LEHRBUCH

Thomas Erben

Einführung in Unix/Linux für Naturwissenschaftler

Effizientes wissenschaftliches Arbeiten mit der *Unix*-Kommandozeile

Springer Spektrum

Einführung in Unix/Linux für Naturwissenschaftler

Thomas Erben

Einführung in Unix/Linux für Naturwissenschaftler

Effizientes wissenschaftliches Arbeiten mit der Unix-Kommandozeile

Springer Spektrum

Thomas Erben
Argelander-Institut für Astronomie
Rheinische Friedrich-Wilhelms-Universität
Bonn, Deutschland

ISBN: 978-3-662-50300-3 ISBN: 978-3-662-50301-0 (eBook)
DOI 10.1007/978-3-662-50301-0

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Spektrum

Planung: Margit Maly

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Spektrum ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer-Verlag GmbH Deutschland
Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Vorwort

Dieses Buch entstand aus einem Skriptum der Vorlesung „Einführung in die EDV für Physikerinnen und Physiker“, die ich als Astronom, zusammen mit Kollegen aus der Physik, seit dem Jahre 2006 an der Universität Bonn betreue. Die Veranstaltung richtet sich an die jährlich ca. 150 Studienanfänger und soll nötige praktische EDV-Kenntnisse für ein Bachelorstudium der Physik vermitteln. Dies beinhaltet eine Einführung in *Unix*, welches in den Naturwissenschaften, vor allem im universitären Bereich, das vorherrschende Betriebssystem ist. Des Weiteren lehren wir das Textsatzsystem LaTeX zum Verfassen wissenschaftlicher Texte und führen die Studenten in die Computerprogrammierung mit *Python* ein. Die erworbenen Kenntnisse werden bei der Durchführung von Praktika, bei der späteren Vorlesung „Numerische Methoden der Physik“ und vor allem in der Bachelorarbeit benötigt. Die EDV wurde in Bonn zum Wintersemester 2006/2007 mit Einführung der Bachelor- und Masterstudiengänge in den Lehrplan des Physikstudiums integriert. Dies trägt der Tatsache Rechnung, dass Computer aus der täglichen Arbeit von Physikern (und Naturwissenschaftlern im Allgemeinen) nicht mehr wegzudenken und ein integraler Bestandteil eines jeden Forschungsprojekts sind. Die EDV-Kenntnisse, die ein Student der Naturwissenschaften heute benötigt, gehen über das hinaus, was sich die meisten während des sehr zeitintensiven Bachelorstudiums *nebenher* allein aneignen können. Die explizite Berücksichtigung der EDV im Lehrplan stellt sicher, dass alle Studenten für die Durchführung der Bachelorarbeit und für die weitere wissenschaftliche Laufbahn solide Grundkenntnisse besitzen.

Das vorliegende Buch deckt im Wesentlichen den Stoff des *Unix*-Blocks unserer Veranstaltung ab. Während wir für LaTeX und *Python* auf vorhandene Literatur zurückgreifen, wurde uns bei der stetigen Entwicklung des *Unix*-Blocks über die Jahre klar, dass wir dies hier nicht konnten (oder wollten). Während es zwar eine große Vielfalt an deutschsprachiger Literatur zu dem Thema gibt, existieren nur wenige Bücher, die speziell auf die Bedürfnisse von Naturwissenschaftlern eingehen und auf dem typischen Kenntnisstand von Studienanfängern aufbauen. Vorhandene Literatur mit diesen Vorraussetzungen hatte nicht die Themenauswahl oder die Tiefe, die unseren Vorstellungen entsprach.

Unser *Unix*-Block zielt primär darauf ab, den Studenten innerhalb von drei bis vier Vorlesungswochen ein effizientes wissenschaftliches Arbeiten unter diesem

Betriebssystem zu ermöglichen. Dies reicht von der grundlegenden Benutzung der *Unix*-Kommandozeile bis hin zu einer Einführung von *Unix* als Programmier- und Datenanalyseumgebung. Die ersten Jahre unserer Veranstaltung haben gezeigt, dass viele Studenten dieses Ziel effizienter im Selbststudium direkt vor einem Computer erreichen als mit einer *traditionellen Vorlesung*. Wir haben daher unsere Materialien schrittweise erweitert und letztendlich für das Selbststudium optimiert. Die Studenten erarbeiten wöchentlich einen Teil der Materialien in Heimarbeit und vertiefen den Stoff in Übungen zusammen mit einem Tutor. Für diejenigen Studenten, die eine audio-visuelle Präsentation des Stoffs (Vorlesung) der reinen Schriftform vorziehen, haben wir das Material zusätzlich in Form von Videos aufbereitet. Neben der reinen Vermittlung des Stoffes ist die Ausbildung zur *Selbsthilfe* ein wesentliches Lernziel unseres Kurses. Bei einem so umfassenden Themenkomplex wie *Unix* müssen wir uns bei der vorhandenen Zeit auf die Vermittlung grundlegender Konzepte und einen Ausschnitt der Möglichkeiten beschränken. Es ist daher unvermeidbar, dass sich die Studenten zur Vertiefung des Stoffes oder bei auftretenden Problemen Informationen effizient beschaffen können, z. B. aus dem Internet. Viele der Aufgaben unseres Kurses und dieses Buches zielen, neben einer Vertiefung des Stoffes, explizit auf dieses Lernziel ab. Die vergangenen Jahre haben gezeigt, dass auch Studenten, die unsere Veranstaltung nur mit geringen Vorkenntnissen besucht haben, die Lernziele gut erreichen können. Meine Hauptmotivation für dieses Buch ist es, die Erkenntnisse und Methoden zur Vermittlung von *Unix*-Kenntnissen an Naturwissenschaftler, die ich mit meinen Kollegen in den letzten zehn Jahren gesammelt habe, auch anderen zugänglich zu machen.

Danksagungen: Mein herzlicher Dank gilt allen, die auf die eine oder andere Weise an der Entstehung dieses Buches beteiligt waren.

Mein besonderer Dank geht an meine Kollegen Ian Brock, Wolfgang Ehrenfeld, Klaus Lehnertz, Jörg Pretz, Oliver Cordes, Michael Döring, Philip Bechtle, Thomas Velz und an alle Tutoren, mit denen ich in den letzten zehn Jahren die Veranstaltung „Einführung in die EDV“ halten und entwickeln durfte. Die sehr freundschaftliche und bereichernde Zusammenarbeit bei dieser Vorlesung hat meine didaktischen Vorstellungen nachhaltig geprägt.

An dieser Stelle möchte ich meiner Mutter, Frau Christa Erben, danken, die mich immer bedingungslos bei allen meinen Projekten unterstützt!

Tatkräftige Unterstützung bei der Fertigstellung des Buches habe ich von Ian Brock, Dominik Klaes, Ole Marggraf und Peter Schneider erhalten. Sie haben das gesamte Manuskript *sehr* ausführlich gelesen, mich auf viele Fehler und Ungenauigkeiten aufmerksam gemacht und mir zahlreiche und konstruktive Verbesserungsvorschläge zukommen lassen.

Explizit bedanken möchte ich mich auch bei den Studenten, die über die Jahre unsere Veranstaltung besucht haben. Dies gilt nicht nur für all diejenigen, die stets motiviert waren und mir Freude bei der Vermittlung des Lehrstoffes bereitet haben, sondern auch für diejenigen, die Schwierigkeiten mit dem Stoff hatten. Die Probleme dieser Gruppe sind eine ständige Motivation zur Verbesserung und zur Erkundung neuer Wege.

Für die Unterstützung des Springer-Verlags bei meinem ersten Buchprojekt und die fachkundige und angenehme Zusammenarbeit mit Frau Sabine Bartels, Frau Margit Maly und Frau Vera Spillner bin ich sehr dankbar. Frau Tatjana Strasser danke ich für ihre sehr akribische Korrektur von Rechtschreibung, Grammatik und Stil.

Bonn im Juni 2016 Thomas Erben

Inhaltsverzeichnis

Einführung 1

Übersicht

Zusammenfassung

In diesem Kapitel beschäftigen wir uns mit der Frage, warum in Naturwissenschaften häufig das Betriebssystem *Unix* und nicht etwa *Microsoft Windows* verwendet wird. Des Weiteren diskutieren wir wichtige Unterschiede und Gemeinsamkeiten beider Betriebssysteme. Zum Schluss des Kapitels erläutere ich Ihnen die Ziele des Buches.

Die meisten von Ihnen lesen diese Zeilen, weil Sie sich innerhalb Ihres Bachelorstudiums mit Computern, und speziell mit dem Betriebssystem *Unix*, auseinandersetzen müssen. Wahrscheinlich haben Sie bereits Erfahrung als Computeranwender unter dem Betriebssystem *Microsoft Windows*, und auf dieser Basis möchte ich Sie in die *Unix*-Welt begleiten.

Das Buch setzt Kenntnisse voraus, die ein regelmäßiger Benutzer von *Microsoft Windows* mitbringt. Dies umfasst den Umgang mit Tastatur und Maus, die Bedienung von Programmen, die auf grafischen Benutzerelementen (Maus, Fenster und Icons) beruhen, und den Umgang mit Dateien und Ordnern/Verzeichnissen. Sie wissen außerdem, wie Sie auf Ihrem Rechner neue Programme installieren und konfigurieren. Des Weiteren gehe ich davon aus, dass Sie sich selbstständig Informationen aus dem Internet beschaffen können (Google, Wikipedia usw.) und genügend Fremdsprachenkenntnisse besitzen, um auch mit englischen Texten und

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_1

Webseiten zurechtzukommen. Bei der Einführung in das Betriebssystem *Unix*, das für die meisten von Ihnen sicher neu ist, werde ich, wenn immer möglich, Analogien und Unterschiede zu *Microsoft Windows* herausarbeiten, um Ihnen den Einstieg zu erleichtern.

1.1 Warum *Unix*?

Im naturwissenschaftlichen Bereich wird hauptsächlich das *Betriebssystem Unix* eingesetzt. *Unix* wurde in den 1970er-Jahren entwickelt, und damals gab es noch keine *grafischen Benutzeroberflächen (Graphical User Interface; GUI)* mit einer Maus als Eingabegerät oder Ähnliches. Einem Rechner wurden Kommandos über eine Tastatur gegeben, und die Ausgabe erfolgte mit einfachem Text auf Papier und später auf Bildschirmen. Computersysteme, die hauptsächlich auf Tastaturinteraktion beruhen, werden auch als *Command Line Interfaces (CLI)* bezeichnet. Es mag etwas verwundern, dass wir uns im 21. Jahrhundert noch mit einem Betriebssystem, welches auf einer Schreibmaschinen-Ein- und -Ausgabe beruht, beschäftigen, wo wir doch hochauflösende Farbbildschirme und ausgefeilte GUIs zur Verfügung haben. Einer der Hauptgründe ist, dass textbasierte Anwendungen deutlich flexibler sind als z. B. mausgesteuerte Vorgänge, dass sie mit deutlich weniger Aufwand zu programmieren sind, und vor allem, dass sie sich deutlich einfacher *automatisieren* lassen; siehe hierzu auch Abschn. 1.3 weiter unten. Natürlich bedient man sich auch in der *Unix*-Welt heute der Annehmlichkeiten grafischer Benutzeroberflächen. Die primäre Interaktion mit dem System findet nach wie vor über die Tastatur statt, aber die Eingabeterminals sind *normale Fensteranwendungen* innerhalb einer Desktop-Umgebung; siehe auch Abb. 1.1. Ein zweiter Hauptgrund für die Attraktivität von *Unix* im wissenschaftlichem Bereich ist, dass das Betriebssystem selbst, und die allermeisten Anwendungsprogramme, *kostenlos* und sogar im Quellcode für jedermann frei verfügbar ist. Oft vorhandene, ärgerliche „Nebenerscheinungen“ bei der legalen Beschaffung von *Microsoft Windows*-Software existieren unter *Unix* nicht. Hierzu gehören: Probierversionen mit Werbebannern und limitierter Testphase, kostenpflichtige Vollversionen mit oft umständlichen und eingeschränkten Nutzungsbedingungen, Registrierungen mit der Folge von Werbe- und Spam-Mails. Ein System, das einerseits kostenlos ist, und andererseits die Möglichkeit bietet, es nach Belieben auf eigene Bedürfnisse anzupassen, ist für die Wissenschaft sehr attraktiv.

1.2 Wichtige Gemeinsamkeiten und Unterschiede zwischen *Unix* und *Microsoft Windows*

Im Folgenden möchte ich einige für Sie wichtige Gemeinsamkeiten und Unterschiede beim Arbeiten zwischen *Unix* und *Microsoft Windows* näher beleuchten. Vieles davon hat direkten Einfluss auf die Themenauswahl dieses Buches:

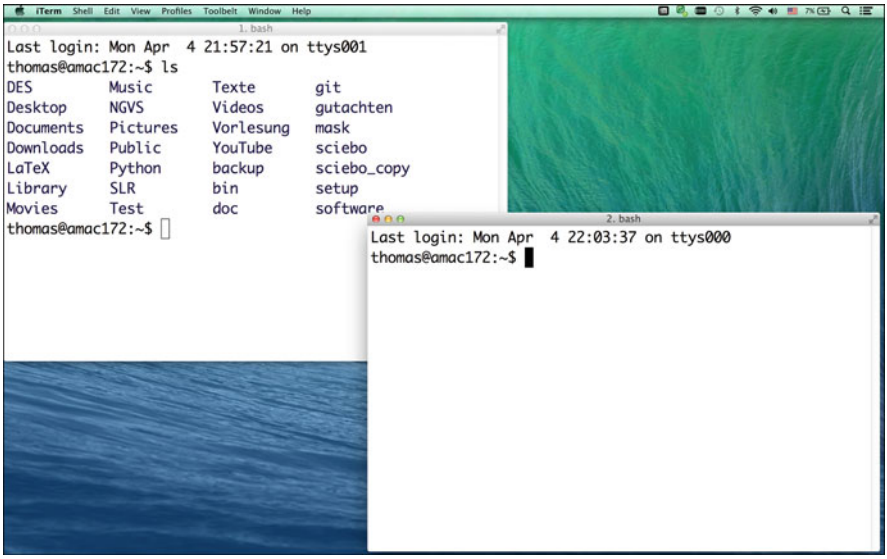

Abb. 1.1 Die Interaktion mit *Unix* findet hauptsächlich über sogenannte *Terminals* (Schreibmaschinenprinzip) statt. Diese Terminals sind heute als Programme in Desktop-Umgebungen integriert

1. Fenstergesteuerte Anwendungen sind unter *Unix* so gut wie identisch zu benutzen wie unter *Microsoft Windows*. Dies sind z. B. ein grafischer Filemanager mit Ordnersymbolen oder der Webbrowser. Viele Anwendungen, die Sie von *Microsoft Windows* her kennen, gibt es in ähnlicher Form auch unter *Unix*. Mit *LibreOffice* (früher *OpenOffice*) steht z. B. ein zu *Microsoft Office* ähnliches Office-Paket zur Verfügung. Wie bereits erwähnt, haben die *Unix*-Varianten hier den großen Vorteil, dass sie *kostenlos* sind und einfach und legal aus dem Internet heruntergeladen werden können. Wie Sie diese Programme bedienen, werde ich in diesem Buch nicht vertiefen. Sie sollten hier mit Ihren *Microsoft Windows*-Kenntnissen keinerlei Probleme haben. Wir konzentrieren uns in diesem Buch auf die *Unix*-Benutzung über die Kommandozeile.
2. Das primäre Medium zur Kommunikation mit *Unix* ist die Tastatur. Im Prinzip kann man ein *Unix*-System *komplett* damit bedienen; auch Anwendungen wie das Surfen im Web lassen sich im Prinzip darüber realisieren – hierfür wird aber natürlich auch in der *Unix*-Welt ein Webbrowser mit grafischer Benutzeroberfläche verwendet. Während sich der Aufwand, die CLI-Arbeitsweise zu erlernen, für Sie am Ende definitiv auszahlt, muss ganz klar gesagt werden, dass *anfangs* der Lernaufwand höher ist als für GUI-Anwendungen. Um ein *Unix*-Kommando auf der Tastatur auszuführen, müssen Sie natürlich dessen Namen und gegebenenfalls Parameter erst kennen oder in der Hilfe nachschlagen. Das bekannte *Alle vorhandenen Bedienelemente durchklicken/ausprobieren, bis passiert, was ich will* steht hier für die allermeisten Aufgaben nicht mehr zur Verfügung!

3. In der *Unix*-Welt ist üblicherweise *alles* auf Englisch. *Unix*-Kommandos sind typischerweise Abkürzungen für englische Worte, z. B. `ls` für *list* (Verzeichnisinhalt anzeigen), `cp` für *copy* (Dateien/Verzeichnisse kopieren) usw. Die Systemdokumentation liegt sehr oft nicht auf Deutsch vor, und auch Online-Texte im Internet sind meist auf Englisch aktueller als die in anderen Sprachen – wenn es sie denn überhaupt gibt. Wahrscheinlich haben Sie auch auf einem *Unix*-Rechner an Ihrer Universität eine englische Tastatur vor sich!

Wichtig

Deutsche Umlaute in *Unix*-Dateinamen

Vermeiden Sie es von Anfang an, in der *Unix*-Welt in Datei- und Verzeichnisnamen deutsche Sonderzeichen zu verwenden! Ein häufiges Phänomen ist z. B. folgendes: Jemand legt mit einer deutschen Tastatur auf seinem *Unix*-Account ein Verzeichnis mit dem Namen `Übung` an. Dann kommt er an einen Rechner mit englischer Tastatur und möchte etwas wie `cd Übung` (`cd` steht für *change directory*) eingeben. Da deutsche Umlaute auf einer englischen Tastatur nicht direkt zugänglich sind, haben viele Anfänger hier zunächst ein Problem. In Anhang C besprechen wir Möglichkeiten, wie Sie auch in einer englischen *Unix*-Umgebung deutsche Sonderzeichen benutzen können.

4. Das wichtigste Dateiformat für alles, was mit Texten oder Konfigurationen zu tun hat, ist unter *Unix* das simple ASCII-Textformat. Unter *Microsoft Windows* erkennen Sie diese Dateien meist an der Endung „`.txt`“, und Sie können sie dort mit einem *Texteditor* wie *Notepad* bearbeiten. Während das Format unter *Microsoft Windows* eine eher untergeordnete Rolle spielt, ist es in der *Unix*-Welt zentral. Auch viele wissenschaftliche Daten (z. B. Datentabellen) oder der Quelltext dieses Buches ist in diesem Format realisiert. Das von *Microsoft Windows* bekannte Problem vieler *proprietärer* Dateiformate, die sich nur mit dem Programm des Herstellers (und oft auch nur mit bestimmten Versionen desselben) bearbeiten lassen, existiert unter *Unix* in dieser Form nicht. Viele *Unix*-Programme befassen sich daher mit der effektiven Erstellung und Bearbeitung dieses Dateiformats.
5. Unter *Microsoft Windows* sind Sie sicher gewohnt, vor allen kritischen Aktionen nochmal gefragt zu werden, z. B. *Wollen Sie die Datei wirklich löschen? Unix* ist generell ein *ruhiges* Betriebssystem. Ein Hauptgrund dafür ist, dass man in den Anfangszeiten von *Unix* Ressourcen schonen und *unnötige* Ein- und Ausgaben vermeiden wollte. Seien Sie sich bewusst, dass *Unix* gegebene Befehle einfach ausführt und, falls der Befehl nicht zwingend eine Ausgabe nach sich zieht, auch nichts quittiert wird. Beachten Sie auch, dass Sie in der Regel selbst bei den kritischsten Befehlen nicht um eine Bestätigung gebeten werden. Ein Befehl wie `rm datei` (`rm` steht für *remove*) löscht `datei` ohne nochmal nachzufragen. Beachten Sie insbesondere im Bezug auf `rm`, dass es unter der *Unix*-Kommandozeile keinen *Papierkorb* gibt, aus dem man irrtümlich gelöschte

Dateien notfalls wiederherstellen kann. Unter *Unix* einmal gelöschte Dateien und Verzeichnisse sind wirklich weg.

6. Wahrscheinlich haben Sie einen eigenen Rechner, den nur Sie, eventuell zusammen mit anderen Familienmitgliedern, benutzen. Wenn Sie den Rechner nicht verwenden, wird dieser normalerweise abgeschaltet. *Unix*-Systeme in größeren Einrichtungen wie z. B. an der Universität laufen oft 24 Stunden am Tag, und sie werden in der Regel von vielen Benutzern, oft sogar gleichzeitig, verwendet. Des Weiteren ist auf *Unix*-Systemen ein *kollaboratives* Arbeiten üblich. An wissenschaftlichen Projekten arbeiten oft viele Leute zusammen, und es ist vernünftig, dass jeder auf gemeinsame Projektdaten zugreifen darf. Viele Systeme gehen so weit, dass alle Dateien, die ein Benutzer anlegt, per Default für *alle Leute*, die Zugang zu der betreffenden Maschine haben, lesbar sind. Für Sie haben diese Punkte folgende praktische Konsequenzen bei der Arbeit mit *Unix* in einem vernetzten System (z. B. an Ihrer Universität):
   a) Sie müssen sich auf jedem *Unix*-System zwangsweise als Benutzer mit einem Passwort anmelden.
   b) Sie müssen lernen, wie Sie sich über verfügbare Ressourcen ihres Systems informieren und wie Sie diese effektiv benutzen. Sie sollten z. B. wissen, wie Sie ein bestimmtes Programm identifizieren und eventuell aus dem System entfernen, wenn Sie merken, dass beim Aufruf eines einwöchigen Programmlaufs ein Fehler passiert ist.
   c) Sie müssen sich mit dem *Unix*-Rechtesystem auskennen. Sie sollten erkennen, welche Rechte andere Benutzer an Ihren Daten haben und wie Sie gegebenenfalls Rechte an sensiblen Daten entziehen!
7. Sie sind es von Ihrem *Microsoft Windows*-System wahrscheinlich gewohnt, neue Programme einfach und überall installieren und verändern zu können. Auf den meisten *Microsoft Windows*-Systemen ist der sogenannte *Administrator*, der uneingeschränkte Rechte am System hat, meist identisch mit einem regulärem Benutzer-Account. Da *Unix* von Anfang an als *Multibenutzersystem* entwickelt wurde, ist hier auch stets eine klar definierte *Benutzerhierarchie* vorhanden. Auf jedem *Unix*-System gibt es den speziellen Nutzer `root`. Dieser entspricht dem *Microsoft Windows*-Administrator. Prozesse unter dem `root`-Account haben vollen Zugriff auf das System:

   – Lese-/Schreibzugriff auf alle Dateien,
   – Zugriff auf die gesamte Hardware,
   – `root` darf beliebige Prozesse beeinflussen,
   – `root` kann Benutzerpasswörter *ändern*, sie aber *nicht* einsehen.

   Der `root`-Account wird unter *Unix* lediglich zur Systemwartung, aber nicht zur regulären Arbeit, verwendet. Er kann auch nicht identisch mit einem regulären Benutzer-Account sein. Auch wenn Sie *Unix* auf einem eigenen Rechner installieren sollten, wird Ihr Benutzer-Account immer von `root` getrennt sein!

1.3 GUI versus CLI: Ein konkretes Beispiel

Wir wollen die unterschiedlichen Arbeitsweisen zwischen einem GUI und einem CLI-basierten System an einem konkreten Beispiel näher diskutieren. Dies soll Ihnen verdeutlichen, unter welchen Umständen die eine oder andere Methode für eine Problemlösung besser geeignet ist.

In Abb. 1.2 ist ein einfacher, GUI-basierter Taschenrechner abgebildet. Er ist intuitiv und ohne zeitaufwendiges Studium von Dokumentationen für einfache Rechnungen zu bedienen. Es gibt hier keinerlei Überraschungen, und die Dokumentation schreibt in einem Abschnitt auch: „*The numbered keys, the +/− key, and the +, −, ∗, /, and = keys all do exactly what you would expect them to.*“. Der gesamte Funktionsumfang des Rechners ist auf einen Blick sichtbar und leicht zugänglich.

Ein CLI-Programm für Berechnungen auf der *Unix*-Kommandozeile ist das Programm `bc (basic calculator)`. Im Folgenden gebe ich einige *Unix*-Befehle, um mit `bc` einfache Berechnungen durchzuführen. Bitte lassen Sie sich davon momentan nicht abschrecken. Die Konzepte, die zum Verständnis der Eingaben nötig sind, werden wir schrittweise erarbeiten. Hier die Beispiele:

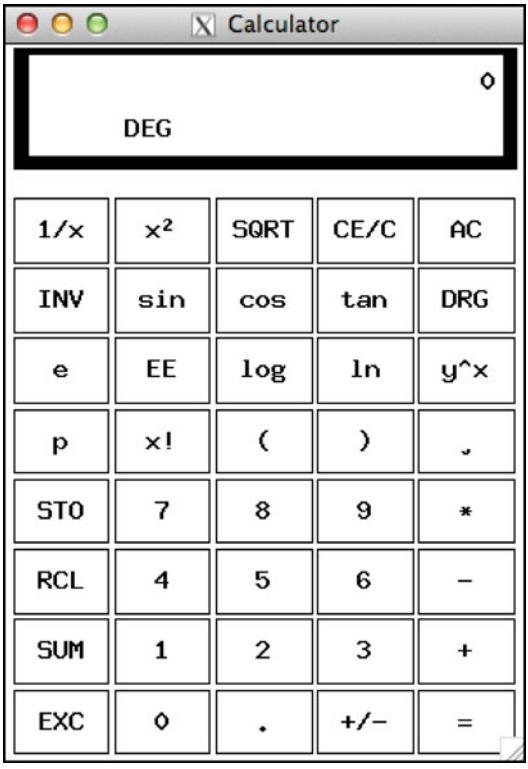

Abb. 1.2 `xcalc` ist ein einfaches Taschenrechnerprogramm, das ohne große Erklärung und Studium von Dokumentation von jedermann benutzt werden kann

```
user$ echo "3.0 + 4.0" | bc -l   <Return>
7.0

user$ echo "6.0 * sqrt(15.0) + 15.34" | bc -l  <Return>
38.577900077244501311O2
```

Hinweise: (1) Falls Sie bereits an dieser Stelle vor einem *Unix*-Terminal sitzen und die Beispiele nachvollziehen wollen, so müssen Sie alles nach dem *Prompt* (hier `user$`) eintippen und die Eingabezeilen mit der *Enter*-Taste (`<Return>`) abschließen. Nach Eingabe von `<Return>` werden die Befehle ausgeführt. Dieses abschließende `<Return>` wird im Folgenden nicht mehr explizit aufgeführt. Unsere Konventionen zur Darstellung von *Unix*-Texteingaben besprechen wir aber noch ausführlich in Abschn. 2.3; (2) beachten Sie, dass Dezimalzahlen mit der englischen Konvention eines *Dezimalpunktes* und nicht mit dem deutschen *Dezimalkomma* dargestellt werden!

Die zwei gegebenen *Unix*-Befehle berechnen die Ausdrücke $3 + 4$ und $6\sqrt{15}+15.34$. Zunächst einmal werden Ihnen die Eingaben `echo "3.0 + 4.0" | bc -l` und `echo "6.0 * sqrt(15.0) + 15.34" | bc -l` recht *kryptisch* vorkommen, und es ist einem Anfänger nicht einsichtig, wie sie zur Berechnung der Ergebnisse führen. Es ist klar, dass `bc` ohne *Unix*-Kenntnisse und ohne einen Blick in die Dokumentation (oder Erklärungen) nicht nutzbar ist. Auch der komplette Funktionsumfang (welche Funktionen kann ich berechnen, wie werden sie angesprochen) erschließt sich nur nach einem Studium der Dokumentation oder nach einer Web-Recherche. Das hier Geschilderte ist charakteristisch für CLI-Anwendungen und der Hauptgrund, warum viele *Microsoft Windows*-Nutzer der *Unix*-Kommandozeile anfangs sehr reserviert gegenüberstehen. Auch ist die GUI-Herangehensweise für sehr viele Probleme definitiv die bessere Wahl, und es besteht kein Grund, sich in *Komplizierteres* einzuarbeiten, um einen Rechner effektiv zu nutzen. Es hat einen guten Grund, dass *Microsoft Windows*, vor allem im privaten Bereich, einen Marktanteil von über 80 % hat! Jedoch haben CLI folgende Eigenschaften und Vorteile, die vor allem in der Wissenschaft zum Tragen kommen:

- Wenn Sie mit `xcalc` eine längere Sequenz an Rechnungen eingeben, können Sie sehr schlecht überprüfen, ob Ihre Eingabe *korrekt* ist. Können Sie auf einen Blick am Ergebnis sehen, ob Sie $6.0\sqrt{15.0} + 15.34 \approx 38.58$, oder $6.0(\sqrt{15.0} + 15.34) \approx 115.28$ eingetippt haben? Bei CLI Eingaben wie

  ```
  user$ echo "6.0 * sqrt(15.0) + 15.34" | bc -l
  38.577900077244501311O2
  ```

  haben Sie typischerweise die komplette Eingabe direkt am Bildschirm (oder in Textdateien) stehen und können sie leicht überprüfen.
- Ähnlich ist es, wenn Sie ein und dieselbe Rechnung noch einmal mit leicht veränderter Eingabe durchführen wollen, z. B.:

```
user$ echo "4.0 * sqrt(15.0) + 15.34" | bc -l
30.831933848296675406 8
```

Bei CLI können Sie einfach die alte Eingabe zurückholen, leicht modifizieren und das Kommando erneut ausführen. Bei `xcalc` müssen Sie die gesamte, fehleranfällige Sequenz wieder komplett *einklicken*.

- CLI-Programme können sehr einfach automatisiert werden, falls eine ähnliche Aufgabe *immer wieder* durchgeführt werden soll. Als Beispiel haben wir eine Textdatei `zahlen.txt` mit Zahlen

```
15.34
16.39
.
.
```

vorliegen. Wir wollen jetzt zu jeder Zahl die dritte Potenz berechnen. Mit `bc` kann man dies z. B. folgendermaßen lösen:

```
user$ for N in $(cat zahl.txt); do \
      echo "${N}^3" | bc -l; done
3609.741304
4402.880119
.
.
```

Hinweis: Der Backslash (\) am Ende der ersten Eingabezeile ist *nicht* mit einzutippen! Er steht hier, um zu signalisieren, dass die Eingabe in der nächsten Zeile weitergeht.

Auch der letzte Befehl ist für Sie zunächst wieder ein kryptisches Gebilde aus Zeichenfolgen, das Sie momentan nicht zu verstehen brauchen. Wir bauen eine Iteration über die einzelnen Zeilen in der Datei `zahl.txt` auf und bilden mit `bc` die dritte Potenz der einzelnen Zahlen. Solche Iterationen sind mit allen textbasierten Programmen leicht zu realisieren, und *jedes* Programm kann somit sofort zur *Massenproduktion* verwendet werden. Mit einem GUI-Programm wie `xcalc`, das diese Funktionalität nicht von Anfang an vorgesehen hat, ist diese Aufgabe ab einer gewissen Zahlenmenge nicht mehr praktikabel zu lösen.

- Da die Schnittstelle für sehr viele *Unix*-Programme einfache Textdateien sind, ist es trivial, Ergebnisse eines Programms sofort mit einem anderen weiterzuverarbeiten. Oben könnte die Liste der mit `bc` berechneten Zahlen sofort mit einem anderen Programm (z. B. einem zur grafischen Darstellung der Häufigkeitsverteilung) weiterverwendet werden. Hingegen ist es nicht möglich, Daten in `xcalc` zu importieren, oder Ergebnisse zur Weiterverarbeitung zu exportieren.
- Durch die Integration aller Funktionalität in eine GUI sind diese natürlich in ihrem Anwendungsbereich beschränkt. `bc` bietet beispielsweise die Möglichkeit, Rechnungen *mit beliebiger Genauigkeit* durchzuführen:

```
user$ echo "scale=200; sqrt(2.0)" | bc -l
1.41421356237309504880168872420969807856967187537 6\
94807317667973799073247846210703885038753432764157\
27350138462309122970249248360558507372126441214970\
99935831413222665927505592755799950501152782060571 47
```

Hier wird $\sqrt{2}$ auf 200 Stellen genau berechnet. Wenn Sie noch mehr Stellen haben möchten, so erhöhen Sie einfach die Zahl nach der `scale=`-Zeichenkette. Es ist sicher einleuchtend, dass dies wegen der Ausgabe in `xcalc` nur umständlich zu integrieren wäre. Auch hat man Probleme, wenn man noch viele neue Funktionen (hyperbolische Funktionen usw.) zur Verfügung stellen wollte, ohne das GUI komplett zu überarbeiten. In einem CLI-System verändern solche Modifikationen die Benutzung des Programms in keiner Weise. Hier kann man sich komplett auf die Implementationen neuer Funktionalität konzentrieren, ohne ständig die Benutzeroberfläche mit umgestalten zu müssen. Dieser Punkt wird für Sie relevant, wenn Sie einmal selbst Analyseprogramme entwickeln müssen.

Das Beispiel macht das Spannungsfeld zwischen einfacher und intuitiver Bedienung eines Programms (GUI) und *weit größerer* Anwendungsmöglichkeiten auf Kosten von mehr Einarbeitungszeit (CLI) sehr deutlich. Es zeigt auch, dass sich beide Herangehensweisen *sehr gut* ergänzen und man diejenige wählen sollte, die sich für ein gegebenes Problem am besten eignet. So würde jeder für einfache Rechnungen *schnell* `xcalc` verwenden. Falls jedoch Überprüfbarkeit von komplexen Ein- und Ausgaben, oftmalige Wiederholung ähnlicher Berechnungen, oder die mögliche Weiterverwendbarkeit von Ergebnissen wichtig sind, wäre zwischen `xcalc` und `bc` letzteres die bessere Wahl. Die Stärken der CLI machen auch deutlich, warum sie gerade in den Naturwissenschaften eine starke Verbreitung besitzen.

1.4 *Unix* versus *Linux*

Bisher habe ich immer von *Unix* gesprochen. Das Buch heißt jedoch „Einführung in *Unix*/*Linux* für Naturwissenschaftler“. Wie bereits erwähnt, ist *Unix* sehr alt und es gibt inzwischen sehr viele Weiterentwicklungen und *Unix-ähnliche* Betriebssysteme. *Linux* ist eines davon. Es wurde Anfang der 1990er-Jahre entwickelt und ist inzwischen das vorherrschende *Unix*-System auf Personalcomputern. Es gibt Unterschiede zwischen den verschiedenen *Unix*-Varianten, welche für dieses Buch aber *größtenteils* unwichtig sind. Sie machen sich für Sie hauptsächlich dadurch bemerkbar, dass die Ausgabe von *Unix*-Kommandos oder Abbildungen bei Ihnen etwas anders aussehen können als bei mir. Das ist aber im Wesentlichen alles. Die in diesem Buch besprochenen Programme existieren auf allen *Unix*-Systemen, oder können dort mit geringem Aufwand installiert werden. Manche Programme können sich in ihren Optionen und ihrem Funktionsumfang auf verschiedenen Systemen unterscheiden. Dies betrifft nicht nur verschiedene *Unix*-Varianten, sondern auch

verschiedene Versionen eines Systems. *Unix* und *Linux* werden stets weiterentwickelt! Wo dies wichtig ist, weise ich aber explizit darauf hin. Ich werde im Folgenden die Begriffe *Unix* und *Linux* synonym verwenden. Ich selbst habe für dieses Buch sowohl auf verschiedenen *Linux*-Versionen als auch unter dem *Unix*-System OSX (*Apple Macintosh*) gearbeitet.

1.5 Ziele des Buches

Mein Hauptziel ist es, Ihnen eine solide Basis für die *Kommandozeilennutzung* eines *Unix*-Systems zu vermitteln. Es ist hier wichtig zu betonen, dass es sich bei CLI und GUI um zwei verschiedene Methoden handelt, um mit einem Computer Arbeit zu erledigen. Sie stehen nicht in Konkurrenz zueinander, sondern ergänzen sich hervorragend. Viele Leute haben die verkehrte Assoziation „*Microsoft Windows* ist GUI“ und „*Unix* ist CLI“. CLI und GUI sind jedoch *nicht* an bestimmte Betriebssysteme gebunden. Man kann auch unter *Microsoft Windows* mit einem CLI arbeiten – auch wenn das kaum jemand tut. Hierzu stehen z. B. die Programme *Eingabeaufforderung* und *PowerShell*[1] zur Verfügung. Genauso lässt sich heute ein *Unix*/*Linux*-System verwenden, ohne je mit einem CLI in Berührung zu kommen. Ein weitverbreitetes Vorurteil ist, dass Sie *Unix*, bzw. ein CLI erlernen sollen, da man hiermit Dinge tun kann, die mit *Microsoft Windows* nicht lösbar sind. Prinzipiell ist jedoch jedes Problem, das mit einem GUI lösbar ist, auch mit einem CLI lösbar, und umgekehrt. Ich beziehe mich hier natürlich nicht auf Programme und Anwendungen, die eventuell *nur* für ein bestimmtes Betriebssystem verfügbar sind; es geht hier nur um die Funktionalität, die ein Betriebssystem zur Verfügung stellt. Allerdings sind Probleme, die mit einer Methode leicht zu lösen sind, mit der anderen Methode unter Umständen nur kompliziert, umständlich oder mit nicht vertretbarem Zeitaufwand zu erledigen.

Sie sollten das Erlernen der CLI-Herangehensweise als essenzielles Werkzeug betrachten, um die Effizienz bei der Erledigung wissenschaftlicher Arbeiten mit einem Computer signifikant zu erhöhen! Dies wird Ihnen später auch unter *Microsoft Windows* zugute kommen. Sie können z. B. mit *Cygwin* (siehe Abschn. 3.1) *Unix*-Werkzeuge und ihre Vorteile direkt in Ihr *Microsoft Windows*-System integrieren.

Keines der hier behandelten Themen wird in diesem Buch *komplett und erschöpfend* behandelt. Bei vielem kann ich die existierenden Möglichkeiten nur *andeuten*. Deutlich wichtiger ist mir die Vermittlung der *grundlegenden Ideen und Prinzipien*, die hinter einzelnen Konzepten stehen. Mit diesem Wissen sollte es für Sie sehr einfach sein, *Unix*-Literatur zu verstehen und sich eigenständig tiefer gehende Kenntnisse zu einzelnen Themen zu beschaffen.

Viel des hier dargestellten Stoffes lässt sich auch anders behandeln, als ich es getan habe. Hier habe ich mich auf jeweils *eine* Möglichkeit konzentriert, die leicht zu vermitteln ist, und die ich für Sie als Anfänger für besonders geeignet halte. An

[1] siehe z. B. https://en.wikipedia.org/wiki/Windows_PowerShell.

vielen Stellen hat dies zur Folge, dass ich auf *Details und Feinheiten* nicht eingehe oder dass meine Darstellung nicht 100 % präzise ist! Sobald Sie mehr Erfahrung mit *Unix* bekommen, werden Sie sich automatisch mit vielen Themen tiefer und eingehender beschäftigen, als es für eine erste Einführung notwendig und sinnvoll ist.

Ein weiteres Anliegen ist eine deutliche, und ausführliche Behandlung von typischen Fallstricken und Fehlerquellen, mit denen Anfänger oft zu kämpfen haben.

Das HOWTO zu diesem Buch

2

Übersicht

Zusammenfassung

Es sollen in diesem Kapitel die Themenauswahl des Buches, sein Aufbau, und Konventionen für den benutzten Schriftsatz erklärt werden. Des Weiteren gebe ich einige Hinweise, wie Sie mit dem Buch arbeiten sollten. Das Kapitel stellt also ein kleines *HOWTO* zum Buch dar. Dem Begriff HOWTO (frei übersetzt: Ein „Wie macht man“) werden Sie in der *Unix*-Welt sehr oft begegnen. Er steht für eine kompakte, aber üblicherweise vollständige Anleitung.

2.1 Themenauswahl

Der Hauptteil des Buches ist in 15 aufeinander aufbauende Kapitel untergliedert. Sie werden von der Installation eines *Linux*-Systems auf einem *Microsoft Windows*-Rechner, über die grundlegende Benutzung der *Unix*-Kommandozeile, bis zu den Anfängen der sogenannten *Shell-Programmierung* von mir begleitet. Die primäre Zielgruppe des Buches sind Studenten naturwissenschaftlicher Fächer am Anfang ihres Studiums. Während es keinen Zweifel daran gibt, dass *jeder* Student eines naturwissenschaftlichen Fachs zu Anfang seines Studiums *etwas Unix* lernen muss, gehen die Meinungen über das „Wie viel ist *etwas Unix*?“ recht weit auseinander. Den ersten Kontakt zu *Unix* bekommen viele bei der Durchführung von Praktika oder bei der Mitarbeit an einem Forschungsprojekt. Spätestens bei der Bachelorarbeit werden es die allermeisten von Ihnen mit einem *Unix*-System zu tun haben. Wie viele *Unix*-Kenntnisse hierfür nötig sind, hängt primär von dem Themenbereich Ihres Forschungsprojekts ab. Müssen Sie unter *Unix* große

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_2

Datenmengen eines Experiments auswerten, so benötigen Sie im Allgemeinen *deutlich* tiefer gehende Kenntnisse als z. B. jemand, der Ergebnisse einer Literaturrecherche zusammenschreiben muss. Des Weiteren lassen sich viele Aufgaben, welche Datenextraktion und Manipulation aus Textdateien betreffen, sowohl mit *Unix*-Werkzeugen als auch mit einer Programmiersprache lösen. Hierin liegt ein Hauptgrund dafür, dass das Erlernen fortgeschrittener *Unix*-Techniken und *Unix*-Programme oft nicht als notwendig erachtet und meines Erachtens *vernachlässigt* wird. Neben *Unix* ist das Erlernen einer Programmiersprache für die allermeisten Studenten ohnehin nötig. Meine Erfahrung ist, dass einfache Aufgaben der Datenanalyse, die mit *Unix*-Werkzeugen erledigt werden können, damit *am effektivsten und schnellsten* vonstatten gehen. Eine Beschäftigung damit zahlt sich für Sie also auf alle Fälle aus!

