools available to perform data analysis and visualization. R is a completely free software that emulates S-Plus. S-Plus in turn was initially developed by AT&T's Bell Laboratories to provide a software tool for professional statisticians who wanted to combine state-of-the-art graphics with powerful model-fitting capability.

R is a very well established platform for scientists in general and computational biologists in particular. Besides from being a programming environment for statisti-

cal computing, R is also a data visualization tool. Several subject specific packages are available to analyze and visualize experimental data, e.g. for evolutionary biology, the evaluation of biochemical assays, nucleotide and amino acid sequence analysis, microarray data interpretation and more. Bioconductor is the name of an international project that brings developments from various research teams together and that provides tools for the analysis and comprehension of high-throughput genomic data.

The introduction given in this book shall set you on the right track to develop more and more sophisticated skills in using and combining available packages and apply them to your own scientific problems.

2.1.8 Worked Examples

This might be the most important part of this book. In Part VI on p. 343 I present four chapters that can be used for teaching and training. All are based on real-world problems and come with a detailed instruction of how to solve the problem. All exercises include data processing and visualization of some sort.

Most programmers would agree that one learns a programming language by using and adapting programs of others. This is learning by mimicking. You should go the same way with the worked examples. Start by following the instructions—then leave the path and start playing with both the data and data processing. Maybe you develop nice modifications to the examples and like to send them to me—I would be very happy. Last but not least, the worked examples shall give you a glimpse of where you could employ command line data processing in your own projects.

2.2 Prerequisites

In order to perform the exercises shown in this book—and there are lot—you need to have access to a computer running either Linux, Unix or Mac OS X (the newest Apple operating system) or the free Windows Unix emulator CygWin (see below). As you will learn soon, these systems are very similar. Thus, all the things we are going to learn will work on all Linux, Unix and Mac OS X computers. On a normal installation all required programs should be installed. Otherwise, contact the system administrator.

My personal recommendation is to install Linux as a virtual machine. This operating system independent approach is explained in Sect. 4.1.2 on p. 40 (see also Sect. 3.8.1 on p. 30).

Alternatively, you can install the free *Cygwin* Unix emulator on a computer running the Microsoft Windows operating system (from Win95 upwards, excluding WinCE) (see Sect. 3.8.2 on p. 30) or to start with *Knoppix* Linux (see Sect. 3.9 on p. 31).

Knoppix runs from a CD-ROM and requires no installation on the hard disk drive. Neither your operating system nor the data on your computer will be touched.

2.3 Conventions

What you see on your computer screen is written in `typewriter` style and boxed. I will refer to this as the *Terminal*. The data you have to enter are given behind the `$` character. Key labels are written in a box. For example, the key labelled “Enter” would be written as `(Enter)`. Commands that appear in the text are written in `typewriter`, too. When necessary, space characters are symbolized by “`_`” in the text. Thus, “`_ _ _`” means that you have to type three consecutive spaces. In the following example you would type `date` as input and get the current date as output. Thereafter, from lines 3-5 I indicate how I mark commands or output that spans several lines.

```

1  $ date
2  Thu Feb 13 18:53:26 CET 2003
3  $ # this is a very long comment line that does nothing but indicating how I      +
4    highlight commands or output that spans two or more lines; NOTE the plus signs +
5    at the end of the lines
6  $
```

In most cases you will find some text behind the terminal which describes the terminal content: in Terminal 1, line 1, we check for the current date and time.

Boxes labelled “Program” contain script files or programs. These have to be saved in a file as indicated in the first or second program line: `# save as hello.sh`. You will find the program under the same name on the accompanying website. As terminals, programs are numbered.

```

1  #!/bin/bash
2  # save as hello.sh
3  # This is a comment
4  echo "Hello World"
```

Finally, there are “Files”, which usually contain text files that are processed. At the end of most chapters you will find exercises. These are numbered, too. The solution can be found in Sect. A.7 on p. 431.

2.4 Additional Resources

This book is accompanied by a website (<http://www.staff.hs-mittweida.de/~wuenschi/doku.php?id=rwbook2>) and a blog (Fig. 2.4). Suggestions and comments are highly welcome.

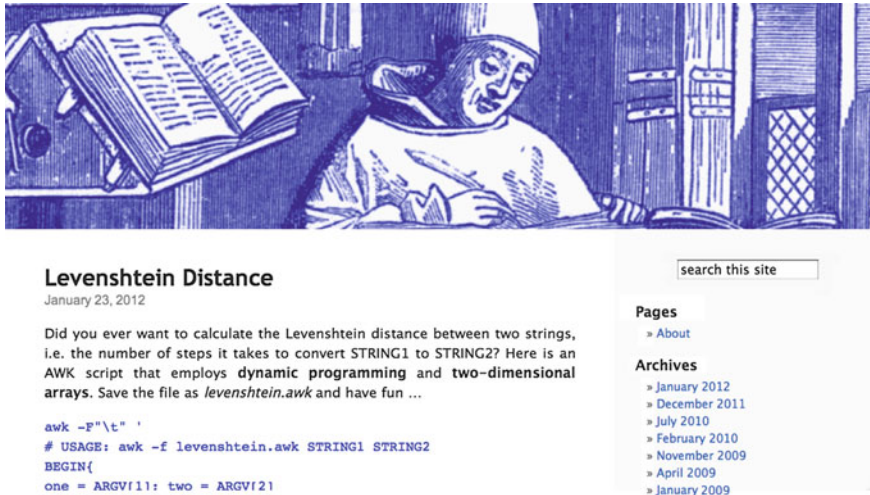


Fig. 2.4 Blog about AWK. Take a look or participate at my blog at <http://www.awkologist.com>

Part II

Computer and Operating Systems

Chapter 3

Unix/Linux

This chapter's intention is to give you an idea about what operating systems in general, and Linux in particular, are. It is not absolutely necessary to know all this. However, since you are going to work with these operating systems you are taking part in their history! I feel that you should have heard of (and learned to appreciate) the outline of this history. Furthermore, it gives you some background on the system we are going to restrain; but if you are hungry for practice you might prefer to jump immediately to Chap. 4 on p. 39.

3.1 What is a Computer?

First things first! What is really going on when we switch on a computer? Well, if the power supply is plugged in, the main units will start to be alive. The first thing that starts is the BIOS (basic input/output system). The BIOS has all the information, e.g., which processor type is installed, what size the hard disk drive has, if there is a floppy disk or CD-ROM drive and so on, to initiate booting. The most important computer components are the *processor* (CPU, central processing unit), some kind of *memory* (RAM: random access memory; HDD: hard disk drive; FDD: floppy disk drive) and the *input-output devices* (usually a keyboard and the screen). Figure 3.1 shows a microprocessor system, which has all components necessary to be called a computer.

As important as the hardware (those things you can touch) is the software (application programs that you run, like a word processor) of a computer. The heart, that is the immediate interface between the hardware and the software, is the *operating system*. An operating system is a set of programs that control a computer. It controls both hardware and software. Most desktop computers have a single-user operating system (like Windows 2000 or lower, or Mac OS 9 or lower), which means that only one person can use the computer at a time. Furthermore, many older operating systems, such as MS-DOS (MicroSoft Disk Operating System), can even handle only one job

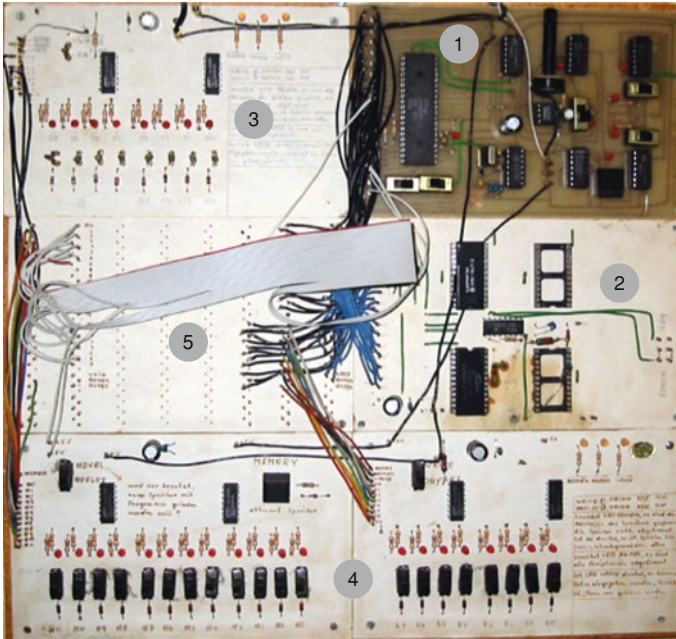


Fig. 3.1 Microcomputer System. A simple computer based on Zilog's Z80 processor (1). The system is equipped with 4kB RAM (random access memory) (2). Data input and output are realized with electric switches and lamps, respectively (3, 4). All components are connected via the system bus (5). The speed of the processor can be adjusted between single step and 1MHz. The whole system was built by the author in the late 1980s

at a time; but most computers, especially in these days, are able to execute dozens of applications in parallel. In order to use the computer's processor efficiently, one should provide it with several tasks. This is what multitasking operating systems, such as Unix or Linux, do; but they go even a bit further and allow access for several users in parallel. That is called a multiuser operating system. Thus, with Unix or Linux many people can use the computer at the same time and several jobs can be run in parallel. Of course, every user has to login to his account on the computer from his own terminal, consisting of at least a keyboard and a screen connected to a very simple and cheap computer.

3.2 Some History

In the early 1960s, computers were as large as a wardrobe, expensive, had little memory and processor power and could only be used by single users. In this environment, Ken Thompson and Dennis Ritchie, who worked for the Bell Laboratories of AT&T, developed the first version of Unix in 1969 (Fig. 3.2). At this time, Unix

was based on the research operating system *Multics* (multiplex information and computing system). Multics was an interactive operating system that was designed to run on the GE645 computer built by General Electric. With time, Multics developed into Unix, which ran on the PDP-7 computer built by Digital Equipment corporation (today Compaq). The new operating system had multitasking abilities and could be accessed by two persons. Some people called it Unics (uniplexed information and computer system). However, the operating system was limited to the PDP-7 hardware and required the PDP-7 assembler (this is the part which translates the operating system commands to processor code). It was Dennis Ritchie from Bell Laboratories who rewrote Unix in the programming language C. The advantage of C is that it runs on many different hardware systems—thus, Unix was now portable to hardware other than PDP-7 machines. Unix took off when Dennis Ritchie and Ken Thompson published a paper about Unix in July 1974 (Ritchie and Thompson, 1974). In the introduction to their paper, they wrote: “Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as \$40,000, and less than two manyears were spent on the main system software”.

3.2.1 Versions of Unix

With time, Unix became popular among AT&T companies, including Bell Laboratories and academic institutions like the University of California in Berkeley. From 1975 onward, the source code of Unix was distributed for a small fee. From that point on, Unix started to take off (Fig. 3.2).

There were only nearly a dozen worldwide Unix installations in spring 1974, three dozen were registered in spring 1975, around 60 in autumn 1975 and 138 in autumn 1976. AT&T itself was not very interested in selling Unix. Thus, from 1975 onward, Unix’s development took largely place outside Bell Laboratories, especially at the University of California in Berkeley. In contrast to proprietary operating systems like Microsoft’s DOS, Unix was now developed and sold by more than 100 companies including Sun Microsystems (Solaris), Hewlett Packard (HP/UX), IBM (AIX), and the Santa Cruz Operation (SCO-Unix). Of course, this led to the availability of different and partially incompatible Unix versions, which is a major problem for software developers. However, two Unix variants floated on top of all distribution:

AT&T System V—Developed in 1976 by the Bell Laboratories, the AT&T System V was distributed to many international universities. In 1989, AT&T founded the Unix-System Laboratories (USL) for further development of the source code of System V. In 1991, a cooperation between USL and the company Novell started, which led to UnixWare running on Intel platforms. In 1996, Novell sold its Unix department to the Santa Cruz Operation (SCO), which is still developing Unix.

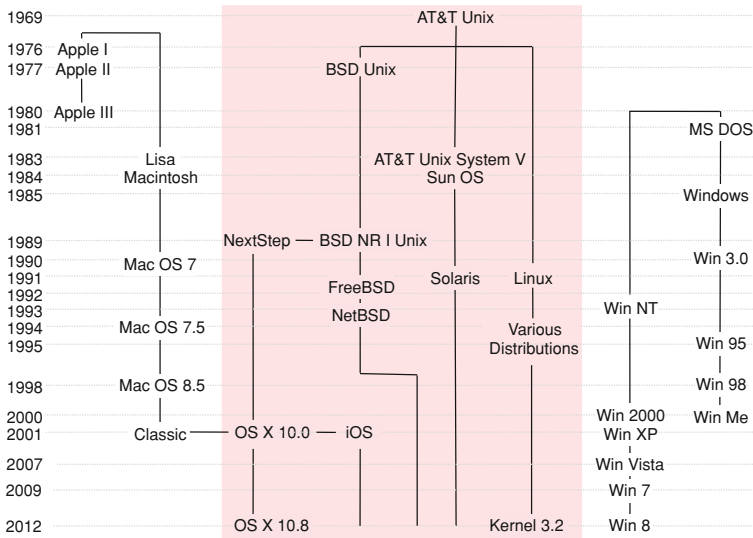


Fig. 3.2 Evolution of Operating Systems. Note the recent hybridization of Unix and Macintosh with Mac OS X in 2001. Unix-based operating systems are *shaded*

BSD v4 (Berkeley Software Distribution)—In parallel to AT&T, the University of California in Berkeley developed Unix up to version BSD v4. This version was running on Vax computers. In 1991, the spin-off company Berkeley Software Design Inc. was founded and, from that year on, has been selling BSD-Unix commercially. However, in parallel, there are still freeware versions available, namely FreeBSD and NetBSD. Apple's new operating system Mac OS X is actually based on BSD (see Sect. 3.7 on p. 29).

Thus, the presently available Unix versions are based on either AT&T System V or BSD v4 (Fig. 3.2). In order to secure the compatibility between different Unix version *The Open Group* was founded in 1996.

3.2.2 The Rise of Linux

Linux is a very young operating system. Its first version was distributed by Linus Torvalds in 1991. The following news group posting was the first official announcement of the new operating system:

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)

Newsgroups: comp.os.minix Subject: What would you like to see most in minix?

Summary: small poll for my new operating system

Message-ID: 1991Aug25.205708.9541@klaava.Helsinki.FI

Date: 25 Aug 91 20:57:08 GMT

Organization: University of Helsinki

Hello everybody out there using minix—I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things). I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them.

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes—it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have.

At that time, Finn Linus Torvalds was a student at the University of Helsinki (Finland). Since he offered the software freely and with source code via the internet, many programmers around the world had access to it and added more components to it, like improved file organization, drivers for different hardware components and tools like a DOS emulator. All these enhancements were again made available for free, including the source code. Thus, if there was any error with a software component it could be fixed by experienced programmers. A very important part of that concept was established already in 1984 by Richard Stallman with the *GNU* (GNU is not Unix) and *FSF* (Free Software Foundation) projects. GNU programs are freely available and distributable under the *General Public Licence* (GPL). In short, this means that GNU software can be used, developed, and redistributed for free or commercially. However, the source code must always be a free part of the distribution. Many Unix users used GNU programs as substitute for expensive original versions. Popular examples are the text editor *emacs*, the GNU C-compiler (*gcc*), and diverse utilities like *grep*, *find*, and *gawk*. Almost all open-source activities are now under the roof of the *Open Source Initiative* (OSI) directed by Eric S. Raymond.

As a matter of fact, Linux was not developed out of the blue but, from the beginning on, used GNU elements (like the operating system Minix). Once the GNU C-compiler was running under Linux, all other GNU utilities could be compiled to run under Linux. Only the combination of the Linux kernel with GNU components, the network software from BSD-Unix, the free *X Windows System* (see Sect. 3.3) from the MIT (Massachusetts Institute of Technology) and its *XFree86* port for Intel-powered PCs and many other programs, converted Linux to a powerful operating system which could compete with Unix.

3.2.3 *Why a Penguin?*

You might already have come across the nice penguin with his large orange feet and beak. The penguin is the mascot and symbol for Linux. Why is this? Well, although once bitten by a penguin in an Australian zoo, Linus Torvalds (the inventor of Linux) loves penguins (Torvalds and Diamond 2001). Once the idea to use a penguin as logo for Linux was born, Linus Torvalds screened a variety of suggestions and finally chose the version from Larry Ewing, a graphics programmer and assistant system administrator at the Institute for Scientific Computing at the A&M University in Texas, USA. The penguin's name is *Tux*, which is derived from the dinner jacket called tuxedo (short, tux).

3.2.4 *Linux Distributions*

There is one major problem when many people around the world develop and update an operating system: it is an awful task to collect all necessary and up-to-date components from the internet. Thus, different companies appeared, which took over the job and distributed complete sets of the Linux kernel with software packages, drivers, documentation, and more or less comfortable installation programs and software package managers. Currently, there are many different Linux distributions on the market, the most famous among them are: Ubuntu, Debian, Fedora, Red Hat, SuSE, Debian, Mandrake, and Knoppix. The good thing with Knoppix Linux is that you can run it from the CD-ROM. It does not require any installation and does not touch your hard disk drive at all, although you have access to it if you want. Therefore, it is well suited for playing around and I highly recommend its use for all the exercises in this book (see Sect. 3.9 on p. 31).

Finally, there are some minimal distributions available on the net. This means you can have Linux on two floppy disks or so and install a minimum system on an old computer (i396 or i486).

3.3 X-Windows

We will work only with the command line. That looks a bit archaic, like in the good old DOS times. Anyway, Linux also provides a *graphical user interface* (GUI). In fact, it provides many GUIs and thus is much more powerful than Microsoft Windows. Graphical user interfaces are systems that make computing more ergonomic. Basic elements are windows which are placed on the screen (desktop). Nowadays, no operating system without a graphical user interface could survive. I guess you know what I am talking about from your Windows system.

The GUI for Unix-based operating systems is called *X Windows System*, or just X. It was developed in 1984 at the Massachusetts Institute of Technology (MIT) and initially called "X Version 10". It was part of a larger project called Athena. The

goal of that project was to integrate many different devices in one GUI. In 1987 “X Version 11, Release 1” was made public. From then on, the software was developed by the community and is now available as “X Version 11, Release 6”, also called “X11R6”.

X-windows is completely network based. This means that the output of a program on computer A can be visualized on a screen connected to computer B. Computers A and B can be at completely different places on the Earth. This has the great advantage that one can run programs on a powerful remote computer and visualize the output on a rather simple local machine.

Since there is a great diversity of graphical user interfaces, things are a bit different when you change from one system to another. In 1993, the companies developing and distributing Unix (Hewlett-Packard, IBM, The Santa Cruz Operation Inc., Sun-Soft, Univel and Unix Systems Laboratories) formed an alliance and agreed on a common standard called *Common Desktop Environment* (CDE). CDE is a commercial product based on X11R6. The most common desktop environments for Linux are *KDE* (<http://www.kde.org>) and *Gnome* (<http://www.gnome.com>). Both are freely available.

3.3.1 How Does it Work?

The graphical desktop can be split into several parts: the *X-server*, the *X-clients*, the *window manager*, and the *desktop*; but the Linux desktop is something different from a Windows desktop.

The X-server is hosted by the computer (or graphical terminal station) you are sitting in front of. An X-server is nothing more than the black and white chess pattern you see very shortly before the graphical background of your window manager appears. The X-server handles the graphical presentation and is responsible for the communication between the hardware (in particular the graphic card, which handles the screen) and the software (the X-programs). If you would run only an X-server, no graphics would be possible. It offers neither a menu, nor windows or any other features you need. Here, the X-client comes into play. The work of the X-server is done with X-clients. These X-clients use libraries that are integrated in the X-server and contain the information how to display graphics. If you see an X-terminal (console) on your monitor, it is an X-client. The communication between X-server and X-client works through the network. That is the reason why you can start an X-client on any computer in the network and see it somewhere else (this is the way the X-terminals work; you start the programs on a powerful server and sit in front of a simple terminal). Even if you work on an isolated computer and have no network card installed, X-client and X-server communicate via the network. In that case, Linux simulates a network (loopback). X-server and X-clients alone are not really comfortable to work with. Useful functions like “Maximizing”, “Minimizing”, and “Close Window” are not included in the functionality of the X-server and X-clients but made available by a window manager. There are several window managers available, like FVWM (<http://www.fvwm.org>), IceWM

(<http://www.icewm.org>), Window Maker (<http://www.windowmaker.org>), Sawfish, formerly Sawmill (<http://www.sawmill.sourceforge.net>) and Metacity (<http://www.gnome.org/softwaremap/projects/Metacity>).

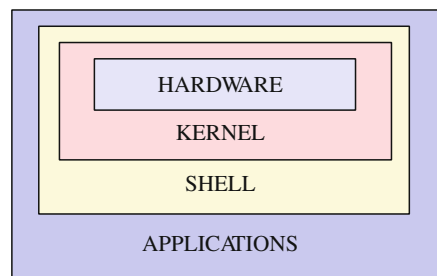
A long time ago, only these three components (X-server, X-client, and window manager) existed. However, in the past years, an additional “thing” has come into existence: the desktop. The desktop offers functionalities similar to those we are used to from Windows, like putting program and file icons onto the desktop, which can be started or opened by double clicking, respectively. However, with Linux, usually a single click is sufficient. Of course, icons existed long before (e.g. after minimizing a window with FVWM), but the functionality was not as great as with a desktop. KDE and Gnome are two popular desktops running on Linux. They come with their own window managers, but both programs could be used with alternative window managers as well.

3.4 Linux Architecture

We have already heard that Unix and Linux are operating systems. How are they organized? In principle, we distinguish four components: the *hardware*, the *kernel*, the *shell*, and *applications* (Fig. 3.3).

The hardware, i.e., the memory, disk drives, the screen and keyboard, and so on, is controlled by the kernel. The kernel is, as the name implies, the heart of Linux. Only the kernel has direct control over the hardware. Linux’s kernel is programmed in C. It is continuously improved; this means errors are corrected and drivers for new hardware components that are available are included. Thus, when you buy a Linux version, the kernel version is usually indicated. The newest kernel is not necessarily the best: it might have new errors. For us, the users, it is very uncomfortable to communicate with the kernel. Therefore, the shell was developed. The shell is an envelope around the kernel. It provides commands to work with files or access floppy disk drives and such things. You are going to learn more about the shell in Chap. 8 on p. 97. Finally, there are the applications like OpenOffice or ClustalX or awk or perl. These communicate with the kernel using system calls. You see, the Linux operating system is clearly structured. This is part of Unix’s philosophy: “Keep it simple, general and extensible”.

Fig. 3.3 The Kernel. The kernel is the heart of an operating system. It is wrapped around the hardware and accessed via the shell



3.5 What is the Difference Between Unix and Linux?

Essentially: the price! And: Linux is not free Unix but a Unix-like operating system! Otherwise, for the user, the differences are almost invisible. One point is that Linux runs on a wide range of different processors. You probably have heard of Intel processors like Pentium, Celeron, Core i5, and so on. Most personal computers have processors produced by either Intel or AMD, although other producers like Cyrix exist. Linux has been ported (adapted) to all these processors. Unix is much more restricted in this respect.

3.6 What is the Difference Between Linux and Windows?

Essentially: standardization! It should be clearly stated that Windows provides a stronger standardized system than Linux. This is the reason why Windows is much more widespread and almost all companies design their software for the Windows platform.

Of course, apart from standardization there are some hard facts, which make both operating systems clearly different. For example, with the exception of Windows NT, which also runs on Alpha processors, the desktop and server-oriented Windows operating systems run only on Intel and AMD processors. Furthermore, Unix-based operating systems are much closer to the data. Thus, it is much easier to format and analyze files.

3.7 What is the Difference Between Linux and Mac OS X?

Essentially: None! In fact, the latest version of Apple's operating system (*Mac OS X*, Macintosh operating system 10) is based on BSD, which is a Unix clone (see Sect. 3.2.1 on p. 23). If you want to work on a Unix/Linux-like terminal, you can run the application *Terminal*. It is a regular program that can be found under "Applications → Utilities". It is possible to do everything you learn in this book in the Mac OS X Terminal, too. If you encounter problems, please report them to me. Apple has also developed its own graphical user interface, called *Aqua*. It basically works as X-Windows (see Sect. 3.3 on p. 26) does. If you wish, you can even install native X-Windows and run it on the Mac. I must admit that Aqua is the crown jewel of Mac OS X. It is lightyears ahead of KDE or Gnome (see Sect. 3.3 on p. 26) and makes Mac OS X the most user-friendly Unix-based operating system around. There has been a whole bunch of software ported to Mac OS X, including *Open Office* and the graphic program *Gimp*. There is an interesting project called *Fink*. The Fink Project wants to bring the full world of Unix open-source software to Mac OS X. They modify Unix software so that it runs on Mac OS X and make it available for

download as a coherent distribution. The Fink Project uses a special tool in order to facilitate easy installation procedures. Take a look at *fink.sourceforge.net*.

Why did Apple switch to Unix? In 1985, Steve Jobs, the founder of Apple, left to start the company NeXT, whose NeXTStep operating system was based on BSD Unix. When Apple bought NeXT in 1996, Jobs, NeXTStep (then called OpenStep) and its Terminal program came along with it. From there, it was only a small step to fuse both systems.

3.8 One Computer, Two Operating Systems

You might not wish to eliminate your Windows operating system before knowing how much you like Unix or one of its derivatives. A common solution to this problem is to install both operating systems on the same computer as a *dual boot* system. This is only possible with Linux and Windows, since both can use the same processor architecture (see Sects. 3.5 and 3.6). When you start up you have to choose which operating system you want to work with. If you want to write a document in *Microsoft Word*, you can boot up in Windows; if you want to run *Genesis*, a general-purpose neural network simulator for Linux, you must shut down your Windows session and reboot into Linux. The problem is that you cannot do both at the same time. Each time you switch back and forth between Windows and Linux, you have to reboot again. This can quickly get tiresome. Therefore, you might want to use two operating systems on one computer—in parallel! There are several possibilities.

3.8.1 VMware

A powerful solution is offered by the software package *VMware*. With VMware, you can either run Linux on top of Windows or vice versa. The great advantage lies in the fact that you really run both operating systems at the same time. You switch to the other operating system as you would switch from one program to another—just activate the application window. Furthermore, you run the real operating system, no emulator, which only imitates an operating system as is the case with CygWin (see below). The disadvantages, namely calculation power and high RAM requirements, are out of date. Modern supermarket computers for 400 Euro or US\$ already offer enough performance. Take a look at their homepage (<http://www.vmware.com>) for more information.

3.8.2 CygWin

Cygwin is a Unix environment for Windows. It consists of two parts: (a) a dynamic link library (cygwin1.dll), which acts as a Unix emulation layer providing substantial

Unix API (application programming interface) functionality and (b) a collection of tools, ported from Unix, which provide Linux look and feel. In addition, CygWin is really easy to install. Take a look at *cygwin.com* about the installation procedure and make sure you install the following packages: `bash`, `gawk`, `sed`, `grep` and `perl`.

3.8.3 Wine

Wine is a Unix implementation of the Win32 Windows libraries, written from scratch by hundreds of volunteer developers and released under an Open Source license. Thus, Wine is the counterpart of CygWin. The Wine project (<http://www.winehq.com>) started in 1993 as a way to support running Windows 3.1 programs on Linux. Bob Amstadt was the original coordinator, but turned it over fairly early on to Alexandre Julliard, who has run it ever since. Over the years, ports for other Unix versions have been added, along with support for Win32. Wine is still under development, and it is not yet suitable for general use. Nevertheless, many people find it useful in order to run a growing number of Windows programs. There is an application database for success and failure reports for hundreds of Windows programs.

3.8.4 Others

Of course, there are many other possibilities and you should check out yourself which solution is suitable for your setup. With *Win4Lin* you can run Windows programs under Linux (<http://www.netraverse.com>). Another option might be *Plex86* (<http://plex86.sourceforge.net>). Plex86 is an extensible free PC virtualization software program which will allow PC and workstation users to run multiple operating systems concurrently on the same machine. Plex86 is able to run several operating systems, including MSDOS, FreeDOS, Windows9x/NT, Linux, FreeBSD, and NetBSD. It will run as much of the operating system and application software natively as possible, the rest being emulated by a PC virtualization monitor. I am sure there is more software available and even more to come. Check out yourself.

3.9 Knoppix

Knoppix is a free Linux distribution developed by Klaus Knopper, which can be run from a CD-ROM. This means you put the CD-ROM into your computer's CD-ROM drive, switch on the computer and start working with Linux. Knoppix has a collection of GNU/Linux software, automatic hardware detection and support for many graphics cards, sound cards, SCSI and USB devices and other peripherals. It is not necessary to install anything on the hard disk drive. Due to on-the-fly decompression, the CD

carries almost 2 GB of executable software. You can download or order Knoppix from (<http://www.knopper.net/knoppix>).

The minimum system requirements are an Intel-compatible CPU (486, Pentium or later), 20 MB of RAM for the text-only mode or at least 96 MB for the graphical mode with KDE. At least 128 MB of RAM is recommended if you want to use OpenOffice. Knoppix can also use memory from the hard disk drive to substitute missing RAM. However, I cannot recommend using this option, since it decreases the performance. Take a look at the Knoppix manual in order to learn how to do this. Of course, you need to have a bootable CD-ROM drive or a boot floppy if you have a standard CD-ROM drive (IDE/ATAPI or SCSI). For the monitor, any standard SVGA-compatible graphic card will do. If you are one of those freaks playing the newest computer games and always buying the newest graphic card you might run into trouble; but I am sure that you then own at least one out-dated computer you can use for Linux. As a mouse you can either use a standard serial or PS/2 or an IMPS/2-compatible USB-mouse.

Before you can start Knoppix, you need to change your computers BIOS (basic input/output system) settings to boot from the CD. When you start up your computer you are normally asked whether you want to enter the system setup (BIOS). Usually, one of the following key (combinations) is required: **(Del)** or **(Esc)** or **(F2)** or **(Ctrl)+(Alt)+(Esc)** or **(Ctrl)+(Alt)+(S)**. When you succeed in changing the settings you are done; but be careful not to change anything else! That might destroy your computer. However, if your computer does not support the option to boot from the CD-ROM, or you are afraid of doing something wrong, you have to use a boot disk. You can create this disk from the image in the file `/KNOPPIX/boot.img` on the CD-ROM. Read the manual on the Knoppix CD-ROM for more details on this issue. Once prepared, you put the CD in the drive and power up the computer. After some messages the system halts and you see the input prompt `"boot:"`. Now you can hit **(F2)** and optimize Knoppix to your needs. For example, `knoppix lang=de` enables the German keyboard layout.

3.9.1 Vigyaan Knoppix for Biosciences

Vigyaan is a special Knoppix distribution that has been designed to meet the needs of computational biologists and chemists. It has been developed by Pratul Agarwal and is available for free download at <http://www.vigyaancd.org>. Although, the Webpage has not been updated since 2006, I can still recommend Vigyaan Knoppix as a good starting point to play with Linux. The big advantage is that tools and data are included on the CDROM, e.g. genomes from *Escherichia coli* K12 and the cyanobacterium *Synechococcus* sp. WH 8102, molecular viewers like RasMol and tools like Jmol, Clustal, BLAST and BioPerl, to name only some. Vigyaan, by the way, is the Sanskrit word for science and technology.

3.10 Software Running Under Linux

Of course, it is not my intention to name all available software packages here. The objective is to give you an overview of which software is available and which Windows-based software it substitutes. Needless to say that almost all software is freely available.

3.10.1 *Bioscience Software for Linux*

There is a lot of software available for all fields of academics. Being a biologist myself, I will restrict myself to list some important biological software packages available for Linux. An updated list can be obtained from <http://www.bioinformatics.org/software>.

3.10.1.1 Sequence Analysis Tools

Emboss—Emboss is the **E**uropean **M**olecular **B**iology **O**pen **S**oftware **S**uite. Emboss is freely available and specially developed for the needs of molecular biologists. Currently, Emboss provides a comprehensive set of over 100 sequence analysis programs. The whole range from DNA and protein sequence editing, analysis, and visualization is covered. Restriction analysis, primer design, and phylogenetic analysis can all be performed with this software package. Take a look at <http://www.emboss.sourceforge.net>.

Staden Package—The Staden Package is a software package free for academics (charge for commercial users) including sequence assembly, trace viewing/editing and sequence analysis tools. It also includes a graphical user interface to the Emboss suite. More information can be obtained at <http://www.staden.sourceforge.net>.

Blast+—Blast (**B**asic **L**ocal **A**lignment **S**earch **T**ool) is a set of similarity search programs designed to explore online sequence databases. Alternatively, you can set up your own local databases and query those. In Chap. 9 on p. 115 we will download, install and run this program.

ClustalW—ClustalW is probably the most famous multiple-sequence alignment program available. It is really powerful and can be fine-tuned by a number of program options. In Chap. 9 on p. 115 we will download, install, and run this program.

3.10.1.2 Molecule Structure Analysis

Dino—Dino (<http://www.dino3d.org>) is a real-time three-dimensional visualization program for structural biology data. Structural biology is a multidisciplinary research

area, including X-ray crystallography, structural NMR, electron microscopy, atomic-force microscopy, and bioinformatics (molecular dynamics, structure predictions, surface calculations etc.). The data produced by these different research areas are very diverse: atomic coordinates (models and predictions), electron density maps, surface topographs, trajectories, molecular surfaces, electrostatic potentials, sequence alignments, etc. Dino aims to visualize all these structural data in a single program and to allow the user to explore relationships between the data. There are five data types supported: structure (atomic coordinates and trajectories), surface (molecular surfaces), scalar fields (electron densities and electrostatic potentials), topographs (surface topography scans), and geoms (geometric primitives such as lines). The number and size of the data the program can handle is limited only by the amount of working memory (RAM) present in the system.

3.10.2 *Office Software and Co.*

Apart from a growing number of science-related programs, Linux gets more and more attractive for personal users. This is because almost everything known from Windows is nowadays also available for Linux—and more...

3.10.2.1 Editors

Editors—You can choose between a variety of different text editors. There is the whole range from single line to graphical editors available. Just to name a few: `emacs`, `vi` (see Sect. 7.2.1 on p. 86), `ed`, `pico` (see Sect. 7.2.2 on p. 88), `gedit`, `kedit`.

3.10.2.2 Office

Office Applications—With *OpenOffice* (<http://www.openoffice.org>) there is a whole bunch of office applications available. Among these are Writer (like MS Word), Calc (like MS Excel), Impress (like MS Powerpoint), Draw (like Corel Draw), HTML Editor, Math Editor, and others. Common MS Office files can be opened, edited and even saved in MS Office format. OpenOffice uses for its files the markup language XML (Extended Markup Language). Furthermore, the files are automatically compressed.

Of course, there are more options. *KOffice*, integrated to the KDE desktop, offers office applications and there are a number of stand-alone programs.

3.10.2.3 Graphics

Image Processing—*Gimp* (<http://www.gimp.org>) is a very powerful image processing with a number of plugins and which can do basically everything Adobe Photoshop can do.

Vector Graphics—Apart from OpenOffice Draw you can work, for example, with *XFig* (<http://www.xFigure.org>) to create vector graphics.

3.10.3 Graphical Desktops

Graphical Desktop—All versions of Microsoft Windows desktops look pretty much the same. This is an advantage when it comes to, e.g., changing from one version to another. However, it can be boring. It does not matter if you install different themes because it would function in the same way as before, and you cannot change the functionality. If you are missing a function you can only hope that it will become available later—when you have to buy it, of course. Linux has a completely different philosophy about graphical desktops. Variety is the theme (see Sect. 3.3.1 on p. 27)!

3.10.4 Others

Others—Okay, as you can imagine there are zillions of software packages available. Basically, all needs will be satisfied. You can synchronize your Palm, install relational databases, choose from a bunch of web browsers and email programs, and so on ...

Part III

Working with Linux

Chapter 4

The First Touch

In this chapter, you will learn the basics in order to work on a Unix-based computer: *login*, execute *commands*, *logout*. Everything you are going to do is happening at the command line level. This means, the look and feel will be like in good old DOS times. It will look like stone-age computing. However, you should remember that although a graphical interface is often nice and comfortable, it consumes a lot of power and only hinders us from learning what is really important. Furthermore, the Linux command line is extremely powerful. You will soon get accustomed to it and never want to miss it again.

Let us face it ...

4.1 Just Before the First Touch

Oh. Wait. If you are not working on a Linux or MacOSX computer and if you do not have access to such a machine via the network, then it is a good time now to think about how to gain access ...

I described some possibilities in Sect. 3.8 on p. 30. Here, I focus on the Ubuntu Linux distribution. Ubuntu is based on Debian Linux and was released in 2004. Since then, Ubuntu became very popular and you will most probably find somebody who runs it and may help you if you face problems.

4.1.1 Running Linux from USB or CDROM

If you are not yet sure about installing Ubuntu on your computer, you can try it out without affecting your current system. In any case, you have to download a Linux distribution as an ISO-image file. Go to <http://www.ubuntu.com> and download Ubuntu Desktop (I used version 12.04 LTS 32 bit; file *ubuntu-12.04-desktop-i386.iso*).

Ubuntu provides very well-written instructions on how to burn the ISO-image on a CDROM or prepare a bootable USB-stick. Take a look at their website.

You are now able to boot Ubuntu Linux from the CDROM or USB-stick, respectively, without any changes to your hard disk drive, i.e., installed operating system (see also Sect. 3.9 on p. 31). Enjoy.

The disadvantage of running Linux from a bootable device is that you have no access to your original operating system, hmm. The solution brings a virtual machine. With a virtual machine, you can run almost any operating system on any operating system—you can even run several operating systems in parallel, if your computer has the power. There are many different virtualization solutions available and even more different Linux distributions. In this section, I will show how to set up free Ubuntu Linux on a free Oracle virtual machine.

4.1.2 Running Linux as a Virtual Machine

4.1.2.1 Installing Oracle VM VirtualBox

Oracle Corporation is a big player when it comes to software. Since its foundation in 1970 in the Silicon Valley in California/USA, Oracle is best known for its Oracle database system. Due to a clever acquisition in 2010, it owns Sun Microsystems and with it MySQL (Chap. 16 on p. 295), Solaris Unix, OpenOffice, the programming platform Java and ... and ... and VirtualBox.

Go to <https://www.virtualbox.org> and download the VirtualBox platform package that matches your host (i.e. your computer's) operating system (I work with version 4.1.18). Follow the installation instructions.

4.1.2.2 Installing Ubuntu

I assume that you installed Oracle VM VirtualBox on your computer and that you downloaded Ubuntu Desktop, too. Now start the VirtualBox and click “Machine” → “New...”. This starts the installation wizard. First, you have to choose a name and select the operating system (OS) type you are going to install. Choose *CompBiol* as name, *Linux* as operating system, and *Ubuntu* as version. Continue. Then, you have to set the amount of memory to be allocated to the virtual machine. If you have no clue what RAM (random access memory) is or how much of it you have, just go with the suggested value, else choose about a quarter of your RAM. Continue. Now, tick “Start-up Disk” and “Create new hard disk”. This creates a container file on your hard disk that will be used by the virtual machine. The size will be set later. Continue. Now, you have to choose the file type of the virtual hard disk. If you are using other virtualization software and possibly move the image, then use VMDK. VMDK was developed by and for VMWare, but Sun xVM, QEMU, VirtualBox, SUSE Studio, and .NET DiscUtils also support it. I opt for the native (default) VDI.

Continue. For the disk size, I recommend a dynamically allocated file size. This way the image files growth with its task (your demand), as our liver does. Continue. You can leave the defaults for the virtual disk location and size. Continue & Create & Create.—Now, create a shared folder. That will simplify file exchange between your virtual and real machine. Choose a folder and tick the “Auto-mount” option.

Start the machine: “Machine” → “Start”. The first run wizard opens. Continue. Select the file *ubuntu-12.04-desktop-i386.iso* as installation medium. Continue & Start. When you see the “Install” window, choose to install and do not try Ubuntu. Here, you can also choose your preferred language. Continue. If you wish, tick the box to download updates while installing. Continue. Erase and Install. Continue & Install Now. During the installation process, you will be asked for your location, keyboard layout, your name, a computer name, and a password. When requested, press “Restart Now”. After you have restarted and logged into the system, you should install the Guest Addition: “Devices” → “Install Guest Additions...”. After entering your password and another restart, you are done.

One final step is necessary to gain access to the shared folder. As you will learn later, users belong to groups in Linux. Unfortunately, access to auto-mounted shared folders is only granted to the users in the group *vboxsf*. The quickest way to join this group is by execution of a terminal command. But where is the terminal? Figure 4.1 helps you how to find it.

Follow the commands in Terminal 2 in order to add yourself to the group *vboxsf*. With `groups rw`, where *rw* is your username, we check to which group a user belongs. The command in line 3 adds user *rw* to group *vboxsf*. Unfortunately, you have to reboot once again.

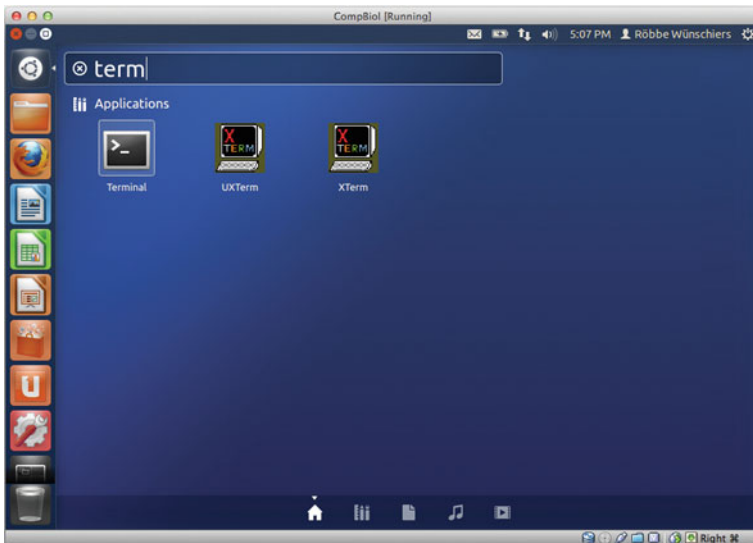


Fig. 4.1 VirtualBox. Ubuntu Linux as a Virtual Machine

```

1 $ groups rw
2 rw : rw adm cdrom sudo dip plugdev lpadmin sambashare
3 $ sudo usermod -G vboxsf -a rw
4 [sudo] password for rw:
5 $ groups rw
6 rw : rw adm cdrom sudo dip plugdev lpadmin sambashare vboxsf
7 $

```

Leave the terminal open because this is our playground throughout this book.

4.1.2.3 Upgrading the System

After installation of Ubuntu, you should upgrade its components to the newest version to ensure security. Although you could use Ubuntu's software center, I show the terminal way to do this in Terminal 3.

```

1 $ sudo apt-get update
2 [sudo] password for rw:
3 ...
4 Fetched 185 kB in 5s (31.6 kB/s)
5 Reading package lists... Done
6 $ sudo apt-get upgrade
7 Reading package lists... Done
8 Building dependency tree
9 Reading state information... Done
10 The following packages have been kept back:
11   linux-generic-pae linux-headers-generic-pae linux-image-generic-pae
12 The following packages will be upgraded:
13   accountsservice apparmor apport apport-gtk apt apt-transport-https apt-utils
14 ...
15 297 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
16 Need to get 61.6 MB/148 MB of archives.
17 After this operation, 4,163 kB of additional disk space will be used.
18 Do you want to continue [Y/n]?
19 ... # THIS WILL TAKE A WHILE
20 $

```

The crucial commands can be found in lines 1 and 6. The remaining lines show parts of the output. `sudo apt-get update` updates the internal package list, while `sudo apt-get upgrade` upgrades all installed software packages to the newest version. Both commands require administrator (root, super user) rights (`sudo` means: super user do) and a connection to the Internet.

4.1.2.4 Installing MySQL and R

Both MySQL (<http://www.mysql.com>) and R (<http://www.r-project.org>) are powerful tools to organize, analyze, and visualize data that are dealt with in detail in Part V on p. 293 and applied in Part VI on p. 343. To install the free MySQL database server and client, you need to execute

```
sudo apt-get install mysql-server mysql-client
```

During the installation process, you will be asked to set an administrator (root) password for the database system. I strongly recommend you to do so and even strongly recommend you to remember that password. Working with MySQL will be explained in Chap. 16 on p. 295.

With

```
sudo apt-get install r-base
```

you install the free software environment for statistical computing and graphics named R. An introduction into R is given in Chap. 17 on p. 317.

4.1.2.5 Installing Bioscience Software

The advantage of a widely used Linux version as Ubuntu is that many software packages have been made available as easy to install Linux packages. The following commands will help you to install bioscience software used in this book, in particular in Part VI on p. 343. The advantage of installing such packages over installing them manually is that there is no need to care about the path to the program. For e.g., you can execute Jmol by typing `jmol` from anywhere in the system. Furthermore, the Linux system informs you about new releases of the packages.

Jmol `sudo apt-get install jmol`—A molecule structure viewer.

ClustalW `sudo apt-get install clustalw`—A command line sequence alignment tool.

ClustalX `sudo apt-get install clustalx`—A graphical sequence alignment tool.

BLAST+ `sudo apt-get install ncbi-blast+`—The newest version of the basic local alignment search tool of the National Center for Biological Information (NCBI), U.S.

BLAST `sudo apt-get install blast2`—The old version of BLAST.

Finally, I highly recommend you to take a look at the Emboss package (<http://emboss.sourceforge.net>) that can be installed with `sudo apt-get install emboss`.

4.2 Login

The process of making yourself known to the computer system and entering to your Linux account is called logging in. There are two prerequisites: first you need to connect to the computer, and second you must have an account on that computer. This is like withdrawing money from a bank account. You must identify yourself to the cash machine before you get any money and you must, of course, have a bank account (with some money in it). There are several ways to connect to a Unix-based computer.

4.2.1 Working with CygWin or Mac OS X

In case you want to work with Apple's Mac OS X operating system (see Sect. 3.7 on p. 29) or the Unix emulator CygWin which runs in the Windows environment (see Sect. 3.8.2 on p. 30), then you need not care about any password. When you start CygWin, you will directly end up at the command line of the bash shell. In Mac OS X, you find the *Terminal* application under "Applications → Utilities". Again, starting the application will immediately bring you to the command line. In any case, you should look up your username with the command `whoami` (just type `whoami` and then press `(Enter)`).

4.2.2 Working Directly on a Linux Computer

You might have installed Linux on your computer or ran it from a CD-ROM on your computer (like the Knoppix distribution of Linux, see Sect. 3.9 on p. 31). In that case, you boot the computer and get directly to the login screen. Depending on the system settings, this might be either a graphical login or a command line login. In both cases, you enter your *username* and *password* (with Knoppix you do not need a username and password; you are directly locked in as user *knoppix*). With the graphical login, you immediately end up on a nice-looking graphical desktop—most probably KDE or Gnome. In that case, you have to open a terminal window in order to follow the examples given in this book. If the command line login appears after booting the computer, you have to follow the instruction given in Terminal 4 on the next page.

4.2.3 Working on a Remote Linux Computer ...

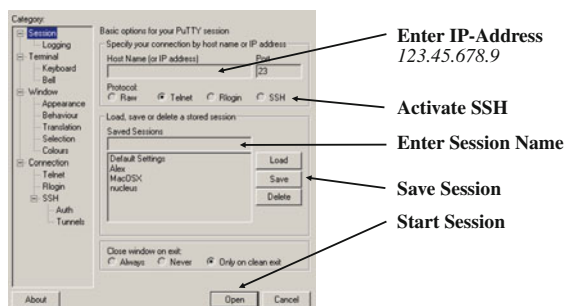
A quite common tool to connect to a remote computer is *telnet*. However, it is very insecure. Nowadays, most Linux systems do not allow telnet sessions. Chapter 6 on p. 71 gives an overview of how you can securely login to a remote Linux system from either Linux or MacOSX.

4.2.3.1 ... from a Windows Client

The freeware Windows application PuTTY allows one to connect to a Linux Server with a secure protocol (SSH). The program can be downloaded at: <http://www.chiark.greenend.org.uk/~sgtatham/putty>.

Although there are many possible settings, you basically need the IP address of the computer on which you have an account (see Fig. 4.2).

Fig. 4.2 The PuTTY Configuration Window. The most important information needed is the IP address or host name of the computer you want to work on remotely



After connecting, a login window pops up. Here, you have to enter your username and password. The windows content might look pretty much like Terminal 4.

```

Terminal 4: Logging In
1 login as: Freddy
2 Freddy@134.95.189.1's password:
3 Last login: Mon Mar 17 14:04:07 2003 from 134.95.189.37
4 [Freddy@nukleus Freddy]$

```

In Terminal 4, line 1, the username *Freddy* is entered. Then, line 2 pops up asking for the password of *Freddy* at the computer named *134.95.189.1*. Note: Usually the computer's name is not a number but a name. You will not see anything happening on the screen while you type in your password. You type blind. This is for reasons of security. After successful login, lines 3 and 4 pop up. Line 3 shows the date of your last login and from where you last logged in (in this case from the computer with the network identification number *134.95.189.37*). Finally, as shown in line 4, you are logged in to the system and the “\$” character indicates that the command line is ready for input. Line 4 is also called the *shell prompt*; the dollar character “\$” is the prompt character. Shell prompts usually end with \$ or %. The prompt can be customized, thus your own shell prompt might look different. A prompt that ends with the hash character (#) usually means that you are logged in as *superuser* or *root*. The root account belongs to the administrator who has access to all system components and owns all rights. In Terminal 4, the shell prompt tells you that you are connected as user *Freddy* to the computer *nukleus*, and you are currently in the directory *Freddy* (that is *Freddy*'s home directory). The messages you see at login differ from system to system; but this general outline should work out in most cases. I guess it is clear that you have to press the **(Enter)** or **(Return)** key after you have entered your name and password, respectively. Furthermore, make sure to type your username at the *login* prompt and your password at the *password* prompt without errors. Backspacing to delete previous characters may not work (though Linux is more forgiving than Unix). If you make a mistake, repeatedly use the **(Enter)** key to get a new *login* prompt and try again. Also make sure to use the exact combination of upper- and lowercase letters. *Linux is case-sensitive!*

When you login to your account, it might look different. However, as a general rule, you get three messages after a command line login: (a) *motd*—This is the *message of*

the day. Here, the system administrator might inform you about system-down times or just present a message to cheer you up. (b) *last login*—As mentioned above, this is the date of your last login. (c) *You have new mail*—This message does exactly what it says. It tells you that at least one new email is waiting for you in your email program.

4.3 Using Commands

Once you have logged in to the computer you can start working. Let us start with some simple commands in order to get accustomed to the Linux environment. Commands are entered in the same way as you just entered your username: enter the command `date` and press the **Enter** key.

```

1 $ date
2 Tue Jan 23 17:23:43 CET 2003
3 $

```

Terminal 5: Date

As shown in Terminal 5, this command will show the computer system's date and will end with giving you an empty command line again. In fact, `date` is a program. It collects data from the system and uses the *standard output* to display the result. The standard output is the screen. You could also *redirect* the output into a file. This is illustrated in Terminal 6.

```

1 $ date > file-with-date
2 $

```

Terminal 6: Redirecting Date to File

The character “>” redirects the output of the command `date` into a file which is named *file-with-date*. If it does not exist, this file is created automatically. Otherwise, it will be overwritten. You could use “>>” to append the output to an existing file. Of course, since the output is redirected, you do not see the result. However, as shown in Terminal 7, you can use the command `cat` to display the content of a file.

```

1 $ cat file-with-date
2 Tue Jan 23 17:25:44 CET 2003
3 $

```

Terminal 7: Showing Date File Content

Most commands accept options. These options influence the behavior of the command (which is, as said, in fact a program). Options are often single letters prefixed with a dash (–, also called hyphen or minus) and set off by spaces or tabs. For a list of all strange characters and their names, see Sect. A.7 on p. 431.

```

1 $ cat -n file-with-date
2 1 Tue Jan 23 17:25:44 CET 2003
3 $

```

Terminal 8: Showing Date File Content

In the example in Terminal 8, the option `-n` to the command `cat` prints line numbers.

4.3.1 History and Autocompletion

The Bash shell supports nice little functions that help you by reducing the typing effort, and thus the chance of introducing “mutations”—I am talking about the command history and auto completion functionality.

Each time you hit the $\uparrow$ key, you will see past commands you have been executing. This is the history function. You can move backward ($\uparrow$) and forward ($\downarrow$) and execute the current command by hitting Enter Enter . You can edit the current visible command by navigating through the line with the arrow keys ($\leftarrow$ and $\rightarrow$) and deleting ($\leftarrow$) or entering characters before executing the command.

It is even possible to query the command history. From the command line prompt press $\text{Ctrl}+\text{R}$. You will get the message (reverse-i-search) '':. You can now enter a query term. The terminal will show you the most recent command line that matches the query term. To get even older entries that match the query string, just press $\text{Ctrl}+\text{R}$ again and again ... Once the desired command line is visible, you can edit ($\leftarrow$ and $\rightarrow$) and execute it Enter . Press $\text{Ctrl}+\text{C}$ if you want to quit the query without any command execution.

The autocompletion helps you to complete a command, folder name, filename, or variable name. This is achieved by pressing the tabulator key ($\rightarrow$) after a couple of characters of the word you are trying to complete has been typed in. If, when hitting tabulator key, the word is not completed, then there are perhaps multiple possibilities for the completion. Hit $\rightarrow$ again and it will list all possibilities. Imagine you typed `da` in the terminal. Now hit the tabulator key ($\rightarrow$)—nothing happens. Hit it again. Now, you will be presented with all possible commands that start with the letters `da`. Similarly, when you start to type a filename, the autocompletion helps to fill in the missing characters.

4.3.2 Syntax of Commands

In general, commands can be simple one-word entries such as the `date` command. They can also be more complex. Most commands accept *arguments*. An argument can be either an *option* or a *filename*. The general format for commands is: `command option(s) filename(s)`. Commands are entered in lowercase. Options often are single characters prefixed with a dash (see Terminal 8). Multiple options can be written individually (`-a -b`), or sometimes, combined (`-ab`)—you have to try it out. Some commands have options made from complete words or phrases. They start with two dashes, like `--confirm-delete`. Options are given before filenames. Commands, options, and filenames must be separated by white spaces. In a few cases, an option has another argument associated with it. The `sort` command is one example (see Sect. 7.1.2 on p. 79). This command does sort the lines of one or more text files according to the alphabet. You can tell `sort` to write the sorted text to a file, which name is given after the option `-o` (output).

```

1  $ date > file-with-date
2  $ cat file-with-date
3  Tue Jan 23 17:25:44 CET 2003
4  $ date >> file-with-date
5  $ cat file-with-date
6  Tue Jan 23 17:25:44 CET 2003
7  Tue Jan 23 17:25:59 CET 2003
8  $ sort file-with-date
9  Tue Jan 23 17:25:44 CET 2003
10 Tue Jan 23 17:25:59 CET 2003
11 $ sort -o sorted-file file-with-date
12 $ cat sorted-file
13 Tue Jan 23 17:25:44 CET 2003
14 Tue Jan 23 17:25:59 CET 2003
15 $

```

Terminal 9 summarizes what we have learned up to now. You should now have a feeling for options. In line 1, we save the current date and time in a file named *file-with-date* (when you do the above exercises, you will in fact overwrite the old file content). In line 2, we use the command `cat` to print out the content of *file-with-date* on the screen. With the next command in line 4, we add one more line to the file *file-with-date*. Note that we use `>>` in order to append and do not use `>` to overwrite the file. After applying `cat` again, we see the result in lines 6 and 7. We then sort the file with the command `sort`. Of course, that does not make sense, since the file is already sorted. The result is immediately printed onto the screen, except that we advise `sort` to do it differently. We do that in line 11. With the option `-o` (output), the result is saved in the file *sorted-file*. The file we want to sort is given as the last parameter. The result is printed in lines 13 and 14.

4.3.3 Editing the Command Line

What do we do if we have made a mistake in the command line? There are a number of commands to edit the command line. The most important ones are listed below:

<code>(BkSp)</code>	deletes last character
<code>(Del)</code>	deletes last character
<code>(Ctrl)+(H)</code>	deletes last character
<code>(Ctrl)+(U)</code>	deletes the whole input line
<code>(Ctrl)+(S)</code>	pauses the output from a program
<code>(Ctrl)+(Q)</code>	resumes the output after <code>(Ctrl)+(S)</code>

It is highly recommended not to use **Ctrl**+**S**. However, when you get experienced it might become useful for you.

4.3.4 Change Password

Together with your username, your password clearly identifies you. If anyone knows both your username and password, he or she can do everything you can do with your account. You do not want that! Therefore, you must keep your password secret. When you logged in for the first time into Linux, you probably got a password from the system administrator; but probably you want to change it to a password you can remember. To do so, you use the command `passwd` as shown in Terminal 10.

```
Terminal 10: Changing the Password with passwd
1  $ passwd
2  Changing password for user Freddy.
3  Changing password for Freddy
4  (current) UNIX password:
5  New password:
6  Retype new password:
7  passwd: all authentication tokens updated successfully.
8  $
```

First, you have to enter your current password. Then, in line 5 in Terminal 10 you are requested to enter your new password. If you choose a very simple password, the system might reject it and ask for a more complicated one. Be aware that passwords are case sensitive and should not contain any blanks (spaces)! You should also avoid using special characters. If you log in from a remote terminal the keyboard might be different. To avoid searching for the right characters, just omit them right from the beginning. After you have retyped the password without mistake, the system tells you that your new password is active. Do not forget it!

4.3.5 Help: I'm Stuck

When you start working in the command line you will also start to make mistakes. This is normal and part of the learning curve. But it might also happen that you get stuck—hmm. What can you do when your terminal is seemingly not responding to any keystrokes? This may, e.g. happen when you leave a command unfinished but hit **Enter**. In that case, try hitting **Ctrl**+**C** at once. This terminates the current execution and should bring you back to the terminal.

4.3.6 How to Find out More

Imagine you remember a command name but you cannot recall its function or syntax. What can you do? Well, of course, you always have the option of trial and error.

However, that might be dangerous and could corrupt your data. There are much more convenient ways.

First, many commands provide the option `-h` or `--help`. Guess what: *h* stands for help. Actually, `--help` works more often than `-h` and might be easier to remember, too. The command `ls -h`, for example, lists file sizes in a human readable way—you must use `ls --help` instead in order to obtain help. By applying the help option, many commands give a short description of how to use them.

There is, however, a much more informative way than using the help option. Almost all Unix systems, including Linux, have a documentation system derived from a manual originally called the *Unix Programmer's Manual*. This manual has numbered sections. Each section is a collection of manual pages, also called *manpages*, and each program has its own manpage. Most Linux installations have individual manual pages stored on the computer. Thus, they can be accessed at anytime. To do so, use the command `man` (manual). For example, if you want to find information about the command `sort`, then you type “`man sort`”. Usually, the output is directly sent to a page viewer (pager) like *less* (see Sect. 7.1.5 on p. 81). If not, you can pipe it to *less* by typing “`man sort | less`”. You can then scroll through the text using the (↑), (↓), (PgUp) and (PgDn) keys. You leave the manpage by pressing (Q). You should note that manual pages are not available for all entries. Especially, commands like `cd`, which are not separate Linux programs but part of the shell, are not individually documented. However, you can find them in the shell's manual pages with `man bash`.

The `info command` (information) command serves a similar purpose. However, the output format on the screen is different. To read through the output, press the space bar (Space); to quit the documentation press (Q). It is not possible to scroll through the output.

If you have a faint idea of a command, use `apropos keyword`. This command searches for the *keyword* in an index database of the manual pages. The keyword may also contain wildcards (see Sect. 8.12 on p. 108).

4.4 Logout

Almost as important as to know how to login to a system, you should know how to logout. You cannot simply switch off the computer. Usually, the computer will be a big server sitting in another building or even city; but even if you run Linux at home on your desktop computer, you should never, ever simply switch off the power supply. In case you work on a remote server, you just have to logout. Several commands allow you to logout from the system. The two most common ones are the command `logout` or the key combination (Ctrl)+(D). You are done. If you work on your home computer, you will see a new “login:” prompt. You should login again as superuser (root) and halt the system with the command `halt`. A lot of information will flush over the screen until it says something like: “Run Level 0 has been reached”. Then you can switch off the power, if it does not do so automatically.

Exercises

There is no other way to learn Linux than by working with it. Therefore, you must exercise and play around.

- 4.1.** Login to your Unix-based computer. What information do you get from your computer?
- 4.2.** Write a command and use the erase key to delete characters. The erase key differs from system to system and account to account. Try out `(BkSp)`, `(Del)` and `(Ctrl)+(H)`.
- 4.3.** Use the command `date` and redirect the output into a file named *the_date*. Append the current date again to that file and print the resulting file content onto the screen.
- 4.4.** Change your password. After you have changed it successfully, you might want to restore your old one. Do so!
- 4.5.** Logout from your current session.

Chapter 5

Working with Files

Before people started working with computers they used to work at desks (nowadays, people work with a computer on their lap on the sofa). They had information in files which were organized in different folders. Well, now we work on virtual desktops and still use files and folders. Figure 5.1 gives an example of how files are organized in directories.

Files are the smallest unit of information the normal user will encounter. A file might contain everything from nothing to programs to file archives. In order to keep things sorted, files are organized in folders (directories). You can name files and directories as you wish—but you must avoid the following special characters:

/ | \ - < > , # . ~ ! \$ & () [] { } ' " ? ^

You should also avoid to use the space character “ ”. In order to separate words in filenames the underscore character “_” is commonly used. Linux discriminates between upper- and lowercase characters. A file named *file* can be distinguished from a file called *File*. This is a very important difference in Microsoft Windows. In fact, in Linux everything is a file. Even devices like printers or the screen are treated as if they were files. For example, on many systems the file *tty1* stands for the serial port number 1. Table 5.1 gives you a short description of the most common system directories and their content.

5.1 Browsing Files and Directories

A special case of files are folders or directories. They give the operating system a hierarchical order. In Linux, the lowest level, the root directory, is called “/”. When you login to the computer you end up in your home directory which is “/home/username”. In order to find out in which directory you are type `pwd` (print working directory).

```
Terminal 11: Command pwd
1 $ pwd
2 /home/Freddy
3 $
```

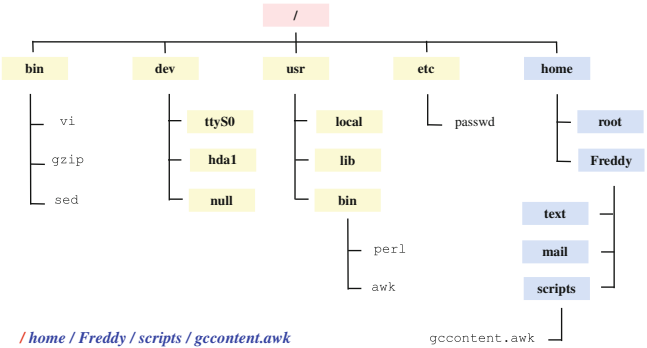


Fig. 5.1 Linux File System. This sketch illustrates the organization of directories (folders) and files in Linux. The root directory is shown at the *top* in *red*. System directories are colored *yellow* while user directories are highlighted in *blue*. As an example, the path to the file *gccontent.awk* is shown

Table 5.1 Names and description of some standard Linux directories

Directory	Content
/bin	Essential system programs
/boot	Kernel and boot files
/dev	Devices like printers and USB or serial ports
/etc	Configuration files
/home	User’s home directories
/lib	Important system files
/lost&found	Loose file fragments from system checks
/mnt	External drives like floppy disk
/proc	System information
/root	Root’s home directory
/sbin	Administrative programs
/tmp	Temporary files
/usr	Static files like programs
/var	Variable files like log files

As shown in Terminal 11, the command `pwd` (print working directory) prints out the *path* to the home directory of the user *Freddy*. The path is specific for each file and directory and contains all parent directories up to the root directory “/”. Thus, line 2 of Terminal 11 on the previous page reads: the directory *Freddy* resides in the directory *home*, which in turn resides in the directory *root* (*/*). The name of the home directory always equals the username, here *Freddy*. If you want to see what is in your home directory, type the command `ls` (list).

```

Terminal 12: Command ls
1 $ mkdir Awk_Programs
2 $ mkdir Text
3 $ mkdir .hidden
4 $ date > the.date
5 $ date > Text/the.date
6 $ ls Awk_Programs
7 Text the.date
8 $ ls -a
9 . .. Awk_Programs .hidden Text the.date
10 $ ls -l
11 total 12
12 drwxrwxr-x 2 rw rw 4096 Apr 1 16:50 Awk_Programs
13 drwxrwxr-x 2 rw rw 4096 Apr 1 16:51 Text
14 -rw-rw-r-- 1 rw rw 30 Apr 1 17:16 the.date
15 $ ls -lh
16 total 12K
17 drwxrwxr-x 2 rw rw 4.0K Apr 1 16:50 Awk_Programs
18 drwxrwxr-x 2 rw rw 4.0K Apr 1 16:51 Text
19 -rw-rw-r-- 1 rw rw 30 Apr 1 17:16 the.date
20 $ ls Text
21 the.date
22 $

```

In Terminal 12 we see the effect of different commands. In lines 1–3 we create the directories *Awk_Programs*, *Text* and *.hidden*, respectively. That is done with the command `mkdir` (make directory). Note that these directories are created in the current directory. In line 4 we create the file *the.date* with the current date (see Terminal 6 on p. 46) in the current directory. In line 5 we create a file with the same name and content; however, it will be created in the subdirectory *Text* (a subdirectory is the child of directory; the root directory is the mother of all directories). Next, we apply the command `ls` to check what is in the current directory. This command prints out all directories and files in the current directory. Note that the directory *.hidden* is missing in the list. Since it is preceded by a dot the folder is hidden.—Note: Hidden files start with a dot.—By default the `ls` command does not show hidden files and directories. Actually, many system files are hidden. It prevents the directories from looking chaotic. In line 8 we add the option `-a` (all) to the `ls` command. With this option we force the list command to show hidden and regular files and directories. Depending on your system settings files and directories, as well as hidden files and hidden directories, might appear in different colors. Thus you can directly distinguish files from directories (the corresponding option for `ls` is `-G`).

Now we want to get some more information about the files and directories. We use the option `-l` (list). With this option the command `ls` lists the files and folders and gives additional information. First, in line 11, the size of the directory is shown. Here, 12KB are occupied. In line 15 we add the option `-h` (human). We obtain the same output, but human-readable. Now we see that the number 12 means 12 K, which reads 12KB. In line 20, we list the content of the directory *Text*. It contains the file *the.date*, which we saved at that location in line 5. With the option `-R` (recursively) you can list the content of subdirectories immediately (see Terminal 15 on p. 63).

5.2 Moving, Copying and Renaming Files and Directories

Two very important commands are `cp` (copy) and `mv` (move). Their function is to copy and move files or directories from one location to another, respectively. The general syntax is `cp source destination` or `mv source destination`. Terminal 13 on the facing page exemplifies the use of `cp`. Interestingly, there is no command to rename a file or directory. Why is this? Well, renaming is essentially the same as moving. Thus, `mv seq.txt seq.fasta` renames the file *seq.txt*.

5.3 File Attributes

When you apply the command `ls -l`, you can see that any file or directory is connected to certain attributes. Let us take a closer look at line 13 of Terminal 12 in Fig. 5.2.

The following list describes the individual file attributes in more detail. The numbers correspond to the italic numbers in Fig. 5.2.

1. Type

The first character indicates whether the item is a directory (d), a normal file (-), a block-oriented device (b), a character-oriented device (c) or a link (l). Actually, a directory is a special type of file. The same holds for the special directories “.” and “..” which can be seen in line 9 in Terminal 12 on the preceding page.

2. Access Modes

The access modes or permissions specify three types of users who are either allowed to read (r), write (w), or execute (x) the file. The first block of 3 characters indicates your own rights, the next block of 3 characters the group’s rights, and the last block of 3 characters the right for any user. The dash character (-) means that the respective permission is not set.

In line 17 in Terminal 12 on p. 55 we see a directory (d). Everybody is allowed to read (r) and access (x) the directory. However, only the user *rw* and the group *rw* is allowed to create files (w) in that directory. In line 19 we see an ordinary file (-) which is not executable (x). In Sect. 5.5 on p. 59 we will learn how the permissions can be changed.

drwxrwxr-x	2	rw	rw	4.096	Apr 1 16:51	Text
<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>
permissions	nodes	user	group	size	last change	name
file type						

Fig. 5.2 File Attributes. Data obtained by the command `ls -l`. The *numbers in italics* correspond to the description in the text

3. **Number of Links**

This item indicates the number of links or references to this file or folder. By default, there are two links to directory and one link to files. Any subdirectory increases the number of directory links by one.

4. **Owner**

The user who created the file or directory. Later, the ownership can be changed with the command `chown`. Then the current owner is shown.

5. **Group**

This item shows to which group the file or directory belongs. The group ownership can be changed with the command `chgrp`. In some Linux versions this item is not shown by default. In that case you have to use the option `-g` with the `ls` command.

6. **Size in Byte**

The size of the file or directory. The exact number of bytes is shown. If the option `-h` (human) is used then the file size is rounded and the character K (Kilobyte) or M (Megabyte) is used. The size of a directory is that of the directory file itself, not of all the files in that directory. Furthermore, the size of the directory file is always a multiple of 1024, since directory entries are arranged in blocks of 1024 B (= 1 KB).

7. **Modification Date and Time**

Here, the date and system time when the file was last modified is shown. For a directory, the date and time when the content last changed (when a file in the directory was added, removed, or renamed) is shown. If a file or directory was modified more than 6 months ago, then the year instead of the time is shown.

8. **Name**

Finally, the name of the file or directory is shown.

We will come back to certain aspects of file attributes later.

5.4 Special Files: `.` and `..`

Each directory contains two special files named *dot* and *dotdot*. These files point to the directory and to the parent directory, respectively. Figure 5.3 illustrates this.

In Terminal 13 we learn more about the meaning of the directories “.” (dot) and “..” (dot dot).

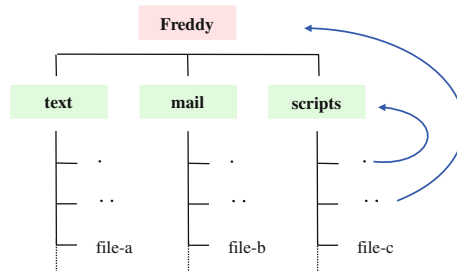


Fig. 5.3 Meaningful Dots. The two special files *dot* and *dotdot* can be found in every directory. They are frequently used together with commands like `cd` (change directory) and `cp` (copy) and refer to the current (*green*) or parent (*red*) directory, respectively

```

1  $ pwd
2  /home/Freddy
3  $ mkdir temp
4  $ date > the.date
5  $ cd temp
6  $ pwd
7  /home/Freddy/temp
8  $ ls
9  $ cp ../the.date .
10 $ ls
11 the.date
12 $ cd ..
13 $ pwd
14 /home/Freddy
15 $ rm -r temp

```

Let us first check where we are. We use the command `pwd` (print working directory). Then we create the directory *temp* and the file *the.date* in the current directory */home/Freddy*. Next, in line 5, we change into the directory *temp* with the command `cd` (change directory). By using `pwd` we can prove that we changed into the directory *temp* which we created in line 3. With the `ls` command in line 8 we show that the directory is empty. Now we copy the file *the.date* which we created in directory */home/Freddy* into the currently active directory */home/Freddy/temp* (where we are right now). Therefore, we apply the command `cp` (copy). The parent directory to the currently active directory has the shortcut “`..`”, whereas the current active directory has the shortcut “`.`”. The syntax of the copy command is “`cp source destination`”. Thus, with `../the.date` we call the file *the.date* which is one directory up. With the single dot we specify the directory we are in. This is the destination. Thus the whole command in line 9 copies the file *the.date* from */home/Freddy* to */home/Freddy/temp*. Note that you have duplicated the file. In order to move the file you have to substitute the command `cp` for the command `mv` (move). In line 12 we jump back to our home directory. Again we make use of the “`..`” shortcut. Then, in line 15, we remove the directory *temp* and all its content. Therefore, we apply the command `rm` (remove) with the option `-r` (recursively). We can use only `rm` if we

want to erase one or more files. However, to erase directories, even empty ones, you will need the option `-r`. Why is that? Well, we just saw that even an empty directory contains two, though hidden, files: `“.”` and `“..”`.

5.5 Protecting Files and Directories

We learned already in Terminal 12 on p. 55 that one characteristic of Linux files is ownership. Not everybody can do everything with files. This is an immense advantage since it makes the system safe. Note: You must be aware that the system administrator can access all files and directories. Thus, you should never keep secret information on a foreign computer.

5.5.1 Directories

Access permissions to directories help to control access to the files and subdirectories in that directory. If a directory has *read permission* (`r`), a user can run the `ls` command in order to see the directory content. He can also use wildcards (see Sect. 8.12 on p. 108) to match files in this directory. If a directory has *write permission* (`w`), users can add, rename, and delete files in that directory and all subdirectories. To access a directory, that is to read, write or execute files or programs, a user needs *execute permission* (`x`) on that directory. Note: To access a directory, a user must also have execute permission to all of its parent directories, all the way up to the root directory `“/”`.

5.5.2 Files

Access permissions on a file always refer to the file's content. As we saw in the previous paragraph, the access permissions of the directory where the file is located control whether the file can be renamed or removed. *Read permissions* (`r`) and *write permissions* (`w`) control whether you can read or change the file's content, respectively. If the file is actually a program it must have *execute permission* (`x`) in order to allow a user to run the program.

5.5.3 Examples

Let us take a look at some examples in order to clarify the situation. Look back at Terminal 12 on p. 55 line 14. The first package of information, consisting of 10

characters, gives us some information about the file. First, indicated by the leading dash `_` `rw-r--r--`, we learn that the file is indeed a file and not a directory or something else. Now, let us look at the first block of permissions: `rw-r--r--`. This block indicates the file owner's rights. The next block `-rw-r--r--` indicates the group's right and the last block `-rw-r--r--` indicates the access rights of all users. The following list exemplifies a number of different file attribute combinations.

<code>- rwx rwx rwx</code>	A file which can be read and modified by everybody. This setting gives no security at all.
<code>- rw- r-- r--</code>	This represents the standard setting for a file. The file can be read and modified by the owner, but can only be read by all the others.
<code>- r-- --- ---</code>	This is the most secure setting. The owner can read but cannot modify the file. All the others can neither read nor modify the file.
<code>- rwx r-x r-x</code>	The standard setting for executable files. Everybody is allowed to run the program but only the user can change the file.
<code>d rwx r-x r-x</code>	The standard setting for directories. The user is allowed to access the directory and write and delete entries. All the other users can access the directory and list its content. However, they are not allowed to write or delete entries.
<code>d rwx--x --x</code>	Here the user has the same rights as in the example above. Other users are allowed to access the directory (with the command <code>cd</code>); however, they cannot list the content with the command <code>ls</code> . If the name of entries is known to the other users, they can, depending on the access rights of these entries, be read, written, and executed.

5.5.4 Changing File Attributes

Now, in Terminal 14, we will play with file associations. Less common are the commands “`chown user file`” and “`chgrp group file`”. They are used to change the user and the group ownership of a file or directory, respectively. The former commands can be executed only by the superuser root. We will concentrate on the command “`chmod mode file`”.

Table 5.2 Parameter that can be used with the command `chmod`

User	Type	Rights
u—user	+ add	r—read
g—group	– delete	w—write
o—others		x—execute
a—all		

```

1  $ id
2  uid=502(Freddy) gid=502(Freddy) Groups=502(Freddy)
3  $ date > thedate
4  $ ls -l
5  -rw-rw-r-- 1 Freddy Freddy 30 Apr 15 21:04 thedate
6  $ chmod g-w,o-r,a+x thedate
7  $ ls -l
8  -rwxr-x--x 1 Freddy Freddy 30 Apr 15 21:04 thedate
9  $ chmod g=w thedate
10 $ ls -l
11 -rwx-w---x 1 Freddy Freddy 30 Apr 15 21:04 thedate
12 $ chmod 664 thedate
13 $ ls -l
14 -rw-rw-r-- 1 Freddy Freddy 30 Apr 15 21:04 thedate
15 $
```

In line 1 of Terminal 14 we check our identity with the command `id`. We learn that Freddy has the user identity (uid) and group identity (gid) 502. The user Freddy belongs only to the group Freddy. In line 3 we generate the file *thedata*, which contains the current date. We do this only in order to have a file to play around with. Now let us check the file attributes with the command `ls -l` (list as list). Line 5 gives the result. For the meaning of the attributes look back to Sect. 5.3 on p. 56. In line 6 we actively change the file attributes with `chmod`. In fact, we deny the group (g) access to modify (w) and others (o) to read (r) the file. All users (a) get the permission to execute (x) the file. Line 8 shows the changes. To change the access modes we state the user category, the type of change, and the permissions to be changed. The letter code used for this command is shown in Table 5.2. Entries can be combined to something like *ug+rw*. Entries are separated by commas.

Now, we apply another method to change file attributes. This is shown in line 9 in Terminal 14. Here the required access mode setting is directly entered. Again, settings can be combined and several entries are comma-separated, as in *ug=rw, a=x*. If you exclude the type, this is interpreted as nothing. With “`chmod go= filename`” the permissions are set to exactly nothing for group and others.

Finally, the last method we use is based on a number code. The generation of the code is shown in Table 5.3. The numbers are allowed to lie between 0 and 7. For each user group (user, group, others) the code is calculated by adding the values for read (r = 4), write (w = 2), and execute (x = 1) permission. If all permissions are required, the values add to 7, if only read and write permissions are to be set, the numbers add to 6 and so forth.

In line 12 of Terminal 14 we apply the code. With the command `chmod 664 thedate` we change the access permissions back to the default values: the user

Table 5.3 How the number code is generated

User	Type	Rights
r w x	r w x	r w x
4+2+1	4+2+1	4+2+1
7	7	7

and users of the same group are allowed to read (r) and modify (w) the file *thedata*, whereas all the other users are only allowed to read the file.

If you want to change the permissions for many files at once you just list them one after the other: `chmod 777 file1 file2 file3`. If you use the wildcard “*” instead of filenames, you change permissions for all files in the directory at once. If you use the number 0, you reset all permissions. For example, “`chmod 660 *`” sets the permissions of all files in the current directory to reading and writing for you and your group only.

5.5.5 Extended File Attributes

There is a new file attribute development on the way to getting established. It is developed by *Remy Card* and needs the *Linux second extended file system*. The corresponding program `chattr` (change attributes) largely expands the possibilities to restrict file and directory access. The letters “ASacdijsu” select the new attributes for the files: do not update time stamp (A), synchronous updates (S), append only (a), compressed (c), no dump (d), immutable (i), data journaling (j), secure deletion (s), and undeletable (u). For example, by making files append-only, their content can only be extended but cannot be deleted. Using the compressed (c) flag automatically compresses the file to save disk space. You have to check if you can use these extended file attributes on your system.

5.6 File Archives

File archives are frequently used when files are distributed via the Internet or backed up onto a CD-ROM. In this section you will learn some basics on how to work with archived files.

From Windows you may be used to archive files with *WinZip*. This program creates a compressed file or even compressed directories. This is not possible with Linux—at least not at the command line level. With Linux you first create an archive of files or directories and then compress the archive file.

The most important command to create or extract an archive is `tar` (tape archive). The name already implies what the main purpose of the command is: to create a file archive for storage on a tape. Even nowadays, large datasets are stored on magnetic

tapes. The reason is the high density of information that can be stored on magnetic tapes (maybe you also remember the *Datasette* from the *Commodore 64* times). Well, let us come back to the `tar` command. With `tar` you can put many directories and files into one single file. Thus, it is very easy to recover the original file organization. For that reason programs and other data are often distributed in the archived form. In your local working directory you then extract the archive and get back the original file and directory structure. As with many programs, in addition to the original `tar` program a GNU version called `gtar` is available on many systems. With `gtar` it is possible to compress files before they are put into the archive. By this, around 50 % of memory can be saved (and, if you think further, money!). On most systems `gtar` completely substitutes for `tar`, although the command name `tar` has been kept. Thus, you do not have to bother about `gtar`. The output of `tar` can be sent either to a file or to a device like the floppy disk, a CD-ROM burner or a streamer. Now let us use `tar`. First go to your home directory (`cd`) and create a directory called *tartest* (`mkdir`). Change into the newly created directory (`cd`) and create the file *tartest_file* with `date > tartest_file`. Then create the directories *tata1* and *tata2*. Within each of these two directories create 3 files named *tatax_file* with the command `man tar > tata1_file1` (x is the number of the directory and y the file number). By redirecting (`>`) the output of the `man` command we create files with some content. You should end up with a list of files shown in Terminal 15.

```
Terminal 15: Listing Files Recursively
1  $ ls -R
2  .:
3  tartest_file  tata1  tata2
4
5  ./tata1:
6  tata1_file1  tata1_file2  tata1_file3
7
8  ./tata2:
9  tata2_file1  tata2_file2  tata2_file3
10 $
```

In Terminal 15 we use the command `ls` with the option `-R` (recursively; note that the R is uppercase). With this command we list the content of the current working directory and the content of all its subdirectories. The output says that in the current working directory (“.” in line 2) we find the file *tartest_file* and the directories *tata1* and *tata2* (line 3). In the subdirectory *./tata1* (line 5) we find the files *tata1_file1* to *tata1_file3* (line 6) and so on.

Before we start using the `tar` command we should look at some of its options:

- c **C**reates a new archive.
- x **E**xtracts files from an existing archive.
- t **P**rints a **t**able of contents of an existing archive. Do not forget to use the option **f** together with **t**! If you even add the option **v**, you will get a detailed content list with file sizes and so on. However, listing the content works only for uncompressed archives.
- f **E**xpects a **f**ile *name* for the archive; the archive file will be named *name*. It is very helpful to end the file with *.tar*
- v **V**erbose, this means the program will print out what it is doing.
- z This option is available only with GNU-tar. It will compress the files after they have been added to the archive. Note that you have to decompress the archive with the same option in order to extract the files. It is very helpful to name compressed archives with the ending *.tgz*.

Of course, there are many more options. Take a look at the manual pages (see Sect. 4.3.6 on p. 49). As a big exception, the `tar` command takes the options without any dashes (-). Note that you should give your archives an appropriate filename extension: *.tar* for uncompressed and *.tgz* for compressed archives. Otherwise, you will not recognize the file as an archive! Now let us take a look at Terminal 16.

```

_____ Terminal 16: Creating File Archives with tar _____
1  $ pwd
2  /home/Freddy/tartest
3  $ tar cvf ../daten.tar .
4  ./
5  ./tata1/
6  ./tata1/tata1_file1
7  ./tata1/tata1_file2
8  ./tata1/tata1_file3
9  ./tata2/
10 ./tata2/tata2_file1
11 ./tata2/tata2_file2
12 ./tata2/tata2_file3
13 ./tartest_file
14 $ tar cfz ../daten.tgz .
15 $ ls -lh ../daten*
16 -rw-rw-r-- 1 Freddy Freddy 100K Apr 19 15:55 ../daten.tar
17 -rw-rw-r-- 1 Freddy Freddy 5.7K Apr 19 15:55 ../daten.tgz
18 $

```

In line 1 we check that we are in the directory *tartest*. We then create (c) an archive with the filename (f) *daten.tar*, which will be written into the parent directory (.). The archive should contain everything which is in the current working directory (.). We choose to follow the progress of the program (v). In line 14 we create in the parent directory the archive *daten.tgz*, which is compressed (z). Furthermore, we do not wish to follow the program's progress (no verbose). In line 15 we use the command `ls` to list all files in the parent directory that begin with *daten*. Furthermore, we instruct `ls` to give file details (-l) and print the file size human-readable (-h). As

you can see, the compressed archive is much, much smaller than the uncompressed version.

Now take a look at the archives *daten.tar* and *daten.tgz* by typing `cat ../daten.tar | less` and `cat ../daten.tgz | zless`, respectively. You can scroll through the file content using `↑`, `↓`, `PgUp` and `PgDn`. In order to get back to the command line press `Q`.

How do we unpack an archive? Well, this is easy. Go to your home directory, where the files *daten.tgz* and *daten.tar* are, and make a directory called *tarout*. Then you have two possibilities to unpack the archive into *tarout*: either you change to the directory *tarout* and use “`tar xfz ../daten.tgz`” or you stay in your working directory and use “`tar xfz daten.tgz -C tarout/`”. The `tar` command always extracts the archive into the present working directory, unless you explicitly name an existing output directory after the option `-C` (note that now the dash is required!). If you wish to extract the uncompressed archive *data.tar* you just omit the option `z`. It is very helpful to check the content of an archive with either “`tar ft archive_name`” for a short listing, or “`tar ftv archive_name`” for a detailed listing. This works only for uncompressed archives, for compressed archives you must add the option `z`!

5.7 File Compression

In order to save time, files distributed via the Internet are usually compressed. There are several tools for file compression available. Most data-compression algorithms were originally developed as part of proprietary programs designed to compress files being interchanged across a single platform (e.g. PkZIP for MS-DOS and TAR for Unix). Data-compression modules have since developed into general purpose tools that form a key part of a set of operating systems. Two compression mechanisms, ZIP and TAR, have become ubiquitous, with a wide range of programs knowing how to make use of files compressed using these techniques. In Sect. 5.6 on p. 62 we learned how to archive files using the command `tar`. Tar files are often compressed using the `compress` or `gzip` commands, with the resulting files having *.tar.z* (or *.tar.z* and *.tz*) or *tar.gz* (or *.tgz*) extensions, respectively (Table 5.4).

zip/unzip—The most commonly used data-compression format is the ZIP format. ZIP files use a number of techniques to reduce the size of a file, including file shrinking, file reduction and file implosion. The ZIP format was developed by Phil Katz for his DOS-based program PkZIP in 1986 and is now widely used on Windows-based programs such as WinZip. The file extension given to ZIP files is *.zip*. The content of a zipped file can be listed with the command “`unzip -l filename.zip`”.

gzip/gunzip—The GNU compression tool *gzip* is found on most Linux systems. Files are compressed by “`gzip filename`” and automatically get the extension *.gz*. The original file is replaced by the compressed file. After uncompressing using

Table 5.4 File-compression commands and file extensions

Command	Extension(s)
zip	.zip
compress	.tar.z – .taz – .tz
gzip	.tar.gz – .tgz
bzip2	.bz2

“gzip -d *filename*” or “gunzip *filename*” the compressed file is replaced by the uncompressed file. In order to avoid replacement you can use the option -c. This redirects the output to standard output and can thus be redirected into the file. The command “gzip -c *filename* > *filename2*” creates the compressed file *filename2* and leaves the file *filename* untouched. With the option -r you can recursively compress the content of subdirectories.

gzip is used when you compress an archive with tar z (see Sect. 5.6 on p. 62). Then you get the file extension .tgz.

bzip2/bunzip2—The command bzip2 is quite new and works more efficiently than gzip. Files are usually 20–30 % smaller compared to gzip compressed files.

compress/uncompress—The compression program compress *filename* is quite inefficient and not widespread any more. Compressed files carry the extension .Z. Files can be uncompressed using compress -d *filename* (decompress) or uncompress *filename*.

Table 5.4 gives an overview of file extensions and their association with compression commands.

5.8 Searching for Files

After a while of working, your home directory will fill up with files, directories, and subdirectories. The moment will come when you cannot remember where you saved a certain file. The find program will help you out. It offers many ways in which to search for your lost file. find searches in directories for files or directories (remember that directories are a sort of file) that fulfill certain attributes. Furthermore, you can define actions to be executed when the file(s) are found. More than one attribute can be combined with a logical and or a logical or. Per default, find combines all search attributes with the logical and. To use or, the attributes must be separated by the -o and be enclosed in brackets. Note: The brackets must be preceded by an escape character. The syntax of the search command is find *directory attributes*. Some important options are:

Again, there are many more options available. You are always welcome to learn more about the commands by looking into the manual pages, in this case by applying man find (see Sect. 4.3.6 on p. 49). Let us perform a little exercise in order to get a better feeling for the find command. Go to your home directory (cd) and create

<code>-type</code>	Search for files (f) or directories (d)
<code>-name</code>	The name should be put into double quotes (“ ”) and may contain wildcards (*,?,[],...)
<code>-size</code>	Search for files larger (+) than a number of 512-byte blocks or characters (add a c). You can also search for files that must be smaller (–) than the given size.
<code>-ctime</code>	Search for files that were last changed x days ago. You can also search for files that are older (+) or younger (–) than x days.
<code>-exec</code>	Execute a command on all found files. The syntax is: “command_{ } \;”. Notice the space characters! The curled brackets represent the found files.

a directory called *find-test* with the subdirectory *sub*. Now create two files within *find-test* (date>find-test/1 and date>find-test/2) and one file within *find-test/sub* (date>find-test/sub /3.txt).

```

Terminal 17: Finding Files with find
1  $ mkdir find-test
2  $ mkdir find-test/sub
3  $ date>find-test/1
4  $ date>find-test/2
5  $ date>find-test/sub/3.txt
6  $ find find-test -type f -size -6 -name "[0-9]"
7  find-test/1
8  find-test/2
9  $ find find-test -type f -size -6 -name "[0-9]*"
10 find-test/1
11 find-test/2
12 find-test/sub/3.txt
13 $ find find-test -type f -size -6c -name "[0-9]*"
14 $ find find-test -type f -size -6 -name "[0-9]*"
15     -name "*.txt"
16 find-test/sub/3.txt
17 $ mkdir find-found
18 $ find find-test -type f -size -6 -name "[0-9]*"
19     -name "*.txt" -exec cp {} find-found \;
20 $ ls find-found/
21 3.txt
22 $

```

In Terminal 17 for lines 1–5 we first create the directories and files as described above. In line 6 we search in the directory *find-test* for files (`-type f`) with a size less than 6 times 512 byte (`-size -6`) and a filename that contains any number from 0 to 9 (`-name [0-9]`). The brackets `[ ]` are wildcards (see Sect. 8.12 on p. 108). The output is displayed in lines 7–8. Two files are found. In line 9 we allow the filename to be a number followed by any other character or none (`-name [0-9]*`). Again, the brackets `[ ]` and the star `*` are wildcards. Now, all three files are found. In line 13, we use the same attributes as in line 9, except that the file must contain less than 6 characters (`-size -6c`). There is no hit. Next, we search with the same options as above but the filename must contain any number from 0 to 9 followed by any other character or none (`*`) and end with *.txt* (`-name *.txt`). There is only one hit

(*find-test/sub/3.txt*). In line 18/19 we use the same search pattern as in line 14/15, but apply the `cp` command to copy this file into the directory *find-found* (`-exec cp {} find-found \;`), which we created in line 17. A check with the `ls` command in line 20 shows us that we were successful.

5.8.1 Selecting Files for Backups

There are several ways to create backups (see Sect. 6.4 on p. 73). The command `find` provides a nice way to select specific files for a backup.

```
find *.seq -exec cp {} Backup/{}.backup \;
```

This command would copy all files ending with *.seq* from the current directory to subdirectory *Backup* and append the suffix *.backup*. This simple example shall only illustrate how you could set up a powerful engine to select files to be archived.

5.9 Display Disk Usage

Eventually, after playing a lot with files and after some analysis of next-generation sequencing data, your hard disk drive is full. The command `df` (display free space) shows you available and used space on mounted devices as shown in Terminal 18.

```

1  $ df -h
2  Filesystem      Size  Used Avail Use% Mounted on
3  /dev/sdal        7.0G  3.1G  3.6G  46% /
4  udev             494M  4.0K  494M   1% /dev
5  tmpfs            201M  780K  200M   1% /run
6  none             5.0M   0    5.0M   0% /run/lock
7  none             501M  200K  501M   1% /run/shm
8  COMPIOL          932G  453G  479G  49% /media/sf_COMPIOL
9  $ du -hs Thio
10 552K      Thio
11 $ du -h -d0 *
12 68K      Backup
13 12K      genomes.txt
14 3.8M     Horizontal
15 20K      MySQL
16 144K     R
17 39M     Sequencing
18 12K     table.tab
19 552K     Thio
20 $

```

The option `-h` makes the output human readable. The command `du` (display disk usage) shows the space occupied by individual files or folders. In line 9 in Terminal 18 we analyze the space used by folder *Thio*. Then, in line 11, check the disk usage of all (*) files and folders in the current directory. For folders the depth is set to zero

(`-d0`). This means that we do not like to see information for files and folders within the folders in the current directory.

Exercises

The following exercises should strengthen your file-handling power.

5.1. Go to your home directory and check that you are in the right directory. List all files in your home directory. List all the files in that directory including the hidden files. List all the files including the hidden files in the long format.

5.2. Determine where you presently are in the directory tree. Move up one level. Move to the root directory. Go back to your home directory.

5.3. Create a directory called *testdir* in your home directory, and a directory named *subdir* in the *testdir* directory (*testdir/subdir*). List all the files in the *testdir/subdir* subdirectory, including the hidden files. What are the entries you see? Remove the *subdir* subdirectory in your home directory.

5.4. Using the `cd` and the `ls` commands, go through all directories on your computer and draw a directory tree. Note: You may not have permission to read all directories. Which are these?

5.5. What permissions do you have on the */etc* directory and the */etc/passwd* file? List all the files and directories in your */home* directory. Who has which rights on these files, and what does that mean? Print the content of the file */etc/passwd* onto the screen.

5.6. In your home directory create a directory named *testdir2*. In that directory create a file containing the current date. Now copy the whole directory with all its content into the folder *testdir*. Remove the old *testdir2*. Check that you have successfully completed your mission.

5.7. Create the file *myfile* with the current date in the directory *testdir*. Change permissions of this file: give yourself read and write permission, people in your group should have read, but no write permission, and people not in your group should not have any permissions on the file.

5.8. What permissions do the codes 750, 600, and 640 stand for? What does `chmod u=rw,go-rwx myfile` do? Change permissions of all files in the *testdir* directory and all its subdirectories. You should have read and write permission, people in your group have only reading permission, and other people should have no permissions at all.

5.9. Create a new directory and move into it with a one-liner.

5.10. Use different compression tools to compress the manual page of the `sort` command. Calculate the compression efficiencies in percents.

5.11. Create the directory *testpress* in your home directory. Within *testpress* create 3 non-empty files (redirect manual pages into these files). From your home directory use the commands `(un)compress`, `(un)zip`, `g(un)zip`, and `b(un)zip2` to compress and uncompress the whole directory content (use the option `-r` (recursively)), respectively (only for `gzip`). What does the compressed directory look like?

5.12. Create a hidden file. What do you have to do in order to see it when you list the directories content?

5.13. Create a number of directories with subdirectories and files and create file archives. Extract the file archives again. Check whether files are overwritten or not.

5.14. Play around with copying, moving, and renaming files. What is the difference between moving and renaming files? What happens to the file's time stamp when you copy, move, or rename a file?

5.15. Create a directory in a directory, a so-called subdirectory. What happens to the number of links?

5.16. Create a file in a directory. Change this file. What happens to the modification date of the directory? Add a new file. What happens now? Rename a file with the command `mv` (move). What happens now?

5.17. Play around with the `find` command. Create directories, subdirectories, and files and try out different attributes to find files.

Chapter 6

Remote Connections

It is not uncommon in computational biology that you have to work on remote computers (servers). They often provide much more computing power than your local computer (client) or serve as web servers or even data storages. This rises several questions such as: how do I execute commands at a remote server and how can I transfer files between my local client and the remote server. There are several solutions with different levels of comfort. In Sect. 4.2.3 on p. 44 I quickly introduced PuTTY as a way to connect to a remote computer from Microsoft Windows. In this chapter I like to go one step further and show you the Linux way. This involves not only execution of commands at the remote server but also transferring files and even setting up a remote desktop at your local client.

6.1 Downloading Data from the Web

Let us get started with the easiest remote connection: data download from the internet. There are two standard commands to do so, `wget` and `curl`. Here I demonstrate their basic application although both commands can be tuned with dozens of attributes.

6.1.1 `wget`

The command `wget` (formerly `geturl`) is installed on most Linux systems. Its function is extremely easy. You only have to give the URL (uniform resource locator) to the web resource you wish to download. Terminal 19 illustrates this.

```
Terminal 19: Usage of wget
1 $ wget "ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt"
2 --2012-09-28 14:38:12--  ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt
3      => 'gbrel.txt'
4 Resolving ftp.ncbi.nih.gov (ftp.ncbi.nih.gov)... 130.14.250.11
5 Connecting to ftp.ncbi.nih.gov (ftp.ncbi.nih.gov)|130.14.250.11|:21... connected.
```

```

6 | Logging in as anonymous ... Logged in!
7 | ==> SYST ... done.      ==> PWD ... done.
8 | ==> TYPE I ... done.    ==> CWD (1) /genbank ... done.
9 | ==> SIZE gbrel.txt ... 329828
10 | ==> PASV ... done.     ==> RETR gbrel.txt ... done.
11 | Length: 329828 (322K) (unauthoritative)
12 |
13 | 100%[=====] 329,828      261K/s   in 1.2s
14 |
15 | 2012-09-28 14:38:16 (261 KB/s) - 'gbrel.txt' saved [329828]
16 |
17 | $

```

What is actually happening is that we download the file *gbrel.txt* from the NCBI server. This file contains information about the most recent release of the GenBank database. In the same way you could download any other file or webpage.

6.1.2 *curl*

curl works similar to *wget* but is (a) more common on MacOSX systems and (b) prints the downloaded data to *stdout*, i.e. the screen. Thus, you have either to redirect the output to a file or use the attribute *-o* and state a filename.

6.2 Secure Copy: *scp*

In order to upload or download files from a remote server securely, the command *scp* (secure copy) was introduced. It uses a secure network protocol. Terminal 20 shows its usage.

```

Terminal 20: Download File with scp
1 | $ scp "awkologist@servername.com:CompBiol/met-sulfur.pdb" .
2 | The authenticity of host 'servername.com' can't be established.
3 | RSA key fingerprint is 95:3a:4f:54:27:4f:11:30:db:dc:e6:0e:a8:55:92:aa.
4 | Are you sure you want to continue connecting (yes/no)? yes
5 | Warning: Permanently added 'servername.com' (RSA) to the list of known hosts.
6 | awkologist@servername.com's password:
7 | met-sulfur.pdb                                100% 486      0.5KB/s   00:00
8 | $ cat met-sulfur.pdb
9 | ATOM      309  SD  MET  A   42      16.961   80.516   63.280   1.00  10.80      S
10 | ATOM      371  SD  MET  A   49      22.387   84.086   68.399   1.00  11.46      S
11 | ATOM      574  SD  MET  A   78      35.312   69.282   54.566   1.00  21.89      S
12 | ATOM      671  SD  MET  A   92      33.258   78.879   56.959   1.00  10.81      S
13 | ATOM     1085  SD  MET  A  153       5.038   76.385   48.852   1.00  32.02      S
14 | ATOM     1190  SD  MET  A  167      14.200   62.640   55.932   1.00  30.62      S
15 | seal-macbook:PROJECTS rw$

```

The general command is:

scp source target

which is similar to the `cp` command. Since the source is remote it requires further specification: `username@servername:path`. The path has to be specified from the home directory. In Terminal 20 I assume that user *awkologist* has an account at server *servername.com* and that there is a file named *met-sulfur.pdb* in directory *CompBiol*, which is a subdirectory from the home directory. Thus the complete path would be */home/minoc/pgx-17/awkologist/CompBiol/met-sulfur.pdb*. The target is the current directory (`.`). If you connect for the first time, you are asked if you really like to continue (lines 2–5).

If you would like to upload a file, just change the order of source and target. You can download whole directories with the option `-r` (recursively). Likewise, you may use wildcards.

6.3 Secure Shell: `ssh`

Again, I assume that user *awkologist* has an account at server *servername.com*. Now let us see how we can establish a remote connection and actually work on this server with a secure shell (`ssh`).

```
Terminal 21: Establishing Remote Connection with ssh
1  $ ssh -Y awkologist@servername.com
2  awkologist@servername.com's password:
3  Welcome to the Computational Biology Network
4
5  The system will be down for maintenance at Sunday.
6  Your resources are okay.
7
8  Have a lot of fun ...
9
10 awkologist@cbn $ xclock &
11 [1] 25829
12 awkologist@cbn $
```

It is so easy and so powerful. In line 1 in Terminal 21 we run `ssh` with the option `-Y`, which enables trusted X11 forwarding. As come to this in a second. After the correct password was entered, some system message may appear or not. And then you are logged in. You probably recognized the changed command line prompt in line 10. Since we enabled X11-forwarding, we can execute graphical commands, i.e. programs that produce graphical output in the form of new windows. This, however, requires that you logged in from a X-windows system, too (e.g. Ubuntu). In line 10 the program `xclock` is started, which opens a window containing an analog clock. With `&` we push that program (or process) to the background.

6.4 Backups with `rsync`

Backups are extremely important. I experienced already several hard disk crashes and it was only at the first occasion that I had no backup. Of course there are several ways and levels how backups can be generated. The simplest way is to copy all files

in your home directory to either an external disk drive via, e.g. USB, or a remote disk drive via, e.g. the network. You might also select specific files for a backup as shown in Sect. 5.8.1 on p. 67. Here I like to show you how to perform a backup of files via the network to a remote computer with the command `rsync` (remote synchronization). The `rsync` remote-update protocol allows `rsync` to transfer just the differences between two sets of files across the network connection, using an efficient checksum-search algorithm. Running `rsync` as a cron job (see Sect. 8.11 on p. 107) is a nice way to backup your data regularly in the background. Let us take a look at Terminal 22, which shows you how to use `rsync` in practice.

```
Terminal 22: Backups with rsync
1 $ rsync -avz Data/ awkologist@servername.com:Data/
2 Password:
3 building file list ... done
4 Nutrilyzer/.DS_Store
5 Nutrilyzer/Organisms/
6 Nutrilyzer/Organisms/.DS_Store
7 ...
8 $
```

What it does is easy. All files in the local (client) folder *Data* are copied to folder *Data* at the server *servername.com*. The attributes `-avz` instruct `rsync` to transfer all files in “archive” mode, which ensures that symbolic links, file attributes, file permissions, ownerships, etc. are preserved. Additionally, compression will be used to reduce the size of data portions of the transfer. Read the next section to see how to avoid the password request each time you trigger `rsync`. This comes very handy when running the backup as a cron job.

6.5 ssh, scp & rsync without Password

It can be very annoying to always to enter `ssh -Y user@server` and the password if you need to connect frequently to the same server. It gets even more annoying, when you include this in a script file. In that case you might to use an electronic version of a sliding gate that automatically opens when you approach. The common way to achieve this is by using private/public key pairs. The private key, stored in a file, remains at your computer and you must keep an eye on it—it must stay private. The public key, again stored in a file, is handed to the server. To create a private/public key pair proceed as show in Terminal 23.

```
Terminal 23: Creating Private/Public Key Pairs
1 # from client's home do ...
2 client$ mkdir ~/.ssh
3 client$ chmod 700 ~/.ssh
4 client$ ssh-keygen -q -f ~/.ssh/id_rsa -t rsa
5 # do NOT enter passphrase
6
7 # upload public key from client to user's home at server ...
8 client$ scp ~/.ssh/id_rsa.pub awkologist@servername.com:
9
```

```
10 # setup the public key on server
11 server$ mkdir ~/.ssh
12 server$ chmod 700 ~/.ssh
13 server$ cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
14 server$ chmod 600 ~/.ssh/authorized_keys
15 server$ rm ~/id_rsa.pub
```

Lines 1–8 have to be executed at you local computer (the client) and lines 9–14 at the remote server. In line 4 in Terminal 23 we execute the key generator `ssh-keygen`. The option `-t rsa` tells the application to generate a RSA (RSA stands for Rivest, Shamir and Adleman, the developers of the RSA public-key cryptosystem) key pair. The passphrase should be left empty by pressing **(Enter)**. If you enter a passphrase it will be requested instead of your original password each time you connect to the remote computer (server). This was exactly what we wanted to avoid. In line 8 we load the generated public file to the user's home directory at server.