Ich selbst bin im Bereich der beobachtenden optischen Astronomie tätig, und meine Haupttätigkeit ist die Analyse und Auswertung *sehr* großer Mengen astronomischer Daten. Hierbei nutze ich *Unix* nicht nur, um Programme auszuführen, sondern auch als Programmier- und Entwicklungsumgebung. Des Weiteren betreibe ich umfangreiche und komplexe Datenanalyseverfahren, welche zentral auf *Unix*-Werkzeugen der Kommandozeile beruhen. Der hier vermittelte Stoff orientiert sich an dem, was ich als *Unix*-Einführung für Bachelor- und Masterstudenten in meinem Bereich für sinnvoll und notwendig erachte. Im Sinne des oben Diskutierten deckt der Umfang die Bedürfnisse der allermeisten Studenten sehr gut ab und geht für viele sicher über das *absolut notwendige Minimum* hinaus.

Im Einzelnen enthält das Buch folgende Themenbereiche:

1. In Kap. 3 diskutiere ich Möglichkeiten, wie Sie sich eine *Unix*/*Linux*-Umgebung zur Arbeit mit diesem Buch aufsetzen können. Des Weiteren werden Sie Ihre ersten Schritte auf dem *Unix*-Terminal gehen.
2. Die Kap. 4 bis 9 behandeln Kernthemen, um mit der *Unix*-Kommandozeile effektiv arbeiten zu können. Sie lernen dort etwas über die Struktur von *Unix*-Befehlen, das *Unix*-Dateisystem und wie Sie Programme unter *Unix* ausführen und verwalten können. Die hierin eingebetteten Kapitel über das *Unix*-Rechtesystem und über den effektiven Umgang mit der Shell mögen weniger wichtig erscheinen. Sie sind es aber nicht! Da *Unix*-Rechner im Allgemeinen innerhalb eines vernetzten Systems mit vielen Nutzern organisiert sind, ist es wichtig, über die Rechte an seinen Dateien und Verzeichnissen Bescheid zu wissen – vor allem, da *Unix*-Standardrechte der Intuition vieler Leute widersprechen! Es ist oft nötig, Kommandos mit langen Options- und Argumentlisten wiederholt einzutippen. Die hiermit verbundene Tipparbeit wird von Anfängern oft als langatmig und lästig empfunden und ist nicht selten Grund für unnötige Frustrationen. Hauptgrund hierfür ist meist Unkenntnis effektiver Techniken, um lange Zeichenketten mit nur wenigen Tastaturanschlägen einzugeben. Aus diesem Grund sollten die in Kap. 7 vorgestellten Techniken von Anfang an verinnerlicht werden! Sogenannte Sonderzeichen der Shell lernen Sie nach und nach auf Ihrem Weg zu einem erfahrenen *Unix*-Benutzer kennen. Hauptproblem für einen Anfänger ist, dass fehlende Erfahrung und Kenntnis über das Thema

oft zu sehr unverständlichen Fehlern und Problemen führen. Daher führe ich das Thema explizit sehr früh ein und widme den Sonderzeichen auch ein gesondertes Kapitel innerhalb der *Unix*-Kernthemen.

3. Wie jedes Programm bietet auch die *Unix*-Shell individuelle Konfigurationsmöglichkeiten. Hiermit kommt jeder *Unix*-Benutzer recht schnell in Berührung, z. B. um dem System mitzuteilen, wo es von uns installierte Programme finden kann, oder um Kurzformen für lange und immer wiederkehrende Befehlsfolgen zu definieren. In die Grundlagen der Shell-Konfiguration werden Sie in Kap. 10 eingeführt.
4. Wenn ich gefragt werde, welche *Unix*-Befehle man neben den *Kernkommandos* aus Kap. 4–9 unbedingt kennen sollte, so ist das `find`-Kommando ganz oben auf meiner Liste. Es bietet eine extrem große Vielfalt an Möglichkeiten, um ein *Unix*-System nach Dateien und Verzeichnissen zu durchforsten und Aktionen mit ihnen durchzuführen. Dieser Aufgabenkomplex ist offensichtlich für jeden *Unix*-Benutzer sehr wichtig, für viele als essenzieller Bestandteil der täglichen Arbeit. Kap. 11 führt in das Kommando ein.
5. Die restlichen Kap. 12–15 dienen als erster Einstieg in *Unix* als Programmier- und Datenanalyseumgebung. Sie bilden die oben angesprochenen *fortgeschrittenen Unix-Themen* in diesem Buch. *Unix*-Programme arbeiten nach dem sogenannten *Baukastenprinzip*. Ein einzelnes *Unix*-Programm hat typischerweise *nur eine einzige* und sehr begrenzte Aufgabe. Komplexere Aufgabe werden durch *Kombination* und Aneinanderreihung einzelner Programme gelöst. Zwei zentrale Bestandteile zur Kombination einzelner Programme sind sogenannte *Datenpipelines* und *Shell-Skripte*. Mit grundlegenden Kenntnissen über die Pipelines und einigen wenigen *Unix*-Programmen lassen sich bereits recht komplexe Aufgaben der Datenanalyse *sehr effizient* lösen. Sehr häufig werden Sie hier Textdateien als Speichermedium wissenschaftlicher Daten begegnen. Was es unter *Unix* bei Verwendung dieses Dateityps zu beachten gibt, erfahren Sie in Kap. 12. In den Komplex der Datenpipelines und *Unix*-Werkzeuge führe ich Sie in den Kap. 13 und 14 ein. Aus dem Bereich der Shell-Skripte möchte ich Ihnen so viel vermitteln, dass Sie sich wiederholende Analyseaufgaben effizient automatisieren können. Das tun wir in Kap. 15.
6. Das Buch wird durch fünf Anhänge ergänzt. Anhang A beschäftigt sich mit den sogenannten *Regulären Ausdrücken*. Sie sind eines von vielen Hilfsmitteln, um in großen Textdateien (z. B. Datentabellen) Einträge nach bestimmten Kriterien zu filtern. Ich halte sie für *extrem* nützlich, und Grundkenntnisse zu dem Thema sind in sehr vielen wissenschaftlichen Anwendungen von großem Nutzen. Da es sich bereits um ein fortgeschrittenes Spezialthema handelt, behandle ich sie als ergänzendes Material in einem Anhang

   Der zweite Anhang beschäftigt sich mit Datenarchiven unter *Unix*. Vielen von Ihnen ist hier sicher das *Microsoft Windows*-Pendant der `zip`-Dateien geläufig. Sie werden sehr schnell mit *Unix*-Datenarchiven arbeiten müssen, und Anhang B fasst alles Wesentliche zu dem Thema zusammen.

   Wie bereits in Abschn. 1.2 angesprochen, ist Englisch in der *Unix*-Welt die vorherrschende Sprache. Viele Naturwissenschaftler bevorzugen es, auf lange

Sicht mit einer englischen *Unix*-Umgebung zu arbeiten (englische Tastatur, englische *Unix*-Version). Warum dies so ist und was Sie bei der Arbeit in einer deutschen *Unix*-Umgebung beachten sollten, diskutiere ich in Anhang C.

Literaturhinweise für ein weiteres Studium von *Unix* (Anhang D) und die Lösungen zu den Aufgaben (Anhang E) schließen das Buch ab.

2.2 Hinweise zur Bearbeitung des Stoffes

Das Buch ist zum Selbststudium konzipiert. Die Kapitel bauen aufeinander auf, und ich empfehle Ihnen, sie in der gegebenen Reihenfolge zu bearbeiten – am besten direkt vor einem *Unix*-Terminal am Computer. Der Text sollte *komplett* gelesen werden. Ich habe absichtlich auf Referenz-Zusammenfassungen der Befehle und Ähnlichem verzichtet. Hierfür gibt es eine große Zahl sehr guter Bücher und frei verfügbarer Texte im Internet – siehe auch Anhang D. Im Text sind sehr häufig Beispiele eingestreut, die Sie direkt ausprobieren und als Startpunkt eigener Experimente nutzen sollten. Ein sehr wichtiger Teil sind die im Text eingestreuten Übungsaufgaben. Sie dienen einerseits zur Überprüfung Ihres Wissen und andererseits als Anregung, sich intensiver mit dem Stoff zu beschäftigen. Für viele Aufgaben ist es nötig, dass Sie sich, z. B. im Internet, zusätzliche Informationen beschaffen. Alle Aufgaben kommen mit sehr ausführlichen Lösungsvorschlägen, die oft auch neue Aspekte und Details des Stoffes besprechen. Damit Sie unvoreingenommen an jede Aufgabe herangehen, habe ich absichtlich auf die Angabe von Schwierigkeitsgraden verzichtet. Wichtige Dateien, die zur Bearbeitung der Beispiele und der Übungsaufgaben nötig sind, stelle ich Ihnen als Begleitmaterial im Internet zur Verfügung; siehe https://github.com/terben/Einfuehrung_in_Unix. Wie Sie die angebotenen Dateien auf Ihrem *Unix*-Account nutzen können, sehen Sie in Aufgabe 5.3.

Viele Studenten verinnerlichen den Stoff einfacher, wenn ihnen die Themen nicht nur in Schriftform, sondern auch audio-visuell präsentiert werden. Für diese Zielgruppe stelle ich die Themen des Buches auch in Video-Form auf dem Youtube-Kanal „Thomas Erben – Tutorials und Lehrvideos“[1] zur Verfügung. Sowohl das Buch als auch die Video-Tutorials sind in sich komplett, und können sowohl unabhängig voneinander als auch in Kombination zum Studium benutzt werden.

2.3 Textdarstellungen

In diesem Abschnitt sollen die verwendeten Textdarstellungen für alles, was mit *Unix*-Befehlen bzw. deren Ein- und Ausgaben zu tun hat, vorgestellt werden.

- Für alles, was mit *Unix*-Befehlen zu tun hat (Kommandos, Optionen, Argumente), verwenden wir in diesem Buch die `Nicht-Proportionalschrift`

[1] siehe https://www.youtube.com/channel/UCgaFgieXi6HIryaFyhhzQtg.

(Schreibmaschinenschrift). Andere Programm- und Softwarenamen erscheinen *kursiv* (z. B. *Unix* oder *Microsoft Windows*).

- Textabschnitte, die Ein- und Ausgaben auf der Konsole zeigen, werden ebenfalls in Schreibmaschinenschrift wiedergegeben. Zeilen für Benutzereingaben beginnen mit `user$`, was den Benutzer-Prompt im gegenwärtigen Verzeichnis symbolisieren soll. Ausgabezeilen enthalten dagegen kein Prompt-Symbol:

  ```
  user$ ls
  EDV_WS1314_skript.pdf      EDV_WS1314_skript.tex
  user$
  ```

  Um *Unix*-Befehle auszuführen, müssen Eingabezeilen mit `<Return>` abgeschlossen werden. Dies schreiben wir aber nicht explizit aus.
- Beziehen wir uns auf Tastatursonderzeichen, so werden diese innerhalb von spitzen Klammern mit der Schreibmaschinenschrift gesetzt, z. B. `<Return>` (*Enter*-Taste), `<Tab>` (*Tabulator*-Taste), `<Ctrl>` (*Control*-Taste; im Deutschen auch als *Steuerungstaste* bezeichnet), `<Space>` (*Leertaste*).
- Die *Unix*-Shell erlaubt die Eingabe einiger effektiver Kommandos in Verbindung mit der `<Ctrl>`-Taste. Diese sogenannten Control-Kommandos werden durch gleichzeitiges bzw. sukzessives Drücken der `<Ctrl>`-Taste und weiterer Tasten aktiviert. Wir schreiben hierfür z. B. `<C-a>` für „Betätigen der Taste `a`, während die `<Ctrl>`-Taste gedrückt ist." und `<C-x-e>` für „Betätigen der Taste `x` und danach der Taste `e`, während die `<Ctrl>`-Taste gehalten wird."
- In vielen *Unix*-Umgebungen und Programmiersprachen markiert das *Doppelkreuz* „#" einen *Kommentar*. In einer Zeile, in der „#" auftaucht, ist alles, was nach diesem Zeichen kommt, als Kommentar zu verstehen, der von *Unix* bei der Befehlsabarbeitung nicht beachtet wird:

  ```
  user$ ls      # Der Befehl 'ls' listet Dateien auf.
  ```

  Wenn Sie den Effekt eines *Unix*-Befehls nachvollziehen möchten, brauchen Sie also alles, was auf ein „#" folgt, *nicht* mit abzutippen.
- *Unix*-Befehle haben oft *sehr viele* optionale Parameter. Wenn wir klar machen wollen, dass ein Parameter optional ist, so schreiben wir ihn in eckige Klammern, z. B.: `ls [DIR]`. Falls dem Befehl `ls` kein Argument übergeben wird, so listet er das gegenwärtige Arbeitsverzeichnis (im Folgenden als *current working directory* oder *cwd* bezeichnet).
- Manchmal sind Ein- und Ausgabezeilen so lang, dass in diesem Text ein Zeilenumbruch nötig wird. In solchen Fällen symbolisiert ein Backslash (\) am Ende der Zeile, dass sie in der nächsten weitergeht. Zum Beispiel könnten wir die Zeile

  ```
  user$ ./photo.py --in=i.asc --out=o.asc --col=green
  ```

  im Text auch so darstellen:

  ```
  user$ ./photo.py --in=i.asc \
  --out=o.asc \
  --col=green
  ```

Die Bedeutung beider Darstellungen ist aber vollkommen äquivalent. Auch auf der Konsole können Sie lange Eingabezeilen mit dem \ unterteilen. In diesem Fall *maskiert* der \ das nachfolgende `<Return>`, welches ansonsten das bisher eingegebene Kommando ausführen würde.

- In *Unix*-Ausdrücken spielen *Sonderzeichen* wie Klammern oder Leerzeichen eine wesentliche Rolle. Falls irgendwo die exakte Anzahl eingegebener Leerzeichen wichtig ist, werden diese durch das Symbol „␣“ dargestellt. Zum Beispiel: Sie sollten unter *Unix/Linux* von Dateinamen mit Leerzeichen Abstand nehmen und Zeichenketten wie `06-britney␣spears␣␣i␣love␣rocknroll.mp3` als Namen vermeiden!

Erster Kontakt mit *Unix*

3

Übersicht

Zusammenfassung

Unix erlernt sich nur durch aktives Studium direkt am Computer. Das Lesen dieses Buches alleine wird Ihnen nicht helfen, den Stoff zu verinnerlichen und ihn im Bedarfsfall anzuwenden. In diesem Kapitel diskutieren wir einige Möglichkeiten, wie Sie sich eine geeignete *Unix*-Umgebung auf einem eigenen Rechner installieren können.

Ab Kapitel 4 gehe ich davon aus, dass Sie Zugang zu einem Computer mit einem *Unix*/*Linux*-Betriebssystem haben und den Stoff *aktiv* direkt an einem Rechner bearbeiten können. Wahrscheinlich stehen Sie vor einem der drei folgenden Szenarien:

1. Sie haben bereits Zugang zu einem *Unix*-System, z. B. über einen Benutzer-Account an einem *Unix*-Rechnerpool Ihrer Universität. Wenn Sie diesen Zugang für die Arbeit mit diesem Buch nutzen können und wollen, so brauchen Sie im Moment nichts weiter zu tun und können mit Abschn. 3.2 fortfahren.
2. Sie haben einen *Apple Macintosh*-Rechner oder Laptop. *Apple Macintosh*-Computer haben mit *OSX* direkt ein *Unix*-ähnliches Betriebssystem, und auch hier brauchen Sie bezüglich Programminstallation für den Moment nicht viel zu tun. Als Terminal empfehle ich Ihnen, das Programm `iterm` (siehe https://www.iterm2.com/) zu verwenden und eventuell an dieser Stelle zu installieren. Wahrscheinlich müssen Sie im Laufe der Zeit auch das eine oder andere *Unix*-

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_3

Programm nachinstallieren. Das *Fink*-Projekt (siehe http://www.finkproject.org/) ist eine von mehreren Möglichkeiten, um sich komfortabel *Unix*-Programme auf einem *Apple Macintosh* zu installieren und zu verwalten.

3. Sie haben einen Rechner mit einem *Microsoft Windows*-System zur Verfügung. Hier gibt es mehrere Möglichkeiten, sich neben der *Microsoft Windows*-Umgebung ein *Unix* oder *Linux* zu installieren. Hierauf gehe ich im Folgenden näher ein.

3.1 *Unix*/*Linux* auf einem *Microsoft Windows*-Rechner nutzen

Im Folgenden diskutiere ich einige Möglichkeiten, um ein *Unix*/*Linux*-System auf einem *Microsoft Windows*-Rechner aufzusetzen. Es muss klar gesagt werden, dass sich sowohl die verfügbaren *Unix*-Varianten als auch die Möglichkeiten zum Aufsetzen eines solchen Systems sehr schnell weiterentwickeln. Das Folgende wurde im Frühjahr 2016 geschrieben. Es ist gut möglich, dass dieser Abschnitt wenige Jahre später ganz anders geschrieben werden sollte. Zum Beispiel wurde die Möglichkeit, dass ein Hauptbetriebssystem mit Softwarelösungen ein *Gastbetriebssystem* als Programm laufen lässt, erst nach 2010 durch erschwingliche und leistungsstarke Rechner realisiert. Dies ist die unten diskutierte Variante einer *virtuellen Maschine*. Sehr positiv ist, dass es jetzt ohne aufwendige und problematische Rechnerkonfigurationen möglich ist, sich neben einem *Microsoft Windows*-Betriebssystem ein *Linux*-Betriebssystem zu installieren und beide nebeneinander zu betreiben.

Ich führe hier lediglich Alternativen auf, die *längerfristig* eine komfortable Möglichkeit bieten, um mit *Unix* unter *Microsoft Windows* zu arbeiten.

- **Die Installation von *Windows-Cygwin***
  Cygwin (siehe https://www.cygwin.com/) ist kein *echtes Unix*, sondern eine *Microsoft Windows*-Bibliothek, welche das Ausführen von *Unix*-Programmen unter *Microsoft Windows* erlaubt. Nach der Installation können Sie ein Terminalfenster öffnen und darin *Unix*-Programme ausführen. Die Programme selbst werden komfortabel über einen *Paket-Manager* installiert und verwaltet. Das System halte ich für sehr gut und ich nutze es selbst für *Unix*-Aufgaben unter *Microsoft Windows*. Für die Bearbeitung dieses Buches ist es im Prinzip vollkommen ausreichend. Allerdings ist das System gegenüber einem kompletten *Linux* sehr rudimentär und die Bedienbarkeit bei Weitem nicht so komfortabel. Da *Cygwin* gegenüber *Linux*-Distributionen (siehe unten) nur eine sehr untergeordnete Rolle spielt, ist sowohl die Dokumentation als auch die Unterstützung im Internet für letztere *deutlich* besser. Die Erfahrung zeigt, dass *Cygwin* für Anfänger mit seiner sehr einfachen Grundinstallation zunächst attraktiv ist. Oft auftretende Probleme bei der nachfolgenden Integration von Zusatzprogrammen oder das Aufsetzen von Programmen mit grafischer Benutzeroberfläche sind dann aber ohne Hilfe für Anfänger nur schwer zu lösen. Sobald Sie mehr Erfahrung mit *Unix* haben, halte ich *Cygwin* für eine hervorragende Möglichkeit, *Unix*-Werkzeuge in die Arbeit mit *Microsoft Windows* zu integrieren – siehe auch Abschn. 1.5.

- **Die Installation einer *Linux*-Distribution**
  Es ist möglich, mehrere Betriebssysteme auf seinem Rechner zu installieren. Beim Hochfahren des Rechners wählt man dann über einen Boot-Manager aus, welches System benutzt werden soll. Um *Unix* zu nutzen, bietet sich eine sogenannte *Linux-Distribution* als zweites Betriebssystem an.

  Selbst von der *Unix*-Variante *Linux* gibt es inzwischen sehr viele verschiedene Versionen und *Distributionen*. Wie bereits in Abschn. 1.1 erwähnt, ist *Linux* kostenlos und sein Quellcode für jedermann frei zugänglich. Dies erlaubt es, ein *Linux*-System (fast beliebig) zu verändern und neu zusammenzustellen. Dies hat viele verschiedene Distributionen mit unterschiedlichen Vorlieben und Schwerpunkten hervorgebracht. Mögliche Schwerpunkte können sein: ein *Unix*-System, das auf Programmieren ausgelegt ist, oder ein System, das möglichst klein ist, um es auch von einer CD oder einem USB-Stick betreiben zu können. Im wissenschaftlichen Bereich sind die Distributionen *Debian* (siehe http://www.debian.org/) und *Ubuntu* (siehe http://www.ubuntu.com/ und https://ubuntuusers.de/) weitverbreitet. Ich halte *Ubuntu* für eine sehr gute Wahl für *Unix*-Anfänger.

  Viele Distributionen bieten auch Varianten an, die sich direkt von einer CD oder einem USB-Stick booten lassen. Dies hat den großen Vorteil, dass man sich die Systeme erst einmal in Ruhe anschauen und auf seinem Rechner testen kann, bevor man sich zu einer Installation entscheidet.
- **Die Installation einer *Linux*-Distribution innerhalb einer *virtuellen Umgebung***
  Neben der Installation einer *Linux*-Distribution als *Dual-Boot System* kann sie auch als *Gastbetriebssystem direkt unter Microsoft Windows* betrieben werden. Die Installation von Gastbetriebssystemen und die Zuweisung von Systemressourcen geschehen unter einer sogenannten *virtuellen Maschine*. Man installiert also zunächst eine virtuelle Maschine unter seinem *Microsoft Windows* und danach, innerhalb dieser virtuellen Maschine, sein *Linux*. Eine kostenlose Virtualisierungssoftware, mit der ich gute Erfahrungen gemacht habe, ist *VirtualBox* (siehe https://www.virtualbox.org/). Als Anmerkung sei erwähnt, dass man mit einer virtuellen Maschine nicht auf ein Gastbetriebssystem beschränkt ist. Ich betreibe z. B. unter meinem *Apple Macintosh* eine virtuelle Maschine mit dem Betriebssystem *Microsoft Windows* und zwei verschiedenen *Linux*-Distributionen!

  Wenn Ihr Rechner die nötigen Hardwarevoraussetzungen mitbringt, halte ich diese Möglichkeit, sich eine *Unix*-Umgebung auf einem *Microsoft Windows*-Rechner zu installieren, für sehr attraktiv.

Aufgabe 3.1

Schaffen Sie sich an dieser Stelle eine *Unix*/*Linux*-Umgebung, die es Ihnen erlaubt, mit einem *Unix*-Terminal zu arbeiten.

3.2 Erste Terminaleingaben

Das *Unix*-Kommandozeilen-Interface wird heute in sogenannten *Terminals* realisiert, und diese sind das Hauptwerkzeug, das Sie zur Bearbeitung dieses Buches benötigen. Sie geben dort alle Ihre Kommandos ein und erhalten eventuelle Ausgaben Ihrer Programmaufrufe darauf zurück. In diesem Abschnitt wollen wir uns ein wenig mit dem Terminal vertraut machen. Starten Sie an dieser Stelle ein *Unix*-Terminal. Wie das funktioniert, hängt ein wenig von Ihrer aktuellen Umgebung ab. Auf einer *Unix*/*Linux*-Desktop-Umgebung haben Sie wahrscheinlich auf dem Desktop oder in einem Panel ein entsprechendes *Icon*, oder Sie finden es innerhalb eines Startmenüs. Mein *Apple Macintosh*-Terminal sieht nach dem Start wie in Abb. 3.1 aus. Sie haben die Möglichkeit, das Aussehen Ihres Terminals anzupassen. Sie können z. B. die Vorder- und Hintergrundfarbe anpassen oder die Schriftgröße verändern, um die Schrift besser lesen zu können. Entweder hat das Terminal direkt ein Menü oder Sie finden entsprechende Möglichkeiten in den Systemeinstellungen. Da Sie sehr viel mit dem Terminal arbeiten werden, sollten Sie sich damit *wohlfühlen*. Falls Sie noch keine komfortable Möglichkeit haben, um ein Terminal zu öffnen (z. B. ein Icon auf dem Desktop), so sollten Sie sich diese jetzt schaffen. Erstellen Sie z. B. eine Verknüpfung des Terminalprogramms aus dem Startmenü auf den Desktop. Es ist üblich, während einer *Unix*-Sitzung viele Terminalfenster gleichzeitig offen zu haben bzw. immer wieder neue zu öffnen und alte zu schließen. Dies erlaubt es Ihnen, verschiedene Aufgaben in unterschiedlichen

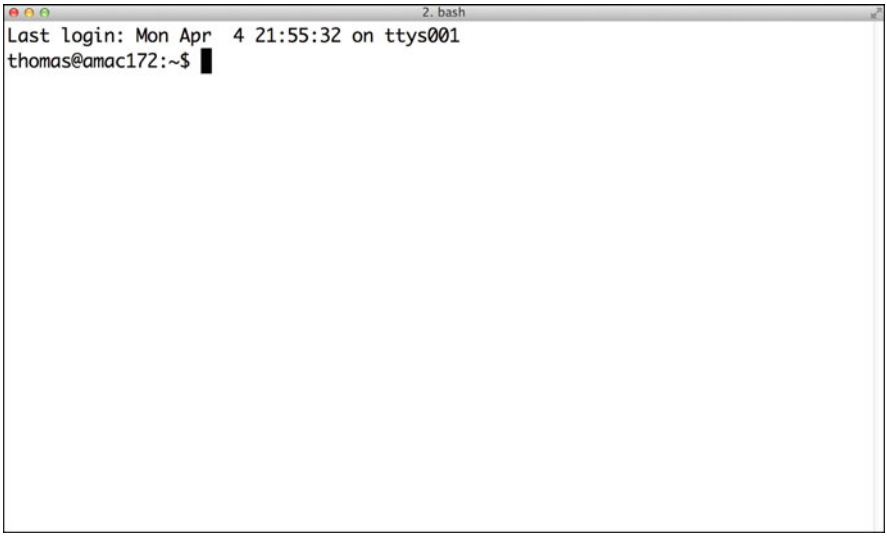

Abb. 3.1 Mein *Apple Macintosh*-Terminal nach dem Start. In der zweiten Zeile sehen wir den Eingabe-Prompt, der auf unsere Befehle wartet

Terminals zu erledigen oder in ein anderes zu wechseln, wenn ein Terminal durch ein laufendes Programm momentan blockiert ist.

Ein neues Terminal zeigt üblicherweise den sogenannten *Eingabe-Prompt* (siehe Abb. 3.1). Er enthält hier Informationen über den Benutzer (thomas) und die Maschine, auf der ich eingeloggt bin (amac172). Die Bedeutung der Tilde (~) lernen Sie in Abschn. 5.2 kennen. Traditionell endet der Prompt mit einem Zeichen wie `$`, `%`, oder `>`. Auch die Informationen, die der Eingabe-Prompt zeigt, lassen sich stark konfigurieren, z. B. lässt sich noch die Uhrzeit hinzufügen usw. Hiermit werden wir uns in Abschn. 10.4 beschäftigen.

Wie *Microsoft Windows* organisiert *Unix* Dateien in einem *Verzeichnisbaum*, was wir uns in Abschn. 5.1 ansehen werden. Hier ist lediglich wichtig zu wissen, dass Sie sich *immer* in einem Verzeichnis befinden, wenn Sie mit einem Terminal arbeiten. In einem neuen Terminal sind Sie üblicherweise in Ihrem *Heimatverzeichnis*, in dem Sie neue Dateien und Verzeichnisse anlegen können. Um zu sehen, welche Dateien und Verzeichnisse sich in Ihrem Heimatverzeichnis befinden, müssen Sie den Befehl `ls` (englische Abkürzung für *list*) verwenden. Eine Befehlseingabe wird durch `<Return>` abgeschlossen, und der eingegebene Befehl wird ausgeführt. Bei mir sieht nach Eingabe dieses Befehls das Terminal wie in Abb. 3.2 aus. Nach der Ausgabe des Inhalts meines Heimatverzeichnisses wartet der Prompt auf den nächsten Befehl.

Wenn Sie ein Terminal auf Ihrem System öffnen und erfolgreich einen `ls`-Befehl eingeben können, haben Sie bereits den allergrößten Teil der notwendigen Installationsarbeiten zur erfolgreichen Bearbeitung dieses Buches hinter sich!

Abb. 3.2 Die Datei- und Verzeichnisliste meines Heimatverzeichnisses nach dem `ls`-Befehl

3.3 Die *Bash*-Shell

Genau genommen ist das Terminal, auf dem Sie gerade Ihre ersten *Unix*-Befehle eingetippt haben, nur die Fensterumgebung und Sie interagieren in Wahrheit mit dem Kommando-Interpreter, der sogenannten *Shell*. Und die Shell ist selbst ein *ganz normales* Programm und nicht eine von vornherein fest integrierte Komponente des Systems. Dies hat, wie fast überall, die wichtige Konsequenz, dass es verschiedene Shells unter *Unix* gibt, die sich in wichtigen und wesentlichen Details voneinander unterscheiden. Denken Sie sich als Analogie die beiden WWW-Browser *Internet Explorer* und *Firefox*. Beide laufen in einer *Microsoft Windows*-Fensterumgebung, leisten im Prinzip dasselbe, unterscheiden sich aber in wichtigen Details.

Alles in diesem Buch bezieht sich auf die sogenannte `bash`-Shell (Bourne Again Shell), welche sich unter *Linux* weitgehend durchgesetzt hat und dort normalerweise als Standard verwendet wird. Falls Sie auf fremden Systemen arbeiten, ist es wichtig zu wissen, welche Shell Sie verwenden. Der Befehl, um das herauszufinden, ist `echo ${SHELL}`. Wenn Sie die `bash` benutzen, ist die Ausgabe des Befehls etwas wie `/bin/bash`:

```
user$ echo ${SHELL}     # gibt die benutzte Shell auf
                        # dem Bildschirm aus
/bin/bash
```

Damit die Befehle und Beispiele so, wie in diesem Buch beschrieben, funktionieren, ist es wichtig, dass Sie mit der *Bash*-Shell arbeiten und eine Ausgabe wie `/bin/bash` bekommen. Falls Sie standardmäßig eine andere Shell benutzen, vor allem wenn Sie eine Shell der *C-Shell-Familie*, also `csh` oder `tcsh`, verwenden sollten, empfehle ich Ihnen, die *Standard-Shell* auf `bash` zu ändern. Konsultieren Sie hierzu gegebenenfalls die Dokumentation Ihres Systems oder das Internet.

3.4 Der Editor `nano`

Wie in Abschn. 1.2 erwähnt, sind unter *Unix* einfache Textdateien von zentraler Bedeutung, und Sie werden von Anfang an viel damit zu tun haben. Zur Erstellung und Bearbeitung dieser Dateien benötigen Sie einen *Editor*.

Exkurs

Editoren unter *Unix*

Wegen der herausragenden Stellung von Textdateien unter *Unix* ist die vorhandene Anzahl an *sehr guten* Editoren extrem hoch. Einige bekannte Vertreter sind: `emacs`, `gedit`, `kate`, `nano`, `vim` (eine Weiterentwicklung des berühmten `vi`). Die ersten Versionen von `vi` und `emacs` sind in den 1970er-Jahren ent-

standen und sicher die am weitesten verbreiteten *Unix*-Editoren. Sie haben beide den großen Vorteil, dass *alle* gängigen, textbasierten *Unix*-Anwendungen direkt und sehr komfortabel von ihnen unterstützt werden, z. B. mit entsprechendem *Syntax-Highlighting*. Fast jeder, der sich länger mit *Unix* beschäftigt, verwendet wegen der schieren Vielfalt an Anwendungsmöglichkeiten irgendwann einen dieser beiden Editoren. Wegen der Unterstützung aller wichtigen Anwendungen kommt man außerdem *für alles* mit einem einzigen Editor aus und man muss sich nicht für jede neue Anwendung noch in einen eigenen Editor einarbeiten. Welcher dieser beiden Editoren letztendlich *besser* ist, ist einer der großen *Glaubenskriege* in der *Unix*-Gemeinde (siehe z. B. http://en.wikipedia.org/wiki/Editor_war).
Obwohl viele neuere Editoren nicht mehr den Funktionsumfang von `emacs` und `vi` haben, bieten sie oft einen schnelleren Einstieg in die grundlegenden und anfangs wichtigen Editierfunktionen. Deshalb sind heute `emacs`/`vi` für Anfänger oft nicht mehr die erste Wahl für einen Editor.

Ich stelle Ihnen mit `nano` im Folgenden einen sehr simpel zu bedienenden Editor vor. Er ist für alle Textmanipulationen, die Sie bei der Bearbeitung dieses Buches benötigen, vollkommen ausreichend. Ich empfehle Ihnen aber, sich bei Gelegenheit Alternativen anzusehen und am Ende den Editor zu verwenden, der Ihnen am besten gefällt.

Der Editor `nano` wird in einem Terminal einfach mit dem gleichnamigen Kommando aufgerufen, z. B.

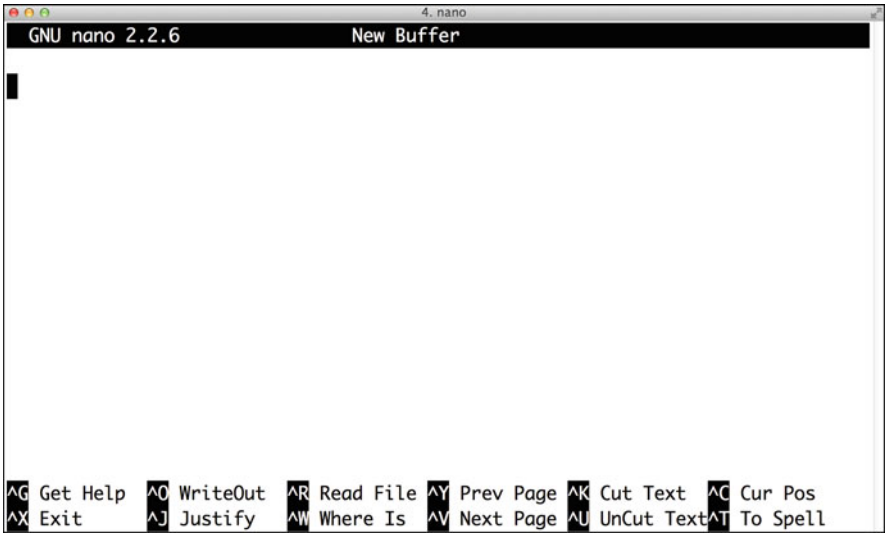

Abb. 3.3 Nach dem Aufruf von `nano` über ein Terminal wechselt dieses zu dem Editor und ein Eingeben weiterer Befehle ist nicht mehr möglich, bis der Editor geschlossen wird

```
user$ nano
     # ruft nano ohne Angabe einer Datei auf
```

oder

```
user$ nano datei.txt
     # ruft nano zum Editieren der Datei
     # 'datei.txt' auf. Diese kann bereits
     # existieren, oder eine neu zu erstellende
     # Datei sein.
```

Nach dem Aufruf des Befehls `nano` (ohne Dateinamen) sieht Ihr Terminal ähnlich zu Abb. 3.3 aus.

Sie können in dem Terminal erst wieder Befehle eingeben, sobald Sie den Editor beendet haben. Öffnen Sie gegebenenfalls einfach ein neues, wenn Sie neben dem Editor weitere Programme aufrufen möchten. Erwartungsgemäß erscheint in `nano` getippter Text direkt in dem Editor, und mit den Cursor-Tasten können Sie sich in bereits geschriebenem Text bewegen. In den unteren Zeilen des Editorfensters sind mögliche Aktionen gezeigt, die durch einfache Tastaturkürzel ausgelöst werden. Hierbei steht „^“ für die `<Ctrl>`-Taste. Mit `<C-x>` können Sie also den Editor beenden usw. Ich denke, das wird Ihnen genügen, um mit dem Editor erste Texte zu erstellen.

Unix-Kommandos

4

Übersicht

Zusammenfassung

Befehle sind der grundlegende Baustein für die Arbeit mit einem Computer. Sie lernen in diesem Kapitel, wie *Unix*-Befehle aufgebaut sind, wie man sie benutzt und wie man sich Informationen über sie verschafft.

4.1 Die Anatomie eines *Unix*-Befehls

In diesem Abschnitt nehmen wir die generelle Syntax eines *Unix*-Befehls genauer unter die Lupe:

```
user$ ls  -l  /usr
 ^     ^   ^     \-> Argument
 |     |   |
 |     |    \-> Option
 |      \
  \       \-> Kommando
   \
    \-> Prompt
```

Notwendiger Bestandteil eines jeden *Unix*-Befehls ist als Erstes das eigentliche *Kommando*, hier `ls` (*list*), welches Dateien und Verzeichnisse auflistet. *Unix*-Kommandos sind üblicherweise Abkürzungen für englische Wörter oder kurze englische Sätze.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_4

Tab. 4.1 Bedeutung gängiger einbuchstabiger *Unix*-Optionen. Die in der zweiten Spalte aufgeführten Befehle werden wir nacheinander in diesem Buch kennenlernen

Einbuchstabige Option	Mögliche Bedeutung mit Beispielbefehlen
`-A`	almost all (`ls`)
`-a`	all (`ls`, `which`)
`-f`	force (`mv`, `cp`); follow (`tail`)
`-i`	interactive (`cp`, `mv`, `rm`)
`-k`	key (`sort`)
`-l`	long format (`ls`)
`-n`	number of lines (`tail`, `head`); numeric (`sort`)
`-r`	reverse (`ls`, sort), recursive (`rm`, `cp`)
`-R`	recursive (`chmod`, `ls`)
`-t`	time (`ls`)
`-v`	verbose (`cp`, `mv`, `rm`), vice-versa (`grep`)

Dem Kommando können (oder müssen) *Optionen* folgen, die das Kommando beeinflussen, hier `-l` (*long listing*). Optionen fangen mit einem „`-`“ oder „`--`“ an. Hierbei leitet der einfache Bindestrich (`-`) traditionell einbuchstabige Optionen ein, und dem `--` folgen komplette englische Worte.

Aufgabe 4.1

Geben Sie nacheinander die Befehle

```
user$ ls /usr
user$ ls -l /usr
user$ ls -r /usr
user$ ls --reverse /usr
user$ ls -l -r /usr
user$ ls -lr /usr
```

ein. Was beobachten Sie? Wofür könnten die Optionen `-l` und `-r` stehen?
Lösungsvorschlag auf Seite 177!

Einbuchstabige Optionen lassen sich gut merken, wenn Sie sich bewusst machen, dass sie für ein englisches Wort stehen, das ihre Funktion beschreibt. Siehe auch Tab. 4.1, welche gängige Optionen von Befehlen listet, die wir in diesem Buch noch kennenlernen werden. Wie Sie in Aufgabe 4.1 gesehen haben, können Optionen kombiniert werden (`ls -l -r`), Sie können mehrere einbuchstabige Optionen unter einem „`-`“ zusammenfassen (`ls -lr`), und viele einbuchstabige Optionen haben auch eine *Langform*, die durch „`--`“ gekennzeichnet ist (`ls -r` und `ls --reverse`).

Neben den Optionen können (oder müssen) einem *Unix*-Kommando noch Argumente folgen. Diese spezifizieren, worauf sich das Kommando bezieht. Im Beispiel

`ls /usr` soll sich `ls` auf das Verzeichnis `/usr` beziehen. Sie haben bereits in Abschn. 3.2 gesehen, dass sich `ls` ohne Argumente auf das Verzeichnis bezieht, in dem Sie sich gerade befinden.

4.2 Hilfe und Informationen zu *Unix*-Kommandos

Um Informationen und Hilfe zu *Unix*-Kommandos und ihren Argumenten zu bekommen, gibt man den Befehl `man` (*manual*) mit dem Kommando, über das man Hilfe benötigt, als Argument:

```
user$ man ls
```

Sobald Sie das `man`-Kommando ausführen, wird ein sogenannter *Pager* (ein simples Textbetrachtungsprogramm) gestartet. Er stellt einfache Funktionen zum Scrollen durch den Text und zur Textsuche zur Verfügung. Nach Aufruf von `man ls` sehen Sie etwa Folgendes:

```
LS(1)                User Commands                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the
       current directory by default). Sort entries
       alphabetically if none of -cftuvSUX nor --sort
       is  specified.

       Mandatory  arguments  to  long  options are
       mandatory for short options too.

       -a, --all
              do not ignore entries starting with .
.
.
.
```

Die *Unix Man-pages* sind standardisiert und enthalten alle dieselben Abschnitte – `NAME` (Programmname), `SYNOPSIS` (Aufrufsyntax), `DESCRIPTION` (Programm- und Optionenbeschreibung) und einige andere, die für Sie anfangs weniger wichtig sind.

Ihr Pager zum Betrachten der *Man*-pages ist höchstwahrscheinlich `less` und seine allerwichtigsten Tastaturkommandos sind:

↑, ↓	Blättern im Text,
`SPACE`	nächste Seite,
`b`	vorherige Seite,
`q`	Pager beenden,
`h`	Anzeigen der Hilfe.