Then we switch to the server. Here we create a directory named `.ssh` and move the content of `id_rsa.pub` (the one we created at the client computer) to `authorized_keys` and remove the original file. Done—from now on you can connect to the remote computer with `ssh` without entering the password. Finally you could generate an alias (see sect. 8.8 on p. 104), such as a nick name for the server, to save you some key strokes. Also cron jobs with `rsync` are not interrupted with password requests anymore.

Chapter 7

Playing with Text and Data Files

After learning some basics in the previous chapter, you will learn more advanced tools from now on. Since you are going to learn programming, you should be able to enter a program, that is, a text. Certainly your programs will not run immediately. This means you need to edit text files, too. In this chapter you will use very basic text editing tools. We will concentrate on the text editor Vim.

With Linux you have the choice from an endless list of text editors. From Microsoft Windows you might have heard about the *Notepad*. That is the standard Windows text editor. With Linux you can choose between `ed`, `vi`, `vim`, `elvis`, `emacs`, `nedit`, `kedit`, `gedit` and many more. In general, we can distinguish between three different types of text editors: (a) One group of text editors is line-orientated. `ed` belongs to this group. You can only work on one line of the text file and then need a command to get to the next line. This means that you cannot use the arrow keys to move the cursor (your current position) through the text. This is stone-age text editing and good only for learning purposes. (b) The next group of text editors is screen-oriented. Vim belongs to this group of editors. You see the text over the whole screen and can scroll up and down and move to every text position using the arrow keys. This is much more comfortable than using a line-oriented editor. We will predominantly work with Vim. It is powerful and comfortable enough for our purpose and usually available on all systems. You can even use it when you have no X-Server (see Sect. 3.3 on p. 26) running, which is often the case when you login to a remote computer. (c) The most comfortable text editors, of course, use the X-Server. However, you should not confuse comfortable with powerful! `nedit` or `kedit` are examples of editors which use the graphical user interface (GUI). Here, you can use the mouse to jump around in the text and copy and paste (drag and drop) marked text. We are not going to work with these nice editors.

7.1 Viewing and Analyzing Files

This section introduces some import tools that allow you to analyze text-based files in the command line.

7.1.1 A Quick Start: *cat*

Now, let us start with a very easy method to input text. For this, we use the command `cat` (concatenate). We have already used `cat` for displaying the content of a text file (see for example Terminal 7 on p. 46). We have also learned about redirecting the output of command into a file by using `>`. Now, when you use `cat` without specifying a filename you redirect the standard input (the keyboard) to the standard output (the screen). Let us try it out.

```
Terminal 24: Printing Text with cat
1  $ cat
2  this is text, now press Enter
3  this is text, now press Enter
4  now press Ctrl-Dnow press Ctrl-D now press Enter AND Ctrl-D
5  now press Enter AND Ctrl-D
6  $
```

In Terminal 24 we first run the command `cat` without anything. After hitting `(Enter)` we get into line 2 and enter the text “this is text, now press Enter”. At the end of line 2 we press `(Enter)`. The text we just entered will be printed again and we end up in line 4. Here we enter “now press Ctrl-D” and then press `(Ctrl) + (D)`. Again, the text will be printed, but now we end up in the same line. Now we press first `(Enter)` and then `(Ctrl) + (D)`. We are back. What happens is that the input from the keyboard is redirected to the screen. When we press `(Enter)` a new line starts before the redirection; but this is not important now. It is important that you realize that we could also redirect the standard input into a file! How?

```
Terminal 25: Writing Text with cat
1  $ cat > text-file
2  bla bla. Now press Enter
3  ahh, a new line! Now press Ctrl-D$ cat text-file
4  bla bla. Now press Enter
5  ahh, a new line! Now press Ctrl-D$
6  $ rm text-file
7  $
```

In Terminal 25 we redirect the standard input into a file named *text-file*. In line 2 we enter some text and then hit `(Enter)`. We end up in line 3, enter some text again and press `(Ctrl) + (D)`. We get back the command line directly. No new line is generated. This is uncomfortable! If we now, as shown in lines 3–5, display the text with `cat text-file`, we will see the same phenomenon in line 5: the dollar character is

on the right of this line, instead of being at a new line. However, if you now press **(Enter)** you get into a new line. Hint: You should always finish a text file with a new line. Finally, in line 6 we remove the file that we have just created with the command `rm`.

It is easy to create a text file, is it not? Now create a file named *gemones.txt* as shown in File 1.

```

1  File 1: gemones.txt
2  H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
3  A. thaliana (plant) - 100,000,000 bp - 25,000 genes
4  S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
5  E. coli (bacteria) - 4,670,000 bp - 3237 genes
6  4 lines text?
7  4 lines text?
8  203 characters?

```

Now let us see what we can do with the text file. We have already learned something about sorting the content of a text file in Terminal 9 on p. 48. Let us take a look at it again.

7.1.2 Text Sorting

With `sort` you can, as the name implies, sort the content of a text file. This can be very helpful in order to make files more readable. The `sort` command comes with some useful options, which are, as usual, separated by a dash.

- n Sorting takes into account the value of numbers. Otherwise, only their position in the alphabet would be considered and 50 would be printed before 8.
- r The sorting result is displayed in reverse order.
- f Lower and uppercase characters are treated equally.
- u Identical lines are printed only once. Thus you get unique or non-redundant output (see Sect. 7.1.3 on p. 81).
- k x Sorts according to the content of field *x*. For example, “-k 3” sorts according to the content of field 3.
- t s Set *s* as a new field separator (instead of the space character). For example, “-t :” sets the colon and “-t \t” sets the tabulator as field separator, respectively.

How does this work in real life?

```

1  Terminal 26: Sorting Text with sort
2  $ sort -nu gemones.txt
3  H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
4  4 lines text?
5  203 characters?
6  $ sort -n gemones.txt | sort -u
7  203 characters?
8  4 lines text?

```

```

8 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
9 E. coli (bacteria) - 4,670,000 bp - 3237 genes
10 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
11 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
12 $ sort -u genomes.txt | sort -n
13 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
14 E. coli (bacteria) - 4,670,000 bp - 3237 genes
15 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
16 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
17 4 lines text?
18 203 characters?
19 $ sort -u genomes.txt | sort -n > sorted-genomes.txt
20 $

```

In Terminal26 we use the options `-n` and `-u` in order to sort the content of the *genomes.txt* file, take care of correct sorting of numbers and get rid of doublets; but what is this? The output of `sort -nu genomes.txt` in lines 2–4 is slightly short! Three organisms are missing. Okay, here we encounter an error (*bug*) in the program `sort`. With such a well-established command as `sort`, this is a rather rare situation. What can we do? First of all you should report the bug to the Linux community and thereby help to improve the commands. In the manual pages (`man sort`) there is a statement where to report bugs: “Report bugs to bug-textutils@gnu.org”. Then you have to think about alternatives to your original task. In our case the alternative is shown in line 5. Here, we learn a new way of redirecting the output of a program: *piping*. Pipes are powerful tools to connect commands (see Sect. 8.6 on p. 103). We have already seen how to redirect the output from a command into a file (`>`). Now we redirect the output from one command to another. Therefore, we use the “`|`” character. The “`|`” is a pipe, and this type of pipe sends the stream of data to another program. The use of pipes means that programs can be very modular, each one executing some narrow, but useful task. In Terminal 26 line 5 we pipe the output of the `sort` command with the option `-n` (sort numbers according to their value) to the `sort` command with the option `-u` (eliminate duplicates). The result will be displayed on the screen. We could also redirect the result into a file, as shown in line 19.

The next example uses the options “`-t`” and “`-k`” in order to define a new field delimiter and the field, which is to be considered for sorting.

```

_____ Terminal 27: Field Directed Sorting with sort _____
1 $ sort -t - -k 3 -n genomes.txt
2 203 characters?
3 4 lines text?
4 4 lines text?
5 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
6 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
7 E. coli (bacteria) - 4,670,000 bp - 3237 genes
8 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
9 $

```

With the `sort` command in line 1 of Terminal27 the file *genomes.txt* is sorted according to the contents of field 3 (`-k 3`). The field delimiter is set to the dash character (`-t -`). With these options, `sort` becomes very versatile and can be used to sort tables according to the values of a certain column.

7.1.3 Extract Unique Lines

Frequently, it is desired to extract unique lines from a text file. One example would be that you delete duplicate gene IDs from a file. The command `sort` can do so with the option `-u`. I still wish to show an alternative to this with the `uniq` command. Terminal 28 shows an example.

```
Terminal 28: Extracting Unique Lines with sort and uniq
1  $ sort pdbfilelist.txt
2  1ANF.pdb
3  1ANK.pdb
4  1ANK.pdb
5  1ETN.pdb
6  2Y1B.pdb
7  $ sort pdbfilelist.txt | uniq
8  1ANF.pdb
9  1ANK.pdb
10 1ETN.pdb
11 2Y1B.pdb
12 $
```

The `uniq` command requires that the data lines are sorted. It will replace successive lines containing the same content with just one copy of this line.

7.1.4 Viewing File Beginning or End

Imagine you have a number of text files but you cannot remember their content. You just want to get a quick idea. You could use `cat` or a text editor, but sometimes it is enough to see just a couple of lines to remember the file content. `head` and `tail` are the commands of choice. `head filename` displays the first 10 lines of the file *filename*. `tail filename` does the same for the last 10 lines. You might want to use the option `-n`. The command

```
head -n 15 structure.pdb
```

displays the first 15 lines of the file *structure.pdb*. Of course, the `-n` option applies to `tail`, too.

Very handy comes the option `-f` of `tail`. It initiates a kind of real-time mode causing `tail` to wait for new lines being appended to the input file. This way you can monitor a log file of a BLAST result file. Monitoring can be stopped typing `(Ctrl) + (C)`. Note that this option is ignored if `tail` reads for a pipe and instead of a file.

7.1.5 Scrolling Through Files

Assuming that you are an eager scientist collecting lots of data and having accordingly large files, you might look for something else than `cat`, `head` or `tail`. Of course,

you want to see the whole file and scroll through it! If you want to read a long file on the screen, you should use `less`. (There is another program called `more`. It is older and thus not powerful. Just to let you know!). The syntax is easy:

```
less filename
```

With `less` you can scroll forward and backward through the text file named *filename*. After invoking `less`, the next page is displayed with `(Space)` and the next line is displayed with `(Enter)`. By pressing `(Q)` you quit `less`. After typing the slash character `(/)` you can enter a query term. With `(n)` and `(p)` you jump to the next and previous lines matching the query pattern, which can of course be a regular expression (Sect. 11 on p. 157). Another nice function is to jump to the end of the file with typing `(G)` or to the beginning of the file with `(1) → (G)`. Accordingly, you can jump to any line, e.g. line 34 by typing `(3) → (5) → (G)`.

On some systems you will find the command `zless`, which can be used in order to view compressed files.

7.1.6 Character, Word, and Line Counting

Another thing we might want to do is count the number of lines, words, and characters of a text file. This task can be performed with the command `wc` (word count). `wc` counts the number of bytes, whitespace-separated words and lines in each given file.

```

_____ Terminal 29: Counting Lines and Words and Characters with wc _____
1  $ wc genomes.txt
2      7      44      247 genomes.txt
3  $

```

As we can see from the output in Terminal 29, the file *genomes.txt* consists of 7 lines, 44 words, and 247 characters.

7.1.7 Splitting Files into Pieces

The `split` command splits files into a number of smaller files. Of the list of option I wish to restrict myself to `-l` (lines).

```

_____ Terminal 30: Splitting Files with split _____
1  $ split -l 2 genomes.txt genomes.
2  $ ls g*
3  genomes.aa genomes.ab genomes.ac genomes.ad genomes.txt
4  $ cat genomes.aa
5  H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
6  A. thaliana (plant) - 100,000,000 bp - 25,000 genes
7  $

```


The `split` command in Terminal 30 on the preceding page split the file *genomes.txt* into small files. Each resulting file contains two lines – 1 2 and the prefix of the resulting files is *genomes.*. The appendices *aa*, *ab* and so forth are set automatically.

7.1.8 Cut and Paste Columns

You are most probably used to cut-and-paste when writing with office applications or the like. This one here is different. The `cut` cuts columns out of a file. Columns are defined as text strings that are separated by a delimiter defined with `-d`. The option `-f` (fields) states the columns that shall be extracted. Terminal 31 shows how it works.

```
Terminal 31: Cutting Columns with cut
1  $ cut -f 1,3 -d "-" genomes.txt
2  H. sapiens (human) - 30,000 genes
3  A. thaliana (plant) - 25,000 genes
4  S. cerevisiae (yeast) - 6034 genes
5  E. coli (bacteria) - 3237 genes
6  4 lines text?
7  4 lines text?
8  203 characters?
9  $
```

The command `paste` on the other hand fuses files to a table contained in one single file. Terminal 32 illustrates how.

```
Terminal 32: Pasting Files to Columns with paste
1  $ cat file-1.txt
2  one11
3  one21
4  $ cat file-2.txt
5  two11
6  two21
7  $ cat file-3.txt
8  three11
9  three21
10 $ paste file-1.txt file-2.txt file-3.txt
11 one11 two11 three11
12 one21 two21 three21
13 $ paste -s file-1.txt file-2.txt file-3.txt
14 one11 one21
15 two11 two21
16 three11 three21
17 $
```

Assuming that you have three files with two lines of content each, `paste` would fuse the file contents to one single tab-delimited file. The option `-s` makes `paste` reading the files in serial order.

7.1.9 Finding Text: *grep*

A very important command is *grep* (global regular expression printer, for a historical background see Sect. 11.1 on p. 157). *grep* searches the named input file(s) for lines containing a match to a given pattern. By default, *grep* prints the matching lines. Terminal 33 illustrates its function.

```

1  $ grep human genomes.txt
2  H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
3  $ grep genes genomes.txt | wc -w
4      36
5  $ grep genes genomes.txt | wc -l
6      4
7  $ grep -c human genomes.txt
8      1
9  $ grep -c genes genomes.txt
10     4
11 $ grep 'H. sapiens' genomes.txt
12 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
13 $ grep 'bacteria\\|human' genomes.txt
14 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
15 E. coli (bacteria) - 4,670,000 bp - 3237 genes
16 $

```

The name *grep* has nothing to do with the verb to grab. It is derived from a function of a very old Unix text editor called *ed*. Terminal 34 shows how the text file *test.txt* can be opened with *ed* (line 4). The command line now accepts *ed* commands (I added comments to the commands in the terminal—you should not type these).

```

1  $ cat test.txt
2  line 1
3  line 2
4  line 3
5  $ ed test.txt
6  21                                # number of characters
7  %P                                # Print all lines
8  line 1
9  line 2
10 line 3
11 g/2/p                              # globally print lines matching RE
12 line 2
13 q                                  # quit
14 $

```

The command *g/2/p* searches globally (*g*) for matches to the regular expression */2/* (see Chap. 11 on p. 157) and prints (*p*) matching lines. Since the shortcut for the regular expression is *RE*, the command becomes *g/RE/p*, which is *grep*. *grep* really is much more powerful than you can imagine right now! It comes with a whole bunch of options, only one of which was applied in Terminal 33. In line 1 we search for the occurrences of the word *human* in the file *genomes.txt*. The matching line is printed out. In line 3 we search for the word *genes* and pipe the matching lines to the program *wc* to count the number of words (option *-w*) of these lines. The

result is 36. Next, in line 5, we count the number of lines (`wc` option `-l`) in which the word *genes* occurs; but we can obtain this result much more easily by using the `grep` option `-c` (count). With this option `grep` displays only the number of lines in which the query expression occurs. How can we search for two strings at once? This is shown in line 13. Note that we enclose the words we are searching for in single quotes (you could also use double quotes); and how about querying for two words that are not necessarily in one line? Use the combination “`\|`” between the words as shown in line 13 of Terminal 33 on the preceding page. This stands for the logical *or*. Be careful not to include spaces!

At this stage you should already have a feeling about the power of the Linux command line. We will get back to `grep` when we talk about regular expressions in Chap. 11 on p. 157. Just one last thing: with the option `--color` you can highlight the matching text in the terminal.

7.1.10 Text File Comparisons

Two interesting commands to compare the contents of text files are `diff` and `comm`. To see how these commands work, create two text files with a list of words, as shown in the following Terminal:

```

1  $ cat>amino1
2  These amino acids are polar:
3  Serine
4  Tyrosine
5  Arginine
6  $ cat>amino2
7  These amino acids are polar and charged:
8  Lysine
9  Arginine
10 $ diff amino1 amino2
11 1,3c1,2
12 < These amino acids are polar:
13 < Serine
14 < Tyrosine
15 ---
16 > These amino acids are polar and charged:
17 > Lysine
18 $ diff -u amino1 amino2
19 --- amino1      2003-05-11 18:42:53.000000000 +0200
20 +++ amino2      2003-05-11 18:43:30.000000000 +0200
21 @@ -1,4 +1,3 @@
22 -These amino acids are polar:
23 -Serine
24 -Tyrosine
25 +These amino acids are polar and charged:
26 +Lysine
27 +Arginine
28 $ diff -c amino1 amino2
29 *** amino1      2003-05-11 18:42:53.000000000 +0200
30 --- amino2      2003-05-11 18:43:30.000000000 +0200
31 *****
32 *** 1,4 ****
33 ! These amino acids are polar:
34 ! Serine

```

```

35 | ! Tyrosine
36 |   Arginine
37 | --- 1,3 ----
38 | ! These amino acids are polar and charged:
39 | ! Lysine
40 |   Arginine
41 | $

```

The result of the command `diff` (difference) indicates what you have to do with the file *amino1* to convert it into *amino2*. You have to delete the lines marked with “<” and add the lines marked with “>”. With the option `-u` (line 18) the same context is shown in another way. The option `-c` (line 28) is used to display the differences only (indicated by a “!”).

The command `comm` (compare) requires that the input files are sorted. We do this in line 1 in Terminal 36. Note: we can write several commands in one line, separating them with the semicolon character (;).

```

----- Terminal 36: Finding Things in Common with comm -----
1 | $ sort amino1>amino1s; sort amino2>amino2s
2 | $ comm amino1s amino2s
3 |       Arginine
4 |       Lysine
5 | Serine
6 | These amino acids are polar:
7 |       These amino acids are polar and charged:
8 | Tyrosine
9 | $ comm -12 amino1s amino2s
10 | Arginine
11 | $

```

The `comm` command prints out its result in three columns. Column one contains all lines that appear only in the first file (*amino1s*), column two shows all lines that are present only in the second file (*amino2s*), and the third column contains all files that are in both files. You can restrict the output with the options `-n`, *n* being one or more column numbers, which are not to be printed. Line 9 in Terminal 36 displays only the content of the third column (“minus 1 and minus 2”), which contains the common file content.

7.2 Editing Text Files

It is very helpful to know how to edit files in the command line. Suppose you are connected to a remote computer without graphical interface. The only chance then is to edit files with a command line editor. There are several such editors available. Here I present two of them. The Pico editor might help you when you forget how to use Vim. I highly recommend you to learn Vim—it is much much much more powerful than Pico.



Fig. 7.1 Pico. The main window of the text editor Pico

7.2.1 The Sparse Editor Pico

Probably the easiest way to handle text editor is called `pico` (**p**ine **c**omposer). It has the same look and feel as an old DOS editor (see Fig. 7.1). However, `pico` is not installed on all Linux systems. Thus, you should make the effort to learn `vi` (see Sect. 7.2.2), which is a bit harder but more universal and which offers you many more possibilities (in fact, you need to memorize only six commands in order to work Vim). The `pico` editor was developed by the University of Washington. You start `pico` with the command `pico` and, optional, with a filename. If the file already exists, it will be opened. When you write a program it is highly recommended to use the option `-w` (wrap). It tells `pico` not to break (wrap) the lines at the right margin of the editing window, but only when you press the `(Enter)` key. Thus, the command to start `pico` would look like this: `pico -w filename`. If you provide no filename, that is fine as well. You can save your text later and provide a filename at that time. When you have started `pico` you will get a kind of a graphical interface (see Fig. 7.1). The top of the display, called status line, shows the version number of `pico` that you use, the current file being edited and whether or not the file has been modified. The third line from the bottom is used to report informational messages and for additional command input. Now comes the comfortable part of `pico`: the bottom two lines list the available editing commands. Depending on your current action, the content of this window might change.

Each character you type is automatically inserted at the current cursor position. You can move the cursor with the arrow keys. If this does not work, you have to use the following commands:

terminal or workstation in existence without having to worry about unusual keyboard mappings. You see, there are really many advantages to learning Vim.

In celebration of its twentieth birthday in 2011 Vim has even been ported to Apple's iOS (<http://applidium.com/en/applications/vim/>) and other mobile devices, e.g., Android (<http://play.google.com/store/apps/details?id=com.easynk.avim>). Astonishingly, it has many features of the native version such as automatic indentation, a visual mode, language-aware syntax highlighting, integrated scripting language that lets you extend functionalities, macros recording and playback, markers management to quickly move around the edited file or multiple clipboards.

7.2.3 Installing Vim

In case that you are working with Ubuntu, e.g., as outlined in Sect. 4.1.2 on p. 40, you need to install Vim. By default, Ubuntu comes with Vi installed. However, Vim is much more comfortable to use. Therefore, execute the following command in the terminal to install Vim:

```
sudo apt-get install vim
```

Even if you now enter `vi`, Linux executes `vim`. Have fun.

7.2.4 Immediate Takeoff

If you cannot wait to use Vim and make your first text file: here you go. Type `vi` to start the editor. Press `i` to start the insertion mode and enter your text. You start a new line by hitting `(Enter)`. With `(Esc)` you leave the insertion mode. With `:wq filename` you quit Vim and save your text in a file called *filename*, whereas with `:q!` you quit without saving. That's it. Vim can be simple!

Note: If you use the cursor keys and see letters instead of a moving cursor do the following: Hit `(Esc)` twice; type a column `(:)`; type `set nocompatible`; press `(Enter)`. Now it should work.

7.2.5 Starting Vim

Now let us get serious. To start Vim, enter: `vi filename`, where *filename* is the name of the file you want to edit. If the file does not exist, Vim will create it for you. You can also start Vim without giving any filename. In this case, Vim will ask for one when you quit or save your work. After you called Vim, the screen clears and displays the content of the file *filename*. If it is a new file, it does not contain any text. Then Vim uses the tilde character (`~`) to indicate lines on the screen beyond the end of the file. Vim uses a cursor to indicate where your next command or text insertion will take effect. The cursor is the small rectangle, which is the size of one character,

and the character inside the rectangle is called the current character. At the bottom of the window, Vim maintains an announcement line, called the *mode line*. The mode line lists the current line of the file, the filename, and its status. Let us now start Vim with the new file *text.txt*. The screen should then look like Terminal 37. Please note that I have deleted some empty lines in order to save rain forest, i.e., paper. The cursor is represented by `[]`.

```

1  []
2  ~
3  ~
4  ~
5  ~
6  ~
7  ~
8  ~
9  ~
10 "text.txt" [New File]          0,0-1      All
```

7.2.6 Modes

Line 10 in Terminal37 shows the mode line. At this stage you cannot enter any text because Vim runs currently in the command mode. In order to enter text, the input mode must be activated. To switch from the command mode to the input mode, press the `i` key (you do not need to press `(Enter)`). Vim lets you insert text beginning at the current cursor location. To switch back to command mode, press `(Esc)`. You can also use `(Esc)` to cancel an unfinished command in command mode. If you are uncertain about the current mode, you can press `(Esc)` a few times. When Vim beeps, you have returned to the command mode. Okay, let us change to the input mode and enter some text.

```

1  This is new text in line 1. Now I press ENTER
2  and end up in the second line. I could also write to the e
3  nd of the line. The text will be wrapped automatically[]
4  ~
5  ~
6  ~
7  ~
8  ~
9  ~
10 -- insert --                      3,54      All
```

You can see some changes in the mode line in Terminal38. “-- insert --” indicates that you are in the input mode. Furthermore, the current cursor position (line 3, column 54) is indicated. Now press `(Esc)` to get back into the command mode, “-- insert --” will disappear. Now let us save the file: press `:w`, and then `(Enter)`.

The mode line will display a message as shown in Terminal 39. If “:w” appears in your text you are still in the input mode!

```

1 This is new text in line 1. Now I press ENTER
2 and end up in the second line. I could also write to the e
3 nd of the line. The text will be wrapped automatically
4 ~
5 ~
6 ~
7 ~
8 ~
9 ~
10 ``text.txt'' [new] 3L, 160C written      3,54      All

```

Commands are very often preceded with the colon “:” character. Let us try another command: type `:set number`. Now you see line numbers in front of each line. With `:set nonumber` you hide them again. Another command: type `:r !ls`. Woop. After hitting `(Enter)` you have a list of all files in your current working directory imported to your text file. That is magic, is it not? Vim executed the shell command `ls` and imported the result into the text file at the current cursor position.

7.2.7 Moving the Cursor

Probably your screen is quite full now. Let us move the cursor. Usually you can use the arrow keys (if you use the cursor keys and see letters instead of a moving cursor do the following: Hit `(Esc)` twice; type a column `(:)`; type `set nocompatible`; press `(Enter)`); but if they do not work you can use the following keys:

```

h    move one character to the left
l    move one character to the right
k    move up one line
j    move down one line

```

There are some more powerful commands for long-distance jumping.

```

0      move to the beginning of the line
$      move to the end of the line
H      move to the top line of the screen
M      move to the middle line of the screen
L      move to the bottom line of the screen
G      move to the last line of the text
nG     move to the n-th line in the text
(Ctrl)+(f)  move one screen forward
(Ctrl)+(b)  move one screen backward

```

You have now learned some powerful tools for moving around in a file. You should memorize only the basic movements and know where to look up for the others (you can download a cheat sheet at <http://www.kcomputing.com/vi.html>).

7.2.8 Doing Corrections

Maybe the most relaxing thing to know is that you can always *undo* changes by typing `u`. With Vim you can even undo many commands, whereas Vi will recover only the last text change. If you are lucky, you can use the keys `(BkSp)` and `(Del)` in order to make *deletions* in the input mode. Otherwise you must make use of the commands in the command modus. To do so, first move the cursor so that it covers the first character of the group you want to delete, then type the desired command from the list below.

<code>x</code>	delete only the current character
<code>D</code>	delete to the end of the line
<code>db</code>	delete from the current character to the beginning of the current word
<code>de</code>	delete from the current character to the end of the current word
<code>dd</code>	delete the current line
<code>dw</code>	delete from the current character to the beginning of the next word

Note that the second letter of the command specifies the same abbreviations as the cursor movement commands do. In fact, you can use delete with all of the cursor movement specifiers listed above, e.g., `dH` would delete everything from the current line to the top line off the screen.

In other cases you will need only to *replace* a single character or word, rather than deleting it. Vim has change and replace functions, too. First, move to the position where the change should begin (the desired line or the beginning of the desired word). Next, type the proper command from the list below. Finally, enter the correct text, usually concluded with `(Esc)` (except for `r`).

<code>cw</code>	Change a word.
<code>C</code>	Overwrite to the end of the line.
<code>r</code>	Replace a single character with another one. No <code>(Esc)</code> necessary.
<code>R</code>	Overwrite characters starting from the current cursor position.
<code>s</code>	Substitute one or more characters for a single character.
<code>S</code>	Substitute the current line with a new one.
<code>:r file</code>	Insert an external file at the current cursor position.

The change command `c` works like the delete command; you can use the text portion specifiers listed in the cursor movement list.

7.2.9 Save and Quit

Vim provides several means of saving your changes. Besides saving your work before quitting, it is also a good idea to save your work periodically. Power failures or system crashes can cause you to lose work. From the command mode, you type `:w` (write) and hit `(Enter)`. In order to save the text in a new file, type `:w filename`. You quit Vim with `:q`. You can save and quit at once with `:x` or `:wq`. If you do not want to save your changes you must force quitting with `:q!`. Be cautious when abandoning Vim in this manner because any changes you have made will be permanently lost. Up to this point you have learned more than enough commands to use Vim in a comfortable way. The next two sections explain some more advanced features which you might wish to use.

7.2.10 Copy and Paste

Frequently, you will need to cut or copy some text, and paste it elsewhere into your document. Things are easy if you can work with the mouse. When you mark some text with the mouse (holding the left mouse button) the marked text is in the memory (*buffer*). Pressing the right mouse button (or, on some systems, the left and right or the middle mouse buttons) pastes the text at the current cursor position. You can apply the same mechanism in the terminal window!

Things are slightly more complicated if you have only the keyboard. First you cut or copy the text into temporary storage, then you paste it into a new location. Cutting means removing text from the document and storing it, while copying means placing a duplicate of the text in storage. Finally, pasting just puts the stored text in the desired location. Vim uses a *buffer* to store the temporary text. There are nine numbered buffers in addition to an undo buffer. The undo buffer contains the most recent delete. Usually buffer 1 contains the most recent delete, buffer 2 the next most recent, and so forth. Deletions older than 9 disappear. However, Vim also has 26 named buffers (a–z). These buffers are useful for storing blocks of text for later retrieval. The content of a buffer does not change until you put different text into it. Unless you change the contents of a named buffer, it holds its last text until you quit. Vim does not save your buffers when you quit.

The simplest way to copy or move text is by entering the source *line numbers* and the destination line numbers. The `m` command moves (cuts and pastes) a range of text, and the `t` command transfers (copies and pastes) text. The commands have the syntax shown below:

<code>:xmy</code>	Move line number x below line number y.
<code>:x, y mz</code>	Move the lines between and including x and y below line z.
<code>:xt y</code>	Copy line x below line y.
<code>:x, y t z</code>	Copy lines between and including x and y below line z.

Another way is to use *markers*. You can mark lines with a letter from a to z. These markers behave like invisible bookmarks. To set a mark you use `mx`, with `x` being a letter from a to z. You can jump to a mark with `'x`. The following list shows you how to apply bookmarks to copy or move text. Note: bookmarks and line numbers can be mixed.

<code>mx</code>	Set a bookmark at the current line. <code>x</code> can be any letter from a–z.
<code>'x</code>	Jump to bookmark <code>x</code> .
<code>: 'x' y co 'z</code>	Copy lines between and including bookmarks <code>x</code> and <code>y</code> below bookmark <code>z</code> .
<code>: 'x' ym 'z</code>	Move lines between and including bookmarks <code>x</code> and <code>y</code> below bookmark <code>z</code> .
<code>: 'x, 'y w file</code>	Write lines between and including bookmarks <code>x</code> and <code>y</code> into a file named <i>file</i> .

One last method uses the commands `d` (delete) or `y` (yank). With this method you can make use of different *buffers*. Go to the line you wish to copy or cut and press `yy` (yank) or `dd` (delete), respectively. Then move the cursor to the line behind which you want to insert the text and type `p` (paste). In order to copy a line into a buffer type `"xyy`, with `x` being a letter from a–z. You insert the buffer with `"xp`. To copy more than one line precede the command `yy` with the number of lines. `2yy` copies three lines and `3yw` copies three words. You see, Vim is very flexible and you can combine many commands. If you are going to work a lot with it you should find out for yourself which commands you prefer.

7.2.11 Search and Replace

Finally, let us talk about another common issue: searching and replacing text. As files become longer, you may need assistance in locating a particular instance of text. Vim has several search and search and replace features. Vim can search the entire file for a given string of text. A string is a sequence of characters. Vim searches forward with the slash (`/`) or backward with the question mark key (`?`). You execute the search by typing the command, then the string followed by `(Enter)`. To cancel the search, press `(Esc)` instead of `(Enter)`. You can search again by typing `n` (forward) or `N` (backward). Also, when Vim reaches the end of the text, it continues searching from the beginning. This feature is called *wraps**can*. Of course, you can use wildcards or regular expressions in your search. We will learn more about this later in Sect. 11.6.1 on p. 172. Let us take a look at the search and replace commands:

<code>/xyz</code>	Search forward for xyz.
<code>?xyz</code>	Search backward for xyz.
<code>n</code>	Go to the next occurrence.
<code>N</code>	Go to the previous occurrence.
<code>:s/abc/xyz/</code>	Replace the first instance of <i>abc</i> with <i>xyz</i> in the current line.
<code>:s/abc/xyz/g</code>	Replace all instances of <i>abc</i> with <i>xyz</i> in the current line.
<code>:s/abc/xyz/gc</code>	Ask before replacing each instance of <i>abc</i> with <i>xyz</i> in the current line.
<code>:%s/abc/xyz/gc</code>	Replace all instances of <i>abc</i> with <i>xyz</i> in the whole file.
<code>:%s/abc/xyz/gc</code>	Ask before replacing each instance of <i>abc</i> with <i>xyz</i> in the whole file.
<code>:x,y s/abc/xyz/g</code>	Replace all instances of <i>abc</i> with <i>xyz</i> between lines <i>x</i> and <i>y</i> .

Remember, this is only a small selection of commands that I present here.

7.3 Text File Conversion (Unix ↔ DOS)

It is common that text files generated either on Linux or DOS/Windows cause problems on the other system. The reason lies in different syntax for the newline command, i.e., the file ending. An easy way out is provided by the commands `unix2dos filename(s)` and `dos2unix filename(s)`, respectively.

If this command is not available on your system you would have to install it. Ubuntu users can run `sudo apt-get install dos2unix`, which installs both `dos2unix` and `unix2dos`.

7.3.1 Batch Editing

Finally, let me show you that Vim commands can be handled like a programming language, i.e., you can store them in a file and execute the commands as a batch job. In the following example we work on File 1 on p. 79 again. First, create a text file containing the Vim commands as shown in File 2. Note that new commands are allowed in this file.

```

1  :g/^[^A-Z]/d
2  :%s/,./g
3  :%s/ - /\t/g
4  :w %<.new
5  :q

```

File 2: *edit.vi*

The first command globally deletes all lines matching the pattern, i.e., all lines not starting with an uppercase letter (see Chap. 11 on p. 157). Then all commas are substituted with dots throughout the file. Command three replaces `\t` with tabulators. Line 4 makes use of the special variable `%` that holds the filename. `%<` folds the filename without the filename extension. Thus, `%<.new` converts the filename *genomes.txt* into *genomes.new*. `:w` writes the edited content to that file. Finally the Vim session is quitted with `:q`.

Terminal 40 shows the execution.

```

1  $ vi -e genomes.txt < edit.vi
2  $ cat genomes.new
3  H. sapiens (human) 3.400.000.000 bp      30.000 genes
4  A. thaliana (plant) 100.000.000 bp 25.000 genes
5  S. cerevisiae (yeast) 12.100.000 bp 6034 genes
6  E. coli (bacteria) 4.670.000 bp 3237 genes
7  $

```

Take a look at Sect. 8.10 on p. 106, how you would run this batch job on several files.

If you have followed this section about Vim up to this stage, you should have obtained a very good overview of its capabilities. It can do more and it offers a whole range of options that one could set to personal preferences. However, since I do not believe that anyone is really going deeper into this, I stop at this point. You are welcome to read some more lines about Vim in focussed books (Qualline 2001; Robbins 1999; Lamb and Robbins 1998).

Exercises

Now sit down and play around with some text. This is elementary!

7.1. Create a text file named *fruits.txt* using `cat`. Enter some fruits, one in each line. Append some fruits to this file.

7.2. Create a second text file named *vegetable* containing a list of vegetables, again, one item per line. Now concatenate fruits and vegetables onto the screen and into a file named *dinner*.

7.3. Sort the content of *dinner*.

7.4. Take some time and exercise with Vim. Open a text document with Vim or type some text and go through the description in this section. You must know the basic commands in order to write and edit text files!

Chapter 8

Using the Shell

The power of Linux lies in its shell. Of course, it is nice and comfortable to run and control programs with a graphical user interface and the mouse. The maximum of flexibility, however, you gain from the shell. In this chapter you will learn some basic features of the shell environment.

8.1 What is the Shell?

One of the early design requirements for Unix was the need for a small kernel. The kernel is that part responsible for the most fundamental functions and operations of the operating system. It can be seen as a subset of the overall operating system. As already mentioned before, the kernel interfaces the user applications with the hardware (see Chap. 3 on p. 21). In order to reduce the amount of code in the kernel, and thus the memory requirement of the kernel, many parts of the operating system were moved into user processes. By this, the directory */bin* filled up with commands like `ls`, `mv`, `mkdir` and so on. You access these commands via the shell. In fact, everything we have done so far and will do later happens “in the shell”.

Since we run commands from the shell, the shell can be seen as a *command interpreter*. It recognizes the command and takes care of its correct execution. This involves handling the hardware via the kernel. In this section we will see that the shell also provides a powerful *programming language*. Thus, the shell is a command interpreter and a programming language. With the programming language you can easily automate processes or combine many commands to one. If you have some experience with DOS: the shell resembles the DOS file *command.com* and DOS’s programming language for *batch files*. You will learn more about the power of the shell.

8.2 Different Shells

Until now we have always talked about *the* shell. However, there are *many* shells around. We will mainly work with the `bash` shell. I wrote *bash* like a command, because it is a command. Whenever you log into a system you can type `bash`. This opens a new `bash` shell for you. You can exit the shell with `exit`. When you type `exit` in your login shell (the shell you are in after you have logged in to the system) you actually logout! When you try to logout from a shell other than your login shell you will see an error message.

```

1 login as: Freddy
2 Sent username "Freddy"
3 Freddy@192.168.1.5's password:
4 [Freddy@rware2 Freddy]$ bash
5 [Freddy@rware2 Freddy]$ logout
6 bash: logout: not login shell: use 'exit'
7 [Freddy@rware2 Freddy]$ exit
8 exit
9 [Freddy@rware2 Freddy]$ sh
10 sh-2.05b$ exit
11 exit
12 [Freddy@rware2 Freddy]$ ksh
13 -bash: ksh: command not found
14 [Freddy@rware2 Freddy]$ csh
15 [Freddy@rware2 ~]$ exit
16 exit
17 [Freddy@rware2 Freddy]$ echo $0
18 -bash
19 [Freddy@rware2 Freddy]$
```

Terminal 20 and Fig. 8.1 illustrate how different shell levels are opened. All these shells are called interactive shells because they wait for your input.

In Terminal 41, the user Freddy logs in to the system and ends up in the login shell in line 4. Now, Freddy opens a new `bash` shell with the command `bash`. If he tries to logout from this shell, Freddy sees an error message (line 6). However, he can exit the shell with the `exit` command and gets back to his login shell in line 9. From here, Freddy could logout with the command `logout`. Now you see the difference between `logout` and `exit`.

The `bash` shell is one of the modern shells. The name is an acronym, standing for *Bourne-Again Shell*. The first shell was the Bourne shell developed in 1978 by Steve Bourne from the Bell Laboratories. All other shells were developed later. Nowadays, you will find on most systems the *Bourne Shell* (`sh`), *C Shell* (`csh`) and *Bash Shell* (`bash`). You can see in Terminal 41 how to enter these shells with their respective

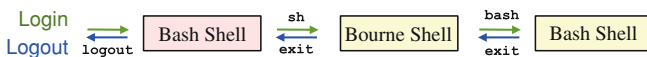


Fig. 8.1 Shell Hopping. From the login shell (red) one can open new shells (yellow). In order to logout again one has to “go back” to the login shell by exiting all opened shells

command. If a shell is not installed on the system, you will see an error message as in line 13. In line 12 Freddy tried to open the *Korn Shell* (`ksh`) but found that it is not installed on his system. Terminal 41 also displays differences of the *command line prompt* (that is something like “[Freddy@rware2 Freddy]\$”). In all other terminals, we omitted this part and you will just see the `$` character (take a look at Sect. 8.9 on p. 105 for an example of how to change the shell prompt). Here, the bash prompt says that the user *Freddy* is logged into the system *rware2* and is currently in his home directory called *Freddy*. In contrast, the Bourne shell prompt in line 10 shows only the version of the shell, and the C shell prompt in line 15 looks similar to the bash prompt. All these prompts can be set up according to your personal preferences. If you do not know in which shell you are, type `echo $0` as shown in line 17 of Terminal 41. This command is available in all shells.

8.3 Setting the Default Shell

When you login into your system you can use the command `echo $0` to check out your default shell. You can easily change your default shell with the command `chsh` (change shell).

```
Terminal 42: The Shell chsh
1  $ cat /etc/shells
2  /bin/sh
3  /bin/bash
4  /sbin/nologin
5  /bin/bash2
6  /bin/ash
7  /bin/bsh
8  /bin/tcsh
9  /bin/csh
10 $ chsh
11 Change Shell for Freddy.
12 Password:
13 New Shell [/bin/bash]: /bin/csh
14 Shell changed.
15 $
```

In Terminal 42 we see how to find out which shells are installed on your system. There is a list available in the file `/etc/shells`. On my system, I can choose between eight different shells. In order to change the login shell, type `chsh`. Now you have to enter your password and give the path to your desired shell. The new shell will be activated after a new login.

8.4 Useful Shortcuts

The bash shell provides some very nice shortcuts. These make your life easier when working in the terminal. One very nice feature is the history file. The bash shell remembers all the commands you typed. They are saved in a file in your home

directory. The file is called `.bash_history`. You can scroll through your last commands with the arrow keys (`↑` and `↓`). With `Ctrl+R` you can search the history file for a command. After you have typed `Ctrl+R` you see the message “reverse-i-search:”. If you now type characters into the command line, bash is querying the `.bash_history` file for the most recent match. With `Enter` you can execute the command and by pressing `Tab` you paste the command into the command line and you can edit it. If you instead hit `Ctrl+R` again, bash searches for the next match in the history file. Try it out to get a feeling for it; but try it with commands like `cd`, not with commands like `rm`, otherwise you might, by error, delete a file.

There is an alternative way to search the history file: by using `grep`. This is very helpful when you remember that you have typed a certain command in the past few days but you do not want to use the up arrow 400 times to find it. The next time you find yourself in such a situation, type:

```
history|grep -i keyword
```

This will read the history file and perform a case-insensitive search on the keyword you are looking for.

```

Terminal 43: Querying Command History
1  $ history|grep -i sort
2      140  man sort
3      141  info sort
4      143  info sort
5      816  sort amino1>amino1s; sort amino2>amino2s
6      1060  ls -l |sort +4
7      1066  ls -l |sort +4 -nr
8      1077  alias llss="lls|sort +4 -n"
9      1085  history|grep -i sort
10 $ !1060
11 ls -l |sort +4 -nr
12 -rwxrw-r-- 1 Freddy Freddy 321 Mai 17 11:54 spaces.sh
13 -rw-rw-r-- 1 Freddy Freddy 59 Mai 17 12:40 err.txt
14 -rw-rw-r-- 1 Freddy Freddy 11 Mai 17 13:51 list.txt
15 -rw-rw-r-- 1 Freddy Freddy 3 Mai 17 13:58 new
16 insgesamt 20
17 $

```

Terminal 43 demonstrates its use. In line 1, we search for all past commands in the history file that used the command `texttsort`. There are eight hits, the most recent one being at the bottom in line 9. Each command is preceded by an unambiguous identifier. We can use this identifier to execute the command. This is done in line 10. Note that you need to precede with an exclamation mark (`!`). This is cool, isn't it?

The following list shows you a couple of shortcuts that work in the bash shell:

↑ and ↓	Scroll up and down in the command history
⏮ and ⏭	Jump to the start or end in the command history
← and →	Move forth and back in the command line
Ctrl+B and Ctrl+F	As ← and →
Alt+B and Alt+F	Move forth and back word-wise
Home and End	Move to the beginning or end of the command line
Ctrl+A and Ctrl+E	As Home and End
Ctrl+L	Clear the screen
Ctrl+R	Query command history
Tab	Expand commands or filenames
! \$	Last word of the previous command, often a file-name
!!	Repeat the last command

The last shortcut in this list deserves our special attention. After you have entered the first characters of a command, you can press the tabulator key (Tab). If the command is already unambiguously described by the characters you entered, the missing characters will be attached automatically. Otherwise you hear a system beep. When you press the tabulator key twice, all possible command names will be displayed. The same functions with filenames. Try it out! This is really a comfortable feature!

8.5 Redirections

We have already used redirections in previous sections. The main purpose was to save the output of a command into a file. Now let us take a closer look at redirections. In Linux, there are three so-called *file descriptors*: standard input (stdin), standard output (stdout), and standard error (stderr).

By default, all programs and commands read their input data from the standard input. This is usually the keyboard. When you enter text using the text editor `vi` the text comes from the keyboard. The data output of all programs is sent to the standard output, which by default is the screen. When you use the command `ls`, the result is printed on the screen (or the active terminal when you use X-Windows). The standard error is displayed on the screen, too. If you try to execute the nonexistent command `textttlls` you will see an error message on the screen.

It is often convenient to be able to handle error messages and standard output separately. If you do not do anything special, programs will read standard input from your keyboard, and they will send standard output and standard error to your terminal's display. The shell allows you to redirect the standard input, output, and error. Basically, you can redirect *stdout* to a file, *stderr* to a file, *stdout* to *stderr*, *stderr* to

Table 8.1 Common standard input/output redirections for the C shell and Bourne shell

Function	csh	sh
Send <i>stdout</i> to file	prog > file	prog > file
Send <i>stderr</i> to file		prog 2> file
Send <i>stdout</i> and <i>stderr</i> to file	prog >& file	prog > file 2>&1
Take <i>stdin</i> from file	prog < file	prog < file
Append <i>stdout</i> to end of file	prog >> file	prog >> file
Append <i>stderr</i> to end of file		prog 2>> file
Append <i>stdout</i> and <i>stderr</i> to end of file	prog >>& file	prog >> file 2>&1

The numbers “1” and “2” stand for the standard output and the standard error, respectively

stdout, *stderr* and *stdout* to a file, *stderr* and *stdout* to *stdout*, *stderr*, and *stdout* to *stderr*.

As already mentioned above, standard input normally comes from your keyboard. Many programs ignore *stdin*. Instead, you enter, e.g., filenames together with the command. For instance, the command `cat filename` never reads the standard input; it reads the filename directly. However, if no filename is given together with the command, Linux commands read the required input from *stdin*. Do you remember? We took advantage of this trick in Sect. 7.1.1 on p. 78. In Table 8.1 you find a summary of the syntax for redirections. You should be aware that different shells use different syntax!

Now let us take a look at some examples from the `bash` shell. In the following list, the numbers “1” and “2” stand for the standard output and the standard error, respectively. Keep this in mind. This will help you in understanding the following examples.

<code>ls -l> list.txt</code>	Output is written into the file <i>list.txt</i>
<code>ls -l 1> list.txt</code>	Output is written into a file <i>list.txt</i>
<code>ls -l>> list.txt</code>	Output is appended to the file <i>list.txt</i>
<code>ls -e 2> error.txt</code>	Error is written into the file <i>error.txt</i>
<code>ls -e 2>> error.txt</code>	Error is appended to the file <i>error.txt</i>
<code>ls>> all.txt 2 >&1</code>	Output and error are appended to the file <i>all.txt</i>
<code>ls &> /dev/null</code>	Send all output to the nirvana. The program will execute its task quietly. <code>/dev/null</code> is the nirvana, a special device that disposes of unwanted output
<code>cmd< file1 > file2</code>	The hypothetical command <i>cmd</i> reads <i>file1</i> and sends the output to <i>file2</i>

You see, there are many ways to redirect the input and output of programs. Let us finally consider that you wish to redirect the output of a command into a file *and* to display it on the screen. Do you have to run the command twice? Not with Unix and Linux! You can use the command `tee`. With

```
ls | tee filename
```

you see the content of the current directory on the screen and save it in a file named *filename*. With the option `-a`, the result is *appended* to the file *filename*.

8.6 Pipes

Pipes let you use the output of one program as the input of another one. For example you can combine

```
ls -l | sort
```

to sort the content of the current directory. We will use pipes later in conjunction with `sed`. For example, in

```
ls -l | sed s/txt/text/g
```

the command `ls -l` is executed and sends (pipes) its output the `sed` program that substitutes all occurrences of *txt* by *text*. Another common way to use a pipe is in connection with the `grep` command (see Sect. 7.1.9 on p. 83). With the line

```
cat publications | grep AIDS
```

all lines of the file called *publications* that contain the word *AIDS* will be displayed. Another great pipe you should remember is the combination with the command `less` (see Sect. 7.1.5 on p. 81). This command displays large texts in a rather nice way: you can scroll through the output. You get back to the command line after hitting `Q`. Imagine you have a large number of files in your home directory and run `ls -l`. Most probably, the output will not fit onto the screen. This means that you miss the beginning of the text. In such cases you should use

```
ls -l | less
```

in order to be able to scroll through all the output. The following command offers an interesting combination and a good example of the power of redirections and pipes:

```
ls -l | tee dir-content | sort +4 -n  
> dir-content-size-sorted
```

The file *dir-content* will contain the content of the current directory. The screen output is piped to `sort` and sorted according to file size (the option `+4` points to column 4). The result is saved in the file *dir-content-size-sorted*.

8.7 Lists of Commands

Usually, if you want to execute several commands in a row, you would write a shell script. However, sometimes it is faster to write the commands on the command line. A list of commands is a sequence of one or more commands separated by “;”, “&&”, or “||”. Commands separated by a “;” are executed sequentially; the shell waits for each command to terminate. The control operators “&&” and “||” denote *AND* lists and *OR* lists, respectively. An *AND* list has the form

```
command1 && command2
```

Command2 is executed if, and only if, command1 returns an exit status of zero (this means that no error has occurred). An *OR* list has the form

```
command1 || command2
```

Command2 is executed if, and only if, command1 returns a nonzero exit status (that is an error). Let us take a look at one example.

```

Terminal 44: Command List
1  $ find ~ -name "seq" && date
2  /home/Freddy/seq
3  /home/Freddy/neu/seq
4  /home/Freddy/neu/temp/seq
5  Mit Mai 28 21:49:25 CEST 2003
6  $ find ~ -name "seq" || date
7  /home/Freddy/seq
8  /home/Freddy/neu/seq
9  /home/Freddy/neu/temp/seq
10 $

```

In both cases, in Terminal 44 the command `find` finishes execution without error. This leads to the execution of the `date` command in the first (&&), but not in the second case (||).

8.8 Aliases

A very nice feature is the alias function. It allows you to create shortcuts (aliases) for every command or combination of commands. Let us assume you have pretty full directories. The command `ls -l` fills up the screen and you even lose a lot of lines, because the screen can display only a limited number of lines. Thus, you wish to pipe the output of `ls -l` to the command `less`. Instead of typing every time `ls -l | less` you can create an alias. The alias is created with the command `alias`:

```
alias lls="ls -l | less"
```

When you now type `lls` the command `ls -l | less` is executed. You can also use your alias in a new alias:

```
alias llss="ls | sort +4 -nr"
```

sorts the output according to file sizes, beginning with the largest file. If you want to see a list of all your active aliases type `alias`. To remove an alias use the command `unalias`, like `unalias llss`.

Aliases are not saved permanently if you do not explicitly tell the system to do so. Furthermore, they are valid only in the active shell. If you open a new shell, you have no access to your aliases. In order to save your alias you must place an entry into the file `.bashrc` in your home directory (use the editor `vi`). Since different systems behave differently, you are on the safe side by putting the same entry into the `.bash_profile` in your home directory (see Sect. A.2.1 on p. 426).

Much more powerful than aliases are shell scripts. With shell scripts you can also use parameters and perform much more sophisticated tasks. You will learn more about shell scripts in Chap. 10 on p. 125.

8.9 Pimping the Prompt

The shell prompt is everything preceding the cursor. Depending on the system's setting, it may contain your username and the computer name. Sometimes, the name of the current active folder is added, too. However, you are free to change the command line prompt—its behavior is stored in the shell variable `PS1`. Terminal 45 shows how to play around with the prompt.

```

_____ Terminal 45: Changing the Command Line Prompt _____
1  rw@vm-robbe:~$ echo $PS1
2  \[\e]0;\u@\h: \w\a\]$(debian_chroot:+($debian_chroot))\u@\h:\w$
3  rw@vm-robbe:~$ MYPS=$PS1
4  rw@vm-robbe:~$ PS1="$ "
5  $ PS1="Hello \u, the time is \t $ "
6  Hello rw, the time is 11:40:30 $ PS1="[\u|\d]\w $ "
7  [rw|Fri Sep 28]~ $ cd Documents/
8  [rw|Fri Sep 28]~/Documents $ PS1=$MYPS
9  rw@vm-robbe:~/Documents$ cd
10 rw@vm-robbe:~$

```

In line 1 we print the current value of `PS1` and store its value in variable `MYPS` in line 3. Then we play around a bit. Finally, in line 8, we set `PS1` back to its original value. As you probably noted, backslash-character combinations are interpreted as instructions. Here are some important ones:

\u	username
\h	computer (host) name
\t	current time
\d	current date
\w	path to working directory
\W	name of working directory
\$	dollar sign for normal user
#	hash character for the super user (root)

The following command brings some color to your life. It will show your username and current directory, both in different colors:

```
PS1="\[\033]0;\w\007\033[32m\]\u|@\h\[\033[33m\w\033[0m\]\n$_"
```

Do not forget the space character behind the dollar character.

You see, shell prompts can be very complicated and I do not want go into details here. Write this line into your personal Bash profile file `~/.bash_profile` in order to make the change permanent.

8.10 Batch Jobs

It frequently occurs that you have to do something with several files. Image you would like to edit several files in a way shown in Sect. 7.3.1 on p. 95. Then you can run a batch job on the batch job—so to say. Terminal 46 shows how.

```

1  $ mkdir Batch
2  $ cd Batch/
3  $ cp ../genomes.txt .
4  $ cp genomes.txt genomes-2.txt
5  $ cp genomes.txt genomes-3.txt
6  $ ls
7  genomes-2.txt  genomes-3.txt  genomes.txt
8  $ for i in *.txt; do vi -e $i < ../edit.vi; done
9  $ ls
10 genomes-2.new  genomes-3.new  genomes.new
11 genomes-2.txt  genomes-3.txt  genomes.txt
12 $

```

I assume that you have File 1 on p. 79 and File 2 on p. 95 in your current directory. Then, in line 1 in Terminal 46 we create a new directory called *Batch*. Next, we change to that directory and copy the file *genomes.txt* from the parent directory to the current directory. In lines 4–5 we create to copies of that file. Line 8 contains the magic command: for all files ending with *.txt*, execute (do) `vi -e $i < ../edit.vi`. Thus, we loop through all files and the current file is stored in the shell variable *i*. This saves a lot of typing!

Real-world use cases can be found in Sect. 9.6 on p. 122 and 10.10 on p. 150.

8.11 Scheduling Commands

Linux gives you the possibility to schedule commands or scripts. It works like an alarm clock. You set the time and date when the alarm should ring. Instead of ringing a bell, a command will be executed. One possible example is to instruct Linux to update a database every night. Or how about writing a script that removes all old backup files every evening?

When you start the Linux system, a number of background processes are started. You can see this when you follow the lines on the startup screen. These background system programs are called *daemons*. Among these daemons is *cron*. Cron checks up a system table for entries. These entries tell cron when to do what. Note: On most systems, you must get permission from the system administrator before you can submit job requests to cron. In order to fill the table with commands that need to be executed repeatedly (e.g. hourly, daily, or weekly), there is a program called *crontab*. The *crontab* command creates a crontab file containing commands and instructions for the cron daemon to execute. For editing the crontab file *crontab* uses the text editor *vi* (see Sect. 7.2.2 on p. 88).

You can use the *crontab* command with a number of options. The most important ones are:

```
crontab -e    Edit your crontab file. If not existent, create the crontab file.
crontab -l    Display your crontab file.
crontab -r    Remove your crontab file.
```

Each entry in a crontab file consists of six fields, specifying in the following order: minute, hour, day, month, weekday, and command(s). The fields are separated by spaces or tabs. The first five fields are integer patterns and the sixth is the command to execute. Table 8.2 briefly describes each of the fields.

Each of the values from the first five fields in Table 8.2 may be either an asterisk (*), meaning all legal values, or a list of elements separated by commas. An element is either a number or an inclusive range, indicated by two numbers separated by a dash (e.g. 10–12). You can specify days with two fields: day of the month and day of the

Table 8.2 Crontab file field entries

Field	Value	Description
minute	0–59	The exact minute when the command is to be executed
hour	0–23	The hour of the day when the command is to be executed
day	1–31	The day of the month when the command is to be executed
month	1–12	The month of the year when the command is to be executed
weekday	0–6	The day of the week when the command is to be executed (Sunday = 0, Monday = 1, Tuesday = 2 and so forth)
command		The complete sequence of commands to execute. Commands, executables (such as scripts), or combinations are acceptable

week. If you specify both of them as a list of elements, cron will observe both of them. For example

```
0 0 1,15 * 1 /mydir/myprogram
```

would run the program *myprogram* in the *mydir* directory at midnight on the 1st and 15th of each month and on every Monday. To specify days by only one field, the other field should be set to *. For example, with

```
0 0 * * 1 /mydir/myprogram
```

the program would run only on Mondays at midnight.

Crontab can also be used with lists, ranges and steps. A list is a set of numbers (or ranges) separated by commas. Examples are 1, 2, 5, 9 or 0-4, 8-12. Ranges of numbers are two numbers separated by a dash. The specified range is inclusive. For example, 8-11 for an hour entry specifies execution at hours 8, 9, 10, and 11. Step values can be used in conjunction with ranges. Following a range with /5 specifies 5 skips through the range (every fifth). For example, 0-23/2 can be used in the hour field to specify command execution every other hour. This is much less typing than 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22. Steps are also permitted after an asterisk, so if you want to execute a command every 2h, just use */2.

8.12 Wildcards

You do not have to be a programmer to appreciate the value of the shell's *wildcards*. These wildcards make it much easier to search for files or content. When the shell expands a wildcard, it is replaced with all the matching possibilities. Let us take a look at the most common wildcards:

*	Matches anything or nothing. “protein*” matches <i>protein</i> , <i>proteins</i> , <i>protein-sequence</i> and so on.
?	Matches any single character. For example, “seq?.txt” matches <i>seq1.txt</i> but not <i>seq.txt</i> .
[]	Matches any character or character range included. “[A-Z]*” matches anything starting with an uppercase character. “[sf]” matches the characters <i>s</i> or <i>f</i> .
[!]	Inverses the match of []. “[!a-zA-Z]” matches anything but letters.
{*.txt, *.doc}	One of the options in the list

These are the most important wildcards. Let us see how we can use them in conjunction with the command `ls`.

```
Terminal 47: Wildcards
1  $ ls
2  1file  cat      list.sh  new      seq2.txt  spaces.sh
3  AFILE  err.txt  list.txt seq1.txt  seq5.txt  end.txt
4  $ ls n*
5  new
6  $ ls seq[0-4].txt
7  seq1.txt seq2.txt
8  $ ls [A-Z]*
9  AFILE
10 $ ls *[^a-z]*
11 1file  err.txt  list.txt  seq2.txt  spaces.sh
12 AFILE  list.sh  seq1.txt  seq5.txt  end.txt
13 $
```

Terminal 47 shows you some examples of the application of wildcards in connection with the `ls` command. In line 1 we obtain an overview of all files present in the working directory. In line 4 we list only files that begin with the lowercase character `n`. In line 6 we restrict the output of the `ls` command to all files that begin with “seq”, followed by a number between 1 and 4, and ending with “.txt”. The `ls` command in line 8 resembles line 4; however, the files must begin with any uppercase character. Finally, in line 10, we list all files with filenames that contain a character other than a lowercase character.

Of course, you can use wildcards also with commands like `rm`, `cp`, or `mv`. However, it is always a good idea first to check your file selection with `ls`. You might have an error in your pattern, and thus erase the wrong files.

8.13 Processes

In Linux, a running program or command is called a process. Ultimately, everything that requires the function of the processor is a process. The shell itself is a program, too, and thus a process. Whenever one program starts another program, a new process is initiated. We are then talking about the *parent process* and the *child process*, respectively. In the case that a command is executed from the shell, the parent process is the shell, whereas the child process is the command you execute from the shell. That in turn means that when you execute a command, there are two processes running in parallel: the parent process and the child process. As we heard before in Chap. 3 on p. 21, this is exactly one strength of Linux. The capacity of an operating system to run processes in parallel is called *multitasking*. In fact, with Unix or Linux you can run dozens of processes in parallel. The operating system takes care of distributing the calculation capacity of the processor to the processes. This is like having a nice toy in a family with several children. There is only one toy (processor). The mother (operating system) has to take care that every kid (process) can play with the toy for

a certain time before it is passed to the next child. In a computer this “passing the toy” takes place within milliseconds.

As a consequence of multitasking, it appears to the user as if all processes run in parallel. Even though you might run only a few processes, the computer might run hundreds of processes in the background. Many processes are running that belong to the operating system itself. In addition, there might be other users logged into the system you are working on. In order to unambiguously name a process, they are numbered by the operating system. Each process has its own unique *process number*.

8.13.1 Checking Processes

To get an idea about the running processes you can use the command `ps`.

```

1  $ ps
2  PID TTY          TIME CMD
3  958 pts/0        00:00:00 bash
4  1072 pts/0        00:00:00 ps
5  $ ps -T
6  PID TTY          STAT TIME COMMAND
7  958 pts/0        S      0:00 -bash
8  1073 pts/0        R      0:00 ps -T
9  $

```

Terminal 48: Listing Processes

Terminal 48, lines 1–4, shows the output of the `ps` command (print process status). The first line is the header describing what the row entries mean:

PID	Process ID	The process ID (identification) is a unique number assigned to each process by the operating system.
TTY	Terminal	The terminal name, from which the process was started.
STAT	State	This entry indicates the state of the process. The process can either sleep (S), run on the processor (R), just be started (I), stopped (T) or be “zombie” (Z), that is ending.
TIME	Run Time	The run time corresponds to the amount of processor computing time the process has used.
CMD	Command	This is the name of the process (program or command name).

As usual, there are a lot more options possible. Quite useful is the option `-T`. It shows you all processes of the terminal in which you are currently working together with the process state. However, the most important information for you are the process ID and name. In the example in Terminal 48 there are two processes running. The

first process is the bash shell and the second process the command `ps` itself. They have the process IDs 958 and 1072, respectively. The output of `ps -T` is shown in lines 6–8. Of course, the shell is still running together with `ps -T`. The *process's state* information can be interesting to identify a stopped (halted) process.

8.13.2 Realtime Overview

Sometimes it is helpful to see how much CPU or memory resources an application requires. When you execute `top` in the command line, a live output will provide you with this inform information.

```

Terminal 49: Command top
1 top - 18:15:20 up 7:26, 1 user, load average: 0.68, 0.55, 0.34
2 Tasks: 143 total, 8 running, 135 sleeping, 0 stopped, 0 zombie
3 Cpu(s): 0.8%us, 2.0%sy, 0.0%ni, 97.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
4 Mem: 1025272k total, 894208k used, 131064k free, 70680k buffers
5 Swap: 1046524k total, 0k used, 1046524k free, 501316k cached
6
7 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8 1052 root 20 0 97504 52m 10m S 1.4 5.3 1:45.44 Xorg
9 2029 rw 20 0 95800 17m 11m R 1.0 1.7 0:42.28 gnome-terminal
10 1701 rw 20 0 265m 61m 31m S 0.6 6.1 1:41.49 unity-2d-shell
11 867 mysql 20 0 318m 32m 5832 S 0.2 3.3 0:39.35 mysqld
12 978 root 20 0 9436 972 632 S 0.2 0.1 0:39.21 VBoxService
13 1773 rw 20 0 20468 2140 1764 S 0.2 0.2 0:13.58 gvfs-afc-volume
14 2122 rw 30 10 151m 51m 34m R 0.2 5.2 0:23.55 update-manager
15 4056 rw 20 0 2836 1160 880 R 0.2 0.1 0:00.18 top
16 4060 root 20 0 0 0 0 S 0.2 0.0 0:00.01 kworker/0:1
17 1 root 20 0 3524 1976 1312 S 0.0 0.2 0:01.61 init
18 2 root 20 0 0 0 0 S 0.0 0.0 0:00.02 kthreadd
19 3 root 20 0 0 0 0 S 0.0 0.0 0:04.65 ksoftirqd/0
20 5 root 20 0 0 0 0 S 0.0 0.0 0:00.42 kworker/u:0
21 6 root RT 0 0 0 0 S 0.0 0.0 0:00.00 migration/0
22 7 root RT 0 0 0 0 S 0.0 0.0 0:02.64 watchdog/0
23 8 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 cpuset
24 9 root 0 -20 0 0 0 S 0.0 0.0 0:00.00 khelper

```

Terminal 49 shows an example how the output looks like. You can hit `o` to order the information by different aspects and `?` to obtain help. `q` quits the live output and brings you back to the command line.

8.13.3 Background Processes

There are two basic ways of running processes (jobs): either in the *foreground* or in the *background*. In foreground mode, the parent process (usually the shell) is suspended while the child (the command or script) process is running and taking over the keyboard and monitor. After the child process has terminated, the parent process resumes from where it was suspended. Take a look at Sect. 10.8 on p. 147 to see how you can get visual desktop notifications when a process has ended.

In background mode, the parent process continues to run using the keyboard and monitor as before, while the child process runs in the background. In this case, it is advisable that the child process gets all its input from and sends all its output to files, instead of the keyboard and monitor. Otherwise, it might lead to confusion with the parent process's input and output. When the child process terminates, either normally or by user intervention, the event has no effect on the parent process, although the user is informed by a message sent to the display.

To exemplify the concept of foreground and background processes we use the command `sleep`. It does nothing but halt processing for a specified number of seconds. The command `sleep 5` will halt for 5 s. An example would be the command `sort`. For long text files execution of `sort` takes very long. Thus, you might want to start it directly in the background. That is done as shown in Terminal 50.

```

Terminal 50: Background Program Execution
1  $ sleep 5
2  $ sleep 5 &
3  [1] 49795
4  $ date                                     # you could do something else
5  Thu Apr 12 13:28:29 CEST 2012
6  $
7  [1]+  Done                                sleep 5
8  $

```

Executing any command with the ampersand (&) will start that command in the background. The `sleep` command is started in the background as shown in line 2 of Terminal 50. You will then see a message giving the *background ID* ([1]) and the *process ID* (23001). Now you can continue to work in the shell (lines 4–6). When the background process has finished you will see a message as shown in line 7. Usually, you have to press **(Enter)** before you see this message. This can be changed for the current session by typing `set -b`. This causes the status of terminated background jobs to be reported immediately, rather than before printing the next primary prompt. You can unset this with `set +b`.

You may also wish to place a process already running in the foreground into the background and resume working in the shell. This is done by pressing **(Ctrl)+(Z)** and then typing `bg`. This brings the process into the background. In order to bring a background process back into the foreground, you have to type `fg processname` or `fg backgroundID`. You can again send it back into the background by stopping (not quitting) the process with **(Ctrl)+(Z)** and then typing `bg`. Note that you quit a process with **(Ctrl)+(C)**.

```

Terminal 51: Listing Background Processes
1  $ ps -x
2  PID TTY          STAT TIME COMMAND
3  1050 pts/1        S    0:00 -bash
4  23036 pts/1        R    0:00 ps -x
5  $ sleep 90 &
6  [1] 23037
7  $ ps -x
8  PID TTY          STAT TIME COMMAND
9  1050 pts/1        S    0:00 -bash

```

```

10 23037 pts/1    S      0:00 sleep 90
11 23038 pts/1    R      0:00 ps -x
12 $ fg sleep
13 sleep 90
14
15 [1]+  Stopped                  sleep 90
16 $ ps -x
17    PID TTY          STAT TIME COMMAND
18   1050 pts/1    S      0:00 -bash
19  23037 pts/1    T      0:00 sleep 90
20  23039 pts/1    R      0:00 ps -x
21 $ bg
22 [1]+ sleep 90 &
23 $

```

Again we play with `sleep 90`. This command just waits 90 s. Of course, you can use any other desired amount of seconds. Terminal 51 shows an example. In line 1 we list all active processes. In line 6, we execute `sleep 90` as a background process. In line 12 we bring `sleep` to the foreground with the command `fg`. Then, which is not visible in Terminal 51, we press `(Ctrl)+(Z)`, thereby pausing (stopping) the process. With the `bg` command we resume execution of `sleep` in line 21. You may actively terminate (kill) any process, provided you own it (you own any process that you initiate plus all descendants (children) of that process). You can also downgrade (but not upgrade!) the priority of any owned process.

8.13.4 Killing Processes

Sometimes a process, like a program, *hangs*. This means you cannot access it any more and it does not react to any key strokes. In that case you can *kill* the process. This is done with the command `kill PID`, where `PID` is the process ID you identify with `ps`. When you kill a process, you also kill all its child processes. In order to kill a process you must either own it or be the superuser of the system.

```

Terminal 52: Killing Processes
1  $ sleep 60 &
2  [1] 1009
3  $ ps
4    PID TTY          TIME CMD
5    958 pts/0    00:00:00 bash
6   1009 pts/0    00:00:00 sleep
7   1010 pts/0    00:00:00 ps
8  $ kill 1009
9  [1]+  Terminated  sleep 60
10 $ ps
11    PID TTY          TIME CMD
12    958 pts/0    00:00:00 bash
13   1011 pts/0    00:00:00 ps
14 $

```

Terminal 52 illustrates process killing. In line 1 we start the process `sleep 60`. With the `ps` command we find out that the process's ID is 1009. In line 8 we kill the

process with `kill 1009`. We got a message that the process has been killed. You might see that message only after hitting `(Enter)` once.

Some processes can be really hard to kill. If the normal `kill` command does not help, try it with option `-9`. Thus the command becomes `kill -9 PID`, with `PID` being the process ID. This should do the job.

Exercises

If you have read this chapter carefully, you should have no problems with the two little exercises...

8.1. Set `bash` as your default shell.

8.2. Print out a list of the active processes, sort the list by command names and save it into a file.

Chapter 9

Installing BLAST and ClustalW

In this chapter, you will learn how to install small programs. As examples, we are using BLAST (Altschul et al. 1990) (Basic Local Alignment Search Tool) and ClustalW (Thompson et al. 1994). BLAST is a powerful tool to find sequences in a database. Assume you have sequenced a gene and now want to check whether there are already similar genes sequenced by somebody else. Then you “blast” your sequence against an online database and get similar sequences, if present, as an output. Now assume you have found 10 similar genes. Of course, you would like to find regions of high similarity, that is, regions where these genes are conserved. For this, one uses ClustalW. ClustalW is a general-purpose multiple-sequence alignment program for DNA or protein sequences. It produces biologically meaningful multiple-sequence alignments of divergent sequences. ClustalW calculates the best match for the selected sequences and lines them up such that the identities, similarities, and differences can be seen. Then evolutionary relationships can be visualized by generating cladograms or phylograms.

Both programs use different installation procedures. BLAST comes as a packed and compressed archive that needs only to be unpacked. ClustalW comes as a packed and compressed archive, too. However, before you can run the program it needs to be compiled. This means you get the source code of the program and must create the executable files from it. I omit in this chapter the installation of BLAST+ and the use of modern (but black box) installation procedures. Both are explained in Sect. [4.1.2.5](#) at page 43.

9.1 Downloading the Programs via FTP

First of all you need to download the programs and save them in your home directory. How does this work? Well, of course, it is necessary that your computer is connected to the Internet! I assume that it is working for you.

Whenever you download a program you basically transfer data via the internet. This means you connect to a remote computer, copy a file to your computer, and finally disconnect from the remote computer. There is a special program for this: *FTP*. FTP is the user interface to the *Internet standard file transfer protocol*, which was especially designed for file transfer via the Internet. The program allows a user to transfer files to and from a remote network site. In fact, it is a very powerful program with many options. We will use only a small fraction of its capabilities. If you wish, take a look at the manual pages (`man ftp`).

9.1.1 Downloading BLAST

We will not download the newest version of BLAST, which is at the time of writing these lines version 2.2.26, but version 2.2.4. This version has less dependencies than the newer ones and should run on all typical Linux installations. Now, let us start. Go into your home directory by typing `cd` (remember: using the command `cd` without any directory name will automatically bring you to your home directory). Make a directory for BLAST by typing `mkdir blast`. Type `cd blast` to change into the *blast* directory. Then type `"ftp ftp.ncbi.nih.gov"`. With this, you tell Linux to connect to the file server named *ftp.ncbi.nih.gov* using the file transfer protocol (*ftp*). Enter *anonymous* as the username. Enter your *e-mail address* as the password. Type `bin` to set the binary transfer mode. Next, type `cd blast/executables/release/2.2.4/` to get to the right directory on the remote file server and then type `ls` to see a list of the available files. You should recognize a file named *blast-2.2.4-ia32-linux.tar.gz*. Since we are working on a Linux system, this is the file we will download. If, instead, you work on a Unix workstation or with Mac OS X Darwin, you should download *blast-2.2.4-ia32-solaris.tar.gz* or *blast-2.2.4-powerpc-macosx.tar.gz*, respectively. To download the required file type the command `get` followed by the name of the file for your system, e.g., `get blast-2.2.4-ia32-linux.tar.gz`. Finally, type `quit` to close the connection and stop the ftp program. The file *blast-2.2.4-ia32-linux.tar.gz* should now be in your working directory. Check it using the command `ls`.

```

1  $ ftp ftp.ncbi.nih.gov
2  Connected to ftp.ncbi.nih.gov.
3      Public data may be downloaded by logging in as
4      "anonymous" using your E-mail address as a password.
5  220 FTP Server ready.
6  Name (ftp.ncbi.nih.gov:rw): anonymous
7  331 Anonymous login ok,
8      send your complete email address as your password.
9  Password:
10 230 Anonymous access granted, restrictions apply.
11 Remote system type is UNIX.
12 Using binary mode to transfer files.
13 ftp> bin
14 200 Type set to I
15 ftp> cd blast/executables/release/2.2.4/
16 250 CWD command successful.
```

```

17 ftp> ls
18 500 EPSV not understood
19 227 Entering Passive Mode (130,14,29,30,197,39).
20 150 Opening ASCII mode data connection for file list
21 -r--r--r-- 1 ftp anonymous 15646697 Aug 29 2002
22 blast-2.2.4-ia32-linux.tar.gz
23 ...
24 -r--r--r-- 1 ftp anonymous 41198447 Aug 29 2002
25 blast-2.2.4-powerpc-macosx.tar.gz
26 226 Transfer complete.
27 ftp> get blast-2.2.4-ia32-linux.tar.gz
28 local: blast-2.2.4-ia32-linux.tar.gz
29 remote: blast-2.2.4-ia32-linux.tar.gz
30 227 Entering Passive Mode (130,14,29,30,196,216).
31 150 Opening BINARY mode data connection for
32 blast-2.2.4-ia32-linux.tar.gz (15646697 bytes)
33 226 Transfer complete.
34 15646697 bytes received in 02:53 (87.96 KB/s)
35 ftp> quit
36 221 Goodbye.
37 $

```

Terminal 53 gives an in-detail description of the `ftp` commands necessary in order to download BLAST. Furthermore, you see the feedback from the remote computer [please note that the output had to be truncated (indicated by ". . .") here and there for printing reasons]. In line 1, we apply the command to connect to the remote FTP server called *ftp.ncbi.nih.gov*. In line 6, we enter the login name *anonymous* and in line 9 the password, which is your email address. With the command `bin` in line 13, we change to the binary transmission mode (which is not absolutely necessary since FTP usually recognizes the file type automatically). Then we change to the directory that contains the executables (line 15) and list its content (line 17). Among others you will recognize a file called *blast-2.2.4-ia32-linux.tar.gz*, which we download with the command listed in line 27. Finally, we quit the session in line 35. The installation process is described in Sect. 9.2 on the following page.

9.1.2 Downloading ClustalW

In contrast to BLAST, ClustalW does not come as a binary file, which can be directly executed after decompressing. Instead, you get the source code and have to compile the source code for your Linux system; but first let us download the source code at <ftp://ftp.ebi.ac.uk/pub/software/unix/clustalw>. The file we need to download is *clustalw1.83.UNIX.tar.gz*. Download the file into a directory called *clustal* (`mkdir clustal` and `cd clustal`).

```

Terminal 54: Downloading ClustalW
1 $ ftp ftp.ebi.ac.uk
2 Connected to alpha4.ebi.ac.uk.
3 ...
4 Name (ftp.ebi.ac.uk:rw): anonymous
5 331 Guest login ok,
6 send your complete e-mail address as password.
7 Password:
8 230-Welcome anonymous@134.95.189.5

```

```

9  ...
10 230 Guest login ok, access restrictions apply.
11 Remote system type is UNIX.
12 Using binary mode to transfer files.
13 ftp> cd pub/software/unix/clustalw
14 ...
15 ftp> get clustalw1.83.UNIX.tar.gz
16 local: clustalw1.83.UNIX.tar.gz
17 remote: clustalw1.83.UNIX.tar.gz
18 500 'EPSV': command not understood.
19 227 Entering Passive Mode (193,62,196,103,227,137)
20 150 Opening BINARY mode data connection for
21   clustalw1.83.UNIX.tar.gz (166863 bytes).
22 100% |*****| 162 KB 151.41 KB/s 00:00 ETA
23 226 Transfer complete.
24 166863 bytes received in 00:01 (148.60 KB/s)
25 ftp> pwd
26 257 "/pub/software/unix/clustalw" is current directory.
27 ftp> !pwd
28 /home/emboss/Freddy/clustal
29 ftp> !ls
30 clustalw1.83.UNIX.tar.gz
31 ftp> quit
32 221-You have transferred 166863 bytes in 1 files.
33 ...
34 221 Goodbye.
35 $

```

Terminal 54 shows details about the FTP session. In addition to what we did in Terminal 53, we play a little bit with shell commands. In line 25 we use the command `pwd` to print the working directory of the remote computer. If we wish to execute a shell command on our local machine we need to precede it with the exclamation mark (!). Thus, in line 27 we print the working directory of our computer. We then check its content with `!ls` and can confirm that the file *clustalw1.83.UNIX.tar.gz* has been downloaded successfully. The installation process itself will be explained in Sect. 9.4 on p. 120.

9.2 Installing BLAST

Be sure you are in the directory *blast* and that you correctly downloaded the file *blast-2.2.4-ia32-linux.tar.gz* into it as described in Sect. 9.1.1 on p. 116. Now, type

```
gunzip blast-2.2.4-ia32-linux.tar.gz
```

Next, type

```
tar -xvf blast-2.2.4-ia32-linux.tar
```

This will extract the archive into the current working directory (see Sect. 5.6 on p. 62). You are done! BLAST is distributed with ready-to-go executable files. You can run the program directly. To run, for example, the program `blastall` you just type in the current directory `./blastall`. Why do you have to precede the command


```

38 | Sbjct: 24 cgatcgatcgat 13
39 | $

```

In line 1 of Terminal 55 we create an empty file called *testdb*. After you have entered the command `cat >testdb` and hit **(Enter)** you end up in an empty line. Everything you enter now will be redirected into the file *testdb*. However, you can correct errors with the **(BkSp)** key. When you hit **(Enter)**, a line break will be generated. You finish by pressing **(Ctrl)+(D)** at the end of line 5. Next, you create the text file *testblast* which contains the query sequence you want to “blast” against the database (i.e., you search similar sequences in the database). In line 9 we generate files which are used by BLAST and which are specific for our database file *testdb*. This is done with the command

```
./formatdb -i testdb -p F
```

The first option (`-i testdb`) identifies the input database file, here *testdb*, which we created earlier. The second option (`-p F`) tells BLAST that the database contains nucleic acid and not protein sequences. Remember that you have to precede the command `formatdb` with `./` because it is not a system command and not yet in the system command path. Let us see what kind of files have been generated. All new files created by `formatdb` start with the database filename we provided (*testdb*). This means we can specifically search for new files using `ls testdb*` as shown in line 10. We should see 3 new files.

Now the interesting part starts! We query the database for our query sequence, which is in the file called *testblast*. Therefore, we apply the command

```
./blastall -p blastn -d testdb -i testblast
```

The first option (`-p blastn`) defines the query program. With `blastn` we choose the normal program to query nucleotide sequences. Furthermore, we specify the name of the database (*testdb*) and the name of the file containing the query sequence (*testblast*). (You could also enter several sequences into the query file. They would be processed one after the other. It is, however, important that you use the *FASTA format* for the file.) The output starts in line 13. Note that, for printing reasons, the output in Terminal 55 is shorter than in reality. The sequence was actually found three times in sequence 1.

9.4 Installing ClustalW

After we have successfully downloaded the compressed source file for ClustalW, we need to uncompress and compile it. From the file extension *.tar.gz* we see that the compression program `gzip` was used (see Sect. 5.7 no p. 65) and that the file is an archive (see Sect. 5.6 on p. 62). One way to unzip the files and extract the archive is

```
gunzip clustalw1.83.UNIX.tar.gz
```

followed by

```
tar -xf clustalw1.83.UNIX.tar
```

A new directory named *clustalw1.83* will be created. Change into this directory (`cd clustalw1.83`). Now we start the compilation process: just type `make`. You will see a number of lines of the type `cc -c -O interface.c` popping up. Finally, you get back to your shell prompt. When you now list (`ls`) the directory's content, you will recognize a number of new files with the file extension `.o` and one new executable file called `clustalw`. That is the program compiled for your system! How do you recognize an executable file? Take a look back into the section on file attributes (see Sect. 5.3 on p. 56).

9.5 Running ClustalW

After you have successfully compiled ClustalW we should check whether it runs or not. We will align three tRNA genes. These are pretty short genes; but still we save ourselves from typing and restrict the sequences to only 25 nucleotides as shown in File 3.

```

File 3: tRNA.fasta
1  >gene1
2  GGGCTCATAGCTCAGCGGTAGAGTG
3  >gene2
4  GGGCCCTTAGCTCAGCTGGGAGAGA
5  >gene3
6  GGGGGCGTAGCTCAGCTGGGAGAGA

```

In line 1 of Terminal 56 we actually start the program ClustalW with

```
./clustalw tRNA.fasta
```

We get a lot of output on the screen, which is omitted here.

```

Terminal 56: Running ClustalW
1  $ ./clustalw tRNA.fasta
2  ...
3  $ cat tRNA.aln
4  CLUSTAL W (1.83) multiple sequence alignment
5
6  gene2      GGGCCCTTAGCTCAGCTGGGAGAGA-
7  gene3      GGGGGCGTAGCTCAGCTGGGAGAGA-
8  gene1      GGGCTCATAGCTCAGC-GGTAGAGTG
9              ***  * ***** ** *****
10 $

```

The alignment is not printed out onto the screen but written into the file *tRNA.aln*. We can take a look at the alignment using `cat tRNA.aln` (see line 3). There are

many more options for ClustalW available. Moreover, in addition to the alignment, a phylogenetic tree is created; but this is not our focus here.

9.6 A Wrapper for ClustalW

If you need to run ClustalW on several or several hundred multi-fasta files, then it becomes convenient to run batch jobs as already shown in Sect. 8.10 on p. 106. For the following example, I assume that you have File 3 on the preceding page in your current directory.

```

1  $ cp tRNA.fasta t-rna-1.fasta
2  $ cp tRNA.fasta t-rna-2.fasta
3  $ cp tRNA.fasta t-rna-3.fasta
4  $ vi t-rna-2.fasta
5  $ vi t-rna-3.fasta
6  $ for i in *.fasta; do clustalw $i; done
7  CLUSTAL 2.1 Multiple Sequence Alignments
8  ...
9  $ head t-rna-?.aln
10 ==> t-rna-1.aln <==
11 CLUSTAL 2.1 multiple sequence alignment
12
13 gene2      GGGCCCTTAGCTCAGCTGGGAGAGA-
14 gene3      GGGGGCGTAGCTCAGCTGGGAGAGA-
15 gene1      GGGCTCATAGCTCAGC-GGTAGAGTG
16            ***  *  *****  **  *****
17
18 ==> t-rna-2.aln <==
19 CLUSTAL 2.1 multiple sequence alignment
20
21 gene2      GGGCCCTTAGCTCTAGCTAGCTGGGAGAGA-----
22 gene3      GGGGGCGTAGCTC-----AGCTGGGAGAGA-----
23 gene1      GGGCAGCTAGCTATCA-TAGCTCAGCGGTAGAGTG
24            ***      *****      *****  *  *  *
25
26 ==> t-rna-3.aln <==
27 CLUSTAL 2.1 multiple sequence alignment
28
29 gene2      -GGGCACCTCTAGCATCAGGCTTGGGAGAGA-
30 gene3      TGGGGGGCGTAAGCTCCAG-CTTGGGAGAGA-
31 gene1      ---GGGCTCATAGCTCAGCAGGTCAGAGGTGT
32            *          ***          *  *****
33 $

```

In lines 1–3 we create three copies of *tRNA.fasta* and in lines 4–5 we open two of them and add nucleotide changes (just imitate an ape hacking the keyboard). Then, in line 6, we initiate the batch job. For all filenames ending with *.fasta*, execute `clustalw $i`, where *\$i* is the shell variable pointing to the current file in the loop. Finally, we apply the `head` command in line 9 to print the first ten lines of all three generated alignment files.

Exercises

In these exercises, you will download and compile the program `tacg` (Mangalam 2002). In order to do so, you require to be connected to the internet. If you succeed, you can already consider yourself as a little Linux freak. `tacg` is a command-line program that performs many of the common routines in pattern matching in biological strings. It was originally designed for restriction enzyme analysis, while this still forms a core of the program, it has been expanded to fulfill more functions. `tacg` searches a DNA sequence read from the command line or a file for matches based on descriptions stored in a database of patterns. These descriptions can, e.g., be formatted as explicit sequences, matrix descriptions, or regular expressions describing restriction enzyme cutting sites, transcription factor binding sites, or whatever. The query result is sent to the standard output (the screen) or a file. With `tacg` you can also translate DNA sequences to protein sequences or search for open reading frames in any frame. It is approximately 5–50 times faster than the comparable routines in Emboss (see Sect. 3.10.1.1 on p. 33). For more details you should take a look at (tacg.sourceforge.net).

9.1. Connect to the FTP server *ftp.sunet.se*, login anonymously and change to the following directory: *pub/molbio/restrict-enz/*. List the directory content and download the file *tacg-3.50-src.tar.gz* into your home directory. Log off and move the file into a directory called *dna_grep*.

9.2. Unzip and extract the archive contained in *tacg-3.50-src.tar.gz*.

9.3. In this exercise you perform the compilation step. Follow the instructions given in the file *dna_grep/tacg-3.50-src/INSTALL*. Check if you were successful by typing `./tacg`.

9.4. If you wish, you can take a look at the documentation in the subfolder *Docs* and play around a little.

Chapter 10

Shell Programming

With this chapter we enter a new world. Until now you have learned the basics of Linux. You have learned how to work with files and create and edit text files. Now we are going to *use* Linux. We take advantage of its power. Up to now everything was quite uncomfortable and I guess that you thought occasionally: “Okay, it is free—but damn uncomfortable!” But now you are a pro! You know what it is about, how to work on the system; now it is time to take advantage of it, squeeze it out, form it, make it working for you, harvest the fruits of learning—by more learning. In this section you start programming! If you thought programming is something for the freaks—forget it. Everybody can do it, but you must like solving problems. Hey, you are a scientist! That is your profession! And it is creative. It is like an art—you will see! It is like solving crosswords: you need to take your time and passion and probably need to look up one or the other thing; but finally you solved it. Throughout the last chapters we used the shell intensively. Now it is time to take a closer look at its functions.

10.1 Script Files

When we write a shell program we will save the program code in a file. We call this file a *script file* and the program itself a *script*. Script files should be named with relatively short names that describe their function. Otherwise you will end up in chaos. It is more than a good idea to give script files the extension *.sh*. This is crucial to keep something called recursion from occurring. Although recursion is a powerful and often used programming technique, unintended recursions may lead to a kind of futile cycle, something that repeatedly executes itself. For example, consider the filename *date* for a script file. This sounds like a clever name for a file that does something with dates. However, as you know, *date* is also a command. If you now happen to call the command *date* from your script called *date* you start a futile cycle. Every time the shell interprets the line in the script file containing *date* a new instance of your script will be started. Now your script is running twice; but the

creation of new instances goes on and on until your system is completely busy with *date* and gives an error message. Most probably you have to restart your system (or the system administrator will have to do this). Always use file extensions for script files.

What does the inside of a script file look like? Well, it is simply a file containing commands to be executed. In this section, we will write shell script files; however, later we will write *sed*, *awk*, and *perl* script files. Remember: *sed*, *awk*, and *perl* are script languages.

In the first line of a shell script file you identify the program that is to interpret the script. For example, in order to invoke the *bash* shell to interpret the script, the first line would look like `#!/bin/bash`. This line begins with the so-called *shebang* “`#!`” (from *sharp* and *bang*). In the next line the script follows. You can and should make use of the possibility to comment your program. Otherwise, you will soon forget what your script does and why. Ergo: always use comments! Comments begin with a hash (`#`). Any text written behind the hash is ignored by the shell.

Okay, let us write our first script. Script files must be executable. I guess this is clear?! A script is a program—and programs have to be executed. Our first script (Program 2) will convert all files ending with *.sh* in the home directory and all subdirectories into executable files.

```

_____ Program 2: con-sh-exe.sh - Creating Executable Files _____
1  #!/bin/bash
2  # save as con-sh-exe.sh
3  # This script converts .sh files to executables
4  echo "Search for *.sh files"
5  find ~ -name "*.sh"
6  echo "Perform task"
7  find ~ -name "*.sh" -exec chmod u+x {} \;
8  echo "Ready:"
9  find ~ -name "*.sh" -exec ls -l {} \;

```

Using the text editor *vi*, enter Program 2 and save it in a file called *con-sh-exe.sh*. Then make the file executable with

```
chmod u+x con-sh-exe.sh
```

In order to run the script you must type

```
./con-sh-exe.sh
```

Now, let us go step by step through the program. Line 1 instructs the *bash* shell to execute all the commands. In principle, we could also instruct another shell or any other command interpreter like *awk* or *perl*. Lines 2 and 3 contain a comment on the program. As said before, lines beginning with a hash (`#`) are ignored by the command interpreter. The command *echo* in lines 4, 6, and 8 prints out a message to the standard output (*stdout*), i.e., the screen. The message is placed between quotation marks (“”). You could also use *echo* without anything. That would print a blank line. In line 5 the script runs the command *find*. It searches in the home

directory (the shortcut is `~`, you could also write `/home/Freddy`) for files having the extension `.sh`. Note that the filename is enclosed in quotation marks. The output of the command is printed, as usual, to the *stdout*. There is no difference whether you run the command from the command line or a script. In line 7 the command `find` is used again. Here the option `-exec` prevents the output to *stdout*. Instead, the files found are directed to the program executed by the option `-exec`. The files are provided in the form of empty curled brackets (`{ }`). Here, the program executed is `chmod` with the option `u+x` (see Sect. 5.5 on p. 59). As said, the curled brackets represent the files found by `find` and for which the permission is to be changed by `chmod`. Note that the command is followed by “\;”. This is obligatory. In line 9, the `find` command is used to list all the files ending with `.sh`. This is done in order to visualize that the permissions are set correctly. If it comes down to the main task: change the permission of all files ending with `.sh` to executable; the script could be minimized to lines 1 and 7. All the rest is luxury but helps to understand what is going on.

10.2 Modifying the Path

In the example above we executed the script with `./con-sh-exe.sh`. We always have to supply the path to the script. If we entered `con-sh-exe.sh` we would receive the error message: “command not found”. However, we could add the directory where we save scripts (let us say we save them in *scripts*) and other executables to the system path. This is a variable that contains all directories that might contain commands. To see the actual state of the path variable enter `echo $PATH`.

```
Terminal 58: The Path Variable
1 $ echo $PATH
2 /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
3 $
```

As shown in Terminal 58, the system checks by default the directories `/usr/local/bin`, `/bin`, `/usr/bin`, and `usr/X11R6/bin` for executable commands. If you wish to add the directory *scripts* to the path, you have to extend the content of the *PATH* variable. This can be done in the file `.bash_profile` or `.profile` (see Sect. A.2.1 on p. 426). Open one of the files with the editor `vi` and add the following line:

```
PATH=$PATH:$HOME/scripts
```

The next time you start the bash shell you can execute scripts sitting in the directory *scripts* directly by entering their name. Alternatively, you can directly activate the new path by typing

```
. $HOME/.bash_profile
```

Another option is to write a script that adds the current working directory to the *PATH* variable. Program 3 shows such a script.

```

1  #!/bin/bash
2  # save as add2path.sh
3  # add the currently active directory to the path variable
4  # assignment only active in the current session
5  PATH=$PATH: "$(pwd)"

```

Program 3: *add2path.sh* - Extending the Path Variable

This is a pretty short script. It actually assigns to the system variable *PATH* its original value, plus a colon (:), and plus the directory you are currently in (`$(pwd)`). Note that `$( )` returns the result of the command enclosed in the parentheses, here, `pwd`.

10.3 Variables

A variable is a symbol that stands for some value—an abstraction for something more concrete. The shell’s ordinary variables are very simple. A variable comes into existence when a value is assigned to it. Its value is what programmers call a *string*: simple text (it can be composed of numeric digits that would represent a number to the human observer or to some other program, but the shell itself is blind to their numeric value).

The assignment of a variable is a matter of only one line:

```
plant=orchid
```

There must be no spaces around the equal character. With this simple command the value “orchid” is assigned to the variable named *plant*. If you wish to recall the content saved in *plant*, you call it with the `echo` command:

```
echo $plant
```

Note that you have to precede the variable name with the dollar character (\$).

```

1  $ plant=orchid
2  $ echo $plant
3  orchid
4  $ orchid=parasite
5  $ plant=$orchid
6  $ echo $plant
7  parasite
8  $ echo "The value of \${orchid} is ${orchid}"
9  The value of ${orchid} is parasite
10 $ echo "$animal"
11
12 $

```

Terminal 59: Shell Variables

Terminal 59 explains how to assign variables. In line 1 we create the variable *plant* and assign the value (content) “orchid” to it. With `echo $plant` we display the content of the variable. In line 4 we save the value “parasite” in the variable *orchid*. Then we assign to the variable *plant* the content of the variable *orchid*. Note that we overwrite the old content of *plant*. As you see in line 5, you can indirectly assign a value to a variable. Finally, in line 8 we use the `echo` command to print some text and the value of the variable *orchid*. Note that you must use the backslash (escape character) in order to print the special character `$`. Otherwise, the dollar character would indicate to the shell that the value of a variable should be printed. In line 10 we recall the content of the variable *animal*. This variable has not been created yet and thus is empty. After a variable has been declared, it is available only in the active shell and only during the active session. If you want to export a variable to other sessions you need to apply the command `export`. This is shown in Terminal 60.

```

Terminal 60: Export Shell Variables
1  $ plant=rose
2  $ sh
3  sh$ echo $plant
4
5  sh$ exit
6  exit
7  $ export plant
8  $ sh
9  sh$ echo $plant
10 rose
11 sh$ exit
12 exit
13 $

```

In line 1 in Terminal 60 we create the variable *plant* and assign the value “rose” to it. Then we change into the Bourne shell with the command `sh`. Here, as depicted in line 3, we recall the value of *plant*. It is empty. Now we exit the Bourne shell, export the variable *plant* with `export plant` and go back into the Bourne shell.

Apart from the variables we create, there are a number of system variables we can use. These variables are also called *environmental variables* or *shell variables*. Environmental variables are available to the whole system and all shells. Shell variables are available only to the current shell. In Terminal 60 we exported a shell variable to the environment. With the command `env` (environment) you can list all available environmental variables. With the `set` command you can list all shell variables. Some interesting environmental variables are shown in Table 10.1.

Table 10.1 Some commonly used environment variables

Variable	Content	Value
<i>SHELL</i>	Active shell	“/bin/bash”
<i>USER</i>	The user	“Freddy”
<i>PATH</i>	Search path	“/bin:/usr/bin:/home/Freddy/scripts”
<i>PWD</i>	Active directory	“/home/Freddy/scripts”
<i>HOME</i>	Home directory	“/home/Freddy”

The list shows the variable name (note that system variables are by convention uppercase), its meaning and an example of its content.

In order to delete a variable, the command `unset` can be used. Then the variable is totally gone and not just empty.

```
Terminal 61: Exporting Shell Variables
1  $ my_var=value
2  $ set | grep my_var
3  my_var=value
4  $ env | grep my_var
5  $ export my_var
6  $ env | grep my_var
7  my_var=value
8  $ unset my_var
9  $ env | grep my_var
10 $ set | grep my_var
11 $
```

The example shown in Terminal 61 shows how you can easily search for a variable in the shell or environment with the help of `grep`. In line 8 the variable `my_var` is removed from the system.

10.4 Input and Output

Of course, a script should be able to produce output and get some input. The shell scripting language provides different possibilities to do so. With the `echo` command you can display messages. With the commands `read` and `line` you can ask the user to provide some input. Finally, shell scripts can be directly fed with input when called from the command line—these are script parameters. In this section we will see examples of all of these.

10.4.1 *echo*

As we saw before, the `echo` command prints out text or the value of variables onto the screen. You should always enclose the text in double quotes, as shown in Terminal 59 on p. 128 line 8. `echo` offers the interesting option `-e`, which instructs `echo` to evaluate the character behind a backslash:

<code>\\</code>	Backslash	<code>\a</code>	Alert, System Bell	<code>\b</code>	Backspace
<code>\n</code>	New Line	<code>\t</code>	Tabulator		

When you type `echo -e "\a"` you will hear the system bell. This function is quite nice to inform the user that the execution of a script has been finished. In a similar

manner you can introduce tabulators or a new line. Thus, it is possible to format the output a little bit.

10.4.2 Here Documents: <<

Assume you want to write a couple of help lines in your script. Of course, you could achieve this with the `echo` command. However, it looks quite messy. For such cases the shell offers the `<<` operator, called *here document*. The operator is followed by any arbitrary string. All following text is regarded as coming from the standard input, until the arbitrary string appears a second time.

```

----- Program 4: text.sh - Printing Text -----
1  #!/bin/bash
2  # save as text.sh
3  # printing text
4  cat <<%%
5  Here comes a lot of text.
6  My home directory is $HOME - cool.
7  Let us use some tabulators:
8      tab1
9      tab2
10 This looks like a nice list.
11
12 %%
13
14 cat <<\%%
15 Here comes a lot of text.
16 My home directory is $HOME - cool.
17 Let us use some tabulators:
18     tab1
19     tab2
20 This looks like a nice list.
21 %%

```

Program 4 illustrates the use of `<<`. In line 4 `%%` has been used as *text range indicator*. The `cat` command gets all the text up to the second occurrence of `%%` in line 12. From line 14 to 21 we use the same construction. However, the operator `<<` is now preceded by a backslash. Therefore, the variable `HOME` will not be expanded but printed as it is: “\$HOME”. Terminal 62 shows the execution and output of the script.

```

----- Terminal 62: Execution of text.sh -----
1  $ chmod u+x text.sh
2  $ ./text.sh
3  Here comes a lot of text.
4  My home directory is /Users/rw - cool.
5  Let us use some tabulators:
6      tab1
7      tab2
8  This looks like a nice list.
9
10 Here comes a lot of text.
11 My home directory is $HOME - cool.
12 Let us use some tabulators:

```



```

13         tab1
14         tab2
15 This looks like a nice list.
16 $

```

10.4.3 *read and line*

Two commands are available to get interactive input into your shell script. The most commonly used command is `read`. On some very old systems the `read` command is not available. Then you have to use `line` instead.

Usually you would first ask the user for the required input. This can easily be done with `echo`. Then you apply the command `read` in order to save the user's input in a variable.

```

----- Program 5: chg-pwd.sh - Changing $PWD -----
1  #!/bin/bash
2  # save as chg-pwd.sh
3  # changed the environmental variable PWD
4  echo
5  echo "-----"
6  echo "PWD is currently set as $PWD"
7  echo
8  echo "Enter a new path and press ENTER"
9  echo "or enter nothing and press ENTER"
10 echo "to leave \"$PWD\" unchanged"
11 read new
12 PWD=${new:-$PWD}
13 echo "PWD is now set to $PWD"
14 echo "-----"
15 echo -e "\a"

```

Program 5 can be considered user-friendly. It uses some `echo` commands in order to give information and format the output. In line 11 the program stops and continues only after the `(Enter)` key has been pressed. Thus, before you hit `(Enter)` you can enter a new path that is to be assigned to `PATH`. The input is saved in the variable `new`. In line 12 it is checked whether the user provided some input or not. If the user only hits the `(Enter)` key, then `new` will be empty. In that case, the old value of `PWD` is saved in `PWD`. However, if `new` is a non-empty variable, the value of `new` will be assigned to `PWD` (see Sect. 10.5.1 on p. 134). In line 15 the script invokes a system beep (see Sect. 10.4.1 on p. 130). Keep in mind that variables are assigned only for the active shell. When you start a script, it runs in its own shell. Thus, after termination of `chg-pwd.sh` `PWD` is unchanged (check it with `echo $PWD`). In order to change `PWD` in the current shell you must start the script with

```
. ./chg-pwd.sh
```

The dot stands for the shell command `source`. In fact, you could also run the script with `source chg-pwd.sh`. The `source` command prevents the script from running in a new sub shell.

10.4.4 Script Parameters

What make scripts very powerful when compared to *aliases* is the possibility to provide parameters. When a shell script is invoked, it receives a list of parameters (also called arguments) from the command line that invoked it. These parameters can then be used to direct the execution of the script. Let us assume you have DNA sequences saved in many individual files. Now, you want to display all files in your home directory and all subdirectories that contain a certain DNA sequence. The shell command would be

```
find -type f -name "*.dna" -exec grep TATAAT {} \;
```

assuming that you wish to query the sequence *TATAAT* in files ending with *.dna*. It is quite a pain always to enter this line. You could write an *alias* for this command; but do you always query for the same sequence? Certainly not! Furthermore, it would be nice to search in other files, too. For example, it is common to save amino acid sequences in files ending with *.aa*. A good option is to write a shell script! Take a look at Program 6.

```
----- Program 6: grep_file_seq.sh - Finding Sequences -----
1 #!/bin/bash -x
2 # save as grep_file_seq.sh
3 # search for a pattern (par2) in *. (par1) files
4 # query home directory and all subdirectories
5 find $HOME -type f -name ".*$1" -exec grep "$2" {} \;
```

Program 6 is quite straightforward. You call the script with

```
./grep_file_seq.sh dna TATAAT
```

This means you first enter the script name and then provide two parameters, *dna* and *TATAAT*, respectively. These parameters are internally saved in the variables *\$1* and *\$2*, respectively. Line 1 of Program 6 indicates that it should be interpreted by the bash shell. We also use the option *-x*. This is a good option for debugging purposes, that is: testing the program. With the option *-x* all executed commands will be displayed (see Sect. 10.9 on p. 148). The next 3 lines contain comments. They help us to remember what the script is about. Line 5 contains the command itself. Here, you see the use of the system variable *HOME* and the parameters *\$1* and *\$2*, respectively. They are used by script as they would be typed in. You can use up to 9 parameters (some systems allow for an unlimited number). The content of these parameters is explained in Table 10.2.

In Program 2 on p. 126 we changed the permission of all files in the home directory and all subdirectories having the extension **.sh*. Now, let us write a script that executes the same task, but only in the currently active directory and only with files we specify. The corresponding script is shown in Program 7.

Table 10.2 Variables to access shell parameters

Variable	Value
<code>\$1 – \$9</code>	The first 9 parameters
<code>\$#</code>	The total number of parameters
<code>\$*</code>	All the parameters
<code>\$0</code>	The name of the script itself

```

_____ Program 7: chmod_files.sh - Make Files Executable _____
1  #!/bin/bash
2  # save as chmod_files.sh
3  # adds execution permission for the user
4  chmod u+x $*

```

Program 7 gives you an example of how one can use the variable containing all parameters (`$*`) in order to supply a command with all parameters at once.

10.5 Substitutions and Expansions

The main job of the shell is to translate a string of characters given by the user (or as a line of a shell script) into a sequence of tokens for execution. These substitution or expansion processes appear in the following order: brace expansion, tilde expansion, parameter expansion, variable substitution, command substitution, arithmetic substitution, and pathname expansion. Let us define a word as a sequence of characters without spaces. Brace expansion is the process of expanding every word contained in a brace expression `{w1, w2, w3}` into a sequence of words where the brace expression is replaced by `w1, w2, w3`, respectively. For example, `abc{de,fg,hi}` expands into the sequence of words `abcde abcfg abchi`. Tilde expansion expands every `~` into the path to your home directory. Variable substitution and parameter expansion substitute the value of variables. Variable substitution replaces every expression containing `var` by the value of variable `$var`. Parameter expansion is a kind of conditional substitution. There are many variations of parameter expansion. Command substitution replaces every expression of the form `$(cmd)`, where `cmd` is a command, by the output of the execution of `cmd`. Hence, `$(pwd)` is replaced by the path to the active working directory.

This all sounds probably pretty complicated. Things will become clearer when you see the examples in the following sections.

10.5.1 Variable Substitution

A very useful feature of the shell is variable substitution or variable expansion. Expansion or substitution means that you assign a construction around a variable

and this construction will be resolved. We already used variable expansion when we worked with shell script parameters in Sect. 10.4.4 on p. 133 or recalled the value of a variable in Terminal 59 on p. 128. Variable expansion is initiated by `${variable}`. Here, *variable* stands for the variable name to be expanded. In the list below, *word* affects in the one or the other way variable expansion.

<code>\${var:-word}</code>	If the variable <i>var</i> is empty, the result of this construction is <i>word</i> . Otherwise, the value of <i>var</i> is the output. <i>var</i> remains unchanged.
<code>\${var:=word}</code>	As above. However, if the variable <i>var</i> is empty, <i>word</i> will be assigned to it. Thus, the value of <i>var</i> might change.
<code>\${var:+word}</code>	If <i>var</i> is empty, nothing happens and the output is empty. Otherwise, the output is <i>word</i> . <i>var</i> itself remains unchanged!
<code>\${#var}</code>	Provides the number of characters of the variable <i>var</i> . In case <i>var</i> is empty, the output is 0.
<code>\${var:offset:length}</code>	Expands up to <i>length</i> characters of <i>var</i> , starting at <i>offset</i> (the first character is 0). If <i>length</i> is omitted, this construction expands from <i>offset</i> to the end of the value of <i>var</i> .

There are many more possibilities, which you might want to look up in the manual pages. Let us take a look at some examples in the following Terminal.

```

1  $ enzyme="hydrogenase specific endopeptidase"
2  $ echo $enzyme
3  hydrogenase specific endopeptidase
4  $ echo ${enzyme:=text}
5  hydrogenase specific endopeptidase
6  $ echo ${enzyme:+text}
7  text
8  $ echo $gene
9
10 $ echo ${gene:-gene not discovered}
11 gene not discovered
12 $ echo $gene
13
14 $ echo ${gene:=ATG...TAA}
15 ATG...TAA
16 $ echo $gene
17 ATG...TAA
18 $ echo ${#gene}
19 9
20 $ echo ${gene:3:3}
21 ...
22 $ echo ${gene:3}
23 ...TAA
24 $

```

The examples given in Terminal 63 should give you an insight into functional aspects of variable substitution. In line 1 we assign the value “hydrogenase specific endopeptidase” to the variable named *enzyme*. In line 4, the value of the variable *enzyme* would be replaced by the text string “text”, if *enzyme* were empty, which is not the case. The value of *enzyme* is returned and displayed in line 5. In line 6, the content of the variable *enzyme* is replaced because it is not empty. In line 10 of Terminal 63 the text string “gene not discovered” is returned. However, the empty variable *gene* remains unchanged. In line 14, a value is assigned to the previously empty variable *gene*. In line 18 we check for the size of the content of *gene*, whereas we extract parts of *gene* in lines 20 and 22. In all these cases the value of *gene* itself remains unchanged.

10.5.2 Command Expansion

Very often it is desired to use the output of a command in a shell script. Therefore, you need to call the command and catch its output. Basically, the call of the command is converted into its output—this is called command expansion. The output of commands can be expanded in two different ways. One can either use `$(command)` or graves: `'command'`. There is no difference between both methods. The following examples make things a bit clearer.

```
Terminal 64: Command Expansions
1  $ echo $(date)
2  Thu Jul 10 15:29:02 WEDT 2003
3  $ echo `date`
4  Thu Jul 10 15:29:13 WEDT 2003
5  $ files=$(ls)
6  $ echo $files
7  dna.seq enzymes.txt structure.pdb test.txt
8  $
```

In lines 1 and 3 of Terminal 64, the command `date` is executed and its result is returned and printed by `echo`. In line 5 we save the output of the `ls` command in the variable *files*. In the next line we print out the content of *files*.

10.6 Quoting

It is very important to understand how the shell handles quotations, that is the use of double quotes, single quotes, and the backslash (escape character). Quoting is used to remove or disable the special meaning of certain characters or words to the shell. Thus, quoting prevents reserved words from being recognized as such, and prevents variable expansion.

10.6.1 Escape Character

The backslash (\) is bash's escape character. It preserves the literal value of the next character that follows and prevents interpretation of the newline character as end-of-command.

```
Terminal 65: Escaping
1 $ echo "Research costs some $s"
2 Research costs some
3 $ echo "Research costs some \$s"
4 Research costs some $s
5 $
```

Terminal 65 shows you an example of the importance of the escape character. In line 1 “\$s” is interpreted as the variable *s*. In contrast, in line 3 the shell recognizes the dollar character as a literal.

10.6.2 Single Quotes

Enclosing characters in single quotes (') preserves the literal value of all characters within the quotes. Note: A single quote may not occur between single quotes, even when preceded by a backslash!

```
Terminal 66: Quoting
1 $ echo 'Research costs some $s'
2 Research costs some $s
3 $ echo 'Research costs some \$s'
4 Research costs some \$s
5 $
```

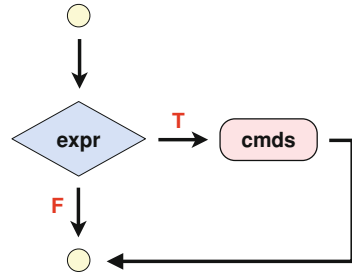
The effect of single quotes is demonstrated in Terminal 66. Note that even the backslash is recognized as literal and not interpreted as escape character.

10.6.3 Double Quotes

Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of the dollar character (\$), the grave character (`), and the backslash (\). The dollars character and grave retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: \$, `, ", \, or the newline character. Thus, a double quote may be quoted within double quotes by preceding it with a backslash.

A double-quoted string that is preceded by a dollar character is recognized as a variable name and will be replaced by its value. After replacement, the string is double-quoted.

Fig. 10.1 The `if...then` Construct. If the expression (*expr*) is true (T), then execute the command(s) (*cmds*). Typical expressions are discussed in Sect. 10.7.2. F = false



10.7 Decisions: Flow Control

One of the most important features of program is the power of flow control. In this section you will learn some fundamental programming constructs. Depending on the “answer” to given “questions”, the program decides where to proceed. This is why we talk of *flow control*: you can influence how the program flows through the lines. Every programming language has the capacity of flow control. You can execute loops (repetitively execute commands), analyze cases, and so on.

10.7.1 *if...then...elif...else...fi*

If...then is a conditional construct. It allows you to make a test before executing an action (see Fig. 10.1).

The conditional statement is introduced by `if` and evaluates an expression. Depending on the *exit status* of the expression, an action, preceded by `then`, is executed or not. Several such constructs can be grouped (`elif...then`). With `else` an action can be executed if all previous expressions were false. The conditional statement is closed with `fi` (reverse of `if`).

Thus, the complete syntax is:

```

if expression1; then
    action1
elif expression2; then
    action2
else action3
fi

```

-optional
-optional
-optional

The `elif` and `else` commands are optional. The rest is obligatory. If *expression1* returns 0 (i.e., true), then *action1* is executed. If *expression1* does not return 0 (i.e., false), then *expression2* is tested. If *expression2* returns 0, then *action2* is executed. Else, *action3* is executed. This means that either *action1*, *action2*, or *action3* is executed. Of course, each action could consist of several actions in separate lines.

Usually commands return the exit status 0 when they have finished their job without any error. This property is used in Program 8.

```

1  #!/bin/bash
2  # save as if-ls.sh
3  # tests if file is present in current dir
4  # needs 1 parameter
5  if ls $1 >/dev/null 2>&1; then
6      echo "$1 exists"
7  else
8      echo "$1 does not exist"
9  fi

```

Program 8: *if-ls.sh* - If...Then

Program 8 is executed with

```
./if-ls.sh file
```

The parameter *file* (which is saved in the variable *\$1*) is used by the `ls` command in line 5. If the file is present in the active directory, then `ls` will return the exit status 0, else 1. In order not to mess up the output screen with the output of `ls`, we redirect its output and errors (see Sect. 8.5 on p. 101) to `/dev/null`, the nirvana.

The sequence “*if command1 then command2 fi*” may also be written as “*command1 && command2*” (see Sect. 8.7 on p. 104). Conversely, “*command1 || command2*” executes *command2* only if *command1* returns an exit status other than 0 (false).

A powerful extension for the conditional *if...then* statement is provided by the command `test`, as shown in Sect. 10.7.2.

10.7.2 *test*

The `test` command, although not part of the shell, is intended for use by shell programs. It can be used for comparisons. For example, “`test -f file`” returns the *exit status* zero if *file* exists and a non-zero exit status otherwise. The exit status of the `test` command (or any other command) can then be analyzed and the behavior of the program be adapted accordingly. In general, `test` evaluates a predicate and returns the result as its exit status. Some of the more frequently used test arguments are given below. Note that *n1* and *n2* represent different numbers or variables containing numbers, *s1* and *s2* represent different text strings or variables containing text.

–Numbers–

<code>n1 -eq n2</code>	True if number <i>n1</i> is equal to number <i>n2</i> .
<code>n1 -le n2</code>	True if number <i>n1</i> is less than or equal to number <i>n2</i> .
<code>n1 -ge n2</code>	True if number <i>n1</i> is greater than or equal to number <i>n2</i> .
<code>n1 -lt n2</code>	True if number <i>n1</i> is less than number <i>n2</i> .
<code>n1 -gt n2</code>	True if number <i>n1</i> is greater than number <i>n2</i> .

–Strings–

<code>-n s1</code>	True if string <i>s1</i> is not empty.
<code>-z s1</code>	True if string <i>s1</i> is empty.
<code>s1 = s2</code>	True if string <i>s1</i> is equal to string <i>s2</i> .
<code>s1 != s2</code>	True if string <i>s1</i> is not equal to string <i>s2</i> .

–Files–

<code>-e file</code>	True if <i>file</i> exists.
<code>-f file</code>	True if <i>file</i> is a file.
<code>-d file</code>	True if <i>file</i> is a directory.
<code>-r file</code>	True if <i>file</i> is readable.
<code>-w file</code>	True if <i>file</i> is writable.
<code>-x file</code>	True if <i>file</i> is executable.
<code>-s file</code>	True if <i>file</i> is not empty.

With the help of `echo` you can analyze the result of the `test` command in the command line.

```

_____ Terminal 67: Function test _____
1  $ test 2 -eq 2; echo $?
2  0
3  $ test 2 -eq 3; echo $?
4  1
5  $ test $USER; echo $?
6  0
7  $

```

The exit status of `test` is saved in the variable `$?` . Thus, as shown in Terminal 67, you can check the exit status with “`echo $?`”. In order to write two commands on one line we have to separate them with a semicolon. If the comparison by `test` returns *true* (exit status 0), then `$?` is zero, and vice versa. The following program gives an example of the combination of *if...then* and `test`.

```

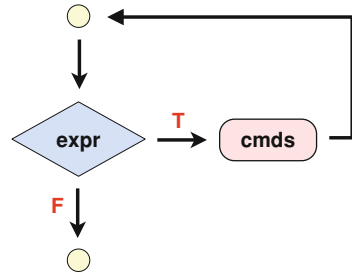
_____ Program 9: test-par.sh - Testing Parameters _____
1  #!/bin/bash
2  # save as test-par.sh
3  # tests if the number of parameters is correct
4  if test $# -ne 1; then
5      echo "Program needs exactly 1 parameter"
6      echo "bye bye"
7      exit 1
8  fi

```

Program 9 checks whether the value of the variable `$#` is not equal to 1. Remember that `$#` contains the number of parameters (see Sect. 10.4.4 on p. 133). Note that the arguments of `test` are separated by spaces. The `if` command construct in line 4 is separated from the `then` command by a semicolon. Otherwise, `then` must be in a new line. In modern times, it is very common to invoke `test` with brackets. Then line 4 would become

```
if [ $# -ne 1 ]; then
```

Fig. 10.2 The `while...do` Construct. While the expression (*expr*) returns true (T), the command(s) (*cmds*) are executed. F = false



Note that there are spaces around the brackets!

Another good example is Program 23 on p. 152 in Sect. 10.11 on p. 152.

10.7.3 `while...do...done`

The `while` construction is used to program loops. A loop is a construct that allows us to execute one or more actions again and again (see Fig. 10.2).

A while loop is aborted when a command-expression exits with a non-zero exit status (i.e., the expression becomes false). The loop is introduced with `while` and ends with `done`.

The syntax is:

```

while expression; do
    action(s)
done
  
```

The following program demonstrates the function of `while...do`.

```

----- Program 10: triplet.sh - While...Done -----
1  #!/bin/bash
2  # save as triplet.sh
3  # splits a sequence into triplets
4  x=0
5  while [ -n "${1:$x:3}" ]; do
6      seq=$seq${1:$x:3} " "
7      x=$((expr $x + 3))
8  done
9  echo "$seq"
  
```

Program 10 expects a DNA sequence from the command line. Thus, the script is executed with

```
./triplet.sh atgctagctcgtagctagctcg
```

The DNA sequence is then split into triplets and printed out. In line 4 we assign the value 0 to the variable `x`. The important line is line 5. Here we use the brackets to invoke the `test` command (see Sect. 10.7.2 on p. 139). We have learned that

the option `-n` will return the exit status 0 when a given string is not empty (see Sect. 10.7.2 on p. 139). This means that the expression

```
[ -n "$1:$x:3" ]
```

is true, as long as triplets can be copied from the variable `$1` (the command line parameter). Remember that `${a:b:c}` gives `c` characters from position `b` of the variable `a` (see Sect. 10.5.1 on p. 134). Thus, line 5 reads: while `$1:$x:3` is not empty, do execute the commands up to `done`. In line 6, a triplet and a space character are added to the variable `seq`. Then, the counter variable `x` is increased by 3 (3 nucleotides = one triplet). Finally, the sequence is displayed.

Together with the command `shift`, the `while` loop can be used to read all command line parameters. With `shift` the command line parameters are shifted. The parameter assigned to `$9` is shifted to `$8`, `$8` to `$7`, and so on. The parameter in variable `$1` is lost. An appropriate way to read all command line parameters would be:

```

_____ Program 11: para.sh - Reading Command Line Parameters _____
1  #!/bin/bash
2  # save as para.sh
3  # prints command line parameters
4  while [ -n "$1" ]; do
5      echo "\$#=$# - \$0= $0 - \$1=$1"
6      shift
7  done

```

The `while` loop in Program 11 cycles as long as there are parameters in `$1`. The output of the program is the number of remaining parameters from variable `$#`, the name of the script from `$0`, and the active parameter from `$1`.

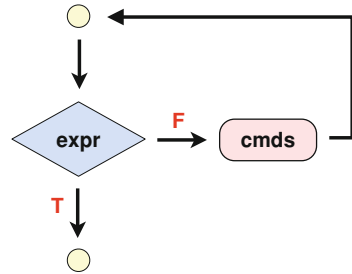
Sometimes you might end up with a loop that does not end: the program hangs. In that case you can quit the program with `(Ctrl)+(C)`. Alternatively, you can program the loop in such a way that it breaks when a certain condition is fulfilled. The corresponding command is `break`. For example, we could stop Program 10 when a stop codon is reached.

```

_____ Program 12: triplet-stop.sh - While...Break _____
1  #!/bin/bash
2  # save as triplet-stop.sh
3  # splits a sequence into triplets
4  x=0
5  while [ -n "${1:$x:3}" ]; do
6      seq=$seq${1:$x:3} "
7      x=$((x + 3))
8      if [ "${1:$x:3}" == taa ] || [ "${1:$x:3}" == tga ]; then
9          break
10     fi
11 done
12 echo "$seq"

```

Fig. 10.3 The `until...do` Construct. The command(s) (*cmds*) are executed until the expression (*expr*) becomes true (T). F = false



Program 12 shows an example of how the execution of a loop can be stopped. The conditional statement *if...then* breaks the *while...do* loop if the current triplet is a *taa* or *tga*. The *or* is represented by the two vertical bars (`|`). Remember that expressions can also be connected by the logical *and* statement, represented by two ampersands (`&&`).

10.7.4 *until...do...done*

Very similar to the *while* loop is the *until* loop. In fact, it is the negation of the former (see Fig. 10.3).

An action is not executed while a certain condition is given but until a certain condition is given. The syntax is:

```

until expression; do
    action(s)
done
  
```

Program 13 is the negation of Program 10 on p. 141. Instead of splitting the command line parameter into triplets *while* there are still nucleotides. Program 13 splits the sequence until there are no nucleotides left in variable *\$I*.

```

----- Program 13: triplet-until.sh - Until...Done -----
1  #!/bin/bash
2  # save as triplet-until.sh
3  # splits a sequence into triplets
4  x=0
5  until [ -z "${1:$x:3}" ]; do
6      seq=$seq${1:$x:3} " "
7      x=$((expr $x + 3))
8  done
9  echo "$seq"
  
```

Note the different option for `test`. Here the option is `-z` (returns 0 if the *string* is empty), in Program 10 on p. 141 it was `-n` (returns 0 if the *string* is not empty).

10.7.5 *for...in...do...done*

Another useful loop is the *for...do* construct. This is also known as a *for* loop. In contrast to the *while* loop, the *for* loop is not dependent on the exit status of an expression but on a list of variables. The syntax is:

```
for variable in list; do
    action(s)
done
```

Let us take a look at a very simple example. I even have the impression that this is the shortest loop script we will write.

```
_____ Program 14: for1.sh - Looping _____
1  #!/bin/bash
2  # save as for1.sh
3  # demonstrates for construct
4  for i in one two three; do
5      echo "$i"
6  done
```

In Program 14 the variable *i* takes successively the string values “one”, “two”, and “three”.

```
_____ Program 15: for-num.sh - Looping with Numbers _____
1  #!/bin/bash
2  # save as for-num.sh
3  # demonstrates for construct
4  for i in $(seq 1 10); do
5      echo "$i"
6  done
```

In Program 15 the variable *i* takes successively the numerical values 1 to 10. Therefore, we use the command *seq*.

The list could also be the result of a command. For example, all lines of a file could be read with *cat*. In the Program 16 the *ls* command provides a list of all script files.

```
_____ Program 16: for-ls.sh - Looping _____
1  #!/bin/bash
2  # save as for-ls.sh
3  # lists all scripts
4  for i in $(ls *.sh); do
5      echo "File: $i"
6  done
```

If the *for* loop is executed without any list, then the command line parameters are successively called. Thus, Program 11 on p. 142 can be run as shown in Program 17.

```

1  #!/bin/bash
2  # save as for-par.sh
3  # gets command line parameters
4  for i do
5      echo "\$#=$# - \$0=$0 - \$i=$i"
6  done

```

Program 17 does not make use of the `shift` command. The price is that the parameter count `$#` does not decrease its value. In other words, all parameters are kept in their corresponding positional variable `$i`.

10.7.6 *case...in...esac*

As the name implies, you can distinguish different cases with `case`. This is especially helpful when you want to create small user interfaces to guide the user through your program. The `case` construct is initiated with `case` and ends with `esac` (the reverse of `case`). The syntax is:

```

case string in
    pattern1)
        action(s)1
    pattern2)
        action(s)2
esac

```

The *string* is compared against the *patterns*. If a pattern matches, the corresponding action will be executed. Again, things get clearer with an example.

```

1  #!/bin/bash
2  # save as case-cp.sh
3  # asks if files shall be backuped
4  # accepts file extension as $1
5  for i in $(ls *$1); do
6      echo "Backup $i? yES/nO/qUIT"
7      read answer
8      case $answer in
9          y*) echo "Backup $i"; cp $i $i.bak;;
10         n*) echo "Skip $i";;
11         q*) echo "Quited"; exit;;
12         *) echo "Skip $i";;
13     esac
14 done

```

Program 18 asks you for each file in the current directory whether you want to create a backup or not. The backup is a copy of the file with the extension `.bak`. In order to restrict the file selection, the program accepts a file extension as command line parameter. Thus, if you call the program with

```
./case-cp.sh .sh
```

only script files will be listed. In line 5 the files list is created with a `for` loop. Line 6 asks whether the current file shall be backed up. Your answer is stored in *answer*. Now starts the case discrimination. If *answer* matches *y**, that is *y* plus anything, then line 9 is executed. If *answer* matches *n* plus anything, then line 10 is executed and so on. If *answer* matches neither *y**, *n**, nor *q**, then line 12 is executed. You can also connect several patterns with a *logical or* (`|`). Thus, `j*[Y*]` would match *yes* and the German *ja*. This can be helpful if you want to create an international interface for your program.

10.7.7 *select...in...do*

Similar to `case` is `select`. However, with `select` a list is generated from which you can choose. The syntax is as follows:

```
select name in words; do
    action(s)
done
```

The list of *words* following `in` can be either individual words or a variable or a command like `ls` that is expanded, generating a list of items. The list is then displayed on the screen (standard output), each preceded by a number. If `in words` are omitted, the positional parameters supplied with the command line are printed (see Sect. 10.4.4 on p. 133). During execution a prompt appears and you are requested to enter a number and press `(Enter)`. You can display a text-like “Enter number: ” by assigning the message to the variable *PS3*. If the entered number corresponds to one preceding the displayed words, then the value of the variable *name* is set to that word. If the line is empty, the words and prompt are displayed again. Any other value causes *name* to be set to null. The input is saved in the variable *REPLY*. After each selection, the list of words is displayed again and again, until a `break` command is executed. Let us take a look at one example.

```

----- Program 19: select.sh - Selections -----
1  #!/bin/bash
2  # save as select.sh
3  # demonstrates select
4  PS3="Select item: "
5  select name in Protein DNA RNA ; do
6      echo "You selected $name"
7      break
8  done
9  echo "bye bye"
```

In line 4 of Program 19 we assign the value “Select item: ” to the variable *PS3*. With this setting the user will be asked to select a item when the list is displayed. Here, the list consists of the three words “Protein DNA RNA”. When you run the script it will look as in Terminal 68.

```
Terminal 68: Execution of select.sh
1  $ ./select.sh
2  1) Protein
3  2) DNA
4  3) RNA
5  Select item: 3
6  You selected RNA
7  bye bye
8  $
```

With line 7 of Program 19 you force the *select...done* loop to break and continue the script after the *done* statement. In our case we just print “bye bye”.

Like the *case* construct, the *select* construct is well suited to build a simple user interface for your scripts. Such user interfaces are often highly welcome by the user because they facilitate intuitive usage of your script.

10.8 Desktop Notifications

The runtime of some data processing or analyzing scripts may be very long. Thus, it might be more productive to work on something else instead of starring at the screen while the script is running. What we want is a notification on the desktop (Fig. 10.4)—or even on your mobile phone. There are solutions available. Here I will focus on Growl for MacOSX (<http://growl.info>) and Windows (<http://www.growlforwindows.com>) and the Ubuntu package *libnotify-bin*.

10.8.1 MacOSX

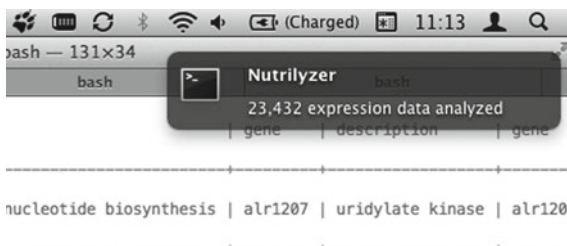
After downloading (<http://growl.info/extras.php#growlnotify>) the diskimage and installing the application you can run *growlnotify* from the command line:

```
growlnotify -t "Title" -m "Message" -s
```

Note that the installation package is placed in *Extras/growlnotify/growlnotify.pkg*. Note also that version 1.2.2 is the last free version of Growl.

You can even get notifications on the go on your iPhone or iPad via Prowl (check the iTunes AppStore) for more details.

Fig. 10.4 Growl. Desktop notification from a shell script on a Mac



10.8.2 Linux

On Ubuntu systems you can install the package *libnotify-bin*. It contains the command line application `notify-send` that allows you to send system notifications. Terminal 69 shows the installation and usage.

```

1  $ sudo apt-get install libnotify-bin
2  ...
3  $ notify-send --help
4  ...
5  $ notify-send 'Title' 'Message'
6  $

```

10.8.3 Windows

There is of course a version for Windows users, too: <http://www.growlforwindows.com>.

10.9 Debugging

A very important part of programming is debugging. If there is an error in your program, this is called a *bug*. The process of searching and removing any bug is called *debugging*. Of course, there is no automatic procedure to remove bugs. You have to go through your program line by line and identify what is wrong. This is the really time-intensive part of programming.

10.9.1 *bash -xv*

The shell offers you some support for debugging. When you start the shell with the option `-x`, it will print each command it is executing. Thereby, you might identify the malleus line. With the option `-v`, each line of the script is printed before

execution. Thus, you see the original command without any command expansion [like `$(command)`] or parameter substitution.

```

----- Program 20: date-vx.sh - Debugging -----
1  #!/bin/bash -vx
2  # save as date-vx.sh
3  # demonstrates debugging
4  date
5  lss
6  now=$(date)
7  echo "bye bye"

```

In the first line of Program 20 we evoke the bash shell with the options `-vx`. The commands of the script should be clear. In line 4 we call the command `date`, in line 5 the non-existing command `lss`, in 6 we assign the current date to the variable *now* and in line 7 we print *bye bye*. Now let us take a look at what we see when we execute the program.

```

----- Terminal 70: Debugging -----
1  $ date-vx.sh
2  #!/bin/bash -vx
3  # save as date-vx.sh
4  # demonstrates debugging
5  date
6  + date
7  Son Mai 25 11:16:44 CEST 2003
8  lss
9  + lss
10 ~/scripts/date-x.sh: line 5 lss: command not found
11 now=$(date)
12 date
13 ++ date
14 + now=Son Mai 25 11:16:44 CEST 2003
15 echo "bye bye"
16 + echo 'bye bye'
17 bye bye
18 $

```

Terminal 70 shows the output of Program 20. You see that all program lines are printed onto the screen. Commands that are executed are preceded by either `+` or `++`. You can also see the error message from the non-existent `lss` command.

When you write larger scripts, you might get confused by all the output. Thus, you might want to restrict the error search to a certain part of your program. This is possible with the `set` command.

```

----- Program 21: date-set.sh - Debugging -----
1  #!/bin/bash
2  # save as date-set.sh
3  # demonstrates debugging
4  date
5  lss
6  set -vx
7  now=$(date)
8  set +vx
9  echo "bye bye"

```

In line 6 of Program 21 we activate the options `-vxx`, in line 8 we deactivate them. By doing this, we restrict debugging to line 7.

10.9.2 *trap*

Some signals cause shell scripts to terminate. The most common one is the interrupt signal `(Ctrl)+C` typed while a script is running. Sometimes a shell script will need to do some cleanup, such as deleting temporary files, before exiting. The `trap` command can be used either to ignore signals or to catch them to perform special processing. For example, to delete all files called `*.tmp` before quitting after an interrupt signal was received, use the command line

```
trap 'rm*.tmp; exit' 2
```

The interrupt signal `(Ctrl)+C` corresponds to signal 2. If this signal is received, two commands will be executed: `rm *.tmp` and `exit`. You can make a shell script continue to run after logout by letting it ignore the hang up signal (signal 1). The command

```
trap '' 1.
```

allows shell procedures to continue after a hang up (logout) signal.

```
----- Program 22: trap.sh - Trap Signal -----
1  #!/bin/bash
2  # save as trap.sh
3  # catch exit signal
4  while true; do
5      echo "test"
6      trap 'echo "bye bye"; exit' 2
7  done
```

Be careful when you enter Program 22. `while` always gets the result `true`. Thus, we initiate an endless loop. The screen will fill up with the message *test*. However, when you press `(Ctrl)+C`, line 6 is executed. It *traps* the termination signal, prints *bye bye*, and exits the script. You should be very careful when you use `trap`. It is always a good option to terminate a program. However, if you misspell, for example, `exit`, then your program will hang.

10.10 Remote Control Interactive Programs

A very important aspect of shell programming is controlling interactive command line programs. One such example is the Surface Racer, which is described in Sect. 21.2.3 on p. 410. This tool identifies atoms that are accessible from the protein surface.

Unfortunately, the program does not read command line parameters but depends on human interaction. Terminal 71 illustrates the problem.

```

Terminal 71: Program Requiring Human Interaction
1  $ ./surface5_0_linux_32bit
2  Surface Racer 5.0 by Oleg Tsodikov
3  Analytical surface area calculation
4
5  Van del Waals radii sets:
6  1 - Richards (1977)
7  2 - Chothia (1976)
8  Press 1 or 2 to choose a van der Waals radius assignment:1          # INTERACTION
9
10 Input PDB file of the structure:1AXB.pdb                             # INTERACTION
11
12 Input the probe radius in Angstroms:5                                # INTERACTION
13
14 Enter a number to choose the calculation mode:
15 1- Accessible surface area only
16 2- Accessible and molecular surface areas
17 3- Accessible, molecular surface areas and average curvature of MS
18 Mode number:2                                                        # INTERACTION
19
20 Reading atomic coordinates and assigning radii ...
21 1000 atoms traced
22 2000 atoms traced
23
24 Solvent accessible surface areas (in Angstrom^2):
25 Total area = 12243.43, Polar area= 7483.12 , Non-polar area= 4760.31
26
27 Molecular surface areas (in Angstrom^2):
28 Total area = 8026.70, Polar area= 3887.07 , Non-polar area= 4139.63
29 Vertices=854 circles=0
30
31 Press <Enter> to check for cavities                                  # INTERACTION
32
33 The structure contains no cavities
34
35 The solvent accessible atomic areas are saved in the file *.txt
36
37 The breakdown of surface areas is in the file result.txt
38
39 Press <Enter> to quit                                                # INTERACTION
40 $

```

As you can see, the program stops six times and waits for user input (I commented those lines in Terminal 71 with INTERACTION). Hmm, we do not like this, right. Imagine you would like to analyze 100 structure files. That would literally glue you to your computer keyboard. How can we circumvent this?

The solution is rather easy. You pipe the required information to the script as shown in Terminal 72.

```

Terminal 72: Program Requiring Human Interaction
1  $ for i in TestSet/*.pdb; do echo "1 $i 5 1 n n" | ./surface5_0_linux_32bit; done
2  Surface Racer 5.0 by Oleg Tsodikov
3  Analytical surface area calculation
4
5  Van del Waals radii sets:
6  1 - Richards (1977)

```

```

7 | 2 - Chothia (1976)
8 | Press 1 or 2 to choose a van der Waals radius assignment:
9 | ...
10 | $

```

The `for` loop is used to call the program for all PDB structure files stored in the subdirectory *TextSet* (see also Sect. 8.10 on p. 106). The magic part is

```
echo "1 $i 5 1 n n" | ./surfrace5_0_linux_32bit
```

It pipes six text strings to *surfrace*, thereby replacing human interaction.

10.11 Examples

There is no other way to learn than by doing. In order to practise the function of shell scripts you will find a number of examples in this section. Save these scripts and execute them. Then start to change things and see how the scripts behave.

10.11.1 Check for DNA as File Content

Program 23 tests whether a file contains the characters *actg* or not.

```

----- Program 23: dna-test.sh - Check File Content -----
1 | #!/bin/bash
2 | # save as dna-test.sh
3 | # test if file contains dna sequence
4 | # file in $1
5 | if test -z $(cat $1) && test -f $1; then
6 |     echo "File $1 is empty"
7 |     exit
8 | fi
9 | grep -sq '^[acgt]' $1
10 | result=$?
11 | if test $result -eq 0 ; then
12 |     echo "File $1 does not contain pure DNA sequence"
13 | elif test $result -eq 1; then
14 |     echo "File $1 does contain pure DNA sequence"
15 | elif [ $result -eq 2 ]; then
16 |     echo "File $1 does not exist"
17 | else
18 |     echo "Some error occurred!"
19 | fi

```

You call the script with the name of the file that you want to analyze:

```
./dna-test-sh seq.dna
```

In line 5 of Program 23 there are two tests connected by `&&` (logical *and*). This means that lines 5 and 6 are executed only if both tests return *true*. Alternatively, one might want to use the logical *or*, which is `||`. The first test in line 5 checks if the file content is empty; the second test checks if the file exists. If one of these two conditions is false, the message in line 6 will be displayed and the program exits. Otherwise, the `grep` command in line 9 checks if the file given with the parameter `$1` contains characters other than *actg*. Depending on the result, `grep` finishes its job with different exit states. If `grep` has found other characters than *acgt*, then the exit status saved in `?` is 0, or else it is 1. If the file does not exist, then the exit status of `grep` is 2. The exit status is saved in the variable *result* in line 10. Then the value of *result* is tested in a row of *if...then...elif...else...fi* as described in Sect. 10.7.1 on p. 138. In line 15 an alternative way to call `test` is shown: the command can be replaced by using brackets `[ ]`.

10.11.2 Time Signal

The following program “beeps” the number of hours. When you run the script at 4 o’clock, you will hear 4 system beeps. You can call the script from the cron daemon every hour in order to have a time signal.

Program 24: *time-signal.sh* - Time Signal

```

1  #!/bin/bash
2  # save as time-signal.sh
3  # gives a time signal every hour when connected to cron
4  time=$(date +%I)
5  count=0
6  while test $count -lt $time; do
7      echo -e "\a"
8      sleep 1 # sleep for one second
9      count=$((count+1))
10 done

```

The heart of Program 24 lies in line 4. “`$(date +%I)`” returns the current hour in 12-hour format (01–12). This value is saved in the variable *time* and used in the `while` loop spanning from lines 6 to 10. The beep signal is generated by the escape sequence “`\a`” in line 7 (see Sect. 10.4.1 on p. 130).

10.11.3 Select Files to Archive

This script asks the user for each file in the active directory whether it should be added to an archive or not. Finally, all selected files are archived in a file called *archive*.

```

1  #!/bin/bash
2  # save as archive-pwd-i.sh
3  # interactively archive files with tar
4  array=( $\$(ls)$ )
5  count=0
6  while test  $\$count -lt \${\#array[*]}$ ; do
7      echo "Archive  $\${array[count]}?$ "
8      echo "  press Enter = no"
9      echo "  press y & Enter = yes"
10     read input
11     case  $\$input$  in
12         y*) list=" $\${list} \${array[count]}$ "
13         esac
14     count= $\$(expr \$count + 1)$ 
15 done
16 echo "Files:  $\$list$ "
17 echo "have been added to the file archive"
18 tar -cf archive  $\$list$ 

```

Program 25 demonstrates the use of command expansion. In line 4 the output of the command `ls` is resolved and saved into an array. Here, an array is a 2D variable, like a staple of paper sheets. Each sheet contains some information. Variable 1 is `array[0]` that contains the first file delivered by `ls`. Variable 2 is `array[1]` that contains the second file of the directory as value and so on. With `$\${\#array[*]}$` we can check how many elements the array has. In line 6 we use this statement for a loop. The loop `while` to `done` is repeated until the command `test` gives the output “false”, that is, until the variable `count` is less than (`-lt`) the size of the array determined by `$\${\#array[*]}$` . What is going on in the `while...done` loop? In line 7, the user is asked whether the file that is saved in the array element `array[count]` should be archived. The user must answer the question as stated in lines 8 and 9. Then follows the `read` command (see Sect. 10.4.3 on p. 132). At this point the program stops and waits for user input. The program resumes execution when the user presses Enter. The text typed by the user is saved in the variable `input`. Between lines 11 and 13 a case discrimination is performed. It starts with `case` and ends with `esac` (the reverse of `case`). In line 11 the program checks whether the variable `input` is equal to something in the following lines, up to the `esac` command in line 13. The case we are interested in is whether the user typed `y` or `yes`. This can be expressed by `y*`: any word that starts with `y`. If `input` matches that case, then the command following `y*` is executed: here we add the file to the variable `list`. The command reads like: save the content of the variable `list`, plus a space character (`␣`), plus the current content of the array element `array[count]` in the variable `list`. In line 14 we increase the content of the variable `count` by one. In order to perform an arithmetical calculation we use the command `expr` (expression). That tells the shell to calculate the rest of the line instead of just adding the character string “+ 1” to the variable `count`. When the value of the variable `count` exceeds the number of files in the current directory (saved in `array[ ]`), then the `while` loop stops and the program continues with line 16. Here the list of files to be archived is printed out. The archiving command `tar` itself is executed in line 18. All files chosen by the user to be archived is provided with the variable `list`.

10.11.4 Remove Spaces

The following program converts spaces in filenames into underscores. Especially filenames created on computers running Windows tend to contain spaces. In principle, both Unix and Linux have no problems with spaces in filenames; however, you have to escape them with the backslash character. This is rather uncomfortable. If you have a number of files that need to be converted, then you will appreciate the following script.

```

1  Program 26: space-convert.sh - Convert Spaces in Filenames to Underscores
2  #!/bin/sh
3  # save as space-convert.sh
4  chgname() {
5      echo "$1" | sed -e 's/[ ][ ]*/ /g' -e 's/[ ]/_/g'
6  }
7  find . -name '* *' | sort | while read name; do
8      file=`basename "$name"`
9      stem=`dirname "$name"`
10     nfile=`chgname "$file"`
11     nstem=`chgname "$stem"`
12     if [ "$file" != "$nfile" ]
13     then
14         mv "$stem/$file" $nstem/$nfile
15     fi
16 done

```

Program 26 begins by defining a user function in line 3. It pipes filenames saved in *\$1* to a little *sed* script. This script actually transforms space characters into one single underscore character. You will learn more about *sed* in Chap. 12 on p. 175. The *find* command in line 6 queries the active directory for files containing space characters. These are sorted by *sort* and finally saved in the variable *name*. The shell command *basename* extracts the filename of *name*, whereas *dirname* extracts the path of the file saved in *name*. In lines 9 and 10 the user function *chgname* is called. In line 13 the original files are renamed.

Exercises

The following exercise sounds easier than it is...

10.1 Go through all the programs and examples in this section and play around. Modify the code and observe the changes.

Chapter 11

Regular Expressions

Take a look at Fig. 11.1. Does this look weird to you? It is the cover of a Japanese book about regular expressions. The title reads *regular expressions* in Kanji script. And if you see regular expressions for the first time, they do look weird and cryptic. However, they are EXTREMELY useful and I cannot overemphasize the word extremely.

I am sure you have more than once used the Internet search engine *Google*. You have not? Then you probably used another search engine—there are plenty of them out there in the Web space. As with all searches, the problem is *finding*. The better you define your search problem, the better your query result will be. In this chapter you will learn how to query for text patterns.

11.1 Regular Expression and Neurons

The basis to regular expressions has been created in the early 1940s by the two neurophysiologists Warren McCulloch and Walter Pitts. They developed models for the function of the nervous system at the level of the neurons (McCulloch and Pitts 1943). Several years later, the mathematician Stephen Kleene formulated an algebra for these models, which he called regular sets. In addition, he designed a notation for algebra and called it *regular expressions*. In the 1950s and 1960s, regular expressions were a popular research topic of theoretical mathematics. Probably, the first use of regular expressions in computers is described in an article entitled *Regular Expression Search Algorithm* by Ken Thompson, alongside Dennis Ritchie, one of the developers of Unix (Thompson 1968).

Later, this work resulted in the development of the first UNIX editor, *ed* (editor). Later, on the basis of *ed*, the line-oriented editor *sed* (stream editor) was developed. Both *ed* and *sed* have a command in common that print lines match a regular expression: *g/regular expression/p* (read: **g**lobal/**regular expression**/**p**rint). This function was needed so frequently that some programmers have developed the stand-alone programs *grep* and *egrep* in the 1970s.



Fig. 11.1 RegEx. Japanese translation of Jeffrey E.F. Friedl's book about regular expressions

11.2 Get Started ...

Some definitions before we start: we are going to use the terms *literal*, *metacharacter*, *target string*, *escape sequence*, and *search pattern*. The following is a definition of these terms:

Literal	A literal is an actual character that we use in our search, for example, to find <i>inu</i> in <i>Linux</i> . <i>inu</i> is a literal string—each character plays a part in the search; it is literally the string we want to find.
Meta Character	Meta characters are one or more special characters that have a unique meaning and are not used as literals in the search. For example, the dot character “.” is a meta character.
Escape Sequence	An escape sequence is a way of indicating that we want to use one of the meta characters as a literal. In a regular expression an escape sequence involves placing a backslash (\) in front of the meta character. In order to find the dot character we have to use the escape sequence “\.”.
Target String	The target string defines the sequences of characters we are searching for. In other words, the target string is the string we want to match in the source file.
Search Pattern	The search pattern, or construct, describes the expression that we are using in order to search our target string.

A regular expression is a set of characters that specify a *search pattern*. They are used when you want to search for lines of text containing a particular sequence of characters, the *target string*. It is simple to search for a specific word or string of characters. Almost every text editor, like Microsoft Word, on any computer system can do this. Unlike simple text queries, regular expressions allow you to search for text which matches a particular pattern. You can search for words of a certain size, or for a word with three *ts*. Numbers, punctuation characters, DNA sequences, you name it, a regular expression can find it. Regular expressions are always used in conjunction with a program. What happens once the program you are using has found the pattern is another matter. Some just search for the pattern and print out the line containing the pattern (`grep`). Editors can replace the string with a new pattern (`vi`). It all depends on the utility. Regular expressions look a lot like the file-matching patterns (wildcards) the shell uses (see Sect. 8.12 on p. 108). Sometimes they even act in the same way: the brackets are similar, and the asterisk (*) acts similarly, too. However, regular expressions are much more powerful than wildcards.

Remember that filename wildcards are expanded before the shell passes the arguments to the program. To prevent this expansion, *regular expression must be embedded within single quotes*. The following examples give you a first impression of how you can find target strings with regular expressions.

<i>Search Pattern</i>	<i>Target String</i>
<code>(DNA RNA)</code>	Search for “DNA” or “RNA”.
<code>(D R)NA</code>	Search for “DNA” or “RNA”.
<code>[tmr]RNA</code>	Search for “tRNA”, “mRNA”, or “rRNA”.
<code>*omics</code>	Search for “Genomics”, or “Proteomics”, or any other “ <i>nomics</i> ”.
<code>b.g</code>	Matches any character between “b” and “g”, like big, bug, bag...
<code>[a-zA-Z]</code>	Matches any single lowercase or uppercase character.
<code>[~a-zA-Z0-9]</code>	Matches any single character which is not a number or letter.
<code>^[eE]nzyme</code>	Matches the words “Enzyme” or “enzyme” at the beginning of a line.
<code>[eE]nzyme\$</code>	Matches the words “Enzyme” or “enzyme” at the end of a line.

As you will learn soon, there are much more powerful queries possible.

11.3 Using Regular Expressions

Regular expressions do not work on their own. They are used together with a command or programming language, such as `grep`, `find`, `vi`, `sed`, `awk`, `perl`, *MySQL* (a database system), *Javascript*, *Java*, *PHP*, and many more. If you want to use `grep` with regular expressions, you must run it with the option `-e`. Most Linux systems have a built-in alias, which is the command `egrep`. This means, using `egrep` is the same as using `grep -e`. Note: in contrast to `grep -e`, masking of meta characters (`? + | ( ) { }`) is not necessary when you use `grep -E` (extended regular expressions).

There is nothing easier than having the wrong regular expression to find a target string. It really needs some exercise in order to figure out how to make it right. Before you apply a search pattern for an important task, you should check it with a small self-made test file. It is important to check whether the regular expression really finds the desired target and that it excludes all non-wanted targets. This becomes especially important when we start to use regular expressions for text replacements. Test your regular expression thoroughly. As you will see, `egrep` (used in this section) and `vi` (see Sect. 11.6.1 on p. 172) are good tools for doing this.

As described above, regular expressions consist of two character types: *meta characters* and *literals*. Meta characters are special characters like wildcards. All other characters are literals. In a way, meta characters describe a desired arrangement of the literals, that is, the target string. For the following examples we will use the file *structure.pdb* (see File 4). It is a cut-down version of a file from the *Brookhaven Protein Data Bank* entry *ICFZ* describing the crystal structure of a protein. The complete file can be downloaded from www.rcsb.org/pdb.

```

File 4: structure.pdb
1  HEADER   Hydrogenase                      23-Mar-99   1CFZ
2  COMPND   Hydrogenase Maturing Endopeptidase Hybd From
3  SOURCE   ORGANISM_SCIENTIFIC: Escherichia coli
4  AUTHOR   Fritsche, Paschos, Beisel, Boeck & Huber
5  REMARK   NCBI PDB FORMAT VERSION 5.0
6  SEQRES   1 A 162  MET ARG ILE LEU VAL LEU GLY VAL GLY ASN
7  SEQRES   2 A 162  THR ASP GLU ALA ILE GLY VAL ARG ILE VAL
8  SEQRES   3 A 162  GLU GLN ARG TYR ILE LEU PRO ASP TYR VAL
9  SEQRES   4 A 162  ASP GLY GLY THR ALA GLY MET GLU LEU LEU
10 HELIX    1 hel ILE A 18  GLN A 28
11 HELIX    2 hel PRO A 92  THR A 107
12 HELIX    3 hel ILE A 138 SER A 152
13 SHEET    1 str ARG A 2  ASN A 10
14 SHEET    2 str ARG A 29  LEU A 32
15 SHEET    3 str TYR A 35  THR A 43
16 ATOM     1 C MET A 1 48.865 25.394 51.393 1.00 54.58 C
17 ATOM     2 CA MET A 1 49.879 24.359 50.932 1.00 59.61 C
18 ATOM     3 CB MET A 1 49.248 23.457 49.877 1.00 62.37 C
19 ATOM     4 CE MET A 1 51.349 24.403 47.765 1.00 71.39 C
20 ATOM     5 CG MET A 1 48.708 24.106 48.629 1.00 66.70 C
21 ATOM     6 N MET A 1 50.347 23.578 52.116 1.00 62.03 N
22 ATOM     7 O MET A 1 47.875 25.011 52.020 1.00 54.99 O
23 ATOM     8 SD MET A 1 49.731 23.948 47.163 1.00 77.15 S

```

You might ask: what is the content of this file? A protein can be characterized by three important features: its amino acid sequence (i.e., primary structure), the presence of α -helices and β -sheets as structural building blocks (secondary structure), and the overall three-dimensional (3D) structure (tertiary structure). With methods like X-ray or NMR spectroscopy one can resolve the tertiary structure of a protein. This means that to each atom, a position in space can be assigned. You know that it is the 3D structure that determines the function of a protein.

All structural information about a protein is saved in a special file format: the *Brookhaven Protein Data Bank File Format*. Our example file resembles this file format. However, the length of fields has been cut down for printing purposes. The first 5 lines of *structure.pdb* (File 4) give some background information, the next 4 lines contain the protein sequence, followed by secondary structure features (lines 10–15) and the position of the atoms. Generally, the file content is organized in lines. The content of each line is indicated by the first word (HEADER, COMPND, SOURCE, AUTHOR,...). Again, please note that both lines and rows of the original file have been truncated!

11.4 Search Pattern and Examples

In this section, you will learn more about different types of regular expressions and how to apply them. Here, we will work with the `grep` command that understands regular expressions, that is, `egrep`. `egrep` works in the same way as `grep` does (see Sect. 7.1.9 on p. 83): it searches the input (usually a file) for a specified text pattern. The input text is treated in lines. Lines that contain the query pattern are printed to the standard output (the screen). In contrast to `grep`, the query pattern used with `egrep` may contain regular expressions. Remember: The search pattern

must be enclosed in single quotes ('...'). This prevents the shell from executing substitutions. Our first example file will be File 4.

11.4.1 Single-Character Meta Characters

A typical way to write the name of an amino acid is the *three-letter code*. The amino acids glycine and glutamine are represented by GLY and GLN, respectively. Thus, they differ by only one character. There are different ways to query for such differences. The meta characters used in such queries are called *single-character meta characters*. These include the following symbols:

- .
 - []
 - [^]
 - [0-9]
- Matches any single character except the newline character.
- Matches any character listed between the brackets. [acgt] matches “a” or “c” or “g” or “t”. The four characters “.”, “*”, “[” and “\” stand for themselves within such a list of characters.
- Matches any character except those listed between the brackets.
- Matches any number in the *range* between 0 and 9. The dash (–) indicates a range of consecutive characters. [0-3a-cz] equals [0123abcz].

Okay, let us see how we can apply these meta characters in regular expressions. Let us assume that we want to match all lines containing *GLU* and *GLN* in File 4 (*structure.pdb*). The appropriate command is shown in Terminal 73.

```

1  $ egrep 'GL.' structure.pdb
2  SEQRES 1 A 162 MET ARG ILE LEU VAL LEU GLY VAL GLY ASN
3  SEQRES 2 A 162 THR ASP GLU ALA ILE GLY VAL ARG ILE VAL
4  SEQRES 3 A 162 GLU GLN ARG TYR ILE LEU PRO ASP TYR VAL
5  SEQRES 4 A 162 ASP GLY GLY THR ALA GLY MET GLU LEU LEU
6  HELIX 1 hel ILE A 18 GLN A 28
7  $ egrep 'GLN' structure.pdb
8  SEQRES 3 A 162 GLU GLN ARG TYR ILE LEU PRO ASP TYR VAL
9  HELIX 1 hel ILE A 18 GLN A 28
10 $ egrep 'GLY' structure.pdb
11 SEQRES 1 A 162 MET ARG ILE LEU VAL LEU GLY VAL GLY ASN
12 SEQRES 2 A 162 THR ASP GLU ALA ILE GLY VAL ARG ILE VAL
13 SEQRES 4 A 162 ASP GLY GLY THR ALA GLY MET GLU LEU LEU
14 $

```

Remember that the regular expression “GL.” in line 1 of Terminal 73 must be embedded in single quotes. “GL.” will match every line of File 4 on p. 161 that contains anywhere the succession of the characters *G* and *L* followed by any other character, which is represented by the dot (.). The matching lines are printed onto the screen. Lines 7 and 10 show the result of the search for the individual amino acids *GLN* and *GLY*.

Now let us search for the amino acids threonine (THR) and tyrosine (TYR). The correct command would be:

```
egrep 'T.R' structure.pdb
```

All lines where an amino acid lies between alanine (ALA) and glycine (GLY) are matched with

```
egrep 'ALA.....GLY' structure.pdb
```

In this example the five dots represent any five consecutive characters, including the spaces. The command matches exactly one line, which is line 7.

Alternatively, you could match the same pattern with

```
egrep 'ALA.....GLY' structure.pdb
```

This example shows that the spaces are treated as normal characters.

The brackets represent a single character from a selection of characters. This selection is contained within the brackets. In our file, three amino acids start with *GL*: glycine (GLY), glutamine (GLN), and glutamic acid (GLU). How can we match these lines containing either GLU or GLY? Now we employ a selection.

```

1  $ egrep 'GL[YU]' structure.pdb
2  SEQRES 1 A 162 MET ARG ILE LEU VAL LEU GLY VAL GLY ASN
3  SEQRES 2 A 162 THR ASP GLU ALA ILE GLY VAL ARG ILE VAL
4  SEQRES 3 A 162 GLU GLN ARG TYR ILE LEU PRO ASP TYR VAL
5  SEQRES 4 A 162 ASP GLY GLY THR ALA GLY MET GLU LEU LEU
6  $

```

The structure [YU] in Terminal 74 represents one character: either *Y* or *U*. The list could be much longer. The list may also contain a range of characters indicated by a dash (-).

```

1  $ egrep '[0-2T]_A' structure.pdb
2  SEQRES 1 A 162 MET ARG ILE LEU VAL LEU GLY VAL GLY ASN
3  SEQRES 2 A 162 THR ASP GLU ALA ILE GLY VAL ARG ILE VAL
4  ATOM 1 C MET A 1 48.865 25.394 51.393 1.00 54.58 C
5  ATOM 2 CA MET A 1 49.879 24.359 50.932 1.00 59.61 C
6  ATOM 3 CB MET A 1 49.248 23.457 49.877 1.00 62.37 C
7  ATOM 4 CE MET A 1 51.349 24.403 47.765 1.00 71.39 C
8  ATOM 5 CG MET A 1 48.708 24.106 48.629 1.00 66.70 C
9  ATOM 6 N MET A 1 50.347 23.578 52.116 1.00 62.03 N
10 ATOM 7 O MET A 1 47.875 25.011 52.020 1.00 54.99 O
11 ATOM 8 SD MET A 1 49.731 23.948 47.163 1.00 77.15 S
12 $

```

In Terminal 75, the pattern “[0-2T]_A” matches one of the characters *O*, *I*, *2*, or *T*, followed by the space character (*_*), followed by the character *A*. In order to invert a selection, it is preceded by a circumflex (^). Thus,

```
egrep '[^0-2T]_A' structure.pdb
```

would match every line where the pattern “[0-2T]_A” does *not* match.

Till now we have used only a single-character pattern. Now let us see how we can match repetitions of a particular pattern.

11.4.2 Quantifiers

The regular expression syntax also provides meta characters which specify the number of times a particular character should match. Quantifiers do not work on their own but are combined with the single character-matching patterns discussed above. Without quantifiers, regular expressions would be rather useless. The following list gives you an overview of quantifiers:

<code>?</code>	Matches the preceding character or list zero or one time.
<code>*</code>	Matches the preceding character or list zero or more times.
<code>+</code>	Matches the preceding character or list one or more times.
<code>{num}</code>	Matches the preceding character or list <i>num</i> times.
<code>{num, }</code>	Matches the preceding character or list at least <i>num</i> times.
<code>{min,max}</code>	Matches the preceding character or list at least <i>min</i> times, but not more than <i>max</i> times. The numbers must be less than 65536 and the first must be less than or equal to the second.

It is easy to get confused with these expressions, especially with the first three. The most universal quantifier is the star (*). In combination with the dot (.) it matches either no character or the whole line. As mentioned above, quantifiers are not used on their own. They always refer to the preceding character or meta character. Let us look at some examples with a new test file named *sequence.dna* (File 5). This file contains two arbitrary DNA sequences in the *FASTA format*. Sequences in FASTA format have a sequence name preceded by the “>” character. The following line(s) contain the sequence itself.

```

1  >seq1 the first test sequence
2  ATGxxxTAaxxATGxxTAAGACGCTAGCTCAGCATCGACTACGATCCT
3  GATAGCTATGTCGATGCTGATGCATGCATGCGGGGGGATTGAAAAAGG
4  CGTGTGTAGCGTAATATATGCTATAGCATTGGCATT
5
6  >seq2 the 2nd test sequence
7  AGCGGCGGCAGTACTGCTATTCGATGTACGGCGATATGCATGGGGATT
8  TAATAAACACAATGCGGTGTAGGGGAAAAATTnAGCATGCAATT

```

A little repetition: how do we get all lines containing sequence names filtered out? Just use

```
egrep '>' sequence.dna
```


Now let us match lines having a start codon (*ATG*) followed by some nucleotides and a stop codon (*TAA*):

```
egrep 'ATG.*TAA' sequence.dna
```

The next regular expression shows all lines containing two or three, but not more, consecutive *As*:

```
egrep '[CGT]AAA?[CGT]' sequence.dna
```

The question mark indicates that the preceding *A* might be present once or not present at all in the matching pattern. The `[CGT]` at the beginning and end of the regular expression prevents that a row of more than three *As* is recognized. This could have also been achieved with `[^A]`, representing any character but *A*:

```
egrep '[^A]AAA?[^A]' sequence.dna
```

If you want to detect all lines with two or more repeated *As* use

```
egrep '[^A]AA+[^A]' sequence.dna
```

The plus character indicates that the preceding *A* should be present at least one time. By using braces (`{ }`) you can exactly define the number of desired repetitions. Thus, to find a repetition of 4 *As* or more, use

```
egrep 'A{4,}' sequence.dna
```

If you want to search for braces you have to escape them with a backslash (see Sect. 11.4.5 on p. 167)!

As you can see from these examples, regular expressions are really powerful pattern-matching tools.

11.4.3 Grouping

Often it is very helpful to group a certain pattern. Then you have to enclose it in parentheses: (...). In this way you can easily detect repeats of the sequence *AT*:

```
egrep 'AT(AT)+' sequence.dna
```

Note that if we left away the first *AT*, we would also match single occurrences of *AT*, though we are looking for repeats, that is, more than one occurrence. The next example searches for repeating glycines in File 4 on p. 161:

```
egrep ' (GLY_) {2,} ' structure.pdb
```

Note that there is a space character after *GLY*. This is necessary because the amino acids are separated by spaces! In order to query for parentheses you must escape them with a preceding backslash (see Sect. 11.4.5).

11.4.4 Anchors

Often you need to specify the position at which a particular pattern occurs. This is often referred to as *anchoring* the pattern:

<code>^</code>	Matches the start of a line.
<code>\$</code>	Matches the end of a line.
<code>\ <</code>	Matches the beginning of a word.
<code>\ ></code>	Matches the end of a word.
<code>\b</code>	Matches the beginning or the end of a word.
<code>\B</code>	Matches any character not at the beginning or end of a word.

Let us come back to File 4 on p. 161 (*structure.pdb*). Let us find each line that begins with the character *H*. The correct command is

```
egrep '^H' structure.pdb
```

Now, you can also easily search for empty lines:

```
egrep '^$' sequence.dna
```

If you want to match all empty lines plus all lines only containing spaces, you use

```
egrep '^_*$' filename
```

With this command you see only empty lines on the screen. More probable is the situation that you want to print all non-empty lines. That can be achieved with

```
egrep '[^_]' filename
```

Empty lines, or lines containing only space characters, do not match this regular expression. Again, note that the space character is treated as any other literal. In regular expressions the space character cannot be used to separate entries!

Regular expressions will also help us to format the output of the `ls` command. How can we list all directories? This task can easily be achieved by piping the output of “`ls -l`” to `egrep`:

```
ls -l | egrep ' ^d'
```

`egrep` checks whether the first character of the file attributes is a “d” (see Sect. 5.3 on p. 56). In a similar manner you could list all files that are readable and writable by all users:

```
ls -l | egrep ' ^.{7}rw'
```

Here the search pattern requires that the 8th and 9th character of a line equals “r” and “w”, respectively.

11.4.5 Escape Sequences

By now, you are probably wondering how you can search for one of the special characters (asterisks, periods, slashes, and so on). As for the shell, the answer lies in the use of the escape character, that is, the backslash (\). In order to override the meaning of a special character (in other words, to treat it as a literal instead of a meta character), we simply put a backslash before that character. Thus, a backslash followed by any special character is a one-character regular expression that matches the special character itself. This combination is called an *escape sequence*. The special characters are:

- . * [\ Period, asterisk, left square bracket and backslash, respectively, which are always special, except when they appear within brackets ([]).
- ^ Caret or circumflex, which is special at the beginning of an entire regular expression, or when it immediately follows the left bracket of a pair of brackets ([]).
- \$ Dollar character, which is special at the end of an entire regular expression.

Wrong application of escape sequences, that is, wrong escaping, is the number one error when using regular expressions. Be aware of this fact and always test your construct before you are starting to do important things with it. Note: Depending on the program with which you are using regular expressions, meta characters like braces and parentheses must be escaped. This is, for example, the case in `sed` (see Sect. 12.4.2 on p. 181).

11.4.6 Alternation

In the first example in Terminal 73 on p. 162 we wanted to match both GLY and GLN. You will now learn an alternative way to achieve this: alternation. Alternation refers to the use of the “|” character to indicate logical OR. Commonly, the alternation

is used with parentheses to limit the scope of the alternative matches. Consider the following regular expression, which accounts for both GLY and GLN:

```
egrep ' (GLY|GLN) ' structure.pdb
```

This construction will give you the same result as is shown in Terminal 73 on p. 162. You can also combine more of these statements as shown in the following example:

```
egrep ' (GLY|GLN|ILE) ' structure.pdb
```

There is one point to keep in mind when you combine even more expressions. The regular expression “one and/or two” is equal to “(one and)|(or two)” but not equal to “one (and|or) two”! You are usually on the safe side by using parentheses!

11.4.7 Back References

In Sects. 11.4.3 on p. 165 and 11.4.6 you have already seen the use of parentheses in order to group constructs. However, the constructs within parentheses are not only grouped but also memorized in an internal memory. This means that you can refer to previously found patterns within your construct. The following meta characters take care of back referencing.

- () Memorizes the match for the regular expression enclosed in parentheses.
- \n Recalls the *n*th match. The match for the first construct enclosed in parentheses is recalled by \1, the match for the second construct enclosed in parentheses is recalled with \2 and so forth.

The application of back referencing becomes clearer with the following example. Assume we wish to find every line containing the repetition of a character. Terminal 76 shows you the solution.

```

_____ Terminal 76: Using Back References _____
1  $ egrep '([A-Za-z])\1' structure.pdb
2  SHEET 1 str ARG A      2  ASN A   10
3  SHEET 2 str ARG A     29  LEU A   32
4  SHEET 3 str TYR A     35  THR A   43
5  $

```

The construct “[A-Za-z]” enclosed in parentheses in line 1 of Terminal 76 matches any alphabetic character, regardless of whether it is upper or lowercase. Then we recall that match with “\1”. Thus, the whole construct matches every repetition of any characters.

With the next example we will find DNA sequence repeats in the file *sequence.dna*.

```

1  $ egrep '([ACTG][ACTG])\1' sequence.dna
2  GATAGCTATGTCGATGCTGATGCATGCATGCGGGGGATGAAAAAGG
3  CGTGTGTAGCGTAATATATGCTATAGCATTGGCATT
4  $

```

The regular expression in line 1 of Terminal 77 matches repeats of DNA nucleotide duplets. The match in line 2 is *GGGGGG*, whereas the match in line 3 is *GTGTGT* and *ATATAT*. With the following construct

```
egrep '([ACTG])([ACTG])([ACTG])\3\2\1' filename
```

we want to match inverted repeats of triplets, such as *TCAACT*, *GCGGCG* or *AAAAAA*. Note that you cannot use the construct

```
egrep '([ACTG]){3}\3\2\1' filename
```

for the same purpose. You would see the error message “bad back reference”. The reason is that the parentheses must be really present. You cannot *virtually* repeat them with a quantifier.

Unfortunately, it is not possible to indicate where the pattern occurs in the matching line. This is a limitation of *egrep* and we will overcome this limitation later with *sed*. Another limitation is that we cannot detect patterns stretching over two or more lines. *egrep* works through the file line by line and only analyzes single lines. Later we will see examples of how to overcome this limitation, too.

11.4.8 Character Classes

As we saw in Sect. 11.4.1 on p. 162, we can define a collection of single characters by enclosing them in brackets. These collections match exactly one single character. Another way to define such ranges is given in Table 11.1. They have the advantage of taking into account any variant of the local language settings or the coding system. Sometimes you might wish to use these extended search pattern sets. However, not all programs understand them. You have to try it out.

11.4.9 Priorities

From algebra you know about the priority of arithmetic operators. The operators of multiplication/division have a larger priority (are evaluated first) than the operators of addition/subtraction (which are evaluated last). There are similar rules for reg-

Table 11.1 Character classes for regular expressions

Pattern	Target
<code>[:alnum:]</code>	Any alphanumeric character 0 to 9 or A to Z or a to z.
<code>[:digit:]</code>	Only the digits 0 to 9.
<code>[:alpha:]</code>	Any alpha character A to Z or a to z.
<code>[:upper:]</code>	Any alpha character A to Z.
<code>[:lower:]</code>	Any alpha character a to z.
<code>[:blank:]</code>	Space and tabulator characters only.
<code>[:space:]</code>	Any space characters.
<code>[:punct:]</code>	Punctuation characters . , " ' ? ! ; :
<code>[:print:]</code>	Any printable character.

ular expressions. The order of priority of operators at the same parentheses level is `[]` (character classes), followed by `*`, `+`, `?` (closures), followed by concatenation, followed by `|` (alternation), and finally followed by the newline character. Parentheses have the highest priority. Thus, if you are in doubt, just use parentheses.

11.4.10 *egrep Options*

`egrep` offers a number of options that help you to fit the output to your needs.

- `-v` Print all lines that do not match the search pattern.
- `-n` Print the matched lines and corresponding line numbers.
- `-c` Print only the number of matches.
- `-i` Ignore the case of the input file. Thus the search pattern will match lower and uppercases.
- `-e` Protect patterns beginning with the dash (`-`) character.

Especially the last option can be very useful. DNA sequences are often saved in lower or uppercase. Thus, it is not desired to distinguish between lower and uppercase characters. You can tell `egrep` to ignore the case with the option `-i` (ignore).

11.5 RE Summary

See Table 11.2 on p. 171

Table 11.2 Regular expressions**Single-Letter Meta Character**

.	Any character
[aA]	Any of the listed characters
[^abc]	Any but the listed characters
[0-9a-z]	Any number or lowercase character

Quantifier

?	Non or one occurrence
+	At least one occurrence
*	Any number of occurrences
{num}	Exactly <i>num</i> occurrences
{min, }	At least <i>min</i> occurrences
{min, max}	Between <i>min</i> and <i>max</i> occurrences

Anchor

^	Beginning of a line
\$	End of a line
\<	Beginning of a word
\>	End of a word

Escape Character

\	Transforms a meta character to a literal when preceded to the meta character

Alternation

(RA1 RA2)	Matches to one of the two regular expressions <i>RA1</i> and <i>RA2</i>

Back References

(RA1)	The match to <i>RA1</i> will be stored and can be recalled with \1. Several parenthesis (memories) can be set
\n	Recall a stored match to a regular expression. <i>n</i> is an integer

(Character Classes)

[:alnum:]	Any alphanumeric character
[:digit:]	Any number
[:alpha:]	Any alphabetic letter
[:upper:]	Any alphabetic uppercase letter
[:lower:]	Any alphabetic lowercase letter
[:blank:]	Space or tabulator character
[:space:]	Space character
[:punct:]	Punctuation and symbols (., “ ’ ? ! ; :)

(Special Characters)

\n	New line character
\t	Tabulator character
\.	Dot character
*	Star character

11.6 Regular Expression Training

The full power of regular expressions can only be accessed by regular training. There are different tools available that can support you. I wish to present to you two possibilities.

11.6.1 Regular Expressions and *vim*

As stated in the beginning, regular expressions are understood by a broad range of programs. Among these is the text editor *vi*. In Sect. 7.2.2 on p. 88 you learned how to work with this editor. The advanced version of *vi*, i.e., *vim* allows you to highlight text. As you will see soon, this is very useful for training regular expressions.

You remember that there is a *command mode* and an *input mode*. Open a new text document, activate the input mode (*i*), and enter the following line of text “Animals and plants both have mitochondria. In addition, plants possess chloroplasts.” Now go back to the command mode (*Esc*) and type “:set hls”. This activates the *highlight search* mode. Now hit the */* key (on my computer I have to press *Shift*+*7*). At the bottom you will see the “/” character. *vi* is now ready to accept a search pattern. Let us first search for a word. Enter “plants” and hit *Enter*. You will see that the word *plants* is highlighted. Now press */* again and search for the following search pattern: “*pl.\{3\}s*”. Note, it depends on your system, whether you have to escape the braces or not—I have to! After pressing *Enter* you will see text fragments highlighted; they all match our pattern. Now, move the cursor to the end of line. Enter the input mode and type the sentence “At many places people do research.” You will see that word *places* is immediately highlighted. The last active search pattern is still active and queries the text in the background. This feature makes *vi* a good tool for studying and testing regular expressions. Another advantage is that *vi* memorizes previously used search pattern in the same way as the shell memorizes the past commands. After pressing */* you can use the arrow *↑* and *↓* in order to scroll through the past search patterns.

11.7 Regular Expression in Genome Research

Although there are number of specific tools to identify sequence pattern in nucleotide or amino acid sequences, regular expressions still fulfill a role here. One example is the simple query for transcription factor binding sites. The protein NtcA (nitrogen control factor A) regulates nitrogen assimilation in several bacteria. The following DNA binding sites for NtcA have been experimentally validated: TG₉TN₉ACA and TG₁₀TN₁₀ACA, where N stands for any nucleotide and the indices stand for repetitions. You could use


```
egrep 'TGT.{9,10}ACA' genome.file
```

to search for this pattern and print all matching lines. The attribute `-c` would provide you with the number of matching lines. The command

```
egrep -o 'TGT.{9,10}ACA' genome.file | wc -l
```

on the other hand would give the number of matching patterns.

Since pattern analysis is part of everyday routine (not only in molecular biology), tools with extended functionality have been created. `agrep` (approximate grep) for example, allows for mutations (Wu and Manber 1992). `tacg` is a program that has been specifically designed for molecular biologists and is optimized for pattern query in DNA-sequences (Mangalam 2002) (see also Exercise 9.6 on p. 122).

The well-known PROSITE database (database of protein domains, families, and functional sites, <http://www.expasy.org/prosite/>) was originally based on the description of amino acid patterns by regular expressions (Sigrist et al. 2002). A typical PROSITE pattern looks as follows:

```
C-G-G-x(4,7)-G-x(3)-C-x(5)-C-x(3,5)-[NHG]-x-[FYWM]
-x(2)-{GP}-C
```

Every letter represents an amino acid. The “x” stands for any amino acid. Thus, the presented pattern begins with the tri-peptide cysteine-glycine-glycine. Minus characters have no meaning. The string `x(4,7)` stands for 4-7 arbitrary amino acids. The corresponding regular expression would be `.{4,7}`. The string `[NHG]` stands for any single amino acid out of these three—well, again a regular expression.

Exercises

Hey folks—I do not have to tell you any more that you *must* practice. In the previous section you learned some basics of programming. In this chapter, you learned the basics of pattern matching. In the coming sections we are going to fuse our knowledge. However, that is only fun when you are well prepared!

11.1. Go carefully through all the examples in this section and play around. Modify the code and observe the changes.

11.2. Find a regular expression that matches a line with exactly three space-separated fields (words).

11.3. Find a search pattern that matches a negative integer.

11.4. Design a search pattern that matches any decimal number (positive or negative) surrounded by spaces.

11.5. Find a search pattern that matches a nucleotide sequence which begins with the start codon ATG and ends with the stop codon TAA. The sequence should be at least 20 nucleotides long.

11.6. Match all lines that contain the word “hydrogenase” but omit all lines which contain the word “dehydrogenase”.

11.7. A certain class of introns can be recognized by their sequence. The consensus sequences is “*GT...TACTAAC...AG*”. The three dots represent an unknown number of nucleotides. Write a search pattern to match these targets.

11.8. A certain group of proteins has the following consensus sequence: *G(R/T)VQGV GFRx13G(D/W)V(C/N)Nx3G*, where *(R/T)* means either R or T and *x13* stands for a sequence of 13 unspecified amino acids. The letters represent individual amino acids (one-letter amino acid code). Design a regular expression matching this target.

11.9. List all files in your home directory that are readable by all users. Do the same for all files in your home directory and all subdirectories.

Chapter 12

Sed

Sed (**s**tream **e**ditor) is a non-interactive, line-oriented, stream editor. What does that mean?

Non-interactive means that the editor takes its editing commands from either the command line or a file. Once started, it runs through the whole text file that you wish to edit. That sounds worse than it is. Sed does not really edit and change the original text file itself but prints the editing result to the standard output. If you want to keep the result, you just pipe it into a file.

Line-oriented means that Sed treats the input file line by line. A line ends with (and a new line begins after) the (invisible) newline character. This meta character is only visible as a line break when you look at the text file with, for example, `cat` or `Vim`. This has the disadvantage that, as with `egrep`, you cannot edit text that is spanning multiple lines. If you would like to change the words “*gene expression*” to “*transcription*” you can only do that when “*gene expression*” sits in one single line. If the end of one line contained the word “*gene*” and the beginning of the following line contained the word “*expression*”, this would not be recognized. It is, however, possible to delete, insert and change multiple lines.

Stream-oriented means that Sed “swallows” a whole file at once. We say: the file is treated like a stream. Imagine a text file as a pile of sheets of paper. Each sheet represents one line of the text file. Now you take one sheet, edit it according to given rules and put it aside. This you do for all sheets—you cannot stop until you have edited the last sheet. Thus, you edit a stream of sheets—Sed is editing a stream of lines.

Sed is a very old Unix tool. It has its roots in `ed`, a *line editor*. `ed`, however, has been completely replaced by editors like `Vim`. In the next section we will learn why Sed survived.

12.1 When to Use Sed?

Why should I learn how to use a simple non-interactive editor when I know how to use Vim?—you might ask. Indeed, this is a valid question in times when everybody talks about economics. Is it economic to learn Sed? Yes! If you want to bake a cake “from scratch”, you can either mix the dough quickly by hand, or set up the Braun super mixer for the same job. Sed is the “handy” way. It is quick and has very little memory requirements. This is an advantage if you work with large files (some Megabytes). Sed loads one line into the computer’s memory (RAM), edits the line and prints the output either onto the screen or into a file; and Sed is damn fast. I know of no other editor that can compete with Sed. Sed allows for very advanced editing commands. However, Sed is only comfortable to use with small editing tasks. For more advanced tasks one might prefer to use AWK or Perl.

What are typical tasks for Sed? Text substitutions! The most important thing one can do with Sed is the substitution of defined text patterns with other text. Here are some examples: removing *HTML tags* from files, changing the *decimal markers* from commas to points, changing words like *colour* to *color* throughout a text document, removing *comment lines* from shell scripts, or *formatting* text files are only a small number of examples of what can be done with Sed.

For repeatedly occurring tasks you can write *sed scripts* or include *sed* in shell scripts (see Program 26 on p. 155 in Sect. 10.11.4 on p. 155).

12.2 Getting Started

How is Sed used? The basic syntax is quite straightforward. What makes Sed scripts sometimes look complicated are the regular expressions which find the text to edit and the shortcuts used for the commands.

Let us start with a small Sed editing command.

```
Terminal 78: The Substitute Command s/.../.../
1  $ cat>test.file.txt
2  I think most spring flowers bloom white.
3  Is that caused by gene regulation?
4  $ sed 's/most/few/' test.file.txt
5  I think few spring flowers bloom white.
6  Is that caused by gene regulation?
7  $ sed -n 's/most/few/p' test.file.txt
8  I think few spring flowers bloom white.
9  $
```

In Terminal 78 we first create a short text file using `cat` (Sect. 7.1.1 on p. 78). In line 4 you find our first Sed editing command. The command itself is enclosed in single quotes: `s/most/few/`. It tells Sed to substitute the first occurrence of *most* with *few*. The slash functions as a delimiter. Altogether, there are four parts to this substitution command:

<code>s</code>	Substitute command
<code>/.../</code>	Slashes as delimiter
<code>most</code>	Regular expression pattern string
<code>few</code>	Replacement string

We will look at the substitution command in more detail in Sect. 12.5.1 on p. 182. The editing command is enclosed in single quotes in order to avoid that the shell interprets meta characters. After executing the Sed command in line 4 of Terminal 78 on the facing page the output is immediately printed onto the screen. Note: The input file *test.file.txt* remains unchanged.

Now you know 90% of Sed. This is no joke. Most people use Sed for substituting one pattern by another; but be aware that the last 10% are more difficult to achieve than the initial 90%—always! In line 7 of Terminal 78 on the preceding page Sed is started with the option `-n`. In this case Sed works silently and prints out text only if we tell it to do so. We do that with the command `p` (print). Each line that matches the pattern *most* is edited and printed, all other lines are neither edited nor printed onto the screen. The next terminal shows you that the single character `p` is a command.

```

1  $ sed -n 'p' test.file.txt
2  I think most spring flowers bloom white.
3  Is that caused by gene regulation?
4  $

```

In the example in Terminal 79 the option `-n` tells Sed not to print any line from the input file unless it is explicitly stated by the command `p`.

Sometimes one likes to apply more than one editing command. That can be achieved with the option `-e`.

```

1  $ sed -e 's/most/few/' -e 's/few/most/' -e 's/white/red/'
2  test.file.txt
3  I think most spring flowers bloom red.
4  Is that caused by gene regulation?
5  $

```

All commands must be preceded by the option `-e`. The example in Terminal 80 shows how Sed works through the editing commands. After the first line of text has been read by Sed, the first editing command (the one most to the left) is executed. Then the other editing commands are executed one by one. This is the reason why the word *most* remains unchanged: first it is substituted by *few*, then *few* is substituted by *most*. With more and longer editing commands that looks quite ugly. You are better off writing the commands into a file and telling Sed with the option `-f` that the editing commands are in a file.

```

1  $ cat>edit.sed
2  s/most/few/
3  s/few/most/

```

```
4 | s/white/red/
5 | $ sed -f edit.sed test.file.txt
6 | I think most spring flowers bloom red.
7 | Is that caused by gene regulation?
8 | $
```

In Terminal 81 we create a file named *edit.sed* and enter the desired commands. Note: Now there are no single quotes or *e*-options! However, each command must reside in its own line.

A good way to test small Sed editing commands is piping a line of text with *echo* to Sed.

```
_____ Terminal 82: Bash Command echo and Sed _____
1 | $ echo "Darwin meets Newton" | sed 's/meets/never met/'
2 | Darwin never met Newton
3 | $
```

In Terminal 82 the text “*Darwin meets Newton*” is piped to and edited by Sed. With this simple construction one can nicely test small Sed editing scripts.

Now you know 95 % of Sed. Not too bad, is it? Well, the last 5 % are usually even worse to *grep* than the last 10 %. Joking apart, after this first introduction let us now take a closer look at Sed.

12.3 How Sed Works

In order to develop Sed scripts, it is important to understand how Sed works internally. The main thing one must consider is: what does Sed do to a line of text? Well, first the line is copied from the input file and saved in the *pattern space*. This is Sed’s working memory. Then editing commands are executed on the text in the pattern space. Finally, the edited pattern space is sent to the standard output, usually the screen. Furthermore, it is possible to save a line in the *hold space*. Okay, let us examine these “spaces” a bit more closely.

12.3.1 Pattern Space

The pattern space is a kind of working memory where a single text line (also called record) is held, while the editing commands are applied. Initially, the pattern space contains a copy of the first line from the input file. If several editing commands are given, they will be applied *one after another* to the text in the pattern space. This is why we can change back a previous change as shown in Terminal 80 on p. 177 (most → few → most). When all the instructions have been applied, the current line is moved to the standard output and the next line from the input file is copied into the pattern space. Then all editing commands are applied to that line and so forth.

12.3.2 Hold Space

While the pattern space is a memory that contains the current input line, the hold space is a temporary memory. In fact, you can regard it as the memory key of a calculator. You can put a copy from the pattern space into the hold space and recall it later. A group of commands allow you to move text lines between the pattern and the hold space:

<code>h</code>	<i>hold-O</i>	Overwrites the hold space with the contents of the pattern space.
<code>H</code>	<i>hold-A</i>	Appends a newline character followed by the pattern space to the hold space.
<code>g</code>	<i>get-O</i>	Overwrites the pattern space with contents of hold space.
<code>G</code>	<i>get-A</i>	Appends a newline character followed by the hold space to the pattern space.
<code>x</code>	<i>exchange</i>	Swaps the contents of the hold space and the pattern space.

You might like to play around with the hold space once you have gotten accustomed to Sed. In the example in Sect. 12.6.3 on p. 192, space is used in order to reverse the lines of a file.

12.4 Sed Syntax

The basic way to initiate Sed is

```
sed 'EditCommand' Input.File
```

This line causes Sed to edit the *Input.File* line by line according to the job specified by *EditCommand*. As shown before, if you want to apply many edit commands sequentially, you have to apply the option `-e`:

```
sed -e 'Edit.Command.1' -e 'Edit.Command.2' Input.File
```

Sometimes you wish to save your edit commands in a file. If you do this, Sed can execute the commands from the file (which we would call a script file). Assume the script file's name is *EditCommand.File*. Then you call Sed with

```
sed -f EditCommand.File Input.File
```

The option `-f` tells Sed to read the editing commands from the file stated next. Instead of reading an input file, Sed can also read the standard input. This can be useful if you want to pipe the output of one command to Sed. We have already seen one example in conjunction with the `echo` command in Terminal 82 on p. 178. A more applied example with `ls` is shown in Terminal 85 on p. 184.

12.4.1 Addresses

As I said above, Sed applies its editing commands to each line of the input file. With addresses you can restrict the range to which editing commands are applied. The complete syntax for Sed becomes

```
sed 'AddressEditCommand' Input.File
```

Note that there is no space between the address and the editing command. There are several ways to describe an address or even a range:

- a *a* must be an integer describing a line number.
- \$ The dollar character symbolizes the last line.
- /re/ *re* stands for a regular expression, which must be enclosed in slashes.
Each line containing a match for *re* is chosen.
- a,b Describes the range of lines from line *a* to line *b*. Note that *a* and *b*
could also be regular expressions or \$. All kinds of combinations are possible.
- x! The exclamation mark negates the address range.
x can be any of the addresses above.

Let us take a look at some examples. We do not edit anything, but just print the text lines to which the addresses match. Therefore, we run Sed with the option `-n`. Sed's command to print a line of text onto the screen is `p` (see Sect. 12.5.6 on p. 189). As an example file let us use the protein structure file *structure.pdb* (File 4 on p. 161). In order to display the whole file content we use

```
sed -n 'p' structure.pdb
```

Do you recognize that this command resembles “`cat structure.pdb`”? What would happen if we omitted the option `-n`? Then all lines would be printed twice! Without option `-n` Sed prints every line by default *and* additionally prints all lines it is told to by the print command `p`—ergo, each line is printed twice.

```

Terminal 83: Addresses
1  $ sed -n '1p' structure.pdb
2  HEADER Hydrogenase 23-Mar-99 1CFZ
3  $ sed -n '2p' structure.pdb
4  COMPND Hydrogenase Maturing Endopeptidase Hybd From
5  $ sed -n '1,3p' structure.pdb
6  HEADER Hydrogenase 23-Mar-99 1CFZ
7  COMPND Hydrogenase Maturing Endopeptidase Hybd From
8  SOURCE ORGANISM_SCIENTIFIC: Escherichia coli
9  $ sed -n '/ATOM 7/, $p' structure.pdb
10 ATOM 7 O MET A 1 47.875 25.011 52.020 1.00 54.99 O
11 ATOM 8 SD MET A 1 49.731 23.948 47.163 1.00 77.15 S
12
13 $ sed -n '/HELIX/p' structure.pdb
14 HELIX 1 hel ILE A 18 GLN A 28
15 HELIX 2 hel PRO A 92 THR A 107
16 HELIX 3 hel ILE A 138 SER A 152

```



```

17 $ sed -n '/HEADER/,/AUTHOR/p' structure.pdb
18 HEADER Hydrogenase 23-Mar-99 1CFZ
19 COMPND Hydrogenase Maturing Endopeptidase Hybd From
20 SOURCE ORGANISM_SCIENTIFIC: Escherichia coli
21 AUTHOR Fritsche, Paschos, Beisel, Boeck & Huber
22 $ sed -n '/HEADER\|AUTHOR/p' structure.pdb
23 HEADER Hydrogenase 23-Mar-99 1CFZ
24 AUTHOR Fritsche, Paschos, Beisel, Boeck & Huber

```

Terminal 83 gives you a couple of examples how to extract certain lines from File 4 on p. 161. Lines 1–8 show examples how to select and display certain lines or line ranges. The address used in line 9 makes Sed print all lines from the first occurrence of “*ATOM 7*” to the last line (which is, in our case, an empty line). The address */HELIX/* in line 13 instructs Sed to print all lines containing the word *HELIX*. In line 22 Sed selects all lines containing either *HEADER* or *AUTHOR*. Note that the logical *or* represented by the vertical bar needs to be escaped (`\|`).

It is important that you understand that the address *selects* the lines of text that are treated by Sed. In Terminal 83 the treatment consisted simply of printing. The treatment could as well have been an editing statement. For example,

```
sed '/^ATOM/s/.*/--deleted--/' structure.pdb
```

would select all lines starting with *ATOM* (*/^ATOM/*) for editing. The editing instruction (*s/.*/--deleted--/*) selects all the content of the selected lines (*.**) and substitutes it with the text “*--deleted--*”. Try it out!

12.4.2 Sed and Regular Expressions

Of course, Sed works fine with regular expressions. However, in contrast to the meta characters we used with *egrep* in Chap. 11 on p. 157, there are some syntactical differences: Parentheses “*()*”, braces “*{ }*” and the vertical bar “*|*” must be escaped with a backslash! You remember that parentheses group commands (see Sect. 11.4.3 on p. 165) and save the target pattern for back referencing (see Sect. 11.4.7 on p. 165), whereas braces belong to the quantifiers (see Sect. 11.4.2 on p. 164).

12.5 Commands

Up to now, we have already encountered some vital Sed commands. We came across substitutions with `'s/.../.../` and know how to explicitly print lines by the combination of the option `-n` and the command `p`. We also learned that the commands `g`, `G`, `h`, `H` and `x` manipulate the pattern space and hold space. Let us now take a tour through the most important Sed commands and their application. For some of the following examples we are going to use a new text file, called *GeneList.txt* (File 6).

```

1 Energy metabolism
2 Glycolysis
3   slr0884: glyceraldehyde 3-phosphate dehydrogenase (gap1)
4   Init: 1147034 Term: 1148098 Length (aa): 354
5   slr0952: fructose-1,6-bisphosphatase (fbpII)
6   Init: 2022028 Term: 2023071 Length (aa): 347
7 Photosynthesis and respiration
8 CO2 fixation
9   slr0009: ribulose biphosphate carboxylase large (rbcL)
10  Init: 2478414 Term: 2479826 Length (aa): 470
11  slr0012: ribulose biphosphate carboxylase small (rbcS)
12  Init: 2480477 Term: 2480818 Length (aa): 113
13 Photosystem I
14  slr0737: photosystem I subunit II (psaD)
15  Init: 126639 Term: 127064 Length (aa): 141
16  ssr0390: photosystem I subunit X (psaK)
17  Init: 156391 Term: 156651 Length (aa): 86
18  ssr2831: photosystem I subunit IV (psaE)
19  Init: 1982049 Term: 1982273 Length (aa): 74
20 Soluble electron carriers
21  sll0199: plastocyanin (petE)
22  Init: 2526207 Term: 2525827 Length (aa): 126
23  sll0248: flavodoxin (isiB)
24  Init: 1517171 Term: 1516659 Length (aa): 170
25  sll1796: cytochrome c553 (petJ)
26  Init: 846328 Term: 845966 Length (aa): 120
27  ssl0020: ferredoxin I (petF)
28  Init: 2485183 Term: 2484890 Length (aa): 97

```

This file contains information about the genome of the cyanobacterium *Synechocystis* PCC 6803. It was sequenced in 1996. The file is a very, very small exemplified assembly of annotated genes (gene names are given in the unambiguous *Cyanobase Format*, like *slr0884*, and, in parentheses, with their scientific specification), their position on the chromosome (*Init* and *Term*), the predicted length of the gene products (aa = amino acids) and their function. The file contains two major functional classes, *energy metabolism* and *photosynthesis and respiration*, respectively. These major classes are divided into subclasses, such as *glycolysis*, *CO₂ fixation*, *photosystem I* and *soluble electron carriers*.

12.5.1 Substitutions

Substitutions are by far the most common, and probably most useful, editing actions Sed is used for. An example of a simple substitution is the transformation of all decimal markers from commas to points. The basic syntax of the substitution command is

```
s/pattern/replacement/flags
```

The *pattern* is a regular expression describing the target that is to be substituted by *replacement*. The *flags* can be used to modify the action of the substitution command in one or the other way. Important flags are:

- g** Make the changes globally on all occurrences of the pattern. Normally only the first occurrence is edited.
- n** The replacement should be made only for the *n*th occurrence of the target pattern.
- p** In conjunction with the option **-n**, only matched and edited lines (pattern space) are printed.
- w file** Same as **p**, but writes the pattern space into a file named *file*.

The flags can be used in combinations. For example, the flag **gp** would make the substitution globally on the line and print the line.

The *replacement* can make use of *back referencing* (see Sect. 11.4.7 on p. 168). Furthermore, the ampersand (&) stands for (and is replaced by) the string that matched the complete regular expression.

```

Terminal 84: Back References
1  $ sed -n 's/^  [A-Z].*$/>>>&/p' GeneList.txt
2  >>> Glycolysis
3  >>> CO2 fixation
4  >>> Photosystem I
5  >>> Soluble electron carriers
6  $ sed -n 's/^  \([A-Z].*\)/--- \1 ---/p' GeneList.txt
7  --- Glycolysis ---
8  --- CO2 fixation ---
9  --- Photosystem I ---
10 --- Soluble electron carriers ---
11 $

```

In Terminal 84 we see two examples of back references. In both cases we use the combination of the option **-n** and the flag **p** to display modified lines only. The target match for the search pattern “`^ [A-Z].*$`” is any line that begins with three space characters (here symbolized by “”), continues with one uppercase alphanumeric character and continues with any or no character up to the end of the line. This search pattern exactly matches the subclasses in *GeneList.txt* (File 6 on p. 182). In line 1 of Terminal 84 the matching pattern is substituted by “`>>>`” plus the whole matching pattern, which is represented by the ampersand character (&). In line 6 we embedded only the last part of the search pattern in parentheses (note that the parentheses need to be escaped) and excluded the three spaces. The matching pattern can then be recalled with “`\1`”.

In order to restrict the range of the editing action to the first 6 lines of File 6 on p. 182, line 6 of Terminal 84 could be modified to

```
sed -n '1,6s/^  \([A-Z].*\)/--- \1 ---/p' GeneList.txt
```

Now let us assume we want to delete the text parts describing the position of genes on the chromosome. The correct command would be

```
sed 's/Init.*Term:[0-9]*_/ /' GeneList.txt
```

Here, we substitute the target string with an empty string. Try it out in order to see the result.

The next example is a more practical one. In order to see the effect of the Sed command in Terminal 85 you must have subdirectories in your home directory.

```

1  $ ls -l ~ | sed 's/^d/DIR: /' | sed 's/^[^Dt]/ /'
2  total 37472
3      rw-rw-r-- 1 Freddy Freddy    30 May  5 16:29 Datum2
4      rw-rw-r-- 1 Freddy Freddy    57 May 11 19:00 amino2s
5  DIR: rwxrwxr-x 3 Freddy Freddy  4096 May  1 20:04 blast
6  DIR: rwxrwxr-x 3 Freddy Freddy  4096 May  1 20:34 clustal
7      rw-rw-r-- 1 Freddy Freddy 102400 Apr 19 15:55 dat.tar
8  $

```

Terminal 85 shows how to use Sed in order to format the output of the `ls` command. By applying the editing commands given in line 1, the presence of directories immediately jumps to the eye. With “`ls -l ~`” we display the content of the home directory [Remember that the tilde character (`~`) is the shell shortcut for the path of your home directory]. In the first Sed editing command (`s/^d/DIR: /`) we substitute all occurrences of `d` at the beginning of a line (^) with “`DIR:` ”. The result is piped to a second instance of Sed which introduces spaces at the beginning of each line *not* starting with a “`D`” (as in “`DIR`”) or “`t`” (as in “`total...`”). With our knowledge about writing shell scripts and the `alias` function (see Sect. 8.8 on p. 104), we could now create a new standard output for the `ls` command.

```

1  $ cat list-dir.sh
2  #!/bin/bash
3  # save as list-DIR.sh
4  # reformats the output of ls -l
5  echo "`ls -l $1|sed 's/^d/DIR: /'|sed 's/^[^Dt]/ /'"`

```

An appropriate shell script is shown in Program 27. The shell script is executed with

```
./list-DIR.sh DirName
```

from the directory where you saved it. The parameter *DirName* specifies the directory of which you wish to list the content. How does the script work? We basically put the command line from Terminal 85 into a shell script and execute the command from within the program by enclosing it in graves (see Sect. 10.5.2 on p. 136). The parameter *DirName* can be accessed via the variable `$1`.

12.5.2 Transliterations

You have got a nice sequence file from a colleague. It contains important sequence data, which you want to process with a special program. The stupid thing is that the sequences are lowercase RNA sequences. However, you require uppercase DNA sequences. Hmmm—what you need is Sed’s transliteration command “`y`”.

```

Terminal 86: Transliterations with y/.../.../
1  $ cat>rna.seq
2  >seq-a
3  acgcguaauuagcgcaugcgaaauaucgcuauuacg
4  >seq-b
5  uagcgcuauuagcgcgcuagcuaggaucaucgacgcg
6  $ sed '/>/!y/acgu/ACGT/' rna.seq
7  >seq-a
8  ACGCGTATTTAGCGCATGCGAATATCGCTATTACG
9  >seq-b
10 TAGCGCTATTAGCGCGCTAGCTAGGATCGATCGCG
11 $

```

Terminal 86 shows a quick solution to the problem. In lines 1–5 we create our example RNA sequence file in *FASTA format*. The magic command follows in line 6. First we define the address (`/>/!`): all lines *not* containing the greater character are to be edited. The editing command invoked by “`y/acgu/ACGT/`” transliterates all “a”s to “A”s and so on. The transliteration command “y” does not understand words. The first character on the left side (a) is transliterated into the first character on the right side (A) and so on. It always affects the whole line.

Now let us generate the complement of the sequences in the file *rna.seq*. The correct comment is

```
sed '/>/!y/acgu/ugca/' rna.seq
```

Could you achieve the same result with the substitution command from Sect. 12.5.1 on p. 182? If you think so and found a solution, please email it to me.

12.5.3 Deletions

We have used already the substitution command to delete some text. Now we will delete a whole line. Assume you want to keep only the classes and subclasses but not the gene information in File 6 on p. 182 (*GeneList.txt*). This means we need to delete these lines. The corresponding command is `d` (delete), the syntax is

```
addressd
```

The `d` immediately follows the address. The address can be specified as described in Sect. 12.4.1 on p. 180. If no address is supplied, all lines will be deleted—that sounds not too clever, doesn’t it? The important thing to remember is that the whole line matching the address is deleted, not just the match itself. Let us come back to our task: extracting the classes and subclasses of *GeneList.txt* (File 6 on p. 182).

```

1 $ sed '/ /d' GeneList.txt
2 Energy metabolism
3 Glycolysis
4 Photosynthesis and respiration
5 CO2 fixation
6 Photosystem I
7 Soluble electron carriers
8 $

```

Since all gene information lines are indented by 5 space characters, these 5 spaces form a nice address pattern. Thus, the task is solved easily, as shown in Terminal 87.

12.5.4 Insertions and Changes

Another common editing demand is to insert, append, change or delete text lines. Note that only whole lines and no words or other fragments can be inserted, appended or changed. These commands require an address. The corresponding commands are

- a *append* Append a text after the matching line.
- i *insert* Insert a text before the matching line.
- c *change* Change the matching line.

Append and *insert* can only deal with a single line address. However, for *change*, the address can define a range. In that case, the whole range is changed. In contrast to all other commands we have encountered up to now, a, i and c are multiple-line commands. What does this mean? Well, they require a line break. For example, the syntax for the append command a is

```

addressa\
text

```

As you can see, the command is followed by a backslash without any space character between the address and the command! That is true for all three commands. Then follows the text that is to be appended after all lines matching the address. To insert multiple lines of text, each successive line must end with a backslash, except the very last text line.

```

1 $ sed '/>/i\
2 > -----' rna.seq
3 -----
4 >seq-1
5 acgcguauuuagcgcgaugcgaaauaucgcuaauacg
6 -----
7 >seq-2
8 uagcgcuaauagcgcgcguagcuaggaucaucgcgcg
9 $

```

Terminal 88 gives an example of an insertion. The trick with multiple-line commands is that the shell recognizes unclosed quotes. Thus, when you press **Enter** after entering line 1 of Terminal 88 the greater character (>) appears and you are supposed to continue your input. In fact, you can enter more lines, until you type the single quote again. This we do in line 2: after our insertion text (a number of dashes) we close Sed's command with the closing single quote followed by the filename. In this example we use the file we created in Terminal 86 on p. 185. What does the command spanning from line 1 to line 2 do? Before every line beginning with a greater character some dashes are inserted.

We can do fancy things with multiple-line commands.

```

Terminal 89: Deletions and Insertions
1  $ sed '/ / /d
2  > /^[A-Z]/a\
3  > =====
4  > / /a\
5  > -----
6  > ' GeneList.txt
7  Energy metabolism
8  =====
9  Glycolysis
10 -----
11 Photosynthesis and respiration
12 =====
13 CO2 fixation
14 -----
15 Photosystem I
16 -----
17 Soluble electron carriers
18 -----
19 $

```

Terminal 89 gives you an idea of how several commands can be used with one call of Sed. Line 1 contains the editing commands to delete all lines containing five successive space characters. This matches the gene-information lines of our example File 6 on p. 182 (*GeneList.txt*). The address in line 2 matches all main classes. A line containing equal characters (=) is appended to this match. Then, to all lines containing three consecutive space characters, a line with dashes is appended. The editing action is executed after hitting **Enter** in line 6. What follows in Terminal 89 is the resulting output.

Let us finally take a quick look at the *change* command.

```

Terminal 90: Changing Text Blocks
1  $ sed '/Photo/, $c\
2  > stuff deleted' GeneList.txt
3  Energy metabolism
4  Glycolysis
5      slr0884: glyceraldehyde 3-P dehydrogenase (gap1)
6      Init: 1147034 Term: 1148098 Length (aa): 354
7      slr0952: fructose-1,6-bisphosphatase (fbpII)
8      Init: 2022028 Term: 2023071 Length (aa): 347
9  stuff deleted
10 $

```

In the example shown in Terminal 90 all lines between the first occurrence of “Photo” and the last line is changed to (substituted by) the text “stuff deleted”.

12.5.5 Sed Script Files

If you need to apply a number of editing commands several times, you might prefer to save them in a file.

```

_____ Program 28: script1.sed - Sed File _____
1  # save as script1.sed
2  # Formatting of GeneList.txt
3  /      /d
4  /^ [A-Z] /a\
5  =====
6  /      /a\
7  -----

```

The script file containing the commands used in Terminal 89 on the preceding page is shown in Program 28. Lines 1 and 2 contain remarks that are ignored by Sed. You would call this script by

```
sed -f script1.sed GeneList.txt
```

Maybe you are using the script very, very often. Then it makes sense to create an executable script.

```

_____ Program 29: script2.sed - Sed Executable _____
1  sed '
2  # save as script2.sed
3  # Provide filename at the command line
4  /      /d /^ [A-Z] /a\
5  =====
6  /      /a\
7  -----
8  ' $*

```

Program 29 would be the result of this approach. Do not forget to make the file executable with “`chmod u+x script2.sed`”. Note that all commands are enclosed in single quotes. Do you recognize the variable “\$*”? Take a look at Sect. 10.4.4 on p. 133. – It contains all command line parameters. In our case these would be the files that are to be edited—yes, you can edit several files at once. In order to execute the same task as shown in Terminal 89 you type

```
./script2.sed GeneList.txt
```

This requires much less typing, especially if you create an *alias* for the program (see Sect. 8.8 on p. 104)!

12.5.6 Printing

We have already used the printing command `p` a couple of times. Still, I will shortly mention it here. Unless the default output of Sed is suppressed (`-n`), the print command will cause duplicate copies of the line to be printed. The print command can be very useful for debugging purposes.

```

1  $ sed '/>/p
2  > s/>/</' rna.seq
3  >seq-1
4  <seq-1
5  acgcguaauuagcgcaugcgaaauauggcuauuacg
6  >seq-2
7  <seq-2
8  uagcgcuauuagcgcgcuagcuaggaucaugcgcg
9  $

```

Terminal 91: Debugging

Terminal 91 shows how to use `p` for debugging. In line 1 the current line in the pattern space is printed. Line 2 modifies the pattern space and prints it out again. Thus, we see the line before and after editing and might detect editing errors.

A special case of printing provides the equal character (`=`). It prints the line number of the matching line.

```

1  $ sed -n '/^[A-Z]/=' GeneList.txt
2  1
3  7
4  $

```

Terminal 92: Printing Line

The editing command in Terminal 92 prints the line number of lines containing major classes in *GeneList.txt* (File 6 on p. 182).

12.5.7 Reading and Writing Files

The read (`r`) and write (`w`) commands allow you to work directly with files. This can be very comfortable when working with script files. Both commands need a single argument: the name of the file to read from or write to, respectively. There must be a single-space character between the command and the filename. At the end of the filename, either a new line must start (in script files) or the script must end with the single quote (as in Terminal 93).

```

1  $ sed -n '/^[A-Z].*/w output.txt' GeneList.txt
2  $ cat output.txt
3  Glycolysis
4  CO2 fixation
5  Photosystem I
6  Soluble electron carriers
7  $

```

Terminal 93: Writing into File

The editing command in Terminal 93 writes the subclasses from File 6 on p. 182 into the file *output.txt*.

```
Terminal 94: Reading a File
1  $ cat>input.txt
2      sub classes...
3  $ sed '/^[A-Z]/r input.txt
4  > / /d' GeneList.txt
5  Energy metabolism
6      sub classes...
7  Photosynthesis and respiration
8      sub classes...
9  $
```

In Terminal 94 we first create an input file called *input.txt*. In line 3 the file *input.txt* is inserted after each occurrence of a main class in File 6 on p. 182. Note that there must be exactly one space character between “r” and the filename, and that there must not be anything behind the filename except a single quote or a new line. In line 4 all other than the main class lines are deleted.

The read command will not complain if the file it should read does not exist. The write command will create a non-existing file and overwrite an existing file. However, write commands *within* one single script will append to the target file!

12.5.8 Advanced Sed

As you can imagine, there are many more possibilities for what you can do with Sed. There are some more advanced commands that allow you to work with multiple-line pattern spaces, make use of the hold space or introduce conditional branches. However, I will not go into details here. These commands require quite some determination to master and are pretty difficult to learn. In my opinion, what you have learned up to this point is by far enough to have fun with Sed. All the advanced stuff starts to become a pain in the neck. Therefore, we save our energy in order to learn how to use AWK and Perl for these more advanced editing tasks. Anyway, if you feel Sed is just the scripting language you have been waiting for, you should read some special literature in either English (Dougherty and Robbins 1997) or German (Herold 2003).

12.6 Examples

You should keep up the habit of trying out some example scripts. Do some modification and follow the changes in the output.

12.6.1 Gene Tree

The following examples reformats the content of File 6 on p. 182 (*GeneList.txt*) into a tree-like output. This facilitates easy reading of the data.

```

Terminal 95: Creating a Gene Tree-View
1  $ sed -n 's/^\([A-Z]*\)/|--- \1/p
2  s/^\([A-Z]*\)/| |--- \1/p
3  s/^\([A-Za-z]*\(\(...\)\)/| | |--- \1/p
4  ' GeneList.txt
5  |--- Energy metabolism
6  | |--- Glycolysis
7  | | |--- gap1
8  | |--- Photosynthesis and respiration
9  | | |--- CO2 fixation
10 | | | |--- rbcL
11 | | | |--- rbcS
12 | | | |--- Photosystem I
13 | | | |--- psaD
14 | | | |--- psaK
15 | | | |--- psaE
16 | | |--- Soluble electron carriers
17 | | |--- petE
18 | | |--- isiB
19 | | |--- petJ
20 | | |--- petF
21 $

```

The Sed editing commands in Terminal 95 largely resemble those in Terminal 84 on p. 183. A speciality resides in line 3 of Terminal 95: here we use the back reference in order to grep the content of the parentheses, that is gene name. You might have realized that the script makes use of the `-n` option and `p` command combination. This makes it easy to exclude the lines starting with “Init...”.

12.6.2 File Tree

Since we started with trees, why not writing a scripting in order to print the directory content in a tree-like fashion?

```

Program 30: tree.sed - Directory Tree
1  #!/bin/bash
2  # save as tree.sed
3  # requires path as command line parameter
4  if [ $# -ne 1 ]; then
5      echo "Provide one directory name next time!"
6      echo
7  else
8      find $1 -print 2>/dev/null |
9      sed -e 's/[^\s]*\s/|--- /g' -e 's/--- |/' -e 's/--- |/' -e 's/--- |/'
10     fi

```

What does Program 30 do? Well, first you should recognize that it is a shell script. In line 4 we check whether exactly one command line parameter has been supplied. If this is not the case, the message “Provide one directory name next time!”

is displayed and the program stops. Note: It is always a good idea to make a script *fool-proof* and inform the user about the correct syntax when he applies the wrong syntax. Otherwise, the `find` command is invoked with the directory name provided at the command line ($\$1$). The `find` option `-print` prints the full filename, followed by a new line, on the standard output. Error messages are redirected into the nirvana (`2>/dev/null`, see Sect. 8.5 on p. 101). The standard output, however, is not printed onto the screen but redirected (`()`) to Sed. In order to understand what Sed is doing, I recommend you to look at the output of `find` and apply the search pattern `"[^\|/]*\|/"` to it. How? Use Vim! In Sect. 7.2.8 on p. 92. we saw that we can read external data into an open editing file. If you type

```
:r! find . -print
```

in the command modus (and then hit `(Enter)`), the output of `"find . print"` is imported into Vim. Then you can play around with regular expressions and see what the search pattern is doing, as described in Sect. 11.6.1 on p. 172.