Exkurs

Namen von *Unix*-Kommandos

Wahrscheinlich werden Sie den Pager-Namen `less` etwas merkwürdig finden. Wie weiter oben beschrieben, stehen *Unix*-Programmnamen üblicherweise für Kurzformen englischer Worte, die ihre Funktion beschreiben. Dies ist jedoch nicht immer der Fall. Der früher meistbenutzte Pager hatte den Namen `more` (`more` kann etwa mit *Zeige weiteren Text* innerhalb der gewohnten *Unix*-Namensgebung verstanden werden). Dieser Pager wurde signifikant weiterentwickelt, und weil man einen neuen Namen haben wollte, der sich prominent von `more` absetzt, hat man `less` genommen. Ganz ähnlich gibt es noch einen Pager mit dem Namen `most`. Etliche weitere Programmnamen, die anfangs etwas merkwürdig anmuten, kamen auf ähnliche Weise zustande. Wir werden später z. B. ein Programm namens `cat` (*concatenate*) kennenlernen, welches einfach einen Text aus mehreren Dateien verbindet, und auf dem Bildschirm ausgibt. Als man auch Text *in umgekehrter Reihenfolge* ausgeben wollte, hat man ein Programm namens `tac` geschrieben.

Man-pages sind üblicherweise *komplett*, aber oft *sehr* knapp gehalten. Dies macht es Anfängern oft schwer, sie komplett zu verstehen. Für Sie am wichtigsten sind anfangs sicherlich die Funktionen von Programmoptionen. Diese stehen ganz am Anfang der Seiten, und Sie sollten keine allzu großen Probleme haben, einen Startpunkt für eine Problemlösung zu finden.

Viele *Unix*-Kommandos geben auch mit der Option `--help` Informationen:

```
user$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current
directory by default). Sort entries alphabetically
if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory
for short options too.
  -a, --all                  do not ignore entries
                             starting with .
.
.
.
```

Der Vollständigkeit halber sei hier erwähnt, dass *Unix*-Kommandos, die keine *externen* Programme sind, sondern von der Shell bereitgestellt werden, *keine Man*-pages besitzen. Hilfe für diese Programme bekommen Sie mit dem Kommando `help`:

```
user$ man cd
No manual entry for cd
user$ help cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  ....
```

Das Kommando `cd` (*change directory*) werden Sie in Kap. 5 kennenlernen. Es dient dazu, sein gegenwärtiges Verzeichnis zu wechseln. Sie müssen für den Moment nicht wissen, *welche* Programme Teil der Shell selbst sind. Behalten Sie dies lediglich im Hinterkopf, falls Sie Hilfe für ein *Unix*-Kommando suchen, dafür aber keine *Man*-page finden.

Wie überall hilft Ihnen natürlich auch *Dr. Google* bei allen Problemen mit *Unix*-Kommandos weiter!

Aufgabe 4.2

a) Erstellen Sie mit `nano` eine Textdatei mit dem Inhalt:

```
Martha      Schmitz     100
Bernhard    Bar         2000
Ulf         Klein       500
Kerstin     Meier       3800
Leon        Dinter      24
```

und speichern Sie sie unter dem Namen `namen.txt` ab.

b) Geben Sie den Befehl

```
user$ sort namen.txt
```

Was bewirkt dieser Befehl?

c) Sehen Sie unter der Hilfe von `sort` nach und sortieren Sie die Datei nach den Nachnamen (dem Inhalt der zweiten Spalte).

d) Sortieren Sie den Inhalt von `namen.txt` nach der dritten Spalte. Was passiert hier? Falls das Ergebnis nicht Ihren Erwartungen entspricht, konsultieren Sie erneut die Hilfe von `sort`.

Lösungsvorschlag auf Seite 178!

Aufgabe 4.3

a) Sie wollen Hilfe über das Kommando `man` bekommen und möchten wissen, welche Optionen Sie ihm übergeben können. Wie lautet der entsprechende Befehl hierzu?

b) Sie möchten wissen, welche Befehle es in *Unix* gibt, um Dateien und Verzeichnisse zu komprimieren[1]. Sie würden hierzu gerne wissen, in welchen Dokumentationen (`man`-pages) das Wort *compress* vorkommt. Ist diese Aufgabe mit `man` lösbar bzw. können Sie hierzu eine geeignete Option finden?

Lösungsvorschlag auf Seite 179!

[1] Sie kennen unter *Microsoft Windows* hier die sogenannten *zip*-Dateien.

Das *Unix*-Dateisystem

5

Übersicht

Zusammenfassung

In diesem Kapitel sehen wir uns zunächst genauer an, wie *Unix* Dateien und Verzeichnisse verwaltet. Sie lernen weiterhin, wie Sie sich im *Unix*-Verzeichnisbaum zurechtfinden können und wie Sie Dateien und Verzeichnisse anlegen, kopieren und löschen können.

5.1 Der *Unix*-Verzeichnisbaum

In Abb. 5.1 sind typische Verzeichnisstrukturen von *Microsoft Windows* und *Unix* abgebildet. In beiden Systemen sind Verzeichnisse in einer hierarchischen, baumartigen Struktur realisiert. Ein solcher *umgedrehter* Baum hat eine *Wurzel* und darunter verzweigt sich eine astartige Struktur auf mehreren Ebenen (Verzeichnisse können wieder Verzeichnisse enthalten usw.). Der Hauptunterschied zwischen *Microsoft Windows* und *Unix* ist, dass es unter *Microsoft Windows* mehrere isolierte Bäume gibt. Jedes Laufwerk, wie die verschiedenen Festplatten, ein DVD-Laufwerk, eventuell über USB angeschlossene externe Festplatten oder Memory Sticks, sind hier in einem eigenen Baum untergebracht, dessen *Wurzel* jeweils durch einen Großbuchstaben, beginnend bei `C:`, gekennzeichnet ist.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_5

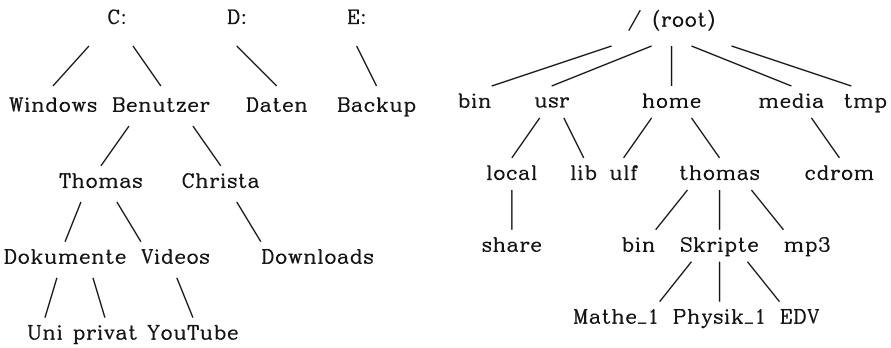

Abb. 5.1 Ausschnitte typischer Verzeichnisstrukturen unter *Microsoft Windows* (links) und *Unix* (rechts). Das *Microsoft Windows*-Verzeichnissystem ist aus mehreren isolierten Bäumen aufgebaut. Jeder Baum ist durch einen Großbuchstaben gekennzeichnet. Dynamische Speichermedien, wie z. B. USB-Sticks, werden hier jeweils mit einem neuen Baum in das System eingebunden. *Unix* verwaltet *alle* Verzeichnisse unter einem einzigen Baum, in den dynamische Elemente bei Bedarf eingebunden und wieder entfernt werden. Dies geschieht z. B. unter dem speziellen Verzeichnis `/media`

Exkurs

***Microsoft Windows*-Laufwerksbuchstaben**

Falls Sie sich wundern, dass die Identifizierung des *Microsoft Windows*-Verzeichnisbaums mit `C:` und nicht mit `A:` beginnt: Die Laufwerksbuchstaben `A:` und `B:` waren früher fest für Diskettenlaufwerke reserviert. Disketten haben eine Kapazität im Megabyte-Bereich, und sie waren in den 1970er- und 1980er-Jahren das vorherrschende externe Speichermedium. Mitte bis Ende der 1990er-Jahre wurden sie durch das Aufkommen erschwinglicher CD-Laufwerke obsolet. Auf der Wikipedia-Seite über „Disketten“[1] finden Sie eine ausführlichere Darstellung dieses Themas.

Unter *Unix* besteht die gesamte Verzeichnisstruktur aus einem einzigen Baum, dessen Wurzel mit / (Slash) gekennzeichnet ist.

Man sagt, dass ein Verzeichnis `B`, das sich innerhalb eines Verzeichnisses `A` befindet, ein *Unterverzeichnis* von `A` ist. Andererseits ist `A` das *Elternverzeichnis* von `B`. Man beachte, dass *jedes* Verzeichnis, außer der Wurzel, *genau ein* Elternverzeichnis besitzt! Ich habe in der Vergangenheit gesehen, dass etliche Probleme, die *Microsoft Windows*-Benutzer mit dem *Unix*-Dateisystem haben, darauf beruhen, dass sie sich der Baumstruktur nicht bewusst sind. Machen Sie sich z. B. klar, dass man in Abb. 5.1 (rechts), um vom Verzeichnis `usr` zum Verzeichnis `tmp` zu gelangen, einen Schritt nach oben zur Wurzel und von dort wieder einen Schritt nach unten machen muss. Es gibt keine *direkte* Verbindung zwischen `usr` und `tmp`!

[1] http://de.wikipedia.org/w/index.php?title=Diskette.

Um auf der *Unix*-Kommandozeile herauszufinden, in welchem Verzeichnis man sich gerade befindet, dient der Befehl `pwd` (*print working directory*)

```
user$ pwd
/home/thomas
```

`/home/thomas` ist mein Heimatverzeichnis. Unter *Unix* trennt der Slash `/` verschiedene Ebenen der Verzeichnisstruktur. Hier ist also `home` ein Unterverzeichnis der Wurzel und `thomas` wiederum ein Unterverzeichnis von `home`. Der gesamte Satz aus `/` und Verzeichnisnamen, der letztendlich zu einem bestimmten Verzeichnis führt, wird als *Pfad* bezeichnet. `/home/thomas` ist also der Pfad zu meinem Heimatverzeichnis.

Hinweise: (1) Unter *Microsoft Windows* wird der Backslash „\“ zum Aufbau von Pfaden benutzt. Mein Heimatverzeichnis dort sieht wie `C:\Benutzer\Thomas` aus; (2) eine häufige Fehlerquelle unter *Unix* ist, dass man sich nicht in dem Verzeichnis befindet, in dem man glaubt zu sein. Speziell, wenn Sie später mit vielen Terminals gleichzeitig arbeiten, verliert man schnell den Überblick, *wo* man gerade ist. Gewöhnen Sie sich am besten an, sich mit `pwd` zu vergewissern, bevor Sie lange und komplexe Programme starten, für die das Verzeichnis essenziell ist! Aus diesem Grund sollte man auch seinen Eingabe-Prompt so konfigurieren, dass er das *cwd* (*currect working directory*) mit anzeigt. Dies werden wir in Kap. 10 besprechen.

5.2 Navigation im *Unix*-Verzeichnisbaum

Um im *Unix* Verzeichnisbaum zu navigieren, bedient man sich des Befehls `cd` (*change directory*):

```
user$ pwd
/home/thomas
user$ cd /tmp
user$ pwd
/tmp
```

Man beachte, dass `cd` keine Bestätigung über den Verzeichniswechsel gibt (ruhiges Betriebssystem; siehe Abschn. 1.2)! Um in das Verzeichnis `/home/thomas/Vorlesung` zu wechseln, kann ich neben dem einfachen

```
user$ cd /home/thomas/Vorlesung
```

auch folgende zwei Schritte ausführen:

```
user$ cd /home/thomas
user$ cd Vorlesung
```

Beginnt das Argument von `cd` nicht mit dem Slash, so bezieht es sich *relativ* auf das Verzeichnis, in dem man sich gerade befindet! Man nennt allgemein Pfade, die *mit* einem Slash beginnen, *absolute Pfade* und die anderen *relative Pfade*.

Es gibt zwei Verzeichnisse, in die jeder Benutzer *sehr* häufig wechseln wird, und deswegen hat `cd` besondere Argumente hierfür. Dies ist einerseits das Heimatverzeichnis:

```
user$ pwd
/tmp
user$ cd ~
user$ pwd
/home/thomas
```

in das man mit dem Tilde (~)-Argument wechseln kann (`cd` *ohne* Argument wechselt ebenfalls in das Heimatverzeichnis). Das zweite ist das Elternverzeichnis des *cwd*:

```
user$ pwd
/home/thomas/Vorlesung
user$ cd ..
user$ pwd
/home/thomas
```

in das man mit `cd ..` gelangt! Man beachte, dass ~ und `..` auch mit *allen* anderen Befehlen, die ein Verzeichnis verlangen, funktioniert, z. B. `ls ..`, `ls ~`.

Aufgabe 5.1

Angenommen, Sie befinden sich mit einem neuen Account auf einer *Unix*-Maschine und sind sich nicht sicher, ob Sie mit einer neu geöffneten Shell in Ihrem Heimatverzeichnis sind. Wie bekommen Sie den *absoluten* Pfad Ihres Heimatverzeichnisses heraus (Angabe einer Befehlsfolge)?
Lösungsvorschlag auf Seite 179!

Aufgabe 5.2

Im Folgenden sei `/home/thomas/` das Heimatverzeichnis des Benutzers `thomas`. Was ist die Ausgabe des jeweils letzten `pwd` Befehls in:

a)
```
user$ pwd
/home/thomas/Vorlesung
user$ cd ../..
user$ pwd
```

b)
```
user$ pwd
/home/thomas/Vorlesung
user$ cd ../../../home/thomas
user$ pwd
```

c)
```
user$ cd ~/Vorlesung
user$ pwd
```

Sind all die `cd`-Befehle sinnvoll? Geben Sie gegebenenfalls kürzere Befehle mit demselben Effekt an.
Lösungsvorschlag auf Seite 179!

5.3 Datei- und Verzeichnismanipulation

Um in *Unix* neue Verzeichnisse anzulegen, dient der Befehl `mkdir` (*make directory*), und um *leere* Verzeichnisse wieder loszuwerden, benutzen wir `rmdir` (*remove directory*).

```
user$ pwd
/home/thomas/
user$ mkdir Test
user$ cd Test
user$ pwd
/home/thomas/Test
user$ cd ..
user$ rmdir Test
user$ cd Test
bash: cd: Test: No such file or directory
```

Wir werden später sehen, wie man Verzeichnisse *mit* Inhalt löschen kann. Eine wichtige Bemerkung ist hier die Namensgebung unter *Unix*. Obwohl es möglich ist, sollten Sie bestimmte Zeichen *nicht* in *Unix*-Datei- und Verzeichnisnamen verwenden. Vermeiden Sie vor allem das unter *Microsoft Windows* populäre Leerzeichen (`<Space>`). Sie wissen bereits, dass das Leerzeichen bei Kommandos zur Trennung von Kommandos, Optionen und Argumenten dient. Sie können sich also leicht vorstellen, dass es bei einem Verzeichnis mit dem Namen `Vorlesung␣1` gewisse Probleme gibt, wenn Sie es mit `rmdir Vorlesung␣1` loswerden wollen. Falls Sie einen Worttrenner in Datei- oder Verzeichnisnamen wollen, so verwenden Sie z. B. den *Unterstrich* „_“. So ist anstatt des Verzeichnisnamens `Vorlesung␣1` die Alternative `Vorlesung_1` eine deutlich bessere Wahl. Sie wissen bereits, dass der Bindestrich „`-`“ eine Option signalisiert, und daher sind Datei- und Verzeichnisnamen, die mit „`-`“ anfangen, auch eine schlechte Idee. Es gibt noch eine ganze Reihe anderer Zeichen, die unter *Unix* eine spezielle Bedeutung haben und daher in Datei- und Verzeichnisnamen nicht auftauchen sollten. Mit diesem Thema werden wir uns später, vor allem in Kap. 8, noch genauer beschäftigen. Wie in Abschn. 1.2 erwähnt, sollten Sie unter *Unix* auch von deutschen Umlauten in Dateinamen Abstand nehmen.

Die letzten verbleibenden, wichtigen Operationen mit dem *Unix*-Dateisystem sind das Kopieren, Umbenennen und Entfernen von Dateien und Verzeichnissen. Hierzu stehen die Befehle `cp` (*copy*), `mv` (*move*) und `rm` (*remove*) zur Verfügung. Alle Kommandos akzeptieren Dateien und/oder Verzeichnisse als Argumente und die Bedeutung sollte jeweils intuitiv klar sein. Ich gebe unten etliche Beispiele, aus denen die Benutzung der Kommandos hervorgeht. Die meisten Schwierigkeiten mit diesen Befehlen beruhen auf Verständnisproblemen mit dem *Unix*-Verzeichnisbaum (siehe Abschn. 5.1).

```
user$ cp datei.txt datei.txt.copy
        # Anlegen einer Kopie von datei.txt.
        # Die Kopie heißt datei.txt.copy und
        # ist im cwd.

user$ mv datei.txt.copy test.txt
        # benennt datei.txt.copy in test.txt um

user$ cp test.txt ..
        # kopiert test.txt ins Elternverzeichnis

user$ cp test.txt /home/thomas/Vorlesung
        # klar, oder?

user$ mv test_1.txt ..
        # verschiebt test_1.txt ins Elternverzeichnis

user$ mv ../test.txt /home/thomas/Vorlesung
        # Fragen?

user$ mkdir Uebung
user$ mv Uebung Uebung_1
        # benennt Verz. Uebung in Uebung_1 um
        # ODER verschiebt Verz. Uebung in ein
        # vorhandenes Verz. Uebung_1 (siehe auch
        # unten)

user$ mv Uebung_1 ~
        # sollte auch klar sein; aller Inhalt
        # in Uebung_1 wird mitverschoben

user$ rm /home/thomas/Vorlesung/test.txt
        # entfernt Datei test.txt aus dem Verz.
        # /home/thomas/Vorlesung

user$ rm -r ../Uebung_1
        # entfernt Verzeichnis ../Uebung_1
        # samt allen Inhalts! (VORSICHT!)
```

Man wird sehr häufig in das *cwd* kopieren wollen. Ähnlich zu der Abkürzung „. .“, die für das Elternverzeichnis steht, gibt es eine Abkürzung für das *cwd*, nämlich den einfachen Punkt „.“.

```
user$ cp /home/thomas/datei.txt .
        # Kopiert datei.txt aus /home/thomas ins cwd
```

Der Befehl

```
user$ cp test.txt Test
```

(Ähnliches gilt für `mv`) kopiert entweder die Datei `test.txt` in die Datei `Test`, *oder* er kopiert die Datei in ein *vorhandenes* Unterverzeichnis `Test`. Dort hat die kopierte Datei dann wieder den Namen `test.txt`! Möchte man für *Unix betonen*, dass man in ein Verzeichnis `Test` kopieren möchte, so kann man den Befehl

```
user$ cp test.txt Test/
```

geben. Das abschließende „/“ sagt *Unix*, dass man wirklich ein Verzeichnis meint. In diesem Fall würde es eine Fehlermeldung geben, falls das Verzeichnis `Test` nicht existiert!

Wichtig

***Unix* fragt in der Regel *nicht* nach!**

Die Datenmanipulationsbefehle `cp`, `mv` und `rm` überschreiben existierende Dateien oder löschen diese, wenn man sie dazu auffordert. Ein Nachfragen findet *in der Regel nicht statt*. Man kann ein Nachfragen bei diesen Kommandos erzwingen, indem man die Befehle mit der Option `-i` oder `--interactive` aufruft. Besondere Vorsicht ist bei dem Befehl `rm` in Verbindung mit der Option `-r` (*recursive*) geboten, wenn man ihn auf Verzeichnisse anwendet. Er löscht das Verzeichnis samt allen Inhalts, also auch eventuell vorhandene Unterverzeichnisse!

Exkurs

Kompatibilität von *Apple Macintosh-*, *Unix-* und *Microsoft Windows*-Dateisystemen

Es ist nicht selbstverständlich, dass Dateisysteme verschiedener Betriebssysteme miteinander kompatibel sind! Damit kommen Sie unter Umständen in Berührung, wenn Sie externe Speichermedien (hauptsächlich USB-Sticks und externe USB-Festplatten) für den Transfer von Daten zwischen verschiedenen Betriebssystemen nutzen. USB-Sticks und externe Festplatten sind typischerweise mit dem sogenannten *FAT32*-Filesystem vorformatiert. FAT32 ist ein altes *Microsoft*-Filesystem, das auf *Microsoft Windows*-Varianten bis ungefähr in das Jahr 2000 Standard war. Es ist heute der *kleinste gemeinsame Nenner*, der eigentlich von allen Betriebssystemen und Endgeräten (z. B. Fernseher oder DVD-Player mit USB-Eingang) *gelesen und beschrieben* werden kann. Falls Sie ein externes Speichermedium, das zum Datenaustauch dient, einmal neu formatieren müssen, so tun Sie dies am besten mit dem FAT32-Dateisystem. Bei anderen Dateisystemen gibt es eventuell lästige Probleme. Zum Beispiel kann ich Daten eines USB-Sticks, der mit dem modernen *Microsoft Windows*-System *NTFS (New Technology File System)* formatiert ist, auf meinem *Apple Macintosh* lesen. Beschreiben lässt sich dieser Stick dort allerdings nicht.

Das FAT32-System hat gegenüber moderneren Dateisystemen einige Nachteile, die aber bei *typischen* Datenübertragungen nicht ins Gewicht fallen. Eine Liste gängiger Dateisysteme und weitergehende Informationen zu dem Thema finden Sie z. B. auf Wikipedia: https://de.wikipedia.org/wiki/Liste_von_Dateisystemen.

Aufgabe 5.3

In späteren Kapiteln werden Sie Dateien benötigen, die ich als Begeleitmaterial im Internet unter https://github.com/terben/Einfuehrung_in_Unix zur Verfügung stelle. Am einfachsten erhalten Sie das gesamte Datenpaket, indem Sie sich das `zip`-Archiv `Einfuehrung_in_Unix-master.zip` über den `Download ZIP`-Button von der Seite herunterladen. Falls Ihnen `zip`-Archive von *Microsoft Windows* her noch nicht geläufig sind, so können Sie sich z. B. auf Wikipedia darüber informieren; siehe https://de.wikipedia.org/wiki/ZIP-Dateiformat. Um unter *Unix* mit `zip`-Dateien zu arbeiten, stehen die Befehle `zip` (Datei zippen/packen) und `unzip` (Datei entzippen/auspacken) zur Verfügung.

Laden Sie sich die Datei `Einfuehrung_in_Unix-master.zip` herunter und entpacken Sie sie mit `unzip`. In dem entpackten Verzeichnisbaum finden Sie für jedes Kapitel, zu dem es Dateien gibt, ein eigenes Unterverzeichnis. Die Dateien `README.md` in den verschiedenen Verzeichnissen enthalten weitere Informationen zu den Dateien.

Hinweise: (1) Wenn Sie mit Ihrem Web-Browser unter *Unix* Dateien herunterladen, so werden diese *wahrscheinlich* in einem der Verzeichnisse `~/Downloads` oder `~/Desktop` abgelegt. Im zweiten Fall erscheint auch ein entsprechendes Icon für die Datei auf Ihrem Desktop. Falls Sie die Datei in keinem der beiden Ordner finden, müssen Sie die Einstellungen Ihres Browsers zurate ziehen; (2) das `zip`-Format ist unter *Microsoft Windows* eines der vorherrschenden Komprimierungsverfahren. Unter *Unix* wird es nur selten benutzt, und dort sind die sogenannten `tar`-Archive am geläufigsten. Anhang B gibt Ihnen hierzu alle wesentlichen Informationen.

Aufgabe 5.4

Sie wollen eine Datei `test.txt` von `~/uebung_01` nach `~` kopieren. Sie befinden sich in `~/uebung_01`. Geben Sie zwei `cp`-Befehle an, die dies erledigen; einmal soll das Ziel als *relativer* und einmal als *absoluter* Pfad angegeben werden.

Lösungsvorschlag auf Seite 181!

Aufgabe 5.5

Sie haben ein Verzeichnis `~/uebung` und Sie befinden sich in `~`, wo Dateien mit Namen `aufgabe_1.txt` und `aufgabe_2.txt` liegen. Was geschieht bei folgenden Befehlen?

a) `cp ~/aufgabe_1.txt ./uebung`

b) `cp ~/aufgabe_1.txt ./uebung/`
c) `cp ~/aufgabe_1.txt ~/aufgabe_2.txt ./uebung`
d) `cp ~/aufgabe_1.txt ~/aufgabe_2.txt ./uebung/`

Was geschieht bei obigen vier Befehlen, falls das Verzeichnis `~/uebung` *nicht* existiert? Was geschieht bei obigen Befehlen, falls wir uns bei ihrer Ausführung irrtümlicherweise in `~/uebung` befinden?
Lösungsvorschlag auf Seite 181!

Aufgabe 5.6

a) Üben Sie die *Unix*-Kommandos `mkdir` und `cd`, indem Sie die Verzeichnisstruktur

```
~/aufgabe/ordner_1/ordner_3
     ...           /ordner_4
     ...  /ordner_2/ordner_5
     ...           /ordner_6
```

erzeugen. Überprüfen Sie ihre Struktur mit dem `ls`-Befehl!
b) Sie haben nach voriger Aufgabe wahrscheinlich das Gefühl, dass man diese Art von Aufgaben effektiver lösen können sollte als mit einer langen und fehleranfällige Kette von `cd`/`mkdir`-Befehlen – vor allem wenn die zu erstellende Verzeichnishierarchie noch komplexer werden sollte. Können Sie eine solche, effektivere Methode finden?
c) Ordnen Sie mit geeigneten `mv`-Befehlen Ihre Struktur in

```
~/aufgabe/ordner_1/ordner_2/ordner_3
       ...                  /ordner_4
       ...                  /ordner_5
       ...                  /ordner_6
```

um. Entfernen Sie danach mit dem `rmdir`-Befehl die Verzeichnisse `ordner_5` und `ordner_6`.
d) Mit dem Befehl `touch` können Sie eine leere Datei erzeugen. Erzeugen Sie eine solche namens `test_1.txt` im Verzeichnis `ordner_4`; gehen Sie also zunächst in `ordner_4` und geben Sie dort `touch test_1.txt` ein. Kopieren Sie diese Datei danach in die Ordner `ordner_3` und `ordner_2`. Überprüfen Sie mit dem `ls`-Befehl, ob sich die Datei auch tatsächlich in allen drei Ordnern befindet.
Lösungsvorschlag auf Seite 181!

Aufgabe 5.7

Jemand möchte wissen, warum es eine schlechte Idee ist, Dateien mit gewissen Sonderzeichen zu benutzen. Er gibt folgende Befehle ein:

```
user$ mkdir ~/test
user$ cd ~/test
user$ touch "Ein␣Lied.mp3"
```

Führen Sie diese Befehle aus und überprüfen Sie mit `ls`, welche Datei erzeugt wird. Löschen Sie die Datei wieder mit `rm`. Eventuell müssen Sie die *Unix*-Hilfe oder das Internet zurate ziehen. Versuchen Sie danach mit dem `touch`-Befehl eine Datei mit dem Namen `$PATH` zu erzeugen, diese in ein anderes Verzeichnis zu verschieben und anschließend wieder zu löschen.
Lösungsvorschlag auf Seite 182!

5.4 Versteckte Dateien und Verzeichnisse

Bis jetzt sind wir davon ausgegangen, dass uns ein `ls`-Befehl *alle* existierenden Dateien und Verzeichnisse anzeigt. Dem muss nicht so sein. Es gibt unter *Unix* sogenannte *versteckte* Dateien und Verzeichnisse. Deren Namen beginnen mit einem Punkt „`.`“:

```
user$ ls
user$ touch test.txt
user$ ls
test.txt
user$ touch .test.txt
user$ ls
test.txt
user$ ls -a
.  ..  .test.txt test.txt
```

Der Befehl `touch` erzeugt eine neue, leere Datei mit dem Namen seines Arguments, falls eine solche noch nicht existiert. Der zweite `touch`-Befehl erzeugt eine Datei mit Namen `.test.txt`, welche ein einfaches `ls` *nicht* anzeigt. Dies geschieht erst mit der Option `-a` (*all*). Hierbei erscheinen auch die *versteckten* Verzeichnisse „`.`“ (steht für das cwd) und „`..`“ (steht für das Elternverzeichnis). Wir können auch reguläre versteckte Verzeichnisse anlegen und nutzen:

```
user$ pwd
/home/thomas/test
user$ mkdir .versteckt
user$ ls
user$ ls -a
. .. .versteckt
user$ cd .versteckt
user$ pwd
/home/thomas/test/.versteckt
```

Sie haben möglicherweise sehr viele versteckte Dateien und Verzeichnisse in Ihrem Heimatverzeichnis. Diese werden vom System hauptsächlich für Programm- oder Systemkonfigurationen verwendet, um die sich ein Benutzer im Normfall nicht kümmern muss, und oft auch nicht kümmern *soll*:

```
user$ pwd
/home/thomas
user$ ls
Desktop    Downloads   mail    Pictures  Python
Documents  edv_ws1516  Music   Public    Templates
user$ ls -a
.               edv_ws1516       .ssh
..              .emacs           Templates
.addressbook    .emacs.d         .mozilla
.adobe          .esd_auth        mail
.bash_history   .fontconfig      Music
.bash_profile   .gconf
.
.
.
```

Wichtig aus Benutzersicht sind die *Shell-Konfigurationsdateien* `.bash_profile` und `.bashrc`. Diese erlauben es uns, die Shell eigenen Bedürfnissen anzupassen, und es ist daher für Benutzer durchaus interessant, diese Dateien zu bearbeiten. Wundern Sie sich momentan bitte nicht, falls Sie diese Dateien in Ihrem Account *nicht* vorfinden sollten. Das Thema *Shell-Konfiguration* wird in Kap. 10 genauer behandelt.

Hinweis: Versteckte Dateien und Verzeichnisse gibt es in ähnlicher Form auch unter *Microsoft Windows*. Erkundigen Sie sich bei Gelegenheit im Internet darüber.

5.5 Wildcards

Oft möchte man viele Dateien auf einmal ansprechen, z. B. wenn Sie von all Ihren Textdateien in einem Verzeichnis eine Sicherheitskopie anlegen möchten. Mit Ihrem bisherigen Wissen könnten Sie dies folgendermaßen lösen:

```
user$ ls
datei_1.txt  datei_2.txt   datei_3.txt
user$ mkdir ~/copy
user$ cp datei_1.txt datei_2.txt datei_3.txt ~/copy
```

Das Ganze wird doch recht viel Tipparbeit, wenn es noch mehr Dateien werden, die man kopieren möchte. *Unix* bietet hierfür sogenannte *Jokerzeichen* oder *Wildcards* an, um viele Dateien mit ähnlichem Namensmuster gleichzeitig anzusprechen. Das wichtigste dieser Joker ist der „`*`“. Er ist ein Ersatz für *kein*, *ein* oder *beliebig viele* beliebige Zeichen:

```
user$ ls
datei_1.txt datei_2.txt datei_3.txt
user$ mkdir ~/copy
user$ cp *.txt ~/copy
        # kopiert alle Dateien, die auf '.txt' enden.
```

Jokerzeichen können *überall* verwendet werden, wo Dateinamen auftauchen. Ein paar weitere Beispiele zur Verdeutlichung und zur Anwendbarkeit:

```
user$ ls *ea*
        # liste alles, was irgendwo im Namen die
        # Buchstabenkombination 'ea' enthält.

user$ ls ea*
        # liste alles, was mit der Buchstaben-
        # kombination 'ea' anfängt.

user$ ls *ea
        # liste alles, was mit der Buchstaben-
        # kombination 'ea' aufhört.

user$ cp * ~
        # kopiere 'alle' Dateien aus dem cwd ins
        # Heimatverzeichnis.

user$ rm *
        # VORSICHT: löscht ALLE Dateien im cwd!
```

Gibt es keine Datei, auf die ein Jokerzeichen passt, so passiert nichts, das Wildcard-Muster bleibt stehen und Sie erhalten eine Fehlermeldung:

```
user$ ls *.txt
ls: cannot access *.txt: No such file or directory
```

Als weiteres Wildcard steht das „?“ zur Verfügung. Es ersetzt *genau ein beliebiges* Zeichen:

```
user$ ls
datei_1.txt datei_2.txt datei_10.txt
user$ mkdir ~/copy
user$ cp datei_?.txt ~/copy
        # kopiert die Dateien datei_1.txt
        # und datei_2.txt

user$ cp datei_??.txt ~/copy
        # kopiert die Datei datei_10.txt
```

Das letzte Wildcard ist ein zu `?` verwandtes Konstrukt, welches ebenfalls für *genau ein*, aber nur vom Benutzer definiertes, Zeichen steht. Die Wildcard besteht aus

Tab. 5.1 Eckiges Klammerpaar als Wildcard

Wildcard-Beispiel	Bedeutung
`[abc]`	*Eines* der angegebenen Zeichen „a“, „b“ oder „c“ ist erlaubt
`[a-k]`	Ein Kleinbuchstabe aus dem angegebenen Bereich des Alphabets a–k ist erlaubt
`[1-5]`	Ein Zeichen aus dem angegebenem Ziffernbereich 1–5 ist erlaubt
`[!a-c]`	*Keines* der Zeichen „a“, „b“ oder „c“ darf vorkommen
`[^a-c]`	Genau dasselbe wie `[!a-c]`
`[A-Cg-i1-579]`	Eine Kombination erlaubter Zeichenbereiche und Zeichen: Erlaubt sind die Großbuchstaben A–C, die Kleinbuchstaben g–i, der Ziffernbereich 1–5 und die Ziffern 7 und 9

einem eckigen Klammerpaar, in das die erlaubten Zeichen eingeschlossen sind. Die Funktionsweise ist aus Tab. 5.1 und den folgenden Beispielen ersichtlich.

```
user$ ls
datei_1.txt datei_2.txt datei_10.txt datei_11.txt
datei_12.txt datei_a.txt datei_b.txt datei_c.txt
datei_a1.txt datei_a2.txt datei_a3.txt

user$ ls datei_[abc].txt
datei_a.txt datei_b.txt datei_c.txt

user$ ls datei_[1-9].txt
datei_1.txt datei_2.txt

user$ ls datei_[!1a].txt
datei_2.txt datei_b.txt datei_c.txt

user$ ls datei_[1-9][0-1].txt
datei_10.txt datei_11.txt
        # Jedes Klammerpaar steht für "genau ein"
        # Zeichen!

user$ ls datei[a-b1-2].txt
datei_a.txt datei_b.txt datei_1.txt datei_2.txt
```

Beachten Sie, dass Wildcards versteckte Dateien (siehe Abschn. 5.4) *nicht* ansprechen, wenn dies nicht explizit verlangt wird:

```
user$ ls
user$ touch test.txt .test.txt
user$ ls -a
.  ..  .test.txt  test.txt
user$ rm *      # keine explizite Einbindung
                # versteckter Dateien bei diesem 'rm'
user$ ls -a
```

```
.  ..  .test.txt
user$ rm .test* # Hier werden die versteckten Dateien
                # 'explizit' angesprochen
user$ ls -a
.  ..
```

Aufgabe 5.8

Sie haben ein Verzeichnis mit 1001 Dateien `datei_000.txt, datei_001.txt, ..., datei_1000.txt`. Welche Dateien werden nach den folgenden `ls`-Befehlen angezeigt?

a) `ls datei_[12][34].txt`
b) `ls datei_[12]?.txt`
c) `ls datei_[12]*.txt`
d) `ls datei_*10.txt`

Lösungsvorschlag auf Seite 183!

Aufgabe 5.9

Sie haben ein Verzeichnis mit 1001 Dateien `datei_000.txt, datei_001.txt, ..., datei_1000.txt`. Geben Sie jeweils *einen* `ls`-Befehl an, der die folgenden Dateien anzeigt:

a) Alle Dateien, die die Ziffernfolge `01` enthalten,
b) die Dateien `datei_000.txt - datei_999.txt`,
c) die Dateien `datei_000.txt - datei_005.txt` und die Datei `datei_009.txt`,
d) die Dateien `datei_010.txt - datei_039.txt`,
e) Die Dateien `datei_000.txt` und `datei_1000.txt`.

Lösungsvorschlag auf Seite 183!

Aufgabe 5.10

Das Programm `echo` gibt einfach sein Argument wieder auf dem Bildschirm aus:

```
user$ echo Thomas
Thomas
user$ echo /home/thomas
/home/thomas
```

Führen Sie den Befehl

```
user$ echo *
```

in einem Verzeichnis aus, in dem sich Dateien befinden. Erklären Sie, was hier passiert. Befragen Sie gegebenenfalls das Internet über die *Maskierung von Unix Sonderzeichen*. Können Sie einen Weg finden, um mit dem Kommando `echo`, ausgeführt in einem Verzeichnis, in dem sich Dateien befinden, einen `*` zu bekommen, also:

```
user$ echo ..... # ..... ist das was Sie finden
                 # müssen
*
user$
```

Lösungsvorschlag auf Seite 184!

Das *Unix*-Rechtesystem

6

Übersicht

Zusammenfassung

Sie lernen, welche Rechte *Unix*-Dateien und -Verzeichnisse haben können, welche *Benutzerklassen* es gibt und wie Sie Rechte eigener Dateien und Verzeichnisse modifizieren können.

6.1 Die Philosophie des *Unix*-Rechtesystems

Heutige Datei- und Verzeichnisrechtesysteme haben als oberste Priorität den Schutz der Daten einzelner Nutzer. Falls Sie bereits mit vernetzten Systemen zu tun hatten, sind Sie es sicher gewohnt, dass nur Sie standardmäßig Zugriff auf Ihre Daten haben. Wenn Sie Daten mit anderen Nutzern teilen möchten, so müssen Sie einzelnen Personen *explizit* Rechte gewähren. Unter *Unix* verfolgt man aus historischen Gründen den entgegengesetzten Ansatz. Wissenschaftliche Großrechner dienen oft der *Zusammenarbeit* an großen Datenmengen. Hier ist es vernünftig, wenn Benutzer Daten von Anfang an *teilen* und zunächst alle Mitarbeiter eines Projekts mindestens *Leserechte* an Daten besitzen. Wenn Sie einen neuen Account an einem *Unix*-Rechner bekommen, ist es nicht unüblich, dass alle anderen Nutzer des Systems fast alle Ihre Dateien standardmäßig lesen können! Werden Daten als *vertraulich* eingestuft, so müssen Rechte an ihnen explizit *entzogen* werden! Das *Unix*-Standardrechtesystem spiegelt diese *kollaborative* Grundeinstellung von *Unix* wider. Es bietet einfache Möglichkeiten, um Rechte zu gewähren oder zu entziehen. Es erlaubt allerdings nicht den Aufbau komplexerer Rechteverteilungen. So kann

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_6

man zwar einfach Daten als *generell* vertraulich deklarieren und *allen* Benutzern Rechte an seinen Dateien entziehen. Es ist dann aber leider nicht ohne Weiteres möglich, *nur ganz bestimmten* Benutzern wieder Rechte zu gewähren! Ähnlich hat man Probleme, *nur einzelnen Nutzern* Rechte zu entziehen! Aus heutiger Sicht besitzt das System etliche Anomalien und Unzulänglichkeiten; siehe z. B. Aufgabe 6.3. Ein Problem ist auch, dass Computernutzer heute davon ausgehen, dass ihre Daten auf vernetzten Systemen zunächst *nur von ihnen selbst* genutzt werden können! Es gibt leider sehr viele *Unix*-Nutzer, die von dem Rechtesystem und seinen Konsequenzen nur unzureichende Kenntnisse besitzen! Aus diesen Gründen wird das *Unix*-Standardsystem heute den Bedürfnissen vieler Institutionen nicht mehr gerecht, und es ist möglich, dass auf Ihrem System ein anderes benutzt wird. In diesem Fall müssen Sie auf die Dokumentation zurückgreifen, um etwas über Ihr Rechtesystem zu erfahren.

6.2 Rechte und Benutzerklassen

Alle relevanten Informationen über bestehende Rechte an Dateien und Verzeichnissen werden von `ls` mit der Option `-l` (*long listing*) angezeigt:

```
user$ ls -l vorl_02.tex
-rw-r--r-- 1 thomas users ...  vorl_02.tex
    ^          ^       ^
    |          |       \
    |          \        - Gruppe
    \           - Besitzer
     - Rechtevektor
```

Die dritte Spalte der Ausgabe zeigt den *Besitzer* (hier `thomas`) der Datei und die vierte Spalte die *Gruppe* (hier `users`), zu der dieser Benutzer gehört. Jeder Benutzer gehört zu einer Gruppe, die ihm bei Account-Erstellung von `root` zugewiesen wird, und Sie können daran nichts ändern. Mögliche Gruppen sind z. B. Studenten, Mitarbeiter, Professoren, Administratoren usw. Wichtig für Sie zu wissen ist, dass man *Unix*-Dateirechte für den Besitzer (das sind Sie selbst), die Gruppe und *alle anderen* getrennt vergeben kann. *Alle anderen* sind hier alle Benutzer, die weder *Sie selbst* sind, noch die zu Ihrer Gruppe gehören! Die effektiven Rechte, die ein Benutzer letztendlich an einer Datei hat, sind in der eben genannten Reihenfolge der Benutzerklassen gegeben. So haben beispielsweise Sie als Eigentümer einer Datei all die Rechte, die Ihnen in dieser Position gegeben sind, ungeachtet der Rechte, die Sie als Gruppenmitglied hätten.

Die erste Spalte der Ausgabe von `ls -l` (im Folgenden bezeichnen wir diese Spalte auch als *Rechtevektor*) gibt Auskunft darüber, welche Rechte jede der drei Benutzerklassen an einer Datei oder einem Verzeichnis hat:

```
user$ ls -l
-rwxr-xr-x 1 thomas users ...  program.sh
-rw-r--r-- 1 thomas users ...  vorl_02.tex
drwxr-xr-x 2 thomas users ...  beispiele/
|\^/\^/\^/
| |  |  |
| \  \  - Rechte für alle anderen
\  \  - Rechte für die Gruppe
 \  - Rechte für den Besitzer
  - Dateityp (- = Datei, d = Verzeichnis, l=Link)
```

Der Rechtevektor enthält insgesamt zehn Stellen. Die erste Stelle gibt an, ob es sich um eine Datei (`-`) oder ein Verzeichnis (`d`) handelt. Unter Umständen sehen Sie hier auch ein `l`, das einen sogenannten *Link* signalisiert. Diese Links sind ähnlich zu *Dateiverknüpfungen* unter *Microsoft Windows*. Auch die zweite Spalte in der `ls -l`-Ausgabe (eine Zahl größer oder gleich eins) hängt mit Links und Dateiverknüpfungen zusammen. Auf weitere Details zu diesem Themenkomplex möchte ich hier aber nicht eingehen. Die restlichen neun Spalten des Rechtevektors sind als Dreierblöcke zu lesen, wobei der erste für den Besitzer, der zweite für die Gruppe und der letzte für *alle anderen* steht. In jedem Dreierblock können drei verschiedene Rechte (`r = read`, `w = write` und `x = execute`) gesetzt oder entzogen (markiert durch ein `-`) sein. Hierbei haben die drei Rechte für Dateien und Verzeichnisse jeweils etwas unterschiedliche Bedeutungen, die in Tab. 6.1 aufgelistet sind. Wenn Sie mit den Rechtebedeutungen und den Beziehungen zwischen Dateien und Verzeichnissen Probleme haben, stellen Sie sich ein Verzeichnis als einen Raum vor und Dateien als Gegenstände in diesem Raum. Die Rechte an dem Raum (am Verzeichnis) spezifizieren dann, ob der Rechteinhaber sich eine Inhaltsliste des Raums verschaffen darf (`r`), ob er Gegenstände aus dem Raum herausnehmen oder hinzufügen darf (`w`) oder ob er ihn *betreten* und auf seinen Inhalt zugreifen darf (`x`). Die Rechte an einem Gegenstand besagen, ob er sich diesen *genau* ansehen (die Datei kopieren oder lesen) darf (`r`), ob er einen Gegenstand sogar verändern kann (`w`) und ob er einen Gegenstand *benutzen* (ein Programm ausführen) darf (`x`).

Tab. 6.1 *Unix*-Rechte und ihre Bedeutungen für Dateien und Verzeichnisse

Recht	Datei	Verzeichnis
`r`	Dateiinhalt lesbar (`cat`, `less`)	Verzeichnis lesbar (`ls`)
`w`	Dateiinhalt veränderbar	Verzeichnisinhalt änderbar, d. h. Dateien und Verzeichnisse können darin angelegt, gelöscht, umbenannt werden etc. (`rm`, `mv`, `cp`, `mkdir`, `rmdir`)
`x`	Datei als Programm ausführbar	Wechsel in Verzeichnis erlaubt (`cd`) und Zugriff auf Dateien innerhalb des Verzeichnisses möglich (Kopie von Dateien aus dem Verzeichnis mit `cp` oder Ausführung von Programmen innerhalb des Verzeichnisses)

6.3 Modifikation von Rechten an eigenen Dateien und Verzeichnissen

Um Rechte an Dateien zu setzen und zu verändern, dient das Kommando `chmod` (*change mode*). Man kann es auf viele verschiedene Arten benutzen, aber die einfachste kann leicht anhand der folgenden Beispiele verstanden werden:

```
user$ chmod u+x program.sh
        # Gib dem Besitzer (u=user) das Recht 'x'

user$ chmod u=rwx program.sh
        # Setze den Rechtevektor des Besitzers
        # absolut. Gewähre dabei alle drei Rechte.

user$ chmod g-wx program.sh
        # Entziehe der Gruppe (g=group) die Rechte 'w'
        # und 'x'

user$ chmod o+r program.sh
        # Gib 'allen anderen' (o=others) das Recht 'r'

user$ chmod o=rx program.sh
        # Setze den Rechtevektor für 'alle anderen'
        # absolut. Gib hierbei die Rechte 'r' und 'x'.

user$ chmod a+x program.sh
        # Gib 'allen' (user, group und others)
        # das Recht 'x'
```

Einige Bemerkungen:

1. Sie werden in Abschn. 9.1 noch genauer lernen, was es mit dem (`x=execute`) Recht für Dateien auf sich hat.
2. Schauen Sie sich die Rechte von Dateien an, die Sie von außen in Ihren *Unix*-Account kopieren (Dateiübertragung von Ihrem *Microsoft Windows*-Rechner über das Internet, USB-Stick, CD). Es kann hier manchmal passieren, dass z. B. *alle Rechte* anfangs gesetzt sind!
3. Beachten Sie, dass das `w`-Flag des Verzeichnisses darüber entscheidet, ob eine Datei, die sich darin befindet, gelöscht werden darf! Es ist daher möglich, eine Datei zu löschen, obwohl man an ihr selbst *kein* Schreibrecht (`w`) hat (man kann einen Gegenstand aus dem Raum nehmen und wegwerfen, aber nicht verändern)! Man kann sich daher nicht vor dem versehentlichen Löschen einer Datei schützen, indem man sich das Schreibrecht an ihr entzieht.
4. Umgekehrt können Sie eine Datei in einem Verzeichnis, für das Sie keine `w`-Erlaubnis haben, keinesfalls löschen, auch wenn Sie Schreibrechte an der

Datei selbst haben (Sie dürfen ein Blatt Papier, das sich in dem Raum befindet, beschreiben, es aber nicht aus dem Raum entfernen)!
5. Eine andere Konsequenz ist, dass Sie Dateien anderer Benutzer (insbesondere des Superusers `root`) löschen können, falls Ihnen das Verzeichnis gehört!

Hier ein Beispiel für *Standardrechte*, wie sie neu angelegte Dateien und Verzeichnisse typischerweise haben:

```
drwxr-xr-x 2 thomas users 40 Oct 11 09:51 rechte

rechte:
total 16
-rwxr-xr-x 1 thomas users 9562 Oct 11 09:51 program.sh
-rw-r--r-- 1 thomas users  724 Oct 11 09:51 thomas.txt
```

Ich habe ein Verzeichnis `rechte` mit den Einstellungen `drwxr-xr-x`. Ich als Besitzer darf also den Verzeichnisinhalt *listen*, den Verzeichnisinhalt *verändern* (die Dateien `program.sh` und `thomas.txt` löschen oder neue Dateien hineinkopieren) und das Verzeichnis mit `cd` betreten. Die anderen zwei Benutzerklassen dürfen den Verzeichnisinhalt mit `ls` auflisten und das Verzeichnis betreten, aber *nichts* am Verzeichnisinhalt verändern! An den Dateien `program.sh` und `thomas.txt` habe ich als Besitzer alle wesentlichen Rechte. Ich darf mir den Dateiinhalt ansehen, die Dateien verändern (z. B. editieren), und ich darf das Programm `program.sh` ausführen (laufen lassen). Die anderen Benutzerklassen dürfen den Dateiinhalt ansehen und das Programm ausführen. Die Dateien verändern dürfen sie nicht!

Das Ganze klingt anfangs verwirrender, als es wirklich ist. Im Normalfall sind die Rechte neu angelegter Dateien vernünftig, und Sie müssen sich nicht weiter darum kümmern. Sie sollten lediglich sicherstellen, dass nur Sie selbst Rechte an *vertraulichen* Daten und Verzeichnissen haben!

Aufgabe 6.1

Welche Rechte brauchen Gruppenmitglieder, damit sie sich eine Datei `testdatei.txt` aus einem Verzeichnis `rechte` *kopieren* dürfen. Geben Sie die hierfür *minimal* nötigen Gruppenrechte an dem Verzeichnis und an der Datei an. Begründen Sie Ihre Antwort. Nehmen Sie an, dass die Gruppenmitglieder wissen, dass es diese Datei in besagtem Verzeichnis gibt!
Lösungsvorschlag auf Seite 184!

Aufgabe 6.2

Im Folgenden befinden sich in `~/hausaufgaben` die Datei `analysis_1.txt` und das Programm `getmean.sh`. Sie wollen,

a) dass sich alle Benutzer die Datei `analysis_1.txt` in `~/hausaufgaben` kopieren und das Skript ausführen dürfen. Sie selbst sollen sämtliche Rechte an ihren Dateien und Verzeichnissen haben.
b) dass Mitglieder Ihrer Gruppe die Dateien in `~/hausaufgaben` kopieren und das Skript ausführen dürfen. Alle anderen Benutzer sollen das Verzeichnis nicht betreten dürfen. Sie selbst sollen sämtliche Rechte an ihren Dateien und Verzeichnissen haben.

Geben Sie für beide Fälle die nötigen Rechte an dem Verzeichnis `~/hausaufgaben` und den beiden Dateien an. Die Rechte sind jeweils für Sie selbst, Ihre Gruppe und alle anderen Benutzer anzugeben. Sie können annehmen, dass alle Benutzer das Recht haben, das Verzeichnis `~/hausaufgaben` zu betreten.
Lösungsvorschlag auf Seite 184!

Aufgabe 6.3

Wie in Abschn. 5.4 diskutiert, befinden sich in Ihrem Heimatverzeichnis etliche *versteckte* Dateien und Verzeichnisse, die mit einem „.“ anfangen und hauptsächlich zur Systemkonfiguration genutzt werden. Eine dieser Dateien ist die `bash`-Konfigurationsdatei `.bashrc`, welche wichtige Einstellungen für Ihre Shell-Sitzungen beinhaltet. Es ist für Benutzer durchaus interessant, diese Einstellungen an eigene Bedürfnisse anzupassen (siehe Kap. 10). Man kann sich hier aber auch seinen Benutzer-Account böse *verkonfigurieren*. Um Letzteres, aber leider auch jegliche Konfigurationsmöglichkeit, zu verhindern, wurde früher auf unseren Universitäts-Accounts diese Datei für jeden Studenten folgendermaßen angelegt:

```
-rw-r--r-- 1 root root  724 Oct 11 09:51 .bashrc
```

Die Datei gehörte dem Superuser `root` und nur dieser durfte die Datei verändern. Alle anderen, insbesondere der Benutzer, in dessen Heimatverzeichnis sich die Datei befand, konnten die Datei *nur lesen*. Dies reichte aus, damit das System sie korrekt verwenden konnte.
Konnte diese Installation verhindern, dass die Studenten ihre Shell neu konfigurieren? Begründen Sie Ihre Antwort!
Lösungsvorschlag auf Seite 185!

Techniken zum effektiven Umgang mit der Shell

7

Übersicht

Zusammenfassung

In diesem Kapitel sehen wir uns einige grundlegende Techniken an, wie man sich bei der täglichen Arbeit auf der *Unix*-Shell viel Tipparbeit sparen kann.

7.1 Kopieren und Einfügen mit der Maus

Die meines Erachtens wichtigste Technik zum Einsparen von Tastenanschlägen unter *Unix* ist das *Kopieren und Einfügen* (im Englischen *copy and paste*) mit der Maus. Die Technik erlaubt es auf sehr einfache Weise, lange oder kompliziert strukturierte Zeichenfolgen (z. B. WWW-Adressen) fehlerfrei zwischen Anwendungen zu übertragen. Unter *Microsoft Windows* können entsprechende Textpassagen mit der Maus markiert werden und mit <C-c> (dem *Kopieren*-Schritt) in die sogenannte *Zwischenablage* kopiert werden. Von dort können sie mit <C-v> beliebig oft an neue Maus-Cursor-Positionen platziert werden (dem *Einfügen*-Schritt). Unter *Unix* wird Text durch einfaches Markieren mit der Maus bei gedrückter linker Maustaste automatisch in die Zwischenablage kopiert. Das Einfügen an einer neuen Maus-Cursor-Position erfolgt durch einfaches Drücken der mittleren Maustaste – siehe auch Abb. 7.1.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_7

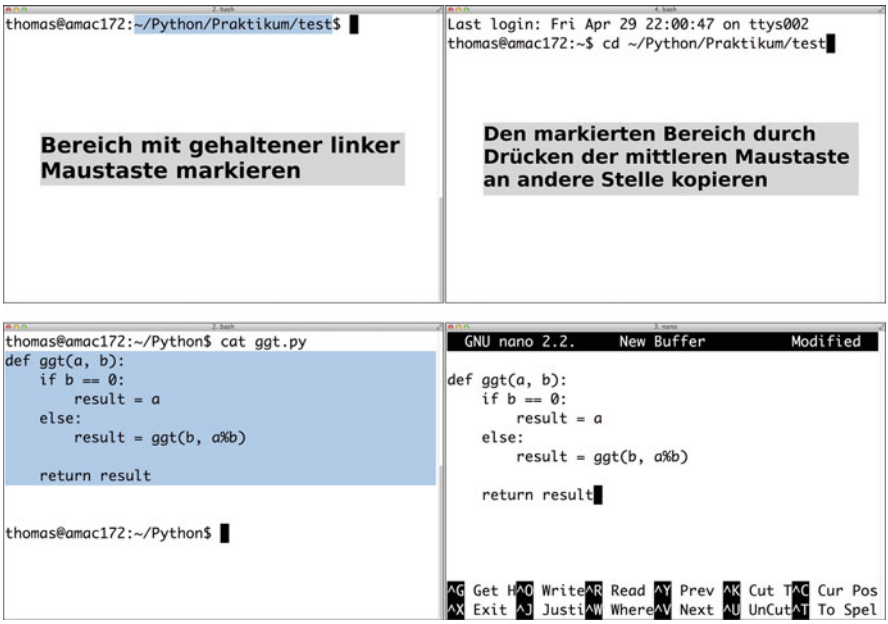

Abb. 7.1 Teile einer Eingabezeile oder gesamte Bereiche einer *beliebigen Unix*-Anwendung können mit der Maus markiert werden. Dazu die linke Maustaste gedrückt halten und Bereiche markieren. Die ausgewählten Textpassagen sind farblich unterlegt. Durch Drücken der mittleren Maustaste kann der markierte Bereich in beliebige Anwendungen an die Position des Maus-Cursors kopiert werden.
Oben links wurde ein Pfad markiert, welcher in einem anderen Terminal (oben rechts) für einen `cd`-Befehl verwendet wird. Unten wurde ein größerer Bereich eines Terminals (links) zur weiteren Verarbeitung in einen Editor (rechts) kopiert

7.2 Tastenkürzel auf der Kommandozeile

Beim Eintippen von Kommandos in die Shell kann es manchmal zu Tippfehlern kommen. Sie brauchen dann das bereits eingegebene Kommando nicht wieder vollständig zu löschen oder neu einzugeben, denn es stehen Ihnen für das Editieren der Kommandozeile etliche Tastenkürzel zur effektiven Arbeit zur Verfügung. Die wichtigsten sind in Tab. 7.1 zusammengefasst.

7.3 Die `<Tab>`-Completion

Die automatische Vervollständigung von Kommandos und Argumenten mithilfe der `<Tab>`-Taste ist meines Erachtens das wichtigste Tastaturhilfsmittel für ein effektives Schreiben auf der Kommandozeile. Die sogenannte `<Tab>`-Completion

Tab. 7.1 Tastenkürzel zum effektiven Arbeiten auf der *Unix*-Shell

Tastenkürzel	Effekt auf der Kommandozeile
↑, ↓	Blättern in alten Befehlen
→, ←	Cursor auf der Kommandozeile vor und zurück bewegen
<C-a>	Gehe zu Zeilenanfang
<C-e>	Gehe zu Zeilenende
<C-k>	Lösche Zeile ab Cursor-Position
<C-l>	Löscht Terminalinhalt und gibt neuen Prompt in der ersten Terminalzeile
<C-r>	Befehlsrückwärtssuche (siehe Abschn. 7.4)
<Tab>	TAB-Completion (siehe Abschn. 7.3)

wirkt bei der Vervollständigung von Kommandos, Dateinamen, Benutzernamen nach der Tilde „∼“ und von *Shell-Variablen*:

```
user$ so<Tab><Tab>        # Liste aller möglichen
                          # Kommandos
soapsuds      soelim        sort          source
socklist      soffice
user$ sor<Tab>
user$ sort                # automatisch ergänzt
#
user$ sort te<Tab><Tab>   # Liste aller möglichen
                          # Dateinamen
test.sh  text.txt
user$ sort tex<Tab>
user$ sort text.txt       # automatisch ergänzt
#
user$ cd ~te<Tab>
teifler  terben   tereno
user$ cd ~ter<Tab>
terben   tereno
user$ cd ~tere<Tab>
user$ cd ~tereno/         # automatisch gesetzt
#
user$ echo ${U<Tab><Tab>
${UBUNTU_MENUPROXY}  ${UPSTART_SESSION}
${UID}               ${USER}
user$ echo ${US<Tab>
user$ echo ${USER}        # von der Shell ergänzt
```

Das letzte Beispiel mit den *Variablen* wird Ihnen nach Studium von Abschn. 10.1 verständlich. Es ist hier hauptsächlich der Vollständigkeit halber erwähnt.

Tab. 7.2 Zusammenfassung der <Tab>-Completion

<Tab>-Kürzel	Erklärung
BEFEHL<Tab>	Name des Kommandos BEFEHL vervollständigen wenn eindeutig
BEFEHL<Tab><Tab>	Liste der möglichen Kommandoalternativen
BEFEHL DATEI<Tab>	Dateinamen DATEI vervollständigen
~BENUTZER<Tab>	Benutzernamen BENUTZER vervollständigen
${VARIABLE<Tab>	Variablennamen VARIABLE vervollständigen

Eine tabellarische Zusammenfassung der <Tab>-Completion finden Sie in Tab. 7.2. Die <Tab>-Completion wird auch innerhalb vieler Programme direkt unterstützt.

7.4 Der History-Mechanismus

Ein weiteres Hilfsmittel, das Ihnen beim Schreiben sehr viel Arbeit abnimmt, sind die verschiedenen *Shell-History*-Funktionen, mit denen man einmal eingegebene Kommandos wiederholen oder in leicht veränderter Form wiederverwenden kann. Von den recht zahlreichen Möglichkeiten sind für mich bei der täglichen Arbeit die folgenden relevant (erste beschriebene Möglichkeit wird von mir am meisten benutzt):

1. Das Blättern durch bereits eingegebene Befehle mit den Cursor-Tasten ↑ und ↓. Hierbei kann jeder angezeigte, alte Befehl auf der Kommandozeile neu editiert, und eventuell leicht modifiziert erneut ausgeführt werden.
2. Die *Rückwärtssuche* in alten Kommandos mit <C-r>. Geben Sie diese Kombination auf der Kommandozeile ein, so wird die Rückwärtssuche in Ihrer Befehls-History aufgerufen. Geben Sie die ersten Buchstaben eines alten Befehls ein (z. B. ls), dann sucht die *Bash* nach alten Eingaben, die die entsprechenden Zeichen enthalten, und schreibt sie direkt in die Eingabezeile hinter Ihrer Eingabe.

   ```
   user$ <C-r>
   user$ (reverse-i-search)'ls':  ls -ltr *21C*sub.fits
   ```

   Es wird der erste Eintrag angezeigt, der auf Ihre Eingabe passt. Erneutes <C-r> führt zur Anzeige weiter zurückliegender, passender Einträge.
3. Sich mit dem Befehl history eine Liste alter Kommandos geben lassen und Relevantes durch *Copy und Paste* mit der Maus wiederverwenden.
4. Es gibt noch die Möglichkeit, Einträge der *History*-Liste mit dem Ausrufezeichen !-Kommando anzusprechen und auszuführen. Zum Beispiel:

```
user$ history
   .
   .
   391  ls
   392  cvs diff .
   393  emacs preprocess.c &
   394  cvs diff .
   395  uname
   396  uptime
user$ !391
ls
vorl_02.tex
user$ !cvs
cvs diff .
.
.
user$ !u
uptime
4:59pm  up 35 days 23:38 ...
user$ !un
uname
Linux
user$
```

Entweder gibt man nach dem `!` die Nummer in der History-Liste an oder einen Teil-String, der das gewünschte Kommando *hinreichend eindeutig* identifiziert. Im obigen Beispiel spricht `!u` das letzte Kommando, das mit `u` beginnt, an, also `uptime`. Erst `!un` identifiziert eindeutig `uname`. Das Wichtigste zum *history*-Befehl ist in Tab. 7.3 zusammengefasst.

Tab. 7.3 Zusammenfassung der wichtigsten Shell-History Befehle

History-Befehl	Kurzbeschreibung
`history`	Die letzten Kommandos nummeriert anzeigen
`!ZAHL`	Kommando mit der History-Nummer `ZAHL` wiederholen
`!TEXT`	*Jüngstes* Kommando mit Textanfang `TEXT` wiederholen
`!!`	Letztes Kommando wiederholen
`!-2`	Vorletztes Kommando wiederholen usw.

Sonderzeichen der Shell

8

Zusammenfassung

Viele Zeichen lösen bei der Eingabe auf der Kommandozeile eine spezielle Aktion der Shell aus. Sie haben dort also eine weitergehende als nur ihre *wörtliche* Bedeutung. Wir sehen uns an, wie Sie die Spezialbedeutung dieser Zeichen bei Bedarf aufheben können. Des Weiteren werden wir hier eine Zusammenfassung der Spezialzeichen geben, die Sie bisher kennengelernt haben oder die wir in späteren Kapiteln dieses Buchs noch behandeln werden.

Sie haben in früheren Kapiteln gesehen, dass in der Shell bestimmte Zeichen Sonderstellungen haben oder besondere Aktionen auslösen. Unter anderem haben Sie davon bisher Folgendes kennengelernt:

- Das Leerzeichen ␣ dient als *Trenner* von Kommando, Kommandooption und Argumenten eines *Unix*-Befehls.
- Die Wildcards `*`, `?` und Zeichen in einem eckigen Klammerpaar, zusammen mit eventuellen Pre- und Postfixen, expandieren bei Befehlsausführung zu *vorhandenen* Dateinamen.
- Die Tilde „∼“ ist ein Alias für Ihr Heimatverzeichnis.

Sie haben auch gesehen, dass die Sonderzeichen zu gewissen Problemen und Einschränkungen führen, wenn man sie in ihrer *wörtlichen* Bedeutung verwenden möchte. Zum Beispiel war es in Aufgabe 5.10 nicht ohne Weiteres möglich, mit dem Kommando `echo` einen einfachen „`*`“ auszugeben. Dateien mit Leerzeichen bereiten Probleme, wenn man sie ansprechen möchte (siehe Aufgabe 5.7). Hier wird oft das Leerzeichen fälschlicherweise als Ende des Dateinamens interpretiert. Ganz generell sollte man Shell-Sonderzeichen in Datei- und Verzeichnisnamen vermeiden. Dies führt fast immer zu Schwierigkeiten. Das Problem aus Benutzersicht ist, dass die Shell ihre Sonderzeichen *vor* einem Programmaufruf behandelt. Dies führt bei

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_8

Tab. 8.1 Möglichkeiten zur *Maskierung/Quotierung* von Sonderzeichen der Shell

Symbol	Wirkung
\	Maskiert das direkt folgende Zeichen.
"..."	Maskiert eine Zeichenkette. Alles außer $, ‘, \ und ! wird geschützt.
'...'	Maskiert *alle* Sonderzeichen einer Zeichenkette.

```
user$ echo *
```

dazu, dass *zuerst* die Sonderfunktion des Asterisk ausgewertet wird und nur das Ergebnis dieser Auswertung an den `echo`-Befehl weitergereicht wird. Der Befehl bekommt hier also den „`*`“ selbst *nie* zu sehen! Soll die Sonderfunktion von Zeichen aufgehoben werden, so müssen diese vor der Auswertung durch die Shell geschützt oder *maskiert* werden (aus dem Englischen *to quote special shell-characters* ist noch der Ausdruck Shell-Sonderzeichen *quotieren*, bzw. *quoten* sehr gebräuchlich). Hierfür gibt es drei Möglichkeiten, die in Tab. 8.1 zusammengefasst sind.

Beispiele:

```
user$ echo *
vorl_02.tex vorl_02.dvi vorl_02.log ...
user$ echo \*
*
user$ echo "*"
*
user$ touch "Das ist ein schlechter Name.txt"
user$ rm Das\ ist\ ein\ schlechter\ Name.txt
user$ echo ${SHELL}   # schon mal gesehen?
/bin/bash
user$ echo "${SHELL}"
/bin/bash
user$ echo '${SHELL}'
${SHELL}
```

Die letzten drei Beispiele werden Ihnen nach dem Studium von Kap. 10 klarer sein!

Aufgabe 8.1

Vervollständigen Sie folgende drei `echo`-Befehle, um die gezeigten Ausgaben zu erhalten:

```
user$ echo ...
"Ein Text"
user$ echo ...
'Mehr Text'
user$ echo ...
'Noch mehr Text"
```

Lösungsvorschlag auf Seite 185!

Wir werden in Kap. 15 sehen, warum es *sehr nützlich* ist, mit den doppelten Anführungsstrichen `"..."` ein Konstrukt zur Verfügung zu haben, das *fast alle* Sonderzeichen quotiert. Um ausnahmslos *alle* Zeichen in ihrer wörtlichen Bedeutung zu verwenden, dienen die einfachen Anführungsstriche.

Falls Sie Fehlermeldungen oder Ausgaben bekommen, die Ihnen *vollkommen unverständlich* erscheinen, so ist häufig die unbewusste Nutzung von Shell-Sonderzeichen die Ursache:

```
user$ echo "Er benutzte das Passwort !Test!"
bash: !Test!: event not found
```

Erklärung: Wie oben beschrieben, schützt die `"..."`-Quotierung das „`!`“ nicht. In Abschn. 7.4 haben wir gesehen, dass das `!` den History-Mechanismus der Shell initiiert. Die Zeichenfolge `!Test!` sucht nach einem Befehl, der mit der Zeichenkette `Test!` beginnt. Da ich einen solchen Befehl in meiner Sitzung nie eingegeben habe, kommt es nach der Eingabe `echo "Er benutzte das Passwort !Test!"` zu der Fehlermeldung `bash: !Test!: event not found`.

Aufgabe 8.2

Sie wollen mit dem `echo`-Befehl den *Microsoft Windows*-Pfad `C:\Benutzer\Thomas` ausgeben:

```
user$ echo C:\Benutzer\Thomas
C:BenutzerThomas
```

Erklären Sie, warum es hier zu der Ausgabe `C:BenutzerThomas` kommt. Geben Sie dann einen Befehl an, der die erwünschte Ausgabe liefert.
Lösungsvorschlag auf Seite 186!

In Tab. 8.2 gebe ich eine Zusammenstellung von Sonderzeichen, die wir bisher behandelt haben bzw. die wir noch besprechen werden.

Tab. 8.2 ist keineswegs eine *vollständige* Auflistung von Shell-Sonderzeichen und Sonderzeichenkombinationen. Es ist als Übersicht gedacht, welche Sonderzeichen auch ein Anfänger kennen sollte. Auf einige Punkte möchte ich noch explizit hinweisen:

1. Mit zunehmender Erfahrung werden Sie lernen, *wann* Shell-Zeichen eine Sonderfunktion haben, und somit genauer wissen, wo man besonders vorsichtig sein muss. Viele Zeichen haben *nur unter ganz bestimmten Umständen* eine Sonderfunktion. Wir haben z. B. die Tilde (~) kennengelernt, welche *am Anfang einer Zeichenkette* zu einem Heimatverzeichnis expandiert, aber ansonsten keine spezielle Bedeutung hat:

   ```
   user$ cd ~terben     # wechselt in das Heimat-
                        # verzeichnis des Benutzers
                        # 'terben'
   ```

Tab. 8.2 Zusammenfassung von Shell-Sonderzeichen. Unterhalb des dicken horizontalen Trennstrichs befinden sich Sonderzeichen, die wir erst in späteren Kapiteln behandeln werden

Sonderzeichen	Referenz Abschnitt	Kurzbeschreibung
#	Abschn. 2.3	Leitet einen Kommentar auf der Kommandozeile ein
␣, <Tab>, <Return>	Abschn. 4.1	Trennzeichen zwischen Kommmandos, Optionen und Argumenten (␣ und <Tab>) bzw. Einleitung der Befehlsausführung (<Return>)
~	Abschn. 5.2	Expandiert *alleinstehend oder am Anfang einer Zeichenkette* zum absoluten Pfad eines Heimatverzeichnisses
/	Abschn. 5.1	Dient als Verzeichnistrenner in *Unix*-Pfaden
*, ?, [...]	Abschn. 5.5	Wildcards für den effektiven Zugriff auf mehrere Dateien und/oder Verzeichnisse
!	Abschn. 7.4	Shell-History-Kommando
\, "..", '..'	Kap. 8	Quotierungskonstrukte, um Shell-Sonderzeichen in ihrer *wörtlichen* Bedeutung zu benutzen
>, <, |	Kap. 13	Lenkt die (Standard-)Ausgabe (>), bzw. die Standardeingabe (<) in eine Datei um. Der vertikale Strich baut eine Pipeline zwischen *Unix*-Befehlen auf
&	Abschn. 9.2.1	Startet einen *Unix*-Prozess im Hintergrund
$, =	Abschn. 10.1	Liefert den Wert einer Shell-Variablen ($), bzw. weist einer Shell-Variablen einen Wert zu (=).
;	Abschn. 10.4.2	Erlaubt die Eingabe mehrerer *Unix*-Befehle in einer Zeile

```
user$ cd ~            # wechselt in mein Heimat-
                      # verzeichnis
user$ touch ~terben
touch: setting times of '/home/terben': \
   Permission denied
user$ touch t~erben # kein Problem!
user$
```

2. Er gibt *Zeichenkombinationen* mit einer Sonderbedeutung. Wir werden in Kap. 13 unter anderem die Kombinationen „2>“ (Umleitung der Fehlerausgabe in eine Datei) und „>&“ (Umleitung von Standardausgabe *und* der Fehlerausgabe in ein und dieselbe Datei) kennenlernen.
3. Der Bindestrich „-“ ist formal kein Shell-Sonderzeichen. Da er für Kommandooptionen eine zentrale Bedeutung hat, sollte man aber Dateinamen, die mit einem Bindestrich anfangen, vermeiden. Um auch *Argumente*, die mit einem Bindestrich beginnen, verwenden zu können, unterstützen viele *Unix*-Kommandos die

spezielle Option „--“. Diese signalisiert, dass *alle* nachfolgenden Zeichenketten als *Argumente* zu behandeln sind:

```
user$ ls
user$ touch -i -x
touch: invalid option -- 'i'
Try 'touch --help' for more information.
user$ touch -- -i -x
user$ ls
-i  -x
user$ rm -i -x
rm: invalid option -- 'x'
Try 'rm ./-i' to remove the file '-i'.
Try 'rm --help' for more information.
user$ rm -- -i -x
user$ ls
user$
```

Etliche Kommandos haben aber auch andere Möglichkeiten, dies zu realisieren. Hier müssen Sie bei Bedarf die Dokumentation befragen.

Wie bereits mehrfach diskutiert, gibt es die meisten Probleme, wenn Sonderzeichen in Datei- oder Verzeichnisnamen auftreten. Daher gebe ich Ihnen an dieser Stelle noch einmal den expliziten Rat, darauf komplett zu verzichten. Ich denke insbesondere an die unter *Microsoft Windows* beliebten Dateinamen mit Leerzeichen.

Aufgabe 8.3

Nennen Sie drei Situationen, in denen der Punkt (`.`) eine Shell-Sonderfunktion hat. Wann kann man den Punkt als Teil eines Dateinamens *ohne* mögliche Überraschungen verwenden?
Lösungsvorschlag auf Seite 187!

Aufgabe 8.4

In einem Verzeichnis befinde sich eine Datei mit Namen `-i`. Was geschieht, wenn man in diesem Verzeichnis den Befehl `rm -rf *` ausführt?
Lösungsvorschlag auf Seite 187!

Programmausführung und Prozesskontrolle 9

Übersicht

Zusammenfassung

Wir besprechen nun, wie Sie Programme, die Sie von externen Quellen auf Ihren Account kopieren und solche, die Sie selbst schreiben, ausführen können. Danach beschäftigen wir uns mit der Kontrolle laufender Prozesse. Viele wissenschaftliche Analyseprogramme beenden sich nicht instantan, sondern laufen oft viele Stunden oder Tage. Daher ist es sehr wichtig, zu wissen, wie Sie solche Programme identifizieren und gegebenenfalls beeinflussen können.

9.1 Programmausführung unter *Unix*

Sie haben bisher *Unix*-Kommandos wie `ls` ausgeführt. In diesem Abschnitt erkläre ich, wie Sie eigene Programme oder solche, die Sie aus dem Internet herunterladen, ausführen können. Unter den Dateien zu dieser Einheit befindet sich das *Shell-Skript* `countdown_process.sh`. Akzeptieren Sie für den Moment einfach, dass es sich um ein ausführbares *Unix*-Programm handelt. Wechseln Sie in das Verzeichnis, in dem sich das Programm befindet:

```
user$ countdown_process.sh
countdown_process.sh: command not found
```


T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_9

Hier gibt es *im Allgemeinen* zwei Probleme zu beachten:

1. Um ein Programm ausführen zu dürfen, ist das `x=execute`-Recht nötig:

```
user$ ls -l countdown_process.sh
-rw-r--r-- 1 thomas users 554 Oct 16 17:07 \
    countdown_process.sh

user$ chmod u+x countdown_process.sh
user$ ls -l countdown_process.sh
-rwxr--r-- 1 thomas users 554 Oct 16 17:07 \
    countdown_process.sh

user$ countdown_process.sh
countdown_process.sh: command not found
```

   Das löst das Problem also noch nicht wirklich.

   Hinweis: Auf modernen *Unix*-Installationen sollte bei Ihnen dieses Beispiel wie beschrieben ablaufen. Es *kann* jedoch sein, dass der letzte Programmaufruf von `countdown_process.sh` bei Ihnen bereits erfolgreich ist. Sie werden innerhalb von Aufgabe 10.1 sehen, unter welchen Bedingungen ein Aufruf von `countdown_process.sh` an dieser Stelle bereits funktioniert.
2. Es ist *hier* nötig, den Pfad des Programms mit anzugeben:

```
user$ pwd
/home/thomas
user$ /home/thomas/countdown_process.sh
.
.
Die Variable 'i' hat jetzt den Wert 2
Die Variable 'i' hat jetzt den Wert 1
Die Variable 'i' hat jetzt den Wert 0

Mit dem Erreichen des Wertes 0 fuer die Variable
'i' ist dieses Skript beendet.
```

   Es geht natürlich auch:

```
user$ ./countdown_process.sh
.
.
.
Mit dem Erreichen des Wertes 0 fuer die Variable
'i' ist dieses Skript beendet.
```

   Sie wundern sich hier wahrscheinlich, dass es bei Programmen wie `ls` nicht nötig ist, den gesamten Pfad anzugeben. Der Grund ist, dass *Unix* einen gewissen Satz an Pfaden hat, *in dem es nach Programmen sucht*, die ohne explizite Pfadangabe aufgerufen werden. Da *Unix* das Programm `countdown_process.sh`

in seinen Standardpfaden nicht findet, kommt es zu der Fehlermeldung `countdown_process.sh: command not found`. Um den Pfad eines *Unix*-Programms zu finden, dient der Befehl `which`:

```
user$ which ls
/bin/ls
```

Das Programm `ls` befindet sich also in `/bin`. Wie Sie den Satz an *Unix*-*Programmsuchpfaden* um eigene Verzeichnisse erweitern können, erkläre ich in Kap. 10. Natürlich möchte man für den Aufruf eigener Programme nicht immer den gesamten Pfad angeben. Üblich ist es hier z. B., ein Verzeichnis `~/bin` (*bin* = *binary*) anzulegen, eigene Programme dorthin zu kopieren und dann `~/bin` dem System als *Unix*-Programmsuchpfad bekannt zu machen.

Für den Moment merken Sie sich einfach, dass Sie ein eigenes Programm mit dem gesamten Pfad aufrufen müssen. Des Weiteren muss immer das `x`-Recht gesetzt sein.

9.2 *Unix*-Prozesskontrolle

Nachdem Sie jetzt wissen, wie Sie Programme starten können, müssen wir uns auch ansehen, wie Programmabläufe, wenn nötig, beeinflusst werden. Dies wird z. B. dann wichtig, wenn Programme nicht mehr *instantan* ein Ergebnis liefern und sofort nach dem Start wieder beendet sind, sondern langwierige Analyseprozesse durchführen. Ein Programm, das auf einem *Unix*-Rechner ausgeführt wird, bezeichnet man hierbei allgemein als *Prozess*. Der Komplex *Prozesssteuerung* oder *Prozesskontrolle* gliedert sich in mehrere Teilbereiche.

9.2.1 Vorder- und Hintergrundprozesse verwalten

Wenn Sie ein länger laufendes Programm in einem Terminal starten, ist die Shell so lange blockiert, bis das Programm beendet wird. Man sagt, das Programm läuft im *Vordergrund* der Shell:

```
user$ firefox
```

Das Terminal meldet sich erst wieder mit einem neuen Prompt zurück, wenn Sie `firefox` beenden. Alternativ kann man Programme im *Hintergrund* laufen lassen, was durch ein hinten angestelltes `&`-Zeichen beim Programmaufruf erreicht wird:

```
user$ firefox &
[1] 28298
user$
```

In diesem Fall wird die Shell sofort nach dem Programmstart wieder für neue Eingaben freigegeben und das Programm läuft im *Hintergrund* weiter. Die Ausgabe `[1] 28298` besagt, dass dies der erste Hintergrundprozess `[1]` in dieser Shell ist

und dass dem Prozess die Identifikationsnummer `28298` zugewiesen wird. Diese `PID` (*Process Identification*) wird wichtig, wenn wir im Hintergrund laufende Prozesse beeinflussen möchten – siehe Abschn. 9.2.2. Man kann auch im Vordergrund laufende Prozesse nachträglich *in den Hintergrund schieben*:

```
user$ firefox
<C-z>
Suspended
user$ bg
[1]    firefox &
```

Nach dem Start eines Vordergrundprozesses drücken Sie `<C-z>`, um den Prozess *anzuhalten*. Er verweilt im Speicher und setzt seine Arbeit erst fort, wenn er explizit dazu aufgefordert wird. Dies geschieht durch den nachfolgenden `bg`-Befehl (*background*). Der Befehl `bg` setzt nicht nur den *zuletzt angehaltenen Prozess* fort, sondern veranlasst ihn auch, seine Aufgaben ab sofort im Hintergrund zu erledigen:

```
user$ firefox
<C-z>
user$ emacs
<C-z>
user$ bg
[3]  emacs &
user$ bg
[2]  firefox &
```

Das Wesentliche, was Sie zu Vorder- und Hintergrundprozessen wissen müssen, ist in Tab. 9.1 zusammengefasst.

Bemerkungen:

1. Man startet üblicherweise alle langfristigen Prozesse von vornherein im Hintergrund.
2. Vorder- und Hintergrundprozesse in einer Shell werden automatisch beendet, sobald man die Shell schließt. Unten beschreibe ich, wie man das für längere Programmläufe verhindern kann.
3. Die Kommandos zur Vorder- und Hintergrundprozesskontrolle beziehen sich auf *die gegenwärtige Shell*. Mit jedem Öffnen eines neuen Terminals wird auch eine neue Instanz der Shell gestartet.