12.6.3 Reversing Line Order

Assume you have a file with lists of parameters in different lines. For reasons I do not know, you want to reverse the order of the file content. Ahh, I remember the reason: in order to exercise!

```

Terminal 96: Reversing Lines
1  $ cat > parameterfile.txt
2  Parameter 1 = 0.233
3  Parameter 2 = 3.899
4  Parameter 3 = 2.230
5  $ sed '1!G
6  > h
7  > $!d' parameterfile.txt
8  Parameter 3 = 2.230
9  Parameter 2 = 3.899
10 Parameter 1 = 0.233
11 $

```

In lines 1–4 of Terminal 96 we create the assumed parameter file, named *parameterfile.txt*. The Sed script consists of three separate commands: `"1!G"`, `"h"` and `"$!d"`. You see, herewe make use of the hold space, as explained in Sect. 12.3.2 on p. 179. The first command in line 5 is applied to all but the first line (`1!`), whereas the third command in line 7 is applied to all but the last line (`$!`). The second command is executed for all lines. When we execute the Sed script on the text file *parameterfile.txt*, the first command that is executed is `h`. This tells Sed to copy the contents of the pattern space (the buffer that holds the current line being worked on) to the hold space (the temporary buffer). Then, the `d` command is executed, which deletes the current line from the pattern space. Next, line 2 is read into the pattern space and the command `G` is executed. `G` appends the contents of the hold space (the previous line)

to the pattern space (the current line). The `h` command puts the pattern space, now holding the first two lines in reverse order, back to the hold space for safe keeping. Again, `d` deletes the line from the pattern space so that it is not displayed. Finally, for the last line, the same steps are repeated, except that the content of the pattern space is not deleted (due to `!`) before the `d`). Thus, the contents of the pattern space is printed to the standard output.

Note that you could execute the script also with

```
sed '1!G;h;$!d' parameterfile.txt
```

In this case, the commands are separated by the semicolon instead of the newline character.

Exercises

Now let us see what you have learned. All exercises are based on the File 4 on p. 161 (*structure.pdb*).

12.1 Change Beisel's name to Weisel.

12.2 Delete the first three lines of the file.

12.3 Print only lines 5 through 10.

12.4 Delete lines containing the word MET.

12.5 Print all lines where the HELIX line contains the word ILE.

12.6 Append three stars to the end of lines starting with "H".

12.7 Replace the line containing SEQRES with SEQ.

12.8 Create a file with text and blank lines. Then delete all blank lines of that file.

Part IV

Programming

Chapter 13

AWK

Awkay, are you ready to learn a real programming language? AWK is a text-editing tool that was developed by Aho, Weinberger, and Kernighan in the late 1970s. Since then, AWK has been largely improved and is now a complete and powerful programming language. The appealing thing for scientists is that AWK is easy enough to be learned quickly and powerful enough to execute most relevant data analysis and transformation tasks. Thus, AWK must not be awkward. AWK can be used to do calculations as well as to edit text. Things that cannot be done with Sed can be done with AWK. Things you cannot do with AWK you can do with Perl. Things you cannot do with Perl you should ask a computer scientist to do for you.

While you work through this chapter, you will notice that features similar to those we saw in the chapter on shell programming (see Chap. 10 on p. 125) reappear: variables, flow control, and input–output. This is typical. All programming languages have more or less the same ingredients but different syntax. On the one hand this is pain in the neck, on the other hand it is very convenient. Because if you have learned one programming language thoroughly, then you can basically work with all programming languages. You just need to cope with the different syntax; and syntax is something you can look up in a book on the particular programming language. What programming really is about is logics. First, understand the structure of the problem you want to solve, or the task you want to execute, then transfer this structure into a program. This is the joy of programming. This said, *if* you already know how to program, *then* focus on the syntax, or *else* have fun learning the basics of a programming language in this chapter. Did you recognize the italic words in the last sentence? There, we just used the control element *if-then-else*, which is part of every programming language.

As you might have guessed, there are several different versions of AWK available. Thus, if you encounter any problems with the exercises in this chapter, the reason might be an old AWK version on your system. It is also possible that not AWK but Gawk is installed on your system. Gawk is the freeware GNU version of AWK. In that case you would have to type `gawk` instead of `awk`. You can download the newest version at rpmseek.com.

13.1 Getting Started

Like Sed, AWK treats an input text file line by line. However, every line is splitted into fields that are separated by a space character (default) or any other definable delimiter. Each field is assigned to a variable: the whole line is stored in `$0`, the first field in `$1`, the second field in `$2`, and so on.

```
Watson and Crick are smart scientists, aren't they?
```

```
-----
$1      $2      $3      $4      $5      $6      $7      $8      = $0
```

Via the variables you have access to each field and can modify it, do calculations or whatever you want to do. As with Sed, you can search for text patterns with regular expressions and execute editing commands or other actions. Furthermore, you can write AWK scripts using variables, loops, and conditional statements. Finally, you can define your own functions and use them as if they were built-in functions.

As an appetizer, let us take a look at an example.

```

1  $ cat>enzyme.txt
2  Enzyme      Km
3  Protease    2.5
4  Hydrolase   0.4
5  ATPase      1.2
6  $ awk '$2 < 1 {print $1}' enzyme.txt
7  Hydrolase
8  $
```

In Terminal 97, we first generate a new text file called *enzyme.txt*. This file contains a table with three enzymes and their catalytic activity (*Km*). In line 6 we run an AWK statement. The statement itself is enclosed in single quotes. The input file is given at the end. The statement can be divided into two parts: a *pattern* and an *action*. The action is always enclosed in braces (`{ }`). In our example the pattern is “`$2 < 1`”. You can read this as: “if the value in the variable `$2` is smaller than 1, then do”. The content of variable `$2` is the value of field 2. Since the default field separator is the space character, field 2 corresponds to the second column of our table, the *Km value*. Only in line 4 of Terminal 97 “Hydrolase 0.4” is the *Km value* smaller than 1. Thus, the pattern matches this line and the action will be executed. In our example the action is “`{print $1}`”. This can be read as: “print the content of variable `$1`”. The variable `$1` contains the value of field 1. In the matching line this is “Hydrolase”. Therefore, field 1 is printed in line 7 of Terminal 97.

The first example gave you a first insight into the function of AWK. After starting AWK, the lines of the input file are read one by one. If the pattern matches, then the action will be executed. Otherwise, the next line will be read. In the following section we learn more about AWK’s syntax.

13.2 AWK's Syntax

As stated before, AWK reads an input file line by line and executes an action on lines that match a pattern. The basic syntax is

```
awk 'pattern action' InputFile
```

In our example in Terminal 97 the pattern was “\$2 < 1”. This is a so-called *relational expression* pattern. However, as you will see later, the pattern could as well be a *regular expression*, a *range* pattern, or a *pattern-matching expression*. The action is always enclosed in braces. In Terminal 97 the action was “print \$1”. With this action or command, the content of the first field is printed. As you can imagine, there are many other actions that can be taken and we will discuss the most important ones later. The whole AWK statement, consisting of pattern and action, must be enclosed in single quotes. As usual, the input filename follows the statement, separated by a space character. You could treat more than one file at once by giving several filenames. AWK's editing statement must consist of either a pattern, an action or both. If you omit to give a pattern, then the default pattern is employed. The *default pattern* is matching every line. Thus

```
awk '{print $0}' InputFile
```

would print every line of the file named *InputFile*. The variable *\$0* represents the whole currently active line of the input file. If you omit the action, the *default action* is printing the whole line. Thus

```
awk '/.*/' InputFile
```

prints all lines of *InputFile*, as well. In this case, we use the regular expression */.** as pattern, which matches every line.

AWK comes with a great number of options. There are two important options you must remember:

- F "x" Determines the field separator. The default setting is the space character. If you want to use another field separator you must use this option and replace *x* with your field delimiter. *x* can consist of several characters.
- f Tells AWK to read the commands from the file given after the -f option (separated by a space character).

The *default field separator* is the space character. You can change the field separator by supplying it immediately after the -F option, enclosed in double quotes.

```

Terminal 98: Option -F
1 $ awk -F":" ' /Freddy/ {print $0}' /etc/passwd
2 Freddy:x:502:502::/home/Freddy:/bin/bash
3 $ awk -F":" ' /Freddy/ {print $1 " uses " $7}' /etc/passwd
4 Freddy uses /bin/bash
5 $

```

In line 1 of Terminal 98, we print the complete line of the system file */etc/passwd* that matches the regular expression */Freddy/*. You remember that Freddy is a username. The field separator is set to be the colon character (:). In line 3 we use AWK to print the path to the shell executable, which the user Freddy is using. Again, we set the field separator to be the colon character (*-F " : "*). For the lines containing the pattern *Freddy* (*/Freddy/*) we employ the print action. Here, the content of fields 1 and 7, represented by the variables *\$1* and *\$7*, respectively, separated by some text that must be enclosed in double quotes (“uses.”), is printed.

With AWK you can write very complex programs. Usually, programs are saved in a file that can be called by the option *-f*.

```

Terminal 99: Option -f
1 $ cat>user-shell.awk
2 /Freddy/ {print $1 " uses " $7}
3 $ awk -F":" -f user-shell.awk /etc/passwd
4 Freddy uses /bin/bash
5 $

```

In lines 1 and 2 of Terminal 99 we write the script file *user-shell.awk*. In line 3 you see how you can call a script file with the option *-f*. As with shell scripts or Sed script files, AWK script files can be commented. Comments always start with the hash character (#).

13.3 Example File

In Sect. 7.1.1 on p. 78 we created a text file with some genomic information (see File 1 on p. 79). At certain points in this section, we are going to recycle this file, which we called *genomes.txt*. Check if you still have the file

```
find ~ -name "genomes.txt"
```

(Remember that “*~*” is the shell’s shortcut for the path to your home directory.) Before we use this file for calculations, we perform two modifications. First, we should erase the three last lines and second, we must remove the comma delimiter in the numbers. How about a little Sed statement?

```

Terminal 100: Modifying genomes.txt
1  $ sed -n '/\?!s/,//gp' genomes.txt
2  H. sapiens (human) - 3400000000 bp - 30000 genes
3  A. thaliana (plant) - 100000000 bp - 25000 genes
4  S. cerevisiae (yeast) - 12100000 bp - 6034 genes
5  E. coli (bacteria) - 4670000 bp - 3237 genes
6  $

```

Terminal 100 gives you a little Sed update. We edit all lines not containing a question mark (`/\?!s/,//gp`—note that the question mark must be escaped because it is a meta character) and substitute globally the comma by nothing (`s/,//g`). Then, we print only the edited lines (option `-n` plus print command `p`). The result of the Sed statement should be redirected to a file named *genomes2.txt*.

```
sed -n '/\?!s/,//gp' genomes.txt > genomes2.txt
```

Now we have a nice file, *genomes2.txt*, and we can start to learn more details about AWK.

13.4 Patterns

As we have already seen, AWK statements consist of a pattern with an associated action. With AWK you can use different kinds of patterns. Patterns control the execution of actions: only if the currently active line, the record, matches the pattern, will be the action or command executed. If you omit to state a pattern, then every line of the input text will be treated by the actions (default pattern). Let us take a closer look at the available pattern types.

Except for the patterns `BEGIN` and `END`, patterns can be combined with the Boolean operators `||` (or), `&&` (and) and `!` (not).

13.4.1 Regular Expressions

Probably, the simplest patterns are regular expressions (see Chap. 11 on p. 157). Regular expressions must be enclosed in slashes (`/ . . . /`). If you append an exclamation mark (`!`) after the last slash, all records *not* matching the regular expression will be chosen. In the following example we use the file *enzyme.txt*, which we created in Terminal 97 on p. 198.

```

Terminal 101: Regular Expressions
1  $ cat enzyme.txt
2  Enzyme      Km
3  Protease    2.5
4  Hydrolase   0.4
5  ATPase      1.2

```

```

6  $ awk '/2/ {print $1}' enzyme.txt
7  Protease
8  ATPase
9  $ awk '!/2/ {print $1}' enzyme.txt
10 Enzyme
11 Hydrolase
12 $

```

For clarity, we first print the content of the file *enzyme.txt*. Then, in line 6 of Terminal 101, the first field (*\$1*) of all records (lines) containing the character “2” is printed. The statement in line 9 inverses the selection: the first field of all records not containing the character “2” is printed. In our example, we used the easiest possible kind of regular expressions. Of course, regular expressions as patterns can be unlimitedly complicated.

13.4.2 Pattern-Matching Expressions

Sometimes you will need to ask the question if a regular expression matches a field. Thus, you do not wish to see whether a pattern matches a record but a specified field of this record. In this case, you would use a pattern-matching expression. Pattern-matching expressions use the tilde operator (*~*). In the following list, the variable *\$n* stands for any field variable like *\$1*, *\$2* or so.

<i>\$n</i> <i>~</i> / <i>re</i> /	Is true if the field <i>\$n</i> matches the regular expression <i>re</i> .
<i>\$n</i> <i>!~</i> / <i>re</i> /	Is true if the field <i>\$n</i> does not match the regular expression <i>re</i> .

The regular expression must be enclosed in slashes. Let us print all lines where the first field does not fit the regular expression */ase/*.

```

_____ Terminal 102: Pattern-Matching Expression _____
1  $ awk '$1 !~ /ase/' enzyme.txt
2  Enzyme           Km
3  $

```

Remember that the default action is: print the line. In Terminal 102, we omit the action and specify only a pattern-matching expression. It reads: if field 1 (*\$1*) does not match (*!~*) the regular expression (*/ase/*), then the condition is true. Only if the condition is true will the whole line (record) be printed.

In the next example, we check which users have set the bash shell as their default shell. This information can be read from the system file */etc/passwd*.

```

_____ Terminal 103: Printing Bash Users _____
1  $ awk -F":" ' $7 ~ /bash/ {print $1}' /etc/passwd
2  root
3  rpm

```

```

4 | postgres
5 | mysql
6 | rw
7 | guest
8 | Freddy
9 | $

```

The AWK statement in line 1 of Terminal 103 checks whether the 7th field of the */etc/passwd* file contains the regular expression “bash”, which must be enclosed in slashes. For all matching records, the first field, that is the username, will be printed. Note that the field separator is set to the colon (:) character. Do you prefer to have the output ordered alphabetically? You should know how to do this! How about

```
awk -F":" ' $7 ~ /bash/ {print $1}' /etc/passwd|sort
```

You just pipe the output of AWK to `sort`.

13.4.3 Relational Character Expressions

It is often useful to check if a character string (a row of characters) in a certain field fulfills specified conditions. You might, for example, want to check whether a certain string is contained within another string. In these cases, you would use relational character expressions. The following list gives you an overview of the available expressions. In all cases, the variable *\$n* stands for any field variable like *\$1* or *\$2*. The character string *s* must always be enclosed in double quotes.

<code>\$n == "s"</code>	Is true if the field <i>\$n</i> matches exactly the string <i>s</i> .
<code>\$n != "s"</code>	Is true if the field <i>\$n</i> does not match the string <i>s</i> .
<code>\$n < "s"</code>	Character-by-character comparison of <i>\$n</i> and <i>s</i> . First, the first character of each string is compared, then the second character, and so on. The result is true if <i>\$n</i> is lexicographically smaller than <i>s</i> : “flag < fly” and “abc < abcd”.
<code>\$n <= "s"</code>	As above; however, the result is true if <i>\$n</i> is lexicographically smaller than or equal to <i>s</i> .
<code>\$n > "s"</code>	Character-by-character comparison of <i>\$n</i> and <i>s</i> . First the first character of each string is compared, then the second character and so on. The result is true if <i>\$n</i> is lexicographically greater than <i>s</i> : “house > antenna” and “abcd > abc”.
<code>\$n >= "s"</code>	As above; however, the result is true if <i>\$n</i> is lexicographically greater than or equal to <i>s</i> .

Attention, it is very easy to type “=” instead of “==”! This would not cause an error message; however, the output of the AWK statement would not be what you

want it to be. With “=” you would assign a variable. It is also worthwhile to note that uppercase characters are lexicographically less than lowercase characters and number characters are less than alphabetic characters.

```
numbers < uppercase < lowercase
```

Thus, “Apple” is less than “apple” or, in other words,

```
Apple < apple
```

is true. I guess we must not discuss that “Protein” is greater than “DNA”, at least from a lexicographical point of view. More confusion is generated by the comparison of words starting with the same character. Let us now take a look at the results of relational character expressions in such cases.

```

Terminal 104: Character Relations
1  $ cat>chars.txt
2  Apple
3  apple
4  a
5  A
6  $ awk '$0 < "a"' chars.txt
7  Apple
8  A
9  $ awk '$0 < "A"' chars.txt
10 $ awk '$0 <= "A"' chars.txt
11 A
12 $ awk '$0 >= "A"' chars.txt
13 Apple
14 apple
15 a
16 A
17 $ awk '$0 > "A"' chars.txt
18 Apple
19 apple
20 a
21 $ awk '$0 > "a"' chars.txt
22 apple
23 $

```

I must admit that the use of the term “apple” in Terminal 104 does not quite give the impression that we are learning a programming language. Anyway, we first generate a text file with the words “Apple” and “apple” and the characters “A” and “a”. In all AWK statements we check the relation of the record (\$0), that is the complete line, with the character “a” or “A”. We do not define any action; thus, the default action (print \$0) is used. You are welcome to try out more examples in order to get a feeling for relational character expressions. You should recognize that, in our example, the lexicographically lowest word is “A”. On the contrary, the highest is the word “apple”:

```
apple > a > Apple > A
```

Another important thing to understand is that “==” requires a perfect match of strings and not a partial match. Take a look back at Terminal 103 on p. 202. There we looked for a partial string match. We checked whether the string “bash” (represented by the regular expression `/bash/`) is contained in the variable `$7`. If we instead used

```
awk -F": " '$7 == "bash" {print $1}' /etc/passwd
```

we would get no match. Because the variable `$7` must then have the value “bash”, but it has the value “/bin/bash”. This is not an exact match!

13.4.4 Relational Number Expressions

Similar to relation character expressions are relational number expressions, except that they compare the value of numbers. In the following list of available operators `$n` represents any field variable and `v` any numerical value.

<code>\$n == v</code>	Is true if <code>\$n</code> is equal to <code>v</code> .
<code>\$n != v</code>	Is true if <code>\$n</code> is not equal to <code>v</code> .
<code>\$n < v</code>	Is true if <code>\$n</code> is less than <code>v</code> .
<code>\$n <= v</code>	Is true if <code>\$n</code> is less than or equal to <code>v</code> .
<code>\$n > v</code>	Is true if <code>\$n</code> is greater than <code>v</code> .
<code>\$n >= v</code>	Is true if <code>\$n</code> is greater than or equal to <code>v</code> .

We have already used relational number expressions in our very first example in Terminal 97 on p. 198. The use of relational number expressions is straightforward. In the above list, we have always used the variable `$n`. Of course, the expression is much more flexible. Any numerical value is allowed on the left side of the relation. In the following example, we calculate the length of the field variable with the function `length( )`.

```

1  $ awk 'length($1) > 6' enzyme.txt
2  Protease      2.5
3  Hydrolase     0.4
4  $

```

The statement in line 1 of Terminal 105 reads: print all records (default action) of the file `enzyme.txt`, where the length of the first field (`length($1)`) is larger than six characters. You will learn more about *functions* later.

13.4.5 Mixing and Conversion of Numbers and Characters

What happens if you mix in your relation numerical and alphabetical values? This is a tricky case and you should really take care here! Rule: If one side of your relation is a string, AWK considers both sides to be strings. Thus, the following statement will lead to a wrong result.

```

Terminal 106: Mixing
1 $ awk '$2 > 2' enzyme.txt
2 Enzyme      Km
3 Protease    2.5
4 $

```

Line 2 in Terminal 106 is printed, even though “Km” is not a numerical value. However, the character string “Km” is larger than the numerical number “2” (see Sect. 13.4.3 on p. 203). AWK always tries to find the best solution. Thus, the statement

```
one=1; two=2; print (one two)+3
```

prints the numeric value 15. AWK assumes that you want to concatenate the variables *one* and *two* with the statement “one two”, leading to 12. Then, AWK assumes that you want to do a calculation and adds 3–12. The result is 15.

If you need to force a number to be converted into a string, concatenate that number with the empty string “”. A string (that contains numerical characters) can be converted into a number by adding 0 to that string: “2.5” converts into 2.5, “2e2” converts into 2,000, “2.5abc” converts into 2.5 and “abc2.5” converts into 0. Thus, the solution to the problem in Terminal 106 is:

```
awk '($2+0) > 2' enzyme.txt
```

With these short examples, I wanted to turn your attention to the problem of mixing letters and numbers. You should always carefully check your statements! Always!

13.4.6 Ranges

A range of records can be specified using two patterns separated by a comma.

```

Terminal 107: Ranges
1 $ awk '/En/,/Hy/' enzyme.txt
2 Enzyme      Km
3 Protease    2.5
4 Hydrolase    0.4
5 $

```


In the example in Terminal 107 all records between and including the first line matching the regular expression “En” and the first line matching the regular expression “Hy” are printed. Of course, pattern ranges could also contain relations.

```

Terminal 108: Pattern Range
1  $ awk '$2+0 > 3, $2+0 < 2' structure.pdb
2  SEQRES 4 A 162 ASP GLY GLY THR ALA GLY MET GLU LEU LEU
3  HELIX 1 hel ILE A 18 GLN A 28
4  ATOM 4 CE MET A 1 51.349 24.403 47.765 1.00 71.39 C
5  ATOM 5 CG MET A 1 48.708 24.106 48.629 1.00 66.70 C
6  ATOM 6 N MET A 1 50.347 23.578 52.116 1.00 62.03 N
7  ATOM 7 O MET A 1 47.875 25.011 52.020 1.00 54.99 O
8  ATOM 8 SD MET A 1 49.731 23.948 47.163 1.00 77.15 S
9  $

```

In the example in Terminal 108 we use File 4 on p. 161. Note that we add zero to the value of variable \$2. By doing this, we ensure that we have a numeric value, even though the second field contains a text string. The example in Terminal 108 shows that the range works like a switch. Upon the first occurrence of the first pattern (here “\$2+0 > 3”), all lines are treated by the action (here the default action: print record) until the second pattern (here “\$2+0 < 2”) matches or becomes true. As long as the “pattern switch” is turned on, all lines match. When it becomes switched off, no line matches, until it becomes turned on again. If the “off switch”, which is the second pattern, is not found, all records down to the end of the file match. That happens in Terminal 108 from line 4 on.

If both range patterns are the same, the switch will be turned on and off at each record. Thus, the statement

```
awk '/AUTHOR/, /AUTHOR/' structure.pdb
```

prints only one line.

13.4.7 *BEGIN and END*

BEGIN and END are two special patterns in AWK. All patterns described so far match input records in one or the other way. In contrast, BEGIN and END do not deal with the input file at all. These patterns allow for initialization and cleanup actions, respectively. Both BEGIN and END must have actions and these must be enclosed in braces. There is no default action.

The BEGIN pattern, or BEGIN block, is executed before the first line (record) of the input file is read. Likewise, the END block is executed after the last record has been read. Both blocks are executed only once.

```

Terminal 109: BEGIN and END Blocks
1  $ awk '
2  > BEGIN {FS="-"; print "Species in the file:"}
3  > {print "\t" $1}
4  > END {print "Job finished"}

```

```
5 > ' genomes2.txt
6 Species in the file:
7     H. sapiens (human)
8     A. thaliana (plant)
9     S. cerevisiae (yeast)
10    E. coli (bacteria)
11 Job finished
12 $
```

In Terminal 109 we work with the file *genomes2.txt*, which we generated in Terminal 100 on p. 201. Our AWK statement spans several lines. You might remember from Sed that the shell recognizes if opened single quotes are closed when you press **(Enter)**. We use this feature in order to enter our AWK program from lines 1 to 5 in Terminal 109. The BEGIN block contains two commands: the variable *FS* is set to the value “-” and some text is printed. As you will see later, the variable *FS* contains the field separator. The default value of *FS* is the space character. As you can see, we can assign the field separator either with the option -F (see Terminal 98 on p. 200) or, as shown here, by assigning the variable *FS* in the BEGIN block. Note that both commands are separated by a semicolon. Alternatively, you could write both commands on two separate lines. Line 3 of Terminal 109 contains an AWK command without pattern. Thus, the command will be applied to all lines (default pattern). The command states that “\t” and the content of field 1 are to be printed. As you will see later, \t is the tabulator meta character. Finally, the END block in line 4 prints a message that the job has been finished.

The special patterns BEGIN and END cannot be used in ranges or with any operators. However, an AWK program can have multiple BEGIN and END blocks. They will be executed in the order in which they appear.

13.5 Variables

We have learned already in the section about shell scripts how to work with variables (see Sect. 10.3 on p. 128). We saw that a variable stands for something else. They are a way to store a value at one point in your program and recall this value later. In AWK, we create a variable by assigning a value to it. This value can be either a text string or a numerical. In contrast to the shell, variables must not be preceded by a dollar character. Variable names must start with an alphabetical character and can then contain any character (including digits). As everything in Linux, variable names are case-sensitive!

13.5.1 Assignment Operators

There are several ways how to assign a value to a variable in AWK. The most common commands (assignment operators) are stated below. In the list *x* represents a variable, *y* a number or text.

<code>x = y</code>	The most common command to assign a value to a variable. <code>y</code> can be either a text or a digit or the result of a command.
<code>x += y</code>	The number <code>y</code> is added to the value of variable <code>x</code> . The result is stored in <code>x</code> . This is the same as <code>x=x+y</code> .
<code>x -= y</code>	The number <code>y</code> is subtracted from the value of variable <code>x</code> . The result is stored in <code>x</code> . This is the same as <code>x=x-y</code> .
<code>x *= y</code>	The number <code>y</code> is multiplied with the value of variable <code>x</code> . The result is stored in <code>x</code> . This is the same as <code>x=x*y</code> .
<code>x /= y</code>	The value of variable <code>x</code> is divided by <code>y</code> . The result is stored in <code>x</code> . This is the same as <code>x=x/y</code> .
<code>x^= y</code>	The value of variable <code>x</code> is raised to the <code>y</code> power. The result is stored in <code>x</code> . This is the same as <code>x=x^y</code> .

Let us take a look at a little example on how to assign variables. Again, we work with the file *genomes2.txt*.

```

_____ Terminal 110: AWK Variables _____
1  $ awk '
2  > {species=species $1 $2 "   "
3  > genes+= $8}
4  > END{print species "\n have " genes " genes altogether"}
5  > ' genomes2.txt
6  H.sapiens   A.thaliana   S.cerevisiae   E.coli
7    have 64271 genes altogether
8  $

```

In Terminal 110, we use a multiple-line command spanning from lines 1 to 5. In line 2, we define the variable *species* and assign to it the value of itself (initially the variable is empty), plus the content of field 1 (*\$1*), plus the content of field 2 (*\$2*), plus three spaces ("___"). Note that although *species*, *\$1*, *\$2*, and "___" are separated by spaces, these spaces are not saved in the variable *species*. In order to generate space characters, they must be stated within double quotes because spaces are normal text characters. In line 3, the content of field 8 of the file *genomes2.txt* is added to the content of the variable *genes*. Initially, the variable is empty (0). Since field 8 of the file *genomes2.txt* contains the number of genes, the variable *genes* contains the sum of all genes, the column sum. In line 4 of Terminal 110, AWK is instructed to print the content of the variables *species* and *genes* together with some text that must be stated in double quotes. The special character "\n" represents the newline character. This means, not \n will be printed, but a new line will be started instead.

13.5.2 Increment and Decrement

Interesting and commonly used variable manipulation tools are the increment (++) and decrement (--) operators.

<code>x++</code>	First the value of variable <i>x</i> is returned. Then the value of <i>x</i> is increased by 1: “ <code>x=1; print x++</code> ” → 1.
<code>++x</code>	First the value of variable <i>x</i> is increased by 1. Then the value of <i>x</i> is returned: “ <code>x=1; print ++x</code> ” → 2.
<code>x--</code>	First the value of variable <i>x</i> is returned. Then the value of <i>x</i> is decreased by 1: “ <code>x=1; print x--</code> ” → 1.
<code>--x</code>	First, the value of variable <i>x</i> is decreased by 1. Then the value of <i>x</i> is returned: “ <code>x=1; print --x</code> ” → 0.

As described in the list above, the increment (`++`) and decrement (`--`) operators increase or decrease the value of a variable by 1, respectively. In fact, you could do the same thing with an assignment operator described in the previous section. Thus, the pre-increment `++x` is equivalent to `x+=1` and the post-increment `x++` is equivalent to `(x+=1) - 1`. The same holds for `--`: the pre-decrement `--x` is equivalent to `x-=1` and the post-decrement `x--` is equivalent to `(x-=1) - 1`. The increment and decrement operators are nice, easily readable shortcuts for the assignment operators. As you might have figured out, the difference between the pre- and the post-increment (or decrement) is the return value of the expression. The pre-operator first performs the increment or decrement and then returns a value, whereas the post-operator first returns a value and then increments or decrements.

```

_____ Terminal 111: Incrementing and Decrementing _____
1  $ awk 'BEGIN{
2  > x=1; y=1; print "x="x, "y="y
3  > print "x++="x++, "++y="++y, "x="x, "y="y}'
4  x=1 y=1
5  x++=1 ++y=2 x=2 y=2
6  $

```

Terminal 111 shows you how the pre- and post-increment operator performs in real life. Both increase the value of the variables *x* and *y* by 1. However, the return values of `x++` and `++y` are different.

Terminal 111 demonstrates also the use of the `BEGIN` block for small experimental scripts. The complete AWK script is written within the `BEGIN` block. Thus, we do not need any input file. This is a convenient way to write small scripts that demonstrate only the function of something without referring to any input file.

13.5.3 Predefined Variables

AWK comes with a number of predefined variables that can be used and modified. While some variables provide you with valuable information about the input file or record, others allow you to adapt the behavior of AWK to your own needs.

13.5.3.1 Positional Variables

We have worked already intensively with positional variables. They are the only variables in AWK that are preceded by the dollar character. The positional variable `$0` contains the whole active record, while `$1`, `$2`, `$3` and so on, contain the value of the different fields of the record.

13.5.3.2 Field Separator and Co

There are a number of very helpful variables which help you to define what structures AWK interprets as records and fields. The best way to assign a value to any of the variables in the following list is to use a `BEGIN` block.

FS	Field Separator. The value of <i>FS</i> is the input field separator used to split a record into the positional variables <i>\$1</i> , <i>\$2</i> and so on. The default value is a single space character. However, any sequence of spaces or tabs is recognized as a single field-separator in the default setting. Furthermore, leading spaces of tabulators are ignored. You can also assign a regular expression to <i>FS</i> .
RS	Record Separator. The value of <i>RS</i> defines a line of text. The default setting is “ <code>\n</code> ”, which is the newline character. However, you could also assign different values to <i>RS</i> .
OFS	Output Field Separator. This variable defines which sign is used to separate fields delimited by commas in AWK’s <code>print</code> command. The default setting is a single space character.
ORS	Output Record Separator. The output record separator defines the end of each <code>print</code> command. The default setting is the newline character “ <code>\n</code> ”.
FIELDWIDTHS	A space-separated list of field widths to use for splitting up the record. Assigning a value to <i>FIELDWIDTHS</i> (in a <code>BEGIN</code> block) overrides the use of the variable <i>FS</i> for field splitting. Command is only available in <code>gawk</code> .

Quite commonly used is the variable *FS*. However, especially in cases where the output of scientific software is analyzed, the *FIELDWIDTHS* variable can be interesting. Let us take a look at the general use of these variables.

```

1  $ awk 'BEGIN{FIELDWIDTHS="3 4 3"}
2  {print $0":\n", $1, $2, $3, $4}' enzyme.txt
3  Enzyme      Km:
4      Enz yme
5  Protease    2.5:
6      Pro teas e
7  Hydrolase   0.4:
8      Hyd rola se

```

```

9 | ATPase      1.2:
10 |   ATP ase
11 | $

```

Terminal 112 gives you an impression of how to work with the variable *FIELDWIDTHS*. In line 1 we assign the value “3 4 3” to it. This tells AWK to split each new input line (record) into three fields. The first field is 3, the second 4 and the third again three characters long. The rest of the record is omitted. Thus, the variable \$4 is empty. Note, however, that the variable \$0 still contains the whole record. \$0 is not affected by any field splitting.

Now, let us take a look at what we can do with the output field separator.

```

_____ Terminal 113: Output Field Separator _____
1 | $ awk 'BEGIN{OFS="---"; print "Hello", "World"}'
2 | Hello---World
3 | $ awk 'BEGIN{OFS="---"; ORS="<<<"; print "One", "Two"}'
4 | One---Two<<<$

```

In Terminal 113 we use only a BEGIN block to demonstrate the function of the *OFS* and *ORS* variables. As you can see, the assigned values are used to format the output of the `print` command. Note that the output field separator comes into effect only when commas are used with the `print` command. Note also that the input prompt (\$) follows immediately after the output of the AWK script. That is because we changed the default output field separator, the newline character (\n), to something stupid like “<<<”.

13.5.3.3 File and Line Information

Some variables provide you with valuable information about the file you are just working with.

NF	Number of Fields. This variable returns the number of fields of the current record (line of text).
NR	Number of Records. This variable returns the number of records that have been read from the input file(s). Note that when you “digest” several files with AWK, the number of processed records is not reset when the new file is processed!
FNR	File Number of Records. This variable returns the number of records that have been read from the currently active file. In contrast to <i>NR</i> , <i>FNR</i> is reset when a new record from a new file is read!
FILENAME	Filename. This variable returns the name of the currently processed file. <i>FILENAME</i> changes each time a new file is read.

Okay, let us run a stupid small script that shows us what this is all about.

```

Terminal 114: File Information
1  $ awk 'BEGIN {print "Lines & Fields in Files:}"
2  > {print FILENAME ": \t", FNR" - "NF, "\t total:" NR}
3  > ' enzyme.txt genomes2.txt
4  Lines & Fields in Files:
5  enzyme.txt:      1 - 2   total:1
6  enzyme.txt:      2 - 2   total:2
7  enzyme.txt:      3 - 2   total:3
8  enzyme.txt:      4 - 2   total:4
9  genomes2.txt:    1 - 9   total:5
10 genomes2.txt:    2 - 9   total:6
11 genomes2.txt:    3 - 9   total:7
12 genomes2.txt:    4 - 9   total:8
13 $

```

The AWK script in Terminal 114 makes use of all variables we encountered in the list above. In line 2, we command AWK to read out all available information about the input file given after the script in line 3. Again, we give our output some style by formatting it with the `print` command. We have learned already that commas separate output fields and lead to the output of the variable *OFS* (output field separator). The default output is a space character. The “\t” represents the tabulator character. It helps us to align the output. Any text we wish to print must be enclosed in double quotes. The example in Terminal 114 shows how the variables *FILENAME*, *NR*, *NF*, and *FNR* are set. Take a look at the original files in order to understand the output: *enzyme.txt* is shown in Terminal 97 on p. 198 and *genomes2.txt* is shown in Terminal 100 on p. 201.

13.5.3.4 Command Line Parameters

The variable *ARGV* is in fact an array (see Sect. 13.5.4 on p. 215). Thus, behind the facade of *ARGV* is not just one but many entries. You can access these by using *indices*. *ARGV[0]* is the first element of the array; it always contains the value “awk”. *ARGV[1]* contains the first command line parameter, *ARGV[2]* the second command line parameter, and so on. The variable *ARGC* returns the number of used *ARGV* variables. Thus, *ARGV[ARGC-1]* is the last command line parameter (minus 1, because the first element is 0). Note that command line parameters need to be between the AWK script and the input file(s), separated by space characters. The syntax is:

```

      awk  'script'  par1      par2      InputFile(s)
      ARGV[0]      ARGV[1]  ARGV[2]      ARGC=3

```

Let us see how we can use command line parameters.

```

Terminal 115: Command Line Arguments
1  $ awk 'BEGIN{item=ARGV[1]; ARGV[1]=""}
2  > $1 ~ item
3  > ' rotea enzyme.txt
4  Protease    2.5
5  $

```

In line 1 of Terminal 115, we copy the value of the command line parameter in variable *ARGV[1]* to variable *item*. Then we erase the content of *ARGV[1]*. Important: you must erase all assigned *ARGV[n]* (with $n > 1$) because they will otherwise be regarded as input files, which are not present. In order to understand this, try out the following AWK statement:

```
awk 'BEGIN{item=ARGV[1]} $1 ~ item' rotea enzyme.txt
```

You will receive an error message saying that the file or directory could not be found. With *ARGV[1] = ""* in line 1 in Terminal 115 we erase the content of *ARGV[1]*. Thus, you have to transfer the command line parameters to other variables in the *BEGIN* block and then delete the content of all *ARGV*'s. What our small script in Terminal 115 does is read one command line parameter and use it in order to find out whether it is found in the first field of any line of the input file. If so, the line will be printed out (default action).

Let us summarize what is important to remember: (a) The variable *ARGV* is an array. The first command line parameter sits in *ARGV[1]* and so forth. Transfer the content of *ARGV*'s to new variables inside a *BEGIN* block. Erase the content of *ARGV*'s in the *BEGIN* block. That is it.

Why would we want to use command line parameters? They are helpful when you write complex AWK scripts that are saved as files. Similarly to shell scripts, you can then use the command line parameters to influence the behavior of your script.

13.5.3.5 Shell Environment

When you write AWK scripts you might want to access some of the shell variables, like the user's home directory. For this purpose the special variable *ENVIRON* exists. In fact, this variable is an *associative array*. You need not know exactly what that is. You will learn more about arrays in Sect. 13.5.4. For now, it is more important to see how we have to use it. Assume you want to get the path to your home directory from within an AWK script.

```

Terminal 116: Using Shell Variables in AWK
1  $ awk 'BEGIN{print ENVIRON["HOME"]}'
2  /home/Freddy
3  $

```

The correct command is shown in Terminal 116. The brackets indicate that the variable is an array. In the same way you can gain access to all environment variables.

Take a look back into Sect. 10.3 on p. 128 for a small list of shell variables. You will find a complete list in the manual pages for `bash`: “`man bash`”.

A last hint: you cannot change environmental variables from within AWK scripts. This you can only do with shell scripts.

13.5.3.6 Others

There are more AWK built-in variables available that we are not going to touch here. As always, you are welcome to take a look at the manual pages of AWK.

13.5.4 Arrays

We now learn something about a very important type of variables: *arrays*. We worked already with the arrays *ARGV* and *ENVIRON* in the previous section. In general, an array is a table of values called *elements*. The elements of an array are identified by their *indices* (also called *keys*). In AWK, indices can be either numbers or text strings. Therefore, we call the arrays also *associative arrays* or *hashes*. Internally, indices are always treated as strings. For array names the same rules apply as for variable names. In addition to the name you need to state the array index, which is always done in brackets. The following line gives an example of how to assign array elements.

```
data[1]=123; data[2]="junk"; data["input"]=23.45
```

In this example the array name is *data*. As indices we used *1*, *2*, and *input*. Note that strings must be enclosed in double quotes. Arrays can also have multiple dimensions like

```
data[0,0]=1; data[0,1]=2; data[1,0]=3;data[1,1]=4
```

This is a 2-dimensional array. The indices for each dimension must be separated by commas.

As usual, let us take a look at a small example.

```

1  $ echo "atgccg" | awk 'BEGIN{codon["atg"]="MET"
2  > codon["ccg"]="VAL"; FIELDWIDTHS="3 3"}
3  > {print codon[$1], codon[$2]}'
4  MET VAL
5  $
```

In Terminal 117 you find a mini DNA translator. We pipe a 6-nucleotide-long DNA sequence to an AWK script. In the `BEGIN` block we assign 2 amino acids (MET and VAL) to the array *codon*. As indices we use the codons *atg* and *ccg*, respectively. Finally, we use the built-in variable *FIELDWIDTHS* to define how to split the input

string into fields. Here, we take two times three characters, our codons. In line 3 of Terminal 117, we use the resulting field variables *\$1* and *\$2* as indices in order to get back to the encoded amino acids. This small example shows you how powerful associative arrays can be.

But now let us take a little tour through the world of arrays.

13.5.4.1 Assigning Array Elements

Assigning a value to an array element is as easy as assigning a value to a variable:

```
array[index]=value
```

The index can be either a number, a text string, or a variable that becomes extended. This means the variable is replaced by its value (as we saw in Terminal 117 with the positional variables *\$1* and *\$2*). If you use a text string it must be enclosed in double quotes.

A very powerful way to create and assign an array is the command `split`:

```
split(string, array, fieldseparator)
```

This function divides a string at the given field separator and stores the resulting pieces in the array named *array*. The first piece would be stored in *array[1]*. The command returns the number of pieces that have been generated and saved.

```

1  $ awk '{pieces=split($2,data,".")}'
2  > {print pieces, "pieces have been saved"}
3  > {print "Integer: "data[1]"\tDecimal: "data[2]}
4  > ' enzyme.txt
5  1 pieces have been saved
6  Integer: Km   Decimal:
7  2 pieces have been saved
8  Integer: 2    Decimal: 5
9  2 pieces have been saved
10 Integer: 0    Decimal: 4
11 2 pieces have been saved
12 Integer: 1    Decimal: 2
13 $

```

Terminal 118 gives a nice example of the `split` command. In order to recall the content of the file *enzyme.txt* take a look at Terminal 120. What are we doing in Terminal 118? Well, first we split the content of variable *\$2* at the point character. Since we know that there is only one point in this field, the decimal delimiter, we expect to obtain two elements for the array *data*. These we print out. We now have the integer part and decimal part of the Km value separated. As you can see from line 6 of Terminal 118, only one array element is generated if the defined delimiter is not found. This is the case in the first line of *enzyme.txt*.

As with the input line field splitting, leading and trailing spaces as well as running spaces are ignored when the field separator is set to a single space character. If you

do not provide any field separator with the `split` command, the AWK's variable `FS` will be used instead. If the field separator is set to "" (empty string), then the string is split after every character. Terminal 119 illustrates this behavior.

```

Terminal 119: Splitting a String after each Character
1  $ echo "1234" | awk '{split($0,a,""); print a[2]}'
2  2
3  $

```

13.5.4.2 Referring to an Array Element

There are two ways in which you can refer to an array element. The most common one is to use the array's name and an index:

```
array[index]
```

The other way is to use the expression in:

```
index in array
```

The result of the expression is true if the element `array[index]` exists.

13.5.4.3 Scanning an Array

There is no way to check whether a certain value is contained within an array. This is a pity, but the bitter truth. Instead, AWK provides a special command to scan through a complete array:

```
for (variable in array) commands
```

This `for...in...` command generates a loop through all previously used indices of the array `array`. These indices are accessible via the variable `variable`. The reality looks like this:

```

Terminal 120: Looping Through Array Elements with for...in...
1  $ awk '/[0-9]/ {data[$1]=$2}
2  END{for (x in data) print "Index="x, " Value="data[x]}
3  ' enzyme.txt
4  Index=Hydrolase Value=0.4
5  Index=Protease Value=2.5
6  Index=ATPase Value=1.2
7  $ cat enzyme.txt
8  Enzyme Km
9  Protease 2.5
10 Hydrolase 0.4
11 ATPase 1.2
12 $

```

The script in Terminal 120 saves all values of the first and the second field of lines with a number in the array *data*. In the END block we use the `for...in...` loop to recall all array elements and print out their value (`data[x]`), together with the index (`x`). In line 7 we print out the file *enzyme.txt* again. If you compare the output of the AWK script with the file content you will recognize that the order of the output is strange. There is nothing we can do about this. The order in which the elements of the array are accessed by AWK cannot be influenced. It depends on the internal processing of the data.

13.5.4.4 Deleting an Array or Element

In order to delete a complete array you just use the command

```
delete array
```

where *array* is the array name. If you want to delete only a certain element you can use

```
delete array[index]
```

Once an array element has been deleted, it is gone forever. There is no way to recover it.

13.5.4.5 Sorting an Array

With the command `asort` you can sort the elements of an array according to their value. Note that this command is only available in gawk.

```

1  $ awk '/[0-9]/ {data[$1]=$2}
2  END{asort(data)
3  for (x in data) print "Index="x, "  Value="data[x]}
4  ' enzyme.txt
5  Index=1    Value=0.4
6  Index=2    Value=1.2
7  Index=3    Value=2.5
8  $

```

In Terminal 121 we use the `asort` command in an END block. As you will notice in the output, all original indices of the array *data* are irrecoverably lost! This is not quite what you want to happen. The only thing you can do is use the `asort` command as

```
asort(inarray, outarray)
```

With this construction, AWK copies the *inarray* into *outarray* and then sorts *outarray*. In this way, only the indices of the newly generated *outarray* are destroyed, whereas your original *inarray* remains untouched.

If you want to sort the indices of an array, you have a problem—a problem that can be attacked with a little program. You will find it in Sect. 13.11.1 on p. 243.

13.5.5 Shell Versus AWK Variables

You should be aware of the fundamental difference between shell variables (see Sect. 10.3 on p. 128) and AWK variables. To recall shell variables you must precede them with the dollar character. When assigning a value to a shell variable, you do not precede it with the dollar character. In AWK, you *never* use the dollar character with variables. The only exceptions are special variables like the record (\$0) or fields (\$1, \$2...).

13.6 Scripts and Executables

Up to this point, we have had many examples of scripts that spanned more than one line. This is not quite convenient, especially not when you want to reuse these scripts. Then it would be much nicer to have them stored in files. Like shell scripts and Sed scripts you can, of course, also save AWK scripts. The syntax to call a script file is

```
awk -f ScriptFile InputFile(s)
```

As usual, script files can be commented. Comments always begin with the hash character (#).

```

1  # save as sort1.awk
2  # sort on value
3  /[0-9]/ {data[$1]=$2}
4  END{asort(data)}
5  for (x in data) {print "Index="x, " Value="data[x]}
6  }
```

Program 31 matches the script we used in Terminal 121. You call the script with

```
awk -f sort1.awk enzyme.txt
```

Like shell scripts, you can make AWK scripts executable. This is very convenient when you use the scripts a lot and want to create an alias.

```

1  #!/bin/awk -f
2  # save as sort1-exe.awk
3  # sort on value
4  /[0-9]/ {data[$1]=$2}
```

```
5 | END{asort(data)
6 |   for (x in data) {print "Index="x, "   Value="data[x]}
7 | }
```

Program 32 gives an example of an executable AWK script. It is important to remember to use the option `-f` in the first line. Of course, you must make the file executable with

```
chmod u+x sort1-exe.awk
```

The script is executed with

```
./sort1-exe.awk enzyme.txt
```

This is basically all you need to know in order to create AWK scripts.

13.7 Decisions: Flow Control

In this section, you will learn how to influence the flow of your AWK programs. You should already be an expert in this—at least, if you have worked carefully through the corresponding section on shell programming (see Sect. 10.7 on p. 138). Decisions, or *control statements* as one might call them, are almost the most important part of a programming language. If you look at Sed, you will realize that this is a very limited tool—because it lacks solid control structures. AWK’s control structures belong to the world of actions, this means they must be enclosed in braces. The control statements in this section check the state of a condition, which can be either true or false. This condition is typically one of the relational expression presented in Sects. 13.4.2 on p. 202, 13.4.3 on p. 203 and 13.4.4 on p. 205. Okay, let us see how we can use control statements.

13.7.1 *if...else...*

If you read up to this point *then* read on, or *else* start with Chap. 13 on p. 197. This is what the `if...else` construct is doing: it checks if the first condition is true. If it is true, the following statement is executed, or else an alternative is executed. This alternative is optional.

```
{if (condition) {
    command-1(s)}
else {                # this part is optional
    command-2(s)}     # this part is optional
}
```

The “`else {command-2(s)}`” is optional and can be omitted. Let us take a look at a little program.

```

1  # save as if-else.awk
2  # demonstrates if-else structure
3  # use with enzyme.txt
4  {if ($2 < 2) {
5      sum_b+= $2
6  }
7  else {
8      sum_s+= $2
9  }
10 }
11 END{
12     print "Sum of numbers greater than or equal 2: "sum_b
13     print "sum of numbers smaller than 2      : "sum_s
14 }

```

Program 33 shows an example of how to use the “`if...else`” construction. The program is executed by

```
awk -f if-else.awk enzyme.txt
```

It simply calculates the sum of the value of all second fields of the file *enzyme.txt* that are smaller than 2 (if `($2 < 2)`) and larger than or equal to 2 (else). Remember that “`sum_b += $2`” (where *sum_b* is a variable) is the same as “`sum_b = sum_b + $2`” (see Sect. 13.5.1 on p. 208). The output of the program is shown in the following Terminal.

```

1  $ awk -f if-else.awk enzyme.txt
2  Sum of numbers greater than or equal 2: 1.6
3  sum of numbers smaller than 2      : 2.5
4  $ cat enzyme.txt
5  Enzyme      Km
6  Protease    2.5
7  Hydrolase   0.4
8  ATPase      1.2
9  $

```

Just to remind you of the content of *enzyme.txt*, it is printed out with the `cat` command.

13.7.2 *while...*

While you read this script, concentrate!—The `while` statement does something (concentrating), while a condition is true (reading). In contrast to the “`do...while`” construction (see Sect. 13.7.3), first the state of the condition is checked, and if the condition is true commands are executed. The result is a loop that will be repeated until the condition becomes false. The syntax is:

```
{while (condition) {
    command(s) }
}
```

It is quite easy to end up in an endless loop with the `while` command. Keep this in mind whenever you use it!

```

Program 34: while.awk - while
1  # save as while.awk
2  # demonstrates the while command
3  # reverses the order of fields
4  BEGIN{ORS="" }
5  {i=NF}
6  {while (i>0) {
7      print $i"\t"; i--}
8  }
9  {print "\n"}
```

Program 34 shows you an application which prints out the fields of each line in reverse order. This means the last field is becoming the first, the second last is becoming the second and so forth. Execution and output of the program is shown in Terminal 123.

```

Terminal 123: Looping with while
1  $ awk -f while.awk enzyme.txt
2  Km      Enzyme
3  2.5     Protease
4  0.4     Hydrolase
5  1.2     ATPase
6  $
```

In the `BEGIN` block in Program 34 we reset the output record separator variable `ORS` (see Sect. 13.5.3 on p. 210). As a result, the `print` command will not generate a new line after execution. For each line of the input file the number of fields (variable `NF`) is stored in the variable `i`. In the `while` loop, first the highest field number (`$i = $(NF)`) is printed, followed by a tabulator character “`\t`”. Next, `i` is decremented by 1 (`i--`). The `while` loop stops when `i` becomes 0. Then a newline character “`\n`” is printed and the next line (record) read from the input file.

13.7.3 *do...while...*

Do not drink alcohol *while* you learn how to use control structures. In contrast to the `while` command (see Sect. 13.7.2), first one or more commands are executed and then it is checked if the condition is true. Only if the condition is still true, is execution of the commands repeated. The syntax is:


```

{do {
    command(s)}
while (condition)
}

```

Since the `do...while` construct is very similar to the `while` command, I will not present any example here. Just remember that the command exists and that it, in contrast to `while`, executes the `command(s)` at least once, even if the condition is false.

13.7.4 *for...*

For each section, there is an example you should carefully study. The `for` loop is a special form of the `while` loop. We saw already a, though unusual, example of a `for` in Sect. 13.5.4 on p. 215. There it was used to read out the complete content of an array.

The normal `for` loop consists of three parts: *initialization*, *break condition* and *loop counter*. The syntax is:

```

{for (initialization; condition, counter){
    command(s) }
}

```

The first part, the initialization, is executed only once. Here, a counter variable is set. The condition is checked in every loop. Only when the condition is true the commands will be executed and the loop counter be executed.

As loop counter one uses usually the increment (`++`) or decrement (`--`) command. The following program prints the quadratic powers of the numbers 1–10.

Program 35: *for.awk* - *for*

```

1  # save as for.awk
2  # demonstrates the for command
3  # calculates quadratic numbers
4  BEGIN{ORS=" "}
5  for (i=1; i<=10; i++){
6      print i**2}
7  print "\n"

```

I guess the program almost explains itself. The whole script is executed in a `BEGIN` block because we do not need any input file. The output record separator variable `ORS` is set to the space character. This prevents the `print` command from generating line breaks. “`i**2`” raises the value of the variable `i` to the power of 2 (see Sect. 13.8.2 on p. 228). In line 7 we print a newline character (`\n`). Execution of the program is shown in Terminal 124.

```

1 $ awk -f for.awk
2 1 4 9 16 25 36 49 64 81 100
3 $

```

13.7.5 Leaving Loops

Being in a loop is fine, as long as you want to be there. However, it might be necessary to leave a `while` or `for` loop before it is officially finished by the conditional state. Of course, there are nice statements for this.

break—With the `break` command you can simply jump out of a loop. The program execution continues at the first command following the loop.

```

while (condition){
    command-L1
    command-L2
    break                >-
    command-L3}          |
command                 <-

```

continue—With the `continue` command you leave the actual loop. However, in contrast to `break`, it brings you back to the beginning of the loop. Only the remaining commands in the loop are skipped and the next cycle is initiated.

```

while (condition){ <-
    command-L1      |
    command-L2      |
    continue        >-
    command-L3}
command

```

next—The `next` command forces AWK to immediately stop processing the current record and load the next record from the input file. The `next` command is not only interesting in the context of loops but can be generally used in AWK scripts. With the command `nextfile`, you can even force AWK to skip the current input file and load the first line of the next input file. If there is no other input file, `nextfile` works like `exit`.

exit—The `exit` command causes AWK to stop the execution of the whole AWK script. The syntax is

```
exit n
```

The number *n* is an optional extension. If you use it, it will be the return code of your AWK script. After execution of the `exit` statement, the script first tries to execute

the `END` block. However, if no `END` block is present, the program stops immediately. If the `exit` command is executed in the `BEGIN` block, AWK does not try to jump to the `END` block but stops execution directly.

13.8 Actions

Until now, we have spent a lot of time on understanding patterns, variables, and control structures. This is indispensable knowledge in order to work with a programming language like AWK. However, the real power of AWK unfolds when we learn about its actions (functions) in this and the following sections. You have learned already that AWK actions are always enclosed in braces. Everything enclosed in braces are actions, which are also called functions. We have already made extensive use of the `print` function. When we worked with arrays in Sect. 13.5.4 on p. 215 we used the functions `asort` and `split`. In this section, we will deal with many more powerful functions. In Sect. 13.8.5 on p. 234 you will even learn how we define our own functions.

13.8.1 Printing

The default `print` command is “`print $0`”. Thus, if you use only `print` in your script, AWK will interpret this as “`print $0`”. The syntax of the `print` command is

```
print item1, item2, ...
```

The items to be printed out are separated by commas. If you want to print text, it must be enclosed in double quotes:

```
print "Text: " variable
```

Everything within double quotes will be printed as it is, whereas *variable* is assumed to be a variable. Thus, the value of the variable will be printed. Each comma is substituted by the value of the output field separator variable *OFS*. The default value is the space character “”. Furthermore, each `print` command is finished by printing the value of the output record separator variable *ORS*. The default value is the newline character “`\n`”.

Usually, `print` prints to the standard output, that is the screen. However, you can also use *redirections* in order to print into a file.

```
awk 'print $0 > "output.txt"' enzyme.txt
```

This command would print all lines of the file *enzyme.txt* into the file *output.txt*. If the file *output.txt* does not exist, it will be created. If it exists, it will be overwritten, unless you use `>>`, which appends the output to an existing file.

Now let us take a look at some special characters. They are used to print tabulators, start new lines or even let the system bell ring. The following list shows the most important escape sequences.

<code>\a</code>	You will hear an <i>alert</i> (system bell) when you print this character.
<code>\b</code>	This prints the <i>backspace</i> character. The result is that the printing cursor will be moved backwards.
<code>\n</code>	When this character is printed a <i>new line</i> will be started.
<code>\t</code>	A <i>horizontal tabulator</i> is printed.
<code>\v</code>	A <i>vertical tabulator</i> is printed.

The following Terminal gives an example of how to use escape sequences.

```

1  $ awk '{print $1"\b\b\b"x"}' enzyme.txt
2  Enzxme
3  Protexse
4  Hydrolxse
5  ATPxse
6  $

```

The AWK statement in line 1 of Terminal 125 prints out the first column of the file *enzyme.txt*. After the content of the positional variable `$1` is printed, the cursor is placed three positions back (`\b\b\b`), then the character “x” is printed. By this, the original content of the line is overwritten.

The command `printf` provides more control over the format of the output. The syntax is:

```
printf (format, item1, item2,...)
```

Here, *format* is a string, the so-called *format string*, that specifies how to output the items. Note that `printf` does not append a newline character! It prints only what the format string specifies. Both the output field separator variable *OFS* and the output record separator variable *ORS* are ignored. The format string consists of two parts: the *format specifier* and an optional *modifier*. Table 13.1 shows a number of available format specifiers and gives examples of what they do.

Note that you need to give a format specifier for each item you want to print. That is illustrated in Terminal 126.

```

1  $ awk 'BEGIN{printf("%g\n", 10.63, 5)}'
2  10.63
3  $ awk 'BEGIN{printf("%g%g\n", 10.63, 5)}'
4  10.635
5  $ awk 'BEGIN{printf("%g - %g\n", 10.63, 5)}'
6  10.63 - 5
7  $

```

Table 13.1 Function of format specifiers for `printf`

Spec.	Value	Result	Comment
<code>%s</code>	10.63	<i>10.63</i>	Print a string
<code>%i</code>	10.63	<i>10</i>	Print only the integer part of the number
<code>%e</code>	10.63	<i>1.063000e+01</i>	Print a number in scientific notation
<code>%f</code>	10.63	<i>10.630000</i>	Print a number in floating point notation
<code>%g</code>	10.63	<i>10.63</i>	Print a number either in floating point or scientific notation, whichever uses less characters

In line 1 of Terminal 126 we use the format string “`%g\n`”. It reads: print a number in the shortest possible way, followed by a newline character. Remember that `printf` does not start a new line by itself. As you can see from the output in line 2, the second item is omitted. In line 3 we specify to print two numbers. They are printed one after the other. In line 5 we introduce a separator between the numbers.

With modifiers you can further specify the output format. Modifiers are specified between the `%` sign and the following letter in the format specifier. Now let us take a look at some available modifiers.

- w Width. This is a number specifying the desired minimum width of a field. If more characters are required, the field width will be automatically expanded accordingly. By default, the output is right aligned.
- Preceding the *width* with the minus character leads to a left alignment of the output.
- . In numeric output, positive values will be prefixed with a space and negative values with a minus sign.
- + In numeric output, positive values will be prefixed with the plus sign and negative values with the minus sign.
- .n Precision. A periodic followed by an integer gives the precision required for the output: “`%.ns`”: maximum *n* characters are printed; “`%.ni`”: minimum number of digits is *n*; “`%.ng`”: maximum number of significant digits; “`%.ne`” or “`%.nf`”: *n* digits to the right of the decimal point.

The following Terminal shows some examples of how the modifiers are used in conjunction with the format modifiers for strings (`%s`) and floating point numbers (`%f`).

Terminal 127: Modifiers

```
1 $ awk 'BEGIN{
2   printf("%s - %f\n", "chlorophyll", 123.456789)}'
3 chlorophyll - 123.456789
4 $ awk 'BEGIN{
5   printf("%6s - %6f\n", "chlorophyll", 123.456789)}'
6 chlorophyll - 123.456789
7 $ awk 'BEGIN{
8   printf("%15s - %15f\n", "chlorophyll", 123.456789)}'
9     chlorophyll -      123.456789
10 $ awk 'BEGIN
11 {printf("%-15s - %-15f\n", "chlorophyll", 123.456789)}'
```

```

12 | chlorophyll      - 123.456789
13 | $ awk 'BEGIN
14 | {printf("%+15s - %+15f\n", "chlorophyll", 123.456789)}'
15 |      chlorophyll -      +123.456789
16 | $ awk 'BEGIN{
17 | printf("%.6s - %.6f\n", "chlorophyll", 123.456789)}'
18 | chloro - 123.456789
19 | $ awk 'BEGIN{
20 | printf("%15.6s - %15.6f\n", "chlorophyll", 123.456789)}'
21 |      chloro -      123.456789
22 | $ awk 'BEGIN{
23 | printf("%-15.6s - %-15.6f\n", "chlorophyll", 123.456789)}'
24 | chloro      - 123.456789
25 | $

```

Terminal 127 gives you an impression of the function of the modifiers. As you can see, they are placed between the % character and the format control letter (here, s and f). All number values without or preceding the period character (.) are *width* values, specifying the minimum width. In lines 11, 14, and 23 the width value is preceded by another formatting character. All numbers behind the period character specify the precision of the output. For details, you should consult the list above. Finally, AWK provides the command `sprintf`. With this function, AWK redirects the output into a variable instead of a file.

```

_____ Terminal 128: Printing into Variables _____
1 | $ awk 'BEGIN{
2 | > out=sprintf("%s","Hello World")
3 | > print out}'
4 | Hello World
5 | $

```

In the example in Terminal 128 we use `sprintf` in order to assign some formatted text to the variable `out`. In line 3, the content of `out` is directed onto the standard output (printed on the screen).

13.8.2 Numerical Calculations

Apart from the standard arithmetic operators for addition ($x+y$), subtraction ($x-y$), division (x/y), multiplication ($x*y$), and raising to the power ($x**y$ or x^y), AWK comes with a large set of numeric functions that can be used to execute calculations. The following list gives you a complete overview of the built-in arithmetic functions.

<code>int(x)</code>	Truncates x to integer.
<code>log(x)</code>	The natural logarithm function.
<code>exp(x)</code>	The exponential function.
<code>sqrt(x)</code>	The square root function.
<code>sin(x)</code>	Returns the sine of x , which must be in radians.
<code>atan2(y, x)</code>	Returns the arctangent of y/x in radians.
<code>cos(x)</code>	Returns the cosine of x , which must be in radians.
<code>rand()</code>	Returns a random number between 0 and 1, excluding 0 and 1. With “ <code>int(n*rand())</code> ” you can obtain nonnegative integer s random number less than n .
<code>srand([x])</code>	Uses x as a new seed for the random number generator. x is optional. If no x is provided, the time of day is used. The return value is the previous seed for the random number generator.

The use of these functions is quite straightforward.

```

1  $ awk 'BEGIN{srand()}
2  print log(100), sin(0.5), int(5*rand()), 3*2}'
3  4.60517 0.479426 1 9
4  $

```

In line 1 in Terminal 129 we set a new value for random number generator. Then the results of some calculations are printed.

13.8.3 String Manipulation

Probably, the most important functions are those that manipulate text. The following functions are of two different types: they either modify the target string or not. Most of them do not. All functions give one or the other type of return value: either a numeric value or a text string. As with the functions for numerical calculations in Sect. 13.8.2, this list of functions is complete.

substr(target, start, length)

Substring. The `substr` function returns a *length* character long substring of *target* starting character at position *start*. If *length* is omitted, the rest of *target* is used.

```

1  $ awk 'BEGIN{target="atgctagctagctgc"}
2  > print substr(target,7,3)
3  > print target}'
4  gct
5  atgctagctagctgc
6  $

```

Terminal 130 illustrates how to use `substr`. Here, we extract and print three characters from the 7th position of the value of the variable `target`.

gsub(regex, substitute, target)

Global Substitution Each matching substring of the regular expression *regex* in the target string *target* is replaced by the string *substitute*. The number of substitutions is returned. If *target* is not supplied, the positional variable `$0` is used instead. An `&` character in the replacement text *substitute* is replaced with the text that was actually matched by *regex*. To search for the literal “`&`” type “`\\&`”.

```

_____ Terminal 131: Extracting Text Strings with gsub _____
1  $ awk 'BEGIN{target="atgctagctagctgc"
2  > print gsub(/g.t/, ">&<", target)
3  > print target}'
4  3
5  at>gct<a>gct<a>gct<gc
6  $

```

In the example in Terminal 131, the regular expression “`g.t`”, enclosed in slashes, is replaced by the matching string (`&`) plus greater and less characters. Note that the target string itself is changed and that the function returns the number of substitutions made.

sub(regex, substitute, target)

Substitution Just like `gsub()`, but only the first matching substring is replaced.

```

_____ Terminal 132: Extracting Text Strings with sub _____
1  $ awk 'BEGIN{target="atgctagctagctgc"
2  > print sub(/g.t/, ">&<", target)
3  > print target}'
4  1
5  at>gct<agctagctgc
6  $

```

As shown in Terminal 132, `sub` substitutes only the first occurrence of the regular expression. Thus, the return value can only be 1 (one substitution) or 0 (regular expression not found in target string).

gensub(regex, substitute, how, target)

General Substitution. Search the target string *target* for matches of the regular expression *regex*. If *how* is a string beginning with “`g`” or “`G`” (global), then all matches of *regex* are replaced with *substitute*. Otherwise, *how* is a number indicating which match of *regex* is to be replaced. If *target* is not supplied, the positional variable `$0` is used instead. Back referencing is allowed (see Sect. 11.4.7 on p. 168). Note that unlike `sub()` and `gsub()`, `gensub` returns the modified string. The original target string is not changed.


```
_____ Terminal 133: Extracting Text Strings with gsub _____  
1 $ awk 'BEGIN{target="atgctagctagctgc"  
2 > print gsub(/g.t/, ">&<", 2, target)  
3 > print target}'  
4 atgcta>gct<agctgc  
5 atgctagctagctgc  
6 $
```

Terminal 133 gives you an example of the function `gsub`. Here, we substitute the second occurrence of the regular expression in *target*. Note that the original string is not modified. Instead, the modification of the text in the variable *target* is the output of `gsub`.

index(target, find)

Search. This function searches within the string *target* for the occurrence of the string *find*. The index (position from the left, given in characters) of the string *find* in the string *target* is returned (The first character in *target* is number 1). If *find* is not present, the return value is 0.

```
_____ Terminal 134: Determining Match Positions with index _____  
1 $ awk 'BEGIN{target="atgctagctagctgc"  
2 > print index(target, "gct")  
3 > print target}'  
4 3  
5 atgctagctagctgc  
6 $
```

Terminal 134 gives you an example of the function `index`. It simply returns the position of the given string.

length(target)

Length. Another easy-to-learn command is `length`. As the name implies, this function returns the length of the string *target*, or the length of the positional variable *\$0* if *target* is not supplied.

```
_____ Terminal 135: Determining String Length with length _____  
1 $ awk 'BEGIN{target="atgctagctagctgc"  
2 > print length(target)  
3 > print target}'  
4 15  
5 atgctagctagctgc  
6 $
```

Terminal 135 gives you an example of how to use `length`.

match(target, regex, array)

Matching. The `match` function returns the position where the longest, leftmost match by the regular expression *regex* occurs in the target string *target*. If *regex* cannot be found, 0 is returned. The built-in variables *RSTART* and *RLENGTH* are set to the

index (position from the left, counted in characters) and the length of the matching substring, respectively.

If *array* is used, the element *array*[0] will be set to the matching substring of *target*. Elements 1 through *n* are filled with the portions of *target* that match the corresponding parenthesized subexpression of the regular expression *regex* (see the section on back referencing, Sect. 11.4.7 on p. 168).

```

1  $ awk 'BEGIN{target="atgctagctagctgc"
2  > print match(target,/(g.(.*)t)/,array)
3  > print "Target: "target
4  > print "0: "array[0], "1: "array[1], "2: "array[2]}'
5  3
6  Target: atgctagctagctgc
7  0: gctagctagct 1: gctagctagct 2: tagctag
8  $

```

Terminal 136 shows an example of *match*. Note that the contents of *array*[1] and *array*[2] overlap by five characters. This is due to our nice arrangement of parentheses in the regular expression! Take a careful look at it and figure out what is going on.

split(target, array, regex)

Splitting. The *split* function splits the string *target* and assigns the resulting products to the elements of an array called *array* (consult also Sect. 13.5.4 on p. 215). The first element of the array is *array*[1] and not *array*[0]! The desired field separator to split *target* is given by the regular expression *regex* which, of course, could also be a string. The number of generated fields is returned. If *regex* is omitted, the built-in field separator variable *FS* will be used instead.

```

1  $ awk 'BEGIN{target="atgctagctagctgc"
2  n=split(target,array,/g.t/)
3  for (i=1; i<=n; i++){printf "%s", array[i] " "}
4  print ""}'
5  at a a gc
6  $

```

In Terminal 137 we split the string saved in the variable *target* at positions defined by the regular expression “*g.t*”. The resulting fields are saved in the array named *array*, whereas the number of resulting fields is saved in *n*. The *for* loop in line 3 is used to read out all elements of the array.

asort(array, destination)

Array Sorting. This function returns the number of elements in the array *array*. The content of *array* is copied to the array *destination*, which then is sorted by its values (for more information see Sect. 13.5.4 on p. 215). The original indices are lost. Sorting is executed according to the information given in Sect. 13.4.3 on p. 203 (apple > a > A > Apple).

```

1  $ awk 'BEGIN{target="atgctagctagctgc"
2  n=split(target,array,/g.t/); asort(array)
3  for (i=1; i<=n; i++){printf "%s", array[i] "  "}
4  print ""}'
5  a  a  at  gc
6  $

```

The script in Terminal 138 is the same as in Terminal 137, but extended by the `asort` function. Thus, the elements of the array are sorted according to their value, before they are printed out.

strtonum(target)

String To Number. This function examines the string *target* and returns its numeric value. You will hardly need this function.

```

1  $ echo "12,3" | awk '{print strtonum($0)}'
2  12
3  $ echo "12.3" | awk '{print strtonum($0)}'
4  12.3
5  $ echo "alpha12.3" | awk '{print strtonum($0)}'
6  0
7  $

```

Terminal 139 illustrates the function of `strtonum`.

tolower(target)

To Lowercase. Returns a copy of the string *target*, with all uppercase characters converted into their corresponding lowercase counterparts. Nonalphabetic characters are left unchanged. For an example take a look at its counterpart `toupper`.

toupper(target)

To Uppercase. The opposite of the function `tolower()`.

```

1  $ awk 'BEGIN{target="atgctagctagctgc"
2  > print toupper(target)
3  > print target}'
4  ATGCTAGCTAGCTGC
5  atgctagctagctgc
6  $

```

Well, the function `toupper` really does what it says. Terminal 140 gives you an example.

13.8.4 System Commands

In the same way that you invoke an AWK statement from a shell script, you can invoke a shell command, or system call, from your AWK script. The `system` function executes the command given as a string.

```
Terminal 141: System Calls
1  $ awk 'BEGIN{system("pwd")}'
2  /home/Freddy
3  $ awk 'BEGIN{print system("pwd")}'
4  /home/Freddy
5  0
6  $ awk 'BEGIN{print system("pwd")}'
7  sh: line 1: pwdd: command not found
8  127
9  $ awk 'BEGIN{print "pwd" | "/bin/bash"}'
10 /home/Freddy
11 $
```

Terminal 141 demonstrates different ways in which you could execute a system call. In line 1, we execute a simple system call with the command `pwd`. The result is printed to standard output. In line 3 we include the `print` function. Now, AWK prints out the return value of the system call. 0 means that there was no error, in contrast to lines 6–8. In line 9, we redirect the output of the print command to the shell. This is a quite convenient way to execute a shell command.

In Sect. 13.9 on p. 238 you will learn how you can use the output of a system call in an AWK program.

13.8.5 User-Defined Functions

It is not my intention to present all available functions here. A complete list of AWK's functions can be found in the manual pages; but what happens if your desired function is missing? Then you have to create your own. This possibility extends AWK's capabilities immensely. User-defined functions are called just like built-in functions, but *you* tell AWK what to do. Thus, with user-defined functions, AWK brings you one big step closer to becoming a smart programmer.

In order to define a function you use the `function` command. It can appear anywhere in the script, since AWK first reads the whole script and then starts to execute it. The syntax of the `function` command is:

```
function name(par1, par2,...){
    command(s)
}
```

That is it. The *name* is the name your function is to have. Here, the same rules apply as for variables and arrays. However, note that you can use any particular name only once: either as variable name, array name, or function name. The *parameters* are variables you deliver to your function when you call it. Several parameters are

separated by commas. The *commands* are the function's soul. They define what the function is actually supposed to do.

As usual, let us take a look at an example.

```

Terminal 142: Function without Parameter
1  $ awk 'function out() {printf "%-5s %-10s\n", $2, $1}
2  {out()} ' enzyme.txt
3  Km      Enzyme
4  2.5     Protease
5  0.4     Hydrolase
6  1.2     ATPase
7  $

```

In line 1 of Terminal 142 we define a function called *out*. The function does not need any parameters, therefore the parentheses remain empty. The action of our function is to print out the positional variables *\$2* and *\$1* in a nicely formatted way. In line 2 we call the function, as if it were a normal command, with *out()*.

Now let us see how we can use parameters.

```

Terminal 143: Function with Parameters
1  $ awk 'function out(pre) {printf pre"%-5s %-10s\n", $2,$1}
2  {out("-> ")} ' enzyme.txt
3  out("# ") ' enzyme.txt
4  -> Km      Enzyme
5  # Km      Enzyme
6  -> 2.5     Protease
7  # 2.5     Protease
8  -> 0.4     Hydrolase
9  # 0.4     Hydrolase
10 -> 1.2     ATPase
11 # 1.2     ATPase
12 $

```

The function in Terminal 143 is almost the same as the function in Terminal 142. However, now we use the parameter *pre*. The value of *pre* is assigned when we call the function with *out(" -> ")* in line 2. Thus, the value of *pre* becomes "*->* ". In line 3 we call the function with yet another parameter. In the *printf* command in line 1 the value of *pre* is used as a prefix for the output. The result is seen in lines 4–11.

Up to this point we simplified the world a little bit. We have said the function *out()* in Terminal 142 does not require any parameter. This is not completely true: it uses the variables *\$1* and *\$2* from the script. In fact, function parameters are local variables. If we define parameters they become local variables. A local variable is a variable that is local to a function and cannot be accessed or edited outside of it, no matter whether there is another variable with the same name in script or not. On the other hand, a global variable can be accessed and edited from anywhere in the program. Thus, each function is a world for itself. The following example illustrates this.

```

Terminal 144: Variables in Functions
1  $ awk 'BEGIN{a="alpha"
2  > print "a:", a
3  > fun(); print "b:", b}
4  > function fun(){print a; b="beta"}'
5  a: alpha
6  alpha
7  b: beta
8  $

```

The function `fun` defined in line 4 in Terminal 144 does not declare any parameters. Thus, there are no local variables. The function does have access to the variable `a` to which we assigned a value within the script in line 1. In the same way, the script has access to the variable `b` that has been assigned in the function body in line 4. Note that it makes no difference whether you define your function at the beginning, in the middle or at the end of your script. The result will always be the same.

The next two examples demonstrate the difference between local and global variables in more detail.

```

Program 36: fun_1.awk - Variables in Functions
1  # save as fun_1.awk
2  BEGIN{
3  a="alpha"; b="beta"
4  print "a:", a; print "b:", b
5  fun("new")
6  print "a:", a; print "b:", b
7  }
8  function fun(a,b) {
9  print "fun a:", a; print "fun b:", b
10 b="BETA"
11 }

```

In Program 36, we use the two variables `a` and `b` within the script and within the function `fun`. However, for the function `fun` the variables `a` and `b` are declared as parameters, and thus are local variables (line 8). Note that when we call the function in line 5 with the command `fun("new")`, we deliver only one parameter (`new`) to it. Terminal 145 shows what the program does.

```

Terminal 145: Variables in Functions
1  awk -f fun_1.awk
2  a: alpha
3  b: beta
4  fun a: new
5  fun b:
6  a: alpha
7  b: beta

```

In line 1 of Terminal 145 the program is executed. Lines 2 and 3 show the value of the variables `a` and `b`, respectively, as defined by lines 3 and 4 of Program 36. Line 5 of Program 36 calls the function `fun` and delivers the parameter “new” to it. This parameter is assigned to the local variable `a` of function `fun`. The local variable `b` of function `fun` remains unassigned. This means it is empty. In line 9 of Program 36 the content of the local variables is printed. The result can be seen in lines 4 and 5 of Terminal 145. As expected, local `a` has the value “new”, whereas local `b`

is empty. Although we assign the string “BETA” to *b* within the function `fun`, this value will neither leave the environment of the function nor change the value of global *b* (“beta”). This we check with line 6 of Program 36 on the previous page. The result of this line can be seen in lines 6 and 7 of Terminal 145.

Now, with the knowledge from Program 36 on the previous page and Terminal 145 in mind, let us take a look at Program 37.

```

_____ Program 37: fun_2.awk - Variables in Functions _____
1  # save as fun_2.awk
2  BEGIN{
3      a="alpha"; b="beta"
4      print "a:", a; print "b:", b
5      fun("new")
6      print "a:", a; print "b:", b
7  }
8  function fun(a) {
9      print "fun a:", a; print "fun b:", b
10     b="BETA"
11 }

```

The only difference to Program 36 is the declaration of parameters in line 8. The output of Program 37 is shown in Terminal 146.

```

_____ Terminal 146: Variables in Functions _____
1  awk -f fun_2.awk
2  a: alpha
3  b: beta
4  fun a: new
5  fun b: beta
6  a: alpha
7  b: BETA

```

The main message of Program 37 is that *a* is a local and *b* a global variable. Thus, *b* is accessible from anywhere in the script and can be edited from everywhere in the script.

Finally, what happens if we call a function with more parameters than are declared? Then we will receive an error message, because the undeclared parameter will end up nowhere.

The previous examples were educational exceptions. In larger programs you should always use local variables. Otherwise, it might become really tricky to find out where a potential bug in your program lies.

With the command `return` we can specify a return value for our function. This is illustrated in the next example.

```

_____ Terminal 147: Leaving Functions with return _____
1  $ awk '
2  function square(x,a){y=x**2+23x-a; return y}
3  {print $1 --- "$2" --- "square($2,5)}
4  ' enzyme.txt
5  Enzyme --- Km --- 18
6  Protease --- 2.5 --- 24.25
7  Hydrolase --- 0.4 --- 18.16
8  ATPase --- 1.2 --- 19.44
9  $

```

The `return` command in line 1 of Terminal 147 invokes that the function call (`square($2,5)`) in line 2 returns the value of `y`. Alternatively, without the `return` command, you need to call the function and then state that you want to print the value of `y`. Of course, this works only for a global variable `y`. This is shown in Terminal 148.

```

1  $ awk '
2  function square(x,a){y=x**2+23x-a}
3  {print $1" --- "$2" --- "square($2,5)y}
4  ' enzyme.txt
5  Enzyme --- Km --- 18
6  Protease --- 2.5 --- 24.25
7  Hydrolase --- 0.4 --- 18.16
8  ATPase --- 1.2 --- 19.44
9  $

```

What does our function `square` do? Well, it simply does some math and calculates the result of the quadratic equation from the `Km` values in column two of the file `enzyme.txt`. Why is the result for “`Km`” 18? Because AWK does strange things when calculating with characters. Therefore, you should check whether your variable really contains numbers and no characters.

```

1  $ awk '
2  > function square(x,a){
3  > if (x~/[A-Z]/){
4  >   return ("\"x\" no number")}
5  > else{
6  >   y=x**2+23x-a; return y}
7  > }
8  > {print $1" --- "$2" --- "square($2,5)}
9  > ' enzyme.txt
10 Enzyme --- Km --- "Km" no number
11 Protease --- 2.5 --- 24.25
12 Hydrolase --- 0.4 --- 18.16
13 ATPase --- 1.2 --- 19.44
14 $

```

Terminal 149 shows you one possible solution to this problem.

13.9 Input with `getline`

In all previous examples, AWK got its input from one or several input files. These were specified in the command line. With this technique we are restricted to access one input file at a time. With the command `getline` you can open a second input file. This feature expands AWK’s capabilities enormously; but `getline` is even more powerful, since you can use it in order to communicate with the shell. With this section we will round up our knowledge on AWK.

In general, `getline` reads a complete line of input (let it be from a file or the output from a shell command). Depending on the way you apply `getline`, the record is saved in one single variable or split up into fields. Furthermore, the value of different

built-in variables may be influenced, in particular the positional variables (`$0` and `$1...$n`) and the “number of fields” variable (`NF`). The return value of `getline` is 1 if it reads a record, 0 if it reached the end of a file or `-1` if an error occurred.

13.9.1 Reading from a File

Let us assume the following scenario. You have one file containing the names of enzymes and the value of their catalytic activity. In a second file, you have a list of enzyme names and the class to which they belong. Now you wish to join this information into one single file. How can you accomplish this task?

One way is to use MySQL (see Sect. 16.3.6 on p. 312). The AWK way is to use `getline`! In order to go through this example step by step, let us first create the desired files. The first file is actually *enzyme.txt*. You should still have it. The second file, we name it *class.txt*, has to be created.

```

Terminal 150: Content of enzyme.txt and class.txt
1  $ cat enzyme.txt
2  Enzyme      Km
3  Protease    2.5
4  Hydrolase   0.4
5  ATPase      1.2
6  $ cat > class.txt
7  Protease = Regulation
8  ATPase = Energy
9  Hydrolase = Macro Molecules
10 Hydrogenase = Energy
11 Phosphatase = Regulation
12 $

```

Terminal 150 shows the content of the example files we are going to use. Note that both files use different field separators: in *enzyme.txt* the fields are space delimited, whereas in *class.txt* the fields are separated by “`_`”.

Now, let us take a look at the program we are applying to solve the task.

```

Program 38: enzyme-class.awk - File Fusion
1  # save as enzyme-class.awk
2  # assigns a class to an enzyme
3  # requires enzyme.txt and class.txt
4  BEGIN{FS=" "
5      while ((getline < "class.txt") > 0){
6          class[$1]=$2}
7  FS="_"
8  }
9  {if (class[$1]==""){
10     print $0 "\tClass"}
11  else{
12     print $0 "\t" class[$1]}
13  }

```

Program 38 is executed with

```
awk-f enzyme-class.awk enzyme.txt
```

The program is separated into two main parts: a `BEGIN` block (lines 4–8) and the main body (lines 9–13). In line 4 we assign the string “`_=_`” to the field separator variable `FS`. Then we use a `while` loop, spanning from line 5 to 8. This loops cycles as long as the following condition is true:

```
(getline < "class.txt") > 0
```

What does this mean? With

```
getline < "class.txt"
```

we read one line of the file *class.txt*. Note that the “less” character (`<`) does not mean “less than”, but is used like redirections. The complete line is saved in `$0`. Then, the lines are split according to the value of the field separator `FS` and saved in `$1...$n`, where `n` is the number of fields (which is saved in `NF`). As said before, `getline` returns 1 if it could read a line, 0 if it reached the end of the file, and `-1` if it encountered an error. Thus, what we do in line 5 is to check whether `getline` reads a record. In line 6 we use an array called *class* (see Sect. 13.5.4 on p. 215). The enzyme name, saved in `$1`, is used as the array index, whereas the corresponding class, saved in `$2`, is used as corresponding array value. For example, the element *class*["ATPase"] has the value “Energy”. Now we know what the `while` loop is doing: it reads records from the file *class.txt* and saves its fields in the array called *class*. Finally, in line 7, we set back the field separator to the space character. Now, the main body of Program 38 on the previous page starts. We use no pattern; this means each line of the input file *enzyme.txt* is read. In line 9 we check if the value of the array element *class*[`$1`] is empty. For the first record of *enzyme.txt* the value of `$1` is “Enzyme”. However, the array element *class*["Enzyme"] is empty, because there is no corresponding line in the file *class.txt*. Thus line 8 would be executed. It prints the original record of *enzyme.txt* (saved in `$0`), plus two tabulators (`\t`), plus the word “Class”. Otherwise, if the array element is not empty, `$0`, plus two tabulators, plus the value of the array element *class*[`$1`] is printed. The resulting output is shown in Terminal 151.

```

1  $ awk -f enzyme-class.awk enzyme.txt
2  Enzyme      Km      Class
3  Protease    2.5    Regulation
4  Hydrolase   0.4    Macro Molecules
5  ATPase      1.2    Energy
6  $

```

In the previous example, the record caught by `getline` was saved in `$0` and split into the positional field variables `$1...$n`. Additionally, the variable `NF` (number of fields) was set accordingly. However, this might not be appropriate because under certain circumstances the record read from the file by `getline` would overwrite the record from the input file stated in the command line. In such cases, you can save the line read by `getline` in a variable. The syntax is:

```
getline Variable < "FileName"
```

The record is then saved in *Variable* and neither *\$0* nor *NF* are changed. Of course, the record is not split into fields either. However, that you could do with the `split` function (see Sect. 13.8.3 on p. 229).

The following example shows you how to save a record in a variable.

```

1  $ awk '
2  > {getline class < "class.txt"
3  > print class <> " $0}
4  > ' enzyme.txt
5  Protease = Regulation <> Enzyme      Km
6  ATPase = Energy <> Protease      2.5
7  Hydrolase = Macro Molecules <> Hydrolase    0.4
8  Hydrogenase = Energy <> ATPase      1.2
9  $
```

The script shown in Terminal 152 reads for each line of *enzyme.txt* one line of *class.txt* and prints out both. The line from *class.txt* is saved in the variable *class*.

13.9.2 Reading from the Shell

In the previous section, we used the redirection character “<” in order to read a file with `getline`. Guess what, with the pipe character “|” we can redirect the output of a command to `getline`. The syntax is

```
Command|getline Variable
```

The use of the variable is optional. If you do not use the variable, the record caught by `getline` is saved in *\$0* and *NF* is set to the number fields saved in *\$1...\$n*. Splitting is executed according to the value of the field separator variable *FS*. All these variables remain unchanged, if you use a variable.

Now, let us take a look at a real-world example: we want to read the output of the shell’s `date` and `pwd` commands.

```

1  $ awk 'BEGIN{
2  > "date" | getline
3  > "pwd" | getline pwd
4  > print "Date & Time: " $0
5  > print "Current Directory: " pwd}'
6  Date & Time: Fre Jun 27 13:40:27 CEST 2003
7  Current Directory: /home/Freddy/awk
8  $
```

Terminal 153 shows the application of `getline` with and without a variable. In line 2, the output of the shell command `date` is saved in the positional variable *\$0*, whereas line 3 assigns the output of the shell command `pwd` to the variable *pwd*.

What happens if the shell command produces a multiple-line output? Then we need to capture the lines with a `while` loop.

```

Terminal 154: Reading From Multiple Files with getline
1  $ awk 'BEGIN{
2  > while ("ls" | getline >0){
3  >   print $0
4  > } }'
5  class.txt
6  enzyme-class.awk
7  enzyme.txt
8  genomes2.txt
9  structure.pdb
10 $

```

The shell command `ls` is a good example of a command often producing an output several lines long. Terminal 154 gives you an example of how a `while` loop can be used to catch all lines. Remember: `getline` returns 1 if a record was read. Thus, the loop between lines 2 and 4 cycles as long as there is still a record to read.

13.10 Produce Nice Output with L^AT_EX

If you are already an advanced user and familiar with L^AT_EX, you might like to include AWK or shell output in L^AT_EX documents. Program 39 shows you how.

```

Program 39: call-shell.tex
1  \documentclass{article}
2  \begin{document}
3  \section*{Demonstration of \texttt{$\backslash$input\{}}
4  This is the result of a \LaTeX\ document.
5
6  The result of \verb+\input{"date"}+ is: \input{"date"}. 7
7  The result of \verb+\input{"date" | awk '{print $1}'+ is:
8  \input{"date" | awk '{print $1}'}.
9  \end{document}

```

You call the script with

```
pdflatex -shell-escape call-shell.tex
```

from the Bash shell. The magic command is `\input{ | ... }`, which includes the return value of the Bash command in a L^AT_EX document. The resulting PDF file is shown in Fig. 13.1.

Demonstration of `\input{}`

This is the result of a L^AT_EX document.

The result of `\input{"date"}` is: Sat Sep 22 21:40:42 CEST 2012 .

The result of `\input{"date" | awk '{print $1}'}` is: Sat .

Fig. 13.1 L^AT_EX. Importing AWK and Bash output to a L^AT_EX document

13.11 Examples

This section contains some examples which are to invite you to play around with AWK scripts. In Part VI on p. 343 and in particular in Chap. 21 on p. 409 you will find more elaborate examples.

13.11.1 Sort an Array by Indices

With the `asort` command from Sect. 13.5.4 on p. 215 we can sort an array according to its values but not to its indices. This small script demonstrates how to circumvent this obstacle. The script is started with

```
awk -f sort-array-elements.awk enzyme.txt
```

Note that we are using the file *enzyme.txt*.

```

1  # save as sort-array-elements.awk
2  # sorts an array by value of
3  # array elements
4  /[0-9]/ {data[$1]=$2; j=1}
5  END{
6    j=1
7    for (i in data) {ind[j]=i; j++}
8    n=asort(ind)
9    for (i=1; i<=n; i++) {print ind[i], data[ind[i]]}
10 }
```

The trick in Program 40 is that we generate a second array *ind* that contains the indices of the array *data*. Then *ind* is sorted and used to sort *data*. Take your time to get through and understand the program.

13.11.2 Sum up Atom Positions

The following script uses the File 4 on p. 161 (*structure.pdb*). It sums up the decimal numbers of the atom positions. Okay, I admit that this program does not really make sense. However, it is good for learning.

```

_____ Program 41: atom-sum.awk - Add Atom Positions _____
1  # save as atom-sum.awk
2  # uses the file structure.pdb
3  # sums up the values in the ATOM line
4  /^ATOM/ {
5    for (i=1; i<=NF; i++){
6      if ($i ~ /[0-9].[0-9]/){
7        sum+=$i
8      }
9    }
10   print "Line "NR, "    Sum= "sum
11 }

```

You execute Program 41 with

```
awk -f atom-sum.awk structure.pdb
```

The program file and the structure file are required to be in the same directory. The `for` loop in line 5 runs for every field in the current record. The following `if` statement checks whether the current field contains a point-separated number. If so, the number is added to the variable *sum* and printed in line 10, together with the line number.

13.11.3 Convert FASTA ↔ Table Formats

FASTA is a well-established file format in life sciences, because it is computer readable. Each line starting with `>` indicates the annotation line. What follows is the sequence in one or more lines. To work on a nucleotide or protein sequence file with AWK, it would be much nicer if the complete sequence would sit in one line as one field. Program 42 converts a (multiple)FASTA file into a tabular file. The tabular file contains one sequence per line. The annotation precedes the sequence and is separated from the sequence by a tabulator.

```

_____ Program 42: fasta2tbl.awk - Convert FASTA to Table File _____
1  # Save as fasta2tbl.awk
2  # Usage: awk -f fasta2tbl.awk inputfile.fasta
3  BEGIN{ORS="" }
4  {
5    if (substr($0,1,1)==">"){
6      if (NR==1){print substr($0,2,length($0)-1)"\t"}
7      else{print "\n"substr($0,2,length($0)-1)"\t"}
8    }
9    else{print $0}

```

```

10 }
11 END{print "\n"}

```

Thus, if you like to work on a tabular sequence file, you must set the field separator variable *FS* to `\t`. How does *fasta2tbl.awk* work? The `BEGIN` block sets the output record separator to an empty string (default is line break). Then we extract the first character of *\$0* with `substr` (see Sect. 13.8.3) and check if it is equal to `>`. If this is true and if we are at the first record, we print *\$0* from the second character to the end, else we proceed this string by a newline character. In all other cases we just print *\$0*, which is a sequence line.

Program 43 converts a tabular file back into a FASTA file.

```

----- Program 43: tbl2fasta.awk - Convert Table to FASTA File -----
1  # Save as tbl2fasta.awk
2  # Usage: awk -f tbl2fasta.awk inputfile.tbl
3  BEGIN{FS="\t"}
4  {
5    i=1
6    print ">"$1
7    while (i<=length($2)){
8      print substr($2,i,80)
9      i=i+80
10   }
11 }

```

In the `BEGIN` block we set the input field separator to a tabulator, because this is what separates the annotation from the sequence. Thus, field 1 becomes the annotation, which is printed preceded by a `>` character. Then we loop through field 2 and split it into 80 character pieces—this is for the sequence.

I set for both functions *aliases* on my system because I use them very often (see Sect. 8.8 on p. 104).

13.11.4 Mutate DNA Sequences

Script 44 mutates DNA sequences in such a tab file. In every 1–10 nucleotides a base might be changed. This example shows you why it might be handy to have a sequence file in tabular format as shown in the previous example (see Sect. 13.11.3).

```

----- Program 44: mutator.awk - Mutate DNA -----
1  # store as mutator.awk
2  # execute as: awk -f mutator.awk infile.tab
3  # infile.tab must be tab delimited: ID \t SEQ
4  BEGIN{ srand(); FS="\t"; ORS="" }
5  {
6    split(tolower($2),dna,"")
7    for(i=1;i<=length(dna);i++){
8      r=int((rand()*4)+1); f=int((rand()*10)+1)
9      i=i+f
10     if(r==1){r="A"}; if(r==2){r="T"}; if(r==3){r="G"}; if(r==4){r="C"}
11     dna[i]=r

```

```

12     }
13     print $1"\t"
14     for(i=1;i<=length(dna);i++) {print dna[i]}
15     print "\n"
16 }

```

What does the file do? In the BEGIN block we initiate the random number generator, set the field separator to tabulator and the output record separator to new line character. The `split()` command in line 6 splits the sequence in field \$2 at every character (" ") into an array named *dna*. In addition, \$2 is set to low case with function `tolower()`. Line 7 initiates a loop that loops through all arrays elements, i.e. all nucleotides. Line 8 generates to random numbers. *r* ranges from 1 to 4 and determines the nucleotide, while *f* ranges from 1 to 10 and determines the jump length to the next nucleotide to be mutated. The actual mutation takes place in line 11. Line 14 prints the mutated sequence.

13.11.5 Translate DNA to Protein

The following program translates a DNA sequence into the corresponding protein sequence. The program is executed with

```
awk -f dna2prot.awk
```

When executed without input filename, the program accepts input from the command line. Your input will be translated when you hit **Enter**. Then you can enter a new DNA sequence. To abort the program press **Ctrl+C**. Of course, you could also pipe a sequence to the program or load a file containing a DNA sequence.

```

----- Program 45: dna2prot.awk - Translate DNA -----
1  # save as dna2prot.awk
2  # translates DNA to protein
3  BEGIN{c["atg"]="MET"; c["ggc"]="THY"; c["ctg"]="CYS"}
4  {i=1; p=""}
5  {do {
6      s=substr($0, i, 3)
7      printf ("%s ", s)
8      {if (c[s]=="") {p=p"    "} else {p=p c[s]" "}}
9      i=i+3}
10 while (s!="")}
11 {printf("\n%s\n", p)}

```

Let us shortly formulate the problem: the input will be a DNA sequence, the output is to be a protein sequence. Thus, we need to chop up the sequence string into triplets. This we do with the function `substr`, as shown in line 6 of Program 45. The codon is saved in the variable *s*. Then we look up if the value of *s* is in our translation table (line 8). The translation table is in an associative array called *c*. If the codon in *s* does not exist in the array *c* (thus, *c[s]* is empty), four spaces are added to the variable *p*, or else, the value of *c[s]* is added to *p*. We repeat this process as long as there

are codons (*s* must not be empty). All the rest is nice printing. Note that we use the “do...while” construct here. If we started with a check if *s* is empty, the loop would never start because it is initially empty!

13.11.6 Calculate Atomic Composition of Proteins

The following program is a result of my current research. As part of this, we need to know the atomic composition of a set of proteins which is saved in a FASTA formatted file. The result should be in FASTA format, too. The atomic composition of the 20 proteinogenic amino acids is available from a file called *aa_atoms.txt* (see File 7).

```

_____ File 7: aa_atoms.txt _____
1 Name Symbol Code Mass(-H2O) SideChain Occ.(%) S N O C H
2 Alanine A Ala 71.079 CH3- 7.49 0 0 0 1 3
3 Arginine R Arg 156.188 HN=C(NH2)-NH-(CH2)3- 5.22 0 3 0 4 10
4 Asparagine N Asn 114.104 H2N-CO-CH2- 4.53 0 1 1 2 4
5 AsparticAcid D Asp 115.089 HOOC-CH2- 5.22 0 0 2 2 3
6 Cysteine C Cys 103.145 HS-CH2- 1.82 1 0 0 1 3
7 Glutamine Q Gln 128.131 H2N-CO-(CH2)2- 4.11 0 1 1 3 6
8 GlutamicAcid E Glu 129.116 HOOC-(CH2)2- 6.26 0 0 2 3 5
9 Glycine G Gly 57.052 H- 7.10 0 0 0 0 1
10 Histidine H His 137.141 N=CH-NH-CH=C-CH2- 2.23 0 2 0 4 5
11 Isoleucine I Ile 113.160 CH3-CH2-CH(CH3)- 5.45 0 0 0 4 9
12 Leucine L Leu 113.160 (CH3)2-CH-CH2- 9.06 0 0 0 4 9
13 Lysine K Lys 128.17 H2N-(CH2)4- 5.82 0 1 0 4 10
14 Methionine M Met 131.199 CH3-S-(CH2)2- 2.27 1 0 0 3 7
15 Phenylalanine F Phe 147.177 (C6H5)-CH2- 3.91 0 0 0 7 7
16 Proline P Pro 97.117 -N-(CH2)3-CH- 5.12 0 0 0 3 6
17 Serine S Ser 87.078 HO-CH2- 7.34 0 0 1 1 3
18 Threonine T Thr 101.105 CH3-CH(OH)- 5.96 0 0 1 2 5
19 Tryptophan W Trp 186.213 (C6H5)-NH-CH=C-CH2- 1.32 0 1 0 9 9
20 Tyrosine Y Tyr 163.176 (C6H4OH)-CH2- 3.25 0 0 1 7 7
21 Valine V Val 99.133 CH3-CH(CH2)- 6.48 0 0 0 3 6

```

This file gives more information than we currently need. The first line specifies the content of the space-separated fields in each line. The first fields contain the full amino acid name, followed by its one-letter and three-letter code, the molecular mass excluding one water molecule (which is split off during the formation of the peptide bound), the side chain composition, its natural abundance in nature and finally the number of sulfur (S), nitrogen (N), oxygen (O), carbon (C), and hydrogen (H) atoms of the side chain. Our program first reads the information of this file and then loads the protein sequences from the FASTA file. The corresponding filename is given as command line parameter. The full code is given in the following Program.

```

_____ Program 46: aacomp.awk - Protein Atomic Composition _____
1 # save as aacomp.awk
2 # calculates atomic composition of proteins
3 # usage: awk -f aacomp.awk protein.fasta
4
5 # ASSIGN DATA ARRAY
6 BEGIN {
7   while (getline < "aa_atoms.txt" > 0) {
8     # read number of S,N,O,C,H:

```

```

 9      data[$2]=$4 " " $7 " " $8 " " $9 " " $10 " " $11
10      }
11  }
12  # DECLARE FUNCTION to calculate fraction
13  function f(a,b){
14      if (a != 0 && b != 0){
15          return a/b*100
16      }
17  }
18  # PROGRAM BODY
19  {
20      if ($0 ~ /^>/) {
21          print L, MW, S, N, O, C, H, "-",
22                f(S,t), f(N,t), f(O,t), f(C,t), f(H,t)
23          print $0
24          t=0; L=0; MW=0; S=0; N=0; O=0; C=0; H=0
25      }
26      else {
27          i=1; L=L+length($0); tL=tL+length($0)
28          do {
29              aa=substr($0,i,1)
30              i++
31              # begin main
32              split(data[aa],comp) # comp[1]=mass...
33              MW=MW+comp[1]; S=S+comp[2]; N=N+comp[3]
34              O=O+comp[4]; C=C+comp[5]; H=H+comp[6]
35              tMW=tMW+comp[1]; tS=tS+comp[2]; tN=tN+comp[3]
36              tO=tO+comp[4]; tC=tC+comp[5]; tH=tH+comp[6]
37              T=tS+tN+tO+tC+tH; t=S+N+O+C+H
38              # end main
39          }
40          while (aa != "")
41      }
42  }
43  END {
44      print L, MW, S, N, O, C, H, "-",
45            f(S,t), f(N,t), f(O,t), f(C,t), f(H,t)
46      print "\n>SUMMARY"
47      print tL, tMW, tS, tN, tO, tC, tH, "-",
48            f(tS,T), f(tN,T), f(tO,T), f(tC,T), f(tH,T)
49  }

```

Okay, I agree, this is a rather large program. Let us go through it step by step. As usual, all lines beginning with the hash character are comments. In Program 46, we use quite a number of them in order to keep track of its parts.

The BEGIN body spans from lines 6 to 11. Here we read the data file *aa_atoms.txt*. Take a look at Sect. 13.9.1 on p. 239 for a detailed explanation of the `while` construct. The second field of the file *aa_atoms.txt*, which is available via the variable `$2`, contains the single-letter amino acid code. This we use as index for the array *data* in line 9. Each array element consists of a space-delimited string with the following information: molecular weight and the number of sulfur, nitrogen, oxygen, carbon, and hydrogen atoms of the side chain. Thus, for alanine *data* would look like:

```
data["A"] = "71.079 0 0 0 1 3"
```

In line 13 we declare a function. Remember that it does not make any difference where in your program you define functions. The function uses the two local variables

a and b . The function basically returns the ratio of $a-b$ as defined in line 15 of Program 46.

The main body of the program spans from lines 19 to 42. In line 20, we check if the current line of the input file starts with the `>` character. Since that line contains the identifier, it is simply printed by the `print` command in line 23. Before that, the data of the preceding protein are printed (lines 21 and 22), i.e. the length (L), molecular weight (MW), count of atoms (S , N , O , C , H), a dash, and the fraction of the atom counts. This is accomplished by calling the function `f` with the number of atoms (S , N , O , C , H) and the total number of side chain atoms of the protein (t) in line 22. Since there is no protein preceding the first protein (wow, what an insight) we will get the dash only when the program passes these lines for the very first time (see line 2 of Terminal 155 on p. 250). For the same reason, we check in line 13 whether any parameter handed over to the function equals zero; otherwise, we might end up with a division by zero. In line 24 of Program 46 we set all counters to zero. If the current line does not start with the `>` character, we end up in the `else` construct starting in line 26. A local counter i is set to 1 and the number of amino acids in the current line [`length($0)`] is added to the counter variables L and tL . While the value of L reflects the length of the current protein, tL is counting the total number of amino acids in the file (this is why tL is not reset in line 24). Now we reach the kernel of the program. The `do...while` construct between lines 28 and 40 of Program 46 is executed as long as there are amino acids in the active line. These are getting less and less because in line 29 we extracted them one by one with the `substr` function (see also Sect. 13.8.3 on p. 229). The value of the variable aa is the current amino acid. In line 32, we use the `data` array we generated in line 9 and apply the current amino acid (sitting in aa) as key. The array elements are split with the `split` function (see also Sect. 13.8.3 on p. 229) into the array `comp`. The default field delimiter used by the `split` function is the space character. This is why we introduced space characters in line 9. The result of this is that `comp[1]` contains the molecular weight, `comp[2]` the number of sulfur atoms, and so forth. These data we add to the corresponding variables in lines 33–36.

Finally, we apply the `END` construct to print out the data of the last protein in lines 44 and 45 (which resemble lines 21 and 22) and the cumulative data in lines 47 and 48 of Program 46.

Now let us see how the program performs. As an input file you can use any file containing protein sequences in FASTA format. File 8, named *proteins.seq*, is one possible example.