Tab. 9.1 Tastenkombinationen und Kommandos zur Vorder- und Hintergrundprozessverwaltung

Tastenkombination / Kommando	Wirkung
`BEFEHL ... &`	Prozess `BEFEHL` im Hintergrund starten
`<C-z>`	Aktuellen Vordergrundprozess stoppen
`bg`	Letzten angehaltenen Prozess im Hintergrund ausführen
`fg`	Letzten angehaltenen Prozess im Vordergrund ausführen
`jobs`	Hintergrundprozesse der gegenwärtigen Shell anzeigen

9.2.2 Prozesse finden und beenden

Als Nächstes sehen wir uns an, wie wir laufende Prozesse finden und beenden können. Dies ist z. B. dann nötig, wenn wir bemerken, dass ein laufendes Programm fehlerhaft ist. Ein häufig auftretender Programmierfehler sind sogenannte *Endlosschleifen*. Ein gewisser Programmabschnitt soll wiederholt durchgeführt werden, bis irgendeine Bedingung eintritt. Falls aus irgendeinem Grund diese *Abbruchbedingung nie* eintritt, würde das Programm *für immer* weiterlaufen und Rechenzeit verschwenden. Das Beenden eines im Vordergrund laufenden Prozesses ist hierbei recht einfach und kann durch <C-c> erreicht werden:

```
user$ ./endless_process.sh
Die Variable 'i' hat jetzt den Wert 10
Die Variable 'i' hat jetzt den Wert 11
Die Variable 'i' hat jetzt den Wert 12
Die Variable 'i' hat jetzt den Wert 13
Die Variable 'i' hat jetzt den Wert 14
.
.
<C-c>     # <C-c> beendet Prozesse im Vordergrund!
^CJa, ja, ich beende mich ja schon .....
user$
```

Das Programm `endless_process.sh` ist ähnlich zu `countdown_process.sh`, das Sie oben gesehen haben. Hier hat der Programmierer den Fehler gemacht, den Countdown-Zähler nach oben statt nach unten zu zählen, sodass *„Die Variable ‚i' hat jetzt den Wert 0"* nie erreicht wird! Sobald wir das merken, können wir <C-c> drücken, um den Prozess zu beenden. Komplizierter wird die Sache, wenn der Prozess im Hintergrund läuft, da wir hier keine direkte Kontrolle mehr über ihn haben. Hier kommt die oben erwähnte `PID` ins Spiel:

```
user$ ./endless_process.sh &
[3] 3222
Die Variable 'i' hat jetzt den Wert 10
Die Variable 'i' hat jetzt den Wert 11
Die Variable 'i' hat jetzt den Wert 12
Die Variable 'i' hat jetzt den Wert 13
Die Variable 'i' hat jetzt den Wert 14
.
.
```

Ein Drücken von <C-c> hat hier keinen Effekt – der Prozess läuft unbeeindruckt im Hintergrund weiter. Gehen Sie in ein neues Terminal:

```
user$ ps -a
  PID TTY          TIME CMD
 3222 pts/1    00:00:00 endless_process
```

```
 6344 pts/1    00:00:00 sleep
 6352 pts/0    00:00:00 ps
 6829 pts/1    00:00:36 emacs
15807 pts/1    00:00:44 acroread
```

Das Kommando `ps` (*process status*) listet laufende Prozesse. Bitte beachten Sie hier, dass die Ausgabe bei Ihnen natürlich *ganz anders* aussieht und davon abhängt, welche Prozesse unter ihrem Account gerade laufen. Die Ausgabe besteht aus folgenden Spalten: (1). die `PID`; (2). eine Identifikationsnummer für das Terminal, in dem der Prozess läuft (*Teletype*, `TTY`); (3). die vom Prozess akkumulierte CPU-Zeit und (4). das Kommando des Prozesses. Die für uns wichtige Information ist die `PID` von `endless_process.sh`:

```
user$ kill 3222
```

Der Befehl `kill` *bittet* den Prozess mit der `PID 3222`, sich zu beenden. Im *Computerjargon* sagt man, dass dem Prozess ein *Signal* geschickt wird. Hier ist es das Signal mit der Aufforderung, die Arbeit einzustellen. Auch das oben demonstrierte *Anhalten* und *Fortsetzen* von Prozessen wird über *Signale* realisiert. Der Befehl `kill PID` ist identisch mit `kill -TERM PID` – schicke dem Programm das `-TERM` (*Terminate*) Signal. Nach Eingabe des `kill`-Befehls erscheint in dem Terminal, in dem `endless_process.sh` lief, etwas wie:

```
Die Variable 'i' hat jetzt den Wert 697
Die Variable 'i' hat jetzt den Wert 698
Die Variable 'i' hat jetzt den Wert 699
Ja, ja, ich beende mich ja schon .....

[6]+  Terminated              ./endless_process.sh
# eventuell müssen Sie hier noch <Return> drücken, um
# zum Befehlsprompt zurückzukehren.
user$
```

Sie werden irgendwann einmal feststellen, dass ein *normales* `kill`-Kommando einen Prozess nicht *immer* beendet. Dies passiert, wenn der Prozess aus irgendeinem Grund nicht mehr in der Lage ist, *freundliche* Signale zu empfangen oder zu verarbeiten. In diesem Fall müssen Sie den Befehl `kill -KILL` verwenden, was einen Prozess *auf alle Fälle* beendet. Angewendet auf das Programm `endless_process.sh` sähe obiger Dialog etwas anders aus:

```
Die Variable 'i' hat jetzt den Wert 85
Die Variable 'i' hat jetzt den Wert 86
Die Variable 'i' hat jetzt den Wert 87

[3]+  Killed                  ./endless_process.sh
user$
```

Es fehlt hier die Meldung `Ja, ja, ich beende mich ja schon .....` und das Beenden wird mit `Killed` anstatt mit `Terminated` quittiert. Auf die

Bitte (`kill -TERM`), sich zu beenden, gebe ich noch eine Meldung aus. Im Fall von `kill -KILL` hat das Programm keine Möglichkeit mehr, vor seiner Beendigung irgendwie zu reagieren!

Bemerkungen:

1. Die Aufgabe des `kill`-Programms ist es, Signale an Programme zu versenden. Obwohl die Signale `-TERM` (eine Bitte an ein Programm, sich zu beenden) und `-KILL` (bedingungslose Beendigung eines Programms) sicher die am häufigsten verwendeten Optionen sind, so sind es doch nur zwei aus über 30(!) möglichen. Daher finde ich den Namen dieses Programms sehr ungünstig gewählt.
2. Benutzen Sie `kill -KILL` nur, wenn es absolut nötig ist. Hier hat das Programm keinerlei Möglichkeiten mehr, vor der Beendigung irgendetwas zu tun, z. B. temporäre Dateien zu löschen oder noch nicht gespeicherte Daten auf die Platte zu schreiben.
3. Behalten Sie im Hinterkopf, dass ein Prozess unter Umständen selbst neue Prozesse initiieren kann. Man spricht hier von *Eltern-* und *Kindprozessen*. Falls man solch einen Prozess beenden muss, sollten auch alle Kindprozesse mit aufgeräumt werden. Die Identifikation von Prozessketten und Abhängigkeiten ist hierbei oft nicht mehr einfach.
4. Der Befehl `ps` in der hier vorgestellten Form zeigt nur einen sehr geringen Teil der an ihrem Rechner laufenden Prozesse. Schauen Sie sich bei Gelegenheit die möglichen Optionen genauer an. Ein weiteres, *sehr* nützliches Werkzeug, um sich laufende Prozesse anzusehen, ist das Programm `top` oder dessen Weiterentwicklung `htop`. Während `ps` nur eine *Momentaufnahme* von Prozessen liefert, gibt `top` in periodischen Abständen eine aktualisierte Anzeige von Systeminformationen und von laufenden Prozessen. (Das `top`-Kommando ist mit dem *Microsoft Windows-Taskmanager* vergleichbar.) Ein typischer `top`-Bildschirm ist in Abb. 9.1 dargestellt. Mit der Eingabe von `q` können Sie das Programm beenden. Auch hier lohnt es sich, die Optionen dieses Programms genauer anzusehen, insbesondere um nach gewissen Prozessen zu filtern. (Ein Drücken von `u` in einem `top`-Terminal erlaubt z. B. das Selektieren aller Prozesse eines bestimmten Benutzers.) Es ist mir wichtig, dass Sie im Moment folgende wesentliche Punkte mitnehmen: (1) Alle Programme haben eine Identifikationsnummer (`PID`), die man mit Programmen wie `ps` und `top` herausfinden kann; (2) mit dem `kill`-Kommando lassen sich im Hintergrund laufende Prozesse, falls nötig, *von Hand* beenden. Hierzu benötigt man wiederum die PID.

9.2.3 Lang laufende Prozesse verwalten

In der Regel werden noch laufende Prozesse eines Benutzers vom System beendet, sobald er die Shell, in der die Prozesse laufen, schließt. Programme werden in diesem Fall intelligent mit dem `-TERM` (*Terminate*)-Signal vom System beendet. Wenn Sie längere Programme starten, sind Sie in der Regel nicht die gesamte Programmlaufzeit am System angemeldet und Sie schließen daher alle offenen

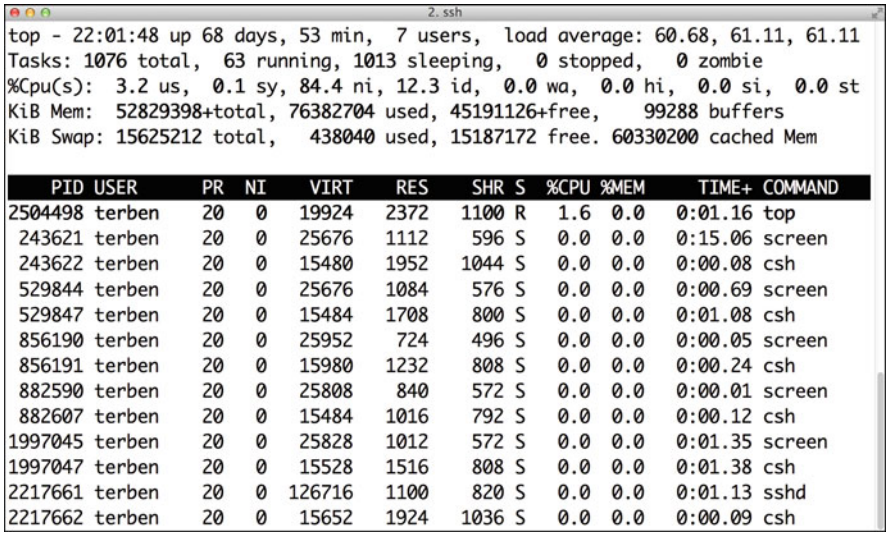

Abb. 9.1 `top`-Momentaufnahme: Die Anzeige von `top` variiert mit der Größe Ihres Terminals. Wenn Sie das Terminalfenster größer machen, werden weitere Zeilen mit laufenden Prozessen angezeigt. Die Standardsortierung erfolgt nach der CPU-Belastung, die ein Prozess verursacht. Dies, und vieles andere ist durch einfache Tastendrücke modifizierbar. `h` zeigt eine Hilfe

Shell-Instanzen irgendwann. In diesem Fall müssen Sie Programme so starten, dass sie auch nach dem Beenden ihrer Shell im Hintergrund weiterlaufen. Dies geschieht in der einfachsten Form mithilfe des Kommandos `nohup` (*no hangup*):

```
user$ nohup ./endless_process.sh &
nohup: ignoring input and appending output to \
'nohup.out'
user$
```

Eventuell müssen Sie nach dem `nohup`-Befehl nochmal `<Return>` drücken. Die Ausgabe besagt, dass das Programm `endless_process.sh` keine Eingaben von der sogenannten *Standardeingabe* (siehe Abschn. 13.2) bekommt und die gesamte Ausgabe in die Datei `nohup.out` geschrieben wird. Wenn Sie sich ausloggen, ist es ja sinnvoll, dass Programmausgaben, die normalerweise an das Terminal gehen, irgendwo gespeichert werden und nicht einfach *verloren gehen*:

```
user$ ls -ltr
.
.
-rw------- 1 thomas users  14087 Oct 18 22:18 \
  nohup.out
```

Im vorliegenden Fall würde das Programm nie enden und die Datei `nohup.out` immer länger werden. Beenden Sie gegebenenfalls den `endless_process.sh`-Prozess mit einem `kill`-Befehl.

Hinweis: Obwohl der Prozess `endless_process.sh` über den *Umweg* `nohup` gestartet wurde, erscheint er in der Prozessliste unter dem Namen `endless_process.sh`.

Falls Sie einmal vergessen haben, einen langfristigen Prozess mit `nohup` zu starten, können Sie ihn auch nachträglich noch *von der Shell entkoppeln* und vor einer automatischen Beendigung beim Schließen der Shell schützen. Hierzu dient das Kommando `disown`. Erkundigen Sie sich bei Gelegenheit darüber.

Das Allerwichtigste, was Sie aus diesem Abschnitt momentan mitnehmen sollten, ist, dass man Prozesse auch so starten kann, dass sie nach einem Logout im Hintergrund auf dem System weiterlaufen. Eine komfortablere und deutlich flexiblere Art für diesen Aufgabenkomplex ist z. B. das Programm `screen`. Erkundigen Sie sich bei Interesse im Internet darüber.

Aufgabe 9.1

Sie starten aus Versehen den Editor `vi`:

```
user$ vi
```

Wahrscheinlich wissen Sie nicht, wie Sie den Editor wieder verlassen können. Finden Sie die `PID` Ihres `vi` Prozesses und beenden Sie ihn mit einem `kill`-Kommando.

Lösungsvorschlag auf Seite 187!

Aufgabe 9.2

Starten Sie erneut `endless_process.sh` über `nohup`:

```
user$ nohup ./endless_process.sh &
nohup: ignoring input and appending output to \
'nohup.out'
user$
```

Loggen Sie sich komplett aus dem System aus und nach ein paar Minuten wieder ein. Überzeugen Sie sich davon, dass Ihr `endless_process.sh`-Prozess noch läuft, und beenden Sie ihn. Schauen Sie sich danach die erzeugte Datei `nohup.out` an.

Hinweis: Schauen Sie für eine geeignete Option zum Auffinden des `endless_process.sh`-Prozesses in die Dokumentation von `ps`.

Lösungsvorschlag auf Seite 188!

Shell-Konfiguration 10

Übersicht

Zusammenfassung

In diesem Kapitel sehen wir uns an, wie wir unsere Shell individuell an eigene Bedürfnisse anpassen können. Dies betrifft z. B. das in Kap. 9 diskutierte Problem, neue Verzeichnisse in den *Unix*-Suchpfad für ausführbare Programme zu integrieren. Andere, häufig genutzte Konfigurationsmöglichkeiten sind das Aussehen des Eingabe-Prompts oder die Definition von Kurzformen für lange, häufig benutzte, *Unix*-Befehle (die Definition sogenannter *Aliasse*).

Bevor wir zum eigentlichen Thema, der Shell-Konfiguration, kommen, müssen wir zwei neue Konzepte einführen: die *Shell-Variablen* und wie wir *Unix*-Befehle, die in einer Textdatei stehen, ausführen können.

10.1 Shell-Variablen

Eine sogenannte Variable der Shell (oder einer anderen Programmiersprache) ist eine *Zeichenkette*, die einen *Wert* repräsentiert, z. B.

```
user$ NAME=Thomas
```

Hier definiere ich eine neue Variable `NAME` und weise ihr den *Wert* `Thomas` zu. Um den aktuellen Wert einer Variablen zu erfragen, benutzen wir das Konstrukt `${VARIABLE}`:

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_10

```
user$ echo ${NAME}
Thomas
```

Hinweis: In vielen Quellen werden Sie sehen, dass der Variablenzugriff dort *ohne* die geschweiften Klammern erfolgt:

```
user$ echo $NAME
Thomas
```

Dies ist in den *allermeisten* Situationen vollkommen gleichwertig zur Variante mit den Klammern. Es gibt jedoch einige Fälle, in denen sie zwingend nötig sind. Hierauf möchte ich nicht weiter eingehen, und ich verwende in diesem Buch für die Variablenauswertung stets die Variante *mit* den geschweiften Klammern.

Man beachte, dass bei einer Variablendefinition sowohl zwischen dem Variablennamen, dem Gleichheitszeichen und dem Variablenwert *keine* Leerzeichen stehen dürfen:

```
user$ NAME=Thomas
user$ echo ${NAME}
Thomas
    # alles o.k.
user$ NAME =Thomas
NAME: command not found
    # NAME wird als Befehl
    # interpretiert!
user$ NAME= Thomas
Thomas: command not found
    # NAME erhält eine 'leere' Zeichenkette
    # als Variablenwert und 'Thomas' wird
    # als Befehl interpretiert.
```

Was Sie momentan sonst noch über Variablen wissen sollten, fasse ich in einer Auflistung zusammen:

- Eine *Unix*-Shell hat viele Variablen bereits vordefiniert. Sie werden dort benutzt, um wichtige Konfigurationsinformationen zu speichern, die bei Bedarf ausgelesen werden können. Eine Übersicht über momentan in Ihrer Shell definierte Variablen (und noch einiges mehr) gibt der Befehl `set`:

  ```
  user$ set
  BASH=/bin/bash
  .
  COLORTERM=Terminal
  COLUMNS=80
  .
  .
  ```

  Als Beispiel sieht *Unix* jedes Mal, wenn Sie den Befehl `cd ~` geben, in der Variable `HOME` nach, welches Ihr Heimatverzeichnis ist:

```
user$ echo ${HOME}
/home/thomas
```

Wegen ihrer zentralen Rolle für die Shell-Konfiguration müssen wir uns in diesem Kapitel mit den Shell-Variablen beschäftigen.

- Sie können als Benutzer beliebige Variablen definieren und ihnen Werte zuweisen. Sie können auch bereits bestehende Variablen nach Belieben verändern. Sobald Sie *Unix*-Konfigurationsvariablen ändern, sollten Sie aber wissen, was Sie tun:

```
user$ HOME=/export/amac172_1/
user$ cd ~
user$ pwd
/export/amac172_1
```

- Neu definierte Variablen sind nur in der Shell, in der sie definiert werden, gültig! Geben Sie z. B. obigen Befehl `echo ${NAME}` in einer anderen Shell, so werden Sie sehen, dass die Variable dort undefiniert ist.
- Man definiert Variablen, die zu Konfigurationszwecken dienen, mit dem `export`-Kommando:

```
user$ export NAME=Thomas
```

Ohne diesen Zusatz steht die Variable *nur* der Shell, *nicht* aber aufgerufenen Programmen zur Verfügung. Das Kommando `export` macht also Variablen *global* für alle Kindprozesse, die von der Shell initiiert werden, sichtbar. Für Konfigurationsvariablen wollen Sie das! Definieren Sie der Einfachheit halber alle Variablen, die mit System- oder Shell-Konfiguration zu tun haben, mit `export` und kümmern Sie sich nicht weiter darum. Man nennt Variablen, die mit `export` definiert wurden, auch *Umgebungsvariablen*.

- Sollen in einer Variablen Werte mit Shell-Sonderzeichen definiert werden, so muss, wie üblich, geeignet maskiert werden:

```
user$ NAME=Thomas Erben
user$ echo ${NAME}
Thomas
user$ NAME="Thomas Erben"
user$ echo ${NAME}
Thomas Erben
```

- Es hat sich eingebürgert, für Shell-Variablennamen nur Großbuchstaben, Ziffern und den *Unterstrich* „_“ zu verwenden; formal erlaubt sind aber auch Kleinbuchstaben.

Die für Sie im Moment wichtigste Shell-Variable ist sicherlich `PATH`. Sie gibt Auskunft über die in Abschn. 9.1 besprochenen Suchpfade für Programme:

```
user$ echo ${PATH}
/soft/anaconda/bin:/usr/local/sbin:\
/usr/local/bin:/usr/sbin:/usr/bin
```

Die Variable listet durch „:“ getrennte Pfade, in denen *Unix* nach einem Programm sucht, das ohne expliziten Pfad aufgerufen wird. Wichtig ist, dass hierbei die Pfade nach der Reihenfolge, wie sie in `PATH` definiert sind, durchgegangen werden und *das erste* Programm, das gefunden wird, aufgerufen wird. Dies ist dann relevant, wenn es aus irgendeinem Grund mehrere Programme mit demselben Namen in den Verzeichnissen des Suchpfades geben sollte! Obwohl dies auf den ersten Blick nicht wünschenswert erscheint, kommt es hin und wieder vor. Vor allem, wenn man *mehrere Versionen* eines Programms installiert hat:

```
user$ which python
/soft/anaconda/bin/python
user$ which -a python
/soft/anaconda/bin/python
/usr/bin/python
```

Erklärung: Wie bereits in Abschn. 9.1 diskutiert, zeigt der Befehl `which`, wo sich ein bestimmtes Programm im Suchpfad befindet. Hier wird bei einem Aufruf von `python` das Programm unter `/soft/anaconda/bin/python` aufgerufen. Der Befehl `which -a` zeigt *alle* Programme mit Namen `python`, die sich im Suchpfad befinden – in der Reihenfolge ihres Auftretens. Hier tauchen zwei Versionen des `python`-Interpreters auf, da ich, neben der System-Standardinstallation, eine weitere Version dieses Programms auf meinem Rechner installiert habe. Denken Sie an diese Möglichkeit, falls Ihnen ein Programmaufruf unerwartete Ergebnisse oder unerwartete Versionsnummern liefert!

Um an die bestehende `PATH`-Variable einen eigenen Pfad anzuhängen, benutzt man etwas wie:

```
user$ echo ${PATH}
/soft/anaconda/bin:/usr/local/sbin:\
/usr/local/bin:/usr/sbin:/usr/bin
user$ export PATH=${PATH}:/home/thomas/bin
user$ echo ${PATH}
/soft/anaconda/bin: ... :/usr/bin:/home/thomas/bin
```

Erklärung: Der Befehl `export PATH=${PATH}:/home/thomas/bin` gibt der Variablen `PATH` den ursprünglichen Wert `${PATH}` und hängt daran `:/home/thomas/bin` an.

Hiermit kann ich von jetzt an auch Programme in `/home/thomas/bin` ohne Pfadangabe aufrufen:

```
user$ export PATH=${PATH}:/home/thomas/bin
user$ pwd
/home/thomas/vorlesung
user$ endless_process.sh
endless_process.sh: command not found
user$ mv endless_process.sh /home/thomas/bin
user$ which endless_process.sh
```

```
/home/thomas/bin/endless_process.sh
user$ endless_process.sh
Die Variable 'i' hat jetzt den Wert 10
Die Variable 'i' hat jetzt den Wert 11
.
.
```

Man beachte, dass wir neue Pfade an die Variable `PATH` *angehängt* haben. Es ist auch möglich, neue Pfade *voranzustellen*:

```
# Bei PATH-Modifikationen, die neue Pfade
# voranstellen, sollte man sehr vorsichtig sein.
user$ export PATH=/home/thomas/bin:${PATH}
```

Falls Sie keinen *guten* Grund haben, dies zu tun, sollten Sie davon Abstand nehmen! Wie oben diskutiert, wird immer *das erste* Programm mit einem bestimmten Namen aufgerufen, welches im Suchpfad gefunden wird. Das kann zum Problem werden, wenn das neue Verzeichnis Programme mit Namen enthält, die es bereits gibt und Sie sich dessen *nicht bewusst* sind! Manchmal kommt es aber vor, dass Sie bereits existierende Programme mit eigenen Versionen ersetzen wollen. In dem Beispiel von oben:

```
user$ which -a python
/soft/anaconda/bin/python
/usr/bin/python
```

habe ich *absichtlich* den systemweit installierten `python`-Interpreter unter `/usr/bin/python` durch eine eigene Version ersetzt. Um dies zu erreichen, habe ich den Pfad `/soft/anaconda/bin/` meiner existierenden `PATH`-Variablen vorangestellt. Mitzunehmen ist hier: Neue Pfade sind grundsätzlich an die bestehende `PATH`-Variable anzuhängen. Voranstellen sollten Sie neue Pfade nur, *falls absolut nötig* und wenn Sie genau wissen, was Sie tun!

Um eine Änderung der `PATH`-Variablen dauerhaft zu speichern, wird sie in einer sogenannten *Konfigurationsdatei* abgelegt. Konfigurationsdateien sind in erster Linie Textdateien, die abzuarbeitende *Unix*-Befehle enthalten. Wie *Unix* solche Dateien verarbeitet, sehen wir uns in den folgenden Abschnitten an.

Aufgabe 10.1

Manche Leute führen folgende Modifikation ihrer `PATH`-Variablen durch:

```
echo$ export PATH=${PATH}:.
   # Es wird ein '.' an PATH angehängt.
```

Welche Folgen hat diese Änderung der Variablen?
Lösungsvorschlag auf Seite 189!

10.2 *Unix*-Befehle aus einer Datei ausführen

Wir wollen eine Liste von *Unix*-Kommandos in eine Datei schreiben und diese möglichst einfach ausführen. Dies ist besonders dann interessant, wenn dieselbe Befehlssequenz oft ausgeführt werden soll. Zur Demonstration habe ich eine Datei `befehle.sh` mit folgendem Inhalt:

```
user$ cat befehle.sh
NAME="Thomas"
echo "Variable NAME: ${NAME}"
echo "Verzeichnis vor cd:"
pwd
cd ~/buch
echo "Verzeichnis nach cd:"
pwd
```

Der Befehl `cat` (*concatenate*) gibt in der hier gezeigten Form einfach den Inhalt einer Textdatei (hier `befehle.sh`) auf dem Bildschirm aus. Jede Zeile der Datei enthält einen *Unix*-Befehl, und es sollte klar sein, was die schrittweise Abarbeitung dieser Befehle bewirkt. Um solche Dateien mit *Unix*-Befehlen automatisch von der *Bash* abarbeiten zu lassen, gibt es zwei grundsätzliche Möglichkeiten. Der für uns hier wichtige Unterschied beider Methoden ist, *ob* die ausgeführten Befehle die gegenwärtige *Shell-Umgebung* (die in einer Shell definierten Variablen) beeinflussen oder nicht.

10.2.1 Befehlsabarbeitung ohne Einfluss auf die gegenwärtige Shell-Umgebung

Wir können Shell-Befehle in einer Datei von der *Bash* abarbeiten lassen, indem wir die Datei als Argument dem Befehl `bash` übergeben:

```
user$ pwd
/home/thomas                # Wir befinden uns in HOME
user$ echo ${NAME}          # Eine Variable NAME ist
                            # nicht definiert!

user$ bash befehle.sh       # Hier wird die Datei in
                            # einem neuen Prozess
                            # abgearbeitet
Variable NAME: Thomas       # Befehlsabarbeitung
Verzeichnis vor cd:         # innerhalb des bash-
/home/thomas                # Kommandos
Verzeichnis nach cd:
/home/thomas/buch
```

```
user$ echo ${NAME}          # Die in der Datei
                            # definierte Variable ist
                            # hier ‚nicht' gesetzt!

user$ pwd
/home/thomas                # Wir befinden uns nach wie
                            # vor in HOME!
```

Erklärung: Hier wird durch `bash befehle.sh` ein *neuer* `bash`-Prozess gestartet, innerhalb dessen die Befehle in der Datei abgearbeitet werden. Die `echo`- und `pwd`-Befehle zeigen klar, dass innerhalb dieses neuen Prozesses eine Umgebungsvariable `NAME` angelegt und das Verzeichnis gewechselt wird. Nach Beendigung des `bash`-Befehls haben sich diese Änderungen allerdings nicht auf die gegenwärtige Sitzung ausgewirkt! Die Variable `NAME` ist hier nach wie vor undefiniert, und das Verzeichnis ist dasselbe, aus dem der Befehl `bash befehle.sh` ursprünglich gestartet wurde.

Mitzunehmen ist hier: Kindprozesse beeinflussen *niemals* die Umgebung ihres Elternprozesses! In vorliegendem Fall wird aus der gegenwärtigen Shell-Sitzung der Kindprozess `bash befehle.sh` gestartet, innerhalb dessen die Befehle in der Datei `befehle.sh` abgearbeitet werden.

10.2.2 Befehlsabarbeitung mit Einfluss auf die gegenwärtige Shell-Umgebung

Neben der Abarbeitung einer Shell-Befehlsdatei mit einem `bash`-Befehl kann man eine solche Datei auch *innerhalb* des laufenden Shell-Prozesses, *ohne* einen Kindprozess zu starten, aufrufen. Dies nennt man auch *Eine Datei in der Shell sourcen*:

```
user$ pwd
/home/thomas                # Wir befinden uns in HOME
user$ echo ${NAME}          # Eine Variable NAME ist
                            # nicht definiert!

user$ source befehle.sh     # Hier wird die Datei mit
                            # einem source-Befehl
                            # ‚innerhalb' des laufenden
                            # Shell-Prozesses
                            # abgearbeitet
Variable NAME: Thomas       # Befehlsabarbeitung
Verzeichnis vor cd:         # innerhalb des source-
/home/thomas                # Kommandos
Verzeichnis nach cd:
/home/thomas/buch
```

```
user$ echo ${NAME}
Thomas
user$ pwd
/home/thomas/buch
```

Erklärung: Das *Sourcen* einer Datei ist effektiv dasselbe, wie die in der Datei enthaltenen Befehle nacheinander auf der Kommandozeile einzutippen. Darin enthaltene Variablendefinitionen und Verzeichniswechsel sind daher nach Beendigung des `source`-Befehls nach wie vor gültig!

Der `source`-Befehl kann durch einen einfachen Punkt abgekürzt werden:

```
user$ . befehle.sh     # vollkommen identisch zu
                       # 'source befehle.sh'!
Variable NAME: Thomas
.
.
user$
```

Neben den zwei bisher bekannten Sonderstellungen des Punktes in der Shell (Repräsentant des *cwd*, Repräsentant für versteckte Dateien/Verzeichnisse, wenn er das erste Zeichen des Namens ist), ist er also auch noch ein Shell-Kommando!

Wahrscheinlich haben Sie bereits das Potenzial der hier vorgestellten Techniken zum Ausführen von *Unix*-Befehlen aus Dateien erkannt. Man kann effektiv mehrere *Unix*-Befehle, die immer wieder ausgeführt werden sollen, zu einem *Programm* zusammenfassen. In der Tat sind sogenannte *Shell-Skripte* primär solche Auflistungen von *Unix*-Befehlen. Wir werden dies in Kap. 15 näher beleuchten. Shell-Skripte sollen die Umgebung der aufrufenden Shell nicht beeinflussen. Daher wird für sie der Aufruf über das `bash`-Kommando genutzt. Das Sourcen von Kommandodateien spielt eine wichtige Rolle bei der Shell-Konfiguration. Das wollen wir uns jetzt ansehen.

10.3 Die Shell-Konfigurationsdateien `.bash_profile` und `.bashrc`

Am Ende von Abschn. 10.1 haben wir gesehen, wie wir den *Unix*-Programmsuchpfad mithilfe der `PATH`-Variablen modifizieren können. Wir haben noch das Problem, dass die Pfaderweiterung *nur* in der gegenwärtigen Shell gültig ist. Wir müssten die Variablendefinition also in jeder neu geöffneten Shell wieder eingeben. Zur Lösung dieses Problems gibt es sogenannte *Konfigurationsdateien*. Dies sind in der Regel *versteckte Dateien* (siehe Abschn. 5.4) in Ihrem Heimatverzeichnis. Der Inhalt bestimmter Konfigurationsdateien wird von *Unix* jedes Mal *automatisch* abgearbeitet, sobald eine neue Shell-Instanz geöffnet wird. Sie sind also der ideale Platz, um benutzerspezifische Konfigurationen, wie z. B. eine Erweiterung der `PATH`-Variablen, dauerhaft zu speichern. Die für uns im Moment wichtigsten

Konfigurationsdateien sind ~/.bash_profile und ~/.bashrc. Ich möchte hier nicht zu sehr auf die Details eingehen, warum es *mehrere* Konfigurationsdateien gibt und welche internen Mechanismen mit diesen Dateien ablaufen. Leider gibt es aus historischen Gründen noch *deutlich* mehr als die eben erwähnten Konfigurationsdateien und diese werden teilweise auch in modernen *Unix*-Installationen benutzt.

Exkurs

Fallstricke bei der Shell-Konfiguration

Leider ist es sehr schwer, ein einfaches Rezept für eine Shell-Konfiguration zu geben, welches für *jeden* von Ihnen gut und vernünftig ist. Dies hat hauptsächlich zwei Gründe: (1) Abhängig von Ihrem installierten System haben Sie schon eine mehr oder weniger umfangreiche Konfiguration. Hier wollen wir möglichst wenig *verkonfigurieren* und für Sie zum Schlechteren verändern; (2) es gibt aus historischen Gründen verschiedene *Kombinationen* von Konfigurationsdateien, die im Prinzip dasselbe tun. Werden jedoch verschiedene Möglichkeiten vermischt, so hat dies oft unerwünschte Folgen. Unsere *Bash*-Shell (*Bash* steht für *Bourne Again Shell*) ist ein Nachfolger der sogenannten *Bourne-Shell*. Letztere unterstützte eine Konfigurationsdatei namens .profile. Für die *Bash* wurde die neue Konfigurationsdatei .bash_profile (und weitere!) eingeführt. Man hat sich dazu entschlossen, auch noch die ältere .profile-Konvention der *Bourne*-Shell in der *Bash* zu unterstützen. Es ist eine der Stellen, an denen man sich für *eine* Möglichkeit entscheiden sollte. Ein Vorhandensein von .profile und .bash_profile führt meist zu unvollständiger oder fehlerhafter Konfiguration. Leider hat sich unter den verschiedenen *Unix*-Systemen bis heute keine *Standardkombination* etabliert, und verschiedene *Unix*-Systeme und *Linux*-Distributionen benutzen verschiedene Kombinationen von Konfigurationsdateien. Des Weiteren ist eine *optimale* Wahl von Ihrem speziellen *Unix*-System abhängig. Dies sind die Hauptgründe für die Probleme, ein einfaches Rezept für eine optimale Shell-Konfiguration bereitzustellen.

Ich gebe im Folgenden einige praktische Empfehlungen, wie Sie, in Abhängigkeit von einer bestehenden Konfiguration, fortfahren können. *Unix*-Experten werden sie sicher nicht für *optimal* halten. Mein Hauptaugenmerk sind *möglichst einfache* Prozeduren, die in der Praxis sehr gut funktionieren:

1. Sie haben in Ihrem Heimatverzeichnis eine der Kombinationen .profile und .bashrc *oder* .bash_profile und .bashrc vorliegen: Dies ist die wahrscheinlichste Variante auf einem *Linux*-System. Fügen Sie *neue* Konfigurationen der Datei .bashrc hinzu, während Sie bestehende Konfigurationen in der Datei modifizieren, in der sie bereits stehen. Es ist z. B. sehr wahrscheinlich, dass die PATH-Variable bereits in einer der Dateien definiert ist.

2. Sie haben eine der Dateien `.profile` *oder* `.bash_profile`, aber *keine* `.bashrc`-Datei in Ihrem Heimatverzeichnis: Diese Möglichkeit ist auf einem *Apple Macintosh*-Computer sehr wahrscheinlich. Führen Sie hier alle Modifikationen in der jeweilig bestehenden Datei durch.
3. Sie finden *keine* der Dateien `.profile`, `.bash_profile` und `.bashrc` in Ihrem Heimatverzeichnis: Dies ist z. B. auf neuen Benutzer-Accounts an Universitäten nicht unüblich. Hier werden alle Standardkonfigurationen zunächst aus systemweiten Dateien gelesen. In diesem Fall empfehle ich Ihnen das Anlegen einer leeren `~/.bashrc`-Datei:

   ```
   user$ touch ~/.bashrc
   ```

   Danach erstellen Sie eine `~/.bash_profile`-Datei mit folgendem Inhalt:

   ```
   user$ cat ~/.bash_profile
   source ~/.bashrc
   ```

   Alle folgenden Konfigurationen werden in der Datei `~/.bashrc` durchgeführt.

   Erklärung: Welche der Dateien `~/.bashrc` oder `~/.bash_profile` beim Öffnen einer Shell abgearbeitet wird, hängt, vereinfacht gesagt, davon ab, *wie* die Shell geöffnet wird. Ein Öffnen innerhalb eines neuen Terminalfensters oder das Starten einer Shell durch das Einloggen auf einem entfernten Rechner (`ssh`-Befehl) sind z. B. verschiedene Möglichkeiten, eine neue Shell zu öffnen. Die durchzuführenden Konfigurationen sind aber im Prinzip dieselben. Daher speichern wir alle Konfigurationen in der Datei `~/.bashrc`. Falls das Öffnen einer Shell die Datei `~/.bash_profile` abarbeiten möchte, so wird hier durch einen `source`-Befehl der Inhalt der Datei `~/.bashrc` geladen. Effektiv werden also alle Benutzerkonfigurationen mit dem Inhalt der Datei `~/.bashrc` durchgeführt.

Andere Möglichkeiten sind auf modernen *Unix*-Systemen in neu eingerichteten Benutzer-Accounts sehr unwahrscheinlich. Man kann aber nicht viel falsch machen, wenn man seine Konfigurationen in einer bestehenden, oder neu angelegten, `~/.bashrc`-Datei durchführt.

Auch wenn Sie sich auf Ihrem System für *eine* Möglichkeit bei den `~/.profile`- und `~/.bash_profile`-Dateien entschieden haben, müssen Sie aufpassen, dass Sie nicht irgendwann *beide* Dateien haben. Manche Programme wollen z. B. bei Installation ein neues Verzeichnis zur `PATH`-Variablen hinzufügen und legen unter Umständen eine der Konfigurationsdateien an, falls sie noch nicht existiert. In der Regel wird vor solchen Änderungen aber nachgefragt. Ich empfehle Ihnen, solche Modifikation *immer* per Hand zu erledigen, anstatt dies automatisch durchführen zu lassen!

Aufgabe 10.2

a) Finden Sie heraus, welche Konfigurationsdateien in Ihrem Heimatverzeichnis vorhanden sind und wie Sie auf Ihrem System Benutzerkonfigurationen durchführen können.
b) Legen Sie ein Verzeichnis an, in dem Sie zukünftig eigene Programme ablegen wollen. Üblich ist etwas wie `~/bin`.
c) Modifizieren Sie die Systemkonfiguration, indem Sie das in b) erstellte Verzeichnis als Programmpfad an die `PATH`-Variable anhängen.
d) Öffnen Sie eine neue Shell. Verifizieren Sie, dass die Umgebungsvariable `PATH` den neuen Pfad enthält.
e) Kopieren Sie das Programm `countdown_process.sh` in das unter b) angelegte Verzeichnis und führen Sie es ohne Angabe eines absoluten Pfades aus.

10.4 Nützliche Konfigurationen

Zum Abschluss dieses Kapitels möchte ich noch einige nützliche Konfigurationen besprechen. In der Datei `beispiele_bashrc` zu diesem Kapitel finden Sie erweiterte und kommentierte Konfigurationen, die Sie gegebenenfalls direkt in Ihre eigenen Konfigurationsdateien übernehmen können.

10.4.1 Wichtige Umgebungsvariablen

Sie werden mit der Zeit sehen, dass das Verhalten vieler Programme durch Umgebungsvariablen beeinflusst werden kann. Diese sollten Sie entsprechend in Ihren Konfigurationsdateien setzen. Überprüfen Sie aber *vor* der Neudefinition einer Variablen, ob diese in einer bestehenden Konfiguration nicht schon vorhanden ist. Im Folgenden bespreche ich einige Variablen, um die Sie sich möglichst bald kümmern sollten:

1. Viele Terminalprogramme erlauben den internen Aufruf eines Texteditors. In einer Shell können Sie z. B. jederzeit mit der Tastenkombination `<C-x-e>` einen Editor aufrufen. Der Standard ist hier meist der `vi`, welchen die meisten Anfänger nicht nutzen möchten. Um einen vom Benutzer bevorzugten Editor zu verwenden, lesen Programme eine der Umgebungsvariablen `EDITOR` oder `VISUAL` aus. Setzen Sie diese Variablen auf den *absoluten Pfad* Ihres Wunscheditors:

```
user$ which nano
/usr/bin/nano    # der absolute Pfad zu nano
user$ less ~/.bashrc
.
export EDITOR=/usr/bin/nano # EDITOR und VISUAL auf
export VISUAL=/usr/bin/nano # den abs. Pfad setzen!
.
```

2. Ähnlich zu einem Editor verwenden verschiedene Programme einen Pager, so z. B. das *Unix*-Kommando `man` (siehe Abschn. 4.2). Hier empfehle ich Ihnen, sicherzustellen, dass *immer* der Pager `less` verwendet wird. Dies können wir durch Setzen der Variablen `PAGER` erreichen:

   ```
   user$ which less
   /usr/bin/less    # der absolute Pfad zu less
   user$ less ~/.bashrc
   .
   export PAGER=/usr/bin/less
   .
   ```

3. Sie sollten eventuell den Befehls-Prompt an Ihre Bedürfnisse anpassen. Die wesentliche Variable ist hier `PS1`, und sie bietet umfangreiche Konfigurationsmöglichkeiten. Der *Minimal-Prompt*, den ich hier im Buch zu Demonstrationszwecken verwende, kann mit:

   ```
   user$ export PS1="user\$␣"
   user$
   ```

   realisiert werden. Meine reale `PS1`-Variable und der Prompt sehen folgendermaßen aus:

   ```
   user$ export PS1="\u@\h:\w\$␣"
   thomas@amac172:~$ cd test
   thomas@amac172:~/test$ cd test
   ```

   In meinem Prompt ist der Benutzername (thomas), der Maschinenname (amac172) und das *cwd* enthalten. Wie das Aussehen des Prompts und die Konfigurationszeichenkette `\u@\h:\w\$␣` in Beziehung stehen, ist aus Tab. 10.1 ersichtlich. Ich empfehle Ihnen auf alle Fälle, das gegenwärtige Verzeichnis in Ihren Prompt mit aufzunehmen. Wer möchte, kann hier noch weitergehen und z. B. bestimmte Promptelemente farbig gestalten. Hierüber gibt das Internet Auskunft.

Tab. 10.1 Steuerzeichen zur Konfiguration der `PS1` Prompt-Umgebungsvariablen

Steuerzeichen	Bedeutung
`\d`	Datum in der Form `Sun Dec 24`
`\H`	Kompletter Rechnername (z. B. `rechner.beispiel.de`)
`\h`	Rechnername bis zum ersten Punkt (z. B. `rechner` falls der komplette Rechnername `rechner.beispiel.de` ist)
`\t`	Zeit in der Form `hh:mm:ss`
`\u`	Benutzername
`\w`	Arbeitsverzeichnis
`\W`	Arbeitsverzeichnis (nur Teil nach dem letzten `/`)
`\$`	$-Zeichen für normalen Benutzer, # für root
`\#`	Nummer des aktuellen Kommandos (in der aktuellen Sitzung)
`\!`	History-Nummer des aktuellen Kommandos (über Sitzungen hinweg)

Wichtig

Probleme durch unterschiedliche Systemkonfigurationen

Hin und wieder hat man mit *vollkommen unverständlichen* Fehlern der Art „Warum geben *genau dieselben* Programmaufrufe auf *ein und demselben Rechner* für verschiedene Benutzer unterschiedliche Ergebnisse?“ zu kämpfen.

Oft sind unterschiedliche Systemkonfigurationen und/oder unterschiedlich definierte Umgebungsvariablen der Grund solcher Probleme. Vergleichen Sie daher betreffende Einstellungen, bevor Sie das Problem bei den Programmen selbst suchen. Für eine Liste *aller*, in einer Shell gesetzten, Variablen steht das Kommando `set` zur Verfügung. Möchten Sie sich zunächst auf einen Vergleich der Umgebungsvariablen beschränken, so können Sie den Befehl `env` zur Auflistung dieser benutzen.

Lesen Sie zu diesem Komplex auch Anhang C, falls Sie mit einer deutschen *Unix*-Variante arbeiten.

10.4.2 Alias-Definitionen

In Kap. 7 haben wir etliche Techniken kennengelernt, wie Sie sich viel Tipparbeit auf der Shell ersparen können. Thematisch gehören die sogenannten *Aliasse* dorthin. Da man sie aber nur in Verbindung mit einer Shell-Konfiguration sinnvoll einsetzen kann, beschäftigen wir uns erst hier mit ihnen.

Bestimmte Befehle tippt man sehr oft mit gewissen Optionen, z. B. den Befehl `ls -ltr`, um sich ein ausführliches, nach Erstellungsdatum sortiertes, Listing aller Dateien und Verzeichnisse im *cwd* zu besorgen. Mit Aliassen können wir Kurzformen für solche Fälle definieren. Die folgenden Beispiele sollten die grundlegende Benutzung des `alias`-Kommandos klarmachen:

```
user$ alias ll="ls -ltr" # Der Befehl 'll' ist jetzt
                         # eine Kurzform für 'ls -ltr'.
user$ ll
-rw-r--r-- 1 thomas users    188 Feb  1 16:25 test.txt
-rw-r--r-- 1 thomas users   1741 Feb  1 16:25 test1.txt
.
user$ alias h="history"  # Der Befehl 'h' ist jetzt
                         # eine Kurzform für 'history'.
user$ h
.
  563  alias ll="ls -ltr"
  564  ll
  565  h
user$ alias rm="rm -i"     # Der Befehl 'rm' wird zu
                          # 'rm -i' umdefiniert!
user$ rm test.txt
rm: remove regular empty file 'test.txt'?
```

Man achte besonders auf das letzte Beispiel, bei dem ein *bestehendes* Kommando mit `alias` *umdefiniert* wird. Der Befehl `alias` (ohne Argumente) gibt eine Liste definierter Aliasse, und mit dem Befehl `unalias` kann ein Alias wieder gelöscht werden:

```
user$ alias
alias ll='ls -ltr'
alias rm='rm -i'
user$ unalias rm
user$ alias
alias ll='ls -ltr'
```

Aliasse sind in manchen Situationen auch sehr nützlich, um *mehrere* Befehle auszuführen. Bisher haben wir mit *Unix* auf der Kommandozeile jeweils einen einzigen Befehl gegeben. Man kann auch mehrere, durch Strichpunkt getrennte, Befehle auf eine Zeile schreiben. Diese werden dann nacheinander abgearbeitet:

```
user$ echo "Ein wenig Text"
Ein wenig Text
user$ echo "Etwas mehr Text"
Etwas mehr Text
user$ echo "Ein wenig Text"; echo "Ein wenig mehr Text"
   # Zwei echo-Befehle auf einer Kommandozeile!
Ein wenig Text
Ein wenig mehr Text
```

Mit dieser Technik lässt sich beispielsweise folgender, sehr nützlicher Alias definieren:

```
user$ alias eb="nano ~/.bashrc; source ~/.bashrc"
   # eb editiert erst ~/.bashrc mit nano
   # und sourced die Datei danach in der gerade
   # laufenden Shell.
```

Erklärung: Wenn wir eine Konfigurationsdatei ändern, so sind neue Definitionen erst *nach* dem Öffnen einer neuen Shell in dieser wirksam. Wollen wir die Definitionen auch in der laufenden Shell nutzen, so müssen wir die modifizierte Konfigurationsdatei explizit sourcen. Das oben definierte Alias `eb` (*edit bashrc*) ruft erst den Editor `nano` zur Modifikation der Datei `~/.bashrc` auf. Nach Schließen des Editors wird die Datei gesourced und vorgenommene Änderungen stehen sofort in der gegenwärtigen Shell zur Verfügung.

Genau wie Shell-Variablen sind Alias-Definitionen nur in der Shell gültig, in der sie definiert werden. Um sie sinnvoll und dauerhaft nutzen zu können, müssen sie in die Shell-Konfiguration eingebunden werden. In der Datei `beispiele_bashrc` liste ich einige nützliche und gängige Alias-Definitionen. Bevor Sie neue Aliasse definieren, sollten Sie mit dem `alias`-Befehl überprüfen, welche bereits auf Ihrem Benutzer-Account vordefiniert sind.

Hinweis: Falls Sie eine vorinstallierte Systemkonfiguration haben, schauen Sie nach Dateien mit Namen wie `~/.alias` oder `~/.bash_alias`. Manche Systeme lagern Alias-Definitionen in eine eigene Datei aus und *sourcen* diese Alias-Datei innerhalb der Konfiguration.

Dateien finden mit dem `find`-Kommando 11

Zusammenfassung

Dateien mit bestimmten Eigenschaften im *Unix*-Verzeichnisbaum zu finden und später Aktionen mit ihnen oder an ihnen durchzuführen, ist eine zentrale Aufgabe in der täglichen Arbeit mit *Unix*. Das wichtigste Werkzeug zum Auffinden von Dateien ist das `find`-Programm, welches wir in diesem Kapitel kennenlernen werden.

In den verbleibenden Kapiteln dieses Buches werden wir hauptsächlich *Unix*-Werkzeuge zur Bearbeitung von Textdateien kennenlernen. Als einziges Programm, das nicht mit der direkten Bearbeitung von Textdateien zu tun hat, möchte ich Ihnen `find` *(Dateien finden)* detaillierter vorstellen. Es ist meines Erachtens das Wichtigste dieser Kategorie, und Sie werden sicher sehr schnell Gebrauch davon machen. Das `find`-Kommando erlaubt es Ihnen, den *Unix*-Verzeichnisbaum systematisch nach Dateien und/oder Verzeichnissen zu durchforsten. Es besitzt eine *sehr große* Anzahl an Optionen, von denen ich im Folgenden einige anhand von Beispielen erwähne. In der einfachsten Form lautet seine Syntax `find VERZEICHNIS -name MUSTER`. Dieser Befehl durchsucht, startend von `VERZEICHNIS`, *rekursiv* den gesamten *Unix*-Verzeichnisbaum nach Dateien und Verzeichnissen, auf die `MUSTER` zutrifft. Hier einige konkrete Beispiele, um die Vielfalt dieses Befehls *anzureißen*:

```
user$ find /home/thomas -name test.txt
```

Dies würde eine Datei oder ein Verzeichnis mit Namen `test.txt` in meinem Heimatverzeichnis und in *allen darunter liegenden* Verzeichnissen suchen. Falls eine Datei oder ein Verzeichnis mit Namen `test.txt` mehrfach existiert, so werden alle angezeigt.

```
user$ find /home/thomas -name '*.txt'
```


T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_11

Dies würde mir, startend von meinem Heimatverzeichnis, alle Dateien und Verzeichnisse anzeigen, die auf `.txt` enden. Beachten Sie, dass das Argument nach der Option `-name` *gequotet* ist. Dies ist nötig, damit `MUSTER` mit Shell-Sonderzeichen bei `find` ankommen und nicht vorher von der Shell ausgewertet werden (siehe Kap. 8). Das `MUSTER` kann alle in Abschn. 5.5 behandelten Wildcards mit der entsprechenden Bedeutung verwenden.

```
user$ find /home/thomas -type d        # Typ: directory
```

listet rekursiv alle *Verzeichnisse* unterhalb meines Heimatverzeichnisses. Die Option `-type f` (*Typ: Files*) würde dagegen nur Dateien zeigen.

```
user$ find /scratch -user thomas -mtime -3
```

listet alle Dateien und Verzeichnisse unterhalb von `/scratch`, die Benutzer `thomas` (`-user thomas`) gehören und deren Inhalt innerhalb der letzten drei Tage geändert wurde (`-mtime -3`). `-mtime [+- x]` steht für *(modification time)* älter „+“, oder jünger „-“ als `x` Tage. Dieser Befehl ist sehr nützlich, um differenziert Sicherheitskopien bestimmter Dateien und Verzeichnisse anzulegen (siehe auch Aufgabe 13.5)!

Hinweis: Festplatten unter dem Namen `/scratch` werden oft verwendet, um Benutzern temporär zusätzlichen Plattenplatz zur Verfügung zu stellen.

```
user$ find /home/thomas -size +3M
```

Dieser Befehl findet alle Dateien in meinem Heimatverzeichnisbaum, die eine Größe von 3 Megabyte oder mehr haben. Dies ist nützlich, um *Platzfresser* aufzuspüren.

Aufgabe 11.1

Beschreiben Sie, was bei folgenden Befehlen passiert:

```
user$ find /home/thomas -name '*.txt'
```

und

```
user$ find /home/thomas -name *.txt
```

Hinweis: Der zweite Befehl ist *deutlich* gehaltvoller, als die meisten anfangs denken! Eine Lösungsantwort wie „Hier wurde vergessen, die Zeichenkette `*.txt` zu quotieren.“ wird der Aufgabe nicht gerecht. Wenden Sie für einen Start Ihrer Überlegungen den Befehl `find . -name *.txt` *innerhalb des Verzeichnisses* `~/test` auf die unten stehenden drei Verzeichnisstrukturen an und überlegen Sie sich, wie Ihre Ergebnisse zustande kommen.

1. `~/test/test.txt`
   `~/test/test_kind/test.txt`
   `~/test/test_kind/test_1.txt`

2. Hier wird im Vergleich zu oben in ~/test die Textdatei test.txt in test_1.txt umbenannt:

```
~/test/test_1.txt
~/test/test_kind/test.txt
~/test/test_kind/test_1.txt
```

3. Hier sind in ~/test die Dateien test.txt *und* test_1.txt vorhanden:

```
~/test/test.txt
~/test/test_1.txt
~/test/test_kind/test.txt
~/test/test_kind/test_1.txt
```

Lösungsvorschlag auf Seite 190!

Aufgabe 11.2

a) Beschreiben Sie, was bei folgendem Befehl passiert:

```
user$ find /home/thomas -type f -name '*.txt' \
   -exec chmod g+r {} \;
```

b) Viele Programme legen Zwischenprodukte oder Sicherheitskopien an, die man irgendwann wieder loswerden möchte. Zum Beispiel erzeugt der Editor emacs in regelmäßigen Abständen Dateien, die mit einer Tilde enden (z. B. test.txt~) und Sicherheitskopien von gerade bearbeiteten Textdateien sind.
Geben Sie einen find-Befehl der, startend von Ihrem Heimatverzeichnis, den gesamten Verzeichnisbaum nach Dateien, die mit einer Tilde enden, durchsucht und diese löscht.

Lösungsvorschlag auf Seite 191!

Aufgabe 11.3

Geben Sie einen find-Befehl, der rekursiv unter Ihrem Heimatverzeichnis alle Dateien zeigt, die die Benutzerklasse *others* verändern darf. (Es sind alle Dateien zu finden, die für *others* das w-Recht gesetzt haben.) Wahrscheinlich müssen Sie hier etwas ausführlicher in die Hilfen oder das Internet schauen.

Hinweis: Wahrscheinlich haben Sie in Ihrem Verzeichnisbaum *keine* Datei, die obige Bedingung erfüllt. Der Befehl dient hauptsächlich zur Prüfung, ob sich irgendwo solche Dateien eingeschlichen haben, z. B. von einem Internet-Download. In solchen Fällen ist es gut, sich irgendwo in seinem Verzeichnisbaum eine solche Datei anzulegen, um zu testen, ob der Befehl diese findet und somit auch wirklich funktioniert!

Lösungsvorschlag auf Seite 191!

Textdateien unter *Unix* 12

Übersicht

Zusammenfassung

In diesem Kapitel besprechen wir, was bei der Nutzung von Textdateien unter *Unix* zu beachten ist. Des Weiteren schauen wir uns effektive Techniken an, um uns schnell einen Überblick über den Inhalt von Textdateien zu beschaffen. Wir beenden dieses Kapitel mit einer Diskussion über Textdateien zur Speicherung wissenschaftlicher Daten.

12.1 Textdatei ist nicht gleich Textdatei

Sie haben in diesem Buch bereits mehrfach mit Textdateien gearbeitet, z. B. bei der Systemkonfiguration. Man möchte meinen, dass dieses einfache Dateiformat, welches nichts als *Klartext* enthält, *standardisiert* wäre und dass es damit keine besonderen Probleme gäbe. Dem ist leider nicht so!

12.1.1 Textdateien auf verschiedenen Betriebssystemen

Das erste Problem ist, dass verschiedene Betriebssysteme Textdateien teilweise unterschiedlich kodieren und dass man sie nicht ohne Probleme von einem Betriebssystem auf das andere übertragen kann. Textdateien enthalten neben den Klartextzeichen auch *unsichtbare Zeichen*, wie z. B. Zeilenumbrüche. Leider haben sich *Microsoft Windows*, *Unix* und *Apple Macintosh* die Freiheit genommen, die Zeilenenden unterschiedlich zu behandeln. Es kann zu recht schwer zu finden-

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_12

den Fehlern führen, falls Sie beispielsweise eine Textdatei, die Sie auf einem *Microsoft Windows*-System erstellt haben, auf eine *Unix*-Maschine kopieren und dort einfach mit ihr weiterarbeiten. Heute erkennen viele Programme automatisch, wenn eine Textdatei nicht dem Format seines Systems entspricht, und führen intern notwendige Konvertierungen durch. Darauf verlassen sollte man sich aber nicht! Hauptproblem ist hier, dass es meist nicht sofort auffällt, wenn man eine Textdatei im verkehrten Format vor sich hat. Im folgenden Beispiel habe ich die Dateien `ascii_windows.txt` und `ascii_unix.txt` mit *demselben Inhalt*. Der `grep`-Befehl, den wir in Abschn. 14.1 ausführlich besprechen werden, sucht in der hier gezeigten Form nach Zeilen, welche mit einem „z“ enden:

```
# Die Dateien ascii_windows.txt und ascii_unix.txt
# scheinen denselben Inhalt zu haben:
#
user$ cat ascii_windows.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0 1 2 3 4 5 6 7 8 9
user$ cat ascii_unix.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0 1 2 3 4 5 6 7 8 9
# Jetzt suchen wir nach Zeilen, die mit einem
# 'z' enden:
user$ grep 'z$' ascii_unix.txt
abcdefghijklmnopqrstuvwxyz          # alles o.k.!
user$ grep 'z$' ascii_windows.txt   # Ups!
user$
```

Obwohl der `cat`-Befehl hier keinerlei Probleme vermuten lässt, erkennt der letzte `grep`-Befehl leider das unter *Microsoft Windows* kodierte Zeilenende nicht richtig.

Um solche Probleme zu vermeiden, muss man Textdateien nach einem Transfer eventuell *umwandeln*, wenn man sie mit verschiedenen Betriebssystemen bearbeiten und zwischen ihnen übertragen möchte. Für diese Zwecke stehen unter *Unix* Programme wie `dos2unix` oder `fromdos` (Umwandlung von *Microsoft Windows* nach *Unix*), bzw. `unix2dos` oder `todos` (Umwandlung von *Unix* nach *Microsoft Windows*) zur Verfügung:

```
user$ fromdos ascii_windows.txt
       # fromdos überschreibt die alte Datei!
user$ grep 'z$' ascii_windows.txt
abcdefghijklmnopqrstuvwxyz          # alles klar!
```

Im Folgenden schauen wir uns an, wie Sie das Format einer Textdatei erkennen können.

12.1.2 Datentypen unter *Unix* und das `file`-Kommando

Unter *Microsoft Windows* besitzen Dateien typischerweise eine Endung aus drei Buchstaben, um ihren Typ zu identifizieren. Zum Beispiel steht `.pdf` für Dateien des Typs *Portable Document Format* oder `.jpg` für Bilder des *JPEG*-Formats. Diese Kennzeichnung erlaubt es *Microsoft Windows* z. B., Dateien mit bestimmten Anwendungen zu verknüpfen. Dieses sehr sinnvolle Konzept hat sich inzwischen auch in der *Unix*-Welt weitgehend durchgesetzt, was aber nicht immer so war. Um unter *Unix* festzustellen, *was* wirklich in einer Datei steckt, gibt es das Programm `file`. Es analysiert den *Inhalt* einer Datei, um dessen Typ zu bestimmen:

```
user$ file sinus.png
sinus.png: PNG image data, 718 x 575 ..
        # sinus.png ist eine Bilddatei

user$ mv sinus.png sinus.pdf
        # So etwas sollte man natürlich 'nicht'
        # machen!

user$ file sinus.pdf
sinus.pdf: PNG image data, 718 x 575 ..
```

Der Befehl `file` erkennt also die wahre Natur einer Datei trotz komplett falscher Endung! Er erlaubt es uns auch festzustellen, ob eine gegebene Textdatei vom *Unix*- oder vom *Microsoft Windows*-Typ ist:

```
user$ file ascii_unix.txt
ascii_unix.txt: ASCII text
user$ todos file ascii_windows.txt
   # Wir hatten die Datei file ascii_windows.txt
   # oben ins Unix-Format umgewandelt. Deshalb
   # mache ich das hier rückgängig.
user$ file ascii_windows.txt
ascii_dos.txt: ASCII text, with CRLF line terminator
```

Die Ausgabe „`with CRLF line terminator`“ sagt im Wesentlichen, dass die Datei im *Microsoft Windows*-Format kodiert ist. „`ASCII text`“ (ASCII steht für „American Standard Code for Information Interchange“) bedeutet, dass die Datei nur Zeichen enthält, die auf einer amerikanischen/englischen Tastatur zu finden sind, also insbesondere keine deutschen Umlaute oder andere europäischen Sonderzeichen enthält. Mit dieser Thematik beschäftigen wir uns im Folgenden weiter.

12.1.3 Textdateien mit internationalen Sonderzeichen

Die Kodierung für Textdateien war anfangs primär für Zeichen, welche sich auf einer amerikanischen oder englischen Tastatur befinden, definiert. Solche Dateien

sind nach dem, bereits im vorigen Abschnitt erwähnten, ASCII-Standard kodiert. Man beachte, dass dieser Standard nicht die Behandlung von Sonderzeichen vieler Sprachen (z. B. unsere deutschen Umlaute) oder gar die Behandlung anderer Alphabete (z. B. das griechische) vorgesehen hat. Für das Problem wurden zunächst viele *lokale* Standards entwickelt, welche den ASCII-Standard um spezielle Zeichen einzelner Länder oder Regionen erweitert haben. In Westeuropa war der Standard *Latin1* bzw. *ISO8859-1* lange Zeit erste Wahl.

In der Zwischenzeit hat sich der um 1993 entwickelte Standard *UTF-8* (8-Bit UCS Transformation Format) weitgehend etabliert. Er bietet die Möglichkeit, alle Zeichen, die weltweit in Gebrauch sind, zu kodieren. Er wird auch ständig mit eventuell neuen Zeichen aktualisiert.

Falls man Textdateien mit Sonderzeichen benutzt, so muss, ähnlich wie in Abschn. 12.1.1, sichergestellt sein, dass sowohl die Textdatei, als auch die sie verarbeitenden Programme *dieselbe Sprache sprechen*:

```
user$ file deutsch_utf8.txt
deutsch.txt: UTF-8 Unicode text
     # Text im UTF-8-Format
user$ cat deutsch_utf8.txt
Die süße Hündin läuft in die Höhle des Bären.
user$ file deutsch_latin1.txt
deutsch_latin1.txt: ISO-8859 text
     # Text im latin1 (ISO-8859) Format
user$ cat deutsch_latin1.txt
Die s��e H�ndin l�uft in die H�hle des B�ren.
```

Hier ist mein Terminal für das UTF-8-Format eingestellt und kommt daher mit den Sonderzeichen der Latin1-Datei nicht zurecht. Für diese Fälle bietet *Unix* das Programm `iconv`, um Kodierungen von Textdateien ineinander umzuwandeln:

```
user$ iconv -f latin1 -t utf-8 -o deutsch_utf8_neu.txt\
   deutsch_latin1.txt
     # Der Befehl wandelt die Datei
     # deutsch_latin1.txt aus dem latin1
     # in das utf-8-Format um. Die
     # resultierende Datei heisst
     # deutsch_utf8_neu.txt.
user$ cat deutsch_utf8_neu.txt
Die süße Hündin läuft in die Höhle des Bären.
     # alles o.k.
```

Die Unterstützung von Textdateien mit internationalen Sonderzeichen innerhalb von *Unix*-Programmen hat sich in den Jahren seit circa 2005 extrem verbessert. Dennoch ist der ASCII-Standard nach wie vor der *kleinste gemeinsame Nenner*. Bei Weitem nicht alle Programme kommen mit Zeichen außerhalb des ASCII-Standards zurecht. Vor allem in Textdateien, die wissenschaftliche Daten repräsentieren, sollte auf jegliche Sonderzeichen verzichtet und alles im ASCII-Format abgespeichert

werden – allein schon deshalb, damit Sie solche Dateien problemlos mit Kollegen austauschen und weitergeben können.

Wahrscheinlich werden Sie auch deutsche Texte bearbeiten wollen und können dabei nicht immer auf die Verwendung von Umlauten verzichten. Beachten Sie hierbei stets Folgendes, um Probleme zu vermeiden:

1. Arbeiten Sie stets mit dem UTF-8-Format! Dies betrifft einerseits die Kodierung von Textdateien, andererseits System- und Programmeinstellungen. Überprüfen Sie z. B. in Ihren Terminaleinstellungen, dass dort die UTF-8-Kodierung benutzt wird. Auch muss *Unix*-Programmen, falls sie es unterstützen, explizit gesagt werden, dass sie UTF-8-Kodierung zu erwarten haben. Ob Ihr System hier richtig konfiguriert ist, erfahren Sie mit dem Befehl `locale`:

   ```
   user$ locale
   LANG=en_US.UTF-8
   .
   .
   ```

   Ich arbeite auf einem System mit US-amerikanischen Einstellungen und UTF-8-Kodierung. Falls Sie ein deutsches *Unix* haben, sollten Sie hier etwas wie `LANG=de_DE.UTF-8` sehen. Falls Sie jedoch etwas wie `LANG=de_DE.iso88591` bekommen, sollten Sie die `locale`-Einstellungen Ihres Systems auf UTF-8 ändern. Konsultieren Sie hierzu das Internet oder Ihre Dokumentation. Bei modernen *Unix*-Installationen sind die Einstellungen aber typischerweise bereits alle auf UTF-8 gesetzt!
2. Das Latin1-Format wird immer noch viel benutzt, und es ist wahrscheinlich, dass Sie über kurz oder lang solche Dateien erhalten. Sie erkennen das spätestens, wenn Sie beim Anschauen *merkwürdige Zeichen* anstatt Umlaute bekommen. Behandeln Sie die Dateien dann erst mit `file` und `iconv`, bevor Sie mit ihnen weiterarbeiten.

In den restlichen Kapiteln des Buches werde ich Textdateien explizit mit ihrer Kodierung benennen, also z. B. *ASCII-Textdateien*.

12.2 ASCII-Textdateien effektiv anzeigen

In den verbleibenden Kapiteln werden wir uns intensiver mit *Unix*-Techniken und *Unix*-Werkzeugen zum Bearbeiten von ASCII-Textdateien beschäftigen. Hierbei ist es oft nötig, sich neue Dateien schnell anzusehen. Für das einfache Anschauen des Inhalts, auch großer Dateien, lohnt sich der Aufruf eines Editors oft nicht. Hier stellt *Unix* unter anderen die Programme `cat`, `head`, `tail` und `less` zur Verfügung:

- `cat [-n] DATEI(EN)` (*concatenate*) gibt einfach die Dateien `DATEI(EN)` auf dem Bildschirm aus. Die Option `-n` *(numbers)* versieht jede Zeile mit einer Zeilennummer.

Tab. 12.1 Die wichtigsten Tastaturkommandos für den Pager `less`

Taste	Wirkung
↑, ↓	Blättern im Text
SPACE	Nächste Seite anzeigen
b	Vorherige Seite anzeigen
g	Zum Anfang der Datei springen
G	Zum Ende der Datei springen
q	Pager beenden
h	Hilfe anzeigen
/MUSTER	Nach MUSTER vorwärts suchen
?MUSTER	Nach MUSTER rückwärts suchen
n	Wiederholte Suche in gleicher Richtung durchführen
N	Wiederholte Suche in anderer Richtung durchführen
v	Den in der Umgebungsvariablen EDITOR definierten Editor aufrufen (siehe Abschn. 10.4)

- `head [-n ANZAHL] DATEI(EN)`, `tail [-n ANZAHL] DATEI(EN)` geben die ersten (letzten) Zeilen von `DATEI(EN)` auf dem Bildschirm aus. Standardmäßig sind dies zehn Zeilen. Die Anzahl kann mit der Option `-n` variiert werden. Der Befehl `tail` kann auch zur Überwachung von Dateien, die sich fortlaufend verändern, benutzt werden, z. B. von *Log-Dateien*. Mit der Option `-f` *(follow)* initiiert `tail` einen dauerhaften Prozess, der alle neu zu der Datei hinzukommenden Zeilen anzeigt.
- Der Pager `less` ist ein etwas komfortableres Programm zum einfachen Betrachten von Dateien. Er stellt Befehle zum Navigieren und zum Auffinden bestimmter Textstellen bereit. Falls es nötig wird, kann innerhalb des Pagers auch ein Editor für die gerade betrachtete Datei aufgerufen werden. Sie haben bereits Kontakt mit einem Pager in Verbindung mit dem `man`-Kommando gehabt. Er erlaubt primär ein seitenweises Betrachten einer Datei. Die wichtigsten Tastaturkommandos zur Navigation innerhalb von `less` sind in Tab. 12.1 zusammengefasst.

Außer zum einfachen Betrachten von Dateien sind diese Kommandos auch für die in Abschn. 13.3 diskutierten *Pipelines* von Bedeutung.

12.3 Datenspeicherung innerhalb von ASCII-Textdateien

Sie werden es bei Ihrer wissenschaftlichen Arbeit höchstwahrscheinlich viel mit Datentabellen in ASCII-Textdateien zu tun haben. Ein Beispiel ist die Datei `exposures.txt`, mit der wir im Folgenden noch viel arbeiten werden. Sie enthält Informationen über Beobachtungen astronomischer Daten an einem Großteleskop:

```
695833p D3 03AQ02 1543 0.79 1 i
695834p D3 03AQ02 1543 0.75 1 i
```

```
695835p D3 03AQ02 1543 1.01 2 i
695839p D3 03AQ02 1543 0.76 1 i
695840p D3 03AQ02 1543 0.74 1 i
695841p D3 03AQ02 1543 0.74 1 i
.
716128p D4 03BQ02 1707 0.96 1 r
716129p D4 03BQ02 1707 0.93 1 r
716130p D4 03BQ02 1707 0.9 1 r
.
715504p W1p4p3 03BQ02 1700 0.8 1 g
715505p W1p4p3 03BQ02 1700 0.76 1 g
715506p W1p4p3 03BQ02 1700 0.76 1 g
.
.
```

Die Bedeutung der einzelnen Spalten ist: (1) Identifikationsnummer der Beobachtung (`ID`); (2) Beobachtungsregion (insgesamt sind 198 verschiedene in der Datei enthalten); (3) Beobachtungsperiode der Beobachtung (eine Periode umspannt mehrere Monate); (4) Tag der Beobachtung kodiert als Zahl; (5) Maß für die Qualität der Beobachtung (je kleiner die Zahl, desto besser); (6) Maß für die Wetterbedingungen während der Beobachtung (je höher, desto schlechter); (7) Beobachtungskonfiguration (einer der fünf Buchstaben `u, g, r, i` und `z`). Die Datei folgt einem sehr allgemeinen Muster für die Speicherung von Daten in einer ASCII-Textdatei: Es wird ein Datensatz *pro Zeile* gespeichert, und jede Zeile enthält *alle* Informationen bezüglich eines bestimmten Datensatzes! Im Englischen spricht man bei einem Datensatz auch von einem *record* und von dem Format als *one record per line*. Ich betone dies, da *fast alle Unix*-Werkzeuge, die mit ASCII-Textdateien arbeiten, eine Datei *zeilenweise* bearbeiten und besonders auf dieses Format zugeschnitten sind. Im Folgenden werde ich Ihnen unter anderem zeigen, wie Sie mithilfe von *Unix*-Programmen Fragen wie die folgenden sehr leicht beantworten können:

1. Für welche Beobachtungsregionen liegen Aufnahmen vor?
2. Für welche Beobachtungsregionen liegen Beobachtungen in den drei Konfigurationen `g, r` und `i` vor?
3. Was ist der beste und der schlechteste Qualitätswert (fünfte Spalte) in jeder Beobachtungskonfiguration?
4. Liste alle Beobachtungs-IDs von Region `W1p4p3` in der Konfiguration `i`.

Die vorliegende Datei `exposures.txt` hat insgesamt 17 899 Einträge:

```
user$ wc exposures.txt
17899 125293 576384 exposures.txt
```

Der Befehl `wc` (*word count*) gibt Informationen über die Anzahl von Zeilen, Wörtern und einzelnen Zeichen in einer Textdatei. Ein *Wort* ist hier eine Zeichenkette von

einem oder mehreren Zeichen, die durch *Leerzeichen* (␣, `<Tab>`, Zeilenanfang, Zeilenumbrüche) begrenzt sind. Wegen der Größe der Datei und der Menge an Einträgen ist es nicht möglich, obige Probleme *mal eben fix per Hand* durch Laden der Datei in einen Editor zu lösen. Allerdings lassen sich diese Aufgaben *sehr effektiv* mit sehr wenigen *Unix*-Programmen in Verbindung mit sogenannten *Datenpipelines* lösen. Diesen Aufgabenkomplex werden wir in Kap. 13 und 14 angehen.

Datenströme und Datenpipelines 13

Übersicht

Zusammenfassung

In diesem Kapitel lernen Sie, wie Sie Ausgaben von *Unix*-Programmen effizient weiterverwenden können. Die einfachste Form ist die Speicherung von Bildschirmausgaben in Textdateien. Der nächste Schritt ist der Aufbau von Prozessketten, in welchen die Ausgabe von Programmen *direkt* in andere Prozesse weitergeleitet werden. *Unix*-Programme, die meist nur einen sehr beschränkten Funktionsumfang haben, können so für sehr komplexe Aufgaben effizient miteinander verbunden werden.

13.1 Die Ausgabeströme `STDOUT` und `STDERR`

Bis jetzt haben wir *Unix*-Befehle ausgeführt, und die Programme haben uns das Ergebnis auf dem Terminal präsentiert. Manchmal würden wir die Ausgabe von Programmen gerne weiterverwenden:

```
user$ ls -ltr
-rw-r--r-- 1 thomas users   4584 Oct  9 18:24 howto.tex
-rw-r--r-- 1 thomas users   3395 Oct 11 09:26 linux.tex
-rw-r--r-- 1 thomas users   6243 Oct 11 12:46 draft.tex
-rw-r--r-- 1 thomas users    603 Oct 11 13:00 tutor.txt
-rw-r--r-- 1 thomas users  15272 Oct 12 00:31 EDV.txt
```

`ls` stellt die Option `-S` (*sort by file size*) zur Verfügung, um Dateien nach der Dateigröße sortiert anzuzeigen. Wir wollen im Folgenden zu Demonstrationszwe-

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_13

cken annehmen, dass es diese Option nicht gibt und wir mit `ls` nur obige Ausgabe erhalten können. Unsere Aufgabe besteht also darin, obige Liste nach der Dateigröße (fünfte Spalte) zu sortieren, *ohne* auf die `-S` Option von `ls` zurückzugreifen! Angenommen, wir hätten obige Ausgabe in einer Datei namens `datei.txt`, so könnten wir den `sort`-Befehl (siehe Aufgabe 4.2) zur Lösung verwenden:

```
user$ sort -k5,5 -n datei.txt
-rw-r--r-- 1 thomas users    603 Oct 11 13:00 tutor.txt
-rw-r--r-- 1 thomas users   3395 Oct 11 09:26 linux.tex
-rw-r--r-- 1 thomas users   4584 Oct  9 18:24 howto.tex
-rw-r--r-- 1 thomas users   6243 Oct 11 12:46 draft.tex
-rw-r--r-- 1 thomas users  15272 Oct 12 00:31 EDV.txt
```

Technisch betrachtet *Unix* die Ausgabe eines Programms als einen *Datenstrom*. Der Datenstrom, der für die Ausgabe zuständig ist, wird als *Standardausgabe* oder im Englischen kurz als `STDOUT` (*Standard Output*) bezeichnet. Standardmäßig ist `STDOUT` mit dem Terminal des Programms verbunden. Sie lässt sich allerdings vom Benutzer *in eine Datei umlenken*. Dies geschieht mit dem „>“-Operator:

```
# STDOUT wie gehabt auf dem Terminal
#
user$ ls -ltr
-rw-r--r-- 1 thomas users   3395 Oct 11 09:26 linux.tex
-rw-r--r-- 1 thomas users    603 Oct 11 13:00 tutor.txt
.
.
# STDOUT in die Datei 'listing.txt' umgelenkt
#
user$ ls -ltr > listing.txt
user$ cat listing.txt
-rw-r--r-- 1 thomas users   3395 Oct 11 09:26 linux.tex
-rw-r--r-- 1 thomas users    603 Oct 11 13:00 tutor.txt
.
.
```

Der Befehl `ls -ltr > listing.txt` schreibt also seine Ausgabe in die Datei `listing.txt`, anstatt sie auf dem Terminal zu zeigen! Hiermit lässt sich obiges Problem einer nach der Dateigröße sortierten Inhaltsliste leicht lösen:

```
# STDOUT in 'listing.txt'
#
user$ ls -ltr > listing.txt

# nach der 5. Spalte numerisch sortieren
#
user$ sort -k5,5 -n listing.txt
```

Bemerkungen und Weiterführendes:

- **Wichtig:** Ausgabeumlenkungen, wie unser `ls -ltr > listing.txt`, überschreiben die Datei `listing.txt`, falls sie schon existiert!
- Falls ein Befehl überhaupt keine Ausgabe erzeugt, bleibt nach einer Ausgabeumlenkung eine leere Datei mit der Dateilänge null zurück.
- Manchmal möchte man eine Datei durch eine modifizierte Version *ersetzen*:

```
user$ cat namen.txt
Thomas  Erben
Alfred  Anderl
Karl    Klein
```

Hier würde ich gerne die Datei `namen.txt` durch eine Version mit alphabetisch sortierten Namen ersetzen. Ein spontaner Lösungsansatz mit einer Ausgabeumlenkung wäre:

```
user$ sort namen.txt > namen.txt
   # Das geht gründlich schief!!
user$ cat namen.txt
user$ ls -ltr namen.txt
-rw-r--r-- 1 thomas users 0 Feb 16 22:50 namen.txt
```

Anstatt einer sortierten Datei bekommen wir also eine *leere* zurück. Unsere ursprüngliche Datei existiert nicht mehr!

Erklärung: Die in dem Befehl `sort namen.txt > namen.txt` involvierten Prozesse laufen nicht *in der erwarteten Reihenfolge* ab. Es wird *nicht* erst die Datei eingelesen, dann sortiert und danach neu geschrieben. In vorliegendem Fall wird als Erstes durch die Ausgabeumlenkung eine *leere* Datei `namen.txt` erzeugt. Erst danach versucht der `sort`-Befehl aus dieser, jetzt leeren, Datei zu lesen! Hier kommt man leider nicht um eine Zwischendatei herum:

```
user$ sort namen.txt > tmp.txt
user$ cat tmp.txt
Alfred  Anderl
Karl    Klein
Thomas  Erben
user$ mv tmp.txt namen.txt
```

- Anstatt mit der Ausgabeumlenkung eine neue Datei zu erzeugen, kann man die Ausgabe auch an eine bereits bestehende anhängen. Dies geschieht mit dem „>>“-Operator:

```
user$ echo "hello world" > hello.txt
user$ cat hello.txt
hello world
user$ echo "hello again" >> hello.txt
user$ cat hello.txt
hello world
hello again
user$
```

- Neben der *normalen Ausgabe* eines Programms gibt es unter Umständen auch *Fehlermeldungen*. *Unix* unterscheidet hier den Datenstrom der *Standardausgabe* (`STDOUT`) von dem der *Fehlerausgabe* (`STDERR`; *Standard Error*). Auch `STDERR` ist standardmäßig mit dem Terminal verbunden. Ebenso wie `STDOUT` lässt sich `STDERR` in eine Datei umlenken, nämlich mit dem Operator „`2>`“:

  ```
  user$ cat gibtsnicht.txt
  cat: gibtsnicht.txt: No such file or directory
  user$ cat gibtsnicht.txt > fehler.txt
  cat: gibtsnicht.txt: No such file or directory
  user$ cat gibtsnicht.txt 2> fehler.txt
  user$ cat fehler.txt
  cat: gibtsnicht.txt: No such file or directory
  ```

  Natürlich kann man Umleitungen von `STDOUT` und `STDERR` auch kombinieren:

  ```
  # Umleitungen in verschiedene Dateien
  #
  user$ ls > ausgabe.txt 2> fehler.txt
  # Umleitungen in die gleiche Datei
  #
  user$ ls > ausgabe.txt 2> ausgabe.txt
  # Umleitungen in die gleiche Datei (Kurzform)
  #
  user$ ls >& ausgabe.txt
  ```

  Hierbei ist der Befehl `ls >& ausgabe.txt` völlig identisch zu `ls > ausgabe.txt 2> ausgabe.txt` und lediglich eine Kurzschreibweise. Die meisten Leute verwenden fast ausschließlich die `ls >& ausgabe.txt`-Form der Umleitung.

13.2 Der Eingabestrom `STDIN`

Nachdem wir Datenströme zur Ausgabe betrachtet haben, sehen wir uns jetzt den entgegengesetzten Vorgang der *Eingabe* an. Der zugehörige Datenstrom wird als `STDIN` (*Standard Input*) bezeichnet. So, wie die Ausgabe standardmäßig mit dem Terminal verbunden ist, ist `STDIN` *per Default* an die Tastatur gekoppelt. Sie können mit vielen *Unix*-Programmen direkt über die Tastatur kommunizieren. Geben Sie im Folgenden das Kommando `sort` (*ohne* Argumente). Danach tippen Sie nacheinander `Martha <Return> Albert <Return> Thomas <Return> <C-d>`:

```
user$ sort
Martha  <Return>
Albert  <Return>
```

```
Thomas  <Return>
<C-d>
Albert
Martha
Thomas
user$
```

Erklärung: Das Kommando `sort` ohne Argumente wartet auf Eingabe von `STDIN`. Diese wird mit `Martha <Return> Albert <Return> Thomas <Return>` geliefert. Das abschließende `<C-d>` bezeichnet das *Ende der Eingabe*, und `sort` führt seine Aufgabe (Sortierung) anhand der bereitgestellten Daten aus.

Die Einführung von `STDIN` erscheint den meisten anfangs verwirrend und, vor allem, unnötig! Sie haben bisher gesehen, dass man `sort` eine Datei mit zu sortierenden Daten *als Argument übergibt* und etwas wie:

```
user$ sort datei.txt
```

aufruft. Bitte beachten Sie hier, dass es sich bei *einem Argument, das von einem Programm als Datei mit zu sortierendem Inhalt interpretiert wird*, und der Dateiübergabe via `STDIN` um zwei grundlegend verschiedene Dinge handelt! In der Tat gibt es *Unix*-Programme, die ihre Daten *nur* über `STDIN` empfangen können und die die Möglichkeit eines Dateiarguments nicht bieten. Die Nützlichkeit des `STDIN`-Konzepts wird in Abschn. 13.3 bei den *Pipelines* deutlich werden.

Wie die Dateiausgabeströme lässt sich auch `STDIN` umlenken, sodass eine Datei anstatt der Tastatur als Eingabemedium für ein Programm dient. Dies geschieht mit dem `<`-Operator:

```
user$ cat tr.txt
Milch macht muede Maenner munter.
user$ tr 'M' 'm' < tr.txt > tr_neu.txt
user$ cat tr_neu.txt
milch macht muede maenner munter.
```

Das Programm `tr` (*translate*) kann seine Daten *nur* über `STDIN` empfangen. Im Beispiel dient die Datei `tr.txt` als Datenlieferant. Das Programm übersetzt aus `STDIN` alle Zeichen des ersten Arguments (hier `M`) in die des zweiten Arguments (hier `m`). Das Ergebnis des Befehls wird über eine Ausgabeumlenkung in der Datei `tr_neu.txt` gespeichert.

Aufgabe 13.1

Vielleicht haben Sie schon erlebt, dass Sie ein *Unix*-Programm aufrufen und anscheinend *nichts passiert*. Es kommt keine Fehlermeldung, und auch die Shell meldet sich *nicht* mit einem Prompt zurück:

```
user$ cat
  # Anscheinend passiert hier gar nichts und
```

```
# Sie müssen etwas wie <C-c> drücken, um zur Shell
# zurückzukehren.
```

Erklären Sie, was hier vor sich geht.
Lösungsvorschlag auf Seite 192!