File 8: *proteins.seq*

```

1 >seq1
2 MRKLVFSDTERFYRELKTALAQGEEVEVITDYERYSDLPEQLKTIFELHKNKSGMWVNV
3 TGAFIPYSFAATTINYSALYIFGGAGAGAIFGVIVGGPVGAAGGGGIGAIVGTAVATL
4 GKHHVDIEINANGKLRFKISPSIK
5 >seq2
6 MVAQFSSSTAIAGSDSFDIRNFDQLEPTRVKNKYICPVCGGHNSINPNNGKYSYNE
7 LHRDIREAIKPWTQVLEERKLGSTLSPKPLPIKAKKPATVPKVLDDVPSQLRICLLSGE
8 TTPQPVTPDFVPKSVAIRLSDSGATSQLKEIKEIEYDYGNGRKAHRFSCPCAAAPKGR

```

```

9 | KTFSVSRIDPITNKVAWKKEGFWPAYRQSEATAIIKATDGIPVLLAHEGEKCVEASRL
10 | LASITWVGSSSDRDILHSLTQIQHSTGKDFLLAYCVDNDSTGWNKQQRKEICQQAGVS

```

The program is executed as shown in Terminal 155. Please note that lines 4–5 and 7–8 comprise usually one line of output.

```

Terminal 155: Output of aacomp.awk
1 | $ awk -f aacomp.awk proteins.seq
2 | -
3 |
4 | >seq1
5 | 142 15179.5 2 38 55 411 794 - 0.153846 2.92308 4.23077
6 | 31.6154 61.0769
7 | >seq2
8 | 295 32608.2 10 110 141 852 1718 - 0.353232 3.88555 4.98057
9 | 30.0954 60.6853
10 |
11 | >SUMMARY
12 | 437 47787.7 12 148 196 1263 2512 - 0.290487 3.58267 4.74461
13 | 30.5737 60.8085
14 | $

```

As stated above, we first get one line containing only a dash. Then follow the data for all proteins. Finally, we get the summary for all proteins. This is as if we would concatenate all proteins to one and perform the calculation. The order of the output is: protein length in amino acids, molecular weight, number of sulfur, nitrogen, oxygen, carbon, and hydrogen atoms in the side chain, a dash, and the fraction of sulfur, nitrogen, oxygen, carbon, and hydrogen atoms.

13.11.7 Dynamic Programming: Levenshtein Distance of Sequences

This is probably the most difficult example of the whole book. Take your time and go through this example slowly and with concentration—then you will master it! An important task in biology is to measure the sequence difference between nucleotide or protein sequences. With a reasonable quantitative measure in hand we can then start to construct phylogenetic trees. Therefore, each sequence is compared with each sequence. The obtained distance measures would be filled into a so-called distance matrix, which in turn would be the basis for the construction of the tree. What is a reasonable distance measure? One way is to measure the dissimilarity between sequences as the number of editing steps to convert one sequence into the other. This resembles the action of mutations. Allowed editing operations are deletion, insertion, and substitution of single characters (nucleotides or amino acids) in either sequence. Program 47 on p. 252 calculates and returns the minimal number of edit operations required to change one string into another. The resulting distance measure is called Levenshtein distance. It is named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965 (Levenshtein 1966). The greater the distance, the more different the strings (sequences) are. The beauty of

this system is that the distance is intuitively understandable by humans and computers. Apart from biological sequence comparison, there are lots of other applications of the Levenshtein distance. For example, it is used in some spell checkers to guess which word from a dictionary is meant when an unknown word is encountered. This function is implemented in many programs and you surely know it from Google.

How can we measure the Levenshtein distance? The way in which we will solve the problem to calculate the Levenshtein distance is called *dynamic programming*. Dynamic Programming refers to a very large class of algorithms. The basic idea is to break down a large problem into incremental steps so that, at any given stage, subproblems with subsolutions are obtained. Let us assume we have two sequences: ACGCTT (sequence 1) and AGCGT (sequence 2). The algorithm is based on a 2D matrix (array) as illustrated in Fig. 13.2.

The array rows are indexed by the characters of sequence 1 and the columns are indexed by the characters of sequence 2. Each cell of the array contains the number of editing steps needed to convert sequence 1 into sequence 2 at the actual cell position. In other words; each cell $[row, col]$ (row and col being the row and column index, respectively) represents the minimal distance between the first row characters of sequence 1 and the first col characters of sequence 2. Thus, cell $[2, 2]$ contains the number of editing steps needed to change AC to AG, which is 1 (convert C to G), and cell $[2, 4]$ contains the number of step to convert AC to AGCG, which is 2 (delete the Gs). How do we obtain these numbers? During an initialization step column 0 is filled with numbers ranging from 1 to the length of sequence 1 (red shadowed in Fig. 13.2). This corresponds to the number of insertions needed if sequence 2 was zero characters long. Equally, the row 0 is filled with numbers ranging from 1 to the length of sequence 2. This corresponds to the number of deletions if sequence 1 was zero characters long. Now, the algorithm starts in the upper left-hand

		<i>b</i>				
		A	G	C	G	T
	0	1	2	3	4	5
A	1	①	1	2	3	4
C	2	1	1	1	2	3
G	3	2	1	2	1	2
C	4	3	2	1	2	2
T	5	4	3	2	2	2
T	6	5	4	3	3	②

$m[5,2]$

Fig. 13.2 Dynamic Programming. Construction of a matrix to calculate the Levenshtein distance between two strings. The DNA sequences, saved in variables a and b , respectively, are written along both sides of a matrix. The cells *underlaid in red* are filled in the initialization step of the algorithm (lines 13 and 14 in Program 47). Then, the cell values are *calculated from the top left to the bottom right* (the start and end cell are *encircled*). For example, the value of cell $[5, 2]$, saved in the 2D array m , depends on the content of the *upper* and *left* three neighbors. The Levenshtein distance between the two sequences can be read directly from the *bottom right cell*—here the distance is 2

corner of the array (cell [1, 1]). This cell is filled with the minimum of the following evaluation: (a) the value of the cell above plus 1, (b) the value of the left cell plus 1, or (c) the value of the upper left diagonal cell plus 0 if the corresponding sequence characters match or plus 1 in case they do not match. Mathematically speaking, we are looking for

$$cell[row, col] = \min \begin{cases} cell[row - 1, col] + 1 \\ cell[row, col - 1] + 1 \\ cell[row - 1, col - 1] + 1 & \text{if characters do not match} \\ cell[row - 1, col - 1] & \text{if characters do match} \end{cases}$$

where *row* and *col* are the current row and column, respectively. The number added to the cell value is called the *cost* for the editing step. In our example the cost for insertions, deletions, and transformations is the same. For the case of sequence mutations this would correspond to the assumption that insertions, deletions, and transformations are equally likely to occur, which, of course, is a simplification. Note that the cells are treated from the top left to the bottom right. A cell can only be filled when the values of all its upward and left neighbors have been filled. The bottom right cell will contain the Levenshtein distance between both sequences. Now let us take a look at the program.

```

1  # save as levenshtein.awk
2  # calculates Levenshtein distance between two text strings
3  # usage: awk -f levenshtein.awk STRING1 STRING2
4
5  BEGIN{
6    one = ARGV[1]; two = ARGV[2]
7    print one " <==> "two;
8    print "Levenshtein Distance: "distance(one, two)
9  }
10 function distance (a, b){
11   la = length(a); lb = length(b)
12   if(la==0){return lb}; if(lb==0){return la}
13   for(row=1;row<=la;row++){m[row,0] = row}
14   for(col=1;col<=lb;col++){m[0,col] = col}
15   for(row=1;row<=la;row++){
16     ai = substr(a,row,1)
17     for(col=1;col<=lb;col++){
18       bi = substr(b,col,1)
19       if(ai == bi){cost = 0}
20       else{cost = 1}
21       m[row,col]=min(m[row-1,col]+1,m[row,col-1]+1,m[row-1,col-1]+cost)
22     }
23   }
24   return m[la,lb]
25 }
26 function min (a, b, c){
27   result = a; if(b < result){result = b}; if(c < result){result = c}
28   return result
29 }

```

In line 6 of Program 47, we save the sequences obtained from the command line the variables *one* and *two*, respectively. Therefore, we employ the array *ARGV* (see

Sect. 13.5.3.4 on p. 213). In line 8, we call the subroutine *distance* with both sequences as parameters. The subroutine itself starts in line 10. In line 11, the parameters *one* and *two* are saved in the variables *a* and *b*. In line 11, we determine the length of *a* and *b* and save the values in *la* and *lb*, respectively. Now we check if one of the sequences is zero characters long (line 12). In that case, the Levenshtein distance equals the length of the existing sequence. In lines 13 and 14 the first row and column are initialized (see Fig. 13.2 on p. 251). Take a look at Sect. 13.7.4 on p. 223 to recall the syntax of the `for` construct. The `for` loops in lines 15 and 17 step through each row and column element. In lines 16 and 18, the characters corresponding to the actual matrix element are extracted from the sequences saved in *a* and *b* (check Sect. 13.8.3 on p. 229 to recall the syntax of the `substr` command). If these characters match (checked in line 19), then *cost* is set to 0, or else to 1. The construct in line 21 assigns a value to the current cell *[row, col]* saved in the array *m*. Therefore, the subroutine *min* is applied. This subroutine, spanning from lines 26 to 29, simply returns the smallest of three numbers. What is exactly is happening in line 21? The cost for an insertion or deletion is always 1. Thus, 1 is added to values above the current cell (*col-1*) and left of the current cell (*row-1*). Depending on the value of *cost*, 1 or 0 is added to the cell diagonally above and to the left of the current cell (*row-1,col-1*). The minimum of these three numbers is assigned to the current cell *[row, col]* in array *m*. In this way, the program surfs through the matrix. The last cell to which a value is assigned is the bottom right one (see Fig. 13.2 on p. 251). Its value corresponds to the Levenshtein distance of both strings and is the return value (line 24) of the subroutine *distance*. Execution of the program is shown in Terminal 156.

```

1  $ awk -f levenshtein.awk atgctatgtcgtgg tcacgtacgtacg
2  atgctatgtcgtgg <==> tcacgtacgtacg
3  Levenshtein Distance: 7
4  $

```

How about the complexity of our algorithm? Well, assuming that the length of each sequence is *n*, the running time as well as the memory demand for the matrix is n^2 . Finally, a little extension for our program that should help you to understand calculating the Levenshtein distance. Add the command `print` after lines 15 and 23 and add the command `printf ("%3s", m[row,col])` after line 21, respectively. Then, the program prints out the matrix you know from Fig. 13.2 on p. 251 as shown in the following Terminal.

```

1  $ awk -f levenshtein-book.awk ACGCTT ACGCT
2  ACGCTT <==> ACGCT
3
4      0  1  2  3  4
5      1  1  1  2  3
6      2  1  2  1  2
7      3  2  1  2  2
8      4  3  2  2  2
9      5  4  3  3  2
10     Levenshtein Distance: 2
11     $

```

I hope you liked our excursion to dynamic programming and all the other examples! I know it is sometimes a hard job to read other people's programs, but take your time and go through the examples. This is the best way to learn AWK. Learning is largely a process of imitating. Only when you can reproduce something can you start to modify and develop your own stuff.

Exercises

Do some exercises which are hopefully not awkward. For the exercises, create two files, one containing the line "1,2,3,4,5" and another containing the line "one;two;three;four;five". Name them *numbers.txt* and *words.txt*, respectively.

13.1. Reverse the order of the numbers in *numbers.txt* and separate them with dashes (-).

13.2. Print the first line of *numbers.txt* followed by a column (:), followed by the first line of *words.txt*. Do this for all lines in both files and save the result in *number-words.txt*. The first line of the output should look like this: "1 : one".

13.3. Design an AWK script that creates the following output:

```
1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

Use the `printf` command to format the output.

13.4. In this exercise, you are going to use the file *genomes2.txt* from Terminal 100 on p. 201. Add one field saying how many base pairs an average gene has.

Chapter 14

Perl

We have to deal with Perl! Why? Because it is a great and, especially among scientists, widely applied programming language—but not least also for historical reasons: Perl was initially developed to integrate features of Sed and AWK within the framework provided by the shell. As you learned before, AWK is a programming language with powerful string manipulation commands and regular expressions that facilitate file formatting and analysis. The stream editor Sed nicely complements AWK. In 1986, the system programmer Larry Wall worked for the US National Security Agency. He was responsible for building a control and management system with the capability to produce reports for a wide-area network of Unix computers. Unhappy with the available tools, he invented a new language: Perl. Besides integrating Sed and AWK he was inspired by his background in linguistics to make Perl a “human”-like language, which enables the expression of ideas in different ways. This feature makes Perl scripts sometimes hard to read because there are too many ways to do one thing. The first version of the *Practical Extraction and Report Language*, Perl, was released as open source in 1987. Since then, Perl has been growing and growing and growing—now even providing tools for biologists in the form of special commands via *Bioperl* (see Sect. 14.13 on p. 285).

14.1 Intention of This Chapter

This chapter is not intended to be a complete guide to Perl. There are some really good books on this topic available, even some especially for biologists: James Tisdall (2001) gives a very nice introduction to Perl, whereas Rex A. Dwyer’s book (2003) requires some prior knowledge on programming. Both books are based on examples taken from biology. As a general introduction to Perl, I liked to work with the book from Deitel et al. (2001); but there are many others around!—In the course of the previous chapters you have learned how to work with Linux and how to program with AWK. This chapter is only going to show you the most important features of

Perl and its syntax. Do you remember what I said in Chap. 13 on p. 197: you must learn only one programming language in order to understand almost all programming languages. The rest is a question about the right syntax. I can highly recommend you to buy one of these little pocket-sized *Pocket Reference* books on Perl (Vromans 2000). An old version is all you need. It provides a good compilation of all available commands and costs only around 5 Euro or US\$.

14.2 Running Perl Scripts

Guess what: the command in order to execute Perl scripts is “Perl”. Like AWK, you can either run small scripts from the command line:

```
perl-e 'print "Hello World\n"
```

or execute perl scripts saved in a file

```
perl scriptfile.pl
```

or make a script file executable:

```
Terminal 158: Perl Executable
1  $ cat > execute.pl
2  #!/bin/perl
3  # save as execute.pl
4  print "Hello World\n";
5  $ chmod u+x execute.pl
6  $ ./execute.pl
7  Hello World
8  $
```

In the latter case, the script file must begin with “# ! /bin/perl”.

You might have noticed the differences to AWK. There is no option necessary to execute a script file, while you need the option `-e` in order to run a command line script.

Note that each line in script files ends with the semicolon character!

14.3 Variables

In Perl, scalar variables are preceded by the dollar character. The only exception is the assignment of arrays and hashes (see below). Perl discriminates between three different variable types: *scalar variables*, *arrays*, and *hashes*. Scalar variables are normal variables having either a text string or a number as values. In Sect. 13.5.4 on p. 215 you learned what arrays are. An array variable stores a list of scalar variables. Each individual array *element* has an *index* and a *value*. The following array named

x consists of three elements with different values:

```
x[0]="bacteria"  x[1]="245.73"  x[2]="\n"
```

The first element has the index “0”. If the index was a text instead of a number, we are talking about associative arrays or hashes:

```
x["ATG"]="start"  x["TGT"]="Cysteine"  x["TGA"] ="stop"
```

While AWK did not discriminate between arrays and hashes, Perl does, as you will see soon.

Variables are very flexible in Perl. The value of a variable is interpreted in a certain context. Thus, the value might be interpreted either as a number or a text.

```

1  $ perl -e '$x=7;
2  > print 1+$x, "\n";
3  > print "1+$x\n" '
4  8
5  1+7
6  $
```

In Terminal 159 we assign the value “7” to the variable called x . In the arithmetic calculation in line 2, the value of x is considered to be a number. This is also called a scalar context. In contrast, in line 3, x is considered to be a text character because it is enclosed in double quotes. It is important to keep this in mind.

14.3.1 Scalars

The names of scalar variables always begin with the dollar character (\$).

```

1  $ perl -e '$x="20"; print "$x\n" '
2  20
3  $
```

Terminal 160 shows you how to assign a value to, and print the value of, a scalar variable. Of course, you can do all kinds of arithmetic calculations (+, -, *, /, **), increment (++) and decrement (--) numeric values, and use arithmetic operators for the assignment of scalar variables. The following list gives you an overview of the most common assignment operators for numeric scalar variables:

$\$a += x$	Addition: $\$a = \$a + x$
$\$a -= x$	Substraction: $\$a = \$a - x$
$\$a *= x$	Multiplication: $\$a = \$a * x$
$\$a /= x$	Division: $\$a = \$a / x$

<code>\$a **= x</code>	Exponentiation: <code>\$a = \$a ** x</code>
<code>\$a++</code>	Post-increment: first use the value of <code>\$a</code> in the expression in which <code>\$a</code> appears, then increment <code>\$a</code> by 1.
<code>++\$a</code>	Pre-increment: first increment <code>\$a</code> by 1, then use the new value of <code>\$a</code> in the expression in which <code>\$a</code> appears.
<code>\$a--</code>	Post-decrement: first use the value of <code>\$a</code> in the expression in which <code>\$a</code> appears, then decrement <code>\$a</code> by 1.
<code>--\$a</code>	Pre-decrement: first decrement <code>\$a</code> by 1, then use the new value of <code>\$a</code> in the expression in which <code>\$a</code> appears.

In summary, apart from the dollar character preceding variable names, the handling of scalar variables is pretty much the same as in shell programming or AWK.

14.3.2 Arrays

Arrays allow you to do almost everything with text strings and are one of the most frequently used and most powerful parts of Perl. The name of an array variable is preceded with the vortex character (@) when assigned and when the whole array is recalled but with the dollar character (\$), when a single element is recalled.

```

1  $ perl -e '@files=ls'; print "@files"
2  absolute.pl
3  age.pl
4  circle.pl
5  $ perl -e '@files=ls'; print "$files[1]"
6  age.pl
7  $

```

In the example in Terminal 161 the output of the shell command `ls` is assigned to the array named `files`. In line 1 all array elements are printed, while line 5 prints only the second array element. Remember that the first array element has the index 0.

14.3.2.1 Assigning Array Elements

There are different ways in which an array can be filled with values.

```

1  $ perl -e '$a[2]=4; print "$a[2]\n"'
2  4
3  $ perl -e '@a=(DNA, RNA, Protein); print "$a[1]\n"'
4  RNA
5  $ perl -e '@a=(DNA, RNA, Protein); print "@a[1,2]\n"'
6  RNA Protein
7  $

```

One way is to assign each element individually. This is shown in line 1 of Terminal 162. Another way is to assign a *list* to an array. A list, as shown in line 3, contains a number of comma-delimited elements enclosed in parentheses. The individual

list elements are automatically assigned to array elements, starting with element 0. Individual elements can be recalled as shown in line 3, whereas line 5 demonstrates how to recall more than one element by supplying an index list ([1, 2]). The list to be assigned to an array can be provided explicitly. Terminal 161 demonstrated how the output of a shell command can be assigned to an array. Here, the output is handled as if it was a list.

14.3.2.2 Adding and Removing Elements

Once an array has been assigned, you might want to add or remove array elements. There are some very comfortable commands available for these tasks. In the following list, we assume that the array is named *array*.

shift (@array)

Remove First Element. Removes and returns the first element of *array*.

unshift (@array, list)

Add To Beginning. Adds the list before the first position of *array*. Returns the length of the resulting array. The list could just as well be a single string.

pop (@array)

Remove Last Element. Removes and returns the last element of *array*.

push (@array, list) *Add To End.* Adds the list behind the last position of *array*. Returns the length of the resulting array. The list could just as well be a single string.

@array=()

Delete Array. Removes all array elements by assigning an empty list to it.

delete @array[a,b,...]

Delete Elements. Deletes the specified elements of *array*. Returns a list of the deleted elements.

In Terminal 163 you see some simple examples of some of the functions we have just learned.

```

1  $ perl -e '@a=(1,2,3); print pop @a,"\t",@a,"\n"'
2  3      12
3  $ perl -e '@a=(1,2,3); print shift @a,"\t",@a,"\n"'
4  1      23
5  $ perl -e '@a=(1,2,3); unshift @a,(5,6); print "@a\n"'
6  5 6 1 2 3
7  $ perl -e '@a=(1,2,3); push @a,(5,6); print "@a\n"'
8  1 2 3 5 6
9  $ perl -e '@a=(1,2,3);
10 > print "Del: ",delete @a[1,2],"\tRest: @a\n"'
11 Del: 23 Rest: 1
12 $

```

I suppose the examples speak for themselves. In line 9 of Terminal 163 we apply a multiple-line command. Note that you need to finish each line (except the last one) with the semicolon character.

14.3.2.3 Reading Arrays

Perl provides a very convenient command in order to read out all elements of an array.

```

1  $ perl -e '@a=(one,two,three,four);
2  > foreach $e (@a){ print $e}'
3  onetwothreefour
4  $ perl -e '@a=(one,two,three,four);
5  > foreach $e (@a){ print "$e\n"}'
6  one
7  two
8  three
9  four
10 $

```

In the example shown in Terminal 164 we assign a list to the array *a* and use the function *foreach* to read out all array elements. *foreach* creates a loop which will be cycled until the last element of the array *a* is reached. The array elements are saved in the variable *e*.

14.3.2.4 Selecting Elements

Of course, you can use the *foreach* function in order, for example, to search for certain elements or modify individual elements that fit a search pattern; but Perl has shortcuts for this.

grep (condition, @array)

The function *grep* is named after the Unix utility of the same name, which performs searches on text files (see Sect. 7.1.9 on p. 83). Similarly, the Perl function *grep* searches through an array (or a scalar) and creates a new array containing only elements that satisfy the *condition* statement.

```

1  $ perl -e '@a=(a,1,A,aa,ab,10,5,11);
2  > @g=grep(/^.$/, @a);
3  > print "@a\n@g\n"'
4  a 1 A aa ab 10 5 11
5  a 1 A 5
6  $ perl -e '@a=(a,1,A,aa,ab,10,5,11);
7  > @g=grep(/[0-9]{1,}/, @a);
8  > print "@a\n@g\n"'
9  a 1 A aa ab 10 5 11
10 1 10 5 11
11 $

```

In Terminal 165, we create an array named *a* and use the `grep` function in line 2 to select all array elements that consist of one single character. Therefore, we apply the regular expression “`^.$`”. In line 7 all array elements which have numbers as values are picked. In both examples the original array, followed by the selection, are printed.

map (function, @array)

With the `map` function each element of an array can be modified.

```

1  $ perl -e '@a=(a,1,A,aa,ab,10,5,11);
2  > @g=map( ">".$_."<" , @a);
3  > print "@a\n@g\n"
4  a 1 A aa ab 10 5 11
5  >a< >1< >A< >aa< >ab< >10< >5< >11<
6  $

```

Terminal 166 illustrates the function of the `map` function. Again, we make use of the special variable “`$_`”. The dots in line 2 are Perl’s operators for string concatenation.

14.3.2.5 Editing Element Order

There are two important commands that affect the order of the array elements.

reverse @array

Reverse Element Order. Reverses the order of the array elements. The original array remains untouched. The `reverse` function returns the new array.

```

1  $ perl -e '@a=(one,two,three,four);
2  > @r=reverse @a; print @r,"\n"
3  fourthreetwoone
4  $ perl -e '@a=(one,two,three,four);
5  > @r=reverse @a; print "@r\n"
6  four three two one
7  $

```

Terminal 167 demonstrates the `reverse` function. Excursion: note the effect of different quotation settings in the `print` functions in lines 2 and 5. In Sect. 14.6.1 on p. 274, we will focus on the `print` function.

sort {condition} @array

Sort Elements. The `sort` function returns a copy of the array sorted alphabetically. The original array remains unchanged. The condition, which is optional, can be set to “`$a<=>$b`” in order to sort the array elements numerically.

```

1  $ perl -e '@a=(a,1,A,aa,ab,10,5,11);
2  > @alpha=sort @a;
3  > @num=sort {$a<=>$b} @a;
4  > print "@alpha\n@num\n"
5  1 10 11 5 A a aa ab
6  a A aa ab 1 5 10 11
7  $

```

The example in Terminal 168 illustrates the application of the `sort` function. Be aware that `sort` sorts the array elements by default alphabetically.

14.3.2.6 Array Information

The following list gives you an overview of some functions in order to obtain information about an array. In the list we assume that the array is named *array*.

<code>scalar @array</code>	Returns the total number of elements of <i>array</i> .
<code>\$#array</code>	Returns the index of the last array element of <i>array</i> .
<code>\$array[-1]</code>	Returns the last array element's value.

Strangely enough, even when you delete an array element or assign an empty string to it, the element does still exist:

```

1  $ perl -e '@a=(a,1,A,aa,ab,10,5,11);
2  > print scalar @a,"\n";
3  > delete $a[3];
4  > print scalar @a,"\n";
5  > foreach (@a){$i++;}
6  > print $i,"\n";
7  > $a[4]="";
8  > print "@a\n"
9  8
10 8
11 8
12 a 1 A 10 5 11
13 $

```

The Perl script in Terminal 169 makes use of the `scalar` function in order to demonstrate the behavior of an array with deleted elements: the number of array elements does not change!

14.3.3 Hashes

A hash is an unordered collection of “*index* => *value*” pairs. Hashes are also known as associative arrays, since they define associations between indices and values. While Perl’s arrays can only digest numeric indices, these can be either numeric or strings in hashes. The indices are sometimes referred to as *keys*. You might ask: why does Perl discriminate between arrays and hashes, while AWK does not (see Sect. 13.5.4 on p. 215)? Well, hashes have higher memory demands. Thus, Perl gives you the opportunity to be economic with your RAM (random access memory). There is one other important thing you should know about hashes: they can be used as databases. Okay, pretty easy databases, but anyway. A hash can be exported into

a file and is then available for other Perl programs. We will talk in more detail about this feature later (see Sect. 14.7 on p. 276).

The name of hashes is preceded with the percent character (%) when they are assigned and with the dollar character (\$) when a single element is recalled.

```

1  $ perl -e '%h=(A,Adenine,C,Cytosine);print "%h{C}\n"'
2  Cytosine
3  $

```

Terminal 170 gives a short example of a hash variable. Here, the hash is created by assigning a list to it. The list is read in duplets. Thus, the first list element is used as the first index, which points to the second list element, and so forth. However, there are different methods of how to create hashes and assign values to them.

14.3.3.1 Creating Hashes

In principle, hashes can be created in two ways: either by assigning a list or array to the hash (see Terminal 170) or by assigning values to single hash elements.

```

1  $ perl -e '$code{C}="Cys"; print "$code{C}\n"'
2  Cys
3  $ perl -e '%code=(C,Cys,A,Arg); print "$code{C}\n"'
4  Cys
5  $ perl -e '%code=(C=>Cys,A=>Arg); print "$code{C}\n"'
6  Cys
7  $ perl -e '%code=(C=>Cys,A=>Arg); print "@code{C,A}\n"'
8  Cys Arg
9  $

```

In Terminal 171 you find all three possible ways to create a hash variable (here named *code*). In line 1 a single hash element is created by direct assignment. In lines 3 and 5 two differently formatted lists are assigned to the hash variable *code*. The last method (line 5) is probably the most readable one. Note that, in contrast to arrays, hash elements are enclosed in braces and not in brackets. Although not employed in the given examples, it is often wise to enclose text characters in single quotes, like

```
%code = ('C' => 'Cys')
```

From this example you see that Perl is pretty much forgiving for different formats, also with respect to space characters.

Line 7 of Terminal 171 illustrates how to recall more than one hash element at once. It is important to precede the hash variable name with the vortex character (@), because you recall a list.

Hashes are not interpreted when enclosed in double quotes. This affects the action of the `print` command.

```

Terminal 172: Hash Variable in Double Quotes
1  $ perl -e '@nuc=(a,c,g,t);
2  > %hash=(C,Cys,A,Arg);
3  > print "@nuc\n";
4  > print @nuc,"\n";
5  > print "%hash\n";
6  > print %hash,"\n"
7  a c g t
8  acgt
9  %hash
10 AArgCCys
11 $

```

As you can see in the example in Terminal 172, arrays are interpreted both within double quotes and outside. If an array is double-quoted, the elements are separated by spaces in the output. Hashes are interpreted only outside of double quotes. Thus, the output does not look that nice. However, you can assign the space character to the output field separator variable “\$,”. Just add the line

```
$,=" ";
```

before the first `print` command in Terminal 172. Try it out. \$, is a built-in variable. More of these are described in Sect. 14.3.4 on p. 266.

14.3.3.2 Deleting Values

You just saw how to create hashes and add values to them. How can you delete an element?

```

Terminal 173: Deleting Hash Element
1  $ perl -e '%hash=(C,Cys,A,Arg);
2  > print %hash,"\n";
3  > delete $hash{C};
4  > print %hash,"\n"
5  AArgCCys
6  AArg
7  $

```

As with arrays, the `delete` function removes an element, here from the hash. This is illustrated in Terminal 173.

14.3.3.3 Hash Information

Of course, you can also obtain some information about hashes. The first thing you might want to know is the size of a hash variable. For this purpose you can use the `keys` function.

```

Terminal 174: Determining Size of Hash
1  $ perl -e '%hash='ls -l';
2  > print eval(keys %hash),"\n"
3  7      # value depends on the number of files in your directory
4  $

```

The example in Terminal 174 uses the `keys` function to evaluate the size of the hash variable called *hash*. In line 1 we use the system call `ls -l` in order to fill the hash (although the resulting hash variable does not make too much sense). We then call the `keys` function embedded in the `eval` (evaluate) function. If we omitted `eval`, “`keys %hash`” would return all indices (keys). This is shown in the next example.

```

Terminal 175: Hash Keys and Values
1  $ perl -e '%hash=(C,Cys,A,Arg);
2  > print keys %hash,"\n";
3  > print values %hash,"\n" '
4  AC
5  ArgCys
6  $

```

In Terminal 175 we apply the functions `keys` and `values`. In a text context both return a list with all indices and all values, respectively. Remember that the indices in hashes are also called keys. Of course, you could also save the lists in an array variable instead of printing to standard output.

The `keys` function is very useful in order to print out all elements of a hash variable. The `foreach` construct in the following example resembles the one we used for arrays (see Terminal 164 on p. 260).

```

Terminal 176: Looping Through Hash Elements with foreach
1  $ perl -e '%hash=(C,Cys,A,Arg); $\'="-";
2  > foreach (keys %hash){print $hash{$_}}
3  > $\'="-"; print "\n" '
4  Arg-Cys-
5  $

```

Note that we do not specify any output variable in the `foreach` construct in Terminal 176. In Terminal 164 on p. 260 we used the variable `$e`. If we do not specify any variable, then the default input/output variable `$_` is used. In the example in Terminal 176 we also make use of the output record separator variable `$\'`. By default, this is set to nothing. This is why we always use the newline escape sequence (`\n`) to generate a line break.

14.3.3.4 Reverse a Hash

In Sect. 14.3.2 on p. 258 we saw how the function `reverse` can be applied to reverse the order of an array. If applied to hashes, `reverse` returns a list of pairs in which the indices and values are reversed. Thus, the indices are now values and the values are now indices.

```

Terminal 177: Reversing Hash Elements
1  $ perl -e '%hash=(C,Cys,A,Arg);
2  > print %hash,"\n";
3  > %hash=reverse %hash;
4  > print %hash,"\n" '
5  AArgCCys
6  ArgACysC
7  $

```

The examples in Terminal 177 illustrate the effect of the `reverse` function on a hash. The `reverse` function only works correctly with hashes if all the values are unique. This is because a hash requires its keys to be unique.

14.3.4 Built-in Variables

We have already come across some built-in variables that Perl provides for different purposes. In fact, there are over 50 built-in variables. Only some of them are listed below.

<code>\$_</code>	<i>Default Variable.</i> The default input, output, and pattern-searching space variable. This is the default variable for about everything.
<code>\$/</code>	<i>Input Record Separator.</i> The default value is the newline character.
<code>\$\</code>	<i>Output Record Separator.</i> Formats the output of the <code>print</code> command. The default value is an empty string, i.e., no record separator.
<code>\$,</code>	<i>Output Field Separator.</i> Formats the output of the <code>print</code> command. The default value is an empty string, i.e., no field separator.
<code>\$"</code>	<i>List Separator.</i> Formats the output of the <code>print</code> command, if an array is enclosed in double quotes. The elements are separated by the value of <code>\$"</code> . The default value is the space character.
<code>\$.</code>	<i>Input Line Number.</i> The current input line of the last filehandle that was read from. Reset only when the filehandle is closed explicitly.
<code>@ARGV</code>	<i>Command Line Parameters.</i> Contains a list of the command line parameters.
<code>%ENV</code>	<i>Environment.</i> Contains the shell variables. The index is the name of an environment variable, whereas the value is the current setting.

Take a look at Perl's manual pages for more variables. Those listed above are the most commonly used ones. Most of the other variables are useful for the experienced programmer doing fancy things but not for us scientists doing basic data analysis and data formatting.

14.4 Decisions: Flow Control

I will not go through the sense of program flow controls any more. We discussed these intensively in Sect. 10.7 on p. 138 and Sect. 13.7 on p. 220. However, you must know the syntax of Perl's control structures. Furthermore, you will see that Perl provides some additional handy flow control commands.

14.4.1 *if...elsif...else*

The syntax of the `if` structure is the same as in AWK (see Sect. 13.7.1 on p. 220), except the additional `elsif` statement. The `if` construct either executes a command if a condition is true or skips the command(s). With the help of `elsif` and `else` several cases can be distinguished.

```
if (condition-1) {
    commands-1}
elsif (condition-2) {
    command(s)-2;}
else {
    command(s)-3;}
```

There are no semicolon characters behind the braces.

14.4.2 *unless, die and warn*

The `unless` selection structure is the opposite of the `if` selection structure. `unless` either performs an action if a condition is false or skips the command(s) if the condition is true.

```
unless (condition) {
    command(s);}
```

Again, there are no semicolon characters behind the braces.

`unless` is often used in combination with the commands `die` or `warn`. `die` allows you to stop program execution and display a message, whereas `warn` displays a message without terminating program execution.

```

_____ Terminal 178: Functions die and warn _____
1  $ perl -e 'unless(1==2){die "Bye bye\n"}
2  > print "Wow: 1 is equal to 2\n";'
3  Bye bye
4  $ perl -e 'unless(1==2){warn "Bye bye\n"}
5  > print "Wow: 1 is equal to 2\n";'
6  Bye bye
7  Wow: 1 is equal to 2
8  $
```

Examples of `die` and `warn` are shown in Terminal 178. In contrast to the nonsense conditions in lines 1 and 4, one would rather test for the existence of a file or something similar.

14.4.3 while...

The `while` construct performs an action while a condition is true (see Sect. [13.7.2](#) on p. 221).

```
while (condition) {  
    command(s);}
```

The `while` command first checks whether the condition is true or false and then executes the `command(s)`. There are still no semicolon characters behind the braces.

14.4.4 do...while...

Strongly related to `while`, the `do...while` construct first executes `command(s)` and then checks for the state of the condition (see Sect. [13.7.3](#) on p. 222).

```
do {  
    command(s);}  
while (condition)
```

There are, as usual, no semicolon characters behind the braces.

14.4.5 until...

The `until` repetition structure is simply the opposite of `while`. The body of `until` repeats while its condition is false, i.e., until its condition becomes true.

```
until (condition) {  
    command(s);}
```

Note that there are no semicolon characters behind the braces.

14.4.6 do...until...

As with `do...while`, the `do...until` statement differs from the `until` statement by the time when the condition is tested with respect to command execution.

```
do {  
    command(s);}  
until (condition)
```

First, the `command(s)` are executed, then the state of the condition is tested. Funny, but there are no semicolon characters behind the braces.

14.4.7 *for...*

The application of Perl's `for` construct exactly resembles the one in AWK (see Sect. 13.7.4 on p. 223).

```
for (initialization; condition; counter){
    command(s);}
```

Hm, are there no semicolon characters behind the braces? No!

14.4.8 *foreach...*

The `foreach` construct allows you to iterate over a list of values, which is either provided directly or via an array.

```
foreach (list or array){
    command(s);}
```

There are never, ever semicolon characters behind the braces. The nice thing with explicitly given lists is that you can state ranges like “1 . . 5” or “A . . D”, representing the lists “1, 2, 3, 4, 5” and “A, B, C, D”, respectively.

```
_____ Terminal 179: Looping with foreach again _____
1  $ perl -e 'foreach $no (1..4){
2  > print $no} print "\n"'
3  1234
4  $ perl -e 'foreach (A..D){
5  > print $_} print "\n"'
6  ABCD
7  $
```

The two little scripts in Terminal 179 illustrate the use of lists. In the first example, in each cycle the values 1 to 4 are assigned to the variable `$no`. In the second example the general purpose variable `$_` is used instead.

Some more examples can be found in Terminal 164 on p. 260 and Terminal 165 on p. 260. Especially the commands `grep` and `map` in Terminal 165 on p. 260 are worth to look at.

14.4.9 *Controlling Loops*

Loops are nice, controlled loops are nicer. The `while`, `until`, `for` and `foreach` statements fall into the class of loop structures. Like AWK (see Sect. 13.7.5 on p. 224), Perl allows you to control the program flow through loops.

next—The `next` statement skips the remaining commands in the body of a loop and initiates the next cycle (iteration) of the loop. In the following program flowchart *loop* stands for either `while`, `until`, `for` or `foreach`.

```

loop (condition){      <-
  command(s);          |
  next;                 >-
  command(s);
}

```

The following example illustrates the application of `next`.

```

Terminal 180: Short Cutting Loop with next
1  $ perl -e '@seq=split " ", $ARGV[0];
2  > foreach $nuc (@seq){
3  >   if ($nuc !~ /[acgtACGT]/){
4  >     print "-"; next;
5  >   }
6  >   print "$nuc";
7  > }
8  > print "\n" ' acgTAhCsxdDVGctacSvdg
9  acgTA-C-----Gctac---g $

```

The array element `ARGV[0]` in line 1 of Terminal 180 contains the command line parameter, here the sequence in line 8. This sequence string is split into an array (see Sect. 14.3.2 on p. 258) and each element is extracted, one after the other, in the `foreach` loop. In line 3 we check whether the character is a nucleotide. If it is not, a dash is printed in line 4 and the next cycle starts, or else, the nucleotide is printed. Note that `[acgtACGT]` is a regular expression, which must be enclosed by slashes.

last—The `last` command causes immediate exit from the loop. In the following program flowchart *loop* stands for either `while`, `until`, `for` or `foreach`.

```

loop (condition){
  command(s);
  last;           >-
  command(s);    |
}                |
                 <-

```

The program execution continues with the first statement after the loop structure.

redo—After execution of the `redo` command the loop returns to the first command in the body of the loop without evaluating the condition. In the following program flowchart *loop* stands for either `while`, `until`, `for` or `foreach`.

```

loop (condition){
  command(s); <-|
  redo;      >-|
  command(s);
}

```

The `redo` command is useful when it is necessary to repeat a particular cycle of a loop.

14.4.9.1 Conditions

In Sect. 13.4 on p. 201 we learned a lot about the comparison of text strings and numbers with certain patterns. These comparisons yield either true or false and are commonly used as conditions for program flow-control structures. The list below is a compilation of the most important ones. If you can answer the question with *yes*, the condition is *true*. Usually, *\$B* is not a variable but a fixed number or string.

<code>\$A =~ /regex/</code>	Does <i>\$A</i> match the regular expression <i>regex</i> ?
<code>\$A !~ /regex/</code>	Does <i>\$A</i> not match the regular expression <i>regex</i> ?
<code>\$A == \$B</code>	Is the numeric value of the <i>\$A</i> equal to <i>\$B</i> ?
<code>\$A eq \$B</code>	Is the text string <i>\$A</i> equal to <i>\$B</i> ?
<code>\$A != \$B</code>	Is the numeric value of the <i>\$A</i> unequal to <i>\$B</i> ?
<code>\$A ne \$B</code>	Is the text string <i>\$A</i> unequal to <i>\$B</i> ?
<code>\$A < \$B</code>	Is the numeric value of the <i>\$A</i> smaller than <i>\$B</i> ? Of course, this works also for “>”, “<=”, and “>=”.
<code>\$A lt \$B</code>	Is the text string <i>\$A</i> less than <i>\$B</i> ? This works also with <i>gt</i> (greater than), <i>le</i> (less than or equal to) and <i>ge</i> (greater than or equal to). See Sect. 13.4.3 on p. 203 for rules of string comparison
<code>\$C1 && \$C2</code>	Are conditions <i>C1</i> and <i>C2</i> true?
<code>\$C1 \$C2</code>	Is condition <i>C1</i> or <i>C2</i> true?
<code>\$C1 ! \$C2</code>	Is the state of condition <i>C1</i> not like the state of condition <i>C2</i> ?

As I said above, there are more operators available. Those described here are the most commonly used ones.

14.5 Data Input

Well, it is nice to have a refrigerator, but it is bad if you cannot put things into it or get things out. In a sense, Perl scripts and refrigerators have something in common! In this section you will see different ways which let your program read and write data.

14.5.1 Command Line

The simplest way to feed a program with data, though not the most comfortable one, is the command line. One way is to provide data in the command line when you call the program. We used this technique in Terminal 180 on p. 270. The command line parameters are accessible via the array *ARGV*. The variable *ARGV[0]* is the first element in the command line and so forth. The elements are separated by space characters.

Another way is to ask the user for some input during the execution of the script.

```

1  $ perl -e 'print "Enter Sequence: "; $DNA=<STDIN>;
2  > print "The sequence was $DNA";
3  > print "Thank you\n"'
4  Enter Sequence: gactagtgc
5  The sequence was gactagtgc
6  Thank you
7  $

```

Terminal 181 shows you an example of how you can obtain user input. The important command is “\$DNA=<STDIN>”. <STDIN> is called a filehandle. You will see more of these soon. The filehandle <STDIN> initiates a request. All characters you have typed are saved in the variable *DNA* after you hit (Enter). Note that the newline character created by pressing (Enter) is saved, too. As we will see later in Sect. 14.9 on p. 279, you can remove it with the command `chomp`.

14.5.2 Internal

Your program might require some default input, like a codon table. This you attach to the end of a script after the “__END__” statement.

```

1  # save as data.pl
2  print <DATA>;
3  __END__
4  Here you can save your data
5  in many lines

```

In Program 48 we use the default filehandle <DATA>. With this filehandle all lines following the “__END__” statement can be recalled.

14.5.3 Files

Apart from the default filehandles <STDIN> and <DATA> you can define your own and get input from files. This is what you most probably want to do.

```

1  # save as open-file.pl
2  # input a filename
3  # the content of the file is displayed
4  print "Enter Sequence Filename: ";
5  $file=<STDIN>;
6  unless (open (SEQFILE, $file)){
7  die "File does not exist\n"}
8  @content=<SEQFILE>;
9  close SEQFILE;
10 print @content;

```

Program 49 shows you how to read data from a file. This program is slightly interactive. Line 4 prints a text asking for a filename. Line 5 stops program execution and requires user input. After you have entered the filename, press **(Enter)**. With the `open` command the filename saved in the variable `file` is opened. *SEQFILE* is the filehandle for this file. If you want to refer to the opened file, you have to use its filehandle. It can have any name you like. By convention, filehandles are written in uppercase. We also check with the `unless...die` construct, whether the file exists. In line 8, the whole file content is assigned to the array `content`. Thus, `content[0]` will contain the first line of the file and so forth. Next, we close the file with the command `close`. Again, we use the filehandle to tell the system which file to close. This is obvious here; however, you could have many files open at once. Each file must have its own filehandle. Finally, the value of the array `content`, i.e., the file content, is displayed. In the previous example we assigned the filehandle to an array. In such cases, the whole file is copied to the array, each line being one array element. This is a somewhat special case because usually the input file is read line by line. By default, lines (or better: records) are expected to be separated by the newline character. However, you are free to change the record separator by assigning a value to the input record separator variable `$/`. A typical file-reading procedure is shown in the next example.

```

1  $ perl -e 'open (INPUT, "sequences.txt") or die;
2  > $i=1;
3  > while (<INPUT){
4  > print "Line-$i: $_";
5  > $i++;}'
6  Line-1: >seq11
7  Line-2: accggttggtcc
8  Line-3: >Protein
9  Line-4: CCSTRKSBCJHBJCB AJDHL CBH
10 $

```

In Terminal 182 the file *sequences.txt* is read line by line. The `while` loop iterates as long as there are lines to read. `<INPUT>` becomes false when the end of the file *sequences.txt* has been reached. The active line (record) is available via the special variable `$_`. Note that each line consists of all its characters plus the newline character (line break) at the end. Therefore, we do not use the newline character “`\n`” in the print command in line 4. At the beginning of the program we use the command `or` in conjunction with `open`. This is another way of writing `unless...die`. Programmers like shortcuts!

Sometimes, you might want to read a file as blocks of characters. This can be done with the command `read`.

```

1  $ perl -e 'open (INPUT, "sequences.txt") or die;
2  > $i=1; while (read (INPUT,$frac,10)){
3  > print "Line-$i: $frac\n";
4  > $i++;}'
5  Line-1: >seq11
6  acc
7  Line-2: ggttggtcc
8
9  Line-3: >Protein
10 C

```

```

11 | Line-4: CSTRKSBCJH
12 | Line-5: BHJCBAJDHL
13 | Line-6: CBH
14 |
15 | $

```

The input file we use in Terminal 183 is the same as we used in Terminal 182. However, now we instruct Perl only to read 10 characters from *sequences.txt* with the command

```
read (INPUT, $frac, 10)
```

The `read` command requires the filehandle, a variable name where to save the string and the number of characters to extract. Count the characters in the output by yourself and keep in mind that the invisible newline character counts as well.

14.6 Data Output

As important as data input is data output (think of the refrigerator and a cool beer). Usually, you either print your results onto the screen or save them in a file. We will deal with both cases in this section.

14.6.1 *print, printf and sprintf*

Okay, the `print` command is not really new any more. We have used it over and over in all Terminals in this chapter. Anyway, be aware that the `print` command does not automatically append a newline character. This is one important difference to AWK. However, you can modify the default output field and record separators by changing the values of the variables `$,` and `$\`, respectively. Also remember that, in contrast to variables enclosed in single quotes, those enclosed in double quotes are expanded.

Like AWK, Perl offers you to print formatted text strings either onto the screen (`printf`) or into a variable (`sprintf`). Both commands work exactly as described in Sect. 13.8.1 on p. 225 for AWK. I am sure that you still remember how they worked, don't you?

14.6.2 *Here Documents:<<*

Do you recall *here documents* from shell programming? These allow you to display large blocks of text (refresh your memory in Sect. 10.4.2 on p. 131). You initiate a here document with the `<<` operator. The operator is followed by any arbitrary

string (the identifier)—there must not be any space character between the operator and the identifier. All following text is regarded as coming from the standard input, until the identifier appears a second time. For Perl to recognize the closing identifier, it must be both unquoted and at the beginning of an empty line. No other code may be placed on this line. Usually, variables within the here document are expanded. However, if the identifier is single-quoted, variables are not expanded.

```

_____ Program 50: heredoc.pl - Here Document _____
1  # save as heredoc.pl
2  $text=<<TEXT;
3  This is line one
4      this is line 2
5  TEXT
6  print <<'TEXT';
7  It follows the text
8  just saved in $text
9  TEXT
10 print "$text";

```

Program 50 demonstrates the application of here documents. The output is shown in the following Terminal:

```

_____ Terminal 184: Here Documents _____
1  $ perl heredoc.pl
2  It follows the text
3  just saved in $text
4  This is line one
5      this is line 2
6  $

```

Note that the value of variable *text* in line 7 of Program 50 is not extracted because the identifier “TEXT” in line 6 is single-quoted.

14.6.3 Files

Very often one wishes to save some results directly in a file. This is basically as easy as reading a file.

```

_____ Program 51: save-seq.pl - Save To File _____
1  # save as save-seq.pl
2  # appends a sequence to the file sequences.txt
3  # sequences.txt will be created if it does not exist
4  # sequences are saved in fasta format
5  print "Enter Sequence Name: ";
6  $seqname=<STDIN>;
7  print "Enter Sequence: ";
8  $seq=<STDIN>;
9  open (FILE, ">>sequences.txt");
10 print FILE ">$seqname";
11 print FILE $seq;
12 close FILE;

```

Program 51 saves a sequence together with its name in a file called *sequences.txt*. The sequence name and the sequence itself are provided via command line inputs in

lines 6 and 8. In line 9 we open the file *sequences.txt* in the append mode (>>) and assign the filehandle *FILE* to it. Now, we can use the filehandle in conjunction with the `print` command in order to save data in the file *sequences.txt*. If this file does not exist, it will be created. If it exists, data will be appended to the end. Finally, in line 12, we close the file.

As you could see in the previous example, files can be opened in different modes. The required mode is prefixed to the filename. The following list shows all possible modes with the corresponding prefix.

- < Open a file for reading. This is the default value.
- > Create a file for writing. If the file already exists, discard the current contents.
- >> Open or create a file for appending data at the end.
- +< Open a file for reading and writing.
- +> Create a file for reading and writing. Discard the current contents if the file already exists.
- +>> Open or create a file for reading and writing. Writing is done at the end of the file.

If you do not give any mode (as we did in Terminal 49 on p. 111), the file is opened only for reading.

14.7 Hash Databases

As you have seen in Sect. 14.3.3 on p. 262, hashes are nice tools to associate certain values. This makes you independent of zillions of `if . . . then` constructs. However, storage of data in variables is temporary. All such data are lost when a program terminates. Perl provides the possibility to save hash tables as databases. These are called *DBM* files (DataBase Management). DBM files are very simple databases, though often sufficient.

```

1  $ perl -e '
2  unless (dbmopen(%code, "genetic-code.dbm", 0644)){
3      die "Can not open file in mode 0644\n"
4  }
5  %code=(ATG,M,TCA,S,TTC,F,TGT,C,GAT,D,TGA,"*");
6  dbmclose(%code) '
7  $

```

In Terminal 185 we create a database file with the name *genetic-code.dbm*. In line 2, this file is either opened or, if not existent, created. The command is `dbmopen`, which needs a) the hash name (*code*) preceded by a percentage character, b) the filename (*genetic-code.dbm*) enclosed in double quotes and c) the access mode (*0644*), which is the permission with which the file is to be opened (see Sect. 5.3 on p. 26). In line 5 we assign a list to the hash *code* and then we close the database file. Thereby, the hash is saved in the database format. Note that the last character in the list is a special character (*), which must be enclosed in double quotes.

Now let us see how we can recall the data.

```

1  $ perl -e '
2  unless (dbmopen(%code, "genetic-code.dbm", 0644)){
3      die "Can not open file in mode 0644\n"
4  }
5  print @code{ATG, TGT, TGT, TCA, TGA}, "\n";
6  dbmclose(%code)'
7  MCCS*
8  $

```

The main structure of the program in Terminal 186 is the same as in Terminal 185 on the facing page. However, in line 5 we read out hash elements. In fact, we read several hash elements at once. Note that, in such cases, the name of the hash must be preceded with the vortex character (@).

That is all about Perl's databases. As you have seen, it is very simple and very useful to employ hash tables as databases.

14.8 Regular Expressions

Text processing is one of Perl's most powerful capabilities; and, as you may guess, a lot of this power comes from regular expressions. You got a thorough insight into the usage of regular expressions in Chap. 11 on p. 157. Thus, I will restrict myself here to exemplify how Perl treats regular expressions.

There are some useful built-in variables that store regular expression matches. These are listed in the following table.

\$&	The string that matched.
\$`	The string preceding the matched string.
\$'	The string following the matched string.

Of course, you can also use back referencing as described in detail in Sect. 11.4.7 on p. 168. Thus, \$1 would be the first parenthesized subexpression that matched, \$2 the second and so on.

14.8.1 Special Escape Sequences

In addition to the single-character meta characters we dealt with in Sect. 11.4.4 on p. 166, Perl interprets a number of escape sequences as query patterns. The most important ones are listed below.

\s	Matches space and tabulator characters.
\S	Inverse of \s
\d	Matches numeric characters.
\D	Inverse of \d
\A	Matches the beginning of a text string.
\Z	Matches the end of a text string.

The first four escape sequences may be used inside or outside of character classes (`[...]`).

14.8.2 Matching: `m/.../`

The most simple thing you might want to do is to extract a regular expression match from a string. The command would be

```
$var = ~ m/regex/
```

In this case the value of the variable `var` remains unchanged and the match to `regex` would be saved in the variable `$&`. You can actually omit the command “`m`”. Thus,

```
$var = ~ /regex/
```

would do the same job. If you omit the binding operator (`=~`), the value of the built-in variable `$_` would be used.

The following program gives you an example of the construct:

```

_____ Program 52: match.pl - Regular Expressions: match _____
1  # save as match.pl
2  while (<DATA>){
3    $text=$text.$_}
4    print "$text\n";
5    $text =~ m/cc.>/s;
6    print "Prematch: $'\n";
7    print "Match: $&\n";
8    print "Postmatch: $'";
9  _ _END_ _
10 >seq11
11 accggttggtcc
12 >Protein
13 CCSTRKSB CJHBJCBAJDHLCBH

```

In Program 52 the data are read from the end of the file. Each line is concatenated to the variable `text`. The dot in line 3 is the concatenation command. In line 4 we print out the content of `text`. Then we search for the pattern “`cc.>`”. The modifier `s` tells Perl to embed newline characters; thus, they can be matched with dot characters (`.`). However, the caret (`^`), standing for the beginning of a line, and the dollar character (`$`), standing for the end of a line, no longer work in this mode. The following list gives an overview of the most important modifiers:

- `i` *Ignore*. The query is case-insensitive.
- `s` *Single Line*. Treat the string as a single line with embedded newline characters.
- `m` *Multiple Line*. Treat the string as multiple lines with newline characters at the end of each line.

As with the match command `m`, regular expressions are used with the substitution command `s` and the transliteration command `tr`, respectively. These commands are discussed in Sects. 14.9.1 and 14.9.2, respectively.

14.9 String Manipulations

As stated above, Perl is strong in text manipulations. This is what it was developed for. Let us first see how we can substitute and transliterate certain text patterns, before we go through the most important string manipulation commands.

14.9.1 Substitute: `s/.../.../`

I guess you are pretty much accustomed to the substitution command because we have used it already a number of times with Sed and AWK. The syntax is

```
$var = ~ s/start/ATG/g
```

This would substitute all (global modifier: `g`) occurrences of “start” with “ATG” in the variable called `var`. The value of `var` is replaced by the result. If you omit the binding operator (`=~`), the value of the built-in variable `$_` would be changed.

14.9.2 Transliterate: `tr/.../.../`

The transliteration “`tr/.../.../`” resembles the “`y/.../.../`” command we know from Sed (see Sect. 12.5.2 on p. 184). In fact, you can even use “`y/.../.../`”—it has the same function as “`tr/.../.../`”. The syntax is

```
$var = ~ tr/acgt/TCGA/
```

If the value of the variable `var` was a DNA sequence, it would be complemented and converted into uppercase. Thus, “atgcgt” would become “TACGCA”. If you omit the binding operator (`=~`), the value of the built-in variable `$_` would be changed. The transliteration comes with two very useful modifiers.

- `d` Deletes characters that have no corresponding replacement.
- `s` Squashes duplicate consecutive matches into one replacement.

For example, the following command would replace all *a*'s with *A*'s, all *b*'s with *B*'s, and delete all *c*'s and *d*'s:

```
tr/abcd/AB/d
```

Without the modifier *d*, all *c*'s and *d*'s would be replaced by the last replacement character, here the “*B*”.

An example of the application of the *s* modifier is:

```
tr/A-Z/A-Z/s
```

This would convert the word “PRRROTTEIINN” into “PROTEIN”.

14.9.3 Common Commands

In this section you will learn how to apply the most frequent text string manipulation commands. Most functions are homologous to AWK. Thus, take a look at Sect. 13.8.3 on p. 229 for examples.

chomp target

Remove Input Record Separator. This command is usually used to remove the newline character at the end of a string. In fact, *chomp* removes the value of the input record separator variable *\$/* (which is by default the newline character) from the end of a string. The variable can be either a scalar or an array (then all elements are reached). The variable is modified and the number of changes is returned.

```

1  # save as chomp.pl
2  while (<DATA>){
3    chomp $_;
4    print $_}
5    print "\n";
6  __END__
7  Here you can save your data
8  in many lines

```

Program 53: *chomp.pl* - *chomp*

Try out Program 53 in order to understand the function of *chomp*.

chop target

Remove Last Character. This function chops off the last character of a scalar or the last element of an array. The array is deleted by this action! *chop*, modifies the variable, and returns the chopped character or element.

lc target

Lowercase. Returns a lowercase version of *target*, which remains untouched.

uc target

Uppercase. Returns an uppercase version of *target*, which remains untouched.

length target

Length. Returns the length of the string *target*.

substr(target, start, length, substitute)

Extract Substring. Returns a substring of *target*. The substring starts at position *start* and is of length *length*. If *length* is omitted, the rest of *target* is used. If *start* is negative, *substr* counts from the end of *target*. If *length* is negative, *substr* leaves that many characters off the end of *target*. If *substitute* is specified, the matched substring will be replaced with *substitute*.

index(target, find, start)

Search. This function searches within the string *target* for the occurrence of the string *find*. The index (position from the left, given in characters) of the string *find* in the string *target* is returned (the first character in *target* is number 1). If *find* is not present, the return value is -1 . If *start* is specified, *index* starts searching after the offset of “*start*” characters.

rindex(target, find, start)

Reverse Search. Like *index* but returns the position of the last substring (*find*) in *target* at or before the offset *start*. Returns -1 , if *find* is not found.

With the string manipulation commands presented in this section you can do basically everything. Remember that you first have to formulate clearly what your problem is. Then start to look for the right commands and play around until everything works. Very often, small errors in the use of regular expressions cause problems. You must be patient and try them out thoroughly.

14.10 Calculations

Needless to say, Perl offers some built-in arithmetic functions to play around with. The most important ones are listed in alphabetical order as follows:

<code>abs x</code>	Returns the absolute value of x .
<code>cos x</code>	Returns the cosine of x in radians.
<code>exp x</code>	Returns e to the power of x .
<code>int x</code>	Returns the integer part of x .
<code>log x</code>	Returns the natural logarithm of x .
<code>rand x</code>	Returns a random fractal number between 0 (inclusive) and the value of x . The random generator can be initiated with <code>srand y</code> , where y is any number.
<code>sin x</code>	Returns the sine of x in radians.
<code>sqrt x</code>	Returns the square root of x .
<code>x ** y</code>	Raises x to the power of y .

You need the common logarithm (base 10) of x ? How about your math? Some years ago? Okay, do not worry, I had to look it up, too... The logarithm of x to base b is

$\ln(x)/\ln(b)$ where $\ln$ is the logarithm to base e . An appropriate command would be

```
$x = log($x)/log(10)
```

It would be nice to have a function for this. This we are going to learn next.

14.11 Subroutines

Imagine your programs need to calculate at many places the common logarithm of something. Instead of writing `$x = log($x)/log(10)` all the time (see Sect. 14.10), we would be better off writing our own function. We already came across user-defined functions in AWK (see Sect. 13.8.5 on p. 234). The concept of user-defined functions is the same in Perl; however, they are called *subroutines* and the arguments are passed differently to the subroutine. Thus, the general syntax becomes:

```
sub name{
    my($par1,$par2,...)=@_;
    command(s);
    return $whatever
}
```

You would call this subroutine with

```
name(par1,par2...)
```

The parameters, sometimes called arguments, are passed to the subroutine via the built-in array “@_”. With

```
my($par1,$par2,...) = @_
```

the parameters are collected and saved in the variables *par1*, *par2* and so on. The command `my` restricts the usage of these variables to the subroutine. You should declare all variables used in subroutines with “`my $var`” before you use them. You can also declare them while assigning them, like

```
my $var = 1.5
```

Once a variable is declared in this fashion, it exists only until the end of the subroutine. If any variable elsewhere in the program has the same name, you do not have to worry. Okay, now back to practice. In the following example we create a subroutine that calculates the logarithm of x to the base b .

```

1  # save as sublog.pl
2  # demonstrates subroutines
3  # calculates logarithm
4  print "Calculator for LOG of X to base B\n";
5  print "Enter X: "; $x=<STDIN>;
6  print "Enter B: "; $b=<STDIN>;
7  printf("%s%.4f\n", "The result is ", callog($x,$b));
8
9  sub callog{
10     my($val,$base)=@_;
11     return (log($val)/log($base));
12 }

```

The effect of the first six lines of Program 54 you should know. New is the call of the subroutine *callog* at the end of line 7. Two parameters, the variables *x* and *b*, respectively, are passed to the subroutine *callog*. The subroutine itself spans from line 9 to line 12. In line 10, the parameters are collected and saved in the variables *val* and *base*, respectively. These variables are declared with *my* and thus are valid only within the subroutine. The return value of the subroutine is the result of the calculation in line 11. Ultimately, “*callog* (*\$x*, *\$b*)” is substituted by this return value. If necessary, you could also return a list of variables, i.e., an array. The result is formatted such that it is displayed with four decimal digits (“%.4f”; recall Sect. 13.8.1 on p. 225 for details). Terminal 187 shows the typical output of Program 54.

```

1  $ perl sublog.pl
2  Calculator for LOG of X to base B
3  Enter X: 100
4  Enter B: 10
5  The result is 2.0000
6  $

```

If you need to, you can design your program to call subroutines from within subroutines. Sometimes, subroutines do not require any parameters; then you call them with a set of empty parentheses, like *name* (). You can place your subroutines wherever you want in your program. However, I advise you to write them either at the beginning or at the end. This is handier, especially with large programs.

14.12 Packages and Modules

After some time of working with Perl, you will see that there are some operations you do again and again. This could be the conversion of RNA into DNA, or the translation of RNA into a protein sequence, or calculating the common logarithm. One way to avoid writing the same program code again and again is copy-pasting, another is to include it in *packages*. A package should be saved with the file extension “.pm”. The package itself starts with the line

```
package Name;
```

where *Name* is the package name. Package names must always begin with an upper-case character and should match the package filename! Let us stay with the logarithm problem we worked on in Sect. 14.11 on p. 282. First, we create a package file.

```

1  # save as FreddysPackage.pm
2  # this is a package file
3  package FreddysPackage;
4
5  # calculates logarithm
6  sub callog{
7      my($val,$base)=@_;
8      return (log($val)/log($base));
9  }
10
11 # indicate successful import
12 return 1;

```

Program 55, which is a package file, contains only two special features. Line 3 tells Perl that this is a package and line 12 tells the program which calls the package that everything worked fine. These two lines are necessary to define a package. Now let us use *FreddysPackage.pm*.

```

1  # save as require.pl
2  require FreddysPackage;
3  print "Calculator for LOG of X to base B\n";
4  print "Enter X: "; $x=<STDIN>;
5  print "Enter B: "; $b=<STDIN>;
6  printf("%s%.4f\n", "The result is ",
7      FreddysPackage::callog($x,$b));

```

In line 2 of Program 56 we tell Perl that we require *FreddysPackage*. Note that the file extension is omitted here. In line 7 we call the function *callog* from the package *FreddysPackage* and provide the parameters *x* and *b*. The package name and the function name are separated by the double colon operators (*:* *:*). That's it! After a while, it becomes boring to write the package name every time. You would rather call the subroutine *callog* from the package *FreddysPackage* as if it was a built-in function. In such cases, the package is called a module. Okay, let us go for it.

```

1  # save as FreddysPackage.pm
2  # this is a package file
3  package FreddysPackage;
4
5  use Exporter;
6  our @ISA=qw(Exporter);
7
8  # export subroutine
9  our @EXPORT=qw(&callog);
10
11 # calculates logarithm
12 sub callog{
13     my($val,$base)=@_;
14     return (log($val)/log($base));
15 }
16
17 # indicate successful import
18 return 1;

```

The package in Program 57 has more lines than Program 55. Actually, I will not go into all details here. Whenever you want to make your subroutines in a package file easily accessible, add lines 5, 6, and 9. In line 9 you specify the subroutines you want to export within the parentheses, preceded with the ampersand character (&). If there are more subroutines, you have to separate them by spaces, like

```
our @EXPORT=qw(&sub1 &sub2 &sub3);
```

While the package has become more complicated, the program becomes simpler.

Program 58: *use.pl* - Using Modules

```

1  # save as use.pl
2  use FreddysPackage;
3  print "Calculator for LOG of X to base B\n";
4  print "Enter X: "; $x=<STDIN>;
5  print "Enter B: "; $b=<STDIN>;
6  printf("%s%.4f\n", "The result is ",
7  callog($x,$b));
```

Instead of `require`, the command `use` is applied. In line 7 of Program 58 we call the subroutine `callog` from package *FreddysPackage* as if it was a built-in function.

With the knowledge of this section you should be able to construct your own package with the functions (modules) you frequently use. One great resource for such modules is provided by *CPAN* (Comprehensive Perl Archive Network) at <http://www.cpan.org>. Maybe the day will come when you place your own packages at this site.

14.13 Bioperl

Bioperl is not a new programming language but a language extension (Stajich et al. 2002). Officially initiated in 1995, *The Bioperl Project* is an international association of developers of open-source Perl tools for bioinformatics, genomics, and life-science research. Bioperl is a collection of perl modules that facilitates the development of Perl scripts for bioinformatics applications. At the time of writing the actual bioperl version was 1.6.9 (February 2012). The newest version can be downloaded at <http://www.bioperl.org>. Bioperl does not include ready-to-use programs but provides reusable modules (see Sect. 14.12 on p. 283) for sequence manipulation, accessing of databases using a range of data formats and execution and parsing of the results of various molecular biology programs including BLAST, ClustalW (see Chap. 9 on p. 115).

Can you make use of Bioperl? In principle, yes; however, in order to take advantage of bioperl, you need, apart from a basic understanding of the Perl programming language (which you should have by now), an understanding of how to use Perl references, modules, objects, and methods. We briefly scratched at this in Sect. 14.12 on p. 283. However, it is beyond the scope of this book to cover these topics in detail. Thus, take one of the books I recommended in the beginning, work through the

chapters on references, modules, objects, and methods and have fun with bioperl. A still incomplete online course can be found at <http://www.pasteur.fr/recherche/unites/sis/formation/bioperl>.

14.14 You Want More?

This chapter has introduced basic Perl commands and tools. However, Perl is much more powerful. It is not my intention to tell you everything—well, I do not even know everything—about programming with Perl. With what you have learned here, you have the basic tools to solve almost all data formatting and analysis problems. If you think back to the first chapters of the book: you even had to learn how to login to and logout from the computer. Now you can even write your own programs. If you are hungry for more, I recommend you to buy a thick nice-looking book on Perl and go ahead. There are no limits!

14.15 Examples

Let us finish this chapter with some examples.

14.15.1 Reverse Complement DNA

The following program translates a DNA sequence into an RNA sequence and computes the reverse complement of the latter.

```

_____ Program 59: dna-rna.pl - Transform DNA _____
1  #!/usr/bin/perl -w
2  # save as dna-rna.pl
3  # playing around with sequences
4
5  print "Enter Sequence: ";
6  $DNA=<STDIN>; # ask for a seq
7  chomp $DNA; # remove end of line character
8  $DNA=uc $DNA; # make uppercase
9  print "DNA Sequence:\n$DNA\n";
10
11 $RNA = $DNA;
12 $RNA=~s/T/U/g;
13 print "RNA Sequence:\n$RNA\n";
14
15 $RevCompl=reverse $RNA;
16 $RevCompl=~tr/ACUG/UGAC/;
17 print "Reverse Complement:\n$RevCompl\n";

```

If not made executable, Program 59 is executed with

```
perl dna-rna.pl
```


In the first line we see that the option `-w` is activated. This means that Perl prints warnings about possible spelling errors and other error-prone constructs in the script. In line 5, a message is displayed that informs the user that some input is required. The DNA sequence coming from standard input (`<STDIN>`) is assigned to the variable *DNA*. In line 7, the newline character is erased with the command `chomp`. Then the sequence is made uppercase (`uc`) and displayed. In line 11, the sequence saved in *DNA* is assigned to the variable *RNA*. Then, in line 12, all *T*s are substituted by *U*s. The result is printed in line 13. In line 15, the value of the variable *RNA* is reversed and assigned to the variable *RevCompl*. Next, in line 16, the reversed sequence is complemented by the transliterate function `"tr/.../.../"`. This is quite a nice trick to complement nucleotide sequences. Again, the result is displayed.

14.15.2 Calculate GC Content

It is quite a common problem to estimate the portion of guanine and cytosine nucleotides of a DNA sequence. The following program shows you one way in which to accomplish this task:

```

1  #!/usr/bin/perl
2  # save as gccontent.pl
3  $seq = $ARGV[0];
4  print "$seq\n";
5  $gc = gc_content($seq);
6  printf("%s%.2f%s\n", "GC-Content: ", $gc, "%");
7
8  sub gc_content{
9      my $seq = shift;
10     return ($seq =~ (tr/gGcC//)/length($seq)*100);
11 }

```

Program 60 takes its input from the command line. The DNA sequence from the command line is assigned to the variable *seq* in line 3. After the sequence has been printed, the variable *seq* in line 5 is forwarded to the subroutine `gc-content`, which starts at line 8. The command `shift` in line 9 collects the first element from the array `"@"` and assigns it to the local (my) variable *seq*. Then, in line 10, a very smart action is performed. Do you recognize what is going on?—The transliteration function is applied to erase all upper- and lowercase *G*s and *C*s. The number of replacements is divided by the length of the complete sequences and multiplied by 100, thus delivering the portion of guanines and cytosines. This line is a very good example of the versatility of Perl. `"tr/gGcC//"` is immediately replaced by the modified string. Furthermore, since it is situated in a scalar, i.e., arithmetic, context, `"tr/gGcC//"` is replaced by its size. The result of the whole calculation is then assigned to *seq*. The result is returned and saved in *gc* in line 5. Finally, the result is printed out with two digits after the decimal point (`%.2f`).

14.15.3 Restriction Enzyme Digestion

Restriction enzymes recognize and cut DNA. The discovery of restriction enzymes laid the basis for the rise of molecular biology. Without the accurate action of restriction enzymes (or, more precisely endonucleases) gene cloning would not be possible. Those restriction enzymes that are widely used in genetic engineering recognize palindromic nucleotide sequences and cut the DNA within these sequences. Palindromes are strings that can be read from both sides, resulting in the same message. Examples are: “Madam I’m Adam” or “Sex at noon taxes”.

With the following program, we wish to identify all restriction enzyme recognition sites of one or more enzymes. We do not care where the enzyme cuts (this we leave for the exercises). Furthermore, we assume that a list with restriction enzyme names and recognition sites is provided. What do we have to think of? Well, the most tricky thing will be the output. How shall we organize it, especially with respect to readability? A good solution would be to have first the original sequence, followed by the highlighted cutting sites. Try out the following program:

```

----- Program 61: digest.pl - DNA Digestion -----
1  #!/bin/perl -w
2  # save as digest.pl
3  # Provide Cutter list
4  # Needs input file "cutterinput.seq" with DNA
5  %ENZYMES=(XmaIII => 'cgccgc', BamHI => 'ggatcc',
6  XhoI => 'ctcgag', MstI => 'tgcgca');
7  $DNA="";
8  open (CUTTER,"cutterinput.seq") or die;
9  while (<CUTTER>){
10     chomp $_; $DNA=$DNA.$_;
11 }
12 close CUTTER;
13 $DNA = lc $DNA;
14 @OUTPUT=cutting($DNA,%ENZYMES);
15 @NAMES=keys %ENZYMES;
16 while ($DNA ne ""){
17     $i=-1;
18     printf("%-35s", substr($DNA,0,30,""),"\n");
19     while (++$i < scalar(@NAMES)){
20         printf("%-35s",
21             substr($OUTPUT[$i],0,30,""),"\t$NAMES[$i]\n");
22     }
23 }
24
25 sub cutting{
26     my ($DNA,%ENZYME)=@_;
27     foreach $ENZ (values %ENZYME){
28         @PARTS=split($ENZ,$DNA);
29         $CUT=""; $i=0;
30         while(++$i <= length($ENZ)){ $CUT=$CUT.'+' }
31         foreach $NUC (@PARTS){
32             $NUC=~ s/./-/g;
33         }
34         $CUT=$CUT; push(@OUT,sprintf("%-40s", "@PARTS","\n"));
35     }
36     chomp @OUT; return @OUT;
37 }

```


Terminal 188 shows a typical output of Program 61. The sequence file, containing a single sequence, must be saved in *cutterinput.seq*. All “+” characters indicate recognition sites of the corresponding restriction enzyme.

Unfortunately, Program 188 has a little bug: if the restriction enzyme recognition site lies at the end of the DNA sequence, the last nucleotide is not recognized. Try it out! Do you have a good idea how to cure the problem?

Exercises

The best exercise is to apply Perl in your daily life. However, to start with, I add some exercises referring to the examples in Sect. 14.15 on p. 286.

14.1. Expand Program 59 on p. 286 such that it translates the original DNA sequence into the corresponding protein sequence. Use the standard genetic code.

14.2. Expand Program 59 on p. 286 such that it translates both the original and the complemented into the corresponding protein sequence. Use the standard genetic code.

14.3. Expand Program 59 on p. 286 such that it can load several sequences in fasta format from a file and converts them.

14.4. We used Program 60 on p. 287 in order to calculate the GC content of DNA. Modify it such that it accepts RNA, too.

14.5. What should be changed in Program 60 on p. 287 in order to make it suitable to accept very large DNA sequences, for example from a pipe?

14.6. Modify Program 61 on p. 288 such that the name of the input file can be given at the command line level.

14.7. Modify Program 61 on p. 288 such that the position in terms of nucleotides is displayed at the beginning and end of each line.

14.8. Modify Program 61 on p. 288 such that the list of restriction enzymes and their recognition sites is imported from a file.

14.9. Unfortunately, Program 61 on p. 288 has a little bug: if the restriction enzyme recognition site lies at the end of the DNA sequence, the last nucleotide is not recognized. Try it out! Find a way to cure the problem.

Chapter 15

Other Programming Languages

Every book needs a short chapter—this is mine. It presents an example of how to proceed with programming.

To put it straight: this book is written with the intention to show you how to deal with large datasets. My intention is not to teach you hardcore programming. Though you might happen to end-up in a situation or position to decide how to develop data processing and visualization in a professional setup. At that point, AWK scripts running in the terminal may not be sufficient or satisfying anymore.

During my professional career, I rolled out a number of small (affecting some people) to large (affecting several dozen people) programs. All large projects were programmed either in **Java** or **PHP** by computer scientists. These guys know how to create stable and scalable software. “Real” programming is going far beyond data processing with one-liners or small scripts. Apart from different programming philosophies, you should remember that Java programs run on the client computer, while PHP programs run on a server. Thus, CPU power has to be delivered either by you or your customer.

However, a frequent situation I saw myself confronted with this is: a pure experimentalist asks me if I can help out in solving a data processing and visualization problem. Yes, this is the common answer, because I love problem solving. And now comes the point: the pure experimentalist happens to like my solution (which might involve a combination of AWK and R scripts bundled in a Bash shell script and involving MySQL database queries) and wants to run it on his or her computer—uhhhh. What I do then is wrapping my solution in **PHP** and **HTML** (the latter being the HyperText Markup Language, which web browsers can read). So I create a web page with a data input form and a Process button. The result is then visualized in the browser, or is provided as a downloadable PDF (an example is shown in Fig. 15.1).

Okay, I neither teach you PHP and HTML in this book, nor do I show you how to administrate an Apache server, which ultimately provides the web page. Though if you worked through this book you are well prepared to do so. The keyword for

Part V
Advanced Data Analysis

Chapter 16

Relational Databases with MySQL

This chapter will introduce you to the heart of data management, i.e., databases. In particular, it will show you how to bring your own data into a relational MySQL database. “Relational” means essentially that your data are organized into tables that are related to each other. The database engine we are going to use is the freely available, open source MySQL. It is available for all operating systems. There are many other database engines like Microsoft’s Access, Oracle, PostgreSQL, or BDS. They all use the same language for querying, i.e., retrieving data from the database. This language is called structured query language (SQL). Thus, the same holds as for programming languages: if you know one, you virtually know them all, except for some syntactical variations.

You may have access to a MySQL database via your research group, company, university, or friend—check it out. Otherwise, if you run Ubuntu Linux as suggested in Sect. 4.1 on p. 39, you can install MySQL as described in Sect. 4.1.2.4 on p. 42. Anyway, MySQL server and client software are available for virtually all operating systems (check <http://www.mysql.com>).

Ah, again I will show you the non-graphical interface way of working with MySQL. Though there are some nice tools available both from MySQL and third parties, the free software tool phpMyAdmin (<http://www.phpmyadmin.net>) is quite widespread based on the PHP programming language. But again, learn the basics and enjoy the deeper understanding of what is going on.

16.1 What is MySQL

MySQL is a freely available database management system (DBMS). It consists of a database server and a client. All data are saved and managed at the server side. The client is a software that connects to the server via the network. Interestingly, both server and client can sit on the same computer. There are other database management systems like the free PostgreSQL, the commercial Oracle DB, the commercial

Table 16.1 *annotation.tab*

Gene	Function	Metabolism
alr2938	iron superoxide dismutase	Detoxification
alr4392	nitrogen-responsive regulator	Nitrogen assimilation
alr4851	preprotein translocase subunit	Protein and peptide secretion
alr3395	adenylosuccinate lyase	Purine biosynthesis
alr1207	uridylate kinase	Pyrimidine biosynthesis
alr5000	CTP synthetase	Pyrimidine biosynthesis
all3556	succinate dehydrogenase	TCA cycle

Table 16.2 *expression.tab*

Gene	Expression level
alr1207	8303
alr2938	10323
alr3395	1432
all3556	8043
alr4392	729
alr4851	633
alr5000	5732

Microsoft Access system, and many many more. They all have in common that they use the structured query language (SQL). This is a standardized language that allows communication with a relational database system. Thus, when you know MySQL, you know them all. Different database management systems differ in their architecture and functionality; however, the basis is the same for all. And this chapter can only provide a short introduction to the basics. Well, relational database—not rational—what the heck does this mean?

16.1.1 Relational Databases

A relational database means that data are organized into tables that have a relation to each other. Thus, we have a database that consists of tables, which in turn consists of entries (rows). A relational database is the predominant data storing model, over other models like hierarchical, network, or object-oriented database models. Remember, everything you can store in a table (for God’s sake: Excel) you can put into a relational database.

Throughout this chapter, we will work with two tables that contain gene annotation (Table 16.1) and gene expression data (Table 16.2). Both tables are related to each other by the gene identifier. For e.g. gene *alr2938* codes for an iron superoxide dismutase and has an expression level of 10323.

16.2 Administration

This section describes the administration of a MySQL server and requires that you own administration rights to your computer. It is not absolutely necessary to read unless you are interested. Personally, I think it is always worthwhile to learn what is going on behind the scene.

16.2.1 Get in Touch with the MySQL Server

As mentioned above, the MySQL server, or daemon, is the heart of the database system. It carries all data, executes the queries, and so forth.

Since MySQL is already installed on almost all Linux distributions, I will not describe its installation here (take a look at <http://www.mysql.com>). Though, if you run Ubuntu Linux, take a look at Sect. 4.1.2.4 on p. 42. It is, however, likely that MySQL, although installed, is not running. To check this, execute the command shown in Terminal 189.

```
Terminal 189: Checking if MySQL Server is Running
1  $ mysqladmin ping
2  mysqladmin: connect to server at 'localhost' failed
3  $
```

If you are successful at that point and get the same output as shown in Terminal 189, you can be happy. The MySQL daemon `mysqld` is running and ready to serve you. If you get an output as in Terminal 190, then jump to Sect. 16.2.2 on the next page.

```
Terminal 190: Checking if MySQL Server is Running
1  $ mysqladmin ping
2  mysqladmin: connect to server at 'localhost' failed
3  error: 'Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)'
4  Check that mysqld is running and that the socket: '/tmp/mysql.sock' exists!
5  $
```

If you receive the message “command not found”, then MySQL is most probably not installed on your system. The unlikely case that MySQL commands have not been added to the system path can be tested with `find / -name "mysqladmin"`. This queries the whole directory tree for files containing the word `mysqladmin`. You might like to restrict the query to the folder `usr`, since MySQL is typically installed here. Terminal 191 shows the output of such a query from MacOSX.

```
Terminal 191: Finding MySQL Binaries
1  $ find /usr/ -name "mysqladmin"
2  /usr/local/mysql/bin/mysqladmin
3  find: /usr/local/mysql/data: Permission denied
4  find: /usr/local/mysql/share/mysql/japanese-sjis: Permission denied
5  $
```

Line 2 gives the complete path to the command `mysqladmin`, which is `/usr/local/mysql/bin/`. Take a look at Sect. 10.2 on p. 127 how you add this path to your environment. From now on, I assume that the path to the MySQL commands is known to your system and you just have to type the command name in order to execute it.

16.2.2 Starting and Stopping the Server

As any server, the MySQL Server can be started and stopped. At least for the former you will need administrator rights. Starting or stopping the server can be achieved in different ways. The most convenient way is that your systems take care of it or provide a graphical user interface to perform such tasks. But let us assume for a moment that you have no access to such nice tools. Then you can start the server as shown in Terminal 192.

```
_____ Terminal 192: Starting MySQL _____
1  $ sudo mysqld --user=mysql &
2  Password:
3  061214 13:48:38 [Warning] Setting lower_case_table_names=2 because file system for
4  /usr/local/mysql/data/ is case insensitive
5  061214 13:48:39 InnoDB: Started; log sequence number 0 43655
6  061214 13:48:39 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
7  Version: '5.0.27-standard' socket: '/tmp/mysql.sock' port: 3306
8  MySQL Community Edition - Standard (GPL)
9  $
```

Since running MySQL requires root privileges, we precede the actual command `mysqld` (MySQL daemon) with the command `sudo` (superuser do). This allows execution of administrative commands without logging in as administrator (root). However, in line 2, you are still asked for the root password. The MySQL daemon accepts many options. In line 1 of Terminal 192, we apply the option- `-user` to force the system to run the MySQL daemon as user `mysql`. Otherwise it runs as root, which you should avoid for security reasons. Finally, we tell Linux to run the MySQL server in the background, thus the `&` character. After execution of line 1 and giving the correct password in line 2 of Terminal 192 you get some system notifications—MySQL is running. Let us check this.

```
_____ Terminal 193: Checking if MySQL Server is Running _____
1  $ mysqladmin ping
2  mysqladmin is alive
3  $
```

The MySQL daemon is alive, thus you could now connect to it. But before we do this, I like to show you how to stop the server again in Terminal 194.

```
_____ Terminal 194: Stopping MySQL _____
1  $ mysqladmin -u root shutdown
2  061214 13:52:21 [Note] /usr/local/mysql/bin/mysqld: Normal shutdown
3  061214 13:52:24 InnoDB: Starting shutdown...
4  061214 13:52:26 InnoDB: Shutdown completed; log sequence number 0 43655
```

```
5 | 061214 13:52:26 [Note] /usr/local/mysql/bin/mysqld: Shutdown complete
6
7 | [1]+  Done                                sudo /usr/local/mysql/bin/mysqld --user=mysql
8 | $
```

The command is again `mysqladmin`. The option `-u` (user) tells `mysqladmin` to execute run for the user root. Do not confuse this user root with the root of Linux. The MySQL server has its own user list and root is the default user. Furthermore, only root has the privileges to change setting of the server, set up new user accounts, or shutdown the server. This is why we must run `mysqladmin` for root to shutdown the server. Finally, we provide the option `shutdown` in line 1 in Terminal 194. This is the action `mysqladmin` shall perform. As you can see from the output in line 5, MySQL shuts down successfully. Now, restart the server again.

For the following exercises, I assume that the MySQL server is running.

16.2.3 Setting a Root Password

The first thing you should do after successful installation and start of MySQL is to set a root password. As I mentioned before, the MySQL server provides user accounts, as Linux does. Remember that the accounts are independent of each other. As for Linux, the administrator account for the MySQL server is called root. Depending on your Linux distribution, this account may not be password protected—Ubuntu will ask you for a root password during the installation. You can check this as shown in Terminal 195.

```
Terminal 195: Logging In to MySQL Server
1 | $ mysql -u root
2 | Welcome to the MySQL monitor.  Commands end with ; or \g.
3 | Your MySQL connection id is 2 to server version: 5.0.27-standard
4 |
5 | Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
6 |
7 | mysql> QUIT
8 | Bye
9 | $
```

The command used in line 1 in Terminal 195 is `mysql` together with the option `-u` (user) and the username root. The command `mysql` calls the MySQL Client. With the client, you can connect to the server. Here, we login to the MySQL user account of the administrator root—and it works, which is bad. In line 7 you can see that the bash prompt changed to the MySQL prompt. Executing the MySQL command `QUIT` brings you back to the shell. Of course, you would not like to allow everybody to login as root. Thus, we will set a root password for the MySQL server. This is done as follows.

```
Terminal 196: Setting Root Password
1 | $ mysqladmin -u root -h localhost password "mynewpassword"
2 | $ mysql -u root
```

```

3 | ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
4 | $ mysql -u root -p
5 | Enter password:
6 | Welcome to the MySQL monitor.  Commands end with ; or \ \textrm{g}.
7 | Your MySQL connection id is 2
8 | Server version: 5.5.20 MySQL Community Server (GPL)
9 |
10 | Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.
11 |
12 | Oracle is a registered trademark of Oracle Corporation and/or its
13 | affiliates. Other names may be trademarks of their respective
14 | owners.
15 |
16 | Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
17 |
18 | mysql> QUIT
19 | Bye
20 | $

```

In line 1, we execute `mysqladmin` for the user `root`. With `-h localhost` (`h` = host) we ensure that only the local MySQL server at your computer is affected. The administrative task we want to perform for the user `root` is setting a password, thus `password "mynewpassword"`, where *mynewpassword* is the password to be set. Upon execution you are done. If you would try to login as `root` without password, as shown in line 2 at Terminal 196, you receive an error message telling you that access has been denied. From now on, you must provide the option `-p` (password), which initiates a password request, as shown in lines 4 and 5. This is true for all MySQL commands executed from the shell, e.g., `shutdown` does now require a password, i.e., the option `-p`: `mysqladmin -u root -p shutdown`!

There is another security issue you should solve. The default installation of MySQL usually creates an unnamed user account, again without password protection. Try to login with just the command `mysql` without any option. If you succeed, you have such an unsecured account and should delete it. Therefore, you should login to the MySQL server as `root` and execute the commands as shown in Terminal 197.

```

----- Terminal 197: Erasing MySQL Accounts -----
1 | $ mysql -u root -p
2 | Enter password:
3 | Welcome to the MySQL monitor.  Commands end with ; or \g.
4 | Your MySQL connection id is 11 to server version: 5.0.27-standard
5 |
6 | Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
7 |
8 | mysql> USE mysql;
9 | mysql> DELETE FROM user WHERE password="";
10 | Query OK, 3 rows affected (0.10 sec)
11 |
12 | mysql>QUIT
13 | $

```

Lines 1–7 should be self-explaining by now; we call the MySQL client `mysql` for the user `root` (`-u root`), who has a password protected account (`-p`). The MySQL server can store many databases, each with a unique name. Before we start to work with a particular database, we must tell the server which one we like to use. This

we do with the command `USE` in line 8 of Terminal 197. We work with the MySQL system database named *mysql*, which contains, e.g., data about all user accounts. Line 9 follows your first MySQL query—wow. It deletes all entries from the table *user* where the field *password* is empty. For better readability, I wrote all MySQL commands in upper case. MySQL, however, is case insensitive. Also note that the command has to be followed by a semicolon character. The output in line 10 tells us that 3 entries (rows) have been affected. Finally, we leave the MySQL server and close the client with the command `QUIT`.

Let us now set up an ordinary user account. Then you are actually ready to utilize MySQL.

16.2.4 Set Up an User Account and Database

Up to now, you only have a root account for the MySQL server. You should, however, always work as a normal user to avoid trouble caused by, e.g., too little sleep causing accidental deletions. Therefore, let us set up an user account for a user named *awkologist* with the password *awkology*. You can of course use your own name.

```
Terminal 198: Setting Up Account for User awkologist
1  $ mysql -u root -p
2  Enter password:
3  Welcome to the MySQL monitor.  Commands end with ; or \g.
4  Your MySQL connection id is 55
5  Server version: 5.5.24-0ubuntu0.12.04.1 (Ubuntu)
6
7  Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.
8
9  Oracle is a registered trademark of Oracle Corporation and/or its
10 affiliates. Other names may be trademarks of their respective
11 owners.
12
13 Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
14
15 mysql> GRANT ALL ON compbiol.* TO awkologist@localhost IDENTIFIED BY "awkology";
16 Query OK, 0 rows affected (0.01 sec)
17
18 mysql> QUIT
19 Bye
20 $
```

With the `GRANT` command in line 15 in Terminal 198 we can set access rights for users. Here, we provide all (`ALL`) to all tables (`*`) rights on the database *compbiol* to the user *awkologist* from the *localhost*, e.g., the computer you are sitting in front of. If you would like to allow remote access, you should use either *awkologist@123.123.123.123* or *awkologist@client.com* to specify an IP or client domain, or *awkologist@%* to allow access from any computer. Anyway, user *awkologist* has to identify himself by the password *awkology*. That is it.

Now, let us login as *awkologist* and create the database *compbio1*. Terminal 199 shows how.

```

1  $ mysql -u awkologist
2  ERROR 1045: Access denied for user 'awkologist'@'localhost' (using password: NO)
3  $ mysql -u awkologist -p
4  Enter password:
5  Welcome to the MySQL monitor.  Commands end with ; or \g.
6  Your MySQL connection id is 57
7  Server version: 5.5.24-0ubuntu0.12.04.1 (Ubuntu)
8
9  Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.
10
11  Oracle is a registered trademark of Oracle Corporation and/or its
12  affiliates. Other names may be trademarks of their respective
13  owners.
14
15  Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
16
17  mysql> SHOW DATABASES;
18  +-----+
19  | Database |
20  +-----+
21  | information_schema |
22  | test |
23  +-----+
24  2 rows in set (0.00 sec)
25
26  mysql> CREATE DATABASE mydb;
27  ERROR 1044: Access denied for user 'awkologist'@'localhost' to database 'mydb'
28  mysql> CREATE DATABASE compbio1;
29  Query OK, 1 row affected (0.04 sec)
30
31  mysql> SHOW DATABASES;
32  +-----+
33  | Database |
34  +-----+
35  | information_schema |
36  | compbio1 |
37  | test |
38  +-----+
39  3 rows in set (0.00 sec)
40
41  mysql>

```

Lines 1–2 show an unsuccessful login. This is a common error: the option `-p` has been forgotten. Line 3 shows the correct command. You will be asked to enter your password and have to press the **Enter** key. In line 17, we instruct MySQL to show us all available databases. As you can see, two databases are already present. They belong to the MySQL system. In line 26, we try to create the database *mydb*. Since user *awkologist* has only access to database *compbio1* (see Terminal 198 on the facing page), he cannot create another database. Thus the error message in line 27. The database *compbio1*, however, can be created without any problems (line 28). In line 31, we ask MySQL to display all existing databases again and—there it is: *compbio1*. Now, we have a user and a database—time to play around.

16.3 Utilization

In this section, you will learn some basic operations to store and query your data in a MySQL database. It requires that (a) you have an user account on a MySQL server and full access to a database. If you run your own MySQL server, read Sect. 16.2.4 on p. 301 for a basic setup. Throughout this section, I will work with the user *awkologist* (password *awkology*) who has full access to the database *compbiol*.

16.3.1 Remote Access

Provided that you have an account at a remote computer, e.g., the university or company server, then you can remotely login to that account with the MySQL client. Terminal 200 shows how.

```
Terminal 200: Remote Login
1  $ mysql -u awkologist -p -h db.uniserver.com compbiol
2  Enter password:
3  Reading table information for completion of table and column names
4  You can turn off this feature to get a quicker startup with -A
5
6  Welcome to the MySQL monitor.  Commands end with ; or \g.
7  Your MySQL connection id is 4194633
8  Server version: 5.1.41-3ubuntu12.10 (Ubuntu)
9
10 Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.
11
12 Oracle is a registered trademark of Oracle Corporation and/or its
13 affiliates. Other names may be trademarks of their respective
14 owners.
15
16 Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
17
18 mysql> quit;
19 Bye
20 $
```

The magic command sits right in the first line. What is new is that we pass the URL (*db.uniserver.com*) of the MySQL server to the `mysql` command with the option `-h`. You will, of course, need an internet connection. This way you can connect to any MySQL server on this globe. You only need the client software and a login. If your firewall blocks the connection, open port 3306. This is the default port for MySQL network connection.

16.3.2 MySQL Syntax

In contrast to Linux, MySQL is not case sensitive. However, I will print all MySQL commands in capital letters throughout this chapter. MySQL requires that all commands end with the semicolon character. I am sure that you are forgetting to type it

Table 16.3 MySQL prompts you may see and a description what MySQL is expecting from you

Input prompt	Description
mysql>	Ready for a new MySQL command
->	Waiting for the next line of a multi-line-command
'>	Waiting for completion with a closing single quote
">	Waiting for completion with a closing double quote
`>	Waiting for completion with a closing back tick
*>	Waiting for completion of a comment

from time to time. Since MySQL assumes that your command continues at the next line, its input prompt will change (see line 4 in Terminal 201). By the way, the `USE` command tells MySQL to apply all following commands to the database *compbio1*.

Terminal 201: MySQL Syntax

```
1  mysql> USE compbio1;  
2  Database changed  
3  mysql> SHOW TABLES  
4      -> ;  
5  Empty set (0.00 sec)  
6  
7  mysql> SHOW TABLES  
8      -> \c  
9  mysql>
```

What to do? Either you type the semicolon in that new line (line 4) or you interrupt the current command by typing `\c` (line 8). This resembles command interruption in the shell with `(Ctrl)-C`. The latter, however, would kill the MySQL client. Table 16.3 shows you some possibilities how your MySQL prompt might look like and what it does mean.

16.3.3 Creating Tables

In Sect. 16.2.4 on p. 301, we created the database *compbio1*. This database can now be filled with tables. Remember to execute `USE compbio1` at the beginning of each MySQL session. This will instruct the server to apply all following commands to the specified database. If you forget this, you will get an error message. For the following examples, we are going to work with Tables 16.2 on p. 296 and 16.1 on p. 296. Creation of a table means that we specify what type of data the table contains. Table 16.4 shows some available data types for MySQL tables. Table 16.1 on p. 296 consist of three columns containing text of different length. Okay, Terminal 202 shows how to create the table *annotation* with three fields (columns) of data type `VARCHAR`.

Table 16.4 MySQL data types

Command	Description
INT	Integer
FLOAT	Small floating-point number
DOUBLE	Double-precision floating-point number
CHAR(N)	Text; N characters long (N = 1..255)
VARCHAR(N)	Variable length text up to N characters long
TEXT	Text up to 65535 characters long
LONGTEXT	Text up to 4294967295 characters long
BINARY	Binary data like images

```

1  mysql> CREATE TABLE annotation
2      -> (gene VARCHAR(7), function VARCHAR(30),
3      -> metabolism VARCHAR(30));
4  Query OK, 0 rows affected (0.12 sec)
5
6  mysql> SHOW TABLES;
7  +-----+
8  | Tables_in_compbio1 |
9  +-----+
10 | annotation          |
11 +-----+
12 1 row in set (0.00 sec)
13
14 mysql> DROP TABLE annotation;          # DO NOT EXECUTE
15 mysql>

```

Note that you could put the commands in lines 1–3 in one line. As you can see, we create three fields named *gene*, *function*, and *metabolism*, respectively. The first field accepts text strings of maximum seven characters length, and so forth. The general command is

CREATE TABLE *table*(*fieldname* data type)

In line 6, we execute SHOW TABLES and see—wow—our first MySQL table has been generated. Line 14 introduces the DROP command that can be used to delete a table with all its data. If you accidentally execute that line, you have to start from the beginning. Luckily enough, MySQL has a command history, too. You can scroll through old commands as in the Bash shell with the arrow keys.

In Terminal 203, you see how you can check the format of an existing table with DESCRIBE.

```

1  mysql> DESCRIBE annotation;
2  +-----+-----+-----+-----+-----+-----+
3  | Field | Type | Null | Key | Default | Extra |
4  +-----+-----+-----+-----+-----+-----+
5  | gene  | varchar(7) | YES | | NULL | |
6  | function | varchar(30) | YES | | NULL | |
7  | metabolism | varchar(30) | YES | | NULL | |
8  +-----+-----+-----+-----+-----+-----+
9  3 rows in set (0.00 sec)
10
11 mysql>

```

The meaning of the columns *Field* and *Type*, you know. I skip the meaning of the other columns for this small introduction. Instead, let us see how we get data into the yet empty table skeleton.

16.3.4 Filling and Editing Tables

There are endless number of ways to populate MySQL tables with data. Almost every programming language provides an interface to MySQL—except AWK, sorry. I show you the command line way in either the MySQL command line or by piping data to the MySQL client in the Bash shell.

The direct way is shown in Terminal 204.

```

1  mysql> INSERT INTO annotation
2      -> (gene, function, metabolism) VALUES
3      -> ("alr4851",
4      -> "preprotein translocase subunit",
5      -> "Protein and peptide secretion");
6  Query OK, 1 row affected (0.02 sec)
7
8  mysql> INSERT INTO annotation VALUES
9      -> ("alx2938", "iron superoxide dismutase", "Detoxification");
10 Query OK, 1 row affected (0.01 sec)
11
12 mysql> SELECT * FROM annotation;
13 +-----+-----+-----+
14 | gene   | function                                     | metabolism |
15 +-----+-----+-----+
16 | alr4851 | preprotein translocase subunit | Protein and peptide secretion |
17 | alx2938 | iron superoxide dismutase      | Detoxification |
18 +-----+-----+-----+
19 2 rows in set (0.03 sec)
20
21 mysql>

```

The general command is

```
INSERT INTO table VALUES (data)
```

For the first execution in lines 1–5, we specify the field names (line 2). For the second execution, in lines 8–9, I omitted these. Since we provide three data values and the table has the columns, data allocation is unambitious. However, the correct and error-proof way is the first.

In line 12, we display the whole content of the table *annotation* with the construct

```
SELECT * FROM table
```

In line 9, I misspelled *alx2938*, it should read *alr2938*. In Terminal 205, I illustrate the function of the UPDATE command.

```

1  _____ Terminal 205: Editing Data _____
2  mysql> UPDATE annotation SET gene="alr2938" WHERE gene="alx2938";
3  Query OK, 1 row affected (0.04 sec)
4  Rows matched: 1  Changed: 1  Warnings: 0
5
6  mysql> SELECT * FROM annotation;
7  +-----+-----+-----+-----+
8  | gene   | function                                     | metabolism |
9  +-----+-----+-----+-----+
10 | alr4851 | preprotein translocase subunit | Protein and peptide secretion |
11 | alr2938 | iron superoxide dismutase      | Detoxification |
12 +-----+-----+-----+-----+
13 2 rows in set (0.01 sec)
14
15 mysql>

```

The general syntax of the command is

`UPDATE table SET edit function WHERE query`

The query specifies what we are looking for, i.e., all entries where the value in field *gene* is *alx2938*. Likewise, you could use the `DELETE` command:

`DELETE FROM table WHERE query`

If you omit the query, you delete the content of the whole table, but not the table itself (see Terminal 206).

```

1  _____ Terminal 206: Deleting All Data _____
2  mysql> DELETE FROM annotation;
3  Query OK, 0 rows affected (0.00 sec)
4
5  mysql> SELECT * FROM annotation;
6  Empty set (0.00 sec)
7
8  mysql>

```

I am sure that you are not a friend of typing the whole table into the terminal, right? Terminal 207 shows you how to upload data from your local hard disk to a MySQL database. You can download the data for table *annotation* (see Table 16.1 on p. 296) and table *expression* (see Table 16.2 on p. 296) from the book's website as tabulator delimited files. It is important that the MySQL table name (*annotation*) matches with the filename *annotation.tab* without the extension.

```

1  _____ Terminal 207: Importing annotation.tab _____
2  ### Start in Bash Shell ###
3
4  $ mysqlimport --local -d --ignore-lines=0 -u awkologist -p compbiol annotation.tab
5  Enter password:
6  compbiol.annotation: Records: 8  Deleted: 0  Skipped: 0  Warnings: 2
7  $
8  ### Switch to MySQL Shell ###
9

```

```

10 |mysql> SELECT * FROM annotation;
11 |-----+-----+-----+
12 | gene | function | metabolism |
13 |-----+-----+-----+
14 | alr2938 | iron superoxide dismutase | Detoxification |
15 | alr4392 | nitrogen-responsive regulator | Nitrogen assimilation |
16 | alr4851 | preprotein translocase subunit | Protein and peptide secretion |
17 | alr3395 | adenylosuccinate lyase | Purine biosynthesis |
18 | alr1207 | uridylate kinase | Pyrimidine biosynthesis |
19 | alr5000 | CTP synthetase | Pyrimidine biosynthesis |
20 | all3556 | succinate-dehydrogenase | TCA cycle |
21 | | NULL | NULL |
22 |-----+-----+-----+
23 | 8 rows in set (0.00 sec)
24
25 |mysql> DELETE FROM annotation WHERE function IS NULL;
26 |Query OK, 1 row affected (0.05 sec)
27
28 |mysql>

```

Lines 1 and 8 are only comments. You should start in the Bash shell from the same folder where you saved *annotation.tab*. With the command `mysqlimport`, we upload the content stored in file *annotation.tab* to the database *combiol* owned by user *awkologist*. The option `--local` reads input files locally from the client host and option `--ignore-lines` can be used to ignore the first *n* lines of the input file, e.g., header lines. Option `-d` deletes the content of the table before it is filled. The command `mysqlimport` stores the data in a table that must have the same name as the stem of the input file, here *annotation*. As you can see in line 5, we successfully imported eight records (i.e., lines, entries, and rows) but got two warnings—why is that. This becomes clear when we display the content of the table with the `SELECT` command in line 10. The last line contains no data. This means that *annotation.tab* has one last line that contains only the line break character. If you wish, you can delete this line as shown in line 25.

You might have read somewhere that there is also a `LOAD DATA LOCAL INFILE ...` command to load data from the MySQL terminal. However, there is a security issue with this command. Therefore, I skip it here. Now, let us create the table *expression* and populate it with data.

```

----- Terminal 208: Importing expression.tab -----
1 |### Start in MySQL Shell
2 |
3 |mysql> CREATE TABLE expression(gene VARCHAR(7), expr_value INT);
4 |Query OK, 0 rows affected (0.15 sec)
5 |
6 |mysql>
7 |
8 |### Switch to Bash Shell ###
9 |
10 |$ mysqlimport --local -d --ignore-lines=0 -u awkologist -p combiol expression.tab
11 |Enter password:
12 |combiol.expression: Records: 8 Deleted: 0 Skipped: 0 Warnings: 1
13 |$
14 |
15 |### Switch to MySQL Shell ###
16 |
17 |mysql> SELECT * FROM expression;
18 |-----+-----+-----+

```

```

19 | gene | expr_value |
20 +-----+-----+
21 | alr1207 | 8303 |
22 | alr2938 | 10323 |
23 | alr3395 | 1432 |
24 | al13556 | 8043 |
25 | alr4392 | 729 |
26 | alr4851 | 633 |
27 | alr5000 | 5732 |
28 | | NULL |
29 +-----+-----+
30 8 rows in set (0.00 sec)

31
32 mysql> DELETE FROM expression WHERE gene="";
33 Query OK, 1 row affected (0.06 sec)
34
35 mysql>

```

In Terminal 208, we first create the MySQL table *expression* (lines 3–6), import the content of file *expression.tab* (lines 10–13), and finally display the content of the new table in line 32. Note that I used a different query for deleting the empty entry in line 32. Here, we delete records where a field is empty, whereas in Terminal 207 all records with a field matching *NULL* are deleted. The value *NULL* must not be confused with the number 0 or an empty string. *NULL* simply means that there is no data, not even an empty string.

Okay, now it is time to take a closer look to the heart of databasing—execution of database queries.

16.3.5 Querying Tables

The following terminals show examples of how you can read out information from tables. The elemental command is *SELECT* that comes with an endless number of variations and options. The typical structure is:

```
SELECT field names FROM table names WHERE conditions
```

Terminal 209 introduces one very handy option, i.e., *LIMIT*. It has a similar function as *head* in the Bash shell (see Sect. 7.1.4 on p. 81); it limits the output to the given number of lines.

```

----- Terminal 209: Querying with LIMIT -----
1  mysql> SELECT * FROM expression LIMIT 3;
2  +-----+-----+
3  | gene | expr_value |
4  +-----+-----+
5  | alr1207 | 8303 |
6  | alr2938 | 10323 |
7  | alr3395 | 1432 |
8  +-----+-----+
9  3 rows in set (0.00 sec)
10
11 mysql>

```

In Terminal 210, we really start querying. We print all records of table *expression* where the value of field *expr_value* is less than 1,000.

```

1  mysql> SELECT * FROM expression WHERE expr_value < 1000;
2
3  +-----+-----+
4  | gene      | expr_value |
5  +-----+-----+
6  | alr4392   | 729       |
7  | alr4851   | 633       |
8  +-----+-----+
9  2 rows in set (0.01 sec)
10 mysql>

```

In Terminal 210, we refine the previous query by ordering the output with respect to the values in field *expr_value*.

```

1  mysql> SELECT * FROM expression WHERE expr_value < 1000 ORDER BY expr_value;
2
3  +-----+-----+
4  | gene      | expr_value |
5  +-----+-----+
6  | alr4851   | 633       |
7  | alr4392   | 729       |
8  +-----+-----+
9  2 rows in set (0.01 sec)
10 mysql>

```

It is very common to query for text string that matches a certain pattern. You might be used to use the asterisk character *** as a wildcard. MySQL is different, it uses the percent character *%* as wildcard character (joker). Take a look at Terminal 212.

```

1  mysql> SELECT * FROM annotation WHERE function LIKE '%ase%';
2
3  +-----+-----+-----+
4  | gene      | function      | metabolism |
5  +-----+-----+-----+
6  | alr2938   | iron superoxide dismutase | Detoxification |
7  | alr4851   | preprotein translocase subunit | Protein and peptide secretion |
8  | alr3395   | adenylosuccinate lyase | Purine biosynthesis |
9  | alr1207   | uridylate kinase | Pyrimidine biosynthesis |
10 | alr5000   | CTP synthetase | Pyrimidine biosynthesis |
11 | all3556   | succinate-dehydrogenase | TCA cycle |
12 +-----+-----+-----+
13 6 rows in set (0.00 sec)
14 mysql>

```

Here, we search for all records in table *annotation* where the string stored in field *function* contains the substring *ase*. Test what happens when you omit the first or second wildcard character.

In Terminal 213 on the next page regular expressions (see Chap. 11 on p. 157) are used to query for field values matching a certain pattern.

```

1  ----- Terminal 213: Querying with Regular Expressions -----
2  mysql> SELECT * FROM annotation WHERE metabolism
3  -> REGEXP '(Purine|Pyrimidine) biosynthesis';
4  +-----+-----+-----+
5  | gene   | function                               | metabolism |
6  +-----+-----+-----+
7  | alr3395 | adenylosuccinate lyase                 | Purine biosynthesis |
8  | alr1207 | uridylate kinase                      | Pyrimidine biosynthesis |
9  | alr5000 | CTP synthetase                       | Pyrimidine biosynthesis |
10 +-----+-----+-----+
11 3 rows in set (0.00 sec)
12
13 mysql> SELECT gene, function FROM annotation WHERE metabolism
14 -> REGEXP '(Purine|Pyrimidine) biosynthesis';
15 +-----+-----+
16 | gene   | function                               |
17 +-----+-----+
18 | alr3395 | adenylosuccinate lyase                 |
19 | alr1207 | uridylate kinase                      |
20 | alr5000 | CTP synthetase                       |
21 +-----+-----+
22 3 rows in set (0.00 sec)
23
24 mysql> SELECT gene, function FROM annotation WHERE metabolism
25 -> REGEXP '(Purine|Pyrimidine) biosynthesis'\G
26 ***** 1. row *****
27 gene: alr3395
28 function: adenylosuccinate lyase
29 ***** 2. row *****
30 gene: alr1207
31 function: uridylate kinase
32 ***** 3. row *****
33 gene: alr5000
34 function: CTP synthetase
35 3 rows in set (0.00 sec)
36
37 mysql>

```

As you can see in lines 12 and 23, in Terminal 213, `SELECT` cannot only be used to specify which records to show, but also to specify which fields of these records shall be displayed. The only difference between the commands in line 12/13 and 23/24 is the ending, the semicolon, and `\G`, respectively. Note the huge impact on the output format. If you print records with many fields (columns), MySQL has no chance to display everything onto the screen. It introduces line breaks that make the table basically unreadable. The line-wise display of fields circumvents this problem.

Although neither the strength nor the focus of MySQL, it can perform some basic calculations. Terminal 214 shows one example of the syntax.

```

1  ----- Terminal 214: Mathematics -----
2  mysql> SELECT MIN(expr_value) FROM expression;
3  +-----+
4  | MIN(expr_value) |
5  +-----+
6  |          633    |
7  +-----+
8  1 row in set (0.00 sec)
9
10 mysql>

```


Table 16.5 MySQL. selected operations

Command	Description
MIN (<i>field name</i>)	Returns the smallest value of <i>numeric value</i>
MAX (<i>field name</i>)	Returns the highest value of <i>numeric value</i>
AVG (<i>field name</i>)	Returns the average of <i>numeric value</i>
STD (<i>field name</i>)	Returns the standard deviation of <i>numeric value</i>
COUNT (*)	Returns the number matching records
SUM (<i>field name</i>)	Returns the sum of all values of matching <i>numeric value</i>
EXP (<i>numeric value</i>)	Returns e (the base of natural logarithms) raised to the power of <i>numeric value</i>
LN (<i>field name</i>)	Returns the natural logarithm of <i>numeric value</i>
LOG2 (<i>field name</i>)	Returns the base-2 logarithm of <i>numeric value</i>

There are many other operations possible, some of which are listed in Table 16.5. MySQL maintains a very good online manual at <http://dev.mysql.com/doc>. Here, I only want to give an impression of what could be possible.

16.3.6 Joining Tables

Up to this point we only queried a single table. That is nice but not that exciting. The real power of any database engine lays in the joined query over several tables. Suppose, we like to find the gene identifiers, functions and expression values of all genes whose expression value is less than 5000 and that are involved purine or pyrimidine biosynthesis. This kind of questions are typical for the analysis of DNA-microarray data. Terminal 215 shows a very crude way how to answer this question—computer scientists, please look away for a moment.

```

1  ----- Terminal 215: Joint Query -----
2  mysql> SELECT annotation.gene, annotation.function, expression.expr_value
3  -> FROM annotation, expression
4  -> WHERE annotation.gene = expression.gene AND
5  -> annotation.metabolism REGEXP '(Purine|Pyrimidine) biosynthesis' AND
6  -> expression.expr_value < 5000;
7  +-----+-----+-----+
8  | gene   | function                | expr_value |
9  +-----+-----+-----+
10 | alr3395 | adenylosuccinate lyase | 1432 |
11 +-----+-----+-----+
12 1 row in set (0.00 sec)
13
14 mysql> SELECT a.gene, a.function, e.expr_value
15 -> FROM annotation AS a, expression AS e
16 -> WHERE a.gene = e.gene AND
17 -> a.metabolism REGEXP "(Purine|Pyrimidine) biosynthesis" AND
18 -> e.expr_value < 5000;
19 +-----+-----+-----+
20 | gene   | function                | expr_value |
21 +-----+-----+-----+
22 | alr3395 | adenylosuccinate lyase | 1432 |
23 +-----+-----+-----+

```

```

23 | 1 row in set (0.00 sec)
24 |
25 | mysql>

```

Now, let us see what is going on in Terminal 215. In line 1, we specify which field values we like to display: fields *gene* and *function* from table *annotation* and field *expr_value* from table *expression*. In line 2, we have to give the tables we are working with. Lines 3–5 state our query. The special thing here is that we state that the values of the field *gene* in both tables must match (line 3). This way we link both tables via the field *gene*. Then follow the same statements as seen before.

The query starting in line 13 in Terminal 215 produces the same output. However, we use shortcuts for the table names by the

FROM *table name* AS *shortcut name*

construct. This helps to make the query a bit more readable—at least when you are getting used to SQL syntax.

The more accurate syntax from a computer scientist's point of view for this kind of joined query is the following:

```

SELECT a.gene, a.function, e.expr_value
FROM annotation AS a LEFT JOIN expression AS e ON a.gene = e.gene
WHERE a.metabolism REGEXP '(Purine|Pyrimidine) biosynthesis' AND e.expr_value < 5000;

```

This command gives the same output as the commands in Terminal 215. It translates to: select all the rows from the left side of the join (table *annotation*), and for each row selected, either display the matching value from the right side (table *expression*) or display an empty row. As you can imagine, there is also a *RIGTH JOIN* that does the reverse action.

16.3.7 Access from the Shell

The usage of `mysqlimport` command in Terminal 207 on p. 307 already gave you a hint that MySQL can be manipulated from the bash shell. In Sect. 17.7.1 on p. 339 you will see how to access MySQL databases from R. Here, I like to show how you can execute SQL statements in the Bash shell. Start by creating a text file that contains a number of MySQL statements as shown in File 9.

```

# save as query.sql
USE compbiol
SELECT a.function, e.expr_value
FROM annotation AS a LEFT JOIN expression AS e ON a.gene = e.gene
WHERE e.expr_value < 5000;

```

Terminal 216 shows you how to execute the commands in file *query.sql*.

```

Terminal 216: Quitting and Batch Queries
1 $ mysql -u awkologist -p -B < query.sql
2 Enter password:
3 function      expr_value
4 adenylosuccinate lyase  1432
5 nitrogen-responsive regulator  729
6 preprotein translocase subunit  633
7 $ mysql -u awkologist -p -B < query.sql | sed '1d' | awk -F"\t" '{print $1}'
8 Enter password:
9 adenylosuccinate lyase
10 nitrogen-responsive regulator
11 preprotein translocase subunit
12 $

```

This is done by redirecting the file to `mysql` and application of the option `-B` (batch). When `mysql` runs in the batch mode, it will print the records as tabulator delimited lines. These could be piped to any other Linux command. This we do in line 7 where we delete the first line containing the header, and only print the gene functions. Yet, another option is to run an SQL statement directly as shown in Terminal 217.

```

Terminal 217: Command Line Querying
1 $ mysql -u awkologist -p compbiol -e 'SELECT * FROM expression'
2 Enter password:
3 +-----+
4 | gene      | expr_value |
5 +-----+
6 | alr1207   | 8303       |
7 | alr2938   | 10323      |
8 | alr3395   | 1432       |
9 | all3556   | 8043       |
10 | alr4392   | 729        |
11 | alr4851   | 633        |
12 | alr5000   | 5732       |
13 +-----+
14 $

```

In this example, we give the name of the database in the command line. The option `-e` (execute) precedes the actual SQL statement. Since we left away the option `-B`, the output is printed and formatted in the MySQL fashion.

16.4 Backups and Transfers

Finally, let us take a look at one important command that helps to create a database dump: `mysqldump`. A dump is a full copy of a whole database or single tables that can be used to restore the data.

```

Terminal 218: Baching-Up Database with mysqldump
1 $ mysqldump -u awkologist -p compbiol > dump.sql
2 Enter password:
3 $ less dump.sql
4 $ mysql -u awkologist -p compbiol < dump.sql
5 Enter password:
6 $

```

In line 1 in Terminal 218, we dump the complete database `compbiol` to file *dump.sql*. We then look into this file with `less`. As you will see, the file is full of MySQL statements and your original data. Line 4 shows you how to restore the database.

Finally, with the following bash shell command, you would copy the whole database to another remote MySQL server:

```
mysqldump -u awkologist -p compbiol | mysql -u freddy -p -h remote.computer.com
```

If you are rather interested in transferring a single table, then you write the table name behind the database name separated by a space character.

16.5 How to Move on?

I hope that you got an imagination of what you could do with MySQL. I omitted a lot and even important things like keys and indices. It was not my intention to show everything but to wet your appetite. MySQL provides not only hundreds of commands, SQL can even be used like any programming language with conditionals, loops, variables, and so on. I suggest that you browse through the Web and get inspired by the MySQL manual. However, the best thing—again—is to start with a real world problem. I am sure you have one.

Chapter 17

The Statistics Suite R

Data analysis commonly involves visualization of data. This includes both, plotting the data themselves and plotting properties of the dataset like frequencies of certain numbers. There is a broad range of software tools available to perform such tasks. R is a completely free software that emulates S-Plus. S-Plus in turn was initially developed by AT&T's Bell Laboratories to provide a software tool for professional statisticians who wanted to combine state-of-the-art graphics with powerful model-fitting capability. The nice thing with R is that you neither need to be an expert statistician nor a mathematical nerd. One can use R as an absolute beginner and develop statistics skills on the run. Nobody ever learned statistics by reading a book about it. R helps you to try out different analysis tools and visualize the results. Thus, it supports you in learning statistics.

But, even if you are not interested in statistical analysis of your data, R will be of big help. You can create nice-looking plots, automate their creation, connect to MySQL or NCBI GenBank, and do much more. There are many additional packages available for R. They extend its capabilities to compute specific tasks, e.g., the analysis of DNA-microarray data (Bioconductor).

If you run Ubuntu Linux as suggested in Sect. 4.1 on p. 39, you should install R as described in Sect. 4.1.2.4 on p. 42. But R is available for all virtually operating systems (check <http://www.r-project.org>).

17.1 Getting Started

First you should check if R is installed on your system. Open the Terminal and execute the command `R`. If R is installed on your system you should see something like in Terminal 219.

```
Terminal 219: Starting R
1  $ R
2
3  R version 2.14.1 (2011-12-22)
4  Copyright (C) 2011 The R Foundation for Statistical Computing
```

```

5 | ISBN 3-900051-07-0
6 | Platform: i686-pc-linux-gnu (32-bit)
7 |
8 | R is free software and comes with ABSOLUTELY NO WARRANTY.
9 | You are welcome to redistribute it under certain conditions.
10 | Type 'license()' or 'licence()' for distribution details.
11 |
12 | Natural language support but running in an English locale
13 |
14 | R is a collaborative project with many contributors.
15 | Type 'contributors()' for more information and
16 | 'citation()' on how to cite R or R packages in publications.
17 |
18 | Type 'demo()' for some demos, 'help()' for on-line help, or
19 | 'help.start()' for an HTML browser interface to help.
20 | Type 'q()' to quit R.
21 |
22 | >

```

If you do not get an output similar to the lines shown in Terminal 219, then you probably need to install R on your system. In that case go to Sect. 4.1.2.4 on p. 42 or www.r-project.org and follow the instructions.

Terminal 220 shows you some basic operations in R.

```

Terminal 220: Basic Syntax in R
1 | > x<-12
2 | > y<-23
3 | > x+y
4 | [1] 35
5 | > z<-3*x
6 | > z
7 | [1] 36
8 | > a<-434 # Comment
9 | > a
10 | [1] 434
11 | > ls()
12 | [1] "a" "x" "y" "z"
13 | > rm(a,z)
14 | >

```

In lines 1–2 we assign values to the variables x and y . Note that the R-typical assignment operator is $<-$, however, you could also use $=$. In line 4, we see the result of $x+y$. The preceding $[1]$ is always present. You will see later why. In line 11, we instruct R to printout all variables that have been assigned. In line 13, we use the command `rm()` to remove variables a and z from the R session. This shows you already the typical syntax of R commands: command name plus a pair of parenthesis. If the command requires no arguments, the parentheses are empty (line 11), else they contain the comma-separated arguments.

```

Terminal 221: The First Plot
1 | > a<-1:10
2 | > a
3 | [1] 1 2 3 4 5 6 7 8 9 10
4 | > length(a)
5 | [1] 10
6 | > b<-c(23,56,67:72,98,65) # Command c() -> concatenate
7 | > length(b)

```

```

8 [1] 10
9 > plot(a,b)                                     # Plot opens in extra window
10 > c<-seq(from=10,to=100,by=10)
11 > c
12 [1] 10 20 30 40 50 60 70 80 90 100
13 > points(a,c,col="red",pch=2)
14 > lines(a,c,col="blue")
15 >

```

In Terminal 221, we assign the numbers 1–10 to variable *a* and 23, 56, 67, 68, 69, 70, 71, 72, 98, 65 to variable *b*. The command `length()` prints the number of values assigned to a variable. This introduces an important aspect of R: variables are in fact one-dimensional arrays or vectors. Figure 17.1 shows you the result of the `plot()` command in line 9. Note that an XY-plot is created. Thus, the number of elements in vector *a* must match the length of vector *b*.

In line 10, we assign the numbers 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 to vector *c*. The function `seq()` defines a series. The command `points()` adds the XY-data of *a* and *c* to the plot as points. Attribute *col* specifies the color and *pch*, the plotting character. The `lines()` function in line 14 adds the same data but only as blue lines.

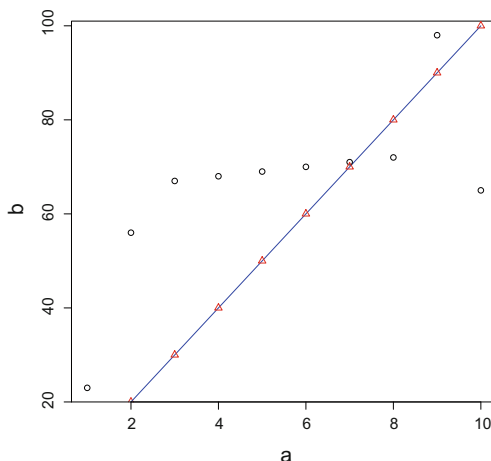
Terminal 222 illustrates how individual vector elements can be retrieved.

```

----- Terminal 222: Referring to Vector Elements -----
1 > b                                     # Vector b
2 [1] 23 56 67 68 69 70 71 72 98 65
3 > b[3]                                 # Third element in vector b
4 [1] 67
5 > b<=70
6 [1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE
7 > b[b<=70]                             # Values of vector b that match condition
8 [1] 23 56 67 68 69 70 65
9 > which(b<=70)                         # Indices of vector b elements that match condition
10 [1] 1 2 3 4 5 6 10
11 >

```

Fig. 17.1 XY-Plot. Result from Terminal 221



In line 1, we print all vector elements. In line 3, we refer to vector element 3. In line 5 we ask which vector element has a value smaller or equal to 70. The result is a Boolean vector, that states for each element the condition is true or false. This can be used in, e.g., `if` clauses (see Sect. 17.4 on p. 325). Finally, we check for either the value (line 7) or index (line 9) of vector elements that match the condition.

Typically, you would load your own data from a data file into R. The procedure is shown in the following section. Alternatively, you might like to retrieve your data from a MySQL table. That requires installation of an additional package as described in Sect. 17.7.1 on p. 339.

17.1.1 Getting Help

As Linux, R comes along with a full user manual. There are two possibilities to open the manual pages. Either you use the `help()` command or you precede the command you are looking for with a questionmark. Thus, both `help("plot")` and `?plot` would open the manual page for the `plot()` function. The manual pages are displayed with the file viewer (`less`), and provides the same functionality (see Sect. 7.1.5 on p. 81). Thus, you get back to the R command line with `Q`.

With `apropos()` you get a list of commands that are related to the query term.

17.2 Reading and Writing Data Files

The simplest way to load a bunch of data into R provides the command `read.table()`. Its application is shown in Terminal 223. But, let us first take a look at the content of the data file *gene-exprA-exprB.tab* we are going to load. It contains gene expression data for a cyanobacterium in two different environmental conditions (File 10).

```

1  gene      exprA      exprB
2  all0001  1109      2202
3  all0002  1094      1373.25
4  ...
5  asr7657  153      249.5
6  asr8504  142      907

```

File 10: *gene-exprA-exprB.tab*

The file contains data for 1251 genes. You can download the file from the book's webpage. As shown, the file consists of three columns and one header row.

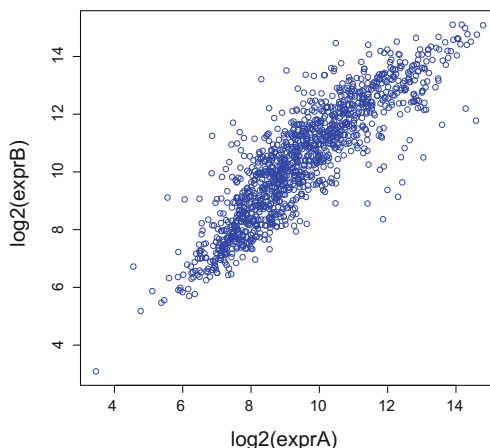
```

1  > data<-read.table("gene-exprA-exprB.tab",header=TRUE,sep="\t")
2  > names(data)
3  [1] "gene" "exprA" "exprB"
4  > attach(data)
5  > plot(log2(exprA),log2(exprB),col="blue")
6  >

```

Terminal 223: Importing Data with `read.table()`

Fig. 17.2 Plotting Gene-Expression Data. Result from Terminal 223



In line 1 of Terminal 223 we load the content of file *gene-exprA-exprB.tab* into the variable *data*. The function `read.table()` requires the filename and path. Here, I skipped the path because the file sits in the same folder from where I called R. The argument *header* is set to `TRUE`, because the first line in the data file contains the column names. As file separator, we set the tabulator with the argument *sep*.

In line 2, we let R display the header names with `names()`. Then, we make these names available to the system with the `attach()` command. This means from now on we can use *gene*, *exprA*, and *exprB* as variables' names. Note that R might modify column names, e.g., spaces would be replaced by underscored characters. Finally, we create an XY-plot of the data as shown in Fig. 17.2.

For the plot, we take the logarithm to base 2 with `log2()`. In Sect. 17.5 on p. 327 you will learn how to retrieve more information about the dataset. But, first let us see how you can write data into a file.

You may guess that the command to write data into a file is `write.table()`. It accepts a number of options as exemplified in Terminal 224. I assume that you are still in the same R session as in Terminal 223, and have the data file *gene-exprA-exprB.tab* loaded and attached in the workspace.

```

Terminal 224: Writing into Files with write.table()
1  > write.table(data, "output-data.csv", append=FALSE, col.names=TRUE,      +
2      row.names=FALSE, quote=FALSE, sep=", ")
3  > system("head -3 output-data.csv")
4  gene,exprA,exprB
5  all0001,1109,2202
6  all0002,1094,1373.25
7  > write.table(data, "output-data.csv", append=FALSE, col.names=TRUE,      +
8      row.names=FALSE, quote=TRUE, sep=", ")
9  > system("head -3 output-data.csv")
10 "gene", "exprA", "exprB"
11 "all0001", 1109, 2202
12 "all0002", 1094, 1373.25
13 > write.table(data, "output-data.csv", append=FALSE, col.names=TRUE,      +
14     row.names=TRUE, quote=TRUE, sep=", ")
15 > system("head -3 output-data.csv")

```

```

16 "gene", "exprA", "exprB"
17 "1", "all10001", 1109, 2202
18 "2", "all10002", 1094, 1373.25
19 > write.table(exprA, "output-data.csv", append=FALSE, col.names=FALSE,
20               row.names=FALSE, quote=FALSE, sep=", ")
21 > system("head -3 output-data.csv")
22 1109
23 1094
24 1968.5
25 > write.table(list(exprA, exprB), "output-data.csv", append=FALSE,
26               col.names=FALSE, row.names=FALSE, quote=FALSE, sep=", ")
27 > system("head -3 output-data.csv")
28 1109, 2202
29 1094, 1373.25
30 1968.5, 7153.5
31 >

```

The arguments are used to `write.table()` specify how the data are formatted. The first argument states what to print. Usually, this is a data frame (see Sect. 17.3). However, as can be seen in lines 19 and 25 you may also choose a single vector or a list of vectors. Next follows the filename. The file will be created in the folder where you called R, except that you set a path. The *append* argument determines whether the data shall be appended to the file, else the file will be overwritten. With *col.names* and *row.names*, we specify if we wish to print column names or row names/numbers. The *quote* option can be set to enclose all data in double quotes. Finally, *sep* specifies the column separator.

The `system()` function executes a Bash command and pipes the output into the R command shell—is this cool? You will see an even smarter way to use `system()` in Sect. 17.5.2.1 on p. 333. But now let us see how data structures look like in R.

17.3 Data Structures

This section describes different data structures in R. It is not a must to read but will help you in understand, e.g., package instructions (see Sect. 17.7 on p. 339).

17.3.1 Vectors

The simplest data structure is the vector. You are already familiar with this data type.

17.3.2 Arrays and Matrices

Arrays are multidimensional vectors. Matrices are two-dimensional (2D) arrays. That is it. R provides the `array()` command to define an array. Terminal 225 shows an example.

```

Terminal 225: Data Structures
1 > v<-array(c(11,21,31,12,22,32,13,23,33,14,24,34,15,25,35),dim=c(3,5))
2 > v
3      [,1] [,2] [,3] [,4] [,5]
4 [1,]  11  12  13  14  15
5 [2,]  21  22  23  24  25
6 [3,]  31  32  33  34  35
7
8 > v[2,]                                # Row 2
9 [1] 21 22 23 24 25
10 > v[,3]                                # Column 3
11 [1] 13 23 33
12
13 > ThreeD<-array(c(111,121,131,112,122,132,113,123,133,114,124,134,115,125,135,
14                  211,221,231,212,222,232,213,223,233,214,224,234,215,225,235),
15                  dim=c(3,5,2))
16 > ThreeD
17 , , 1
18
19      [,1] [,2] [,3] [,4] [,5]
20 [1,]  111  112  113  114  115
21 [2,]  121  122  123  124  125
22 [3,]  131  132  133  134  135
23
24 , , 2
25
26      [,1] [,2] [,3] [,4] [,5]
27 [1,]  211  212  213  214  215
28 [2,]  221  222  223  224  225
29 [3,]  231  232  233  234  235
30
31 > ThreeD[3,5,2]                        # Element in row 3, column 5, table 2
32 [1] 235
33 > ThreeD[3,,1]                          # Row 3 in table 1
34 [1] 131 132 133 134 135
35 > ThreeD[,4,2]                          # Column 4 in table 2
36 [1] 214 224 234
37 > ThreeD[c(1,3),,1]                    # Rows 1 and 3 in table 1
38      [,1] [,2] [,3] [,4] [,5]
39 [1,]  111  112  113  114  115
40 [2,]  131  132  133  134  135
41
42 > matrix(data=c(11,21,12,22,13,23,14,24,15,25,16,26), nrow=2, ncol=6)
43      [,1] [,2] [,3] [,4] [,5] [,6]
44 [1,]  11  12  13  14  15  16
45 [2,]  21  22  23  24  25  26
46 >

```

As you can see, it is pretty simple to create multidimensional arrays and refer to specific cells, rows, or columns. Take a specific look at line 37. Here, we extract rows 1 and 3 from the first table. The function `matrix()` is just another way to create a 2D array.

Up-to-now we only used vectors and arrays of one specific data type, i.e., numbers. What happens when we mix data types as shown in Terminal 226.

```

Terminal 226: Text Array
1 > v<-array(c(11,21,31,12,22,32,13,23,33,14,24,34,15,25,35), dim=c(3,5))
2 > v[2,2]+1
3 [1] 23
4 > v<-array(c(11,21,31,12,22,32,13,23,"text",14,24,34,15,25,35), dim=c(3,5))
5 > v[2,2]+1

```

```

6 | Error in v[2, 2] + 1 : non-numeric argument to binary operator
7 | >

```

Then we run into problems. In line 1 in Terminal 226, we create a numeric array. In line 2, we add 1 to the array element in row two and column two. In line 4 substitute one array element with a text string. As a result, all array elements become text type and calculations are not possible. That stinks, e.g., when you load data as shown in File 10 on p. 320. The answer to the problem is lists.

17.3.3 Lists and Data Frames

Lists are arrays with mixed data types. When we applied the function `read.table()` to import the data from File 10 on p. 320, a list was created (to be exact: a data frame, but this is a type of list as you will see below). This means that although column one contains text type data, we could perform calculation with the data in column two. Terminal 227 illustrates this.

```

_____ Terminal 227: Lists _____
1 | > data<-read.table("gene-exprA-exprB.tab", header=TRUE, sep="\t")
2 | > data[1,]
3 |      gene exprA exprB
4 | 1 all0001 1109 2202
5 | > data[1,2]+1
6 | [1] 1110
7 | >

```

Terminal 228 demonstrates how a list is created manually and how list elements can be retrieved.

```

_____ Terminal 228: Creating Lists _____
1 | > xray<-list(ID=c("1ZDB","1FDS","3EKS"), Score=c(0.32,0.89,0.20))
2 | > xray
3 |
4 | $ID
5 | [1] "1ZDB" "1FDS" "3EKS"
6 |
7 | $Score
8 | [1] 0.32 0.89 0.20
9 |
10 | > xray$ID # Vector ID
11 | [1] "1ZDB" "1FDS" "3EKS"
12 | > xray[1] # Vector ID
13 | $ID
14 | [1] "1ZDB" "1FDS" "3EKS"
15 | > xray[[1]] # Vector ID
16 | [1] "1ZDB" "1FDS" "3EKS"
17 | > xray$ID[2] # 2nd element of vector ID
18 | [1] "1FDS"
19 | > xray[[1]][2] # 2nd element of vector ID
20 | [1] "1FDS"
21 | > names(xray)
22 | [1] "ID" "Score"
23 | > attach(xray)

```

```

24 > ID                                     # Vector ID
25 [1] "1ZDB" "1FDS" "3EKS"
26 > ID[1]                                # 1st element of vector ID
27 [1] "1ZDB"
28 > >

```

Note that the construct `xray[[1]]` identifies the first vector of the list, and `xray[[1]][2]` the second element of the first vector. However, the construct `xray[1,2]` does not work for lists, but only for arrays.

Finally, a data frame is a list with named entries. Terminal 229 illustrates the difference.

```

Terminal 229: Data Frames
1 > xray
2 $ID
3 [1] "1ZDB" "1FDS" "3EKS"
4
5 $Score
6 [1] 0.32 0.89 0.20
7
8 > data.frame(xray)
9      ID Score
10 1 1ZDB  0.32
11 2 1FDS  0.89
12 3 3EKS  0.20
13 >

```

When we transfer the list to a data frame in line 8, the display of the data looks nicer. Thus, a data frame resembles a table, whereas a list consists of a collection of vectors.

17.4 Programming Structures

R is a full-fledged programming language. Thus, you could write complex programs in R. Here, I wish to restrict myself to two fundamental functions, i.e., loops and conditionals. The basic structure of a loop is

```
for(index in vector) { ... }
```

Terminal 230 shows an example where a `for` construct loops through all element indices of vector `exprA`, where the expression value fulfills a certain condition (Fig. 17.3).

```

Terminal 230: Composite Plots
1 > plot(log2(exprA), log2(exprB), col="blue", type="n")
2 > for(i in which(exprA<500)) {points(log2(exprA[i]), log2(exprB[i]), col="red")}
3 > for(i in which(exprA>5000)) {points(log2(exprA[i]), log2(exprB[i]), col="blue")}
4 > for(i in which(exprA>=500 & exprA<=5000)) {points(log2(exprA[i]),
5           log2(exprB[i]), col="green")}
6 >

```

Fig. 17.3 Composite Plot.
Result from Terminal 230

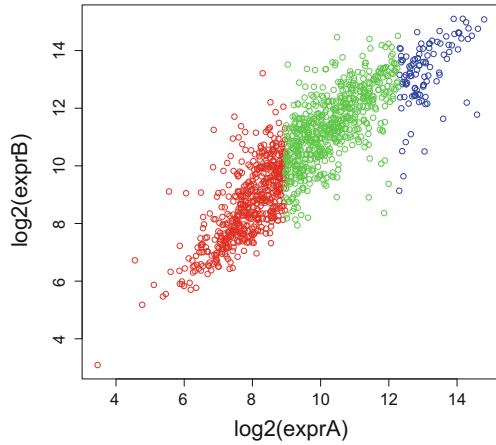
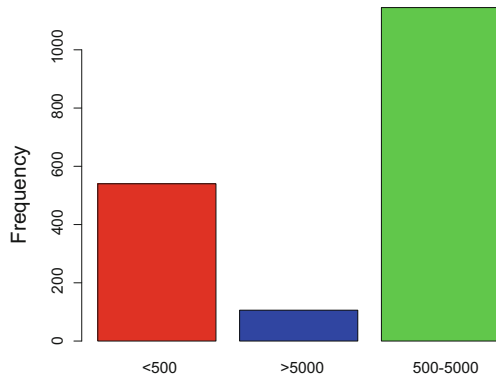


Fig. 17.4 Bar Plot. Result
from Terminal 231



Line 1 creates an empty plot because the attribute *type* is set to none. This way we get a coordinate system with the correct dimensions. The `points()` function in lines 2–5 adds data to the empty plot (see Terminal 221 on p. 318). Depending on the expression value of *exprA*, the data point is colored in red, green, or blue. The resulting plot is shown in Fig. 17.4.

Note that the `which()` condition in line 4 contains a Boolean AND. An OR would be represented by the pipe character.

Terminal 231 shows a rather complex one liner in order to count the number of genes with expression values of a particular range. It makes use of the `if` function:

```
if (condition){...} else{...}
```

The `else` part is optional.

```

Terminal 231: One Liner Creating Bar Plot
1 > a=0; b=0; c=0;
2   for(i in exprA) {
3     if(i<500){a=a+1};
4     if(i>5000){b=b+1}else{c=c+1}
5   };
6   barplot(c(a,b,c),
7   names=c("<500", ">5000", "500-5000"),
8   col=c("red", "blue", "green"),
9   yla="Frequency")

```

The result of the little R script in Terminal 231 is shown in Fig. 17.4. Note that I split the one-liner in order to make the script readable. In line 1, variables *a*, *b*, and *c* are set to zero. Line 2 initiates a loop through all values in vector *exprA*. Then we use *if* clauses to check to which range the current expression value belongs and increment or counter variables accordingly. Finally, we create a bar plot with the result. Attribute *names* adds bar names while attribute *yla* specifies the y-axis label. *col* assigns colors to the bars.

I have to admit that we could have done this easier. The command `sum(exprA<500)` returns the number of elements with values less than 500. Anyway, now you know the syntax of *for* loops and *if* conditionals.

17.5 Data Exploration

Let us work on a more complex dataset in order to illustrate the power of R. Download file *gene-exprA-exprB-len-gc.tab* from the book's website and load it into R using `read.table()`. Terminal 232 shows how.

```

Terminal 232: Getting an Overview over Dataset
1 > nostoc<-read.table("gene-exprA-exprB-len-gc.tab", header=TRUE, sep="\t")
2 > head(nostoc)
3   gene      exprA      exprB      len      gc
4 1 all0001  1109.00  2202.00  1230 508
5 2 all0002  1094.00  1373.25   738 298
6 3 all0004  1968.50  7153.50   948 454
7 4 all0005  4892.75  7768.75  1521 708
8 5 all0006 10175.20 15089.50   552 246
9 6 all0007  4971.00  6861.00   564 264
10 > summary(nostoc)
11   gene      exprA      exprB      len
12 Ampicillin: 1   Min.   : 11.0   Min.   : 8.5   Min.   : 90.0
13 all0001   : 1   1st Qu.: 283.6   1st Qu.: 451.5   1st Qu.: 444.0
14 all0002   : 1   Median : 625.5   Median : 1355.8   Median : 771.0
15 all0004   : 1   Mean    : 1684.3   Mean    : 3153.0   Mean    : 982.2
16 all0005   : 1   3rd Qu.: 1605.8   3rd Qu.: 3740.9   3rd Qu.: 1236.0
17 all0006   : 1   Max.    :28417.5   Max.    :35202.5   Max.    :14811.0
18 (Other)   :1245
19   gc
20   Min.   : 33.0
21   1st Qu.: 178.0
22   Median : 317.5
23   Mean    : 417.7
24   3rd Qu.: 540.0
25   Max.    :6178.0

```

```

26 NA's : 1.0
27 > boxplot(nostoc)
28 >

```

In line 1 we load the data. The `head()` function shows the first couple of lines of the data frame, i.e., the head of the data file. Then, in line 10, we employ `summary()` to get a first overview of the data. For each vector (corresponding to the table column) the descriptive statistics are given. For the text type vector *gene*, an example of the content is printed. For the numeric vectors we see the minimum and maximum values, the mean and median, and the data values separating the first from the second and the third from the fourth quartile. The function `boxplot(nostoc)` plots a box plot of these data as shown in Fig. 17.5.

The nice thing about a box plot (also known as box and whisker plot) is that it gives you an impression of how skewed your data are. The box plot splits the dataset into quartiles. The body of the boxplot consists of a box that goes from the first quartile (Q1) to the third quartile (Q3). The thick bar in the box represents the media. Thus, 50 % of the data lay below and 50 % above the median, and 25 % of the data lay in the bottom box, and 25 % in the top box. The whiskers extend to the minimum and maximum values unless the extreme values are too extreme. How is that defined? A standard measure is the inner quartile range (IQR). It describes the difference between the 75 % (third) quartile and the 25 % (first) quartile. It can be determined with the `iqr()` function. Now, if a data point outside the box lies even outside $1.5 \times IQR$, then it is plotted individually.

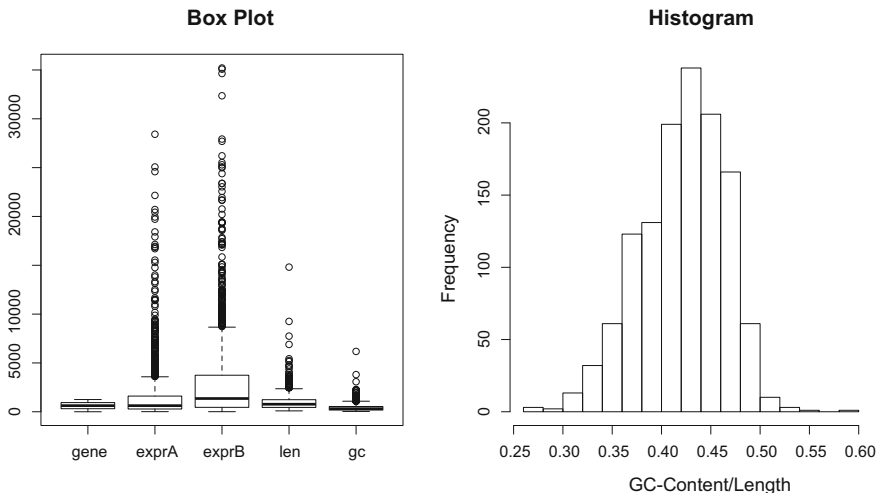


Fig. 17.5 Box Plot and Histogram. *Left* An overview of the distribution of all data in data frame *nostoc*. *Right* A histogram showing the distribution of relative GC-content of all genes

The entry “NA” in *len* and *gc* in Terminal 232 on p. 327 means that one vector element has not been assigned. Terminal 233 shows you how to find out which one.

```

1  > which(is.na(nostoc$len))
2  [1] 1109
3  > nostoc[1109,]
4      gene exprA  exprB len gc
5  1109 Ampicillin  112 118.25 NA NA
6  >

```

The function `is.na()` returns TRUE or FALSE for vector elements that contain no data. The value *NA* must not be confused with the number 0 or an empty string. *NA* simply means that there are no data, not even an empty string.

To get an overview of the distribution of the GC-content of all genes we may plot a histogram. The command

```
hist(nostoc$gc/nostoc$len,main="Histogram",xlab="GC-Content/Length")
```

plot the histogram shown in Fig. 17.5.

Finally, let us produce a matrix of scatter plots with the function `pairs()`:

```
pairs(nostoc,lower.panel=NULL)
```

This short command plots all vectors in data frame *nostoc* against each other. The result is shown in Fig. 17.6.

The empty plots that contain vector names annotate the axis. Thus, the plot in row 3, column 4 plots *len* at the *x*-axis against *exprB* at the *y*-axis.

Terminal 234 shows a way to pimp the matrix of plots by adding the Pearson correlation of all plots to the matrix. Since, the function `cor()` that calculates the Pearson correlation does not accept NAs, we need to create a data frame without non-assigned data.

```

1  > nostoc.clean<-na.omit(nostoc)
2  > panel.pearson<- function(x, y, ...) {
3      horizontal <- (par("usr")[1] + par("usr")[2]) / 2;
4      vertical <- (par("usr")[3] + par("usr")[4]) / 2;
5      text(horizontal, vertical, format(abs(cor(x,y)), digits=2))
6  }
7  > pairs(nostoc.clean,lower.panel=panel.pearson)
8  >

```

This is done in line 1 in Terminal 234 with the command `na.omit()`. The one-liner spanning from lines 2–6 defines a new function named *panel.pearson*. I will not explain all parts of the function. Just note that the correlation is calculated in line 5. Line 7 contains the modified call of the `pairs()` function. The resulting plot is shown in Fig. 17.7.

It becomes clear that there is a strong correlation (Pearson coefficient = 1) between the GC-content of a gene and its length. Wow, who would have expected this? The correlations between *gene* and all other categories are extremely small. This makes perfect sense, too, because *gene* describes categorical data (gene IDs) while all other vectors are numerical.

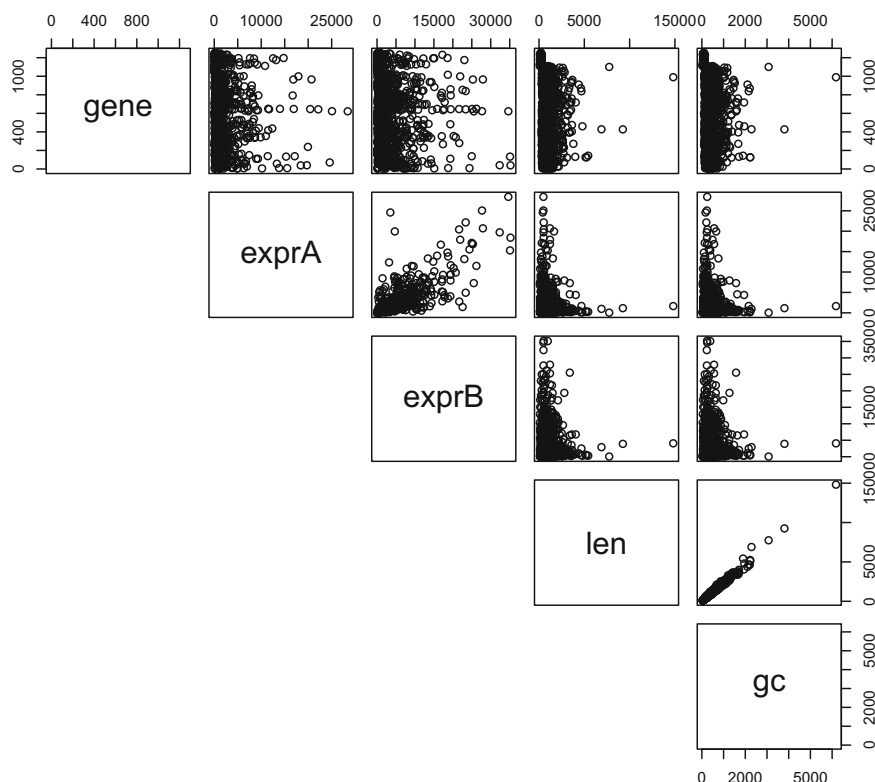


Fig. 17.6 Scatter-Plot Matrix. The `pairs()` function visualizes a plot of all data against each other

17.5.1 Saving Graphic

You probably thought already about a way to save the plots generated by R. Since, the command line provides no *Save as* button, there must be another way. What happens when you execute a command-like `plot()` that opens a graphic window? First of all, you must work in a graphical environment to have the plot opened. Otherwise, you see nothing—but can still save plots as you will learn in a minute. Every graphical output of R is displayed by an X-client (see Sect. 3.3.1 on p. 27). Thus, a data stream is sent to the X-server and from there redirected to an X-client. See the point? Data stream? Does it ring a bell? You can redirect the graphical data stream to other devices, e.g., a second X-client or a file.

The following list shows the most important commands to work with output devices in R.

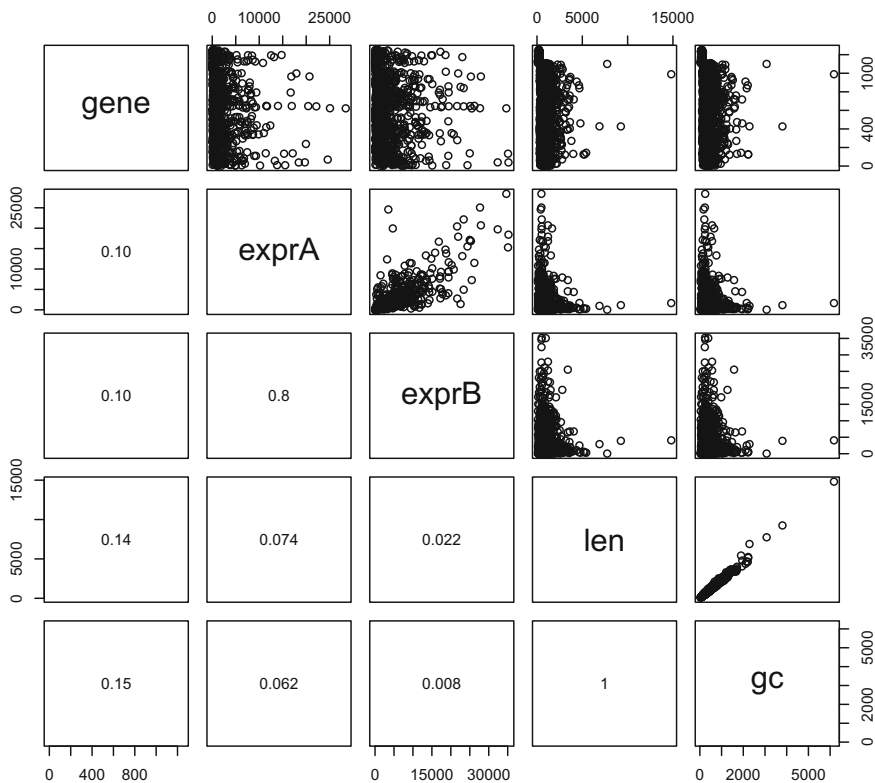


Fig. 17.7 Scatter-Plot Matrix with Pearson Correlation. The `pairs()` function visualizes a plot of all data against each other. The *lower panel* shows the Pearson correlation of the plots

```

1  dev.list()          # List all available devices
2  dev.cur()           # List active device
3  dev.set(n)          # Set n as active device
4  dev.off(n)          # Close device n
```

Table 17.1 lists a number of commands that open a specific graphical device. Some devices can only hold one plot (e.g., `png()`), other may contain several plots (e.g., `pdf()`).

Terminal 235 shows you how to play with devices.

```

1  > dev.list()          # I assume you did not plot anything yet
2  > dev.list()          # List available devices
3  NULL
4  > plot(1,1)           # Open new graphics window
5  > dev.list()
6  X11cairo             # Name of the new window
7  2                    # ID of the new window
8  > pdf("myplot.pdf")   # Opens a PDF device
```

```
9  > dev.list()
10 X11cairo      pdf
11      2        3
12 > plot(2,2)
13 > plot(3,3)
14 > dev.off(3)
15 X11cairo
16      2
17 > dev.cur()
18 X11cairo
19      2
20 > x11()
21 > dev.list()
22 X11cairo X11cairo
23      2        3
24 > dev.set(2)
25 X11cairo
26      2
27 >
```

As you can see, you can open several devices in parallel and set the active graphical device with `dev.set()`.

Table 17.1 Commands that open graphics devices that can be used by R

Command	Output format
<code>pdf()</code>	Creates PDF graphics—can have several pages
<code>png()</code>	Creates PNG bitmap
<code>x11()</code>	Opens a new X11 window—for Linux
<code>quartz()</code>	Opens a new Quartz window—for MacOSX
<code>jpeg()</code>	Creates JPEG bitmap
<code>bmp()</code>	Creates BMP bitmap
<code>tif()</code>	Creates TIFF bitmap
<code>xfig()</code>	Creates XFIG graphics file format
<code>postscript()</code>	Creates PostScript graphics
<code>pictex()</code>	Creates LaTeX/PicTeX graphics

If no filename is given, *Rplot* will be used

17.5.2 Regression

R provides a huge pallet of functions for all types of regression analyses. Here, I wish to restrict myself to show you how to apply a linear fitting model (by the least squares method) with the function `lm()` and how to smooth scatter plots using locally weighted polynomial regression with the function `lowess()`. Let us get hands-on and see how to work with regression lines in two examples.

17.5.2.1 Random(?) Numbers

Random numbers are a big issue—especially in computer languages. How can you generate random numbers with something as ordered as a computer? Random number generators are algorithms that in fact pseudo-random numbers, i.e., a flow of numbers appear random. AWK provides such a pseudo-random number generator (see Terminal 129 on p. 229 and Sect. 13.11.4 on p. 245).

Let us now check how good this generator works. In Terminal 236 we do something that I love very much: we execute a shell command from R.

```

1  > system("awk 'BEGIN{for(i=1;i<=3;i++){print rand()}}'")
2  0.840188
3  0.394383
4  0.783099
5  > system("awk 'BEGIN{for(i=1;i<=3;i++){print rand()}}'", intern=TRUE)
6  [1] "0.840188" "0.394383" "0.783099"
7  > zufall<-system("awk 'BEGIN{for(i=1;i<=50;i++){print rand()}}'", intern=TRUE)
8  > zufall<-as.numeric(zufall)
9  > x<-1:length(zufall)
10 > plot(x,zufall,xlab="",ylab="Random Number") # Plot 1
11 > abline(lm(zufall~x), col="red")
12 > lines(lowess(x,zufall), col="blue")
13 > hist(zufall,main="",xlab="Random Number") # Plot 2
14 >

```

The function `system()` in line 1 of Terminal 236 executes the AWK script. In the same way we can execute shell commands like `pwd` or `ls`. The output of the command is printed into the R shell. To make the data available to R for further processing we have to set the argument `intern=TRUE`. As you can see in line 6, the output is now printed as a vector containing text string elements. In line 8, we convert the vector into a numerical type with `as.numeric()`. Then we create vector `x` that contains as many elements as vector `zufall`. In line 10, we plot these XY-data with edited `x`- and `y`-axis labels. Line 11 adds the linear regression line. It is a combination of the plotting function `abline(a, b)` that adds a straight line to a plot, where `a` is the intercept and `b` the slope. These data are provided by the nested function `lm(y ~ x)` that fits a linear model to the data. The regression line is plotted in red as shown in Fig. 17.8.

In line 12 we add smoothed plot of the data as calculated by `lowess()`. The actual plotting is done by `lines()`. Finally, we create a histogram of the random numbers with function `hist()`. This creates a new plot.

What do we observe? If the random numbers were completely random, we should expect a horizontal regression line. The histogram of the numbers clearly shows that higher numbers are overrepresented. The take-home message is: be careful with random numbers or what is said to be so.

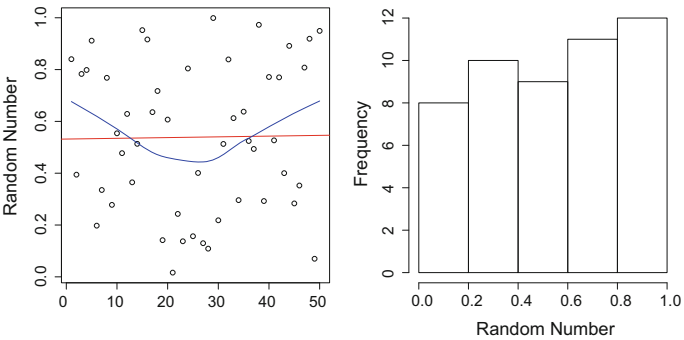


Fig. 17.8 Random Numbers? Fifty random numbers were generated with AWK as shown in Terminal 236 on the preceding page. *Left* Linear regression (red line) and LOWESS fitting (blue) of the random data. If the dataset was completely random, we should expect two horizontal lines. *Right* A histogram of the numbers clearly shows that higher numbers are overrepresented

Table 17.2 Anscombe’s Quartet (Anscombe 1973)

x123	y1	y2	y3	x4	y4
4	4.26	3.1	5.39	8	6.58
5	5.68	4.74	5.73	8	5.76
6	7.24	6.13	6.08	8	7.71
7	4.82	7.26	6.42	8	8.84
8	6.95	8.14	6.77	8	8.47
9	8.81	8.77	7.11	8	7.04
10	8.04	9.14	7.46	8	5.25
11	8.33	9.26	7.81	8	5.56
12	10.84	9.13	8.15	8	7.91
13	7.58	8.74	12.74	8	6.89
14	9.96	8.1	8.84	19	12.5

Four datasets having identical descriptive statistical properties ($\text{mean}(x)=9.0$, $\text{mean}(y) = 7.50$, $\text{variance}(x) = 11$, $\text{variance}(y) = 4.12$), yet appearing very different when plotted (Fig. 17.9). Each dataset consists of 11 XY points. X-data in column 1 belong to Y-data in columns 2–4 and X-data in column 5 belong to Y-data in column 6. In contrast to the original dataset I have ordered the data with rising x-values

17.5.2.2 Anscombe’s Quartet

The Anscombe’s Quartet is a series of four datasets that provide a useful caution against blindly applying statistical methods to data (Table 17.2). Each dataset consists of ten x - and y -values such that the mean and variance of x and y , and the correlation and regression of x versus y are the same. However, as shown in Fig. 17.9 on the facing page, the data look really different.

The quartet provides a good example of how important it is to actually look at data. This is an old hat for some, but I like the quartet for its simplicity and visual impact.

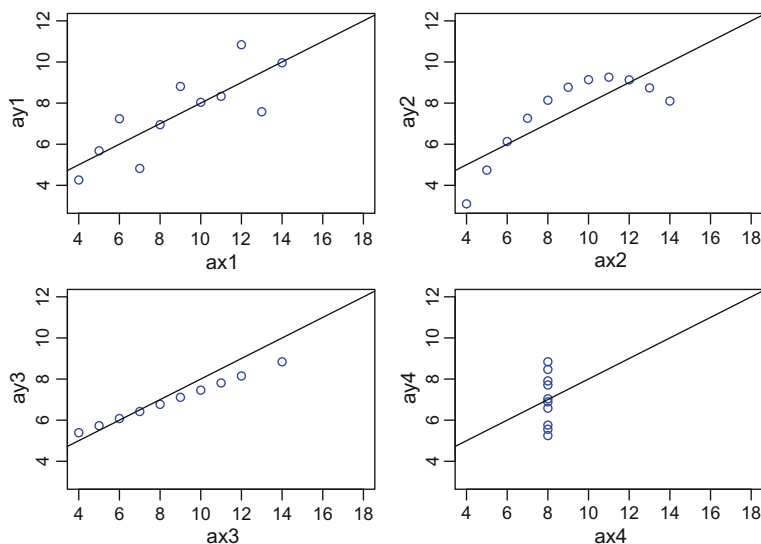


Fig. 17.9 Anscombe's Quartet. Plots of Anscombe's data from Table 17.2. Do not get fooled by a simple regression analysis. The correlation coefficient for all datasets is 0.816 and the linear regression line is always the same: $y = 3.0 \times 0.5x$

The data and graphs were first presented in 1973 (Anscombe 1973). It is quite some fun to read the paper. It ends with the following quote:

Unfortunately, most persons who have recourse to a computer for statistical analysis of data are not much interested either in computer programming or in statistical methods, being primarily concerned with their own proper business. Hence the common use of library programs and various statistical packages. Most of these originated in the pre-visual era. The user is not showered with graphical displays. He can get them only with trouble, cunning, and a fighting spirit. It is time that was changed.

The plots shown in Fig. 17.9 were plotted with the commands shown in Terminal 237.

```

1  > anscombe<-read.table(                                     +
2    "http://www.hs-mittweida.de/wuenschi/data/media/        +
3    compbiolbook/anscombe.tab",header=TRUE,sep="\t")
4  > attach(anscombe)
5  > par(mfrow=c(2,2))
6  > plot(ax1,ay1,xlim=c(4,18),ylim=c(3,12),col="blue")
7  > abline(lm(ay1~ax1))
8  > plot(ax2,ay2,xlim=c(4,18),ylim=c(3,12),col="blue")
9  > abline(lm(ay2~ax2))
10 > plot(ax3,ay3,xlim=c(4,18),ylim=c(3,12),col="blue")
11 > abline(lm(ay3~ax3))
12 > plot(ax4,ay4,xlim=c(4,18),ylim=c(3,12),col="blue")
13 > abline(lm(ay4~ax4))
14 > par(mfrow=c(1,1))
15 >

```

Several things should be noted. In line 1–3 we download the data directly from the Internet. In line 5 we change the graphics setting. As a result, four plots can be printed in one plot. With the `par()` function and the option `mfrow=c(nrows, ncols)` one can define a matrix of $nrows \times ncols$ plots. In line 14 this behavior is reset to the default setting. Note also that the parameters `xlim()` and `ylim()` are used to set the dimensions of the x - and y -axes.

17.5.3 *t*-Test

A very important statistical test method in life sciences is the t -test. The t -test is a parametric test that assumes a normal distribution of the data and a similar sample variance. It is also important that the datasets are unrelated. The t -test is used to compare two datasets, or more specifically, look for a difference between two samples.

The null hypothesis (H_0) for the t -test is that there is no difference between the mean of two samples. Well, we could simply compare the means, but—are they significantly different? The t -test provides us a probability value (p-value) that gives the probability of wrongly rejecting H_0 . Thus, the higher the p-value, the higher the chance that we are wrong in saying there is a meaningful difference between the two means. The p-value ranges between 0 and 1, corresponding to 0 and 100 % probability, respectively. It is a habit to reject H_0 when the p-value is equal to or below 0.05. We are then 95 % confident that we correctly rejected H_0 .

To exemplify how a t -test analysis is performed in R we use two datasets consisting of the GC-content (in %) of all genes in two different *E. coli* strains. Let us first check if the datasets meet the criteria.

```

1  > allk12 <- read.table("all-k12-id-gc.tab", header=FALSE, sep="\t")
2  > allo157 <- read.table("all-o157-id-gc.tab", header=FALSE, sep="\t")
3  > hist(ecolik12[[2]],main="E. coli K12",xlab="GC-Content [%]")
4  > hist(ecolio157[[2]],main="E. coli O157:H7",xlab="GC-Content [%]")
5  > sd(ecolik12[[2]])
6  [1] 5.125811
7  > sd(ecolio157[[2]])
8  [1] 6.01197
9  > t.test(allk12[[2]],allo157[[2]])
10
11      Welch Two Sample t-test
12
13 data:  allk12[[2]] and allo157[[2]]
14 t = 5.4471, df = 9983.783, p-value = 5.241e-08
15 alternative hypothesis: true difference in means is not equal to 0
16 95 percent confidence interval:
17  0.3881420 0.8245378
18 sample estimates:
19 mean of x mean of y
20  50.67152  50.06518
21
22 >

```

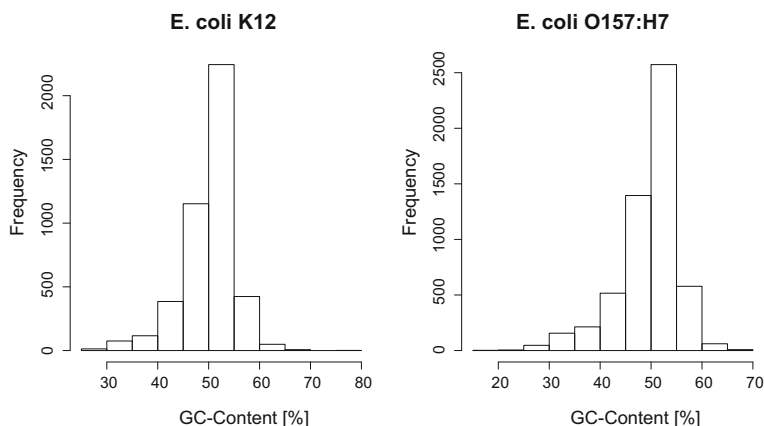



Fig. 17.10 Histograms of Gene GC-Content. The distribution of GC-content of all genes from *E. coli* strain K12 and *E. coli* strain O157:H7 are plotted

In Terminal 238 we first load the datasets (you can download them from the book's webpage) and create the histograms (lines 3–4) shown in Fig. 17.10. The distribution looks like a normal Gaussian distribution—good.

We then use the `sd()` function to calculate the standard deviation (lines 5 and 7). These data look good as well. In line 9 in Terminal 238 we execute the `t.test()` function. As a result we obtain a p-value far below 0.05 ($5.24E^{-08}$). Thus, we can securely reject the null hypothesis that there is no difference between the means and state: the means of the GC-content of genes from *E. coli* strain K12 and *E. coli* strain O157:H7 are significantly different.

17.5.4 Chi-Square Test

Another very important statistical test method in life sciences is the chi-square test (χ^2 -test). The chi-square test is a non-parametric test that assesses whether the difference between observed and expected frequencies are due to chance alone or to something meaningful. Expected frequencies are those you would expect to see according to the null hypothesis H_0 . It is important to note that the chi-square test can only be used with frequencies, not percentages, scale data, or such things. And what are frequencies? Well, counts of items are in different categories.

Typically, H_0 states that two data populations are homogeneous, i.e., that there is no difference in the frequencies. As with the t-test, the chi-square test returns the probability (in form of the p-value) of wrongly rejecting the null hypothesis. Again, the higher the p-value, the higher the chance that we are wrong in saying there is a meaningful difference between two frequency sets. And once again: the p-value

Table 17.3 Genomic backbone and Islands

	Backbone	Island
Adenine	994360	702
Thymine	988925	578
Guanine	1132739	936
Cytosine	1010405	322

ranges between 0 and 1, corresponding to 0 and 100 % probability, respectively. We reject H_0 when the p-value is equal to or below 0.05.

To exemplify the application of the chi-square test we use a dataset that compares the frequencies of nucleotides in two different sets of genes. Both sets are derived from the same organism but one set describes the genomic backbone while the other describes genomic islands. What the heck is this? The backbone is a set of genes that is common among bacteria of one group while the island genes are specific to a certain strain. The dataset is summarized in Table 17.3.

Terminal 239 shows how these data are organized in R. The function `chisq.test()` that actually performs the analysis requires the data as a matrix.

```

1  > matrix(c(994360,988925,1132739,1010405,702,578,936,322), ncol=2)
2      [,1] [,2]
3  [1,] 994360 702
4  [2,] 988925 578
5  [3,] 1132739 936
6  [4,] 1010405 322
7  > chisq.test(matrix(c(994360,988925,1132739,1010405,702,578,936,322), ncol=2))
8
9      Pearson's Chi-squared test
10
11 data:  matrix(c(994360, 988925, 1132739, 1010405,
12                702, 578, 936, 322),ncol = 2)
13 X-squared = 241.2361, df = 3, p-value < 2.2e-16
14 >

```

Line 1 shows how the nucleotide frequencies shown in Table 17.3 on the previous page are transferred to a matrix. In line 7 the actual analysis is executed. As a result we obtain a p-value far below 0.05 ($2.2E^{-16}$). Thus, we can securely reject the null hypothesis that there is no difference between both frequency sets. This means that the nucleotide composition of the backbone is significantly different from the nucleotide composition of the island—a sign of horizontally acquired genes.

17.6 R Scripts

As said and shown above, R is a programming language. Thus, like with Bash, AWK, and MySQL, commands can be written into a file, loaded into R, and get executed. My example requires that both the data file *gene-exprA-exprB.tab* (see File 10 on

p. 320) and the R script *import-export.r* (Program 62) are in the same directory from which you start R.

```
----- Program 62: import-export.r - R Script -----
1 # save as import-export.r
2 data<-read.table("gene-exprA-exprB.tab",header=TRUE,sep="\t")
3 attach(data)
4 png("mioploto.png")
5 plot(log2(exprA),log2(exprB),col="blue")
6 dev.off()
```

The magic command to execute in the Bash command line is:

```
R -- slave --vanilla < script.r > /dev/null
```

This will redirect the R commands stored in *script.r* to R and redirect any output to the nirwana device NULL (*/dev/null*). The command line option *-slave* and *-vanilla* instructs R to work in silence.

17.7 Extensions: Installing and Using Packages

R's functionality and capability can be extended with the help of packages. As of September 2012, the Comprehensive R Archive Network (CRAN) package repository features 4049 available packages. One of the newest addition is the Genetics ToolboX (*gtx*), which provides tools for genetic association analyses. There are also graphical interfaces available. It makes no sense to talk too much about available packages but have a look: <http://www.r-project.org>. To exemplify the installation and application of a package I choose *RMySQL*. This enables you to connect to MySQL databases from R.

17.7.1 Connect to MySQL

To connect to a MySQL server you need to load a specific library. Depending on the Linux system you are working with, this might already be installed. If you execute `library("RMySQL")` in the R command shell and receive no error message but *DONE (RMySQL)*—lucky you. In the more likely case that you receive an error message, you have to install the package by executing

```
install.packages("RMySQL")
```

from the R command line. This will install the package. During the installation you are requested to specify a CRAN mirror server as shown in Fig. 17.11.

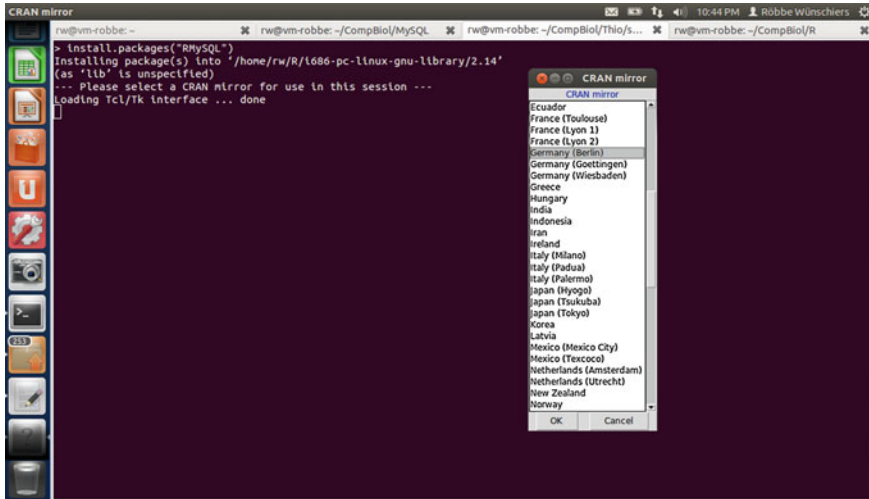


Fig. 17.11 Installation of RMySQL. During the installation you are asked to choose a CRAN mirror server

Now, if you still receive an error like “*installation of package 'RMySQL' had nonzero exit status*”, then the installation failed. Most likely, some MySQL libraries and header files are missing. These contain information that are not needed to run MySQL but for the installation of software that uses MySQL. To install these, you must have administrator rights to the system and execute the following command from the Bash command line.

```
sudo apt-get install libmysqlclient-dev
```

After successful installation of the MySQL client libraries and header files you can execute `install.packages("RMySQL")` once again.

In Terminal 240 we use the newly installed library *RMySQL* and connect to the database *compbiol* we set up in Sect. 16.2.4 on p. 301. We also need the MySQL tables created in Sects. 16.3.3 on p. 304 and 16.3.4.

```

Terminal 240: Connecting to MySQL Database
1  > library("RMySQL")
2  Loading required package: DBI
3  > con <- dbConnect(MySQL(), user="awkologist", password="awkology",
4                      dbname="compbiol", host="localhost")
5  > dbListTables(con)
6  [1] "annotation" "expression"
7  > dbGetQuery(con, "SELECT * FROM expression LIMIT 3")
8      gene expr_value
9  1 alr1207      1000
10 2 alr2938     10323
11 3 alr3395     1432
12 > expression<-dbGetQuery(con, "SELECT * FROM expression")
13 > names(expression)

```

```

14 | [1] "gene"          "expr_value"
15 | > attach(expression)
16 | > mean(expr_value)
17 | [1] 3984.571
18 | > median(expr_value)
19 | [1] 1432
20 | >

```

Prerequisite to connecting to a MySQL server is loading the library *RMySQL* to the current R session. This is done with the `library()` function as shown in line 1 in Terminal 240 on the previous page. In lines 3–4 we establish the database connection. The following commands should be self-explanatory—you are no rookie anymore.

17.7.2 Life-Science Packages

As said, there are hundreds and hundreds of R packages and there are good books available that describe them. Even more documentation can be found on the Internet. The following names list just some packages that might be interesting for you. These packages can be downloaded from the Comprehensive R Archive Network (CRAN) and installed with the `install.packages("...")` function.

<i>nlstools</i>	Bacterial growth and enzyme kinetics
<i>ape</i>	analyses of phylogenetics and evolution
<i>seqRFLP</i>	Simulation and visualization of restriction enzyme cutting pattern from DNA sequences
<i>HardyWeinberg</i>	Graphical tests for Hardy-Weinberg equilibrium (HWE) based on the ternary plot (de Finetti diagram)
<i>Bioconductor</i>	Analysis and processing of DNA-microarray data. For installation see http://www. bioconductor.org
<i>seqinR</i>	Biological Sequences Retrieval and Analysis

I say it over and over again: take a use case of your own, seek for solutions to problems that are close to yours, and adopt them to your needs. That is the way to learn AWK, MySQL, R, data processing, programming, anything.

Part VI

Worked Examples

Chapter 18

Genomic Analysis of the Pathogenicity Factors from *E. coli* Strain O157:H7 and EHEC Strain O104:H4

This guided exercise will show you how to use BLAST to compare two genomes. The objective is to find open reading frames (ORFs) that are unique to one genome.

18.1 The Project

The well-known enterobacterium *Escherichia coli* exists in many different strains. Some of these are pathogenic, some are not. These strains can be distinguished and serologically classified by analyzing two different bacterial proteins, i.e. a cell wall lipopolysaccharide antigen (O) and a flagella protein (H). *Escherichia coli* O157:H7 is an emerging cause of food-borne illness (Cohen and Giannella 1992). This strain was first detected in 1982 as the causative agent of an outbreak of a disease attributed to undercooked hamburgers (Riley et al. 1983). An estimated 73,000 cases of infection and 60 deaths occur in the United States each year. Infection often leads to bloody diarrhea, and abdominal cramps. Up to 30 % of people infected with *E. coli* O157:H7 can develop kidney failure and 3–5 % of these people die. Most infections have been associated with eating undercooked, contaminated ground beef. Person-to-person contact in families and child-care centers are also an important mode of transmission. Infection can also occur after drinking raw milk and after swimming in or drinking sewage-contaminated water.

Based on phylogenetic analyzes *E. coli* O157:H7 and K12 separated from a common ancestor as long as 4.5 million years ago (Reid et al. 2000). During this time the former must have had acquired its pathogenicity, probably by horizontal gene transfer.

Given the fully sequenced and annotated genome sequences, how can we find out which proteins might be the reason for pathogenicity? In this particular project you will computationally compare the translated, annotated genomes of the non-pathogenic strain *E. coli* K12 (Blattner et al. 1997), and the pathogenic strain *E. coli* O157:H7 (Perna et al. 2011). How can that be done?

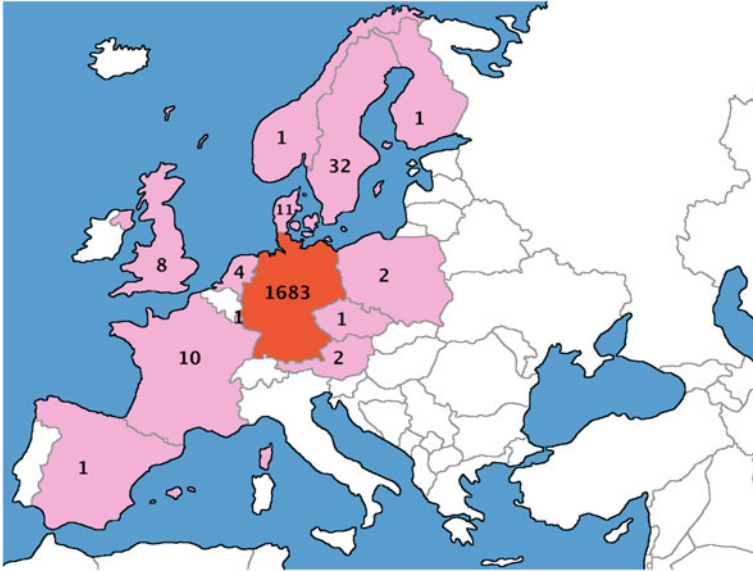


Fig. 18.1 EHEC Pandemic 2011. This map shows the number of infections as of 7 June 2011 in Europe. The outbreak originated from northern Germany. Hospitals recorded the first cases on 1 May, not being aware of the origin of symptoms observed in patients. It was not until 25 May that the rare *E. coli* strain O104:H4 was identified as the causing agent. In retrospective, 2,987 people were infected in Germany. By 26 July 2011 50 people died of EHEC in Germany. Compared to this, only 8 people in total died between 2000 and 2010

In May 2011 another pathogenic strain, namely *E. coli* O104:H4, dominated the daily news in Germany and later in whole Europe (see Fig. 18.1) (Mellmann et al. 2011). As O157:H7, it is an enterohemorrhagic strain (EH) that causes bloody diarrhea and colitis. The release of a Shiga-like toxin named verotoxin may further cause the hemolytic-uremic syndrome (HUS). The infection with *E. coli* O104:H4 was found to be much more virulent than an infection with *E. coli* O157:H7. What could be the reason? What is the genetic difference between these strains? We will use the genome sequence that has been published by the Beijing Genomics Institute shortly after the causing strains had been isolated (ftp://ftp.genomics.org.cn/pub/Ecoli_TY-2482/).

18.2 Bioinformatic Tools and Resources

The most important tool we use during this exercise is BLAST+ (Basic Local Alignment Search Tool) (Altschul et al. 1990, 1997). You might know it from previous work (see Fig. 18.2); however, I am quite sure that you never run it on your local computer.

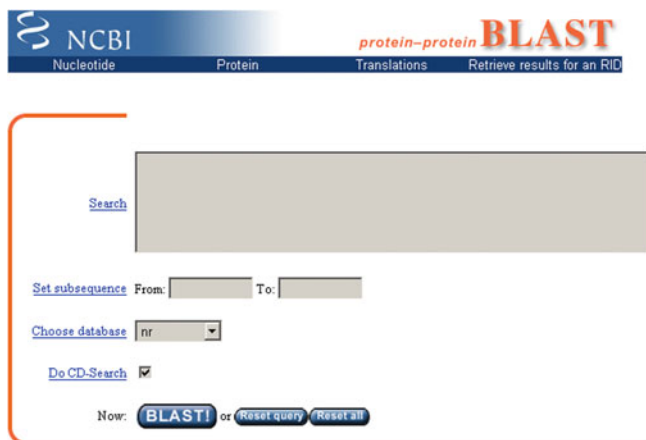


Fig. 18.2 BLAST Server. BLAST as you might have used it before

18.2.1 BLAST+

BLAST+ is a standard program to find similarities between sequences or sets of sequences (Altschul et al. 1990, 1997). It is available for free from the National Center for Biological Information (NCBI). Most people run this program via a web interface (see Fig. 18.2). Here will download, install, and run BLAST+ from the command line on your local computer. BLAST+ is rather a new and improved suite of BLAST tools (Camacho et al. 2009). More information on BLAST can be found at <http://www.ncbi.nlm.nih.gov/books/NBK1762/>. If you like to shortcut the information, take a look at Sect. 4.1.2.5 on p. 43. But be warned: you learn a lot by going the long way!

18.2.2 Genome Databases

There are zillions of biologically relevant databases out there in the web space. Here, we are going to use the NCBI Genome database (Fig. 18.3) and the Pathosystems Resource Integration Center (PATRIC, Fig. 18.4). The former can be accessed at <http://www.ncbi.nlm.nih.gov/genome> and contains the genomes sequences for the nonpathogenic laboratory strain *E. coli* K12 substrain MG1655 and the pathogenic strain *E. coli* O157:H7 substrain EDL933.

The annotated genomic sequence of *E. coli* strain O104:H4 substrain TY-2482 can be download from the Pathosystems Resource Integration Center (PATRIC, see Fig. 18.4) accessible at <http://patric.vbi.vt.edu>. This database integrates information on pathogens, provide key resources and tools to scientists, and help researchers to analyze genomic, proteomic and other data arising from infectious disease research.

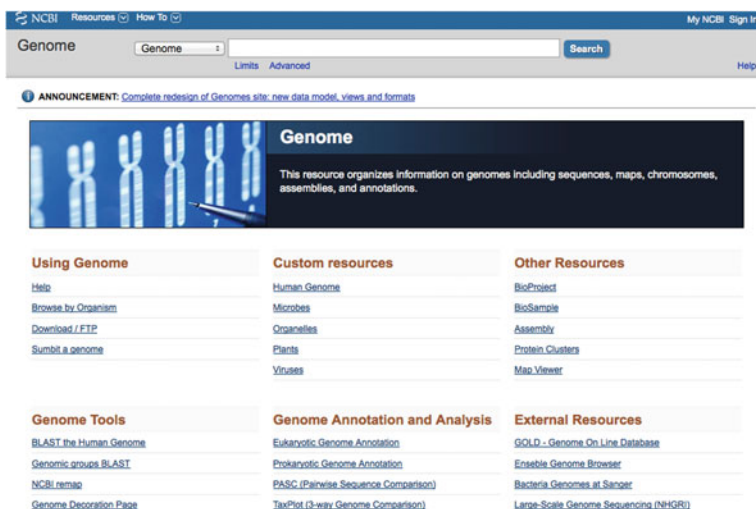


Fig. 18.3 NCBI Genome Database



Fig. 18.4 PATRIC. Pathosystems Resource Integration Center

18.3 Detailed Instructions

This section gives you in-detail instructions on how to proceed.

18.3.1 Downloading and Installing BLAST+

BLAST+ is provided for free from the National Center for Biotechnology Information that is part of the National Institute of Health, USA (NCBI; <http://www.ncbi.nlm.nih.gov>).

18.3.1.1 Download Using the Web Browser

In order to download BLAST, open a web browser (like Konqueror or Mozilla) and enter the following URL <http://www.ncbi.nlm.nih.gov> to get to the NCBI homepage. Then access the FTP server by clicking on *Downloads: Get NCBI data or software*. You find this item under the headline *Get Started*. Click *BLAST (Stand-alone)* and follow the link to the FTP archive with the LATEST version. If this is not version 2.2.25 then go up one directory and enter directory 2.2.25. You should see something like in Fig. 18.5.

Download the compressed archive (.tar.gz) suitable for your operating system into your home directory, e.g., *ncbi-blast-2.2.25+-ia32-linux.tar.gz* or *ncbi-blast-2.2.25+-universal-macosx.tar.gz* for MacOSX users. After completed download of the installation file you can proceed to the installation procedure.

Index von <ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/2.2.25/>

 In den übergeordneten Ordner wechseln

Name	Größe	Zuletzt verändert
 ChangeLog	8 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-3.i686.rpm	113246 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-3.i686.rpm.md5	1 KB	09/09/2011 6:59:00 PM
 ncbi-blast-2.2.25+-3.src.rpm	9673 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-3.src.rpm.md5	1 KB	09/09/2011 6:59:00 PM
 ncbi-blast-2.2.25+-3.x86_64.rpm	100134 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-3.x86_64.rpm.md5	1 KB	09/09/2011 6:59:00 PM
 ncbi-blast-2.2.25+-ia32-linux.tar.gz	129327 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-ia32-linux.tar.gz.md5	1 KB	09/09/2011 7:00:00 PM
 ncbi-blast-2.2.25+-ia32-win32.tar.gz	40067 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-ia32-win32.tar.gz.md5	1 KB	09/09/2011 7:00:00 PM
 ncbi-blast-2.2.25+-sparc64-solaris.tar.gz	130281 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-sparc64-solaris.tar.gz.md5	1 KB	09/09/2011 7:00:00 PM
 ncbi-blast-2.2.25+-src.tar.gz	11644 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-src.tar.gz.md5	1 KB	09/09/2011 7:00:00 PM
 ncbi-blast-2.2.25+-src.zip	14443 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-src.zip.md5	1 KB	09/09/2011 7:00:00 PM
 ncbi-blast-2.2.25+-universal-macosx.tar.gz	144461 KB	31/03/2011 12:00:00 AM
 ncbi-blast-2.2.25+-universal-macosx.tar.gz.md5	1 KB	09/09/2011 7:00:00 PM

Fig. 18.5 BLAST+. Downloading BLAST+ version 2.2.25

18.3.1.2 Installation

I assume that you have downloaded the file *ncbi-blast-2.2.25+-ia32-linux.tar.gz* to your home directory. The BLAST+ program comes compressed and packed. This is visible from the file extensions *.tar.gz*. The extension *.tar* means that several files have been combined to one single archive file using the *tar* command. The extension *.gz* indicates that this archive has been compressed using the program *gzip*. Terminal 241 shows you how to recover the original files. ATTENTION: If your disk space (disk quota) is limited, e.g., because you work on a University server, you can extract the archive only partially. In that case use:

```
tar -xf ncbi-blast-2.2.25+-ia32-linux.tar +
      ncbi-blast-2.2.25+/bin/makeblastdb +
      ncbi-blast-2.2.25+/bin/blastp +
      ncbi-blast-2.2.25+/bin/blastdbcmd
```

which includes all programs of the BLAST+ suite necessary for this chapter.

```

1  _____ Terminal 241: Unpacking BLAST+ _____
2  $ gunzip ncbi-blast-2.2.25+-ia32-linux.tar.gz
3  $ tar -xf ncbi-blast-2.2.25+-ia32-linux.tar           % SEE ATTENTION IN TEXT
4  $ cd ncbi-blast-2.2.25+
5  $ ls
6  bin ChangeLog doc LICENSE ncbi_package_info README
7  $ ls bin
8  blastdb_aliastool  blastp          makeblastdb      segmasker
9  blastdbcheck      blastx          makembindex      tblastn
10 blastdbcmd        convert2blastmask  psiblast        tblastx
11 blast_formatter   dustmasker      rpsblast        update_blastdb.pl
12 blastn           legacy_blast.pl  rpstblastn      windowmasker
13 $ ls -l bin
14 insgesamt 444532
15 -rwxr-xr-x 1 wuensch personal 19317052 2011-03-21 17:38 blastdb_aliastool
16 -rwxr-xr-x 1 wuensch personal 19041884 2011-03-21 17:38 blastdbcheck
17 -rwxr-xr-x 1 wuensch personal 26984780 2011-03-21 17:38 blastdbcmd
18 -rwxr-xr-x 1 wuensch personal 30067404 2011-03-21 17:38 blast_formatter
19 -rwxr-xr-x 1 wuensch personal 30060716 2011-03-21 17:38 blastn
20 -rwxr-xr-x 1 wuensch personal 30059244 2011-03-21 17:38 blastp
21 -rwxr-xr-x 1 wuensch personal 30058924 2011-03-21 17:38 blastx
22 -rwxr-xr-x 1 wuensch personal 19139452 2011-03-21 17:38 convert2blastmask
23 -rwxr-xr-x 1 wuensch personal 19348552 2011-03-21 17:38 dustmasker
24 -rwxr-xr-x 1 wuensch personal 51345    2010-06-28 22:32 legacy_blast.pl
25 -rwxr-xr-x 1 wuensch personal 21054204 2011-03-21 17:38 makeblastdb
26 -rwxr-xr-x 1 wuensch personal 20016456 2011-03-21 17:38 makembindex
27 -rwxr-xr-x 1 wuensch personal 30138092 2011-03-21 17:38 psiblast
28 -rwxr-xr-x 1 wuensch personal 30054924 2011-03-21 17:38 rpsblast
29 -rwxr-xr-x 1 wuensch personal 30055692 2011-03-21 17:38 rpstblastn
30 -rwxr-xr-x 1 wuensch personal 19822772 2011-03-21 17:38 segmasker
31 -rwxr-xr-x 1 wuensch personal 30109548 2011-03-21 17:38 tblastn
32 -rwxr-xr-x 1 wuensch personal 30058796 2011-03-21 17:38 tblastx
33 -rwxr-xr-x 1 wuensch personal 10010    2009-07-09 15:29 update_blastdb.pl
34 -rwxr-xr-x 1 wuensch personal 19716924 2011-03-21 17:38 windowmasker
35 $ rm ../ncbi-blast-2.2.25+-ia32-linux.tar

```

The detailed list command `ls -l` in line 12 prints all files in the current folder onto the screen. All those with the file attribute “x” are executables, i.e., programs. After unpacking the binary files we remove (`rm`) the source file *ncbi-blast-2.2.25+ia32-linux.tar* because it saves us ca. 430 MB disk space. Let us check whether the installation was successful.

```

Terminal 242: Running blastp
1  $ ./bin/blastp -h
2  USAGE
3      blastp [-h] [-help] [-import_search_strategy filename]
4              [-export_search_strategy filename] [-task task_name] [-db database_name]
5              [-dbsize num_letters] [-gilist filename] [-seqidlist filename]
6              [-negative_gilist filename] [-entrez_query entrez_query]
7              [-db_soft_mask filtering_algorithm] [-db_hard_mask filtering_algorithm]
8              [-subject subject_input_file] [-subject_loc range] [-query input_file]
9              [-out output_file] [-evaluate evaluate] [-word_size int_value]
10             [-gapopen open_penalty] [-gapextend extend_penalty]
11             [-xdrop_ungap float_value] [-xdrop_gap float_value]
12             [-xdrop_gap_final float_value] [-searchsp int_value] [-seg SEG_options]
13             [-soft_masking soft_masking] [-matrix matrix_name]
14             [-threshold float_value] [-culling_limit int_value]
15             [-best_hit_overhang float_value] [-best_hit_score_edge float_value]
16             [-window_size int_value] [-lcase_masking] [-query_loc range]
17             [-parse_deflines] [-outfmt format] [-show_gis]
18             [-num_descriptions int_value] [-num_alignments int_value] [-html]
19             [-max_target_seqs num_sequences] [-num_threads int_value] [-ungapped]
20             [-remote] [-comp_based_stats compo] [-use_sw_tback] [-version]
21
22  DESCRIPTION
23      Protein-Protein BLAST 2.2.25+
24
25  Use '-help' to print detailed descriptions of command line arguments
26  $

```

In Terminal 242, we run the program `blastp` with the option `h` for help. Obviously, we get many lines of output—it works!

18.3.2 Downloading the Proteomes

We are now going to download two proteomes from the FTP-server of the National Center for Biotechnology Information, USA (NCBI; <http://www.ncbi.nlm.nih.gov>). Go to your home directory `cd ~` and use the command `wget` to retrieve the file as shown in Terminal 243 (see also Sect. 6.1.1 on p. 71).

```

Terminal 243: Downloading Genomes from NCBI
1  $ wget "ftp://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_K_12_
2  substr_MG1655_uid57779/NC_000913.faa"
3  --2011-12-05 22:04:00--
4  ftp://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_K_12_substr_...
5  => 'NC_000913.faa.1'
6  Resolving ftp.ncbi.nlm.nih.gov... 130.14.250.10

```

```

7 | Connecting to ftp.ncbi.nlm.nih.gov|130.14.250.10|:21... connected.
8 | Logging in as anonymous ... Logged in!
9 | ==> SYST ... done.      ==> PWD ... done.
10 | ==> TYPE I ... done.    ==> CWD (1) /genomes/Bacteria/Escherichia_coli_K12...
11 | ==> SIZE NC_000913.faa ... 1815874
12 | ==> PASV ... done.     ==> RETR NC_000913.faa ... done.
13 | Length: 1815874 (1.7M) (unauthoritative)
14 |
15 | 100%[=====] 1,815,874    901K/s   in 2.0s
16 |
17 | 2011-12-05 22:04:06 (901 KB/s) - 'NC_000913.faa.1' saved [1815874]
18 | $

```

Do the same for the pathogenic *E. coli* O157:H7 from ftp://ftp.ncbi.nlm.nih.gov/genomes/Bacteria/Escherichia_coli_O157_H7_EDL933_uid57831/NC_002655.faa and save it in the same directory. Now let us change the filenames to something more convenient, namely *ecoli-k12.faa* and *ecoli-h7.faa*.

```

_____ Terminal 244: Renaming Proteome Files _____
1 | $ mv NC_000913.faa ecoli-k12.faa
2 | $ mv NC_002655.faa ecoli-h7.faa
3 | $

```

Finally, we are downloading *E. coli* strain O104:H4 from the FTP-server of the Pathosystems Resource Integration Center, USA (PATRIC, <http://www.patricbrc.org>) and rename the file to *ecoli-h4.faa*.

```

_____ Terminal 245: Downloading Genomes from NCBI _____
1 | $ wget "http://brcdownloads.vbi.vt.edu/patric2/genomes/Escherichia_coli_TY-2482/ +
2 |   Escherichia_coli_TY-2482.PATRIC.faa"
3 | --2011-12-05 22:31:54--
4 | http://brcdownloads.vbi.vt.edu/patric2/genomes/Escherichia_coli_TY-2482/E...
5 | Resolving brcdownloads.vbi.vt.edu... 198.82.161.176
6 | Connecting to brcdownloads.vbi.vt.edu|198.82.161.176|:80... connected.
7 | HTTP request sent, awaiting response... 200 OK
8 | Length: 2277906 (2.2M) [text/plain]
9 | Saving to: 'Escherichia_coli_TY-2482.PATRIC.faa'
10 |
11 | 100%[=====] 2,277,906    1.13M/s   in 1.9s
12 |
13 | 2011-12-05 22:31:56 (1.13 MB/s) - 'Escherichia_coli_TY-2482.PATRIC.faa'
14 | saved [2277906/2277906]
15 |
16 | $ mv Escherichia_coli_TY-2482.PATRIC.faa ecoli-h4.faa
17 | $

```

You are done!

Let us take a quick look that how many proteins are there in each file. Since they are saved in FASTA format, we only need to search for the number of “>” characters in each file.

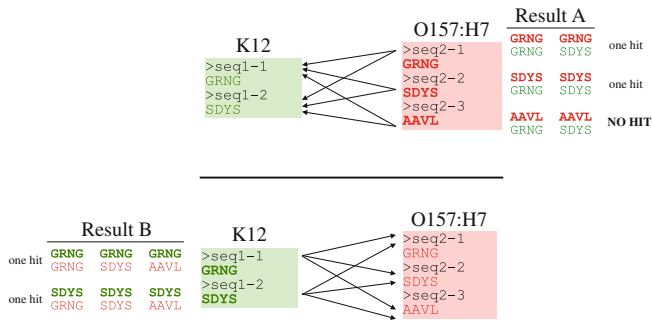


Fig. 18.6 BLAST Direction Matters. Illustrated are the proteomes of *E. coli* K12 and *E. coli* O157:H7 with two and three proteins, respectively. When we blast *E. coli* O157:H7 against *E. coli* K12 (top), then we get no hits (here 100% identity matches) for sequences that are unique to the pathogenic *E. coli* O157:H7. Blasting *E. coli* K12 against *E. coli* O157:H7 (bottom), however, reveals only proteins unique to the laboratory strain *E. coli* K12—this is not what we want

```
Terminal 246: Counting ">"
1 $ grep -c ">" ecoli*faa
2 ecoli-h4.faa:5414
3 ecoli-h7.faa:5298
4 ecoli-k12.faa:4146
5 $
```

Obviously, *E. coli* O157:H7 has an additional app. 1,000 protein compared to *E. coli* K12, while the EHEC strain *E. coli* O104:H4 has even more protein coding sequences.

18.3.3 Comparing the Genomes

As said before, you will use BLAST+ in order to computationally compare the translated, annotated genomes from *E. coli* O157:H7, and *E. coli* K12. This requires two basic steps: the set-up of a database file for one genome, and the query itself. But—for which *E. coli* should we create a database? Figure 18.6 illustrates that it makes a huge difference if we blast *E. coli* K12 against *E. coli* O157:H7 or vice versa.

18.3.3.1 BLAST Database Set Up

I here assume that you successfully installed BLAST+ in your home directory and downloaded the genomes of *E. coli* O157:H7 and *E. coli* K12 into the same folder. We will set up the database file for *E. coli* K12. The command to use is `makeblastdb` from the BLAST+ package.

```

Terminal 247: Creating BLAST+ Database
1 $ ls
2 ecoli-h4.faa  ecoli-h7.faa  ecoli-k12.faa  ncbi-blast-2.2.25+
3 $ ./ncbi-blast-2.2.25+/bin/makeblastdb -in ecoli-k12.faa -dbtype prot          +
4   -title "Escherichia coli K12" -out ecolik12 -parse_seqids
5
6 Building a new DB, current time: 12/05/2011 22:58:11
7 New DB name:      ecolik12
8 New DB title:     Escherichia coli K12
9 Sequence type:    Protein
10 Keep Linkouts:   T
11 Keep MBits:      T
12 Maximum file size: 1073741824B
13 Adding sequences from FASTA; added 4146 sequences in 0.211531 seconds.
14 $ ls -l
15 total 8164
16 -rw-r--r-- 1 wuensch personal 2277906 Nov 22 20:11 ecoli-h4.faa
17 -rw-r--r-- 1 wuensch personal 2160570 Dec  5 22:18 ecoli-h7.faa
18 -rw-r--r-- 1 wuensch personal 1815874 Dec  5 21:52 ecoli-k12.faa
19 -rw-r--r-- 1 wuensch personal  639586 Dec 14 20:48 ecolik12.phr
20 -rw-r--r-- 1 wuensch personal  33256 Dec 14 20:48 ecolik12.pin
21 -rw-r--r-- 1 wuensch personal  33168 Dec 14 20:48 ecolik12.pnd
22 -rw-r--r-- 1 wuensch personal   180 Dec 14 20:48 ecolik12.pni
23 -rw-r--r-- 1 wuensch personal  16616 Dec 14 20:48 ecolik12.pog
24 -rw-r--r-- 1 wuensch personal 130764 Dec 14 20:48 ecolik12.psd
25 -rw-r--r-- 1 wuensch personal  3005 Dec 14 20:48 ecolik12.psi
26 -rw-r--r-- 1 wuensch personal 1317125 Dec 14 20:48 ecolik12.psq
27 drwxr-xr-x 4 wuensch personal    93 Nov 29 09:21 ncbi-blast-2.2.25+
28 $

```

Terminal 247 on the previous page shows the execution of `makeblastdb` with all options we need. Note that lines 3 and 4 are actually one line. I will come to these in a minute. Note also that you have to precede the command with “./” because `makeblastdb` is not a system command. After successful execution of the command eight new files should show up in your active folder. These are database files readable by BLAST+. You do not have to care about these files at all because we named the database *ecolik12*. This name you should remember. Now I come back to the options we used for the execution of `makeblastdb` in Terminal 247 on the preceding page.

- `in` The option `-in ecoli-k12.faa` specifies the input file in FASTA format we would like to build the database from.
- `dbtype` With `-dbtype prot` we tell `makeblastdb` that our sequences are protein and not nucleotide sequences.
- `title` This option sets the title for the BLAST result file.
- `out` Finally, we use the option `-out ecolik12` to name the database and its accompanying files.
- `parse_seqids` Generates an extra database that makes it possible to retrieve individual sequences from the BLAST database by their gene identifier.

The next step is to perform the query.

18.3.3.2 BLAST Query

Now we come to the most important step in the whole procedure. Depending on your computer this step might take some 30 min. Basically, we test for each single protein from *E. coli* O157:H7 whether it can be found in the genome of *E. coli* K12. Luckily, we do not have to perform this query, but BLAST does it for us. The program we use is called `blastp`.

```

Terminal 248: Running Query
1  $ time ./ncbi-blast-2.2.25+/bin/blastp -db ecolik12 -query ecolik-h7.faa
2  -out h7vsk12.txt -evalue .00001
3  Selenocysteine (U) at position 140 replaced by X
4  ...
5  Selenocysteine (U) at position 196 replaced by X
6
7  real    2m37.050s
8  user    2m35.102s
9  sys     0m0.820s
10 $ ls -lh ecolik* h7*
11 -rw-r--r-- 1 wuensch personal 2.2M Nov 22 20:11 ecolik-h4.faa
12 -rw-r--r-- 1 wuensch personal 2.1M Dec 5 22:18 ecolik-h7.faa
13 -rw-r--r-- 1 wuensch personal 1.8M Dec 5 21:52 ecolik-k12.faa
14 -rw-r--r-- 1 wuensch personal 625K Dec 14 20:48 ecolik12.phr
15 -rw-r--r-- 1 wuensch personal 33K Dec 14 20:48 ecolik12.pin
16 -rw-r--r-- 1 wuensch personal 33K Dec 14 20:48 ecolik12.pnd
17 -rw-r--r-- 1 wuensch personal 180 Dec 14 20:48 ecolik12.pni
18 -rw-r--r-- 1 wuensch personal 17K Dec 14 20:48 ecolik12.pog
19 -rw-r--r-- 1 wuensch personal 128K Dec 14 20:48 ecolik12.psd
20 -rw-r--r-- 1 wuensch personal 3.0K Dec 14 20:48 ecolik12.psi
21 -rw-r--r-- 1 wuensch personal 1.3M Dec 14 20:48 ecolik12.psq
22 -rw-r--r-- 1 wuensch personal 31M Dec 5 23:08 h7vsk12.txt
23 $ wc -l h7vsk12.txt
24 707447 h7vsk12.txt
25 $

```

Note that in Terminal 248 lines 1 and 2 are actually one line—I only split the line in order to increase readability. The result of our query will be saved in the file `h7vsk12.txt`. This is rather a large file with 7,07,447 lines and 31 Megabyte (`ls -lh` returns a long (l) file listing with human readable (h) file sizes. We called `blastp` with a number of options.

- `db` With `-db ecolik12` we tell `blastall` to query against the database named *ecolik12*.
- `query` With `-query ecolik-h7.faa` we specify the FASTA format input file containing the protein sequences we like to query against the database.
- `out` In order to define an output file which later holds our results we employ `-out h7vsk12.txt`.
- `evalue` Finally, we use `.00001` to limit the number of hits we wish to display for each protein. The number corresponds to the *expectation threshold* or *expectation value*. It describes the number of unrelated sequences in a similarity search of a sequence database that is expected to achieve a local alignment score as high or higher than the one obtained between the query sequence and the matching database sequence. In other words, the expected value describes the number of

hits one can expect to see just by chance when searching a database of a particular size with a sequence of a particular size. For example, an E value of 1 assigned to a hit can be interpreted as meaning that in a database of the current size one might expect to see 1 match with a similar score simply by chance. This means that the lower the E-value, or the closer it is to zero the more significant the match is. Since we work with related organisms, we can be very stringent, and thus reduce the size of the output.

Now let us take a closer look at the actual result of our query and find ways to answer the initial question: which proteins are potentially causing pathogenicity in *E. coli* O157:H7.

18.3.3.3 The BLAST+ Result File

How does the result file *h7vsk12.txt* look like from inside? Open it in a text editor or text viewer like *less*. To do this type

```
less h7vsk12.txt
```

You can scroll through the text using the arrow and PageUp/Down keys. Quit *less* by pressing **Q**.

```

File 11: h7vsk12.txt
1  BLASTP 2.2.25+
2
3  Reference: Stephen F. Altschul, Thomas L. Madden, Alejandro A.
4  Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J.
5  Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of
6  protein database search programs", Nucleic Acids Res. 25:3389-3402.
7
8  Reference for composition-based statistics: Alejandro A. Schaffer,
9  L. Aravind, Thomas L. Madden, Sergei Shavirin, John L. Spouge, Yuri
10 I. Wolf, Eugene V. Koonin, and Stephen F. Altschul (2001),
11 "Improving the accuracy of PSI-BLAST protein database searches with
12 composition-based statistics and other refinements", Nucleic Acids
13 Res. 29:2994-3005.
14
15 Database: Escherichia coli K12
16           4,146 sequences; 1,312,978 total letters
17
18 Query= gi|15799681|ref|NP_285693.1| thr operon leader peptide [Escherichia
19 coli O157:H7 str. EDL933]
20
21 Length=27
22
23 ***** No hits found *****
24
25 ...
26
27 Query= gi|15799682|ref|NP_285694.1| bifunctional aspartokinase I/homoserine
28 dehydrogenase I [Escherichia coli O157:H7 str. EDL933]
29
30 Length=820
31
Score  E

```

```

32 Sequences producing significant alignments: (Bits) lue
33
34 gi|16127996|ref|NP_414543.1| fused aspartokinase I and homoseri... 1682 .0
35 gi|16131778|ref|NP_418375.1| fused aspartokinase II/homoserine ... 344 e-95
36 gi|16131850|ref|NP_418448.1| aspartokinase III [Escherichia col... 162 e-41
37
38 > gi|16127996|ref|NP_414543.1| fused aspartokinase I and homoserine
39 dehydrogenase I [Escherichia coli str. K-12 substr. MGL655]
40 Length=820
41
42 Score = 1682 bits (4355), Expect = 0.0, Method: Compositional matrix adjust.
43 Identities = 818/820 (99%), Positives = 819/820 (99%), Gaps = 0/820 (0%)
44
45 Query 1 MRVLKFGGTSVANAERFLRVADILESNAQQGVATVLSAPAKITNHLVAMIEKTISGQDA 60
46 MRVLKFGGTSVANAERFLRVADILESNAQQGVATVLSAPAKITNHLVAMIEKTISGQDA
47 Sbjct 1 MRVLKFGGTSVANAERFLRVADILESNAQQGVATVLSAPAKITNHLVAMIEKTISGQDA 60
48
49 Query 61 LPNISDAERIFAELLTGLAAAPGFPLAQLKTFVDQEFQIKHVLHGISLLGQCPSINA 120
50 LPNISDAERIFAELLTGLAAAPGFPLAQLKTFVDQEFQIKHVLHGISLLGQCPSINA
51 Sbjct 61 LPNISDAERIFAELLTGLAAAPGFPLAQLKTFVDQEFQIKHVLHGISLLGQCPSINA 120
52
53 Query 121 ALICRGEKMSIAIMAGVLEARGHNVTVIDPVEKLLAVGHYLESTVDIAESTRRIAASRIP 180
54 ALICRGEKMSIAIMAGVLEARGHNVTVIDPVEKLLAVGHYLESTVDIAESTRRIAASRIP
55 Sbjct 121 ALICRGEKMSIAIMAGVLEARGHNVTVIDPVEKLLAVGHYLESTVDIAESTRRIAASRIP 180
56 ...

```

File 11 gives you just a glance of the content of *h7vsk12.txt*. I truncated the output at lines 25 and 56. The query results start with the version of the BLAST program used and the proper citation for this program (lines 1–13), and the database that has been queried (lines 15–16). Each query sequence is preceded with the string *Query*= (lines 18 and 27). The first sequence yielded no hit while the second yielded three hits. For each hit a sequence alignment is shown (line 45...). Scroll through the file and find out how it is structured.

To find out which proteins are specific to *E. coli* O157:H7, we are interested in all sequences from *E. coli* O157:H7 that do not yield any hit in *E. coli* K12.

18.3.4 Processing the BLAST+ Result File

Of course, it is almost impossible for us to scroll through the whole BLAST+ output file, *h7vsk12.txt*. It is simply too large. What can we do instead? As said above we are interested in query sequences that yield no hit. That means we should extract all protein names that are followed by the line “No hits foun”.

Here comes the programming language AWK into scene. AWK programs, usually called scripts, consist of two parts, a pattern and an action.

```
awk 'pattern{action} filename'
```

The pattern determines the lines of the input file *filename* that shall be treated, while the action is a list of commands to be executed. Type and run the following script:

```
awk '/Query=/ || /No hits/{print}' h7vsk12.txt
```

That should give you some output as shown in Terminal 249

```

Terminal 249: Extracting Lines
1 $ awk '/Query=/ || /No hits/{print}' h7vsk12.txt | head
2 Query= gi|15799681|ref|NP_285693.1| thr operon leader peptide [Escherichia
3 ***** No hits found *****
4 Query= gi|15799682|ref|NP_285694.1| bifunctional aspartokinase I/homoserine
5 Query= gi|15799683|ref|NP_285695.1| homoserine kinase [Escherichia coli
6 Query= gi|15799684|ref|NP_285696.1| threonine synthase [Escherichia coli
7 Query= gi|15799685|ref|NP_285697.1| hypothetical protein Z0005 [Escherichia
8 Query= gi|15799686|ref|NP_285698.1| hypothetical protein Z0006 [Escherichia
9 Query= gi|15799687|ref|NP_285699.1| inner membrane transport protein
10 Query= gi|15799688|ref|NP_285700.1| transaldolase B [Escherichia coli
11 Query= gi|15799689|ref|NP_285701.1| molybdenum cofactor biosynthesis
12 $

```

With the little AWK script in line 1 in Terminal 249, we use the action `print` to extract all lines from *h7vsk12.txt* that contain either the text pattern *Query=* or (`|` `|`) the text pattern *No hits* (the output is truncated since it has been piped to head). The patterns are enclosed in slashes, the whole script is enclosed in single quotes. The command `awk` is necessary to call the so-called commando interpreter for AWK.

Wow, we got a bit closer. From the previous output we have to extract all lines that are preceded by a line containing “No hits found”. This way we remove all lines that belong to proteins that yielded hits. Again we use AWK.