13.3 Pipelines aus *Unix*-Befehlen

Wie wir gesehen haben, schreiben *Unix*-Programme ihre Ausgabe auf `STDOUT`, und eine der Möglichkeiten, um Daten zu beziehen, ist `STDIN`. Dies lässt sich nun ausnutzen, um Programme zu *verbinden*, indem man `STDOUT` eines Programms *direkt* in `STDIN` eines anderen lenkt. Eine sogenannte Pipeline (im Englischen *pipe*; eine Verbindung zwischen `STDOUT` eines Programms und `STDIN` eines folgenden) wird mit dem Operator „|“ aufgebaut:

```
user$ ls -ltr | sort -k5,5 -n
-rw-r--r-- 1 thomas users    603 Oct 11 13:00 tutor.txt
-rw-r--r-- 1 thomas users   3395 Oct 11 09:26 linux.tex
-rw-r--r-- 1 thomas users   4584 Oct  9 18:24 howto.tex
-rw-r--r-- 1 thomas users   6243 Oct 11 12:46 draft.tex
-rw-r--r-- 1 thomas users  15272 Oct 12 00:31 EDV.txt
```

Erklärung: Es wird eine Pipeline zwischen den Kommandos `ls` und `sort` aufgebaut. Die Ausgabe von `ls -ltr` wird direkt in den Befehl `sort -k5,5 -n` geleitet. Ausgabe ist eine, nach Dateigröße sortierte, Verzeichnisliste. Obige Pipeline ist also von der Funktionalität her identisch mit folgender *Befehlsfolge*:

```
user$ ls -ltr > listing.txt
user$ sort -k5,5 -n listing.txt
```

Eine Pipeline macht also das Anlegen von *Zwischendateien* bei Befehlsfolgen, die jeweils von der Ausgabe des letzten Befehls abhängen, unnötig. Natürlich gibt es keinen Grund, dass eine Pipeline nur aus zwei Befehlen besteht:

```
user$ ls -ltr | sort -k5,5 -n | awk '{print $5}'
603
3395
4584
6243
15272
```

Den `awk`-Befehl schauen wir uns in Abschn. 14.2 genauer an. Er extrahiert in der hier gezeigten Form aus einem Datenstrom die fünfte Spalte. Ich werde Ihnen in Kap. 14 etliche *Unix*-Programme vorstellen, die häufig in Pipelines Anwendung finden. Diese Werkzeuge werden oft auch als *Filter* bezeichnet.

Bemerkungen und Weiterführendes:

- Erfahrungsgemäß haben Anfänger mit dem Verständnis und der Konstruktion komplexerer Pipelines Probleme. Dies gilt vor allem für längere Befehlsketten. Sobald Sie eine Pipeline aus mehreren Befehlen konstruieren und feststellen, dass sie nicht das Gewünschte leistet, haben Sie Probleme herauszufinden, *welcher* Teil der Pipeline nicht wie erwartet funktioniert. Es ist daher wichtig, dass Sie sich von Anfang an angewöhnen, Pipelines *zu testen* und *schrittweise* zu entwickeln. Sehen Sie sich bei längeren Konstruktionen zuerst die Ausgabe des ersten Kommandos an, dann die Ausgabe nach dem zweiten usw. Wenn Sie Pipelines auf lange Dateien anwenden möchten, ist es auch eine gute Idee, zum Testen eine kurze, überschaubare Version der Datei zu erstellen. Ein zweiter wichtiger Tipp ist es, sich Pipelines von Anderen anzusehen und zu analysieren.
- Für Befehle (oder Pipelines), die eine längere Ausgabe erzeugen, ist es oft nützlich, sie mit `less` abzuschließen, um sich das Ergebnis anzusehen. Für ein Verzeichnis mit sehr vielen Dateien wäre z. B.

  ```
  user$ ls -ltr | less
  ```

  denkbar. Mit `less` lässt sich das Ergebnis genauer ansehen als in einer einfachen Terminalausgabe. Man kann komfortabel im Text springen und bestimmte Stellen suchen; siehe Abschn. 12.2. Aufgabe 13.4 zeigt Ihnen, wie Sie eine Pipeline mit `less` abschließen und gleichzeitig das Ergebnis in einer Datei speichern können.
- Um mehrere Dateien zu kombinieren, bevor man sie in eine Pipeline steckt, ist der `cat`-Befehl sehr nützlich. Angenommen, Sie wollten Großbuchstaben von Text, der in den Dateien `text_1.txt, .., text_100.txt, ..` gespeichert ist, in Kleinbuchstaben verwandeln. Dann ginge z. B.:

  ```
  user$ cat text_*txt | tr '[:upper:]' '[:lower:]' > \
     out.txt
  ```

  Hier wird besonders deutlich, dass *Unix*-Programme, deren Nutzen für sich alleine betrachtet vielleicht zweifelhaft erscheint, in Pipelines sehr wertvoll sind, wie hier z. B. `cat`.

  Beachten Sie, dass in dem Beispiel die Daten aller Eingangsdateien *kombiniert* und das Ergebnis in *eine einzige* Ausgabedatei gespeichert wird. Wollte man obige Operation separat für jede Eingangsdatei mit jeweils einer eigenen Ausgabedatei durchführen, so müsste dies mit einem *Shell-Schleifenkonstrukt* geschehen. Das werden wir in Abschn. 15.2 kennenlernen.
- Manchmal möchte man in einer Pipeline Daten aus einem Datenstrom mit denen eines Arguments kombinieren:

  ```
  user$ cat namen.txt
  Alfred  Anderl
  Karl    Klein
  Thomas  Erben
  ```

```
user$ cat vornamen.txt
Michael
Andrea
Hubert
# Wir wollen 'nur' die 'Vornamen' aus namen.txt
# und 'vornamen.txt' sortieren und ausgeben:
user$ awk '{print $1}' namen.txt | \
  sort - vornamen.txt > vornamen_sort.txt
user$ cat vornamen_sort.txt
Alfred
Andrea
.
.
Thomas
```

Erklärung: Der `awk`-Befehl (wir besprechen ihn genauer in Abschn. 14.2) isoliert aus `namen.txt` die erste Spalte (die Vornamen). Diese Vornamen werden zusammen mit denen in `vornamen.txt` sortiert und in `vornamen_sort.txt` gespeichert. Um dies zu erreichen, muss der `sort`-Befehl sowohl Daten von `STDIN` als auch aus einem Argument erhalten. Für solche Fälle unterstützen viele Programme den *einfachen Bindestrich* als Argument. Er repräsentiert den Datenfluss aus `STDIN` und er kann mit anderen Argumenten, welche zu bearbeitende Daten enthalten, kombiniert werden.

- In Abschn. 1.2 haben wir diskutiert, dass *Unix* ein ruhiges Betriebssystem ist, welches unnötige Ausgaben, z. B. Meldungen über beendete Programme, vermeidet. Der dort aufgeführte Grund war eine Ressourcenschonung zu den Anfangszeiten von *Unix*. Ein anderes, auch heute sehr relevantes, Argument sind die *Unix*-Pipelines. Diese lassen sich bei Textdateien am effektivsten einsetzen, wenn alle Zeilen des Eingabestroms *genau dieselbe Struktur* haben! Sollte die Eingabe inhomogen sein, so muss dies in Datenströmen aufwendig berücksichtigt werden. Würden Programme z. B. zusätzlich zu relevanten Ausgabedaten *unnötige Meldungen* über den Programmablauf enthalten, so müssten wir diese Statusmeldungen in einer Pipeline wieder explizit herausfiltern. Behalten Sie dies im Hinterkopf, wenn Sie selbst einmal Programme mit einer Textausgabe entwickeln!
- Wir haben Datenströme und Pipelines bisher in Verbindung mit Textausgaben und Textdateien gesehen. Hierauf wird unser Fokus auch für den Rest dieses Buches gerichtet sein. Pipelines sind aber nicht hierauf beschränkt! Als Beispiel haben wir zwei Bilddateien (`sinus.png` und `cosinus.png`) im *PNG*-Format (Portable Network Graphics). Diese wollen wir erst zu einem einzigen Bild (`sinus_cosinus.png`) zusammenfügen und hinterher ins *JPG*-Format (Joint Photographic Experts Group) umwandeln. Unter *Linux* stehen hierfür die Programme `montage` und `convert` aus der *ImageMagick*-Bibliothek zur Verfügung:

```
user$ ls
cosinus.png sinus.png
user$ montage -geometry +1+1 -tile x1 \
  sinus.png cosinus.png sinus_cosinus.png
     # füge die beiden Bilddateien sinus.png und
     # cosinus.png zu einer Datei sinus_cosinus.png
     # zusammen
user$ convert sinus_cosinus.png sinus_cosinus.jpg
     # wandle die Datei sinus_cosinus.png in das
     # JPG Format um
user$ ls
cosinus.png sinus_cosinus.jpg sinus_cosinus.png
sinus.png
```

Zum Betrachten von Bilddateien stellt ImageMagick das Programm `display` zur Verfügung. Die Befehle der *ImageMagick*-Bibliothek können alle durch Pipelines miteinander kombiniert werden, und die Datei `sinus_cosinus.jpg` kann somit ohne das Zwischenprodukt `sinus_cosinus.png` erstellt werden:

```
user$ ls
cosinus.png sinus.png
user$ montage -geometry +1+1 -tile x1 \
  sinus.png cosinus.png - |\
  convert - sinus_cosinus.jpg
user$ ls
cosinus.png sinus_cosinus.jpg sinus.png
```

Aufgabe 13.2

Was ist die Gesamtzahl an Dateien und Verzeichnissen in Ihrem Heimatverzeichnis?

Hinweis: Benutzen Sie eine Pipeline aus den Kommandos `ls` und `wc`.

Lösungsvorschlag auf Seite 192!

Aufgabe 13.3

Konstruieren Sie eine Pipeline aus den Befehlen `head` und `tail` (siehe Abschn. 12.2), um die Zeilen 111–120 unserer Datei `exposures.txt` anzuzeigen.

Hinweis: Um Ihre Pipeline zu überprüfen, können Sie sie mit dem Befehl `cat -n` einleiten. Hiermit erscheinen vor allen Zeilen Ihrer Datei Zeilennummern.

Lösungsvorschlag auf Seite 193!

Aufgabe 13.4

Erkundigen Sie sich über das Kommando `tee` und erklären Sie, was in jedem Einzelschritt der folgenden Pipeline geschieht:

```
user$ ls -ltr | sort -k5,5 -n | tee output.txt | less
```

Lösungsvorschlag auf Seite 193!

Aufgabe 13.5

Erkundigen Sie sich über das Kommando `xargs` und erklären Sie, was in jedem Einzelschritt der folgenden Pipelines geschieht:

```
user$ find /home/thomas -type f -name '*.txt' | \
  xargs wc -l
```

und

```
user$ find . -type f -mtime -7 | \
  xargs tar -cvzf backup_der_letzten_Woche.tgz
```

Der `tar`-Befehl zum Archivieren von Dateien ist in Anhang B beschrieben.
Lösungsvorschlag auf Seite 194!

Aufgabe 13.6

a) Sie wollen eine nach Dateigröße sortierte Liste *aller Dateien* in Ihrem Heimatverzeichnisbaum. Geben Sie hierfür eine Pipeline an. Leider bietet `ls` keine Möglichkeit an, um *nur Dateien* zu listen. Sie müssen sich hier z. B. mit dem `find`-Befehl behelfen und Ihre Pipeline mit

   ```
   user$ find ~ -type f | ......
   ```

   starten.

   Hinweis: Bearbeiten Sie Aufgabe 13.5 vor dieser Aufgabe.

b) Sie wollen wissen, welche die drei größten Dateien im Verzeichnisbaum Ihres Heimatverzeichnisses sind. Geben Sie hierfür eine Pipeline aus *Unix*-Kommandos an.

Lösungsvorschlag auf Seite 196!

Unix-Werkzeuge zur Arbeit mit ASCII-Textdateien

14

Übersicht

Zusammenfassung

In diesem Kapitel stelle ich Ihnen vier der wichtigsten *Unix*-Programme zur effizienten Analyse von Daten in ASCII-Textdateien vor. Sie werden sehen, dass Sie mit diesen vier Programmen und den Datenpipelines von Kap. 13 schon recht komplexe Datenanalysen durchführen können.

14.1 Finden von Text in ASCII-Dateien mit `grep`

`grep` (*Global Regular Expression Print*) ist sicher eines der allerwichtigsten *Unix*-Werkzeuge für Pipelines. Es durchsucht ASCII-Textdateien *zeilenweise* nach bestimmten Mustern und druckt per Default entsprechende Zeilen aus. Seine Syntax ist `grep [OPTIONEN] TEXTMUSTER DATEIEN`. Die einfachste Anwendung des Befehls ist etwa:

```
user$ grep W2m0m0 exposures.txt
831549 05BQ11 2554 0.73 1 W2m0m0 i
831550 05BQ11 2554 0.74 1 W2m0m0 i
831551 05BQ11 2554 0.7 1 W2m0m0 i
831552 05BQ11 2554 0.79 1 W2m0m0 i
.
.
```


T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_14

Tab. 14.1 Wichtige Optionen des Kommandos `grep`

`grep` Option	Erklärung
`-c` (*count*)	Nur die Anzahl der passenden Zeilen zeigen
`-h` (*hide*)	Dateinamen nicht ausgeben (bei mehreren Dateien)
`-i` (*ignore case*)	Groß-/Kleinschreibung ignorieren
`-l` (*list*)	Nur Dateinamen auflisten, nicht die Textzeilen
`-n` (*number*)	Zeilennummern den passenden Zeilen voranstellen
`-v` (*vice versa*)	Die *nicht* passenden Zeilen anzeigen

Dies zeigt alle Zeilen unserer Datei `exposures.txt`, die Informationen über Beobachtungsregion `W2m0m0` haben. Das `TEXTMUSTER`, nach dem mit `grep` gesucht werden soll, kann nicht nur eine einfache Zeichenkette, sondern im Allgemeinen ein sogenannter *Regulärer Ausdruck* sein. Dies erlaubt sehr komplexe und detailreiche Suchoperationen. Da auch eine erste Einführung in dieses Thema etwas umfangreicher ausfällt, finden Sie sie als eigenständiges Kapitel in Anhang A des Buches. Als *Appetitanreger* möchte ich hier nur das Problem „Liste alle Beobachtungs-IDs von Region `W1p4p3` in der Konfiguration `i`“ aus Abschn. 12.3 besprechen. Im vorliegenden Fall könnte man es so lösen:

```
user$ grep W1p4p3 exposures.txt | grep i
```

Dass diese einfache Lösung hier möglich ist, liegt daran, dass der Buchstabe `i` *nur* in der Spalte über die Beobachtungskonfiguration auftritt. Im Allgemeinen wird ein spezieller einzelner Buchstabe öfter in einer Zeile auftauchen, und obiger Befehl würde dann *zu viel* liefern, nämlich *alle* Zeilen, in denen *irgendwo* ein `i` auftaucht! Eine *eindeutige* Identifikation für das gewünschte `i` wäre etwa: „Es befindet sich nach einem Leerzeichen am Ende der Zeile“. Ein *Regulärer Ausdruck*, der nach einem solchen Muster suchen würde, wäre „␣`i$`“:

```
user$ grep W1p4p3 exposures.txt | grep ' i$'
```

Beachten Sie die *Quotierung* des Regulären Ausdrucks wegen der Shell-Sonderzeichen ␣ und `$`!

In Tab. 14.1 liste ich Ihnen noch einige nützliche und häufig benutzte Optionen von `grep`.

Aufgabe 14.1

Der Befehl `ps -eaf` gibt Ihnen eine Liste *aller* auf Ihrem Rechner laufenden Prozesse. Verbinden Sie ihn mit `grep` zu einer Pipeline, um nur Ihre eigenen Prozesse zu sehen.

Lösungsvorschlag auf Seite 196!

Aufgabe 14.2

Bestimmen Sie mithilfe einer Pipeline die Anzahl der Beobachtungen von Region D3 (zweite Spalte in der Datei exposures.txt).

Hinweis: Kommando wc.

Lösungsvorschlag auf Seite 196!

Aufgabe 14.3

Was passiert bei folgenden Befehlen:

```
user$ find . -name '*.txt' -type f | \
  grep 'Test'
```

und

```
user$ find . -name '*.txt' -type f | \
  xargs grep 'Test'
```

Hinweis: Bearbeiten Sie Aufgabe 13.5 vor dieser Übung.

Lösungsvorschlag auf Seite 197!

Aufgabe 14.4

Schauen Sie sich die Dateien emails.txt und nochmal_emails.txt an. Erklären Sie dann, was in jedem Einzelschritt der folgenden Pipelines geschieht:

```
user$ cat emails.txt | tr '<>' '\n' | grep @
```

und

```
user$ cat nochmal_emails.txt | tr '<>' '\n' | grep @
```

Lösungsvorschlag auf Seite 197!

Aufgabe 14.5

Jemand hat eine Datei test.txt mit dem Inhalt:

```
user$ cat test.txt
.
.
| a | b | c | d |
| d | e | f | g |
.
.
```

Er möchte Zeilen, die das Muster „f |“ enthalten, mit einem grep-Befehl finden. Hierfür wird der Befehl

```
user$ grep f | test.txt
test.txt: command not found
```

gegeben. Erklären Sie, wie und warum es zu der Fehlermeldung `test.txt: command not found` kommt. Geben Sie eine korrekte Version des Befehls an.
Lösungsvorschlag auf Seite 199!

14.2 Bearbeitung spaltenorientierter Dateien mit `awk`

Es wird `awk` (`awk` steht für die Initialien seiner Erfinder: *Aho, Weinberger, Kernighan*) nicht gerecht, es als gewöhnliches *Unix*-Werkzeug zu bezeichnen. Es ist eine komplette *Programmiersprache*, die auf die Bearbeitung spaltenorientierter ASCII-Textdateien spezialisiert ist. Um `awk` sinnvoll verwenden zu können, müssen Sie jedoch nur einige der grundlegendsten Anwendungsmöglichkeiten kennen, die ich im Folgenden anhand von Beispielen zeige. Es gibt im Netz unzählige gute Tutorials zu `awk`, und eine intensivere Beschäftigung damit zahlt sich für Sie definitiv aus! Alle folgenden Beispiele beziehen sich auf unsere Demodatei `exposures.txt`.

```
# drucke die erste Spalte der Datei
user$ awk '{print $1}' exposures.txt

# drucke die dritte Spalte der Datei
user$ awk '{print $3}' exposures.txt

# drucke die dritte und fünfte Spalte der Datei
user$ awk '{print $3, $5}' exposures.txt

# drucke die dritte und fünfte Spalte, aber nur wenn
# der Wert der vierten kleiner als 1700 ist:
user$ awk '($4 < 1700) {print $3, $5}' exposures.txt

# drucke die 'sechste minus die fünfte' Spalte:
# (Das Ergebnis ist nicht sinnvoll und dient nur
# zur Demonstration der awk-Syntax)
user$ awk '{print $6 - $5}' exposures.txt

# drucke alle Zeilen, die W1p4p3 enthalten:
user$ awk '/W1p4p3/' exposures.txt

# drucke Spalte 7 zusammen mit Zeilennummern:
user$ awk '{print NR, $7}' exposures.txt

# berechne die Summe aller Werte in der sechsten
```

```
# Spalte:
# (Das Ergebnis ist nicht sinnvoll und dient nur
# zur Demonstration der awk-Syntax)
user$ awk 'BEGIN {summe = 0} {summe = summe + $6} \
       END {print summe}' exposures.txt
```

Das letzte Beispiel dient nur als kleiner *Appetithappen* für die Möglichkeiten, wenn Sie sich näher mit awk beschäftigen.

Mithilfe von awk lässt sich für Sie das Problem „Was ist der beste und der schlechteste Qualitätswert (vierte Spalte) in jeder Beobachtungskonfiguration?“ aus Abschn. 12.3 z. B. wie folgt lösen:

```
grep ' u$' exposures.txt | awk '{print $4}' | \
  sort -n | less
```

Am Anfang und am Ende der Ausgabe kann man den kleinsten und größten Wert für die Beobachtungskonfiguration u ablesen – analog geht man für die anderen Konfigurationen g, r, i und z vor. Mit *etwas mehr* awk bekommt man die Werte direkt:

```
user$ grep ' u$' exposures.txt | awk '{print $5}' | \
  sort -n | awk '{if(NR == 1) {small = $1}} END {\
  large = $1; print small, large}'
```

14.3 Dateien sortieren mit sort und uniq

Sie haben bereits das Kommando sort zum Sortieren von ASCII-Textdateien kennengelernt, und ich möchte hier nicht mehr darauf eingehen. Sehr praktisch ist die Verbindung mit dem Programm uniq, welches mehrfach vorkommende Zeilen aus einer Datei entfernt. Allerdings benötigt uniq zuvor mit sort sortierte Daten, sodass dieser Befehl praktisch nur in einer Pipeline zusammen mit sort auftritt:

```
user$ cat sort_uniq.txt
Martha
Thomas
Michael
Martha
user$ sort sort_uniq.txt
Martha
Martha
Michael
Thomas
user$ sort sort_uniq.txt | uniq
Martha
Michael
```

```
Thomas
user$ sort sort_uniq.txt | uniq -c
      2 Martha
      1 Michael
      1 Thomas
```

Die Option `-c` (*count*) entfernt keine mehrfach vorkommenden Zeilen, sondern zählt die Häufigkeit ihres Auftretens. Mit dem Programmpaar `sort/uniq` lassen sich die verbleibenden Probleme aus Abschn. 12.3 für die Datei `exposures.txt` lösen:

1. „Für welche Beobachtungsregionen liegen Aufnahmen vor?“

   ```
   user$ awk '{print $2}' exposures.txt | sort | uniq
   ```

   Zuerst wird mit `awk` die Zeile mit den Regionen isoliert. Diese wird sortiert, und mehrfach vorkommende Einträge werden aussortiert.
2. „Für welche Beobachtungsregionen liegen Beobachtungen in den drei Konfigurationen `g`, `r` und `i` vor?“

   Das Problem kann auf mehrere Weisen angegangen werden, aber am einfachsten wird es in mehrere Schritte zerlegt. Im Folgenden erstellen wir jeweils Listen mit Regionen, die in `g`, `r` bzw. `i` beobachtet wurden (die ersten drei Pipelines). Die Listen werden jeweils so erstellt, dass keine Mehrfachnennungen von Regionen auftreten. In der vierten Pipeline werden die gerade erstellten Listen kombiniert. Regionen, die in allen Konfigurationen beobachtet wurden, müssen in der kombinierten Liste dreimal vorkommen und können somit leicht identifiziert werden:

   ```
   # Erstelle eine Datei 'g.txt', in der jede Region,
   # die in Konfiguration 'g' beobachtet wurde, genau
   # einmal vorkommt:
   user$  grep ' g$' exposures.txt | \
       awk '{print $2}' | sort | uniq > g.txt
   #
   # analog für Konfigurationen 'r' und 'i':
   user$  grep ' r$' exposures.txt | \
       awk '{print $2}' | sort | uniq > r.txt
   user$  grep ' i$' exposures.txt | \
       awk '{print $2}' | sort | uniq > i.txt
   #
   # Die gesuchten Regionen sind die, welche in der
   # Kombination von g.txt, r.txt und i.txt dreimal
   # vorkommen:
   user$ cat i.txt g.txt r.txt | sort | uniq -c | \
       awk '($1 == 3) {print $2}' > g_r_i.txt
   ```

   Das Ergebnis haben wir noch in einer Datei `g_r_i.txt` gespeichert.

Aufgabe 14.6

In der Datei `staedte.txt` finden Sie Daten deutscher Städte, die Ende 2013 eine Einwohnerzahl von mindestens 20 000 hatten. Die Liste enthält die Spalten *Stadtname*, *Bundesland* und *Einwohnerzahl*. Lösen Sie damit folgende Aufgaben:

a) Welche Städte in der Liste haben die höchste und die geringste Zahl an Einwohnern?
b) Erstellen Sie eine Liste aller Städte mit mindestens 100 000 Einwohnern (Großstädte). Die Liste soll absteigend nach der Einwohnerzahl sortiert sein.
c) Erstellen Sie eine Liste mit den Spalten *Bundesland* und *Anzahl der gelisteten Städte in dem Bundesland*. Die Liste soll nach der Anzahl der Städte absteigend sortiert sein.
d) Wie viele Menschen leben insgesamt in den aufgeführten Städten des Landes Rheinland-Pfalz?

Hinweise: (1) Die Datei `staedte.txt` ist eine ASCII-Textdatei. Daher sind deutsche Umlaute umschrieben, z. B. `ü` → `ue`; (2) um sicherzustellen, dass jede Zeile der Datei *genau drei* Spalten hat, wurden Leerzeichen in Städtenamen durch Unterstriche ersetzt, z. B. `Sankt␣Augustin` → `Sankt_Augustin`; (3) Quelle für die Datei `staedte.txt` ist die Wikipedia-Seite „Liste der Groß- und Mittelstädte in Deutschland“[1]. Die Datei wurde 2014 aus Wikipedia extrahiert.
Lösungsvorschlag auf Seite 199!

Aufgabe 14.7

Erstellen Sie mithilfe einer Pipeline aus der Datei `exposures.txt` eine Liste mit den folgenden Informationen (Spalten): (1) Beobachtungsregion; (2) Anzahl der Konfigurationen, in denen diese Region beobachtet wurde.
Lösungsvorschlag auf Seite 200!

Aufgabe 14.8

Diese Aufgabe bezieht sich wieder auf die Daten in der Datei `exposures.txt`. Sie erfahren, dass Sie für jede Region, die in der Konfiguration `i` beobachtet wurde, mindestens sieben verschiedene Beobachtungen in dieser Konfiguration benötigen. Die Daten sollen zusätzlich folgende Anforderungen erfüllen: (1). Ein Qualitätsmaß von höchstens 0.85 (fünfte Spalte) und (2). Beobachtungen unter guten oder sehr guten Wetterbedingungen (eine Zahl

[1] (siehe https://de.wikipedia.org/wiki/Liste_der_Gro%C3%9F-_und_Mittelst%C3%A4dte_in_Deutschland#Gro.C3.9F-_und_Mittelst.C3.A4dte_nach_Einwohnerzahl).

kleiner als drei in der sechsten Spalte von `exposures.txt`). Sie dürfen annehmen, dass Sie für jede Region bereits mindestens eine Beobachtung haben, die obige Bedingungen erfüllt.

Erstellen Sie eine Liste der Regionen, die erneut beobachtet werden müssen. Neben den Regionen selbst soll Ihre Liste auch die Anzahl der noch durchzuführenden Beobachtungen für jede Region erhalten.

Lösungsvorschlag auf Seite 201!

Aufgabe 14.9

Sie bekommen zusätzlich zur Datei `exposures.txt` eine Datei `verfuegbar.txt`. Letztere enthält zusätzlich zu Aufnahmen, die bereits in `exposures.txt` aufgeführt sind, auch neue Daten, die Sie noch nicht haben. In `verfuegbar.txt` sind nur die Beobachtungs-IDs aufgeführt. Finden Sie heraus, welche Aufnahmen Sie noch nicht haben.

Hinweis: Anders formuliert lautet die Aufgabe: Finden Sie alle Beobachtungen, die in `verfuegbar.txt`, aber nicht in `exposures.txt`, gelistet sind.

Lösungsvorschlag auf Seite 202!

Aufgabe 14.10

Die Aufgabe „Für welche Beobachtungsregionen liegen Beobachtungen in den drei Konfigurationen `g`, `r` und `i` vor?“, die im Text ausführlich behandelt wurde, ist nicht eindeutig formuliert. Unsere Lösung der Aufgabe wäre nicht korrekt, wenn man die Aufgabe als *genau in den drei Konfigurationen `g`, `r` und `i`* lesen würde. Regionen, die zwar Beobachtungen in `g`, `r` und `i` haben, aber zusätzlich in den Konfigurationen `u` und/oder `z` Daten besitzen, wären dann keine gültigen Treffer mehr! Bearbeiten Sie auf Basis unserer Textlösung das Problem Beobachtungen *genau in den drei Konfigurationen `g`, `r` und `i`* .

Hinweis: Die Aufgabe ist mit den im Text behandelten Kommandos lösbar. Einfacher ist jedoch die Verwendung des Kommandos `comm`. Erkundigen Sie sich darüber.

Lösungsvorschlag auf Seite 203!

Aufgabe 14.11

Betrachten Sie die Datei `moby_dick.txt` und die Pipeline:

```
user$ cat moby_dick.txt | tr -cs '[:alpha:]' '\n' |\
      sort | uniq -c | sort -nr -k1,1
```

Beschreiben Sie, wie die Pipeline funktioniert und was ihr Zweck ist.

Hinweise: (1) Bearbeiten Sie zuerst Aufgabe 14.4, falls Sie das noch nicht getan haben; (2) Quelle für die Datei `moby_dick.txt` ist das *Projekt Gutenberg*, welches zahlreiche freie e-Books zur Verfügung stellt; siehe http://www.gutenberg.org/ebooks/2701.

Lösungsvorschlag auf Seite 204!

Shell-Skripte 15

Übersicht

Zusammenfassung

Shell-Skripte erlauben es, mehrere *Unix*-Kommandos zu kombinieren und sich so effektiv *neue*, leistungsfähige Befehle zu programmieren. Lange und oft benötigte Befehlsfolgen müssen so nicht mehr *immer wieder* per Hand eingegeben werden, sondern können komfortabel durch den Aufruf eines Shell-Skripts abgearbeitet werden. Dies erleichtert einerseits die tägliche Arbeit mit *Unix*, und ist andererseits ein zentraler Bestandteil für effizientes wissenschaftliches Arbeiten. Letzteres erfordert es z. B. häufig, ähnliche und einfach zu reproduzierende Analysen mit vielen Datensätzen durchzuführen. Dieses Kapitel führt in die Grundlagen der Shell-Skriptprogrammierung ein.

15.1 Das Hallo-Welt-Shell-Skript

Ein *Hallo-Welt-Programm* soll auf möglichst einfache Weise ein lauffähiges Programm in einer neu zu erlernenden Computersprache liefern. Es dient üblicherweise dazu, sich mit grundlegenden Elementen einer Programmiersprache vertraut zu machen. Das Hallo-Welt-Programm tut in den meisten Fällen nichts anderes, als die Zeichenkette *Hallo Welt!* auf dem Bildschirm auszugeben. Wir wollen in diesem Abschnitt schrittweise eine etwas fortgeschrittenere Version schreiben, um uns einige Techniken von Shell-Skripten anzueignen.

Die grundlegende Idee von Shell-Skripten haben wir bereits in Abschn. 10.2 kennengelernt. Man schreibt *Unix*-Befehle in eine Textdatei und lässt diese von der

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0_15

Shell ausführen. Damit können wir bereits eine erste Version unseres Hallo-Welt-Skripts erstellen:

```
user$ cat hallo_erste_Version.sh
echo "Hallo Welt!"
user$ bash hallo_erste_Version.sh
Hallo Welt!
```

Bash-Shell-Skripte sind üblicherweise ASCII-Textdateien mit der Endung `.sh`. Wie Befehle in solchen Dateien mithilfe des `bash`-Befehls ausgeführt werden, habe ich Ihnen schon in Abschn. 10.2 gezeigt. Wir haben in Abschn. 3.3 besprochen, dass es außer der *Bash* noch andere Shell-Programme gibt. Bekannte Vertreter sind die *C-Shell*, die *Korn-Shell* oder die *Z-Shell*. In all diesen Programmen ist es ebenfalls möglich, Shell-Skripte zu schreiben. Skripte dieser Shell-Varianten erkennt man üblicherweise an Dateiendungen wie `.csh`, `.ksh` oder `.zsh`. Wenn wir in diesem Buch von Shell-Skripten sprechen, so beziehen wir uns *immer* auf die *Bash*-Shell!

15.1.1 Shell-Skripte als ausführbare Programme

Neben der indirekten Ausführung eines Shell-Skripts mit dem `bash`-Kommando können wir auch ein direkt ausführbares Programm erstellen:

```
user$ cat hallo_zweite_Version.sh
#!/bin/bash

echo "Hallo Welt!"
user$ chmod +x hallo_zweite_Version.sh
user$ ./hallo_zweite_Version.sh
Hallo Welt!
```

Erklärung: Die Zeichenfolge `#!`, gefolgt von einem absoluten Programmpfad *ganz am Anfang* einer Skriptdatei, wird als *Shebang-Zeile* bezeichnet. Der Name *Shebang* für diese Zeile kommt wahrscheinlich von einer *Verkettung* der englischen Wörter *haSH* und *bang*. Im *Unix*-Jargon wird das Doppelkreuz oft als *hash* und das Ausrufezeichen als *bang* bezeichnet. Die Shebang-Zeichenfolge `#!` löst bei einem Programmaufruf der Skriptdatei folgende Aktionen aus: (1) Der Rest der Zeile (hier `/bin/bash`) wird als absoluter Pfad eines Programms interpretiert; (2) das Programm wird ausgeführt, wobei die gesamte, zum anfänglichen Skriptaufruf verwendete, Kommandozeile als Argument übergeben wird. Im vorliegenden Fall führt somit unser Aufruf von `./hallo_zweite_Version.sh` effektiv zur Abarbeitung des Befehls `/bin/bash ./hallo_zweite_Version.sh`!

Hier noch einige Bemerkungen zum Shebang-Mechanismus:

1. Falls der absolute Pfad des `bash`-Programms bei Ihnen nicht `/bin/bash` sein sollte, so müssen Sie im Folgenden die Shebang-Zeile immer entsprechend anpassen.
2. Vielleicht sind Sie über den Shebang-Mechanismus überrascht, da das Doppelkreuz in der Shell eigentlich einen Kommentar einleitet und die Shell alles überliest, was nach diesem Zeichen folgt. In der Tat hat die Zeichenkombination `#!` *nur dann* die Shebang-Sonderbedeutung, wenn es die ersten beiden Zeichen in der ersten Zeile einer Skriptdatei sind! Des Weiteren kommt die Sonderbedeutung *nur dann* zum Tragen, wenn die Skriptdatei direkt als Programm ausgeführt wird. Bei einem indirekten Aufruf, wie bei `bash hallo_zweite_Version.sh`, wird die Shebang-Zeile in der Tat einfach als Kommentar betrachtet! Man beachte, dass der Shebang-Mechanismus letztendlich zu einem indirekten Aufruf des Skripts mit dem `bash`-Kommando führt. Bei diesem *zweiten* Aufruf des Skripts wird die Shebang-Zeile überlesen und der Rest des Skripts ausgeführt.
3. Der Shebag-Mechanismus wird von allen *Unix*-Shells und vielen anderen Programmiersprachen (z. B. *Perl* oder *Python*) genutzt. Daher ist in all diesen Sprachen das Doppelkreuz Kennzeichen für eine Kommentarzeile.
4. Shell-Skripte, die mit dem Shebang-Mechanismus zu ausführbaren Programmen gemacht wurden, unterliegen den in Abschn. 9.1 vorgestellten Mechanismen zur Programmausführung.

15.1.2 Variablen und die Kommandosubstitution

In Shell-Skripten sind Variablen zentrale Bestandteile. Bisher haben wir in Kap. 10 gesehen, wie wir Variablen Werte zuweisen und wie wir diese Werte wieder auslesen können. Daneben kann man Variablen auch bei der Befehlsausführung benutzen:

```
user$ ls
test.txt
user$ ALTER_NAME=test.txt
user$ NEUER_NAME=test.txt.copy
user$ cp ${ALTER_NAME} ${NEUER_NAME}
user$ ls
test.txt test.txt.copy
```

Erklärung: Die Auswertung der Variablen `ALTER_NAME` und `NEUER_NAME` findet *vor* der Ausführung des `cp`-Befehls statt! Effektiv wird oben somit der Befehl `cp test.txt test.txt.copy` ausgeführt!

Das Beispiel ist an sich natürlich nicht sehr sinnvoll und dient lediglich der Demonstration. Deutlich interessanter sind solche Konstrukte, wenn der Wert der Variablen in Shell-Skripten *variieren* kann und nicht von vorneherein feststeht. Dazu in Abschn. 15.2 mehr.

Das Dollarzeichen (`$`) ist allgemein ein Shell-Sonderzeichen, das in Verbindung mit einem Ausdruck in Klammerpaar(en) zur *Auswertung* des eingeschlossenen Ausdrucks führt. Neben der Variablenauswertung ist hier die sogenannte *Kommandosubstitution* von zentraler Bedeutung für Shell-Skripte. Hier ein Beispiel zum Einstieg:

```
# Auswertung der Shell-Variablen USER mit dem
# ${...}-Konstrukt:
user$ echo "Mein Benutzername ist ${USER}"
Mein Benutzername ist thomas
    # Anstatt 'thomas' erscheint hier
    # natürlich Ihr Benutzername

# Auswertung des Kommandos pwd mit dem
# $(...)-Konstrukt (einer Kommandosubstitution):
user$ echo "Ich befinde mich im Verzeichnis $(pwd)"
Ich befinde mich im Verzeichnis /home/thomas
```

Erklärung: Der erste Befehl mit der Variablenauswertung sollte klar sein. `USER` ist eine vordefinierte Umgebungsvariable, die den Benutzernamen enthält. Nun zur Kommandosubstitution: Das `$(...)`-Konstrukt führt bei dem zweiten `echo`-Kommando zunächst den *Unix*-Befehl innerhalb der runden Klammern aus (hier `pwd`) und ersetzt sich danach durch dessen `STDOUT`-Ausgabe. Wird das Konstrukt als Teil eines Befehls verwendet, so findet die Kommandosubstitution, genauso wie die Variablenauswertung, *vor* der Ausführung des Befehls statt. Die Kommandosubstitution erlaubt es somit, die Ausgabe eines Befehls als Argument eines anderen Befehls zu verwenden!

Hier zwei weitere Beispiele mit Praxisrelevanz:

1. Wir wollen Dateien löschen, welche selbst in einer Textdatei aufgelistet sind:

```
user$ cat files_to_delete.txt
./test.txt
./test/Dokument.tex

# Wir überprüfen, ob diese Dateien noch vorhanden
# sind:
user$ ls $(cat files_to_delete.txt)
./test/Dokument.tex  ./test.txt

# Lösche die Dateien:
user$ rm $(cat files_to_delete.txt)
user$ ls $(cat files_to_delete.txt)
ls: cannot access ./test.txt: No such ..
ls: cannot access ./test/Dokument.tex: ..
```

Hier liefert die Kommandosubstitution `$(cat files_to_delete.txt)` eine Liste der in `files_to_delete.txt` aufgeführten Dateien. Diese Liste wird jeweils zu Argumenten für `ls` und `rm`.

Hinweis: In Aufgabe 13.5 haben Sie das Kommando `xargs` kennengelernt. Es erlaubt über ein Pipelinekonstrukt ebenfalls, die Ausgabe eines Befehls als Argument eines anderen zu verwenden. Es ist daher nicht überraschend, dass etliche Aufgaben, die man mit einer Kommandosubstitution lösen kann, auch durch eine Pipeline mit `xargs` erledigt werden können. Eine Pipelinevariante für obiges Problem wäre:

```
user$ cat files_to_delete.txt | xargs rm
    # Löscht alle Dateien, die in
    # 'files_to_delete.txt' aufgelistet sind
```

2. Hin und wieder möchte man Datumsangaben in einen Dateinamen integrieren. Dies ist z. B. für Backups oder Log-Dateien sehr sinnvoll. Mit einer Kommandosubstitution kann dies folgendermaßen realisiert werden:

```
user$ echo "Text in der Log-Datei" > \
   log_$(date +'%F').txt
user$ ls
log_2016-04-07.txt
```

Hier integriert die Kommandosubstitution die Ausgabe des Befehls `date +'%F'` (eine Datumsangabe) in den Namen der zu erstellenden Datei.

Bemerkung: Vor allem die *Linux*-Variante des `date`-Kommandos ist *sehr* mächtig und kann bei vielen Problemen, die mit Datumsangaben zu tun haben, weiterhelfen. Leider sind vor allem die sehr nützlichen Möglichkeiten der `-d STRING` (*display date specified by STRING*)-Option in den `man`-pages des Kommandos nicht dokumentiert. Deshalb gebe ich hier einige Beispiele, um die Möglichkeiten anzudeuten:

```
# Das jetzige Datum und die Zeit anzeigen:
user$ date
Thu Apr  7 19:46:19 CEST 2016

# Datums- und Zeitanzeigeformat spezifizieren
# (siehe man-pages)
user$ date +'%F_%T'
2016-04-07_19:47:09

# morgiges Datum anzeigen:
user$ date -d 'tomorrow' +'%F'
2016-04-08
```

```
# noch einige Datumsangaben:
user$ date -d 'last Monday' +'%F'
2016-04-04
user$ date -d '+100 days' +'%F'
2016-07-16
user$ date -d '2000-03-01 +12 days' +'%F'
2000-03-13
```

Bemerkungen:

1. Das `$(...)`-Konstrukt für die Kommandosubstitution ist erst in neueren Versionen der *Bash*-Shell eingeführt worden. Früher hat man dafür einfache, rückwärts geneigte Hochkommas verwendet:

   ```
   user$ ls `cat files_to_delete.txt`
       # Dies ist äquivalent zu obigem
       # ls $(cat files_to_delete.txt)
   ./test.txt ./test/Dokument.tex
   ```

   Obwohl die Form mit den Hochkommas noch viel verwendet wird, und vor allem in älteren Skripten zu finden ist, sollte sie nicht mehr benutzt werden. Das `$(...)`-Konstrukt reiht die Kommandosubstitution in natürlicher Weise in andere Auswertungsmechanismen mit dem Dollarzeichen ein. Auch gibt es mit der Hochkommavariante einige Probleme, auf die ich hier aber nicht eingehen möchte.
2. Es sei explizit darauf hingewiesen, dass eine Kommandosubstitution *beliebige* Befehle verarbeiten kann, z. B. auch solche mit Pipelinekonstrukten:

   ```
   user$ echo "Es sind $(ls -1 | wc -l) Einträge \
   in $(pwd)"
   Es sind 182 Einträge in /home/thomas/Unix_Buch
   ```

3. Falls das Ergebnis einer Kommandosubstitution mehrmals benötigt wird, so kann man es in einer Variablen *zwischenspeichern*:

   ```
   user$ ZEIT=$(date +'%T')
   user$ echo "Die Variable ZEIT enthält: ${ZEIT}"
   Die Variable ZEIT enthält: 01:27:02
   user$ echo "ZEIT wurde um ${ZEIT} definiert."
   ZEIT wurde um 01:27:02 definiert.
   ```

Aufgabe 15.1

Was geschieht bei folgendem Befehl:

```
user$ tar -czvf backup_$(date +'%F').tgz \
   $(find ${HOME} -name \*.txt -type f)
```

Der `tar`-Befehl zum Archivieren von Dateien ist in Anhang B beschrieben.
Lösungsvorschlag auf Seite 205!