```

Terminal 250: Extracting Line Pairs
1 $ awk '/Query=/ || /No hits/{print}' h7vsk12.txt | wc -l
2 awk '{i++; line[i]=$0; if($0~/No hits/){print line[i-1]}}' | head
3 Query= gi|15799681|ref|NP_285693.1| thr operon leader peptide [Escherichia
4 Query= gi|15799699|ref|NP_285711.1| hypothetical protein Z0020 [Escherichia
5 Query= gi|15799700|ref|NP_285712.1| hypothetical protein Z0021 [Escherichia
6 Query= gi|15799704|ref|NP_285716.1| hypothetical protein Z0025 [Escherichia
7 Query= gi|15799716|ref|NP_285728.1| hypothetical protein Z0039 [Escherichia
8 Query= gi|15799732|ref|NP_285744.1| antitoxin of gyrase inhibiting
9 Query= gi|15799733|ref|NP_285745.1| toxin of gyrase inhibiting
10 Query= gi|15799741|ref|NP_285753.1| hypothetical protein Z0065 [Escherichia
11 Query= gi|15799753|ref|NP_285765.1| hypothetical protein Z0078 [Escherichia
12 Query= gi|15799784|ref|NP_285796.1| hypothetical protein Z0110 [Escherichia
13 $ awk '/Query=/ || /No hits/{print}' h7vsk12.txt | wc -l
14 1099
15
16 $

```

In Terminal 250, you see the real strength of Linux (note, that I had to break lines): we can connect many commands and programs. That means the output of one command immediately becomes the input of another command with no need to save the temporary result in a file. This functionality is called a pipe or piping because it is like a process pipeline. The vertical `|` bar is used to connect several commands. Let us take a closer look at line 1 from Terminal 250. The first command is exactly the same as line 1 in Terminal 249:

```
awk '/Query=/ || /No hits/{print}' h7vsk12.txt
```

It is followed by the piping symbol (|) and a second command (AWK script):

```
awk '{i++; line[i]=$0; if($0~/No hits/){print line[i-1]}}'
```

With this script we print each followed by a line containing the text pattern *No hits*. Did you recognize that blocks of commands are separated by semicolons?

Now you have a nice list of proteins that are unique to the pathogenic strain *E. coli* O157:H7. How many (keep the result!)? This is shown in line 15 in Terminal 250 on the facing page. Among these there are many proteins which are either putative, hypothetical, or unknown. In order to cleanup the list we apply the command `egrep`. It allows us to extract lines of a text file matching a given pattern. In our case, we like to exclude all lines containing one of the following words: unknown, Unknown, putative, or hypothetical (note that the case of letters matters in Linux). This is accomplished by applying the option `-v`.

```
egrep -v "([Uu]nknown|[Pp]utative|[Hh]ypothetical)"
```

Let us add this command to the pipe applied in Terminal 250 on the preceding page. The full command now becomes:

```

Terminal 251: Extracting Annotated Proteins
1  $ awk '/Query=/ || /No hits/{print}' h7vsk12.txt |
2  awk '{i++; line[i]=$0; if($0~/No hits/){print line[i-1]}}' |
3  egrep -v "([Uu]nknown|[Pp]utative|[Hh]ypothetical)" | head -20
4  Query= gi|15799681|ref|NP_285693.1| thr operon leader peptide [Escherichia
5  Query= gi|15799732|ref|NP_285744.1| antitoxin of gyrase inhibiting
6  Query= gi|15799733|ref|NP_285745.1| toxin of gyrase inhibiting
7  Query= gi|15799819|ref|NP_285831.1| fimbrial protein [Escherichia coli
8  Query= gi|15799899|ref|NP_285911.1| macrophage toxin [Escherichia coli
9  Query= gi|15799952|ref|NP_285964.1| cII antiterminator protein for prophage
10 Query= gi|15799959|ref|NP_285971.1| tail fiber protein from prophage
11 Query= gi|15799972|ref|NP_285984.1| polarity suppression protein encoded in
12 Query= gi|15799973|ref|NP_285985.1| capsid morphogenesis protein encoded in
13 Query= gi|15799976|ref|NP_285988.1| regulator encoded in prophage CP-933I
14 Query= gi|15799978|ref|NP_285990.1| alpha replication protein of prophage
15 Query= gi|15800218|ref|NP_286230.1| RTX family exoprotein [Escherichia coli
16 Query= gi|15800220|ref|NP_286232.1| membrane spanning export protein
17 Query= gi|15800441|ref|NP_286453.1| methylaspartate ammonia-lyase
18 Query= gi|15800442|ref|NP_286454.1| glutamate mutase subunit E [Escherichia
19 Query= gi|15800444|ref|NP_286456.1| methylaspartate mutase subunit S
20 Query= gi|15800487|ref|NP_286499.1| exonuclease encoded by prophage CP-933K
21 Query= gi|15800488|ref|NP_286500.1| Bet recombination protein of prophage
22 Query= gi|15800492|ref|NP_286504.1| antiterminator Q protein of prophage
23 Query= gi|15800500|ref|NP_286512.1| DNA packaging protein of prophage
24 $

```

Note, that I truncated the output to 20 lines with help of `head -20`. Lines 1–3 are one single line on your screen. How many proteins do you get this way (see Exercise 18.1 on p. 373)? These proteins are specific for *E. coli* O157:H7 and have an annotation. Are there obvious pathogens among these?

18.3.5 Playing with the E-Value

A crucial setting influencing our analysis result is the definition for *equal protein*. A BLAST hit is not found when two proteins are considered not to be equal. This is based on the BLAST E-value. For the previous examples we set it to 0.00001. The lower the E-value of a sequence alignment the more similar are the aligned sequences. The lower we set the expectation threshold (E-value) in `blastp` the less hits we should find for a given *E. coli* K12 sequence in *E. coli* O157:H7.

To verify this assumption, we have to execute `blastp` several times with different parameters. This job becomes easy when we employ a shell script as shown in Program 63. Save the script in a file named *autoblast.sh* and do not forget to make it executable.

```

      Program 63: autoblast.sh - A BLAST Wrapper
1  #!/bin/bash
2  # save as autoblast.sh
3  # changes E-value
4  for i in 1 0.1 0.001 0.0001 0.00001
5  do
6  ./ncbi-blast-2.2.25+/bin/blastp -db ecolik12 -query ecoli-h7.faa      +
7  -out h7vsk12-$i.txt -evalue $i
8  done

```

Lines 7–11 in Program 63 initiate a `for ... ; do ... ; done` construct that loops through all expectations values given in line 7 (see Sect. 10.7.5 on p. 144). The actual E-value is stored in variable *i*. The statement `$i` in line 10 (lines 9 and 10 are in fact one line) is replaced by the current value of variable *i*.

Do not forget to make the file executable with `chmod`.

```

      Terminal 252: Extracting Annotated Proteins
1  $ chmod u+x autoblast.sh
2  $ time ./autoblast.sh
3  Selenocysteine (U) at position 196 replaced by X
4  ...
5  Selenocysteine (U) at position 196 replaced by X
6
7  real      14m34.952s
8  user      14m23.074s
9  sys       0m3.868s
10
11 $ ls h7vsk12*
12 h7vsk12-0.00001.txt  h7vsk12-0.001.txt  h7vsk12-1.txt
13 h7vsk12-0.0001.txt  h7vsk12-0.1.txt   h7vsk12.txt
14 $ wc -l h7vsk12*
15  707447 h7vsk12-0.00001.txt
16  737091 h7vsk12-0.0001.txt
17  769447 h7vsk12-0.001.txt
18  848125 h7vsk12-0.1.txt
19  931356 h7vsk12-1.txt
20  707447 h7vsk12.txt
21  4700913 total
22 $

```

Terminal 252 shows the execution of *autoblast.sh*. After roughly 15 min (dependent on you computer system), we get the final result: five files with the results for different

E-values. The higher the E-value, the more lines we obtain in the result file. Why (see Exercise 18.3 on p. 373)? Now, let us search for sequences that yielded no result. In order to process all result files at once we employ the `for ...; do ...; done` construct from the BASH shell (see Sect. 10.7.5 on p. 144).

```

1  $ for i in h7vsk12-*; do echo -n $i" : "; awk '/Query=/ || /No hits/{print}' $i | +
2  awk '{i++; line[i]=$0; if($0~/No hits/){print line[i-1]}}' | wc -l; done
3  h7vsk12-0.00001.txt : 1099
4  h7vsk12-0.0001.txt : 1083
5  h7vsk12-0.001.txt : 1061
6  h7vsk12-0.1.txt : 981
7  h7vsk12-1.txt : 666
8  $
9  $ for i in h7vsk12-*; do echo -n $i" : "; awk '/Query=/ || /No hits/{print}' $i | +
10  awk '{i++; line[i]=$0; if($0~/No hits/){print line[i-1]}}' | +
11  egrep -v "([Uu]nknown|[Pp]utative|[Hh]ypothetical)" | wc -l; done
12  h7vsk12-0.00001.txt : 314
13  h7vsk12-0.0001.txt : 304
14  h7vsk12-0.001.txt : 297
15  h7vsk12-0.1.txt : 275
16  h7vsk12-1.txt : 189
17  $

```

The value of variable *i* equals each filename beginning with *h7vsk12-* for each loop. File *h7vsk12-0.00001.txt* was generated with the same E-value we used for the generation of *h7vsk12.txt*, i.e. 0.00001. We can observe that the number of open reading frames identified as being unique to *E. coli* strain O157:H7 decreases with increasing E-value. Try to explain this observation (see Exercise 18.4 on p. 373). Does it make sense to you?

18.3.6 Organize Results with MySQL

If you work with large genomes and repetitively extract information from BLAST+ result files you might want to save them in a MySQL database (see Chap. 16 on p. 295). However, it requires to bring your data into a tabular format. Luckily, BLAST+ helps you in this task. The option `-outfmt` allows you to bring your BLAST result into different formats, as shown in Table 18.1.

In the following example, we generate a tabular output for different E-values when blasting *E. coli* strain O157:H7 against *E. coli* strain K12.

```

1  Program 64: autoblast.sh - An Extended BLAST Wrapper
2  #!/bin/bash
3  # save as autoblast.sh
4  # changes E-value; output tabular
5  for i in 1 0.1 0.001 0.0001 0.00001
6  do
7  ./ncbi-blast-2.2.25+/bin/blastp -db ecolik12 -query ecoli-h7.faa +
8  -out h7vsk12-$i.tab -evalue $i -outfmt 6
9  done

```

Table 18.1 Parameter that influence the output format of BLAST+

Number	Output format
0	pairwise
1	query-anchored showing identities
2	query-anchored no identities
3	flat query-anchored, show identities
4	flat query-anchored, no identities
5	XML Blast output
6	tabular
7	tabular with comment lines
8	Text ASN.1
9	Binary ASN.1
10	Comma-separated values
11	BLAST archive format (ASN.1)

Program 64 resembles Program 63 on p. 360 except for two modifications. First, we change the file extension to *.tab* and second, we add the parameter `-outfmt 6`. The latter sets the output format to plain tabular. Run the modified script and take a look at one of the result files. You will see long gene identifiers and many number containing fields. For example, in the following Terminal 254, we pick one of the result files and shorten it a bit by deleting the RefSeq IDs.

```
Terminal 254: Fosrmatting Tabular Output
1 $ sed -e 's/|ref|NP_[0-9.]\+|//g' h7vsk12-0.00001.tab > h7vsk12-0.00001-gi.tab
2 $ head -5 h7vsk12-0.00001-gi.tab
3 gi|15799682 gi|16127996 99.76 820 2 0 1 820 1 820 0.0 1682
4 gi|15799682 gi|16131778 30.57 821 526 17 5 812 16 805 1e-95 344
5 gi|15799682 gi|16131850 30.15 471 288 10 3 460 6 448 9e-41 162
6 gi|15799683 gi|16127997 99.35 310 2 0 1 310 1 310 0.0 639
7 gi|15799684 gi|16127998 99.77 428 1 0 1 428 1 428 0.0 880
8 $
```

Table 18.2 lists, what the different fields stand for?

Table 18.2 Description of the various fields in the BLAST+ output when formatted with `-outfmt 6`

Field	Description	Abbreviation
01	Query ID	quid
02	Subject ID	suid
03	% of alignment identity	iden
04	Alignment length	alen
05	Mismatches in alignment	mism
06	Gap openings in alignment	gapo
07	First nucleotide of query sequence in alignment	qsta
08	Last nucleotide of query sequence in alignment	qend
09	First nucleotide of subject sequence in alignment	ssta
10	Last nucleotide of subject sequence in alignment	send
11	E-value for the alignment	eval
12	Bit score for the alignment	bits

18.3.6.1 Load BLAST+ Results into MySQL

Now let us transfer the file *h7vsk12-0.00001-gi.tab* into a MySQL table called *h7vsk1200001* (for an introduction to MySQL see Chap. 16 on p. 295). In the following Terminal, (which shows the mysql command line) we create the table (lines 1–3), load the data (line 25), and extract all entries for open reading frames with 100 % matches (*iden*) and a bit score (*bits*) higher than 1,800 (line 29). In lines 39–40 at Terminal 255 we display the query gene ID (*quid*), subject gene ID (*suid*), %-identity (*iden*), and alignment length (*alen*) for database entries (i.e. BLAST hits) with a %-identity equal to 100 and an alignment length greater than 800 bp (base pairs).

```

Terminal 255: Using MySQL
1  mysql> CREATE TABLE h7vsk1200001 (quid VARCHAR(15), suid VARCHAR(15), iden FLOAT, +
2      alen INT, mism INT, gapo INT, qsta INT, qend INT, ssta INT, send INT, +
3      eval FLOAT, bits INT);
4  Query OK, 0 rows affected (0.01 sec)
5
6  mysql> DESCRIBE h7vsk1200001;
7  +-----+-----+-----+-----+-----+-----+
8  | Field | Type          | Null | Key | Default | Extra |
9  +-----+-----+-----+-----+-----+-----+
10 | quid  | varchar(15)   | YES  |     | NULL    |       |
11 | suid  | varchar(15)   | YES  |     | NULL    |       |
12 | iden  | float         | YES  |     | NULL    |       |
13 | alen  | int(11)       | YES  |     | NULL    |       |
14 | mism  | int(11)       | YES  |     | NULL    |       |
15 | gapo  | int(11)       | YES  |     | NULL    |       |
16 | qsta  | int(11)       | YES  |     | NULL    |       |
17 | qend  | int(11)       | YES  |     | NULL    |       |
18 | ssta  | int(11)       | YES  |     | NULL    |       |
19 | send  | int(11)       | YES  |     | NULL    |       |
20 | eval  | float         | YES  |     | NULL    |       |
21 | bits  | int(11)       | YES  |     | NULL    |       |
22 +-----+-----+-----+-----+-----+-----+
23 12 rows in set (0.00 sec)
24
25 mysql> LOAD DATA LOCAL INFILE "h7vsk12-0.00001-gi.tab" INTO TABLE h7vsk1200001;
26 Query OK, 21294 rows affected, 978 warnings (0.09 sec)
27 Records: 21294 Deleted: 0 Skipped: 0 Warnings: 978
28
29 mysql> SELECT * FROM h7vsk1200001 WHERE iden = 100 AND bits > 1800;
30 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
31 | quid      | suid      | iden | alen | mism | gapo | qsta | qend | ssta | send | eval | bits |
32 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
33 | gi|15799743 | gi|16128053 | 100 | 968 | 0 | 0 | 1 | 968 | 1 | 968 | 0 | 1991 |
34 | gi|15799798 | gi|16128107 | 100 | 887 | 0 | 0 | 1 | 887 | 1 | 887 | 0 | 1845 |
35 | gi|15804578 | gi|16131818 | 100 | 1407 | 0 | 0 | 1 | 1407 | 1 | 1407 | 0 | 2886 |
36 +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
37 3 rows in set (0.00 sec)
38
39 mysql> SELECT quid, suid, iden, alen FROM h7vsk1200001 WHERE iden = 100 AND
40      alen > 800;
41 +-----+-----+-----+-----+
42 | quid      | suid      | iden | alen |
43 +-----+-----+-----+-----+
44 | gi|15799743 | gi|16128053 | 100 | 968 |
45 | gi|15799798 | gi|16128107 | 100 | 887 |
46 | gi|15799859 | gi|16128170 | 100 | 810 |
47 | gi|15803116 | gi|16130513 | 100 | 857 |
48 | gi|15804578 | gi|16131818 | 100 | 1407 |

```

```

49  +-----+-----+-----+-----+
50  5 rows in set (0.00 sec)
51
52  mysql>

```

In line 25 in Terminal 255, we actually load the data into the table *h7vsk1200001*. At some Linux installation, you might not have the right to execute the `LOAD DATA LOCAL INFILE ... INTO TABLE ...` command. In this case, switch to the terminal and execute

```
mysqlimport --local -D -u awkologist -p compbiol h7vsk1200001.tab
```

To be successful you have to change the filename of *h7vsk12-0.00001-gi.tab* to *h7vsk1200001.tab*.

18.3.6.2 Loading Gene Annotations into MySQL

You probably recognized that the tabular BLAST results do not contain any gene annotations. There is, however, a nice command provided by the BLAST+ suite that enables retrieval of sequence and annotation information from the BLAST database in Sect. 18.3.3.1 on p. 353. Therefore, it was important to set the option `parse_seqids`, while running `makeblastdb`. The magic command's name is `blastdbcmd`.

```

Terminal 256: Extracting Information from BLAST DB with blastdbcmd
1  $ ./ncbi-blast-2.2.25+/bin/blastdbcmd -db ecolik12 -entry "all" | head
2  >gi|16127995|ref|NP_414542.1| thr operon leader peptide [Escherichia ...
3  MKRISTTITTTITTTGNGAG
4  >gi|16127996|ref|NP_414543.1| fused aspartokinase I and homoserine d ...
5  MRVLKFGGTSVANAERFLRVADILESNAQQGVATVLSAPAKITNHLVAMIEKTISGQDALPNISDAERIFAELLTGLAA
6  AQPGFPLAQLKTFVDQEFQAIKHVLHGISLLGQCPDSINAALICRGEKMSIAIMAGVLEARGHNVTVIDPVEKLLAVGHY
7  LESTVDIAESTRRIAASRIPADHVMVMAGFTAGNEKGELVVLGRNGSDYSAAVLAACLRADCCCEIWTVDGVYTCDPRQV
8  PDARLLKSMYSYQAEMLSYFGAKVLHPRITITPIAQFQIPCLIKNTGNPQAPGTLIGASRDEDELVPKGISNLNNMAMFSV
9  SGPGMKGVMGMAARVFAAMSRARISVVLITQSSSEYSISFCVPQSDCVRAERAMQEEFYLELKEGLEPLAVTERLAIIS
10 VVGDGMRTLGRISAKFFAALARANINIVAIAQSSSERSISVVVNNDDATTGVRVTHQMLFNTDQVIEVFVIGVGGVGGAL
11 LEQLKRQQSWLKNKHIDLRVCGVANSKALLTNVHGLNLLENWQEELAQAKEPFNLGRILRLVKEYHLLNPVIVDCTSSQAV
12 $ ./ncbi-blast-2.2.25+/bin/blastdbcmd -db ecolik12 -entry "all" +
13 -outfmt "gi|%g%t" | sed -e 's/\[.\*/\tK12/' -e 's/#/\t/' | head
14 gi|16127995| thr operon leader peptide K12
15 gi|16127996| fused aspartokinase I and homoserine dehydrogenase I K12
16 gi|16127997| homoserine kinase K12
17 gi|16127998| threonine synthase K12
18 gi|16127999| predicted protein K12
19 gi|16128000| Peroxide resistance protein, lowers intracellular iron K12
20 gi|16128001| predicted transporter K12
21 gi|16128002| transaldolase B K12
22 gi|16128003| molybdochelatase incorporating molybdenum into molybdopterin K12
23 gi|16128004| inner membrane protein, Grp1_Fun34_YaaH family K12
24 $ ./ncbi-blast-2.2.25+/bin/blastdbcmd -db ecolik12 -entry "all" +
25 -outfmt "gi|%g%t" | sed -e 's/\[.\*/\tK12/' -e 's/#/\t/' > annotation-k12.tab
26 $

```

In the following Terminal 257, we evaluate file *annotation-k12.tab* in terms of fields length with AWK.

```

1  $ awk -F"\t" '{for(i=1;i<=NF;i++){if(length($i)>field[i]){field[i]=length($i)}}}
2  END{for(x=1; x<=i; x++){print field[x]}}' annotation-k12.tab
3  12
4  170
5  3
6
7  $

```

First, we set the field separator to tabular with the AWK option `-F"\t"`. Then AWK loops through all fields (variable *NF*, number of fields) evaluates the field length and stores the field's character length in array *field* (with the field number as index), if the length exceeds a previously stored value. The final END-block outputs the maximum character length of each field. In my example, the maximum length of the field 1 is 12 characters, of field 2 it is 170 and of field 3 it is 3 characters. This information is helpful for the creation of an appropriate MySQL table as shown in Terminal 258.

```

1  mysql> CREATE TABLE ecoliano (gene VARCHAR(12), anno VARCHAR(170),
2  coli VARCHAR(3));
3  Query OK, 0 rows affected (0.04 sec)
4
5  mysql> DESCRIBE ecoliano;
6
7  +-----+-----+-----+-----+-----+-----+
8  | Field | Type          | Null | Key | Default | Extra |
9  +-----+-----+-----+-----+-----+-----+
10 | gene  | varchar(12)   | YES  |     | NULL    |      |
11 | anno  | varchar(170)  | YES  |     | NULL    |      |
12 | coli  | varchar(3)    | YES  |     | NULL    |      |
13 +-----+-----+-----+-----+-----+-----+
14 3 rows in set (0.00 sec)
15
16 mysql> LOAD DATA LOCAL INFILE "annotation-k12.tab" INTO TABLE ecoliano;
17 Query OK, 4146 rows affected (0.02 sec)
18 Records: 4146 Deleted: 0 Skipped: 0 Warnings: 0
19
20 mysql> SELECT * FROM ecoliano LIMIT 5;
21
22 +-----+-----+-----+-----+-----+-----+
23 | gene          | anno                                     | coli |
24 +-----+-----+-----+-----+-----+-----+
25 | gi|16127995   | thr operon leader peptide              | K12   |
26 | gi|16127996   | fused aspartokinase I and homoserine dehydrogenase I | K12   |
27 | gi|16127997   | homoserine kinase                      | K12   |
28 | gi|16127998   | threonine synthase                    | K12   |
29 | gi|16127999   | predicted protein                     | K12   |
30 +-----+-----+-----+-----+-----+-----+
31 5 rows in set (0.00 sec)
32
33 mysql>

```

In lines 1–2 in Terminal 258, we create a new tables named *ecoliano*. In line 15, we fill this table with the content file *annotation-k12.tab* that we created in lines 25–26 in Terminal 256 on the previous page. It is important that you login to MySQL (typing

the command `mysql`) in the directory where you have saved *annotation-k12.tab*. Otherwise, you would have to provide the complete path to this file in line 15. In line 19, we extract the first five entries (`LIMIT 5`) of table *ecolianno*. Now you should be able to do the same with the annotations from *E. coli* strain O157:H7. How would you proceed (see Exercise 18.5 on p. 373)?

18.3.6.3 Extract Highly Conserved Genes with Annotations

Till-now we created two kinds of tables, that contain BLAST query results and gene annotations. In Terminal 259, we bring both tables, i.e. *h7vsk1200001* and *ecolianno* together. The task is to extract all highly conserved genes from the BLAST result (100 % sequence identity over at least 800 base pairs) table *h7vsk1200001* together with their annotation from *ecolianno*. For both tables the shortcut *b* and *e* are used, respectively.

```

1  mysql> SELECT b.quid, b.suid, b.iden, b.alen, e.anno FROM h7vsk1200001 AS b,
2      ecolianno AS e WHERE b.iden = 100 AND b.alen > 800 AND b.suid = e.gene;
3
4
5  +-----+-----+-----+-----+-----+-----+-----+
6  | quid      | suid      | iden | alen | anno                                     |
7  +-----+-----+-----+-----+-----+-----+-----+
8  | gi|15799743 | gi|16128053 | 100 | 968 | RNA polymerase-associated helica... |
9  | gi|15799798 | gi|16128107 | 100 | 887 | pyruvate dehydrogenase, decarbox... |
10 | gi|15799859 | gi|16128170 | 100 | 810 | outer membrane protein assembly ... |
11 | gi|15803116 | gi|16130513 | 100 | 857 | protein disaggregation chaperone   |
12 | gi|15804578 | gi|16131818 | 100 | 1407 | RNA polymerase, beta prime subunit |
13 +-----+-----+-----+-----+-----+-----+-----+
14 5 rows in set (0.00 sec)
15
16 mysql>
```

As seen from the results in lines 7–11 in Terminal 259, only five genes are matching the query criteria. Two of them are involved in transcription.

18.3.7 Extract Unique ORFs

You might ask yourself if there is a way to do with MySQL what we have done with AWK previously, i.e., extract ORFs unique to the pathogenic *E. coli* strain O157:H7 or O104:H4. Oh yes, of course it is. To answer the question, which ORFs are unique to the pathogenic strain, we could check which ORF IDs of the annotation table *ecolianno.gene* are not present in the BLAST+ result table *h7vsk1200001.suid*. Why? Because, in contrary to the default textual BLAST+ result file *h7vsk1200001.txt*, query sequence IDs (*E. coli* strain O157:H7 ORFs) not yielding a hit in the BLAST+ database (*E. coli* K12 ORFs) are not listed in the tabular output file *h7vsk1200001.tab*.

Still now we have been using the relation between two tables to connect them with an equality statement such as `WHERE b.suid = e.gene`. From a computational point of view, this is crude programming (allowed for biologists only) and it cannot be used in our case. We would like to get ORF IDs from table *ecolianno* WHERE `b.suid != e.gene`. This, however, is misunderstood by MySQL, as shown in Terminal 260.

```

1  mysql> SELECT count(e.gene) FROM ecolianno AS e, h7vsk1200001 AS h
2  WHERE e.gene != h.suid;
3  +-----+
4  | count(e.gene) |
5  +-----+
6  |      88264608 |
7  +-----+
8  1 row in set (6.11 sec)
9
10 mysql> SELECT count(e.gene) FROM ecolianno AS e LEFT JOIN
11 h7vsk1200001 AS h
12 ON h.suid = e.gene WHERE h.suid IS NULL LIMIT 5;
13 +-----+
14 | count(e.gene) |
15 +-----+
16 |          421 |
17 +-----+
18 1 row in set (1 min 29.11 sec)
19
20 mysql> SELECT e.gene, h.suid, e.anno
21 FROM ecolianno AS e LEFT JOIN h7vsk1200001 AS h ON h.suid = e.gene
22 WHERE h.suid IS NULL limit 5;
23 +-----+-----+-----+-----+
24 | gene          | suid | anno                                |
25 +-----+-----+-----+-----+
26 | gi|16127995   | NULL | thr operon leader peptide          |
27 | gi|16128010   | NULL | IS186 transposase                   |
28 | gi|49175991   | NULL | toxic membrane protein, small     |
29 | gi|16128016   | NULL | IS1 repressor TnpA                 |
30 | gi|226524697  | NULL | Inhibitor of glucose uptake        |
31 +-----+-----+-----+-----+
32 5 rows in set (1.40 sec)
33
34 mysql>

```

Simply counting, the number of matching IDs in line 6 (88264608) shows us that there is something wrong in paradise. Obviously, we have to refine our query—in fact, we must employ a JOIN statement.

18.3.8 Visualize and Analyze Results with R

Now let us take a quick visual look at the BLAST+ result file with the statistical package R (see Chap. 17 on p. 317). Prerequisite to have fun with the shown examples is the installation of the R-packages *DBI* and *RMySQL* from the CRAN repository (see Sect. 17.7 on p. 339).

In Terminal 261, we initiate a new R session by typing `R` in the command line. In line 10, we load the library *RMySQL* that allows to connect with a remote MySQL database. After loading the MySQL driver (there are also drivers for Oracle, PostgreSQL and SQLite), we establish the connection in line 13/14. In line 15, we trigger a MySQL query. The result is stored in the variable *data* (which is, in fact, an array). Since we retrieve fields *iden*, *alen*, *eval* from table *h7vsk1200001*, variable *data* have three dimensions or columns.

```

1  $ R
2
3  R version 2.11.1 (2010-05-31)
4  Copyright (C) 2010 The R Foundation for Statistical Computing
5  ...
6  Type 'demo()' for some demos, 'help()' for on-line help, or
7  'help.start()' for an HTML browser interface to help.
8  Type 'q()' to quit R.
9
10 > library(RMySQL)
11 Loading required package: DBI
12 > drv = dbDriver("MySQL")
13 > con = dbConnect(drv,host="localhost",dbname="compbiol", +
14                   user="awkologist",pass="awkology")
15 > data = dbGetQuery(con,statement="SELECT iden, alen, eval FROM h7vsk1200001")
16 > attach(data)
17 > plot(iden,alen,xlab="Identity [%]",ylab="Alignment Length [bp]") # Plot1
18 > abline(lm(alen~iden),col="red")
19 > summary(lm(alen~iden))
20
21 Call:
22 lm(formula = alen ~ iden)
23
24 Residuals:
25     Min       1Q   Median       3Q      Max
26 -297.53  -63.99  -22.85   25.86  1800.89
27
28 Coefficients:
29             Estimate Std. Error t value Pr(>|t|)
30 (Intercept)  219.50175    1.88358   116.53  <2e-16 ***
31 alen         0.97030     0.03696    26.25  <2e-16 ***
32 ---
33 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
34
35 Residual standard error: 145.9 on 21292 degrees of freedom
36 Multiple R-squared:  0.03136,    Adjusted R-squared:  0.03131
37 F-statistic: 689.3 on 1 and 21292 DF,  p-value: < 2.2e-16
38
39 > hist(iden,xlab="Identity [%]",breaks=70) # Plot 2
40 >

```

You can look at the content of column one by typing `data[[1]]`. Try it out. However, it is much more convenient to use the field names as defined in MySQL. For that you have to attach these data as shown in line 16 with the R command `attach()`.

In line 17, in Terminal 261, we plot column *iden* (percent identity of the alignment) against column *alen* (length of the alignment)(plot not shown). With `xlab` and `ylab`,

we set the x -axis and y -axis labels, respectively. Line 18 is more complex—here we plot a linear regression line. The command `abline()` is used to plot straight lines. `abline(2, 5)` plots a line in the current open plot which intercepts the y -axis at 2 and has a slope of 5. In our case, both slope and Y -intercept are provided by the result of the command `lm`. This command is used to fit linear models. In the context with `abline()` a regression line for *alen* versus *iden* is generated. With `col="red"` the line becomes colored in red. With the command shown in line 19, we get information about the goodness of the regression line. R^2 gives a goodness-of-fit of the linear regression (line 36)—with 0.03 this is very bad and it follows that a nonlinear regression would probably describe the data in a much better way. Do not make mistake of using R^2 as your main criterion for whether a fit is reasonable. Only a high R^2 tells you that the data points come very close to the regression line. That does not mean that the fit is statistically good. Line 37 contains the results of the F -statistics—this is a measure for how good the regression line describes the data. With a p -value below 0.05 (here it is $2e-16$), we can be sure that there is a significant correlation between the alignment length and the percent of identical nucleotides. Finally, we plot a histogram (`hist()`) showing the distribution of sequence alignment identities (see Fig. 18.7). With `breaks=70`, we set the number of bins (categories, intervals) that are used to subdivide the data. The y -axis shows the number of data points (frequency) in each bin. We can see that most alignments have an identity of either 100 % or around 28 %. In words, there are either perfectly matching sequences stretches (roughly 2500) or a peak of around 28 % sequence identity. Note that the sum of sequence alignments does not match the number of protein sequences that have been blasted. Do you have an explanation (see Exercise 18.6 on p. 373)? For the following example in Terminal 262, I assume that you are still logged in to R from the previous session or at least saved and restored the session. Otherwise you

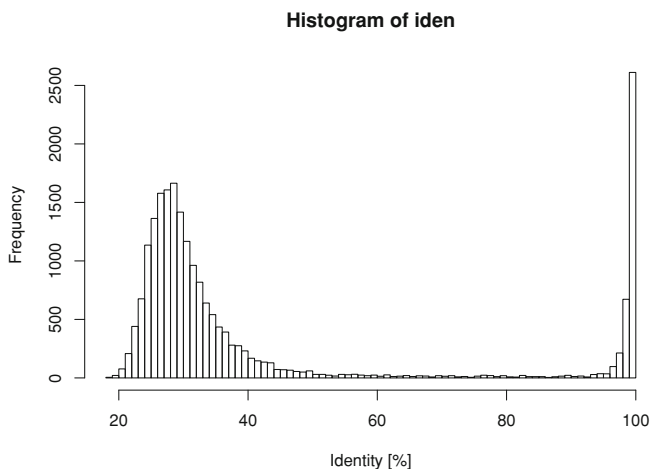


Fig. 18.7 Data Analysis. Histogram of all identity values split into 70 bins. Empty bins between 0–20 are not shown

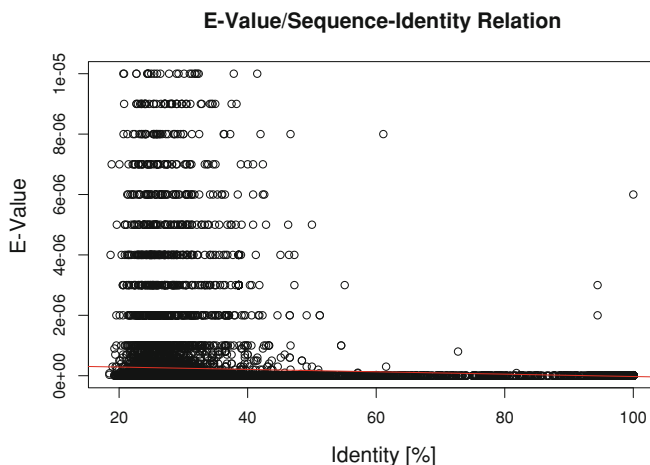


Fig. 18.8 Data Analysis. XY-Plot of all alignment identities against all E-values. The *red line* shows the linear regression

would have to load *RMySQL* again, connect to your MySQL database download and attach the data as shown in Terminal 261 on p. 368. In line 1/2 we create a plot of sequence alignment identities versus E-values (see Fig. 18.8) and add a regression line. Then we perform a regression analysis.

```

Terminal 262: XY-Plot and Regression
1  > plot(iden,eval,xlab="Identity [%]",ylab="E-Value",
2      main="E-Value/Sequence-Identity Relation") # Plot 3
3  > abline(lm(eval~iden),col="red")
4  > summary(lm(eval~iden))
5
6  Call:
7  lm(formula = eval ~ iden)
8
9  Residuals:
10      Min       1Q   Median       3Q      Max
11 -2.899e-07 -2.556e-07 -2.375e-07 -1.232e-08  9.797e-06
12
13  Coefficients:
14              Estimate Std. Error t value Pr(>|t|)
15 (Intercept)  3.655e-07  1.234e-08   29.61  <2e-16 ***
16 iden        -3.917e-09  2.422e-10  -16.18  <2e-16 ***
17 ---
18  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
19
20  Residual standard error: 9.564e-07 on 21292 degrees of freedom
21  Multiple R-squared:  0.01214,    Adjusted R-squared:  0.01209
22  F-statistic: 261.6 on 1 and 21292 DF,  p-value: < 2.2e-16
23
24  >

```

Again, we observe a strong correlation between both parameters. Since the p-value lays below 0.05, we can be sure that there is a significant correlation between E-value and the percent of identical nucleotides.

In Terminal 263, we analyze the maximum alignment length of the BLAST+ by comparing selected proteins from *E. coli* strain K12 and *E. coli* strain O157:H7. Therefore, the alignment lengths are divided into three groups.

```

1  > library(RMySQL)
2  > drv = dbDriver("MySQL")
3  > con = dbConnect(drv,host="localhost",dbname="compbio1",      +
4      user="awkologist",pass="awkology")
5  > dna<-dbGetQuery(con,statement="SELECT a.alen                +
6      FROM h7vsk1200001 AS a, ecolianno AS b                  +
7      WHERE anno LIKE '%DNA polymerase%' AND a.suid = b.gene")
8  > length(dna[[1]])
9  [1] 22
10 > dehydrogenase<-dbGetQuery(con,statement="SELECT a.alen      +
11     FROM h7vsk1200001 AS a, ecolianno AS b                  +
12     WHERE anno LIKE '%dehydrogenase%' AND a.suid = b.gene")
13 > length(dehydrogenase[[1]])
14 [1] 551
15 > cytochrome<-dbGetQuery(con,statement="SELECT a.alen         +
16     FROM h7vsk1200001 AS a, ecolianno AS b                  +
17     WHERE anno LIKE '%cytochrome%' AND a.suid = b.gene")
18 > length(cytochrome[[1]])
19 [1] 41
20
21 > boxplot(dehydrogenase[[1]],dna[[1]],cytochrome[[1]],notch=T,  +
22     names=c("DH","DNA-Pol","CYT"),main="Maximum Alignment Length [bp]")
23 Warning message:
24 In bxp(list(stats = c(69, 236.5, 308, 455, 781, 76, 139, 232.5, :
25     some notches went outside hinges ('box'): maybe set notch=FALSE
26
27 > t.test(dehydrogenase[[1]],cytochrome[[1]])
28
29     Welch Two Sample t-test
30
31 data:  dehydrogenase[[1]] and cytochrome[[1]]
32 t = 2.6552, df = 48.603, p-value = 0.01069
33 alternative hypothesis: true difference in means is not equal to 0
34 95 percent confidence interval:
35   14.95268 108.11093
36 sample estimates:
37 mean of x mean of y
38   347.1416  285.6098
39
40 > mean(dehydrogenase[[1]])
41 [1] 347.1416
42 > mean(cytochrome[[1]])
43 [1] 285.6098
44 > q()
45 Save workspace image? [y/n/c]: y
46 $

```

The first group is stored at variable *dna* in lines 5–7 and consists of proteins whose annotation matches to “DNA-polymerase”. This is done by using the following clause in MySQL: `WHERE anno LIKE '%DNA polymerase%'` where *anno* is the field name and the percentage characters function as wild cards. Thus, any or no text may precede or follow the query term “DNA-polymerase”. The second group in variable *dehydrogenase* consists of the alignment length *alen* of all pro-

teins coding for “dehydrogenases”, while the last group (*cytochrome*) consists of cytochromes. In lines 8, 13 and 18 we count the number of values (lines) in each variable (table column) with `length()`. In lines 21–22, we create the boxplot for datasets `dehydrogenase[[1]]`, `dna[[1]]`, `cytochrome[[1]]`. We apply three attributes to `boxplot()`: (a) With `notch=T` notches are added to the boxes. The notched boxplot shows confidence intervals around the median. If the notches of two boxes do not overlap we can be 95% confident that the two medians differ. The notches extend to $\pm 1.58 \times IQR / \sqrt{n}$, where *IQR* represents the interquartile range and *n* the sample size. (b) `names=c("DH", "DNA-Pol", "CYT")` provides names to the boxes where the command `c()` concatenates the values “DH”, “DNA-Pol” and “CYT” to a list. (c) With `main="..."` we provided a title for the diagram. After execution of `boxplot()` we get the plot as shown in Fig. 18.9.

A visual inspection of the boxes reveals that the medians of DNA-Pol and CYT do not differ, while both seem to differ significantly from the median of DH. Significantly, remember that the notches describe a 95% confidence interval. In line 27, we perform a t-test in order to check if the means of the maximum alignment length in dehydrogenases and cytochromes differ significantly, which they do since the p-value is below 0.05. Check, if there is a correlation for the data from both dehydrogenases and DNA-polymerases, (see Exercise 18.7). In line 44, we quit R by typing `q()`. You have then the choice to either store an image of the workspace, i.e. the values of all variables, or not.

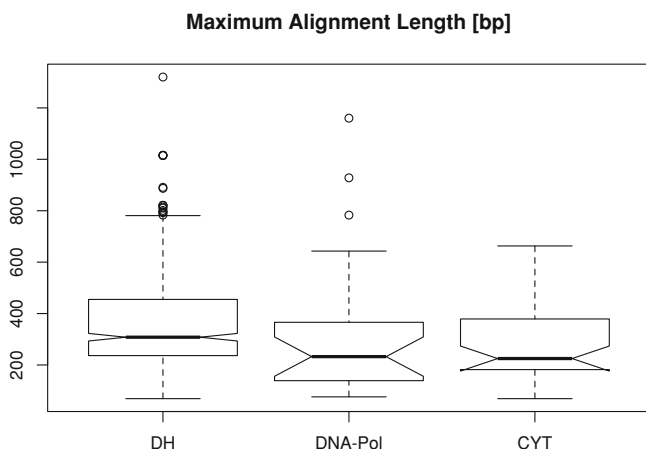


Fig. 18.9 Data Analysis. Boxplot of the distribution of maximum alignment lengths for proteins that contain *dehydrogenase* (DH), *DNA polymerase* (DNA) or *cytochrome* (CYT) in their annotation, respectively

Exercises

18.1 When compared to *E. coli* strain K12, *E. coli* strain O157:H7 has 1,099 unique protein sequences (see Terminal 250 on p. 358). How many annotated unique protein sequences are encoded by *E. coli* strain O157:H7?

18.2 How many unique annotated proteins do you find for (a) *E. coli* strain O104:H4 when compared to *E. coli* strain K12 and (b) for *E. coli* strain O104:H4 when compared to *E. coli* strain O157:H7?

18.3 Blasting one genome against another—or one sequence against a set of other sequences—yields an increasing number of hits with higher E-values. Why?

18.4 Why does the number of open reading frames identified as being unique to the query strain decrease with increasing E-value?

18.5 Add the annotations for *E. coli* strain O157:H7 to the MySQL table *ecolianno*.

18.6 The genome of *E. coli* strain O157:H7 encodes for roughly 5,400 proteins (see Terminal 246 on p. 352). However, the tabular BLAST+ output file *h7vsk12-0.00001.tab* has over 21,000 lines, i.e. alignment. How can that be?

18.7 Is there a significant correlation between maximum alignment lengths of dehydrogenases and DNA-polymerases, too?

Chapter 19

Limits of BLAST and Homology Modeling

19.1 The Project

Very often it is difficult to find homologs sequences just by using BLAST. This is because bad expectation values (E-Values) do not necessarily mean bad results. This project will teach you that it can help to look at the secondary and tertiary structure as well and apply *a priori* knowledge.

Imagine the following scientific setting. A research group has described the maturation process of a hydrogenase in *Escherichia coli*. Hydrogenases are enzymes that catalyze the reduction of protons to hydrogen gas and *vice versa*. These enzymes are complex metalloproteins containing FeS-clusters and an FeFe or FeNi active site with carbon monoxide and cyanide as natural ligands. Maturation, i.e., making, of hydrogenases requires many accessory proteins. One of those is the hydrogenase maturing endopeptidase HybD in *E. coli* (Fritsche et al. 1999). This endopeptidase has been well characterized and even its tertiary structure is available.

We know that many cyanobacteria possess hydrogenases, too (Fig. 19.1). A bidirectional hydrogenase is functionally coupled to either photosynthesis or respiration (not shown) and can catalyze either uptake or production of hydrogen gas, *in vivo*. The uptake hydrogenase is functionally coupled to the nitrogenase. Nitrogenases convert atmospheric nitrogen to ammonia, require a lot ATP and NADH, and release molecular hydrogen as a by-product. Uptake hydrogenases recycle this hydrogen gas thereby saving the cell energy.

Little is known about the maturation of cyanobacterial hydrogenases. Your task is to identify cyanobacterial endopeptidases homologs to HybD [This exercise is based on Wünschiers et al. (2003)]. We restrict the query to three representative cyanobacterial species: *Nostoc punctiforme* PCC 73102, *Nostoc* sp. PCC 7120, and *Synechocystis* sp. PCC 6803. They have been chosen because *Nostoc punctiforme* PCC 73102 has one uptake hydrogenase, *Nostoc* sp. PCC 7120 has one uptake and one bidirectional hydrogenase, and *Synechocystis* sp. PCC 6803 has one bidirectional hydrogenase. The *a priori* knowledge we use is that every hydrogenase maturing endopeptidase must contain the amino acids found in the catalytic active site. Especially the

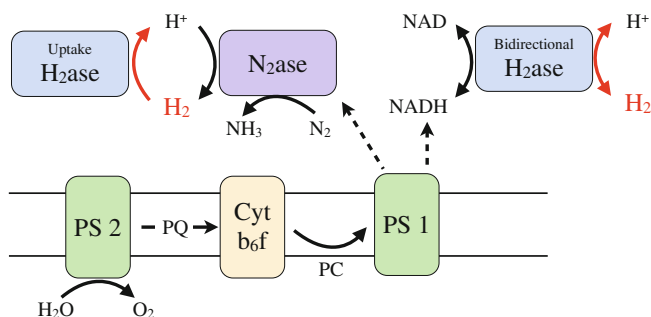


Fig. 19.1 Hydrogenases. This extremely simplified, non-stoichiometric scheme of photosynthesis shows the functional connection of the uptake hydrogenase (H_2ase) and bidirectional hydrogenase to nitrogenase (N_2ase) and photosynthesis, respectively. PS 1: photosystem 1; PS 2: photosystem 2; PQ: plastoquinone; PC: plastocyanin;

metal-ligating residues shown in Fig. 19.2 (red colored) must be present ($NH_2 \cdots [ED] \cdots D \cdots H \cdots COOH$).

19.2 Bioinformatic Tools

To solve this task you will work with BLAST+, the protein secondary structure prediction tool Jpred, the homology modeler SWISS-MODEL, and the structure viewer Jmol.

19.2.1 *genomicBLAST*

For this example we use the web-based version of *genomicBLAST* (Cummings et al. 2002). It allows for the selection of particular target organisms. The program can be found here: http://www.ncbi.nlm.nih.gov/sutils/genom_table.cgi?.

19.2.2 *ClustalW*

BLAST creates local sequence alignments. With *ClustalW* we can create global sequence alignments (see Sect. 9.4 on p. 120 or 4.1.2.5 on p. 43). These help us to see which parts in a bunch of protein sequences are conserved or, as in our case, if conserved functional amino acids align.

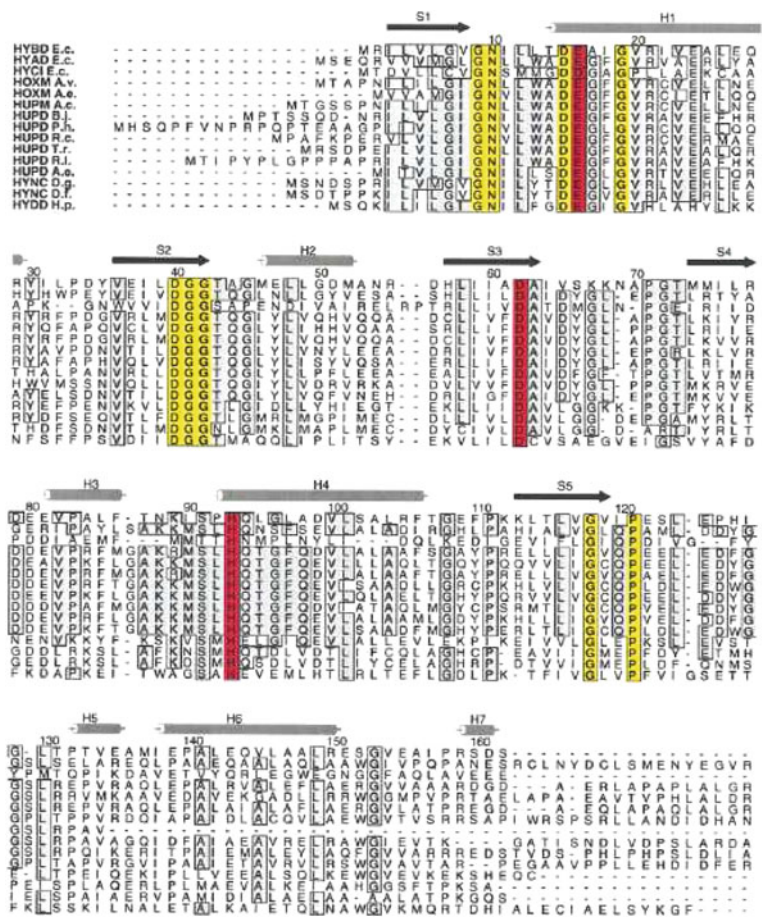


Fig. 19.2 HybD Sequences. Alignment of HybD protein sequences from different sources highlighting the active site in red [Fig. 2 from Fritsche et al. (1999)]

19.2.3 Jpred

Jpred (Version 3) is a web server that predicts the secondary structure, i.e., the position of α -helices and β -sheets, of an amino acid sequence on the basis of a neural network (Cole et al. 2008). The program can be found here: <http://www.compbio.dundee.ac.uk/www-jpred/index.html>.

19.2.4 SWISS-MODEL

SWISS-MODEL is a web-based integrated service dedicated to protein structure homology modeling (Arnold et al. 2006). We will use it to align potential cyanobacterial hydrogenase maturing endopeptidases (our candidates) to the given 3D-structure of HybD from *E. coli*. This way we can check if the conserved active site forming amino acids sit at the right position. SWISS-MODEL can be accessed here: <http://swissmodel.expasy.org/>

19.2.5 Jmol

Jmol is a free, open source molecule structure viewer. Since it is based on Java it runs on all operating systems that can run the Java Runtime Environment (see www.java.com). For the first edition of this book used Rasmol (<http://openrasmol.org/>) as a molecule view. Jmol has become more widespread, therefore I switched. With few exceptions, the command sets are compatible. A short overview of Jmol's commands is given in Appendix A.6 on p. 430. Jmol can be accessed and downloaded here: <http://jmol.sourceforge.net/> Alternatively, look a look at Sect. 4.1.2.5 on the p. 43.

19.3 Detailed Instructions

The instructions given here are very concise. This is because you work mainly with the web browser and probably know how to do that.

19.3.1 Download *E. coli* HybD Sequences

To download the sequence from GenBank goto <http://www.ncbi.nlm.nih.gov>, enter “gi|7546418” (note: there are no spaces) in the query field, select “Protein” in the pull-down menu and hit the “Search” button. The entry entitled “Chain A, Hydrogenase Maturing Endopeptidase HybD From *E. coli*” will open. In the “Display Setting” pull-down menu select “Apply” (Fig. 19.3). Store the result in a file named *hybd.fasta* (see File 12) and rename the sequence identifier to *E. coli*.

File 12: *hybd.fasta*

```

1 >ecoli
2 MRILVLGVGNILLTDEAIGVRIVEALEQRYILPDYVEILDGGTAGMELLGDMANRDHLIADAIVSKKNA
3 PGTMMILRDEEVPALFTNKISPHQLGLADVLSALRFTGEFPKKLTLVGVIPESELEPHIGLTPTEAMIEP
4 ALEQVLAALRESGVEAIPRSDS

```

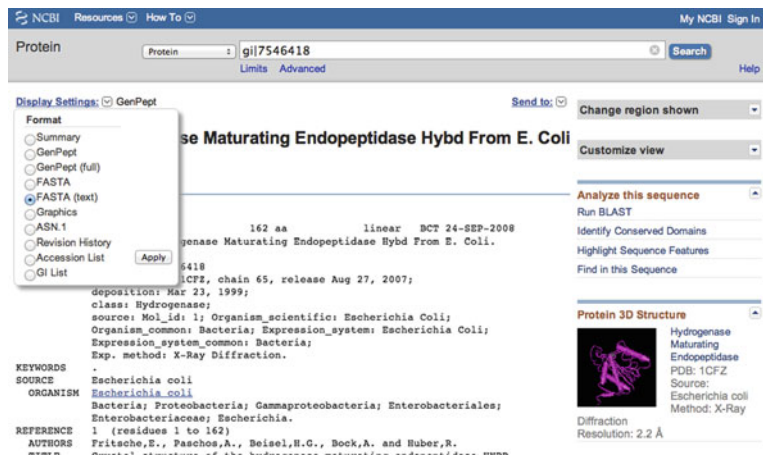


Fig. 19.3 Genbank. Download of the *E. coli* HybD protein sequence in FASTA format

19.3.2 Cyanobacterial BLAST I

Now, we are blasting the HybD protein sequence of *E. coli* against all three cyanobacterial proteomes with genomicBLAST (Fig. 19.5 on p. 380). To do so, goto http://www.ncbi.nlm.nih.gov/sutils/genom_table.cgi?, paste the *E. coli* HybD sequence into the query window, and set both “Query” and “Database” to protein. Further down on the same page select the following cyanobacteria: *Nostoc punctiforme* PCC 73102, *Nostoc* sp. PCC 7120, *Synechocystis* sp. PCC 6803. Now, click the “BLAST” button and on the next page the “View Report” button. The next page shows the BLAST result. Take a careful look at the results—I got 11 matching sequences (Fig. 19.4). You should select (click the check box) all cyanobacterial hits that contain either identical (same amino acid) or similar (amino acid with similar physicochemical properties, marked by an plus sign in the alignment) amino acids at the active site of *E. coli* HybD (NH₂ . . . [ED] . . . D . . . H . . . COOH). Click “Get selected sequences” and choose “Display” in “FASTA (text)”. Add these sequences to *hybd.fasta* and

Accession	Description	Max score	Total score	Query coverage	E value	Max ident
NP_485466.1	hydrogenase maturation protease [Nostoc sp. PCC 7120]	62.8	62.8	91%	3e-13	29%
YP_001864099.1	hydrogenase maturation protease [Nostoc punctiforme PCC 73102]	62.4	62.4	92%	5e-13	30%
NP_489166.1	hypothetical protein [Nostoc sp. PCC 7120]	32.7	32.7	60%	0.028	28%
NP_442995.1	dihydroxy-acid dehydratase [Synechocystis sp. PCC 6803]	28.5	28.5	64%	0.97	27%
NP_442887.1	shikimate 5-dehydrogenase [Synechocystis sp. PCC 6803]	27.7	27.7	58%	1.7	24%
NP_484813.1	hypothetical protein [Nostoc sp. PCC 7120]	26.2	26.2	22%	3.7	33%
YP_001864200.1	hypothetical protein Npun_R0480 [Nostoc punctiforme PCC 73102]	26.6	26.6	54%	4.1	28%
NP_487699.1	hypothetical protein [Nostoc sp. PCC 7120]	25.8	25.8	16%	6.9	46%
YP_001868276.1	chromate transporter [Nostoc punctiforme PCC 73102]	25.8	25.8	29%	7.3	33%
YP_001868236.1	multi-sensor hybrid histidine kinase [Nostoc punctiforme PCC 73102]	25.4	25.4	38%	9.4	31%
YP_001867111.1	dihydroxy-acid dehydratase [Nostoc punctiforme PCC 73102]	25.4	25.4	64%	10.0	25%

Fig. 19.4 genomicBLAST. BLAST query result for *E. coli* HybD against three selected cyanobacterial proteomes

NCBI **genomic BLAST**

BLAST Microbial Fungi Plants Insects Nematodes Environmental Protozoa Eukaryota Help

BLAST with microbial genomes (2460 bacterial/118 archaeal/427 eukaryotic genomes tree)

P - indicates the ability to search against protein sequences, - completed genomic sequence, - Whole Genome Shotgun, - add/remove from selection, See [Help] and [Article] for details.

Attention!
Because genome sequencing centers are now submitting unfinished genomes to the WGS division of DDBJ/EMBL/GenBank, the unfinished genomic sequences were removed from this page. We encourage any submitters of these sequences to submit their unfinished microbial genomes to WGS.

Enter your query sequence as Accession/GI or FASTA:

>HybD
MRILVLGVGNILLTDEAIGVRIVEALEQRYILPDYVEILDGCTAGMELLDGMANRDHLIIADAI
VSKKNAPGTMILRDEEVPALFTNKISHQLGLADVLSALRFTGEPKLTLCVGVIPESLEPHI

Select type of query and database or [BLAST-program](#)

Query: (Protein) Database: (Protein) Blast-program: (blastp) MegaBlast: ()

You may change BLAST options

Expect: (10) Filter: (default) Descriptions: (100) Alignments: (100)

Select all Clear all 3 genomes BLAST Adv. BLAST

Show alphabetical menu

Check ☐ if you want to select only completed genomes

Archaea

Fig. 19.5 BLAST I: Across Taxonomic Groups. Initiate a protein BLAST query with *E. coli* HybD against three selected cyanobacteria proteomes

rename them to either PCC7120, PCC6803, or PCC73102, according to their species origin.

19.3.3 Cyanobacterial BLAST II

In the previous section we blasted a protein sequence from *E. coli* against cyanobacteria. From a taxonomic point of view we compared sequences from proteobacteria and cyanobacteria. Both lineages diverged around 2.6 billion years ago. Thus, it might be difficult to find homologs sequences because of huge sequence differences. Bear in mind that, although the active site might be well conserved, the protein scaffold may be composed of very different amino acids. What to do? One easy approach is to reblast with a match from the target taxonomic group. In our case, we use the first matching cyanobacterial sequence (NP_485466.1). Going this way, I got seven matching sequences (Fig. 19.6).

This time, select (click the check box) all new hits that contain either identical or similar amino acids at the active site. There should be one. But there is also one interesting sequence (ID NP_441000.1) with only two matching active site amino acids (Fig. 19.7). The reason is that the matching D's (aspartic acid) share a very similar distance from the N-terminus. The C-terminal histidine is missing, which

Accession	Description	Max score	Total score	Query coverage	E value	Max ident	Links
NP_485466.1	hydrogenase maturation protease [Nostoc sp. PCC 7120]	316	316	100%	2e-111	100%	G
YP_001864099.1	hydrogenase maturation protease [Nostoc punctiforme PCC 6803]	272	272	98%	3e-94	85%	G
NP_441000.1	hypothetical protein slr1876 [Synechocystis sp. PCC 6803]	28.1	28.1	44%	0.82	30%	G
NP_485129.1	two-component response regulator [Nostoc sp. PCC 7120]	26.6	26.6	28%	3.3	23%	G
NP_484559.1	hypothetical protein [Nostoc sp. PCC 7120]	26.2	26.2	35%	4.0	33%	G
NP_488071.1	periplasmic-binding protein of ABC transporter [Nostoc sp. PCC 7120]	26.2	26.2	33%	4.2	31%	G
NP_484813.1	hypothetical protein [Nostoc sp. PCC 7120]	25.8	25.8	98%	4.5	25%	G

Fig. 19.6 BLAST II: Within One Taxonomic Group. BLAST query result for the best hit from BLAST I (NP_485466.1) against three selected cyanobacteria proteomes

```
>ref|NP_441000.1| hypothetical protein slr1876 [Synechocystis sp. PCC 6803]
ref|YP_005382866.1| slr1876 gene product [Synechocystis sp. PCC 6803 substr. GT-I]
ref|YP_005386035.1| slr1876 gene product [Synechocystis sp. PCC 6803 substr. PCC-P]
ref|YP_005408742.1| slr1876 gene product [Synechocystis sp. PCC 6803 substr. PCC-N]
ref|YP_005651057.1| slr1876 gene product [Synechocystis sp. PCC 6803]
Length=157

GENE_ID: 954313 slr1876 | hypothetical protein [Synechocystis sp. PCC 6803]
(10 or fewer PubMed links)

Score = 28.1 bits (61), Expect = 0.82, Method: Compositional matrix adjust.
Identities = 21/70 (30%), Positives = 31/70 (44%), Gaps = 6/70 (9%)

Query 4 I I G C G N L N R S D D A V G V I I A Q R L Q K Y L A E N P H P H V Q V Y D C G T A G M E V M F Q A R G S K Q L V I I D 63
I I G N R D D V G + A + + A + P H V E + + + I D
Sbjct 11 I I G Y N T L R G D D G V G R Y L A E E - - - I A Q Q N W P H C G V I S T H Q L T P E L A E A I A A V D R V I F I D 66

Query 64 A - - S S T G S E P 71
A + + E P
Sbjct 67 A Q L Q E S A N E P 76
```

Fig. 19.7 BLAST II. An not so obvious candidate: NP_441000.1. Note that the matching D's share a very similar distance from the N-terminus

might be attributed to a bad alignment of surrounding amino acids. Thus, let us add this sequence to our *hybd.fasta* file, too.

Your result file *hybd.fasta* should now contain the *E. coli* HybD sequence plus two cyanobacterial candidates from BLAST I and two additional candidates from BLAST II (see File 13).

File 13: *hybd.fasta*

```
1 >ecoli
2 MRILVLGVGNILLTDEAIGVRIVEALEQRYILPDYVEILDGGTAGMELLGDMANRDHLIIADAIVSKKNA
3 PGTMMILRDEEVPALFTNKIISPQHLGLADVLSALRFTGEFFPKKLTLVGVIPESLEPHIGLTPTEAMIEP
4 ALEQVLAALRESGVEAIPRSDS
5 >PCC7120-1
6 MLTIIGCGNLNRSDDAVGVI I A Q R L Q K Y L A E N P H P H V Q V Y D C G T A G M E V M F Q A R G S K Q L V I I D A S S T G S E
7 P G A V F K V P G E E L A A L P E P S Y N L H D F R W D H A L A A G R K I F P D D F P Q D V T V Y L I E A A N L D F G L E L S P V V Q Q S A
8 D L V V E K I V E I I R N
9 >PCC73102-1
10 M L T I I G C G N L N R S D D A V G V I I A Q H L Q K Y L A E N P H P Y V R V Y D C G T A G M E V M F Q A R G S Q Q L I I D A S S T G S E
11 P G A V F K V P G K E L E A L P E P S Y N L H D F R W D N A L A A G R K I F Q N D F P D D V T V Y L I E A A N L G L G L E L S P I V K H S A
12 D L V F E E V A A L I S Q N I N F
13 >PCC7120-2
14 M K K T V M V I G Y G N D L R S D D G I G Q R I A N E V A S W R L P S V E S L A V H Q L T P D L A D S L A S V D L A I F I D A C L P V H G F
15 D V K V Q L F A A G D I D S N V H T G D P R S L L A L T K A I Y G N C P T A W W V T I P G A N F E I G D R F S R T A E T G K A I A L V K I
16 I Q I L D K V N N L W F E V G A V A
17 >PCC6803-2
18 M P G Q S T K S T L I I G Y N T L R G D D G V G R Y L A E E I A Q Q N W P H C G V I S T H Q L T P E L A E A I A A V D R V I F I D A Q L Q
19 E S A N E P S V E V V A L K T L P E N E L S G D L G H R G N P R E L L T L A K I L Y G V E V K A W W V L I P A F T F D Y G E K L S P L T A R
20 A Q A E A L A Q I R P L V L G E R
```

Template	BLAST I			BLAST II		
	Proteobacteria → Cyanobacteria			Cyanobacteria → Cyanobacteria		
	Target	Match	E-Value	Target	Match	E-Value
<i>E. coli</i> HybD gii7546418	PCC 7120	NP_485466.1	3e-13	PCC 7120	NP_484813.1	4.5
	PCC 73102	YP_001864099.1	5e-13	PCC 73102	no new hit	---
	PCC 6803	no good hit	---	PCC 6803	NP_441000.1	0.82

Fig. 19.8 Summary of BLAST Results. Sequences marked in blue are used for further analysis

Figure 19.8 summarizes the results we got from both BLAST steps. You can see that it can make a huge difference if you blast between or within taxonomic units.

19.3.4 Compute Alignment and Tree Diagram

We now perform a global sequence alignment with ClustalW. Either you use your local installation (see Sect. 9.4 on p. 120) or goto <http://www.genome.jp/tools/clustalw/>—here I assume you work online. Paste the content of *hybd.fasta* (File 13 on the preceding) into the input window (or load your locally stored file), choose CLUSTAL as output format and press “Execute Multiple Alignment” (Fig. 19.9). Now, save the resulting global multiple sequence alignment into a text file named *hybd.align*. Based on the *E. coli* sequence, check if the active site amino acids align properly (Fig. 19.10).

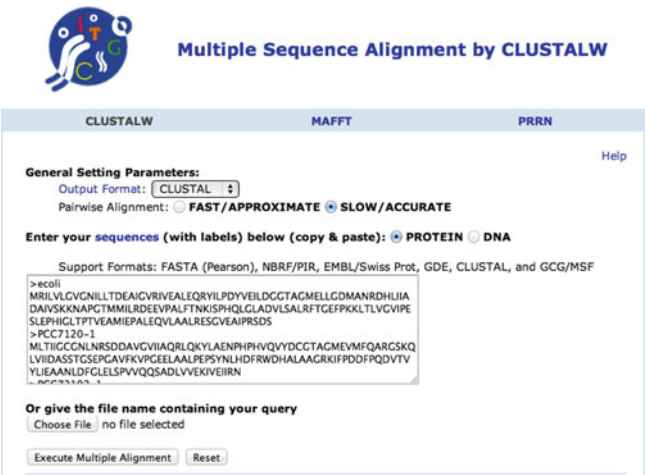


Fig. 19.9 Multiple Sequence Alignment. Screenshot of the ClustalW Web interface at the Kyoto University Bioinformatics Center

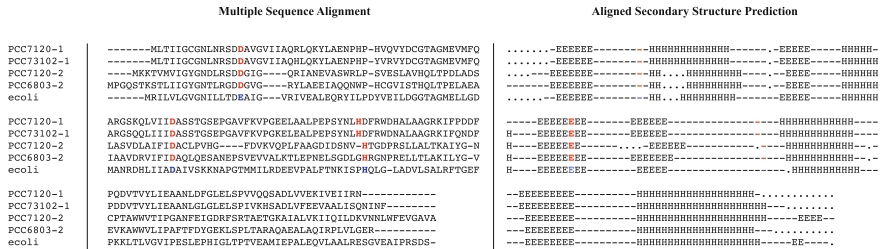


Fig. 19.10 Aligned Primary and Secondary Structure. The multiple sequence alignment was calculated with ClustalW. The secondary structure was predicted with JPred 3 and aligned on the basis of the ClustalW alignment manually. Position of the active site amino acids are highlighted. H: α -Helix; E: β -Sheet; dots in the secondary structure alignment mark manually added sequence gaps

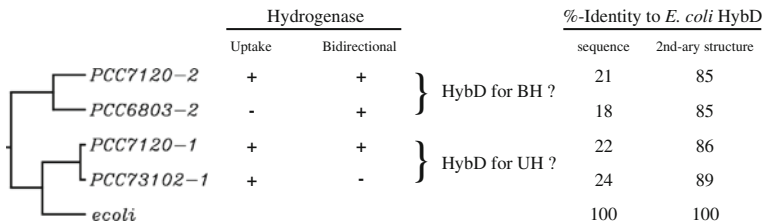


Fig. 19.11 UPGMA Tree. The UPGMA (Unweighted Pair Group Method with Arithmetic Mean) tree clusters all sequences according to their relatedness. In addition, the presence (+) or absence (–) of either the bidirectional (BH) or uptake (UH) hydrogenase is shown. The table on the right summarizes the results from Fig. 19.10. Identities have been calculated with the little AWK script in Program 65 on p. 385 as shown in Terminal 264 on p. 385

We can visualize the ClusterW result by printing a tree diagram. Choose “Rooted phylogenetic tree with branch length (UPGMA)” from the pull-down menu at the top of the page and click “Execute”. The resulting tree should look somewhat like the one shown in Fig. 19.11. The chosen UPGMA (Unweighted Pair Group Method with Arithmetic Mean) method assumes a molecular clock and sets a virtual least common ancestor, which is the root of the tree at the very left. The branch length indicate the relatedness of the sequences at the end nodes. It becomes obvious that PCC7120-1 and PCC73102-1 cluster together, meaning their sequences are very similar to each other. Both cluster with *E. coli* HybD, again indicating sequence similarity. PCC7120-2 and PCC6803-2 are more similar to themselves than to the other sequences, thus forming one cluster, too.

The clustering result allows us to generate an interesting hypothesis: One potential hydrogenase maturing peptidase from *Nostoc* PCC 7120 shows sequence similarity to the potential peptidase from *Synechocystis* PCC 6803 → a cyanobacterium only possessing the bidirectional hydrogenase). The other potential peptidase from *Nostoc* PCC 7120 shows sequence similarities to the potential enzyme from *Nostoc* PCC 73102 → a cyanobacterium only possessing the uptake hydrogenase). Do you see the

Fig. 19.12 Jpred. Screenshot of the Jpred Web interface

point? We can come up with the hypothesis that each hydrogenase in *Nostoc* PCC 7120 has its own maturation peptidase. This hypothesis has been experimentally supported (Wunschiers et al. 2003).

19.3.5 Secondary Structure Examination

The primary structure, i.e., amino acid sequence, gives us already an indication that the sequences we identified in cyanobacterial proteomes are good hydrogenase maturing peptidase candidates—despite bad BLAST E-values. But let us try to collect more evidence and strengthen our confidence by a comparison of the secondary structures. Protein secondary structure features such as α -helices and β -sheets can be pretty well predicted. Here we use Jpred, a web server (Fig. 19.12) that predicts secondary structure features on the basis of a trained neural network (Cole et al. 2008).

The simplest way for us to use Jpred, is by triggering a batch job in the advanced Jpred web interface. Goto <http://www.compbio.dundee.ac.uk/www-jpred/advanced.html> and use the “choose file” “button to select the file *hybd.fasta* (File 13 on p. 381). Select “Batch Mode” as input type, check the box “Skip searching PDB before prediction” and give your Email address. HybD would be a meaningful query name. Trigger the analysis by hitting the “Make Prediction” button.—Now, there is time to visit the rest room.—The results will be sent to your Email, usually within minutes, and look like this:

```

1      _____ Email Result from Jpred _____
2      Your job has completed successfully...
3
4      Query:  MLTIIGCGNLNRSDDAVGVIIAQHLQKYLAENPHFYVRVYDCGTAGMEVMFQARGSQQLI
5      Jpred:   -EEEEEE-----HHHHHHHHHHHHHH-----EEEEEE-----HHHHHHH---EEE
6      Conf:    4356540688887765278999999998614788874578744787457777644886478
7
8      Query:  IIDASSTGSEPGAVFKVPGKELEALPEPSYNLHDFRWDNALAAGRKIFQNDPDDVTVYL
9      Jpred:   EEEEE-----EEEEEE-----HHHHHHHHHHHH-----EEEEEE
10     Conf:    9888537888735888803677777777777763689999999885378884488988
11
12     Query:  IEAANLGLGLELSPIVKHSADLVFEEVAALISQNINF
13     Jpred:   EEE-----HHHHHHHHHHHHHHHHHHHHHH---
14     Conf:    86200677776326899999999999999999884099
15
16     The complete set of outputs can be viewed at:
17     http://www.compbio.dundee.ac.uk/www-jpred/cgi-bin/chklog?jp_TEttI6x

```

For Fig. 19.10 on p. 383 I edited the resulting predictions by formatting them in the same way as the ClustalW alignment. We can see that the active site amino acids are positioned in topologically similar regions and that the secondary structure features are very similar for all sequences. How can we quantify this similarity? Let us employ a very crude but effective method: we simply compare the sequences or secondary structure assignments as if they were text strings and count the identities for each position. This requires that the strings have identical length, which is the reason why I aligned the secondary structures in Fig. 19.10 on p. 383. Program 65 shows one possible solution.

```

1      _____ Program 65: compare-strings.awk - String Comparison _____
2      # USAGE awk -f compare-strings.awk ' string1' ' string2'
3      # NOTE the leading SPACE characters in the embracing single quotes
4      BEGIN{
5          a = ARGV[1]; b = ARGV[2]
6          len_a = length(a)-1; len_b = length(b)-1
7          sum_match = 0; sum_mismatch = 0
8          if(len_a!=len_b){print "ERROR: UNEQUAL STRING LENGTH"; exit}
9          for(i=1;i<=len_a;i++){
10             ai = substr(a,i+1,1); bi = substr(b,i+1,1)
11             if(ai == bi){sum_match++} else{sum_mismatch++}
12         }
13         print "LENGTH = "len_a
14         print "MISMATCHES = "sum_mismatch ("sum_mismatch/len_a*100%")
15         print "MATCHES = "sum_match ("summatch/len_a*100%")
16     }

```

Terminal 264 shows the application of Program 65. Note that lines 1–6 represent one real line. The strings have been concatenated from the ClustalW result (Fig. 19.10 on p. 383). Note also that the strings are embraced by single quotes and are preceded by a space character. This prevents the interpretation of leading dashes as AWK parameters.

```

1      _____ Terminal 264: Calculating String Identities _____
2      $ awk -f compare-strings.awk ' -----MRILVLGVGNILLTDEAIG--VRIVEALEQRYILPD
3      VVEILDGGTAGMELLGDMANRDHLIIADAIVSKKNAPGTMILRDEEVPALEPNKISPHQLG-LADVLSALRFT
4      GEFPKKLTLVGVIPESLEPHIGLTPTEVAMIEPALEQVLAALRESGVEAIPRSDS-' ' MPGQSTKSTLIIGY
5      GNTLRGDDGVG-----RYLAEEIAQGNWP-HCGVISTHQLTPELAEATAAVDRVIFIDAQLQESANEPSVEVVAL

```

```

5 | KTLPENELSGDLGHRGNPRELLTLAKILYG-VEVKAWWVLIPTFTFDYGEKLSPLTARAQAEALAQIRPLVLGE |
6 | R-----' |
7 | LENGTH = 174 |
8 | MISMATCHES = 143 (82.1839%) |
9 | MATCHES = 31 (17.8161%) |
10 | $ |

```

When you compare all cyanobacterial sequences against the *E. coli* sequence as reference, you should get numbers as in Fig. 19.10 on p. 383 (at the right side). Obviously, the secondary structures show a much higher identity than the amino acid sequences. What about the tertiary structures?

19.3.6 Tertiary Structure Alignment (Homology Modeling)

The ultimate step before entering the laboratory and do wet science is to test, if we can align the potential cyanobacterial hydrogenase maturing peptidases to the known structure of the enzyme from *E. coli* (Fritsche et al. 1999). Such a tertiary structure alignment is also called homology modeling. We are going to use the Swiss-Model server at the University of Basel, Switzerland.

Goto <http://swissmodel.expasy.org> and follow the link “Automated Mode”. Figure 19.13 shows the input page. You must submit each cyanobacterial candidate sequence individually. Choose chain A from structure 1CFZ as template. This is the structure from Fritsche et al. (1999). Due to the crystal geometry, the structure 1CFZ contains six identical monomers—called protein chains. Therefore, we restrict the modeling to one single chain.

SwissModel Automatic Modelling Mode

Email:

Project Title:

Provide a protein sequence or a UniProt AC Code:

>PCC7120-1
 MLTIICCCNLNRSDDAVGVIAQLKQYLAENPHPHVQVYDCCTAGMEVMFQARGSKQLVIIDASSTCSE
 PCAVFKVPCGEALALPEPSYNLHDFRWDHALAACRKIFPDFFQDVTVYUEAANLDFGLESPVVQQSA
 DLVVEKIVEIIRN

Advanced options:

Use a specific template: ☒ PDB-ID: Chain:

or

Template file: ☒ no file selected

Fig. 19.13 Swiss-Model. Screenshot of Swiss-Model Input Form

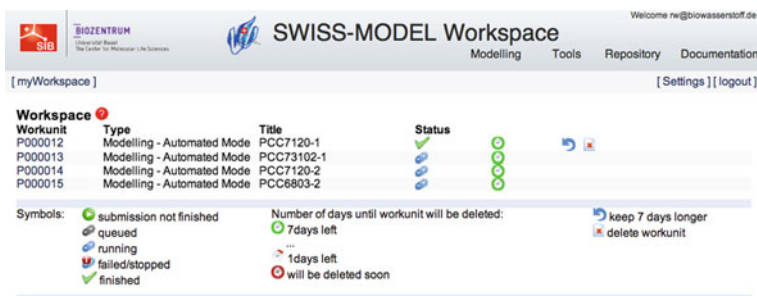


Fig. 19.14 Screenshot of the Swiss-Model Workspace. My workspace holds four modeling projects. One of them has already been completed

After submitting the modeling request, Swiss-Model automatically creates an individual workspace for you (Fig. 19.14). You will receive an information Email with the password. Results are kept for one week on the Swiss-Model server, the workspace itself will never expire.

The predicted structure of completed jobs can be downloaded as PDB file from the result page (Fig. 19.15). Download all four structures and save them as, e.g., pcc7120-1.pdb.

19.3.7 Download and Visualize *E. coli* HybD Structure

Although we worked a lot with the amino acid sequence of HybD and used it as a template for homology modeling, we did not take a look at its structure yet. You learned already that the structure has a rather strange name 1CFZ. This is in fact its PDB ID, i.e., its identifiers at the world largest structure database (PDB = Protein Data Bank). You can download the structure file (PDB file) directly from PDB: <http://www.rcsb.org>. We, however, go another way. The molecular structure viewer Jmol can directly connect to PDB and load a structure file. Let us start by starting Jmol. After downloading Jmol you should have a folder named *jmol-12.2.32*, where 12.2.32 indicates the version number. Assuming that *jmol-12.2.32* is a subfolder of your present working directory, type

```
java -jar jmol-12.2.32/Jmol.jar &
```

in the terminal. This will open Jmol in a separate window in the background (see Sect. 8.13.3 on p. 111). Now, open the menu “File” → “Get PDB” and enter “1CFZ”. You can execute commands from the menu. However, as an advanced user you might go beyond the menu panel and use the rich command language from the Jmol console instead (see Appendix A.6 on p. 430). Then open menu “File” → “Console...”.

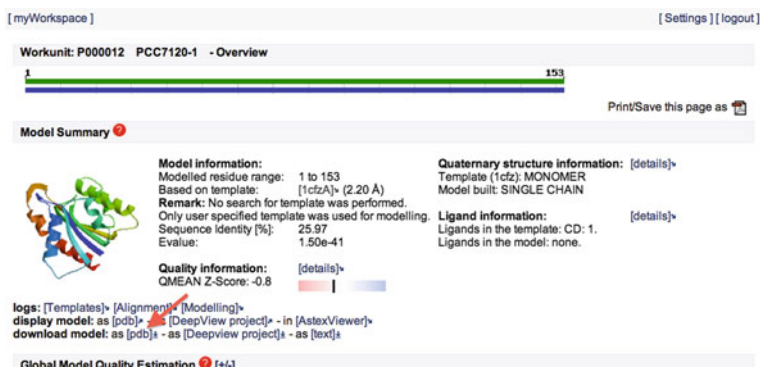


Fig. 19.15 Screenshot of the Swiss-Model Result Page. You should download the PDB file, which is marked by an arrow

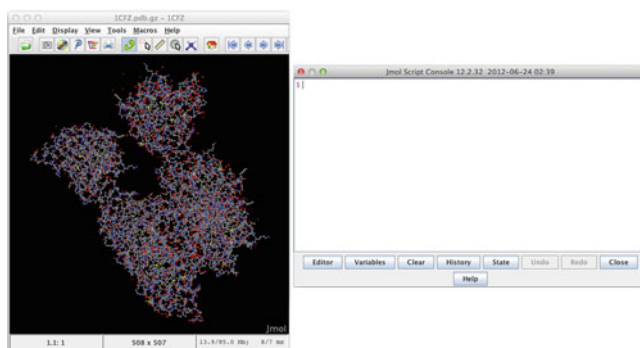


Fig. 19.16 Jmol. PDF file 1CFZ loaded

A second window should open that looks like a Linux terminal—this is a terminal that allows you to control Jmol (Fig. 19.16).

With the following commands you can modify the view at the structure (everything behind the #-sign must not be typed, it is only an explanation of the commands):

```

1  restrict :A                # only show chain A
2  center :A                 # center the view
3  cartoon                   # show secondary structure features
4  spacefill off             # hide filled atoms
5  wireframe off             # hide atom bonds
6  color structure           # color by structure feature
7  select GLU16:A, ASP62:A, HIS93:A # select particular amino acids
8  spacefill 120             # show atoms of selected amino acids
9  wireframe 40              # show bonds of selected amino acids
10 color cpk                 # color by atom identity
11 select CD:A               # select cadmium in chain A
12 spacefill 150             # show selected atom
13 color orange              # color selected atom

```

This looks much better. We can now see the overall structure plus the active site amino acids. With `select CD:A; spacefill 150` we highlight the cadmium atom. In vivo there is a nickel atom. During the crystallization process this has been replaced by cadmium. The nickel atom is coordinated by the active site amino acids and delivered to the maturing hydrogenase prior to proteolytic cleavage of an N-terminal polypeptide. Well, this is the template. How do our cyanobacterial candidates obtained from Swiss-Model look like?

19.3.8 View and Compare Cyanobacterial Structures

Let us now look at our cyanobacterial candidates with Jmol. With open menu “File” → “Open” you can choose a file to visualize. With the commands `cartoon`, `spacefill off` and `wireframe off` you can adopt the view at the structure to our needs. And now select the active site amino acids. **Attention!**—The amino acids are not the same and not at the same position as in *1CFZ.pdb*. Use Fig. 19.10 on p. 383 for reading the exact positions. Additionally, you can execute `show sequence` in the Jmol console to view the current sequence.

After loading all sequences one by one get an impression of how well the structures fit to the *E. coli* template (Fig. 19.17 on the p. 390). But let us go one step further and get out more from Jmol—let us load all structures at once—wow! For this last step in our analysis I assume that you saved the structures obtained by the tertiary structure alignment with Swiss-Model as *pcc7120-1.pdb*, *pcc7120-2.pdb*, *pcc73102-1.pdb* and *pcc6803-2.pdb*, respectively. I further assume that *Jmol.jar* sits in the same folder.

Let us start by creating a Jmol script file named *overlay.jmol* as shown in File 14.

```

File 14: overlay.jmol
1  # execute from Jmol console with "script overlay.jmol"
2
3  load 1CFZ.pdb                # becomes model 1.1
4  load append pcc6803-2.pdb    # becomes model 2.1
5  load append pcc7120-1.pdb    # becomes model 3.1
6  load append pcc7120-2.pdb    # becomes model 4.1
7  load append pcc73102-1.pdb   # becomes model 5.1
8
9  select all
10 cartoon; spacefill off; wireframe off # format for all structures
11
12 select GLU16:A/1.1, ASP62:A/1.1, HIS93:A/1.1, ASP22/2.1, ASP66/2.1, HIS97/2.1,
13 ASP15/3.1, ASP63/3.1, HIS93/3.1, ASP18/4.1, ASP62/4.1, HIS88/4.1, ASP15/5.1,
14 ASP63/5.1, HIS93/5.1
15 spacefill 120; wireframe 40      # format for active site amino acids
16
17 select 1.1                      # select 1CFZ
18 restrict :A; center :A
19 color green
20 select CD:A/1.1                 # select cadmium in chain A
21 spacefill 150; color gray
22
23 select 2.1; color blue          # select PCC6803-2 and paint blue

```

```

24 select 3.1; color yellow           # select PCC7120-1 and paint yellow
25 select 4.1; color orange          # select PCC7120-2 and paint orange
26 select 5.1; color red             # select PCC73102-1 and paint red
27
28 models 0                           # overlay all structures

```

This script is first loading the reference structure 1CFZ and subsequently appending the cyanobacterial structures to the workspace (lines 3–7 in File 14). We then select all structures and format them as cartoons (lines 9–10). Then we select the active site amino acids. We can address them by the construct NamePosition:Chain/Model.Frame. Thus, ASP62:A/1.1 selects aspartate at position 62 in chain A of structure 1.1. As mentioned above, structure 1CFZ contains several identical chains but we restrict ourselves to chain A (line 18). In lines 23–26 we give each cyanobacterial structure an individual color and the command in line 28 initiates the overlaid display.

Open Jmol in the background (see Sect. 8.13.3 on p. 111) with `java -jar Jmol.jar` & from within the folder where you saved the PDB-files. Next, open the Jmol terminal via menu “File” → “Console...”. And now comes the magic step: type `script overlay.jmol` in the Jmol console and hit (Enter). Jmol will now work through all commands in the script file `overlay.jmol` and finish with an overlay of all structures (Fig. 19.17, bottom left image). The nickel (cadmium) atom held by the active site amino acids is shown in gray. Each structure

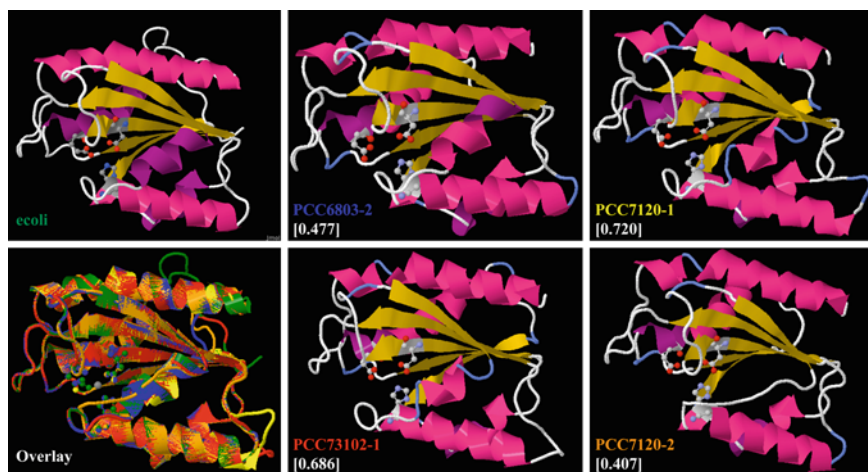


Fig. 19.17 Jmol. *E. coli* HybD (ecoli) and potential homologs from *Nostoc punctiforme* PCC 73102, *Nostoc* sp. PCC 7120, and *Synechocystis* sp. PCC 6803. All structures are shown with detailed active site amino acids. The score printed beneath each name (QMEAN4 score) is an estimate of the reliability of the model (1: good, 0: bad). For the overlay all structures were colored as indicated by their name color and overlaid as described in the text

has an individual color. With the following commands you can instruct Jmol to loop through all structures with one second delay between each sequence:

```

1  animation fps 1          # sets frame rate to 1 frame per sec
2  animation mode loop      # restart after each run
3  animation on             # start animation
4
5  animation off            # stop animation
6  models 0                 # show all structures again

```

Take a minute to recall what we did. We blasted the hydrogenase maturing peptidase HybD from *E. coli* against three cyanobacterial proteomes. From two satisfying hits we took the best and used this to blast again against the selected cyanobacteria. We obtained two more satisfying sequences. The E-values ranged from $3e-13$ to 4.5. By looking at the primary, secondary, and tertiary structure alignment we incremented our confidence that cyanobacteria contain homologs to HybD. The structure overlay shows an astonishing conserved 3D-structure albeit the amino acids sequences are very different.

Chapter 20

Virtual Sequencing of pUC18c

20.1 The Project

The objective of this exercise is to sequence the pUC18c cloning vector. It is divided into the following major parts: (a) downloading the cloning vector, (b) run a virtual sequencing program, and (c) assembling the virtual sequences obtained from sequencing.

20.2 Bioinformatic Tools

In this project you will apply several software tools which are freely available. All programs can be downloaded from the web page accompanying this book: <http://www.staff.hs-mittweida.de/~wuenschi/doku.php?id=rwbook2>.

20.2.1 Sequencer

`sequencer` is a virtual sequencing machine developed by Bernhard Haubold. You supply the software with a genome sequence and tell it the amount of coverage and average length of sequence reads you wish to obtain. Thus, it simulates the performance of a DNA-sequencer (average sequence length) and the effort you put into lab work (coverage).

20.2.2 Assembler

The TIGR Assembler is a classic open-access assembly tool developed by *The Institute for Genomic Research* (TIGR). Its objective is to build all possible consensus

sequences (contigs) from smaller sequence fragments. These fragments typically come from genome sequencing.

20.2.3 Dotter

Dotter is a graphical dot plot program for detailed comparison of two sequences (Sonnhammer and Durbin 1995). Every nucleotide or amino acid from one sequence is compared to every other sequence. The first sequence runs along the *x*-axis and the second sequence along the *y*-axis. In regions where the two sequences are similar to each other, a row of high scores will run diagonally across the dot matrix. If you are comparing a sequence against itself to find internal repeats, you will notice that the main diagonal scores maximally, since it's the 100% perfect self-match. To make the scoring matrix more intelligible, the pairwise scores are averaged over a sliding window which runs diagonally. The averaged score matrix forms a three-dimensional landscape, with the two sequences in two dimensions and the height of the peaks in the third. This landscape is projected onto two dimensions by aid of grayscales—the darker gray a peak is, the higher it is.

Dotter provides a tool to explore the visual appearance of this landscape, as well as a tool to examine the sequence alignment it represents.

20.2.4 GenBank

As an example for sequencing we use the cloning vector pUC18c. As all open accessible sequences the sequence for this vector is available from GenBank hosted by the National Center for Biological Information (NCBI) at <http://www.ncbi.nlm.nih.gov/gquery/gquery.fcgi>.

20.3 Detailed Instructions

The first step is to download all sequences and software we use in this practical. Start by opening a terminal window and create in your home directory a directory called *Sequencing*. Then change into the newly created directory.

```
Terminal 265: Make Directory with mkdir
1  $ cd
2  $ pwd
3  /home/rw
4  $ mkdir Sequencing
5  $ cd Sequencing/
6  $ pwd
7  /home/rw/Sequencing
8  $
```

You should download all files into this directory.

20.3.1 Download the *pUC18c* Sequence

Open a web browser and go to the NCBI homepage: <http://www.ncbi.nlm.nih.gov>. Set the Search pull-down menu to *Nucleotide* and enter *pUC18c*. Choose the entry L09136 called “pUC18c cloning vector”. Click at this entry and choose “FASTA” in the Display pull-down menu and “this page to text” in the Send pull-down menu and click “Send”. Now save the file in text format in the *Sequencing* directory as *puc18c.seq*.

20.3.2 Download and Setup the Virtual Sequencer

The virtual sequencing software tool can be downloaded from the book’s webpage (see above) or via *wget* (see Sect. 6.1.1 on p. 71). You should save this file as *sequencer* in the *Sequencing* directory. In order to make this file executable use the command *chmod* (change modus) as shown in Terminal 266.

```

1  $ ls -lh
2  total 24K
3  -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
4  -rw-rw-r-- 1 rw rw 18K May 18 2006 sequencer
5  $ chmod u+x sequencer
6  $ ls -lh
7  total 24K
8  -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
9  -rwxrw-r-- 1 rw rw 18K May 18 2006 sequencer
10 $

```

There is no need to further install the software. You can immediately run the software as shown in Terminal 267.

```

1  $ ./sequencer
2  sequencer version 1.0, copyright (c) 2004, Bernhard Haubold
3  purpose: simulate shotgun sequencing
4  arguments:
5  -i input file in FASTA format
6  [-o output file; default: \textit{stdout}]
7  [-c coverage; default: 1.0]
8  [-l mean fragment length; default: 500]
9  [-r circular genome?; default: linear]
10 [-s seed for random number generator (< 0); default: system clock]
11 [-n length of output lines; default: 50]
12 [-h print help; default: do not print help]
13 $

```

In case you do not get the same output as the lines shown in Terminal 267, you have a problem.

20.3.3 Download and Install the TIGR Assembler

Installing the TIGR Assembler is a little bit more tricky. First of all it requires the C-Shell (see Sect. 8.2 on p. 98), which is different from the Bash Shell we have been working with. Terminals 268 and 269 show you how to check if the C-Shell is already installed and how to install it.

```

1  $ csh
2  % exit
3  % exit
4  $

```

Terminal 268: C-Shell: Already Installed?

For the latter case I assume you run Ubuntu Linux as described in Sect. 4.1.2 on p. 40).

```

1  $ csh
2  The program 'csh' can be found in the following packages:
3  * csh
4  * tcsh
5  Try: sudo apt-get install <selected package>
6  $ sudo apt-get install csh
7  [sudo] password for rw:
8  Reading package lists... Done
9  ...
10 $

```

Terminal 269: C-Shell: Installation

To install the actual TIGR Assembler, first download the software from the book's webpage (see above) or via `wget` (see Sect. 6.1.1 on p. 71) into the *Sequencing* directory. Like BLAST (see Sect. 18.3.1.2 on p. 349), the TIGR Assembler comes as a compressed and packed file archive. This is visible from the file extensions `.tar.gz`. The extension `.tar` means that several files have been combined to a file archive using the `tar` command. The extension `.gz` indicates that this archive has been compressed using the program `gzip`. Terminal 270 shows you how to recover the original files.

```

1  $ ls -lh
2  total 3.3M
3  -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
4  -rwxrw-r-- 1 rw rw 18K May 18 2006 sequencer
5  -rwxrwx--- 1 rw rw 3.3M Aug 3 09:50 TIGR_Assembler_v2.tar.gz
6  $ gunzip TIGR_Assembler_v2.tar.gz
7  $ tar -xf TIGR_Assembler_v2.tar
8  $ ls -lh
9  total 17M
10 -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
11 -rwxrw-r-- 1 rw rw 18K May 18 2006 sequencer
12 drwxr-xr-x 6 rw rw 4.0K Apr 21 2003 TIGR_Assembler_v2
13 -rwxrwx--- 1 rw rw 17M Aug 3 09:50 TIGR_Assembler_v2.tar
14 $ cd TIGR_Assembler_v2/
15 $ ls -lh
16 total 32K
17 drwxr-xr-x 2 rw rw 4.0K Mar 27 2000 bin
18 -rwxrwxr-x 1 rw rw 114 Apr 21 2003 COPYRIGHT

```

Terminal 270: Unpacking TIGR Assembler


```

19 | drwxr-xr-x 4 rw rw 4.0K Mar 27 2000 data
20 | -rw-rw-r-- 1 rw rw 5.1K Apr 21 2003 LICENSE
21 | drwxr-xr-x 2 rw rw 4.0K Mar 27 2000 obj
22 | -rw-rw-r-- 1 rw rw 2.9K Apr 21 2003 README
23 | drwxr-xr-x 2 rw rw 4.0K Apr 21 2003 src
24 | $

```

Now comes the critical step: installation of the software. All steps necessary to install TIGR Assembler are explained in the *README* file. For your convenience I go through the installation procedure in Terminal 271.

```

----- Terminal 271: Installing TIGR Assembler -----
1 | $ cd src/
2 | $ pwd
3 | /home/rw/Sequencing/TIGR_Assembler_v2/src
4 | $ make
5 | cc -O -c ./asmg.c -o ../obj/asmg.o
6 | In file included from asmg.c:12:
7 | matrices.h:163:8: warning: extra tokens at end of #endif directive
8 | In file included from asmg.c:13:
9 | ...
10 | $ cd ..
11 | $ pwd
12 | /home/rw/Sequencing/TIGR_Assembler_v2
13 | $ ls
14 | bin COPYRIGHT data LICENSE obj README src
15 | $ PATH=$(pwd)/bin:$PATH
16 | $ export PATH
17 | $ cd ~/Sequencing/
18 | $ pwd
19 | /home/rw/Sequencing
20 | $ run_TA
21 | shift: No more words.
22 | $

```

In line 1 of Terminal 271 we change into the source file directory. The compilation of the program is initiated with the `make` command in line 4. What follows are many lines of status information which is not interesting for us. I truncated those lines in Terminal 271. In lines 15–16 we create a new system path and export it to the system. This is necessary in order to be able to run the program from everywhere. With the command in line 17 we go back to the Sequencing directory and check whether the compilation was successful by typing the `run_TA` command in line 20. Since we do not get any error message we can tell the program runs.

20.3.4 Download and Setup Dotter

Finally, we need to install the Dotter software tool. It allows us to draw dot plots. Download the program from the book's webpage (see above) or via `wget` (see Sect. 6.1.1 on p. 71) into the *Sequencing* directory. Alternatively, you can download the program from the developers homepage (<http://sonnhammer.sbc.su.se/download/software/dotter/dotter.LIN>) in which case you should change its name from *dotter.LIN* to *dotter*. Luckily, we directly download an executable file and need no

installation procedure. First we change the name of the file *dotter.LIN* (which indicates that this executable has been compiled for Linux systems) to *dotter* and make this file executable. This is shown in lines 10–11 in Terminal 272.

```

1  $ pwd
2  /home/rw/Sequencing
3  $ wget "http://www.staff.hs-mittweida.de/wuenschi/data/media/
4      compbiolbook/dotter"
5  Resolving www.staff.hs-mittweida.de (www.staff.hs-mittweida.de)...
6      141.55.192.74
7  ...
8  Length: 4804333 (4.6M) [text/plain]
9  Saving to: 'dotter'
10 100%[=====] 4,804,333  1.03M/s  in 5.0s
11 2012-08-03 11:15:58 (945 KB/s) - 'dotter' saved [4804333/4804333]
12 $ ls -lh
13 total 22M
14 -rw-rw-r-- 1 rw rw 4.6M May 18 2006 dotter
15 -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
16 -rwxrwr-- 1 rw rw 18K May 18 2006 sequencer
17 drwxr-xr-x 6 rw rw 4.0K Apr 21 2003 TIGR_Assembler_v2
18 -rwxrwx--- 1 rw rw 17M Aug 3 09:50 TIGR_Assembler_v2.tar
19 $ chmod u+x dotter
20 $ ./dotter
21
22 Dotter - Sequence dotplots with image enhancement tools.
23
24 Reference: Sonnhammer ELL & Durbin R (1995). A dot-matrix program
25 with dynamic threshold control suited for genomic DNA and protein
26 sequence analysis. Gene 167(2):GC1-10.
27
28 Usage: dotter [options] <horizontal_sequence> <vertical_sequence>
29         [X options]
30
31 Allowed types:
32
33         Protein      -      Protein
34         DNA          -      DNA
35         DNA          -      Protein
36
37 Options:
38
39 -b <file>      Batch mode, write dotplot to <file>
40 -l <file>      Load dotplot from <file>
41 -m <float>     Memory usage limit in Mb (default 0.5)
42 -z <int>      Set zoom (compression) factor
43 -p <int>      Set pixel factor manually (ratio pixelvalue/score)
44 -W <int>      Set sliding window size. (K => Karlin/Altschul estimate)
45 -M <file>     Read in score matrix from <file>
46               (Blast format; Default: Blosum62).
47 -F <file>     Read in sequences and data from <file>
48               (replaces sequencefiles).
49 -f <file>     Read feature segments from <file>
50 -H            Do not calculate dotplot at startup.
51 -R            Reversed Greyramp tool at start.
52 -r            Reverse and complement horizontal_sequence (DNA vs Protein)
53 -D            Don't display mirror image in self comparisons
54 -w            For DNA: horizontal_sequence top strand only (Watson)
55 -c            For DNA: horizontal_sequence bottom strand only (Crick)
56 -q <int>     Horizontal_sequence offset
57 -s <int>     Vertical_sequence offset
58 ...
59 Version 3.1, compiled Dec 4 2002
60 $

```

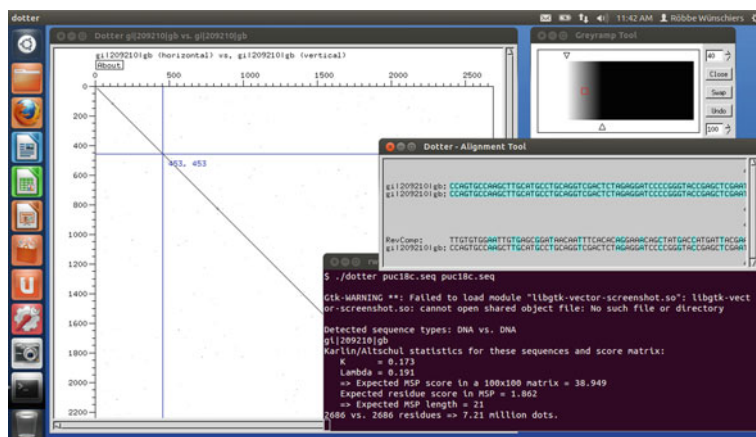


Fig. 20.1 Dot Plot. Dot plotting the pUC18c cloning vector sequence against itself

In line 18 we execute Dotter by typing `./dotter`. You should see many lines of output indicating the usage of the program. In Terminal 272 I truncated some lines. In case you get a warning like: *Gtk-WARNING **: Failed to load module “libgtk-vector-screenshot.so”: libgtk-vector-screenshot.so: cannot open shared object file: No such file or directory*, just ignore it.

Now let us draw a dot plot of the pUC18c vector against itself by typing `./dotter puc18c.seq puc18c.seq` in the terminal. Three windows with content as shown in Fig. 20.1 should open.

These windows show the dot plot itself, the Alignment Tool and Grayramp Tool. By holding the left mouse button and moving the mouse you can position the crosshair in the dot plot. Synchronously, the Alignment Tool shows the alignment for the actual crosshair position.

20.3.5 Virtual Sequencing

The simulation of the sequencing process is not only of educational value. In fact, it is used in order to optimize software for sequence assembly. We simulate sequencing of the pUC18c DNA sequence using the software *sequencer*. This program offers many parameters in order to influence the result. We are only using three of them. Start with the example in Terminal 273.

```

Terminal 273: Using sequencer
1  $ pwd
2  /home/rw/Sequencing
3  $ ./sequencer -c 1 -l 800 -i puc18c.seq
4  >Seq_1_696
5  ATTTTATAGTTAATGTCATGATAATAATGGTTTCTTAGACGTACGGTG
6  ...

```

```

7 GCCTTGATCGTTGGGAACCGAGCTGAATGAAGCCATACCAAACGA
8 >Seq_4_788
9 CGATCAAGCGAGTTACATGATCCCCCATGTTGTGCAAAAAAGCGTTAG
10 ...
11 CTATAAAAATAGGCGTATCACGAGGCCCTTTCGTC
12 $ ./sequencer -c 5 -l 800 -i puc18c.seq
13 >Seq_1_797
14 AACTCGTGC CGC CATACTATTCTCAGAATGACTTGGTTGAGTACTCA
15 ...
16 >Seq_19_822
17 CTTGATCCGGCAAACAAACCACGCTGGTAGCGGTGGTTTTTTTGTTCG
18 ...
19 ACTGGTGAGTACTCAACCAAGT
20 $

```

Again, I had to truncate the output in Terminal 273. The sequences coming from the virtual sequencer are in FASTA format and named according to the sequence number and its length. Thus, the name `>Seq_1_696` tells us that this is sequence number 1 which is 696 nt long. The command `./sequencer -c 1 -l 800 -i puc18c.seq` in line 1 of Terminal 273 yields 4 sequences, the command `./sequencer -c 5 -l 800 -i puc18c.seq` in line 12 yields 19 sequences. Since `sequencer` is based on chance, you might get another result. The average length of the sequences is 800 nt, which is a realistic assumption for Sanger sequencing. Let us take a look at the different options.

- c Coverage; how often shall each nucleotide on average be sequenced?
- l Length; sets the average length of the sequences.
- i Input; names the input file containing the DNA sequence in FASTA format.
- o Output; names an output file where the result is saved. If no output file is given, the result will be shown on the screen.

The higher the coverage, the more sequences you should get. This is illustrated in Fig. 20.2.

Next, we run `sequencer` again and save the results in a file called *output*. Then we visualize the sequencing fragments together with the pUC18c sequence

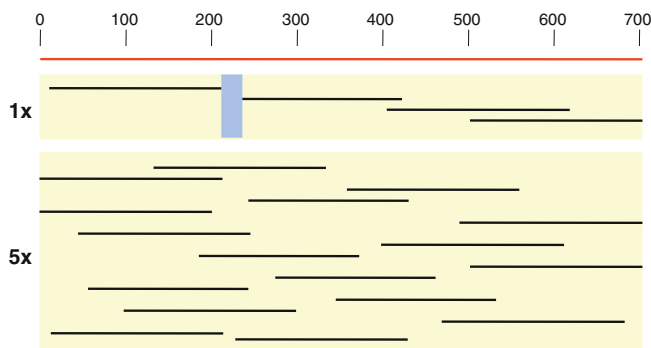


Fig. 20.2 Sequencing Fragments. Sequencing of a 700bp DNA segment with 1-time and 5-times coverage. The sequence length has an average of 200nt. Note that there is a sequencing gap (blue)

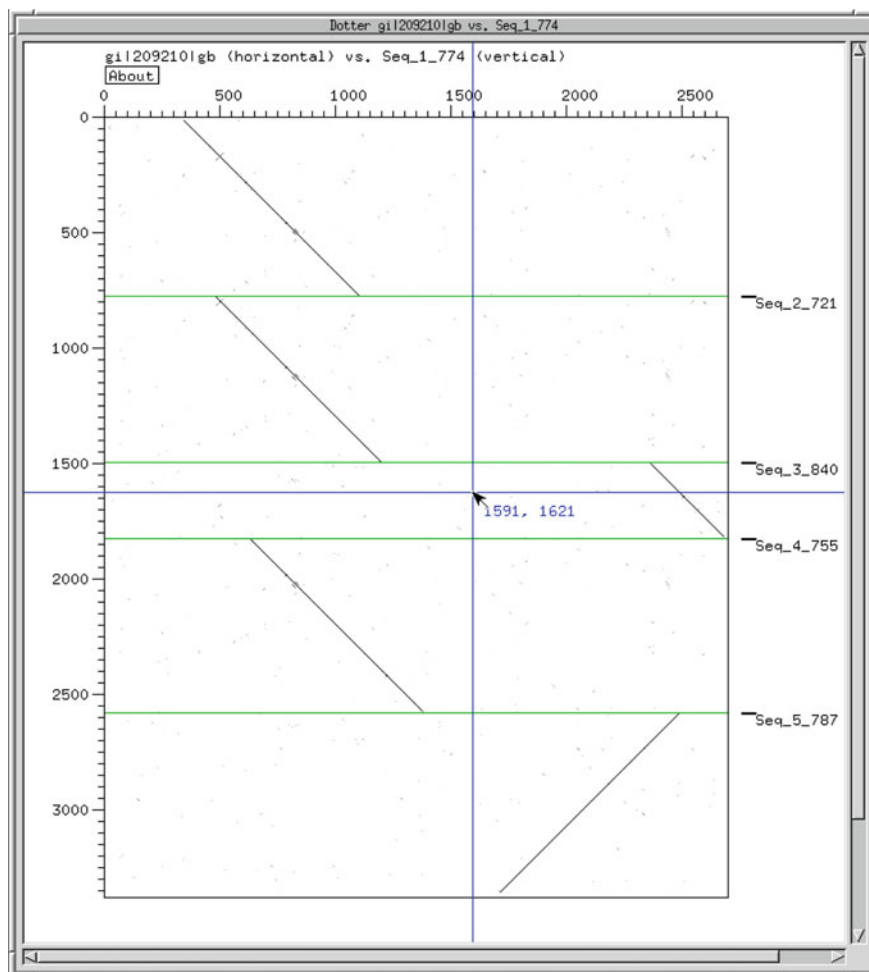


Fig. 20.3 Dot Plot. Dot plot of the pUC18c cloning vector (x-axis) versus 5 sequencing fragments (y-axis)

with `dotter`. The procedure is shown in Terminal 274 and one possible result is illustrated in Fig. 20.3.

```

Terminal 274: Visualizing Sequencing Result
1  $ pwd
2  /home/rw/Sequencing
3  $ ./sequencer -c 1 -l 800 -i puc18c.seq -o output
4  $ ./dotter puc18c.seq output
5
6  Detected sequence types: DNA vs. DNA gi|209210|gb Karlin/Altschul
7  statistics for these sequences and score matrix:
8      K      = 0.173
9      Lambda = 0.191
10     => Expected MSP score in a 100x100 matrix = 38.973

```

```

11      Expected residue score in MSP = 1.861
12      => Expected MSP length = 21
13      2686 vs. 3364 residues => 18.07 million dots.
14
15      $

```

You only get back to the command line prompt after quitting *dotter*.

The diagonal lines in Fig. 20.3 indicate DNA fragments generated by sequencer that are perfectly matching to the pUC18c DNA sequence. Positive slopes of the lines indicate sequences from the reverse DNA strand and vice versa.

20.3.6 Sequence Assembly

In line 3 in Terminal 274 we generated sequencing fragments and saved them in a file called *output*. Now let us try to assemble these sequences in order to recover the original pUC18c sequence. These assembled sequence fragments are called contigs (contiguous sequences). It is the task of creating contigs (optimally only one single contig) from overlapping DNA fragments, TIGR Assembler was designed for. We call the assembler with the command *run_TA* as shown in Terminal 275.

```

_____ Terminal 275: Running the TIGR Assembler _____
1  $ ls -lh
2  total 22M
3  -rwxrw-r-- 1 rw rw 4.6M May 18 2006 dotter
4  -rw-rw-r-- 1 rw rw 3.4K Aug 3 12:00 output
5  -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
6  -rwxrw-r-- 1 rw rw 18K May 18 2006 sequencer
7  drwxr-xr-x 6 rw rw 4.0K Apr 21 2003 TIGR_Assembler_v2
8  -rwxrwx--- 1 rw rw 17M Aug 3 09:50 TIGR_Assembler_v2.tar
9  $ run_TA output
10 $ ls -lh
11 total 22M
12 -rwxrw-r-- 1 rw rw 4.6M May 18 2006 dotter
13 -rw-rw-r-- 1 rw rw 3.4K Aug 3 12:00 output
14 drwxr-x--- 2 rw rw 4.0K Aug 3 12:01 output.align
15 -rw-rw-r-- 1 rw rw 11K Aug 3 12:01 output.asm
16 -rw-rw-r-- 1 rw rw 87K Aug 3 12:01 output.error
17 -rw-rw-r-- 1 rw rw 1.9K Aug 3 12:01 output.fasta
18 -rw-rw-r-- 1 rw rw 2.8K Jul 31 18:17 puc18c.seq
19 -rwxrw-r-- 1 rw rw 18K May 18 2006 sequencer
20 drwxr-xr-x 6 rw rw 4.0K Apr 21 2003 TIGR_Assembler_v2
21 -rwxrwx--- 1 rw rw 17M Aug 3 09:50 TIGR_Assembler_v2.tar
22 $

```

In line 1 in Terminal 275 we first list the content of the *Sequencing* directory. In my case there are 6 unique files. Then, in line 9, we assemble the sequences contained in the file *output* by calling *run_TA output*. If you receive an error message, you probably need to add the TIGR Assembler to the system path again. Type

```
PATH=~ /Sequencing/TIGR_Assembler_v2/bin:$PATH; export PATH
```

and retry.

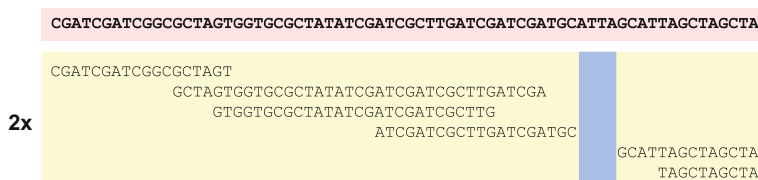


Fig. 20.4 Sequence Assembly. Schematic representation of the assembly of sequences by overlapping matches. The average coverage is around 2-times. There is a small gap (blue) of 4 nt that has not been covered by sequencing

After successful execution of the TIGR Assembler you got three new files (*output.asm*, *output.error*, and *output.fasta*) and one new directory (*output.align*). We are only interested in the file *output.fasta* which contains all successfully assembled fragments. The ultimate goal is to get only one contig that exactly matches the original pUC18c DNA sequence. The basis of building the contigs are overlapping sequence fragments as illustrated in Fig. 20.4.

We can visualize the success of sequence assembly by loading the pUC18c sequence and the contigs to *dotter*. The corresponding command is

```
./dotter puc18c.seq output.fasta
```

The result is shown in Fig. 20.5 on the following page.

20.3.7 Testing Different Parameters

Up to now we always used a coverage of 1 and a sequence length of 800 nt for the virtual sequencing reaction with *sequencer*. How do these parameters influence the quality of sequencing? A good indicator for assessing the quality of the sequencing process is the number of contigs obtained by the TIGR Assembler. The ultimate goal is to get just one contig. The actual number of contigs can easily be retrieved from the number of entries in the file named *output.fasta*. Since the sequences are saved in FASTA format, each sequence name begins with the ">" character. The frequency of those can be counted with the *grep* command as shown in Terminal 276.

```
Terminal 276: Counting Sequences
1 $ grep -c ">" output
2 5
3 $ grep -c ">" output.fasta
4 2
5 $
```

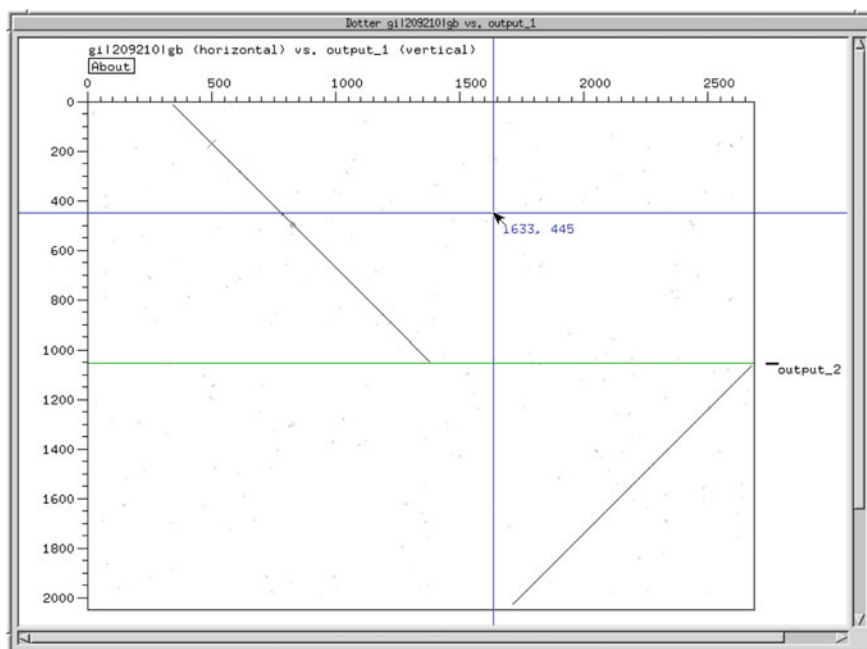


Fig. 20.5 Dot Plot. Dot plotting the pUC18c cloning vector sequence versus assembled sequences (contigs)

In the example in Terminal 276 we got five sequence fragments from the `sequencer` program which could be assembled to two contiguous sequences. Your task now is to check the influence of coverage and sequence length (parameters `-c` and `-l`, respectively) on the number of contigs you get. Which parameter combinations yield just one contig? Generate a plot like the one shown in Fig. 20.6 but use more different sequence lengths between 100 and 400 nt. You do that by changing the number following parameter `-l`.

You can automate the process by connecting several command in one line as shown in Terminal 277.

```

Terminal 277: Counting Sequences
1  $ ./sequencer -c 1 -l 800 -i puc18c.seq -o output;
2    run_TA output; grep -c ">" output output.fasta
3  output:4
4  output.fasta:2
5  $ ./sequencer -c 1 -l 800 -i puc18c.seq -o output;
6    run_TA output; grep -c ">" output output.fasta
7  output:5
8  output.fasta:2
9  $

```

Please note that both lines 1–2 and 5–6 are in fact one line!

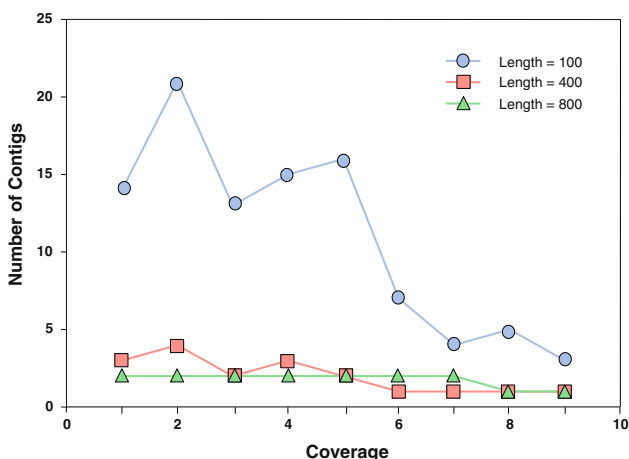


Fig. 20.6 Sequence Assembly. Relation between coverage, sequence length (nt), and number of assembled sequences (contigs)

Now, let us play a bit more with parameters. Save the following Program 66 as *counter.sh* in the *Sequencing* directory and make it executable using the *chmod* command.

```

1  Program 66: counter.sh - Counting Sequences Automatically
2  for l in 100 400 800; do
3    echo "----- Sequence Length: $l -----"
4    for i in 1 2 3 4 5 6 7 8 9; do
5      ./sequencer -c $i -l $l -i puc18c.seq -o output
6      run_TA output
7      SEQ1=$(grep -c ">" output)
8      SEQ2=$(grep -c ">" output.fasta)
9      echo "Coverage $i: $SEQ1 seq - $SEQ2 contigs"
10   done
11 done

```

Make sure that the path variable is set correctly and run the script by typing *./counter.sh*. It generates the data shown in Terminal 278 that I used for Fig. 20.6.

```

1  Terminal 278: Result of counter.sh
2  $ PATH=~ /CompBiol/SeqNew/TIGR_Assembler_v2/bin:$PATH; export PATH
3  $ ./counter.sh # | egrep -v "Aborted" > counter.out
4  ----- Sequence Length: 100 -----
5  Coverage 1: 29 seq - 12 contigs
6  ...
7  Coverage 9: 251 seq - 2 contigs
8  ----- Sequence Length: 400 -----
9  Coverage 1: 7 seq - 2 contigs
10 ...
11 Coverage 9: 66 seq - 1 contigs
12 ----- Sequence Length: 800 -----

```

```

12 Coverage 1: 5 seq - 0 contigs
13 ...
14 Coverage 9: 40 seq - 0 contigs
15 $

```

Again, I truncated the output in Terminal 278. For each combination of sequence length and coverage ten individual virtual sequencing runs are initiated. This should help to show you that the results differ slightly due to the randomness of sequencer.

At some Linux installations the TIGR Assembler throws a lot of errors—but still works. If you observe this (you will see a lot of characters at the screen), you should pipe the output of `counter.sh` to `egrep` as shown in the commented part of line 2 in Terminal 278.

20.3.8 Visualizing Results with R

For the visualization of the effect of sequence length and coverage on the number of assembled contigs we employ R (see Chap. 17 on p. 317). To make things a bit easier we change the output of our script `counter.sh`. The new script is shown in Program 67 and should be saved as `counter2.sh`. Again, do not forget to make the file executable.

```

Program 67: counter2.sh - Counting Sequences Automatically
1 echo "length coverage sequences contigs"
2 for l in 100 400 800; do
3   for i in 1 2 3 4 5 6 7 8 9; do
4     ./sequencer -c $i -l $l -i puc18c.seq -o output
5     run_TA output
6     SEQ1=$(grep -c ">" output)
7     SEQ2=$(grep -c ">" output.fasta)
8     echo $l $i $SEQ1 $SEQ2
9   done
10 done

```

The main difference between the old and the new bash script is the reduced number of virtual sequencing runs and the format of the output. You should redirect the output of `counter2.sh` to a file named `counter.out`. Now open R to plot the data as shown in Terminal 279.

```

Terminal 279: Plotting Result of counter2.sh in R
1 $ ./counter2.sh > counter.out
2 $ R
3 R version 2.14.1 (2011-12-22)
4 ...
5 > data <- read.table("counter.out", header=T, sep=" ")
6 > attach(data)
7 > plot(contigs ~ coverage, subset = length == 100, pch=1, col="red")
8 > points(contigs ~ coverage, subset = length == 400, pch=2, col="blue")
9 > points(contigs ~ coverage, subset = length == 800, pch=3, col="green")
10 > legend("topright", c("100nt", "400nt", "800nt"),
11     col=c("red", "blue", "green"), pch=1:3)
12 >

```

In line 2 in Terminal 279 we open R. In line 5 the data stored in file *counter.out* is loaded into the vector *data*. Then the header line is made accessible as variable name with the *attach* command. Finally, we plot the data in three steps. In each step a subset of the data is plotted with individual symbol (*pch*) and color (*col*) settings. The *legend* command in line 10 adds a legend to the plot, which should look like Fig. 20.7.

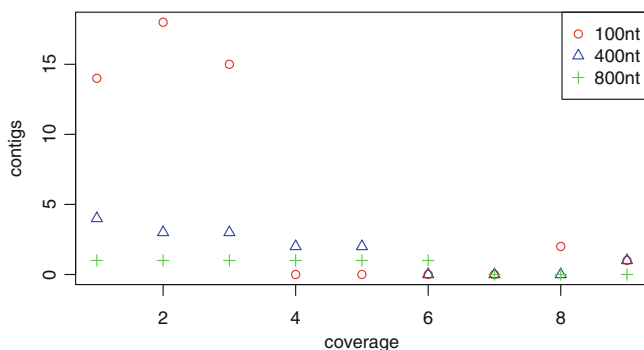


Fig. 20.7 Sequence Assembly. Visualization of the results obtained with Program 67 on the preceding page. The R commands are shown in Terminal 279. Note that you might get different results due to the random simulation by *sequencer*

Chapter 21

Querying for Potential Redox-Regulated Enzymes

21.1 The Project

Now, let us look at another real-world example: the search for topologically close cysteines in protein structure files. The scientific background of this project is biochemistry. The activity of some enzymes is regulated by formation or cleavage of disulfide bonds near the protein surface. This can be catalyzed by a group of proteins known as thioredoxins (Trx) (Fig. 21.1).

Thioredoxins are small proteins with a redox-active disulfide bond present in the characteristic active site amino acid sequence Trp-Cys-Gly-Pro-Cys. They have a molecular mass of approximately 12 kDa and are universally distributed in animal, plant, and bacterial cells. As stated above, the target enzymes, which are regulated by Trx, possess two cysteine residues in close proximity to each other and to the surface. Comparison of primary structures reveals that there is no cysteine-containing consensus motif present in most of the thioredoxin-regulated target enzymes (Schürmann 2000).

Years ago, we found a correlation between the concentration of reduced Trx and the activity of a hydrogenase (Wünschiers et al. 1999). In order to identify potential cysteines that could be targets for Trx, one can nowadays investigate protein structure data of crystallized hydrogenases. We just need to search for cysteine sulfur atoms that are no further than 3 Å apart and close to the surface. But how?

21.2 Bioinformatic Tools and Resources

21.2.1 RCSB PDB

The Research Collaboratory for Structural Bioinformatics (RCSB) is a collaboration between three US-American institutions (RUTGERS: The State University of New Jersey, UCSD: University of California, San Diego, and the University

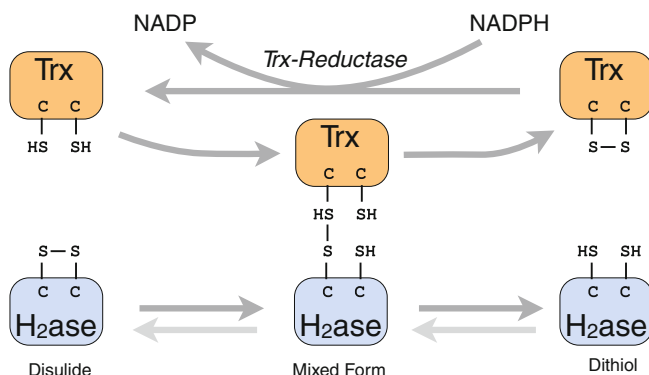


Fig. 21.1 Thioredoxin (Trx). Thioredoxins catalyze the reduction of protein disulfides to dithiol, thereby affecting the activity of the target enzyme. Thioredoxin is recycled by the action of thioredoxin reductase

of Wisconsin-Madison) that jointly run the Protein Data Bank (PDB). As of 10 April 2012, it contains 80,710 structures of biomolecules. Among those are 42,692 enzyme structures (with 39,678 structures having a resolution below 3 Å).

21.2.2 Jmol

Jmol is a free, open source molecule structure viewer. Since it is based on Java, it runs on all operating systems that can run the Java Runtime Environment (see <http://www.java.com>). A short overview of Jmol's commands is given in Appendix A.6 on p. 430. Jmol can be accessed and downloaded here: <http://jmol.sourceforge.net/>. Alternatively, Ubuntu users may use the shortcut shown in Sect. 4.1.2.5 on p. 43.

21.2.3 Surface Racer

Surface Racer is a tool that calculates the accessible surface area (also solvent-accessible surface area) of macromolecules (Tsodikov et al. 2002). The accessible surface is defined by the center of a probing sphere, when it rolls over the surface of the macromolecule (Fig. 21.2). You can download the program at http://apps.phar.umich.edu/tsodikovlab/index_files/Page756.htm.

Surface Racer comes as a compressed file archive containing three files: the binary executable, a *readme.txt*, and a data file called *radii.txt*.

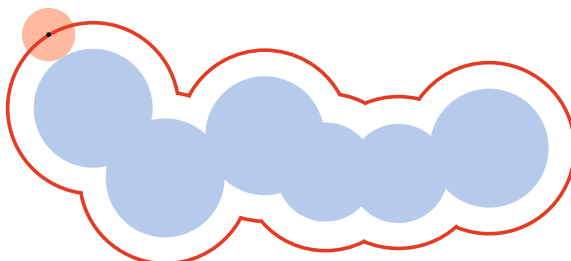


Fig. 21.2 Solvent-Accessible Surface Area. Two-dimensional representation of a probing circle (*red*) that rolls over from (*blue*). The center (*black*) of the circle describes a contour (*red line*). Transferred to a 3D macromolecule, the circle would become a sphere and the contour would become a surface

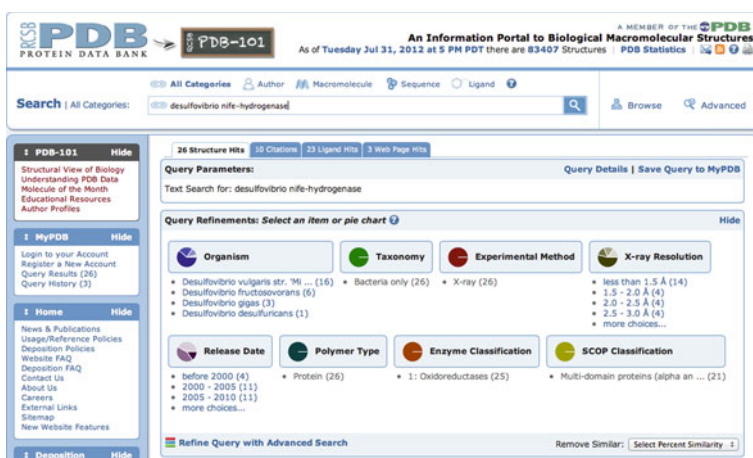


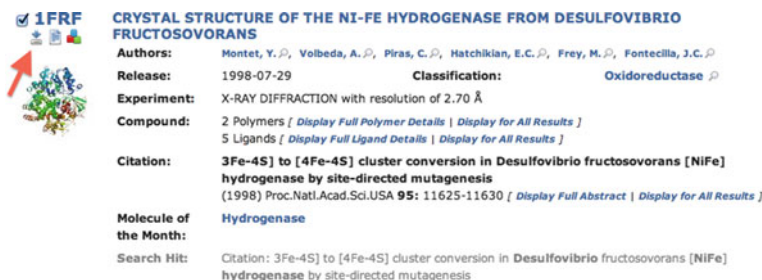
Fig. 21.3 Protein Data Bank. A query for “Desulfovibrio NiFe-Hydrogenase” yields 26 structure hits

21.3 Detailed Instructions

21.3.1 Download Crystal Structure Files

To start with, we are going to download two NiFe-hydrogenase crystal structure files from the RCSB. Therefore, go to <http://www.rcsb.org/pdb/> and query for “Desulfovibrio NiFe-Hydrogenase” (see Fig. 21.3). There are many structure files available. Download 1FRF from *Desulfovibrio fructosovorans* (see Fig. 21.4 on the following page) and 1FRV from *Desulfovibrio gigas* in PDB format to a subfolder of your home directory called *Redox*.

Again, you could also download the files directly from the command line into your present working directory using *wget* (see Sect. 6.1.1 on p. 71). The URL is



1FRF

CRYSTAL STRUCTURE OF THE NI-Fe HYDROGENASE FROM DESULFOVIBRIO FRUCTOSOVORANS

Authors: Montet, Y. *et al.*, Volbeda, A. *et al.*, Piras, C. *et al.*, Hatchikian, E.C. *et al.*, Frey, M. *et al.*, Fontecilla, J.C. *et al.*

Release: 1998-07-29 **Classification:** Oxidoreductase *et al.*

Experiment: X-RAY DIFFRACTION with resolution of 2.70 Å

Compound: 2 Polymers [[Display Full Polymer Details](#) | [Display for All Results](#)]
5 Ligands [[Display Full Ligand Details](#) | [Display for All Results](#)]

Citation: **3Fe-4S] to [4Fe-4S] cluster conversion in Desulfovibrio fructosovorans [NIFe] hydrogenase by site-directed mutagenesis**
(1998) Proc.Natl.Acad.Sci.USA **95**: 11625-11630 [[Display Full Abstract](#) | [Display for All Results](#)]

Molecule of the Month: [Hydrogenase](#)

Search Hit: Citation: 3Fe-4S] to [4Fe-4S] cluster conversion in Desulfovibrio fructosovorans [NIFe] hydrogenase by site-directed mutagenesis

Fig. 21.4 Protein Data Bank. Clicking the small icon indicated by the *arrow* initiates download of the PDB file

<http://www.rcsb.org/pdb/download/downloadFile.do?fileFormat=pdb&compression=NO&structureId=XXXX>, where XXXX must be replaced by the four letter PDB identifier.

21.3.2 Analyze Structures Visually

Let us start with a first visual inspection of the structure of a thioredoxin-regulated hydrogenase. As explained earlier in Sect. 19.3.7 on p. 387, you can directly download a PDB structure file from within Jmol—if you have an Internet connection. Therefore, open Jmol in the background (see Sect. 8.13.3 on p. 111) by typing

```
java -jar jmol-12.2.32/Jmol.jar &
```

in the terminal (I am assuming that *jmol-12.2.32* is a subfolder of your present working directory). In Jmol, open the menu “File” → “Get PDB” and enter “1FRF”. Jmol should now present you the structure. Now, open the Jmol-terminal by opening menu “File” → “Console...” (see Appendix A.6 on p. 430). You should see something like in Fig. 21.5 on the facing page, but with another structure loaded.

Now, search for cysteine sulfur atoms that are close to the surface. Use the (SHIFT) key together with the left mouse button and move the mouse in order to zoom. — The following commands will help you to highlight sulfur atoms of cysteines and measure their distance. Note that you have to click at one atom (you might like to zoom in to exactly hit the desired atom) in order to get the atom number required for the `monitor` command.

	Jmol Commands for 1FRF
1	<code>show pdbheader</code> # displays the PDB file header
2	<code>select all</code> # select all atoms
3	<code>spacefill 50</code> # reduce atom size
4	<code>select cys and sulfur</code> # select cysteine sulfur atoms
5	<code>spacefill 250</code> # increase atom size of selection
6	<code>monitor 3914 5273</code> # measures distance between atoms #3914 and #5273

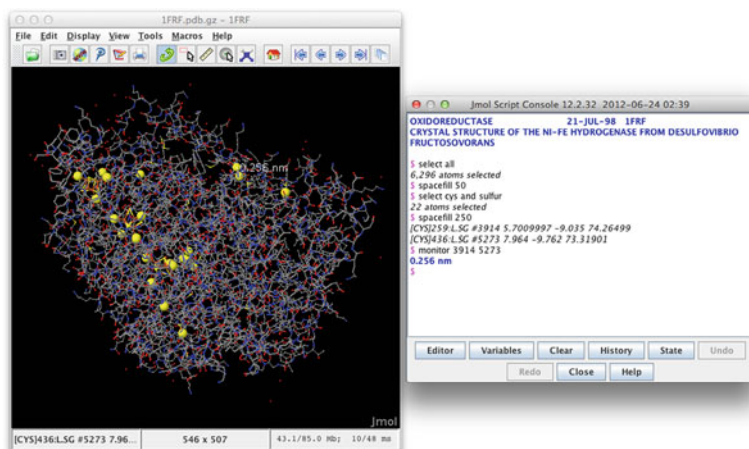


Fig. 21.5 Jmol. Hydrogenase structure 1FRF has been loaded and the view adopted by the commands shown in the Jmol terminal. The distance of cysteine sulfur atoms close to the surface is shown

I hope it becomes clear that searching for potential Trx targets is a rather tedious job. Let us see how AWK can help us.

21.3.3 Analyze Structures Computationally

Before we think about a computational solution, we have to deal with the overall structure of a PDB structure file. In Sect. 11.3 on p. 160, you have already acquired an insight into the organization of files containing structural information. Figure 21.6 on the next page illustrates its general structure. The interesting lines for us look like this:

\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\$10	\$11	\$12
ATOM	5273	SG	CYS	L	436	7.96	9.76	73.31	0.75	14.37	S

The top line describes the field variables as AWK sees the data, while the bottom line shows the actual atomic data in PDB format. The first field (\$1) says that the line contains the information about an atom. The unique atom identity (atom number) is given in the second field (\$2). Fields \$3–\$6 tell us that the atom is the γ -sulfur atom of cysteine, which is the 436th amino acid of protein chain L. Fields \$7–\$9 contain the x-, y-, and z-coordinates of the atom in Å, respectively. Field \$12 classifies the atom as a sulfur atom (S).

Okay, with this information in hand, the approach to nail down our problem should be clear! We catch all lines where \$1 matches ATOM, \$4 matches CYS, and \$12 matches S (we could likewise look for \$3 matching SG), respectively. A decent script

000000000111111112222222222333333333334444444445555555556666666667777777778							
12345678901234567890123456789012345678901234567890123456789012345678901234567890							
ATOM	5272	CB	CYS L 436	7.735	-11.476	73.371	0.75 7.04 C
ATOM	5273	SG	CYS L 436	7.964	-9.762	73.319	0.75 14.37 S
ATOM	5274	N	ALA L 437	7.150	-14.355	74.467	1.00 18.99 N

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	atom	Line content annotation
7 - 11	Integer	serial	Atom serial number
13 - 16	Atom	name	Atom name
17	Character	altLoc	Alternate location indicator
18 - 20	Residue name	resName	Residue name
22	Character	chainID	Chain identifier
23 - 26	Integer	resSeq	Residue sequence number
27	AChar	iCode	Code for insertion of residues
31 - 38	Real(8.3)	x	X coordinates in Angstroms
39 - 46	Real(8.3)	y	Y coordinates in Angstroms
47 - 54	Real(8.3)	z	Z coordinates in Angstroms
55 - 60	Real(6.2)	occupancy	Occupancy
61 - 66	Real(6.2)	tempFactor	Temperature factor
73 - 76	LString(4)	segID	Segment identifier, left-justified
77 - 78	LString(2)	element	Element symbol, right-justified
79 - 80	LString(2)	charge	Charge on the atom

Fig. 21.6 PDB-File Content. When you look at a PDB file in a text editor you will see a lot of information. Every PDB file is presented in a number of lines and each line in the PDB entry file consists of 80 columns. Generally, each line starts with a maximum *six* character string that specifies what the data content of the line is. For our task, we are interested in all lines that contain atom coordinates of cysteine sulfur atoms

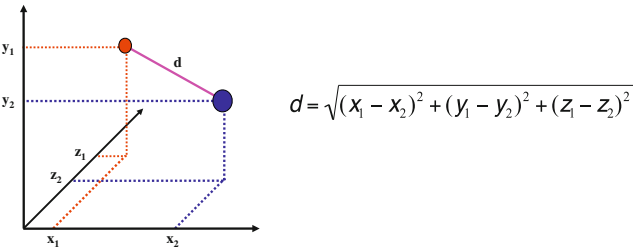


Fig. 21.7 Euclidean Distance. Coordinates of *two* points in the 3D space and the corresponding calculation

to catch these lines is

```
awk ' $1=="ATOM" && $4=="CYS" && $12=="S" ' 1FRF.pdb
```

where *1FRF.pdb* is the structure file. From the matching lines, we extract the coordinates and save them together with the unique atom identifier \$2. This can be done employing an array:

```
cys_x[$2] = $7; cys_y[$2] = $8; cys_z[$2] = $9
```

Finally, we calculate the distance in space of all saved sulfur atoms against all. We are lucky that the coordinates are given in Å; but which formula should be applied to calculate the distance? Well, do you remember your math classes? Figure 21.7 illustrates the magic formula together with a graphical representation of the coordinates of two points (atoms) in the 3D space.

Now, we are still missing two things: the program and the structure files. You should test two hydrogenase structures for the existence of nearby cysteines: the structure with the PDB identifier 1FRF from the bacterium *Desulfovibrio fructosovorans* and the structure with the identifier 1FRV from the bacterium *Desulfovibrio gigas*. Both structure files can be downloaded from the PDB (<http://www.rcsb.org>). If you prefer to download the sequence via the terminal with `wget`, take a look at Terminal 280

```

1  $ wget "http://www.rcsb.org/pdb/cgi/export.cgi/1FRV.pdb"
2  --2012-08-31 13:11:50-- http://www.rcsb.org/pdb/cgi/export.cgi/1FRV.pdb
3  Resolving www.rcsb.org (www.rcsb.org)... 198.202.122.51
4  Connecting to www.rcsb.org (www.rcsb.org)|198.202.122.51|:80... connected.
5  HTTP request sent, awaiting response... 200 OK
6  Length: unspecified [text/plain]
7  Saving to: '1FRV.pdb'
8
9  [          <=>          ] 1,091,718    403K/s   in 2.6s
10
11 2012-08-31 13:11:53 (403 KB/s) - '1FRV.pdb' saved [1091718]
12
13 $

```

Save these files in the same directory as the AWK script. Now, it is time to take a look at exactly this script.

```

1  # save as distance.awk
2  # searches close cysteine sulfur atoms in a structure
3  # requires a structure file (*.pdb)
4  # usage: awk -f distance.awk structure.pdb
5
6  BEGIN{print "Cysteines in the Structure..."; ORS="" }
7
8  $1=="ATOM" && $4=="CYS" && $12=="S" {
9  print $4$6, "
10 cys_x[$6]=$7; cys_y[$6]=$8; cys_z[$6]=$9
11 }
12
13 END{ ORS="\n"
14 for (key1 in cys_x) {
15     for (key2 in cys_x) {
16         dx=cys_x[key1]-cys_x[key2]
17         dy=cys_y[key1]-cys_y[key2]
18         dz=cys_z[key1]-cys_z[key2]
19         distance=sqrt(dx^2+dy^2+dz^2)
20         if (distance < 3 && distance != 0) {
21             i++
22             candidate[i]=key1-"key2": "distance
23         }
24     }
25 }
26 print "\nCandidates ..."
27 for (keys in candidate) {print candidate[keys]}
28 }

```

Down to line 7, nothing important is happening in Program 68—except that we set the output field separator to nothing (the default setting is line break). Line 8 contains the search pattern. As said above, we are looking for all lines in the protein

structure file where the first field ($\$1$) matches “ATOM”, the fourth field ($\$4$) matches “CYS” like cysteine, and the last field ($\$12$) matches “S” like sulfur. Note: It is very important that you place the opening brace for the action body at the end of the line containing the search pattern (line 8) and not at a new line! Otherwise, the program will behave strangely. With line 9, we print out all cysteines in the file together with their unique atom ID ($\$6$). In line 10, we use the unique atom ID as key for three arrays, *cys_x*, *cys_y*, and *cys_z*, containing the coordinates. Note that lines 8–11 are executed for each line of the PDB file, while the BEGIN and END block are only executed once. The END body constitutes the main part of the program performing the distance calculation. Here, we apply two nested `for` loops in order to compare each atom with each other. Remember that the `for` loop returns the array indices (or keys) in an arbitrary unordered way (see also Sect. 13.5.4.3 on p. 217). In line 14, we pick one index from the array *cys_x* and save it in the variable *key1*. The `for` loop in line 15 picks an index from the array *cys_x* too, and saves it in the variable *key2*. Lines 16–19 are devoted to the Euclidean distance calculation with the ultimate distance of the current atom pair stored in variable *distance*. If the distance of the current pair of sulfur atoms is less than 3 Å (line 20) and unequal to 0 (avoids self-to-self comparisons), we add the corresponding unique atom identifier pair to the array *candidate* in line 22. When you carefully analyze the current algorithm, you will recognize that we compare each pair of sulfur atoms twice. Thus, we will calculate the distance of atoms A and B and later of atoms B and A. Exercise 21.1 on p. 422 deals with this problem. Finally, in lines 26–28 of Program 68, we print out the distance values of the candidate array.

Terminal 281 shows the script in action.

```

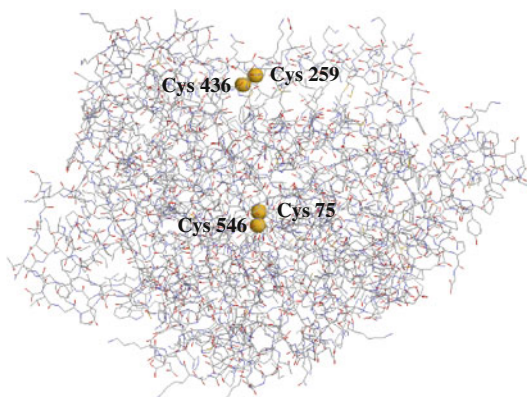
1  $ awk -f distance.awk 1FRF.pdb
2  Cysteines in the Structure...
3  CYS17, CYS20, CYS98, CYS110, CYS114, CYS147, CYS187, CYS212, CYS218, CYS227,
4  CYS245, CYS248, CYS72, CYS75, CYS86, CYS237, CYS259, CYS265, CYS436, CYS457,
5  CYS543, CYS546,
6  Candidates ...
7  436-259: 2.55824
8  75-546: 2.41976
9  546-75: 2.41976
10 259-436: 2.55824
11 $

```

As can be seen, there are 22 cysteines in the structure. Lines 8–11 in Terminal 281 tell us that the sulfur atoms of cysteines 436 and 259, and 75 and 546 are less than 3 Å apart. However, we do not yet know whether these sulfur atoms are close to the protein surface. To check this, we need the molecular structure viewer Jmol again. Figure 21.8 shows the structure of *Desulfovibrio fructosovorans* hydrogenase with highlighted candidate sulfur atoms.

To obtain the view shown in Fig. 21.8, execute the following commands in the Jmol terminal after loading the structure 1FRF:

Fig. 21.8 Potential Thioredoxin Targets in 1FRF. Crystal structure of the NiFe-hydrogenase from *Desulfovibrio fructosovorans* with highlighted candidate sulfur atoms



Highlighting Candidate Sulfur Atoms	
1	hide water # remove water molecules
2	spacefill off # reduce atom spheres
3	select cys436.sg, cys259.sg, cys75.sg, cys546.sg # select gamma-sulfur atoms
4	spacefill 300 # enlarge gamma-sulfur atoms

By turning the molecule around its axis in Jmol, it becomes clear that only the sulfur atoms of cyteines 259 and 436 are at the surface of the protein. Thus, these are the potential targets for Trx. This would be the point to turn from computational biology to experimental biology, i.e., leave the computer and do the experiment...

21.3.4 Expand the Analysis to Many Proteins

If one likes to analyze several protein structure files, the above method becomes time demanding. In this section, we are expanding the approach to the analysis of many PDB files and let AWK create Jmol script files. These script files contain Jmol commands and can be executed as shell or AWK scripts can.

Let us start by downloading a bunch of PDB files from PDB. The database allows filtering (see Fig. 21.9). I downloaded all structures that were derived from *Escherichia* (either directly or via over-expression), have an X-ray resolution below 2 Å and consist of one single protein chain. As of 31 August 2012, 1,082 PDB files matched these criteria. You should save these files in the folder named *EcoliStructures*. If you are, for some reason, not able to download the data from the PDB, you can download the 100MB compressed file archive from the book's Webpage (ecolistructures.tar.gz). Program 69 on the next page shows the expanded version of Program 68 on p. 415. A rather cosmetic change is the omission of the BEGIN block. In line 21, we fill the array *hit* with the IDs of cysteines whose γ-sulfur atoms are less than 3 Å apart. In line 25, we check if there is any matching sulfur pair in the current structure file. Variable *i* can only be greater than 0 if it has been incremented in line 19. Line 26 prints the filename of the candidate structure to the terminal. In line 28, the output

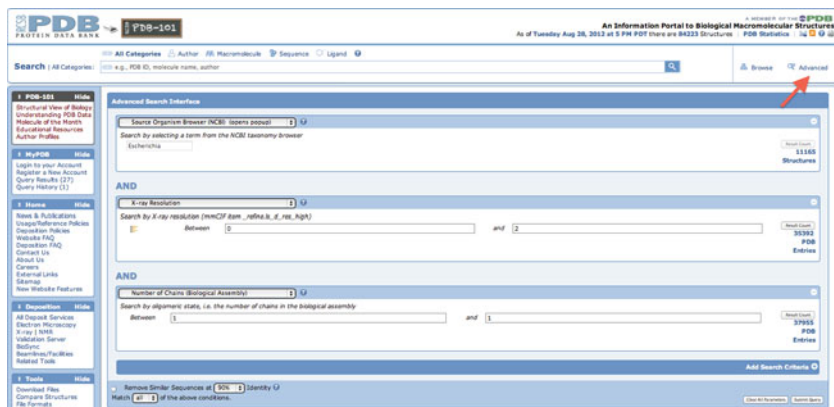


Fig. 21.9 Download PDB Structures. A screenshot of the enabled filters is shown. The *red arrow* marks the link that opens the filter menu

record separator is set to nothing (new line character is the default) and then the variable *file* is filled with the name of current structure file suffixed with ".script". It is this filename that will contain the Jmol script. This is achieved by printing the required Jmol commands into this file with

```
print ... >> filename
```

As in bash shell redirections, one greater sign redirects the content to a new file or overwrites the content of the file if it exists. Two greater signs append the data to an existing file (see Sect. 8.5 on p. 101).

```

Program 69: distance-batch.awk - Batch Calculating Cysteine-Sulfur Distances
1  # save as distance-batch.awk
2  # searches close cysteines in a structure
3  # requires a structure files (*.pdb)
4  # usage: for i in DIRECTORY/*.pdb; do awk -f distance-batch.awk $i; done
5
6  $1=="ATOM" && $4=="CYS" && $12=="S" {
7    cys_x[$6]=$7; cys_y[$6]=$8; cys_z[$6]=$9
8  }
9
10 END{
11   i=0
12   for (key1 in cys_x) {
13     for (key2 in cys_x) {
14       dx=cys_x[key1]-cys_x[key2]
15       dy=cys_y[key1]-cys_y[key2]
16       dz=cys_z[key1]-cys_z[key2]
17       distance=sqrt(dx^2+dy^2+dz^2)
18       if (distance < 3 && distance != 0) {
19         i++
20         candidate[i]=key1-"key2": distance
21         hit[key1]=key1; hit[key2]=key2
22       }

```

```

23     }
24 }
25 if(i!=0){
26     print FILENAME " is a candidate:"
27     for (keys in candidate) {print candidate[keys]}
28     ORS=""
29     file = FILENAME".script"
30     print "load "FILENAME"\n" >> file
31     print "spacefill off\n" >> file
32     print "hide water; select cys; wireframe 200; color blue\nselect" >> file
33     for(keys in hit){cystein=cystein"cys"keys".sg,"}
34     print substr(cystein,1,length(cystein)-1) >> file
35     print "\nspacefill 300; color yellow" >> file
36     print "\n" >> file
37     ORS="\n"
38 }
39 }

```

Program 70 shows one example of a Jmol script created by AWK for a structure file (2CGL) that has one γ -sulfur pair with a distance less than 3 Å.

```

_____ Program 70: Example Jmol Script Created by distance-batch.awk _____
1  load EcoliStructures/2CGL.pdb
2  spacefill off
3  hide water; select cys; wireframe 200; color blue
4  select cys222.sg,cys68.sg
5  spacefill 300; color yellow

```

I have not talked about the execution of the script, yet. Here, we take advantage of the power of Bash. The holy command is

```
for i in EcoliStructures/*.pdb; do awk -f distance-batch.awk $i; done
```

This executes the AWK script *distance-batch.awk* for all **.pdb* files in the subfolder *EcoliStructures* (see also Sect. 8.10 on p. 106).

21.3.5 Automatically Test for Solvent Accessibility

I can hear you saying already that this is still not efficient enough. Okay, you only have to open files of candidate structures and they are already formatted in a favorable way but—it would be nicer to have it fully automatized. Hmmm. What you need is a program that calculates the distance of a cysteine sulfur atom from the surface. This is not easy—but there are algorithms that predict the accessibility of atoms from the surface, e.g., Surface Racer by Tsodikov et al. (2002), see Sect. 21.2.3 on p. 410. Installation of Surface Racer is straightforward: just download the file from the Webpage. Surface Racer comes as a compressed file archive containing three files: the binary executable, a *readme.txt*, and a data file called *radii.txt*. The latter must sit in the same directory as the executable. The executable itself must be made executable with `chmod u+x`. Furthermore, Surface Racer requires a specific C++

programming library. If you see an error message as shown in Terminal 282, then you have a problem—and installation is not so much straight forward anymore.

```

Terminal 282: Error While Executing Surface Racer
1  $ ls
2  radii.txt  readme.txt  surface5_0_linux_32bit
3  $ ./surface5_0_linux_32bit
4  ./surface5_0_linux_32bit: error while loading shared libraries: libstdc++.so.5:  +
   cannot open shared object file: No such file or directory
5
6  $

```

The solution is to download the library *libstdc++.so.5* from the Internet and install it. Unfortunately, you need administration privileges to do so. If you do not own them, then ask your system administrator. If you own them, then take a look at Terminal 283.

```

Terminal 283: Installing the Missing Library
1  $ wget 'http://ftp.us.debian.org/debian/pool/main/g/gcc-3.3/
2      libstdc++5_3.3.6-20_i386.deb'
3  --2012-09-13 08:39:54--
4  http://ftp.us.debian.org/debian/pool/main/g/gcc-3.3/libstdc++5_3.3.6-20_i386.deb
5  Resolving ftp.us.debian.org... 128.30.2.36, 128.61.240.89, 35.9.37.225, ...
6  Connecting to ftp.us.debian.org (ftp.us.debian.org)[128.30.2.36]:80... connected.
7  HTTP request sent, awaiting response... 200 OK
8  Length: 311392 (304K) [application/x-debian-package]
9  Saving to: 'libstdc++5_3.3.6-20_i386.deb'
10
11 100%[=====] 311,392      424K/s   in 0.7s
12
13 2012-09-13 08:39:55 (424 KB/s) - libstdc++5_3.3.6-20_i386.deb saved [311392/311392]
14
15 $ sudo dpkg -i libstdc++5_3.3.6-20_i386.deb
16 [sudo] password for rw:
17 Selecting previously unselected package libstdc++5.
18 (Reading database ... 150142 files and directories currently installed.)
19 Unpacking libstdc++5 (from libstdc++5_3.3.6-20_i386.deb) ...
20 Setting up libstdc++5 (1:3.3.6-20) ...
21 Processing triggers for libc-bin ...
22 ldconfig deferred processing now taking place
23 $ ./surface5_0_linux_32bit
24 Surface Racer 5.0 by Oleg Tsodikov
25 Analytical surface area calculation
26
27 Van der Waals radii sets:
28 1 - Richards (1977)
29 2 - Chothia (1976)
30 Press 1 or 2 to choose a van der Waals radius assignment: ^C      # I pressed Ctrl-C

```

In lines 1–2 in Terminal 283 we use `wget` to download the library file. In line 15, we run the Debian package manager `dpkg` as user `root` with `sudo` (superuser do). This installs the missing library. In line 23, we check if Surface Racer can be started now—yes, hurray it does.

Program 71 on the following page shows an extension of the AWK script in Program 69 on p. 418 that implements the inclusion of Surface Racer. It takes advantage of AWK's capability to call shell programs like Surface Racer and read their output (recall Sect. 13.9.2 on p. 241).

```

----- Program 71: distance-batch-surface.awk - Extension by Accessibility -----
1  # save as distance-batch-surface.awk
2  # searches close cysteines in a structure
3  # identify solvent accessible sulfur atoms with Surface Racer
4  # requires a structure files (*.pdb)
5  # usage: for i in DIRECTORY/*.pdb; do awk -f distance-batch-surface.awk $i; done
6
7  $1=="ATOM" && $4=="CYS" && $12=="S" {
8  cys_x[$6]=$7; cys_y[$6]=$8; cys_z[$6]=$9
9  }
10
11 END{
12 i=0
13 for (key1 in cys_x) {
14     for (key2 in cys_x) {
15         dx=cys_x[key1]-cys_x[key2]
16         dy=cys_y[key1]-cys_y[key2]
17         dz=cys_z[key1]-cys_z[key2]
18         distance=sqrt(dx^2+dy^2+dz^2)
19         if (distance < 3 && distance != 0) {
20             i++
21             candidate[i]=key1~"key2": "distance
22             hit[key1]=key1; hit[key2]=key2
23         }
24     }
25 }
26 if(i!=0){
27     cmd1="echo \"1 \"FILENAME\" 5 1 n n\" | ./surface5_0_linux_32bit"
28     while((cmd1 | getline) > 0){
29         close(cmd1)
30         txtfile=FILENAME
31         gsub(/\,pdb/, ".txt", txtfile)
32         print FILENAME " is a CANDIDATE"
33         for (keys in candidate) {print candidate[keys]}
34         ORS=""
35         file = FILENAME".script"
36         print "load \"FILENAME\"\n" > file
37         print "spacefill off\n" >> file
38         print "hide water; select cys; wireframe 200; color blue\nselect " >> file
39         for(keys in hit){cystein=cystein"cys"keys".sg,"}
40         print substr(cystein,1,length(cystein)-1) >> file
41         print "\nspacefill 300; color yellow" >> file
42         cmd2="awk ' $11!=0{print $2}' "txtfile
43         while((cmd2 | getline) > 0){
44             print "\nselect atomno=\"$1"; color red; spacefill 100" >> file
45         }
46         close(cmd2)
47         cmd3="awk ' $3==\"SG\" && $11!=0{print $2}' "txtfile
48         while((cmd3 | getline) > 0){
49             print "\nselect atomno=\"$1"; color orange; spacefill 300" >> file
50             print "SURFACE SULFUR HIT -> atom \"$1\"\n"
51         }
52         close(cmd3)
53         print "\n" >> file
54         ORS="\n"
55     }
56 else{print FILENAME " is no candidate"}
57 }

```

The important additions start with line 27. Here, as in lines 42 and 47, we assign a bash command to the variable *cmd1*. Since Surface Racer is an interactive program that requires interaction with a human, i.e., you, we have to apply the

trick described in Sect. 10.10 on p. 150: We let `echo` pipe the required input to the program `surface5_0_linux_32bit`. The Bash command stored in variable `cmd1` is executed in line 28; line 29 closes the virtual Bash session. `surface5_0_linux_32bit` produces a text file that contains the accessibility of all atoms. The filename is the same as for the PDB file, except that the suffix is `txt`, instead of `pdb`. Lines 30–31 assign this filename to variable `txtfile`. Lines 32–41, you have seen before in Program 69 on p. 418. They create the Jmol script file.

In line 42, we design an AWK one-liner that extracts atom numbers of all atoms that are predicted to be solvent accessible, i.e., that are at the surface of the protein. Note that I used a probing sphere (see Fig. 21.2 on p. 411) of size 5 Å. This size influences the final result. We add this information to the Jmol script in line 44: all surface atoms will be shown as small red balls (see Fig. 21.10 on the facing page). The AWK command in line 47 specifically queries for γ -sulfur (SG) atoms at the surface. They will be colored orange in Jmol, as can be seen in line 49. Line 40 prints the atom number to the terminal.

Figure 21.10 on the next page illustrates the view of a candidate structure in Jmol. Program 72 shows the corresponding Jmol script that has been produced by *distance-batch-surface.awk*.

The AWK scripts presented here write quite a lot of information to the terminal, which you might like to save to a file. With the classical redirection `>`, you pipe everything to the file—but then the screen is empty. With the `tee` command you can split the data stream such that it is printed to the terminal and stored in a file (see Sect. 8.5 on p. 101).

```

Program 72: Example Jmol Script Created by distance-batch-surface.awk
1  load EcoliStructures/2CGL.pdb
2  spacefill off
3  hide water; select cys; wireframe 200; color blue
4  select cys222.sg,cys68.sg
5  spacefill 300; color yellow
6  select atomno=1; color red; spacefill 100
7  select atomno=2; color red; spacefill 100
8  ...
9  select atomno=3723; color red; spacefill 100
10 select atomno=200; color orange; spacefill 300

```

I hope you agree that AWK helped us a lot to automatize identification of potential Trx targets. And it is so much fun to develop and evolve such a script.

Exercises

21.1. How can you avoid the display of doublets in Program 68 on p. 415? Does this affect the calculation time?

21.2. So far, so good but: there is always room for optimizations. One nice thing would be not only to see in the terminal that there are cysteine γ -sulfur atoms close to the surface, but also to see if they are part of a less than 3 Å sulfur pair. A first step

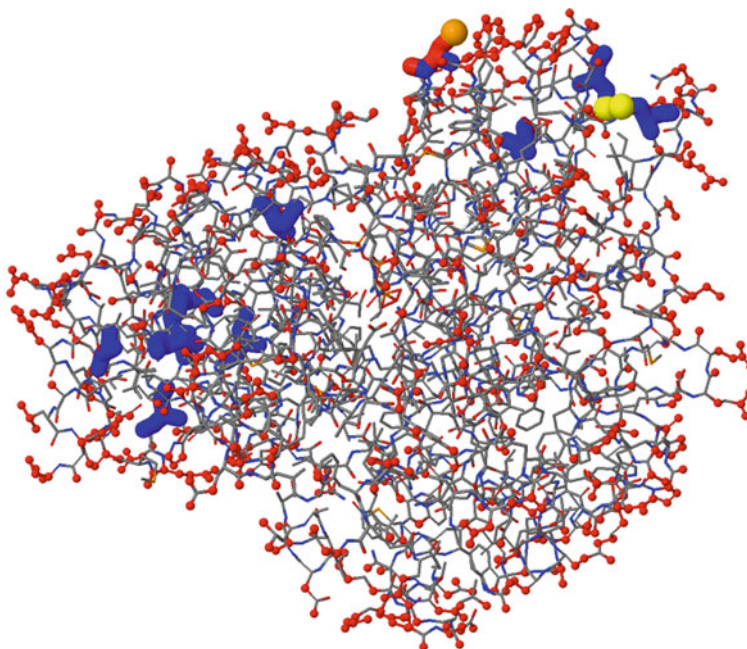


Fig. 21.10 Customized PDB Structure View. The AWK script *distance-batch-surface.awk* (Prog. 4) analyzes PDB structure files (here 2CGL) for the presence of potential Trx targets. For candidate structures, the script writes Jmol script files that customize the visualization of the structure such that cysteines in the structures are shown in **bold** and **blue**, sulfur atoms with less than 3 Å distance are shown as **yellow spheres**, and cysteine γ-sulfur atoms touching the molecular surface are shown as **orange spheres**

would be to print the cysteine ID of surface sulfur atoms. As an example, the output should be changed from

```
EcoliStructures/1DPE.pdb is a CANDIDATE
435-422: 2.02837
422-435: 2.02837
234-6: 2.1969
6-234: 2.1969
SURFACE SULFUR HIT -> atom 3360
SURFACE SULFUR HIT -> atom 3459
```

to something like

```
EcoliStructures/1DPE.pdb is a CANDIDATE
435-422: 2.02837
422-435: 2.02837
234-6: 2.1969
6-234: 2.1969
SURFACE SULFUR HIT IN SULFUR PAIR OF CYSTEINE 422
SURFACE SULFUR HIT IN SULFUR PAIR OF CYSTEINE 435
```

How would you solve this?

Appendix A

Supplementary Information

Röbbe Wünschiers

In this chapter you will find a number of practical hints.

A.1 Keyboard Shortcuts

As with Windows, where you can terminate programs by pressing (Alt)+(F4), there are a number of useful keyboard shortcuts in Linux, too. The following are the most common ones:

(Ctrl)+(Alt)+(F1)	Change to terminal 1.
(Ctrl)+(Alt)+(F2)	Change to terminal 2.
(Ctrl)+(Alt)+(F3)	Change to terminal 3.
(Ctrl)+(Alt)+(F4)	Change to terminal 4.
(Ctrl)+(Alt)+(F5)	Change to terminal 5.
(Ctrl)+(Alt)+(F6)	Change to terminal 6.
(Ctrl)+(Alt)+(F7)	Change to X-Windows.
(Ctrl)+(D)	Logout from the active shell. If the active shell is the login shell, you logout completely from the system.
(Ctrl)+(C)	Interrupt program or command execution.
(Ctrl)+(Z)	Stop program or command execution; send to background with bg .
(Tab)	Autocompletion of commands and filenames in the bash shell.
(↑)	Recalls the last command(s) in the bash shell.
(Ctrl)+(Alt)+(BkSp)	Kill X-Windows.

Biotechnology/Computational Biology, University of Applied Sciences, Technikumplatz 17,
09648 Mittweida, Germany

A.2 Optimizing the Bash Shell

You have learned already that the bash shell is a complex tool. Let us now focus on a few advanced settings.

A.2.1 Startup Files

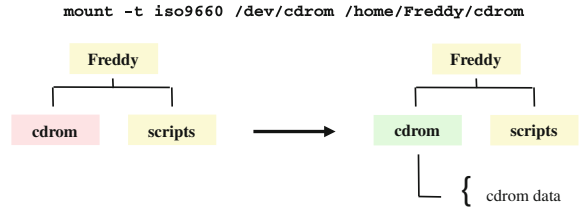
The bash shell uses a collection of startup files to create an environment to run in. Each file has a specific use and may affect login and interactive environments differently. These files generally sit in the */etc* directory. If an equivalent file exists in your home directory it overrides the global settings. In order to understand the different startup files, we must discriminate between different shell types. An interactive login shell is started after you successfully logged in (the leftmost shell in Fig. 8.1 on p. 98). This is our command line. The interactive login shell reads the files */etc/profile* and *~/.bash_profile*. While the entries in the former affect all users, the latter file is specifically for you. Thus, this is the right place to put settings, aliases, or scripts you always want to access. When you open a new shell with command `bash`, an interactive non-login shell is opened (the rightmost shells in Fig. 8.1 on p. 98). It reads the files */etc/bashrc* and *~/.bashrc*. Again, the latter overrules the former. Finally, the file *~/.bash_logout* is read by the shell when you logout of the system.

A.3 Devices

The following list shows you the path of some more or less special devices. Linux regards all hardware as devices. They can all be found in the system folder */dev*. Remember that Linux treats everything as files, even hardware devices.

<i>/dev/null</i>	Efficiently disposes unwanted output. If you save anything here, it is gone. If you read from there, the newline character is returned.
<i>/dev/hda1</i>	First IDE hard disk drive number (<i>a</i>), partition number (<i>1</i>). There can be several hard disk drives or partitions of a hard disk drive available. They are distinguished by their letter and number, respectively. <i>/dev/hdc2</i> would be partition 2 on drive 3
<i>/dev/sda1</i>	First SCSI hard disk drive, partition number 1. The nomenclature is the same as for IDE hard disk drives.
<i>/dev/fd0</i>	The floppy disk drive.
<i>/dev/cdrom</i>	Guess what: this is the CD-ROM drive.

Fig. A.1 Mounting Devices.
In this example the CD-ROM drive is mounted on the directory *cd-rom* in Freddy’s home directory. Usually, only the superuser (root) is allowed to execute the mount command



I must warn you: depending on the Linux installation you are working with, these paths might be different.

A.4 Mounting Filesystems

In Linux, filesystems like CD-ROMs are mounted. This means they look like a folder and you can place that folder wherever you want (Fig. A.1). Usually this is done automatically by the system. If not, this section shall help you to understand what is going on and how to mount devices manually. We have already seen in Sect. A.3 that CD-ROM drives and so on are devices. From the Windows operating system you are used to put an external memory medium like a floppy disk or CD-ROM into the drive and access it via, e.g., the file manager. Depending on the settings of your Linux system you are working with, using external memory media can be more difficult. By default, Linux is protected and you can access external memory media only if the system administrator has given you the required permissions.

When you mount a filesystem, you should know what kind of filesystem it is. There several different ones around:

ext2	Linux filesystem (version 2).
ext3	Linux filesystem (version 3).
ext4	Linux filesystem (version 4).
msdos	MS-DOS filesystem with 8+3 characters for filenames.
vfat	Windows 9x filesystem.
ntfs	Windows NT/2000/XP filesystem.
iso9660	CD-ROM filesystem.

In order to check which filesystems and partitions are available use the *df* commando. This command reports the filesystem disk space usage.

```
$ df -Th
```

Filesystem	Type	Size	Used	Avail	Use%	Mounted on
/dev/hda2	ext3	18G	3.1G	13G	18%	/
/dev/hda1	ext3	99M	14M	79M	15%	/boot

284: Filesystem

5	none	tmpfs	93M	4.0K	93M	1%	/dev/shm
6	\$						

In Terminal 284 we use the `df` command with the options `-T` and `-h`, which leads to the addition of the filesystem-type information and makes the output human-readable, respectively.

Now, let us see how you can add a filesystem like a CD-ROM or DVD drive or USB-stick manually.

A.4.1 Mounting External Memory Devices

Any external memory device is added to the existing filesystem with the command `mount`. Usually, you must be superuser (root) in order to be able to use `mount`. If you are working on your own system, you should have the root password. You can always login as root with the command `su`.

```

1  $ cat /etc/passwd | grep Freddy
2  Freddy:x:502:502::/home/Freddy:/bin/bash
3  $ echo $UID
4  502
5  $ cd
6  $ mkdir CDROM
7  $ su
8  Password:
9  # mount -t iso9660 -o uid=502 /dev/cdrom /home/Freddy/CDROM
10 # exit
11 exit
12 $ ...
13 $ cd
14 $ su
15 Password:
16 # umount /home/Freddy/CDROM
17 # exit
18 exit
19 $

```

When root mounts a filesystem, it belongs to root. In order to allow you to edit files and directories, you need to know your user ID (do not confuse it with the username). Terminal 285 shows you a list of events you have to perform in order to mount a CD-ROM. The first 4 lines demonstrate two possibilities to find out your user ID. Then you change to your home directory (`cd`) and create the directory *CD-ROM*. Now, login as root (`su`) and execute the crucial command in line 9. Then, logout as root (`exit`). Now you can access and edit the CD-ROM. Before you remove the CD-ROM, unmount it! Go back into your home directory (the drive cannot be unmounted if you have a folder or file of it opened) and login as root. The correct `umount` command is shown in line 16. Now you can take out the CD-ROM. The same applies to other devices. If it does not work, you have to find out what filesystem is used by your device and where it is located in the device list.

A.5 Nucleotide and Protein Codes

Of course, you are welcome to learn the codes; however, you might prefer to look them up in Table A.1.

Table A.1 Standard genetic code

1st Position		2nd Position			3rd Position
	U	C	A	G	
U	UUU Phe	UCU Ser	UAU Tyr	UGU Cys	U
	UUC Phe	UCC Ser	UAC Tyr	UGC Cys	C
	UUA Leu	UCA Ser	UAA Stop	UGA Stop	A
	UUG Leu	UCG Ser	UAG Stop	UGG Trp	G
C	CUU Leu	CCU Pro	CAU His	CGU Arg	U
	CUC Leu	CCC Pro	CAC His	CGC Arg	C
	CUA Leu	CCA Pro	CAA Gln	CGA Arg	A
	CUG Leu	CCG Pro	CAG Gln	CGG Arg	G
A	AUU Ile	ACU Thr	AAU Asn	AGU Ser	U
	AUC Ile	ACC Thr	AAC Asn	AGC Ser	C
	AUA Ile	ACA Thr	AAA Lys	AGA Arg	A
	AUG Met/Start	ACG Thr	AAG Lys	AGG Arg	G
G	GUU Val	GCU Ala	GAU Asp	GGU Gly	U
	GUC Val	GCC Ala	GAC Asp	GGC Gly	C
	GUA Val	GCA Ala	GAA Glu	GGA Gly	A
	GUG Val	GCG Ala	GAG Glu	GGG Gly	G

Next comes the single-letter and triple-letter protein code.

Table A.2 The protein code

Alanine	Ala	A	Cysteine	Cys	C
Aspartic acid	Asp	D	Glutamic acid	Glu	E
Phenylalanine	Phe	F	Glycine	Gly	G
Histidine	His	H	Isoleucine	Ile	I
Lysine	Lys	K	Leucine	Leu	L
Methionine	Met	M	Asparagine	Asn	N
Proline	Pro	P	Glutamine	Gln	Q
Arginine	Arg	R	Serine	Ser	S
Threonine	Thr	T	Valine	Val	V
Tryptophan	Trp	W	Tyrosine	Tyr	Y

A.6 Jmol and Rasmol Commands

The following list gives an overview of some important Jmol and Rasmol commands. Both molecule viewers are to a large extent compatible. You can also find cheat sheets on the web:

<code>restrict :A</code>	Only shows chain A
<code>center :A</code>	Centers the display to chain A
<code>select all</code>	Select all atoms
<code>select HIS34:D</code>	Select the histidine 34 of chain D
<code>select :D</code>	Select chain D of the protein
<code>select dna</code>	Select the DNA strand if there is one
<code>select ligand</code>	Select all heteroatoms (all 'non protein' atoms)
<code>select protein</code>	Select all atoms of the protein
<code>select asn</code>	Select all ASN amino acids (other similar)
<code>wireframe off</code>	Turns off the wireframe model
<code>wireframe .3</code>	Turns on the stick model (good for small molecules)
<code>spacefill on</code>	Turns on the spacefill mode (good for small molecules)
<code>spacefill off</code>	Turns off the spacefill mode
<code>backbone on</code>	Turns on the backbone model
<code>backbone 200</code>	Turns the backbone model with thicker lines
<code>backbone off</code>	Turns off the backbone model
<code>cartoon on</code>	Turns on the cartoon mode
<code>cartoon 400</code>	Turns on the cartoon mode with thicker lines
<code>cartoon off</code>	Turns off the cartoon mode
<code>color cpk</code>	Color the selected atoms in CPK mode
<code>color chain</code>	Color the selected atoms by chains
<code>color structure</code>	Color the selected atoms by structure
<code>color group</code>	Color the selected atoms by group
<code>color temperature</code>	Color the selected atoms flexibility
<code>background white</code>	Sets the background color to white
<code>background black</code>	Sets the background color to black
<code>background yellow</code>	Sets the background color to yellow
<code>zoom 90</code>	To scale down to 90
<code>zoom 120</code>	To scale up to 120
<code>rotate x 20</code>	Rotates around the x axis at 20 degree
<code>rotate y 20</code>	Rotates around the y axis at 20 degree
<code>rotate z 20</code>	Rotates around the z axis at 20 degree

A.7 Special Characters

It is quite important that you and I speak the same language when it comes to special characters like parentheses, brackets, and braces. The following is a list for your orientation:

	Space	blank
#	Crosshatch	number sign, sharp, hash
\$	Dollar Sign	dollar, cash, currency symbol, string
%	Percent Sign	percent, grapes
&	Ampersand	and, amper, snowman, daemon
*	Asterisk	star, spider, times, wildcard, pine cone
,	Comma	tail
.	Period	dot, decimal (point), full stop
:	Colon	two-spot, double dot, dots
;	Semicolon	semi, hybrid
<>	Angle Brackets	angles, funnels
<	Less Than	less, read from
>	Greater Than	more, write to
=	Equal Sign	equal(s),
+	Plus Sign	plus, add, cross, and, intersection
-	Dash	minus (sign), hyphen, negative (sign)
!	Exclamation Point	exclamation (mark), (ex)clam
?	Question Mark	question, query, wildcard
@	Vortex	at, each, monkey (tail)
()	Parentheses	parens, round brackets, bananas
(	Left Parentheses	open paren, wane, parenthesees
)	Right Parentheses	close paren, wax, unparenthesees
[]	Brackets	square brackets, edged parentheses
[	Left Bracket	bracket, left square bracket, opensquare
]	Right Bracket	unbracket, right square bracket, unsquare
{ }	Braces	curly braces
{	Left Brace	brace, curly, leftit, embrace, openbrace
}	Right Brace	unbrace, uncurly, rytit, bracelet, close
/	Slash	stroke, diagonal, divided-by, forward slash
\	Backslash	bash, (back)slant, escape, blash
^	Circumflex	caret, top hat, cap, uphat, power
"	Double Quotes	quotation marks, literal mark, rabbit ears
'	Single Quotes	apostrophe, tick, prime
`	Grave	accent, back/left/open quote, backprime
~	Tilde	twiddle, wave, swung dash, approx
_	Underscore	underline, underbar, under, blank
	Vertical Bar	pipe to, vertical line, broken line, bar

References

1. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. *J Mol Biol* 215:403–410
2. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ (1997) Gapped BLAST and psi-BLAST: a new generation of protein database search programs. *Nucl Acids Res* 25:3389–3402
3. Anscombe FJ (1973) Graphs in statistical analysis. *Am Stat* 27:17–21
4. Arnold K, Bordoli L, Kopp J, Schwede T (2006) The SWISS-MODEL workspace: a web-based environment for protein structure homology modelling. *Bioinformatics* 22:195–201
5. Blattner FR, Plunkett G, Bloch CA, Perna NT, Burland V, Riley M, Collado-Vides J, Glasner JD, Rode KC, Mayhew GF, Gregor J, Davis NW, Kirkpatrick HA, Goeden MA, Rose DJ, Mau B, Shao Y (1997) The complete genome sequence of *Escherichia coli*. *Science* 277:1453–1462
6. Camacho C, Coulouris G, Avagyan V, Ma N, Papadopoulos J, Bealer K, Madden TL (2009) BLAST+: architecture and applications. *BMC Bioinformatics* 10:421
7. Cohen MB, Giannella RA (1992) Hemorrhagic colitis associated with *Escherichia coli* O157:H7. *Adv Intern Med* 37:173–195
8. Cole C, Barber JD, Barton GJ (2008) The Jpred 3 secondary structure prediction server. *Nucl Acids Res* 36:W197–W201
9. Cummings L, Riley L, Black L, Souvorov A, Resenchuk S, Dondoshansky I, Tatusova T (2002) Genomic BLAST: custom-defined virtual databases for complete and unfinished genomes. *FEMS Microbiol Lett* 216:133–138
10. Dayhoff M (1964) Computer aids to protein sequence determination. *J Theor Biol* 8:97–112
11. Dayhoff MO, Ledley RS (1962) Comproteins: a computer program to aid primary protein structure determination. *AFIPS conference proceedings, ACM, In*, pp 262–274
12. Deitel HM, Deitel PJ, Nieto TR, McPhie DC (2001) *Perl—How to program*. Prentice Hall, Upper Saddle River
13. Dougherty D, Robbins A (1997) *Sed & AWK*. O'Reilly & Associates, Sebastopol
14. Dwyer RA (2003) *Genomic Perl*. Cambridge University Press, Cambridge
15. Edgar RC (2004) Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucl Acids Res* 32:1792–1797
16. Elhai J, Taton A, Massar JP, Myers JK, Travers M, Casey J, Slupsky M, Shrager J (2009) BioBIKE: a web-based, programmable, integrated biological knowledge base. *Nucl Acids Res* 37:W28–W32
17. Fitch WM, Margoliash E (1967) Construction of phylogenetic trees. *Science* 155:279–284
18. Fleischmann RD, Adams MD, White O, Clayton RA, Kirkness EF, Kerlavage AR, Bult CJ, Tomb JF, Dougherty BA, Merrick JM (1995) Whole-genome random sequencing and assembly of *Haemophilus influenzae* Rd. *Science* 269:496–512
19. Fritsche E, Paschos A, Beisel HG, Böck A, Huber R (1999) Crystal structure of the hydrogenase maturing endopeptidase HybD from *Escherichia coli*. *J Mol Biol* 288:989–998

20. Gonnet G, Hallett M, Korostensky C, Bernardin L (2000) Darwin v. 2.0: an interpreted computer language for the biosciences. *Bioinformatics* 16:101–103
21. Herold H (2003) *awk & sed—Die Profitools zur Dateibearbeitung und -editierung*. Addison-Wesley, Bonn
22. Lamb L, Robbins A (1998) *Learning the vi editor*. O'Reilly & Associates, Sebastopol
23. Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions and reversals. *Sov Phys Doklady* 10:707–710
24. Mangalam HJ (2002) TACG—a grep for DNA. *BMC Bioinf* 3:8
25. Massar JP, Travers M, Elhai J, Shrager J (2005) Biolingua: a programmable knowledge environment for biologists. *Bioinformatics* 21:199–207
26. McCulloch W, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. *Bull Math Biophys* 5:115–133
27. Mellmann A, Harmsen D (2011) Cummings CA, Zentz EB, Leopold SR, Rico A, Prior K, Szczepanowski R, Ji Y, Zhang W, McLaughlin SF, Henkhaus JK, Leopold B, Bielaszewska M, Prager R, Brzoska PM, Moore RL, Guenther S, Rothberg JM, Karch H (2011) Prospective genomic characterization of the German enterohemorrhagic *Escherichia coli* O104:H4 outbreak by rapid next generation sequencing technology. *PLoS ONE* 6:e22751
28. Needleman SB, Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol* 48:443–453
29. Ouzounis CA (2012) Rise and demise of bioinformatics? Promise and progress. *PLoS Comput Biol* 8:e1002487
30. Pedersen M, Phillips A (2009) Towards programming languages for genetic engineering of living cells. *J R Soc Interface* 6:S437–S450
31. Perna N, Plunkett G, Burland V, Mau B, Glasner J, Rose D, Mayhew G, Evans P, Gregor J, Kirkpatrick H, Hackett J, Klink S, Boutin A, Shao Y, Miller L, Grotbeck EJ, Davis NW, Lim A, Dimalanta ET, Potamoudis KD, Apodaca J, Anantharaman TS, Lin J, Yen G, Schwartz DC, Welch RA, Blattner FR (2001) Genome sequence of enterohaemorrhagic *Escherichia coli* O157:H7. *Nature* 409:529–533
32. Qualline S (2001) *Vi improved-VIM*. New Riders Publishing, Indianapolis
33. Reid SD, Herbelin CJ, Bumbaugh AC, Selander RK, Whittam TS (2000) Parallel evolution of virulence in pathogenic *Escherichia coli*. *Nature* 406:64–67
34. Riley LW, Remis RS, Helgerson SD, McGee HB, Wells JG, Davis BR, Hebert RJ, Olcott ES, Johnson LM, Hargrett NT, Blake PA, Cohen ML (1983) Hemorrhagic colitis associated with a rare *Escherichia coli* serotype. *N Engl J Med* 308:681–685
35. Ritchie DM, Thompson K (1974) The UNIX time-sharing system. *Commun ACM* 17:365–375
36. Robbins A (1999) *VI editor pocket reference*. O'Reilly & Associates, Sebastopol
37. Schürmann P (2000) Plant thioredoxin system revisited. *Annu Rev Plant Physiol Plant Mol Biol* 51:371–400
38. Sigrist CJ, Cerutti L, Hulo N, Gattiker A, Falquet L, Pagni M, Bairoch A, Bucher P (2002) PROSITE: A documented database using patterns and profiles as motif descriptors. *Brief Bioinf* 3:265–274
39. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. *J Mol Biol* 147:195–197
40. Sonnhammer EL, Durbin R (1995) A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis. *Gene* 167:GC1–GC10.
41. Stajich J, Block D, Boulez K, Brenner S, Chervitz S, Dagdigian C, Fuellen G, Gilbert J, Korf I, Lapp H, Lehvaslaiho H, Matsalla C, Mungall C, Osborne B, Pocock MR, Schattner P, Senger M, Stein LD, Stupka ED, Wilkinson MD, Birney E (2002) The bioperl toolkit: Perl modules for the life sciences. *Genome Res* 12:1611–1618
42. Thompson K (1968) Programming techniques: regular expression search algorithm. *Commun ACM* 11:419–422
43. Thompson JD, Higgins DG, Gibson TJ (1994) ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. *Nucl Acids Res* 22:4673–4680

44. Tisdall J (2001) Beginning Perl for bioinformatics. O'Reilly & Associates, Sebastopol
45. Torvalds L, Diamond D (2001) Just for fun: the story of an accidental revolutionary. Harper-Business, New York
46. Tsodikov OV, Record MT, Sergeev YV (2002) Novel computer program for fast exact calculation of accessible and molecular surface areas and average surface curvature. J comput chem 23:600–609
47. Venter JC et al (2001) The sequence of the human genome. Science 291:1304–1351
48. Vromans J (2000) Perl 5. O'Reilly & Associates, Sebastopol
49. Wu S, Manber U (1992) Fast text searching: allowing errors. Commun ACM 35:83–91
50. Wüschiers R, Batur M, Lindblad P (2003) Presence and expression of hydrogenase specific C-terminal endopeptidases in cyanobacteria. BMC Microbiol 3:8
51. Wüschiers R, Heide H, Follmann H, Senger H, Schulz R (1999) Redox control of hydrogenase activity in the green alga *Scenedesmus obliquus* by thioredoxin and other thiols. FEBS Lett 455:162–164
52. Zeeberg BR, Riss J, Kane DW, Bussey KJ, Uchio E, Linehan WM, Barrett JC, Weinstein JN (2004) Mistaken identifiers: gene name errors can be introduced inadvertently when using excel in bioinformatics. BMC Bioinform 5:80

Solutions

Before you peek in here, try to solve the problems yourself. In some cases, especially for the programming exercises, you might need to spend an hour or so. Some scripts behave strangely if input files are in DOS format. In these cases use the `dos2unix filename` command to convert the file (see Sect. 7.3 on p. 95).

Solutions to Chap. 4

- 4.1 Take a look at Sect. 4.2 on p. 43.
- 4.2 Take a look at Sect. 4.3.3 on p. 48.
- 4.3 Type `date > the_date`, then `date >> the_date`, and finally
`cat the_date`
- 4.4 Use `passwd`
- 4.5 Use: `exit` or `logout` or `(Ctrl)+(D)`

Solutions to Chap. 5

- 5.1 Use: `cd` or `cd ~`; `pwd`; `ls -a`; `ls -al`
- 5.2 Use: `pwd`; `cd .`; `cd /`; `cd` or `cd ~`
- 5.3 Use: `cd` or `cd ~`; `mkdir testdir`; `mkdir testdir/subdir`;
`ls -a testdir/subdir`; `rm -r testdir/subdir`
- 5.4 Amazing how many directories there are, isn't it?
- 5.5 Use: `ls -l /`; `ls -l /etc/passwd`; probably you have all rights on */etc*
and read-only right on */etc/passwd*; `cat /etc/passwd`
- 5.6 Use: `mkdir testdir2`; `date > testdir2/thedate.txt`;
`cp -r testdir2 testdir`; `rm -r testdir2`
- 5.7 Use: `date > testdir/now`; `chmod 640 testdir/now` or
`chmod u+rw,g+r,o-rwx testdir/now`

- 5.8 `750=rwxr-x---;600=rw-----;640=rw-r-----;use: chmod -R 640 testdir/*`
- 5.9 Use: `mkdir dirname; cd dirname`
- 5.10 Use: `man sort > thepage.txt; ls -l; compress thepage.txt; ls -l; calculate the size ratio and multiply by 100`
- 5.11 Use: `mkdir testpress; man sort > testpress/file1; only the files, not the directory, are compressed`
- 5.12 Use: `date > .hiddendatefile; ls -a`
- 5.13 Take a look at Sect. 5.6 on p. 62.
- 5.14 There is no difference between moving and renaming. Copying does and moving, renaming does not change the time stamp. Check with: `ls -l`
- 5.15 It is increased by 1. Check with: `ls -l`
- 5.16 Creation and renaming does not change, editing does change the time stamp.
- 5.17 Have fun...

Solutions to Chap. 9

- 9.1 Use the same commands described in Sect. 9.1 on p. 115.
- 9.2 Use: `gunzip tacg-3.50-src.tar.gz; tar xf tacg-3.50-src.tar.gz`
- 9.3 Use: `.configure` and then `make -j2`
- 9.4 Use `cat` or `less` for file viewing (see Chap. 7 on p. 77).

Solutions to Chap. 7

- 7.1 Use: `cat > fruits.txt` and `cat >> fruits.txt`. Stop input with `(Ctrl)+(D)`.
- 7.2 Use: `cat > vegetable; cat fruits.txt vegetable > dinner`
- 7.3 Use: `sort dinner`
- 7.4 Take a look at Sect. 7.2.2 on p. 88.

Solutions to Chap. 8

- 8.1 Use: `chsh` and later enter `/bin/bash`
- 8.2 Use: `ps | sort +4 -n | tee filename.txt`

Solutions to Chap. 10

10.1 Hey, do not waste your time—play with the code!

Solutions to Chap. 11

11.1 I cannot help you here

11.2 Use: `egrep '^[_]+[_][_]+$_' file.txt`

11.3 Depending on your system use: `egrep -e '-[0-9]+' file.txt` or `egrep -e '\-[0-9]+' file.txt` Note the use of the '-e' switch. Without it, `egrep` interprets the leading minus character in the regular expression as a switch indicator.

11.5 Use: `egrep ' _-? ([0-9]+\.[0-9]*|[0-9]*\.[0-9]+) ' file.txt`
Here the minus is okay because it is preceded by a space character.

11.6 Use: `egrep '_hydrogenase' file.txt`

11.7 Use: `egrep 'GT.?TACTAAC.?AG' seq.file`

11.8 Use: `egrep 'G[RT]VQGVGFR.{13}[DW]V[CN]N{3}G' sep.file` since the N stands for any amino acid you can replace all Ns by `[GPAVLIMCFYWHKRQVEDST]`

11.9 Use: `ls -l | egrep '{7}r'` or `ls -l | egrep '^.....r'`

Solutions to Chap. 12

12.1 Use: `sed 's/Beisel/Weisel/' structure.pdb`

12.2 Use: `sed '1,3d' structure.pdb`

12.3 Use: `sed -n '5,10p' structure.pdb` or `sed -e '1,4d' -e '11,$d' structure.pdb`

12.4 Use: `sed '/MET/d' structure.pdb`

12.5 Use: `sed -n '/HELIX.*ILE/p' structure.pdb`

12.6 Use: `sed '/^H/s/$/****/' structure.pdb`

12.7 Use: `sed '/SEQRES/s/^\.*$/SEQ/' structure.pdb`

12.8 Use: `sed '/^$/d' text.file`

Solutions to Chap. 13

13.1 Use: `awk 'BEGIN{FS=","}{for (i=NF; i>0; i--){out=out$i" - "}
print out}' numbers.txt`

13.2 `BEGIN{RS=";"; i=1
while (getline < "words.txt" > 0){word[i]=$1; i++}`

```
RS=", "}  
{print $1, ":", word[NR]}
```

```
13.3 BEGIN{n=-1  
    for(i=1; i<51; i++){  
        n++; if(n==10){print " "; n=0}; printf("%2s ", i)}  
    print " "}
```

```
13.4 Use: awk '{print $0, "-", $5/$8, "bp/gene"}' genomes2.txt
```

Solutions to Chap. 14

14.1 Add the following lines at the end of the program:

```
print "Translated Sequence:\n";  
while (length(substr($DNA,0,3)) == 3){  
    $stri=substr($DNA,0,3,"");  
    print aa($stri)}  
sub aa{  
    my($codon)=@_  
    if ($codon =~ /GC[ATGC]/) {return "A"} # Ala  
    elsif ($codon =~ /TG[TC]/) {return "C"} # Cys  
    elsif ($codon =~ /GA[TC]/) {return "D"} # Asp  
    elsif ($codon =~ /GA[AG]/) {return "E"} # Glu  
    elsif ($codon =~ /TT[TC]/) {return "F"} # Phe  
    elsif ($codon =~ /GG[ATGC]/) {return "G"} # Gly  
    elsif ($codon =~ /CA[TC]/) {return "H"} # His  
    elsif ($codon =~ /AT[TCA]/) {return "I"} # Ile  
    elsif ($codon =~ /AA[AG]/) {return "K"} # Lys  
    elsif ($codon =~ /TT[AG] | CT[ATGC]/) {return "L"} # Leu  
    elsif ($codon =~ /ATG/) {return "M"} # Met  
    elsif ($codon =~ /AA[TC]/) {return "N"} # Asn  
    elsif ($codon =~ /CC[ATGC]/) {return "P"} # Pro  
    elsif ($codon =~ /CA[AG]/) {return "Q"} # Gln  
    elsif ($codon =~ /CG[ATGC] | AG[AG]/) {return "R"} # Arg  
    elsif ($codon =~ /TC[ATGC] | AG[TC]/) {return "S"} # Ser  
    elsif ($codon =~ /AC[ATGC]/) {return "T"} # Thr  
    elsif ($codon =~ /GT[ATGC]/) {return "V"} # Val  
    elsif ($codon =~ /TGG/) {return "W"} # Trp  
    elsif ($codon =~ /TA[TC]/) {return "Y"} # Tyr  
    elsif ($codon =~ /TA[AG] | TGA/) {return "*"} # Stop  
    else {return "<$codon=bad-codon>"}  
    }  
}
```


14.2 Add the following lines before the subroutine *aa* of the previous solution:

```
print "\nTranslated Complemented Sequence:\n";
$RevCompl=~tr/U/T/;
while (length(substr($RevCompl,0,3)) == 3){
    $tri=substr($RevCompl,0,3,"");
    print aa($tri)}
```

14.3 Replace the following two lines from the previous program:

```
print "Enter Sequence: ";
$DNA=<STDIN>; # ask for a seq
```

by this block:

```
print "Enter Sequence Filename: ";
$file=<STDIN>;
open (FILE, $file) or die("Could no open file...");
@content=<FILE>; close FILE;
foreach $temp (@content) {$i++; chomp $temp;
    if ($temp =~ /^ > /) {$name[$i]="$temp\n";}
    else {$i--; $seq[$i]=$seq[$i].$temp}}
for ($i=1; $i < (scalar @name); $i++){
    print "\n\n$name[$i]\n"; $DNA=$seq[$i];
```

and add a closing brace “}” before the subroutine *aa*.

14.4 Headache? This makes no difference to the code... Sorry for this joke.

14.5 Do not print out the sequence again.

14.6 Replace the filename *cutterinput.seq* by the variable *\$ARGV[0]*

14.7 Change the *printf* lines to:

```
printf("%4s %-30s %4s\n", $count,
    substr($DNA,0,30,""), $count+29);$count=$count+30;
printf("%s%-35s%s", " ",
    substr($OUTPUT[$i],0,30,""), "\t$NAMES[$i]\n");
```

and add *\$counter=1;* at the beginning of the script.

14.8 Use a database file in order to fill the hash table as described in Sect.14.7 on p. 276.

14.9 Add *\$DNA=\$DNA. " ";* to line 26 and replace the dot in the regular expression in line 32 by *[ACGTacgtuU]* in Program 61 on p. 288.

Solutions to Chap. 18

Please note that the numbers given in this solutions can be slightly different from what you get. The reason is that the genome annotations are constantly updated.

- 18.1 314 annotated proteins are unique to *E. coli* strain O157:H7.
- 18.2 (a) 522 unique proteins. (b) To solve this task you need to create a BLAST+ database for *E. coli* strain O157:H7 as shown in Terminal 247 on p. 353. After BLASTing *E. coli* strain O104:H4 against this database and analyzing the result file you should obtain 346 unique annotated proteins.
- 18.3 The E-value describes the random background noise. An E-value of 1 assigned to a hit means that one might expect to see one match with a similar score simply by chance. The higher the E-value, the more low-scoring BLAST-hits are reported and the higher is the probability that we get random hits, which means background noise.
- 18.4 This result makes perfect sense. Since we decrease the threshold for what is evaluated as a match by BLAST, we will see more hits (see Exercise 18.3 on p. 373) and consequently less ‘no hits’.
- 18.5 Prerequisite to the solution of this exercise is the creation of a BLAST database for *E. coli* strain O157:H7. With this database you have to proceed similarly as shown in Terminals 256 on p. 364 (but you must change the filename to *annotation-h7.tab* in line 24/25 and adopt the first SED substitution from `s/\[.*\tK12/` to `s/\[.*\tH7/`. Then proceed as in Sect. 18.3.6.2 on p. 364.
- 18.6 The reason is that one sequence can give rise to several alignments either because several sequences of the BLAST database match the query sequence or only parts of the query sequence match parts of target sequences.
- 18.7 To test this, execute `t.test(dehydrogenase[[1]], dna[[1]])` in R. The resulting p-value equals 0.9573, which is not below the threshold of 0.05. Thus, there is no significant difference in the means of the maximum alignment lengths of DNA-polymerases and dehydrogenases.

Solutions to Chap. 21

- 21.1 One possible solution is: add to line 14: `list=list "key1"`; add after line 15 `if (index(list, key2) != 0) continue`; and yes, less calculation time is needed.
- 21.2 What you need is the atom number of all cysteine γ -sulfur atoms from the PDB file. You can store them in an array by adding `atom[$2]=$6` to line 8 of Program 71 on p. 421. This command creates an array called *atom* that has the sulfur atom number as key (index) and the corresponding cysteine ID as value. After line 47 you should add two lines:

```
if($1 in atom){print "SURFACE SULFUR HIT IN SULFUR PAIR OF CYSTEINE" atom[$1]"\n"}
else{print "SURFACE SULFUR HIT -> atom "$1"\n"}
```

In these two lines we check if the current surface sulfur atom belongs to a cysteine that is part of a pair. If this is the case, the cysteine ID is printed.

Index

Commands for shell programming, `sed`, `awk`, `perl`, `MySQL` or `R` will be found at these entries

Filename Extension

- `.bz2`, 66
- `.tar.gz`, 65
- `.tar.z`, 65
- `.taz`, 65
- `.tgz`, 65
- `.tz`, 65
- `.zip`, 65

A

Access modes for files, 56

AWK, 197, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254

actions, 225

array, 215

array assignment, 216

array delete, 218

array element, 217

array scan, 217

array sort, 218

`asort`, 218, 232

`BEGIN`, 207

`break`, 224

calculations, 228

character expression, 203

`continue`, 224

default action, 199

default field separator, 199

default pattern, 199

`delete`, 218

`do...while`, 222

`END`, 207

examples, 243

executables, 219

`exit`, 224

field separator, 199

flow control, 220

`for`, 217

`for...`, 223

formatted printing, 226

function, 234

functions, 234

`gensub`, 230

`getline`, 238

getting started, 198

global variable, 235

`gsub`, 230

`if...else`, 220

index, 231

input from a file, 239

input from the shell, 241

introduction, 197, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254

L^AT_EX, 242

length, 231

local variable, 235

match, 231

match expression, 202

`next`, 224

number expression, 205

parameter, 213

pattern, 201

predefined variable, 210

`print`, 225

`printf`, 226

print to file, 225

ranges, 206

A (cont.)

- regular expression, 201
- return, 237
- script files, 219
- shell environment, 214
- split, 216, 232
- string manipulation, 229
- strtonum, 233
- sub, 230
- substr, 229
- syntax, 199
- system bell, 226
- system calls, 234
- text manipulation, 229
- tolower, 233
- toupper, 233
- user functions, 234
- variable assignment, 208
- variable decrement, 209
- variable global, 235
- variable increment, 209
- variable local, 235
- variable predefined, 210
- variables, 208
- while..., 221

B

- Backups, 68, 73
- Batch files, 97
- Bioscience software, 33
- Blast
 - download, 116
 - execute, 119
 - installation, 118
- bunzip2, 66
- bzip2, 66

C

- CPU usage
- CD-ROM, 428
- Change password, 49
- Check disk space, 68
- chgrp, 57
- chown, 57
- ClustalW, 376
 - batch job, 122
 - download, 117
 - execution, 121
 - installation, 120
 - wrapper, 122
- compress, 66

- Computational biology, 7

- Computer basics, 21

- Convert text files, 95

- curl, 72

- CygWin, 30, 44

D

- Devices, 426

- df, 68

- Disk space, 68

- Disk space usage, 427

- dos2unix, 95

- Dotter, 394

- dpkg, 420

- du, 68

- DVD, 428

- Dynamic programming, 251

F

- File access modes, 56

- File archives, 62

- File attributes, 56

- change, 60

- File compression, 65

- File permissions, 56

- change, 60

- File search, 66

- Find file content

- grep, 84

- Find files, 66

- Find text

- grep, 84

FTP

- bin, 116

- cd, 116

- get, 116

- ls, 116

- pwd, 118

- quit, 116

- shell command, 118

- ftp, 116

G

- Genetic code, 429

- Genomic BLAST, 376

- GLP, 25

- GNU, 25

- Graphical user interface, 26

- grep, 84

- gtar, 63

gunzip, 65
gzip, 65

H

Homology modeling, 378
HTML, 291

I

id, 61

J

Java, 291
Jmol, 378, 387, 410, 412
Jmol commands, 430
Jpred, 377

K

Kernel, 28
Keyboard shortcuts, 425
Knoppix, 31

L

L^AT_EX, 242
Levenshtein distance, 250
Linux distributions, 26
Login, 43
Logout, 50

M

Mac OS X, 29, 44
Manual pages, 49
Mount
 CD-ROM, 428
 DVD, 428
 filesystem, 427
 USB stick, 428
mount, 428
MySQL
 AS, 313
 CREATE DATABASE, 302
 CREATE TABLE, 304, 305, 363, 365
 DELETE, 300, 307
 DESCRIBE, 305, 363, 365
 GRANT, 301
 INSERT, 306
 LEFT JOIN, 313
 LIMIT, 309
 LOAD DATA, 308, 363, 365

mysql, 299
mysqladmin, 298–300
mysqld, 297, 298
mysqldump, 314
mysqlimport, 307, 308, 364
QUIT, 301
REGEXP, 310
RIGHT JOIN, 313
SELECT, 306, 308, 309, 363, 366, 368
SHOW DATABASES, 302
SHOW TABLES, 305
UPDATE, 306, 307
USE, 300, 304
backups, 314
batch job, 314
creating databases, 302
creating tables, 304
data types, 304
database transfer, 314
editing fields, 306
execution from shell, 313
field types, 306
installation, 42
joins, 312
login, 299
mathematics, 312
null, 309
regular expressions, 310
remote access, 303
root password, 299
security, 299
setting user rights, 301
set up, 301
wildcard, 310

O

Office software, 34
Operating system, 21

P

Penguins, 26
Perl, 255
 array, 258
 array assignment, 258
 assign hash, 263
 bioperl, 285
 built-in variables, 266
 calculations, 281
 chomp, 280
 chop, 280
 command line parameter, 271
 conditions, 271

P (*cont.*)

CPAN, 285
 <DATA>, 272
 data input, 271
 data output, 274
 delete, 259, 264
 die, 267
 do...until, 268
 do...while, 268
 dynamic programming, 251
 --END--, 272
 eval, 265
 examples, 286
 escape sequences, 277
 executable, 256
 filehandle, 272
 file modes, 276
 flow control, 266
 for..., 269
 foreach, 260, 265
 foreach..., 269
 grep, 260
 hash database, 276
 hashes, 262
 here documents, 274
 if..., 267
 index, 281
 input from a file, 272
 input from the script, 272
 input from the user, 271
 introduction, 255
 keys, 264
 last, 270
 lc, 280
 length, 281
 Levenshtein distance, 250
 lowercase, 280
 map, 261
 modules, 283
 my, 282
 next, 270
 open, 273
 our, 285
 output to a file, 275
 package, 283
 packages, 283
 pattern match, 278
 pop, 259
 print, 274
 printf, 274
 push, 259
 read, 273
 redo, 270
 regular expressions, 277

require, 284
 reverse, 261, 265
 rindex, 281
 scalar, 262
 scalars, 257
 script file, 256
 shift, 259
 sort, 261
 sprintf, 274
 <STDIN>, 272
 string manipulation, 279
 sub, 282
 subroutines, 282
 substitute, 279
 substr, 281
 text manipulation, 279
 transliterate, 279
 uc, 280
 unless, 267
 unshift, 259
 until..., 268
 uppercase, 280
 use, 285
 user functions, 282
 values, 265
 variables, 256
 warn, 267
 while..., 268

Permissions for files, 56

PHP, 291

pico, 87

Processes, 109

Program installation, 115, 116, 118, 120, 122
 make, 121

Programming languages

AWK, 197, 198, 200, 202, 204, 206, 208,
 209, 210, 212, 214, 216, 218, 220,
 222, 224, 226, 228, 230, 232, 234,
 236, 238, 240, 242, 244, 246, 248,
 250, 252, 254

 Java, 291

 Perl, 255

 PHP, 291

Protein code, 429

Putty, 44

R

R, 317

 abline(), 333, 368–370

 apropos(), 320

 array(), 322

 as.numeric(), 333

 attach(), 321, 368

- `boxplot()`, 328, 371
- `chisq.test()`, 338
- `cor()`, 329
- `else()`, 326
- `for()`, 325
- `head()`, 328
- `help()`, 320
- `hist()`, 333, 368, 369
- `if()`, 326, 327
- `iqr()`, 328
- `is.na()`, 329
- `legend()`, 406
- `length()`, 371
- `library()`, 341, 368, 371
- `lines()`, 319
- `lm()`, 332, 333, 368, 370
- `log2()`, 321
- `lowess()`, 332, 333
- `ls()`, 333
- `matrix()`, 323
- `mean()`, 371
- `names()`, 321
- `na.omit()`, 329
- `pairs()`, 329, 406
- `pdf()`, 331
- `plot()`, 330, 370, 406
- `png()`, 331
- `points()`, 319, 326, 406
- `pwd()`, 333
- `read.table()`, 320, 321, 324, 406
- `seq()`, 319
- `sum()`, 327
- `summary()`, 328, 370
- `system()`, 322, 333
- `t.test()`, 337, 371
- `which()`, 326
- `write.table()`, 321, 322
- array, 322
- basic operations, 318
- chi-square test, 337
- data frames, 324
- data structures, 322
- getting started, 317
- graphical devices, 331
- histogram, 333
- installation, 42
- installing packages, 339
- introduction, 317
- lists, 324
- matrix, 322
- MySQL connection, 339, 368
- packages, 339
- programming structures, 325
- regression analysis, 332

- RMySQL, 339, 368, 371
- save graphics, 330
- script _les, 338
- standard deviation, 337
- t-test, 336
- variable assignment, 318
- vectors, 322
- Rasmol commands, 430
- Regular expressions, 157, 158, 160, 162, 164, 166, 168, 170, 172, 174
 - `agrep`, 173
 - alternation, 167
 - anchors, 166
 - back references, 168
 - character classes, 169
 - `egrep`, 159
 - escape sequence, 158, 167
 - genomics, 172
 - grouping, 165
 - how to use, 160
 - literal, 158
 - metacharacter, 158
 - priorities, 169
 - quantifiers, 164
 - search pattern, 158, 161
 - single characters, 162
 - `tag`, 173
 - target string, 158
 - `vi`, 172
- Relational databases, 296
- Remote connection, 71, 72, 74
 - `curl`, 72
 - `scp`, 72
 - `ssh`, 73
 - `wget`, 71
- `rsync`, 73, 74

S

- `scp`, 72, 74
- Search for files, 66
- Search for text
 - `grep`, 84
- Sed, 175, 176, 178, 180, 182, 184, 186, 188, 190, 192
 - addresses, 180
 - change, 186
 - commands, 181
 - delete, 185
 - examples, 190
 - getting started, 176
 - hold space, 179
 - insert, 186
 - line-oriented, 175

S (*cont.*)

- non-interactive, 175
- pattern space, 178
- print, 189
- read files, 189
- regular expressions, 181
- script files, 188
- stream-oriented, 175
- substitute, 182
- syntax, 179
- transliterate, 184
- write files, 189

Shell

- alias, 104
- aliases, 104
- apropos, 50
- background processes, 111
- bash, 98
- .bash_logout*, 426
- .bash_profile*, 426
- .bashrc*, 426
- bg, 112
- break, 142, 146
- bunzip2, 66
- bzip2, 66
- case..., 145
- cat, 46
- cd, 58
- chattr, 62
- chgrp, 60
- chmod, 60
- chown, 60
- chsh, 99
- command expansion, 134, 136
- command lists, 104
- command substitution, 134
- compress, 66
- conditions, 139
- cp, 56, 58
- crontab, 107
- csh, 98
- date, 46
- debugging, 148
- default, 99
- different types, 98
- double quotes, 137
- echo, 126, 130
- env, 129
- escape sequences, 137
- example shell programs, 152
- exit, 98
- export, 129
- fg, 112
- file attributes, 59

- find, 66
- flow control, 138
- for..., 144
- gtar, 63
- gunzip, 65
- gzip, 65
- here document, 131
- id, 61
- if..., 138
- info, 50
- introduction, 97
- key shortcuts, 99
- kill, 113
- ksh, 99
- line, 132
- logout, 50, 98
- ls, 54
- man, 50
- mkdir, 55
- notifications, 147
- optimizing, 426
- parameter, 133
- passwd, 49
- path, 127
- pipes, 103
- processes, 109
- profile*, 127
- programming, 125, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154
- prompt, 105
- ps, 110
- pwd, 53
- quoting, 136
- read, 132
- redirections, 101
- rm, 58
- schedule commands, 107
- script files, 125
- select..., 146
- seq, 144
- set, 112, 129
- sh, 98
- shift, 142
- single quotes, 137
- sleep, 112, 113
- sort, 47
- start-up files, 426
- stderr, 101, 102
- stdin, 101, 102
- stdout, 101, 102
- syntax, 47
- tar, 62
- tee, 102

- test, 139
 - top, 111
 - trap, 150
 - unalias, 105
 - uncompress, 66
 - unset, 130
 - until..., 143
 - unzip, 65
 - variables, 128, 130
 - variable substitution, 134
 - while..., 141
 - wildcards, 108
 - zip, 65
- Shell programming, 125, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154
- Special character, 431
- Special Files (. and ..), 57
- ssh, 73, 74
- SSH, 44, 73
- sudo, 420
- Surface Racer, 410
- SWISS-MODEL, 378
- Syntax, 47
- Synthetic Biology, 9
- Systems Biology, 9
- T**
- tar, 62
- Telnet, 44
- Text file conversion, 95
 - dos2unix, 95
 - unix2dos, 95
- Text tools
 - cat
 - enter text, 78
 - comm, 85
 - compare file content, 85
 - cut, 83
 - diff, 85
 - grep, 84
 - head, 81
 - less, 81
 - more, 81
 - paste, 83
 - pico, 87
 - realtime monitoring, 81
 - sort, 79
 - split, 82
 - tail, 81
 - uniq, 81
 - vim, 88
 - vim, 88
 - wc (word count), 82
 - zless, 82
- TIGR assembler, 393
- U**
- Ubuntu
 - apt-get, 43
 - bioscience packages, 43
 - installation, 40
 - Jmol, 43
 - packages, 42
- uncompress, 66
- Unix versions, 23
- unix2dos, 95
- unzip, 65
- USB stick, 428
- V**
- Vi, 88
- Vim
 - buffers, 94
 - copy and paste, 93
 - cursor keys, 89, 91
 - delete, 92
 - installation, 89
 - line numbers, 91
 - markers, 94
 - modes, 90
 - move cursor, 91
 - quit, 93
 - regular expressions, 172
 - replace, 92, 94
 - save, 93
 - search, 94
 - short introduction, 89
 - start, 89
 - undo, 92
- VirtualBox, 40
- W**
- wget, 71
- Window manager, 27
- X**
- X-Client, 27
- X-Server, 27
- X-Windows, 26
- Z**
- zip, 65