Aufgabe 15.2

Erklären Sie, warum folgender Befehl zu einer Fehlermeldung führt:

```
echo "Text in der Log-Datei" > \
   log_$(date +'%D').txt
bash: log_$(date +'%D').txt: No such file ...
```

Lösungsvorschlag auf Seite 205!

Zum Schluss dieses Abschnitts wollen wir unser bisheriges Hallo-Welt-Skript zu einem kleinen Report über den derzeitigen Status unseres Benutzer-Accounts und unserer Maschine erweitern:

```
user$ cat status.sh
#!/bin/bash

# Das Skript gibt einige Statusinformationen zu
# unserem Benutzer-Account und zu der Maschine, auf
# der wir eingeloggt sind.

echo "Hallo ${USER}!"
echo ""     # schreibe eine Leerzeile

# Das Kommando hostname gibt Information über
# die Maschine, auf der man eingeloggt ist.
echo "Sie sind auf $(hostname) eingeloggt."
echo "Ihr Heimatverzeichnis ist: ${HOME}"
echo "Ihr cwd ist $(pwd)"
user$ chmod +x status.sh
user$ ./status.sh
Hallo thomas!

Sie sind auf aibn234 eingeloggt.
Ihr Heimatverzeichnis ist: /home/thomas
Ihr cwd ist /home/thomas/Unix_Buch
```

Ich möchte hier noch einmal explizit darauf hinweisen, dass alles, was in einer Zeile *nach* einem Doppelkreuz steht, von der Shell bei der Ausführung *nicht* beachtet wird. Dies erlaubt es Ihnen, Ihre Shell-Skripte zu *kommentieren*, was Sie auch tun sollten! Kommentare helfen Ihnen unter anderem, ein Skript schnell wieder zu verstehen, wenn Sie es längere Zeit nicht genutzt haben.

Aufgabe 15.3

Erweitern Sie das Shell-Skript `status.sh` um Informationen über die gegenwärtig eingeloggten Benutzer auf Ihrer Maschine. Hier ein Beispielaufruf meines modifizierten Skripts:

```
user$ ./status.sh
Hallo thomas!

Sie sind auf aibn234 eingeloggt.
Ihr Heimatverzeichnis ist: /home/thomas
Ihr cwd ist /home/thomas/Unix_Buch
Folgende 2 Nutzer sind eingeloggt:
terben
thomas
```

Hinweis: Eine Möglichkeit, um an die Namen eingeloggter Benutzer zu kommen, ist das Kommando `who`.
Lösungsvorschlag auf Seite 206!

15.2 Die `for`-Schleife

Schleifen sind in Programmiersprachen ein zentrales Instrument, um eine Aufgabe, bzw. Teile eines Programms wiederholt auszuführen. Dabei sollen verschiedene Aufrufe des betreffenden Programmteils meist mit *veränderten Parametern* durchgeführt werden. Als Beispiel haben Sie mehrere Textdateien wie z. B. `text1`, `text2`, `text3` und `text4`, bei deren Namen die Endung `.txt` fehlt. Sie wollen diese Dateien umbenennen und die Endung `.txt` an den Namen anhängen. Bisher können Sie diese Aufgabe nur mit einer Folge von `mv`-Kommandos lösen:

```
user$ mv text1 text1.txt
user$ mv text2 text2.txt
.
.
```

Falls Sie deutlich mehr als vier Dateien haben sollten, ist das manuelle Eingeben der `mv`-Befehle nicht mehr praktikabel. Hier ist also *dieselbe* `mv`-Operation mehrmals

durchzuführen, wobei der Dateiname der veränderliche Parameter ist. Zur Lösung dieser Probleme stellt die *Bash* zwei *Schleifenkonstrukte* zur Verfügung, von denen ich mich auf die for-Schleife beschränken möchte. Eine Lösung zum Umbenennen der Dateien mit einer for-Schleife in einem Shell-Skript text_copy.sh wäre:

```
user$ ls
text1  text2  text3  text4
user$ cat text_copy.sh
for DATEI in text1 text2 text3 text4
do
  mv ${DATEI} ${DATEI}.txt
done
user$ bash text_copy.sh
user$ ls
text1.txt  text2.txt  text3.txt  text4.txt
```

Erklärung: Die allgemeine Syntax der for-Schleife ist:

```
for VARIABLE in LISTE
do
  BEFEHLE
done
```

Das Konstrukt führt mehrere *Schleifendurchläufe* mit den zwischen do–done eingebetteten Befehlen durch. Die erste Zeile for VARIABLE in LISTE enthält einen frei wählbaren Namen für eine Shell-Variable VARIABLE. VARIABLE wird vor jedem Schleifendurchlauf mit einem Element der Liste LISTE initialisiert, und BEFEHLE werden mit diesem aktuellen Wert von VARIABLE abgearbeitet. Die Schleife ist beendet, sobald alle Elemente von LISTE aufgebraucht sind. In folgendem Beispiel

```
for DATEI in text1 text2 text3 text4
do
  mv ${DATEI} ${DATEI}.txt
done
```

besteht LISTE aus den vier Dateinamen text1,..., text4 (Elemente der Liste sind in dem for-Schleifen-Konstrukt durch <Space> oder <Tab> voneinander getrennt). Die Variable DATEI wird nacheinander mit den Dateinamen initialisiert und der Befehl mv ${DATEI} ${DATEI}.txt damit durchgeführt.

Was Sie momentan sonst noch über die for-Schleife wissen müssen, fasse ich in einer Aufzählung zusammen:

1. Eventuell fragen Sie sich gerade, was das Schleifenkonstrukt *überhaupt bringt*. Das Problem, im obigen Beispiel viele mv-Befehle eingeben zu müssen, ist

momentan auf die Erstellung der von der Schleife abzuarbeitenden Liste verschoben. Haben wir etwa 1000 Dateien, die umbenannt werden sollen, so wäre mit der `for`-Schleife nicht viel gewonnen, wenn wir die Liste per Hand eingeben müssten! Die notwendigen Listen können jedoch sehr effektiv mit Wildcards (bei Dateinamen) oder allgemeiner durch Kommandosubstitutionen erzeugt werden:

```
user$ cat text_copy_zweite.sh
# Die Liste der for-Schleife kann hier
# mit Wildcards erzeugt werden, da es sich
# um Dateinamen handelt:
for DATEI in text*
do
  mv ${DATEI} ${DATEI}.txt
done

user$ cat text_copy_dritte.sh
# Hier eine Version mit Listenerzeugung
# durch Kommandosubstitution:
user$ cat files_to_rename.txt
text1
text2
.
user$ cat text_copy_dritte.sh
for DATEI in $(cat files_to_rename.txt)
do
  mv ${DATEI} ${DATEI}.txt
done
```

2. Wir haben in unseren Beispielen den Befehlsblock zwischen `do` und `done` mit Leerzeichen *eingerückt*. Dies ist nicht zwingend notwendig, macht ein Skript aber *deutlich* leichter lesbar. Blockstrukturen werden mit einer Einrückung auf einen Blick sichtbar:

```
user$ cat text_copy_demo.sh
# for-Schleife ohne Einrückung des Befehlsblocks
for DATEI in text*
do
echo "Benenne ${DATEI} in ${DATEI}.txt um"
mv ${DATEI} ${DATEI}.txt
done

# Mit Einrückung ist auf einen Blick klar, 'was'
# zum Befehlsblock der Schleife gehört:
for DATEI in text*
```

```
do
  echo "Benenne ${DATEI} in ${DATEI}.txt um"
  mv ${DATEI} ${DATEI}.txt
done
```

3. Man kann eine for-Schleife auch direkt auf der Kommandozeile nutzen. Die einzelnen Befehlsteile müssen dann durch Strichpunkte voneinander getrennt werden:

```
user$ ls text*
text1  text2  text3  text4
user$ for DATEI in text*; do echo \
   "Benenne ${DATEI} in ${DATEI}.txt um"; \
   mv ${DATEI} ${DATEI}.txt; done
Benenne text1 in text1.txt um
.
.
```

   Strichpunkte sind vor dem do, das den Befehlsblock einleitet, zwischen den einzelnen Befehlen des Befehlsblocks und dem abschließenden done nötig.

Zum Abschluss noch ein weiteres Beispiel zur Vertiefung. Wir wollen aus unserer Datei exposures.txt die Aufnahmen aus den Regionen D1, D2, D3 und D4 extrahieren und in jeweils einer eigenen Datei abspeichern. Die zu erstellenden Dateien sollen nur die Beobachtungs-IDs der entsprechenden Regionen enthalten (erste Spalte in exposures.txt). Die Namen der zu erstellenden Dateien sollen aus den Regionen und der Endung .txt bestehen, z. B. D1.txt. Ein Shell-Skript zur Lösung dieser Aufgabe wäre:

```
user$ ls
exposures.txt  regionen.sh
user$ cat regionen.sh
#!/bin/bash

for REGION in D1 D2 D3 D4
do
  grep ${REGION} exposures.txt | awk '{print $1}' > \
     ${REGION}.txt
done
user$ ./regionen.sh
user$ ls
D1.txt  D2.txt  D3.txt  D4.txt  exposures.txt
regionen.sh
user$ cat D1.txt
```

```
712914p
712915p
.
```

Es sollte Ihnen keine Probleme bereiten, die Funktionsweise des Skripts nachzuvollziehen.

Aufgabe 15.4

Sie haben Dateien der Form `text1.asc`, `text2.asc`, ..., `text100.asc`. Die Endung `.asc` wird manchmal verwendet, um zu betonen, dass es sich bei einer Textdatei um eine ASCII-Textdatei handelt. Schreiben Sie eine `for`-Schleife, welche die Dateien so umbenennt, dass die Endung `.asc` durch die Endung `.txt` ersetzt wird.

Hinweis: Sie haben hier das Problem, aus einem Dateinamen die Endung abspalten zu müssen, also z. B. aus der Zeichenkette `text1.asc` das `text1` zu isolieren. Eine Möglichkeit hierzu ist das Kommando `basename`.

Lösungsvorschlag auf Seite 207!

Aufgabe 15.5

Diese Aufgabe ist eine Ergänzung zum letzten `for`-Schleifen-Beispiel im Text. Wir haben aus der Datei `exposures.txt` die Beobachtungs-IDs einzelner Regionen extrahiert und in neue Dateien geschrieben. Hier wollen wir die Beobachtungs-IDs der Regionen `D1`–`D4` etwas genauer aufspalten. Wir wollen je eine separate Datei für jede Region *und* jede Beobachtungskonfiguration (letzte Spalte der Datei `exposures.txt`) erhalten. So soll beispielsweise die Datei `D1_i.txt` alle IDs enthalten, die in Region `D1` mit der Konfiguration `i` beobachtet wurden. Wir haben vier Regionen und die fünf Konfigurationen `u`, `g`, `r`, `i` und `z`. Erstellen Sie ein Skript, welches Dateien für die 20 möglichen Kombinationen von Region und Konfiguration erstellt.

Hinweis: Man kann mehrere `for`-Schleifen ineinander verschachteln!

Lösungsvorschlag auf Seite 208!

Aufgabe 15.6

Diese Aufgabe ist eine Verallgemeinerung von Aufgabe 14.6, Teil d). Schreiben Sie ein Shell-Skript, das für jedes in der Datei `staedte.txt` enthaltene Bundesland ermittelt, wie viele Menschen insgesamt in den aufgeführten Städten des jeweiligen Bundeslandes leben.

Lösungsvorschlag auf Seite 208!

15.3 Shell-Skripte als Wrapper für komplexere Analysen

Bisher habe ich Ihnen hauptsächlich Beispiele gezeigt, in denen Shell-Skripte mehrere *Unix*-Kommandos zusammengefasst haben, um wiederkehrende Aufgaben bequem erledigen zu können. Neben diesem Aufgabenkomplex sind Shell-Skripte für Sie vor allem interessant, um einzelne Datenanalyseprogramme zu kombinieren. Hierbei übernehmen Shell-Skripte oft auch die nötige *Kommunikation* zwischen einzelnen Komponenten Ihrer Analyseverfahren. Ein typischer Ablauf ist, dass ein erster Analyseschritt gewisse Textdateien als Eingabe benötigt, welche von *Unix*-Programmen bequem erstellt werden können. Der erste Analyseschritt liefert ein Ergebnis, das für den zweiten Schritt *angepasst* werden muss. Diese Transformation von Ergebnissen eines Teilschritts zur Weiterverarbeitung kann wiederum mit *Unix*-Werkzeugen sehr effektiv erledigt werden. Man bezeichnet Skripte oder Programme, die entweder andere Programme *verwalten* oder mehrere Teilkomponenten zu einem Gesamtprogramm zusammenfügen, als *Wrapper*.

Als Beispiel für diesen Komplex und einige nützliche Techniken für Wrapper-Skripte betrachten wir das Programm `prime.py`. Es ist ein einfaches *Python*-Skript, welches natürliche Zahlen auf die Primzahleigenschaft testet. Die zu überprüfenden Zahlen werden in einer Textdatei übergeben, und gefundene Primzahlen werden auf `STDOUT` ausgegeben:

```
user$ cat testzahlen.txt
3
10
19
51
user$ ./prime.py testzahlen.txt
3
19
```

Wir wollen schrittweise einen Shell-Skript-Wrapper zur komfortableren Benutzung dieses Programms schreiben. Eine naheliegende Anwendung ist das Testen aller natürlichen Zahlen bis zu einer gegebenen Obergrenze.

Hinweis: Außer der Zwei sind alle Primzahlen ungerade. Um unsere Skripte möglichst einfach und übersichtlich zu halten, verzichten wir im Folgenden der Einfachheit halber auf eine explizite Berücksichtigung der Zwei.

Eine erste Version eines Skript-Wrappers, der die Erstellung der nötigen Zahlentabelle übernimmt und das `prime.py`-Programm aufruft, wäre:

```
user$ cat prime_wrapper_erste.sh
#!/bin/bash

# Die Obergrenze fuer den Primzahltest ist
# das dritte Argument des folgenden 'seq'-Befehls.
# Beachte, dass wir nur ungerade Zahlen testen.
```

```
# Ausser der 2 ist keine gerade Zahl eine Primzahl!
seq 3 2 10000 > daten_tmp.txt

./prime.py daten_tmp.txt

# Loesche die temporaere, nicht mehr benoetigte,
# Zahlentabelle wieder:
rm daten_tmp.txt
user$ bash prime_wrapper_erste_erste.sh
3
.
.
9973
```

Der hier eingeführte `seq`-Befehl (*sequence*) bietet eine einfache Möglichkeit, um eine Sequenz von Zahlen zu erzeugen:

```
# Liste ganze Zahlen zwischen 1 und 3
user$ seq 1 3
1
2
3

# Liste ganze Zahlen zwischen 1 und 6 mit einem
# Inkrement von 2 (zweites Argument). Es werden
# immer nur Zahlen 'kleiner oder gleich' der
# Obergrenze gelistet
user$ seq 1 2 6
1
3
5
```

Um den Wrapper zu verbessern, wäre es wünschenswert, wenn wir die Obergrenze der zu testenden Zahlen nicht immer im Skript verändern müssten, sondern diese, wie bei anderen Programmen auch, als *Argumente* an das Skript übergeben könnten. Wie das funktioniert, sehen wir uns im Folgenden an.

15.3.1 Argumente für Shell-Skripte

Die grundlegende Verarbeitung von Argumenten, die an ein Shell-Skript übergeben werden, ist recht einfach. Alles Wesentliche können wir an folgendem Beispiel demonstrieren:

```
user$ cat argumente.sh
# Ein einfaches Skript, um die Handhabung von
# Shell-Skript-Argumenten zu demonstrieren:

echo "Das Skript heisst: ${0}"
echo "Dem Skript wurden ${#} Argumente uebergeben"
echo "Das erste Argument ist: ${1}"
echo "Das zweite Argument ist: ${2}"

echo "Hier die Werte aller Argumente:"

for ARG in ${@}
do
  echo ${ARG}
done

user$ ./argumente.sh 1 2 zahl thomas test
Das Skript heisst: ./argumente.sh
Dem Skript wurden 5 Argumente uebergeben
Das erste Argument ist: 1
Das zweite Argument ist: 2
Hier die Werte aller Argumente:
1
2
zahl
thomas
test
```

Erklärung: An ein Skript übergebene Argumente werden nach Skriptaufruf in den Variablen `1, 2, ..., N` zur Verfügung gestellt. Hierbei gibt die jeweilige Zahl an, um das *wievielte* Argument es sich handelt. Die Variable „`0`“ enthält den Skriptaufruf *ohne* die Argumente. Die Gesamtzahl der übergebenen Argumente *N* kann über die Variable „#“ erlangt werden. Praktisch ist die Variable „`@`“, welche eine *Liste* aller Argumente enthält und somit benutzt werden kann, um in einer `for`-Schleife über die Argumente zu iterieren.

Hiermit können wir eine bessere, zweite Version unseres `prime.py`-Wrappers schreiben:

```
user$ cat prime_wrapper_zweite.sh
#!/bin/bash

# Die gewuenschte Obergrenze fuer den
# Primzahltest wird hier als Argument
# uebergeben:
OBER=${1}
```

```
# Die Variable(!) '$' enthaelt die PID des
# Skriptprozesses und ist sehr gut fuer
# eindeutige Dateinamen geeignet!
seq 3 2 ${OBER} > daten_tmp.txt_${$}

./prime.py daten_tmp.txt_${$}

# Loesche die temporaere, nicht mehr benoetigte,
# Datentabelle wieder:
rm *_${$}

user$ ./prime_wrapper_zweite.sh 10000
3
.
.
9973
```

Erklärung: Ich speichere zunächst das übergebene Argument in der Variablen `OBER` zwischen. Dies ist nicht zwingend nötig, erlaubt es aber, das Argument, anstatt mit einer *nichtssagenden* Zahl, durch einen *aussagekräftigen* Namen anzusprechen. Dies wiederum erhöht, vor allem bei langen Skripten, signifikant die Lesbarkeit und Übersichtlichkeit. Es kommt Ihnen auch zugute, wenn Sie das Skript später modifizieren und z. B. neue Argumente hinzufügen. Dann brauchen Sie nicht im gesamten Skript zu überprüfen, ob die Zahlen noch mit entsprechenden Argumenten übereinstimmen, sondern Sie müssen lediglich am Skriptanfang die Variablenzuweisungen der Argumente modifizieren!

Neben der Behandlung der Zahlenobergrenze durch ein Skriptargument ist die zweite wesentliche Neuerung der Name der temporären Zahlendatei. Er hat sich seit der ersten Wrapper-Version von `daten_tmp.txt` zu `daten_tmp.txt_${$}` geändert. Die Variable(!) „$“ enthält die Prozessidentifikationsnummer (`PID`; siehe Abschn. 9.2.1) des Skriptprozesses. Da diese Zahl eine *eindeutige* Prozessidentifikation darstellt, ist sie ideal, um temporäre Dateien mit *eindeutigen* Namen zu erzeugen. Stellen Sie sich vor, dass Sie *gleichzeitig* mehrere `prime_wrapper_zweite.sh`-Instanzen laufen lassen. Wenn Sie nicht aufpassen, überschreibt ein Prozess unter Umständen die temporäre Datei eines anderen. Wenn Sie konsequent alle temporären Dateien eines Skripts mit der Endung `_${$}` versehen, sichern Sie sich gegen dieses Problem ab! Des Weiteren werden Sie *alle* temporären Dateien am Skriptende mit einem einfachen `rm *_${$}` wieder los!

Tab. 15.1 fasst noch einmal die Bedeutung der in diesem Abschnitt vorgestellten, speziellen Shell-Variablen zusammen. Sie enthält auch zwei recht nützliche Varianten, die in diesem Abschnitt nicht explizit besprochen wurden.

Tab. 15.1 Ergebnisse der Auswertung einiger in der Shell-Skript-Programmierung *sehr* nützlicher Shell-Variablen

Variable	Kurzbeschreibung
`${0}`	Name des aufgerufenen Skripts
`${i}`	Das i-te Skript-Argument ($i \geq 1$)
`${@}`	Liste *aller* Argumente (zur Iterierung in `for`-Schleifen nutzbar)
`${@:M:N}`	Liste von N-Argumenten, startend von Argument M, z. B. ergibt `${@:2:3}` eine Liste des zweiten, dritten und vierten Arguments (nicht im Text besprochen)
`${#}`	Anzahl der übergebenen Skriptargumente
`${!#}`	Liefert das *letzte* Argument (nicht im Text besprochen)
`${$}`	`PID` des Skriptprozesses

Aufgabe 15.7

Zu schreiben ist ein Shell-Skript zur Verallgemeinerung von Aufgabe 15.5. Die zu behandelnden Felder in dem neuen Skript sollen nicht mehr fix `D1–D4` sein, sondern als Argumente übergeben werden. Für ein Skript mit Namen `regionen_dritte.sh` könnten mögliche Aufrufe wie folgt aussehen:

```
# Behandlung von Region D1
user$ bash regionen_dritte.sh D1
.
# Behandlung von Regionen D3, W1p2p3 und W2m0m0;
user$ bash regionen_dritte.sh D3 W1p2p3 W2m0m0
.
```

Lösungsvorschlag auf Seite 209!

Aufgabe 15.8

Wir wollen ein Shell-Skript `backup.sh` für das Backup eines Verzeichnisses, samt dessen gesamtem Inhalt, schreiben.

- Das Backup soll mit dem `tar`-Befehl durchgeführt werden; siehe Anhang B und Aufgabe 15.1.
- Endprodukte des Skripts sollen die Backup-Datei und eine Log-Datei sein. Die Namen der Dateien sollen wie `backup_2016-04-17.tgz` und `backup_2016-04-17.log` sein, also das Datum des Backups enthalten. Die Log-Datei soll darüber Auskunft geben, welches Verzeichnis gesichert wurde, z. B.:

  ```
  user$ cat backup_2016-04-17.log
  Es wurde /home/thomas/Unix_Buch gesichert.
  ```

- Das Shell-Skript soll zwei Argumente haben: (1). das Verzeichnis, von dem ein Backup zu erstellen ist, und (2). das Verzeichnis, in dem die Backup-Datei und die Log-Datei am Ende des Skripts stehen sollen.

Schreiben Sie ein Skript mit obigen Spezifikationen.
Lösungsvorschlag auf Seite 210!

15.3.2 Prozesse parallelisieren

Die Ausführung unseres `prime.py`-Wrappers ist für große Mengen zu testender Zahlen bereits ein zeitaufwendiger Prozess:

```
user$ time ./prime_wrapper_zweite.sh 1000000 > \
   primzahlen.txt

real    0m16.851s
user    0m16.788s
sys     0m0.031s
```

Wird vor einen auszuführenden Befehl ein `time`-Kommando gesetzt, so wird nach Befehlsende die Rechenzeit, die der Prozess verbraucht hat, angezeigt. Von den erhaltenen Werten ist für uns momentan derjenige nach `real` interessant. Er gibt an, wie lange wir auf die Beendigung des `prime_wrapper_zweite.sh`-Prozesses warten mussten. In diesem Fall hat auf meiner Maschine die Primzahlprüfung aller natürlichen Zahlen bis zu einer Million ca. 17 Sekunden gedauert.

Exkurs

Parallelisierung von Prozessen

Auf modernen Computern können einzelne Prozesse praktisch nur noch durch *Parallelisierung* signifikant beschleunigt werden. Es ist aus technischen Gründen heute sehr schwierig, neue Prozessoren mit einer signifikant höheren Taktrate als ca. 3–4 GHz zu konstruieren. Das Hauptproblem ist, die bei so hohen Taktraten auftretende Wärmeentwicklung mit vertretbarem Aufwand in den Griff zu bekommen.

Bis zum Jahre 2005 wurden hauptsächlich Rechner mit *einer einzigen* Prozessoreinheit, auch als *Einzelkernprozessor* bekannt, hergestellt. Höhere Rechenleistung wurde bis zu diesem Zeitpunkt primär durch eine Erhöhung der Taktfrequenz erzielt. Als man hier auf oben erwähnte Probleme stieß, hat man die Leistungsfähigkeit von Rechnern durch die Implementation von sogenannten *Mehrkernprozessoren* erhöht. *Sehr* vereinfacht gesagt, stellen Mehrkernprozessoren mehrere, unabhängige Prozessoreinheiten mit der maximal möglichen Taktfrequenz zur Verfügung. In Personalcomputern sind heute Prozessoren mit zwei, vier oder acht *Kernen* üblich. Mehr zu dem Thema können Sie z. B.

auf dem Wikipedia-Eintrag zu Mehrkernprozessoren nachlesen (siehe https://de.wikipedia.org/wiki/Mehrkernprozessor).

Mehrere Kerne erlauben es zunächst, unabhängige Programme effektiv parallel zu betreiben. Komplizierter wird die Sache, wenn man mehrere Kerne zur Beschleunigung *eines ganz bestimmten* Prozesses verwenden möchte. Im einfachsten Fall kann man zu analysierende Daten auf *unabhängige* Prozesse aufteilen, und hier kann unter Umständen schon ein Shell-Skript-Wrapper weiterhelfen.

Im Folgenden wollen wir eine dritte Version unseres `prime.py`-Wrappers erstellen, der die Primzahlsuche parallelisiert und von einem Mehrkernprozessor optimalen Gebrauch macht. Bei den Dateien zu diesem Kapitel finden Sie das *Python*-Skript `ncpus.py`, mit dem Sie auf der Kommandozeile die Anzahl der Kerne Ihrer Maschine bestimmen können:

```
# Das Python-Skript 'ncpus.py' zeigt die auf Ihrem
# Rechner vorhandenen Recheneinheiten bzw. Kerne:
user$ ./ncpus.py
4
```

Meine Maschine hat vier Kerne. Dies bedeutet, dass ich insgesamt vier Instanzen des `prime.py`-Programms gleichzeitig *mit etwa voller Leistung* laufen lassen kann – vorausgesetzt, ich habe keine anderen, rechenintensiven Prozesse auf meinem Rechner. Hier ein kleines Skript, welches die wesentlichen Ideen für eine Parallelisierung des `prime.py`-Prozesses illustriert:

```
user$ cat kern_test.sh
#!/bin/bash

# Das Skript startet einfach zwei identische
# Instanzen des prime.py-Skripts, um den Effekt
# eines Mehrkernsystems zu zeigen:

seq 3 2 1000000 > zahlen.txt

# Die folgenden 'prime.py'-Prozesse werden
# im Hintergrund gestartet, sodass sie 'parallel'
# ablaufen.
./prime.py zahlen.txt > prim1.txt &
./prime.py zahlen.txt > prim2.txt &

# 'Warte' auf alle gestarteten Hintergrundprozesse
wait

echo "Alles fertig!"
```

```
user$ time ./kern_test.sh
Alles fertig!

real    0m16.104s
user    0m31.614s
sys     0m0.012s
```

Erklärung: Das Skript startet im Hintergrund zwei *identische* `prime.py`-Aufrufe, die jeweils alle Primzahlen zwischen drei und einer Million liefern. Wie üblich, würden die Prozesse nacheinander ablaufen, falls wir sie nicht im Hintergrund starten würden. Der zweite `prime.py`-Aufruf würde erst dann seine Arbeit aufnehmen, wenn der erste beendet ist. So wird der erste Prozess im Hintergrund gestartet, das Skript direkt fortgesetzt und unmittelbar der zweite `prime.py`-Prozess initiiert. Die beiden Prozesse laufen also zur gleichen Zeit parallel ab! Das folgende `wait`-Kommando *wartet* mit der weiteren Skriptausführung, bis *alle* im Skript initiierten Hintergrundprozesse beendet sind! Im Allgemeinen möchten wir Ergebnisse der parallel ablaufenden Hintergrundprozesse weiterverwenden, und wir müssen daher an geeigneter Stelle auf diese Prozesse *warten*.

Die `real`-Zeitmessung für `kern_test.sh` zeigt, dass wir mit unserem *Mehrkernrechner* in der Tat für die Ausführung von zwei *parallelen* `prime.py`-Prozessen genauso lange warten mussten wie vorher für einen! Die Zeitangabe nach `user` entspricht *ungefähr* der komplett verbrauchten *Prozessorzeit* für unser Skript. Wie zu erwarten, entspricht dies in etwa der akkummulierten Zeit *der beiden* `prime.py`-Aufrufe. Sie ist ungefähr doppelt so hoch wie für einen dieser Prozesse. Die Rechenzeit, die andere Teile des Skripts verbrauchen, z. B. das `seq`-Kommando, sind in diesem Fall vernachlässigbar.

Genau dieselben Zahlen mehrfach auf die Primzahleigenschaft testen zu lassen, ist natürlich wenig sinnvoll. Um die gezeigte Parallelisierungstechnik gewinnbringend in unseren bisherigen `prime.py`-Wrapper zu integrieren, bietet es sich an, die zu testenden Zahlen *aufzuspalten* und verschiedene `prime.py`-Prozesse mit jeweils einem Teil der zu testenden Zahlen zu starten.

Eine Möglichkeit, eine Textdatei in mehrere Teile aufzuspalten, ist das `split`-Kommando:

```
user$ seq 1 10 > zahlen.txt
user$ split -n 3 zahlen.txt
    # splitte 'zahlen.txt' in drei,
    # 'etwa gleich große' Teile
user$ ls
xaa  xab  xac  zahlen.txt
user$ cat xaa
1
2
3
4
```

```
user$ rm xa*
user$ split -n r/3 zahlen.txt
   # Beachte das 'r/3' nach der '-n'-Option
   # Die Zahlen werden hier 'zyklisch'
   # auf drei Dateien verteilt!
user$ ls
xaa  xab  xac  zahlen.txt
user$ cat xaa
1
4
7
10
user$ cat xab
2
5
8
```

Hinweis: Testen Sie eventuell auf Ihrer *Unix*-Installation, ob das `split`-Kommando die Option `-n` (*number of output files*), wie hier gezeigt, unterstützt.

Wir haben jetzt alle Bestandteile zusammen, um unsere Ideen zur Parallelisierung des `prime.py`-Programms in einer dritten Version unseres Wrappers zu realisieren:

```
user$ cat prime_wrapper_dritte.sh
#!/bin/bash

# Dritte Version unseres prime.py-Wrappers.
# Das Skript nutzt einen Mehrkernprozessor und
# erledigt eine Aufgabe mit so vielen parallelen
# Prozessen, wie Kerne verfuegbar sind.

# Die gewuenschte Obergrenze fuer den Primzahltest
# wird als Argument uebergeben:
OBER=${1}

# Die Anzahl verfuegbarer Kerne auf unserer
# Maschine:
KERNE=$(./ncpus.py)

# Wir muessen die zu testenden Zahlen auf
# KERNE Dateien aufteilen und insgesamt KERNE
# prime.py-Prozesse im Hintergrund starten.
seq 3 2 ${OBER} > daten_tmp.txt_${$}

# Die Option '--additional-suffix' erlaubt es,
```

```
# an die Namen der Split-Dateien noch eine
# Endung anzuheften. Dies tun wir mit der PID:
split -n r/${KERNE} --additional-suffix=_${$} \
  daten_tmp.txt_${$}

# Starte die einzelnen prime.py-Prozesse:
for DATEI in $(ls xa*_${$})
do
  ./prime.py ${DATEI} > prim_${DATEI} &
done

# Warte, bis alle prime.py-Hintergrundprozesse
# beendet sind.
wait

# Fuege die einzelnen Ergebnislisten zu einer
# Primzahlliste zusammen und gib sie auf dem
# Bildschirm aus:
cat prim*_${$} | sort -n

# Loesche temporaere, nicht mehr benoetigte,
# Dateien wieder:
rm *_${$}
```

Mit unseren Vorarbeiten sollten Sie keine Probleme haben, das Skript nachzuvollziehen.

```
user$ time ./prime_wrapper_dritte.sh 1000000 > \
   primzahlen.txt

real    0m5.333s
user    0m16.284s
sys     0m0.088s
```

Wir sehen, dass die Gesamtprozessorzeit für die Primzahlrechnung nach wie vor bei ca. 16 Sekunden liegt (`user`-Zeile), aber wir nur noch etwas mehr als fünf Sekunden auf das Prozessende warten müssen! Dass ich hier auf meinem Vierkernrechner *nur* einen Gewinnfaktor von etwas mehr als drei habe, liegt primär daran, dass noch andere Prozesse (Editor, WWW-Borwser, File-Manager usw.) auf meinem System laufen und nicht die gesamte Leistung für `prime_wrapper_dritte.sh` zur Verfügung steht.

Aufgabe 15.9

Wir haben in unserem `prime_wrapper_dritte.sh`-Wrapper die zu testenden Zahlen auf mehrere Prozesse aufgespalten. Hierzu wurde der Befehl:

```
split -n r/${KERNE} ...
```

verwendet. Wie im Text besprochen, bewirkt dieser Befehl eine *zyklische* Verteilung der Zahlen auf die einzelnen Parallelprozesse. Können Sie einen Grund angeben, warum wir hier die *zyklische* Verteilung gewählt haben? Was würde passieren, wenn wir anstatt obigem `split`-Befehl die Variante `split -n ${KERNE} ...` verwenden würden?
Lösungsvorschlag auf Seite 211!

Zum Schluss dieses Abschnitts möchte ich betonen, dass das Beispiel primär dazu dient, Ihnen eine *sehr einfache* Möglichkeit für die Prozessparallelisierung mit Shell-Skripten zu zeigen. Ich habe oben in dem Exkurs zum Thema geschrieben, dass *Prozesse auf modernen Computern nur noch durch Parallelisierung signifikant beschleunigt werden können*. Hierin ist die implizite Annahme enthalten, dass die verwendeten Analyseverfahren bereits *optimal* sind. Generell sollten Sie, gerade bei langwierigen wissenschaftlichen Analysen, *vor* einer Parallelisierung stets überprüfen, ob eine Laufzeitoptimierung nicht durch eine Verbesserung der verwendeten Algorithmen erzielt werden kann. Wir haben in unserem Beispiel ein *sehr einfaches* Programm zum Testen der Primzahleigenschaft benutzt, um einen Wrapper für das *effektive* Problem „Finde alle Primzahlen bis zu einer gewissen Obergrenze“ zu entwickeln. Für dieses spezielle Problem gibt es aber *sehr viel effektivere* und schnellere Verfahren als das simple Testen aller Zahlen! Ein Beispiel ist das sogenannte *Sieb des Eratosthenes* (siehe z. B. den Wikipedia-Artikel https://de.wikipedia.org/wiki/Sieb_des_Eratosthenes).

Eventuell werden Sie unsere Techniken erweitern und verfeinern wollen. In unserem Beispiel konnten wir die zu analysierenden Daten (einzelne Zahlen) recht einfach auf N Kerne aufteilen und den gesamten Prozess mit insgesamt N Aufrufen des Analyseprogramms (hier `prime.py`) abschließen. Oft hat man aber z. B. M Datensätze, die nicht weiter zerlegt werden können und welche man mit $N < M$ Kernen analysieren muss. Eine Strategie ist, die Daten nacheinander in *Blöcken* zu N Prozessen abzuarbeiten, bis die M Datensätze aufgebraucht sind. Dies erfordert ein wenig *Verwaltungsaufwand*, der ohne größere Probleme in unsere Skripte integriert werden könnte.

Es gibt inzwischen allerdings auch etliche *Unix*-Programme, die Sie bei solchen fortgeschrittenen Parallelisierungen umfangreich unterstützen können. Erkundigen Sie sich bei Bedarf z. B. über das `GNU parallel`-Programm (siehe http://www.gnu.org/software/parallel/).

15.4 Zusammenfassung und Bemerkungen

Die Shell-Skripte erlauben es Ihnen, oft wiederkehrende Arbeitsschritte zusammenzufassen und Analyseverfahren zu koordinieren und zu verwalten. Mein Hauptziel war es, Ihnen notwendige und einführende Techniken zu zeigen, um dies auf erste eigene Probleme anwenden zu können. Gerade bei diesem Kapitel ist es wichtig, noch einmal zu betonen, dass es sich *nur um einen ersten Einstieg* in die Materie handelt und keinesfalls um eine umfassende Behandlung des Themas Shell-Skripte! Ich möchte dieses Kapitel mit einer Zusammenfassung und einigen Anregungen für Ihr weiteres Vorgehen beenden:

1. Shell-Skripte verdeutlichen sehr schön das *Unix-Baukastenprinzip*. Kommandos mit einem *sehr begrenzten* Funktionsumfang (denken Sie z. B. an die hier benutzten Programme `seq` und `split`) werden zu leistungsfähigeren Programmen mit erweitertem Funktionsumfang *kombiniert*. Einzelne Kommandozeilenprogramme mit einem sehr großen Funktionsumfang zu entwickeln, ist meist nicht nötig. Üblicherweise überprüft man, ob man nicht bereits vorhandene Programme *ergänzen* kann, um ein Problem zu lösen. Ein Vorteil dieser Herangehensweise ist, dass Programme mit einem kleinen Funktionsumfang ihre Sache in der Regel *sehr gut und effizient erledigen* und dass Sie bei der Nutzung dieser Programme in eigenen Skripten auf hochqualitative Komponenten zurückgreifen können. Wesentliche Techniken zur Kombination einzelner Komponenten sind unter anderem die Pipelines, Variablen, die Kommandosubstitution und Schleifenkonstrukte. Diese Grundelemente finden sich in allen *Unix*-Shell-Varianten wieder.
2. Sobald Sie selbst Programme entwickeln, sollten Sie darauf achten, dass sich diese in das *Unix*-Baukastenprinzip integrieren lassen. Konkret bedeutet dies für Programmein- und -ausgaben:
   - Datenausgaben sollten auf `STDOUT` erfolgen. Dies ermöglicht es Ihnen, Ihr Programm in eine Pipeline zu integrieren. Falls die Ausgabe in einer Datei benötigt wird, kann dies durch eine Ausgabeumlenkung realisiert werden. Ähnlich sollten Sie, wenn möglich, eine Dateneingabe über `STDIN` vorsehen.
   - Vermeiden Sie es, Ihre Datenausgaben auf `STDOUT` mit *Statusmeldungen* Ihres Programms zu vermischen. Dies erschwert die Verwendung Ihres Programms in Pipelines, da man *unwesentliche* Ausgaben hier wieder *herausfiltern* muss. Sie sollten *notwendige* Status- und Fehlermeldungen auf `STDERR` ausgeben oder eine Log-Datei verwenden.
3. Ich habe Ihnen in diesem Kapitel einige *Unix*-Kommandos zur Textmanipulation vorgestellt, wenn wir sie gebraucht haben. Sobald Sie Operationen durchführen müssen, für die Sie bisher keine Lösung haben, so starten Sie am einfachsten eine Web-Recherche über *Google*, und Sie finden für gewöhnlich sehr schnell eine Lösung. Sie werden sehen, dass für *sehr viele* Probleme der Textmanipulation Lösungen mit dem Programm `awk` existieren und im Netz vorgeschlagen

werden. Daher möchte ich hier wiederholen, dass sich eine nähere Beschäftigung damit für Sie auf alle Fälle lohnen wird!

4. Ich habe Ihnen hier nur *einen kleinen Teil* der Möglichkeiten und des Umfangs der *Bash*-Shell-*Programmiersprache* gezeigt. Sie werden sich recht schnell fragen, wie Sie einige Dinge, die hier nicht besprochen wurden, in Skripten verwirklichen können. Naheliegende Probleme wären „Wie teste ich, ob der Benutzer wirklich zwei Argumente übergeben hat?“ oder „Wie überprüfe ich, ob ein Pfad, der mir als Argument übergeben wurde, überhaupt existiert, und wie breche ich das Ganze im Bedarfsfall mit einer Fehlermeldung ab?“. Dies sind Probleme der *bedingten Programmausführung* (gewisse Programmteile werden nur unter bestimmten Bedingungen ausgeführt), für die es natürlich Shell-Konstrukte gibt. Mit einer Web-Recherche finden Sie zu diesem oder anderen Themen sehr schnell nötige Informationen. Mit dem hier vermittelten Wissen werden Sie keine Probleme haben, diese Webseiten oder Bücher zu dem Thema zu verstehen. Ich gebe Ihnen in Abschn. D.4 konkrete Empfehlungen für weitergehende Literatur zu dem Thema.

Anhang A
Reguläre Ausdrücke

Übersicht

Zusammenfassung

In diesem Anhang gebe ich eine Einführung in die sogenannten *Regulären Ausdrücke*. Sie erlauben es, Textdateien nach einer großen Vielfalt von Textmustern zu durchsuchen und entsprechende Textstellen eventuell zu modifizieren. Innerhalb dieses Buches können Sie sie unmittelbar mit dem Befehl `grep` benutzen. Jedoch werden Reguläre Ausdrücke auch von vielen anderen *Unix*-Programmen (z. B. `awk`) und allen gängigen Programmiersprachen direkt unterstützt. Eine Beschäftigung mit dem Thema zahlt sich für Sie also deutlich über die Grenzen der *Unix*-Werkzeuge hinaus aus!

A.1 Finden von Textmustern mit `grep`

Wir haben in Abschn. 14.1 das Kommando `grep` verwendet, um in Textdateien nach bestimmten, genau spezifizierten Zeichenketten zu suchen. Zum Beispiel hat der Befehl `grep W2m0m0 exposures.txt` alle Zeilen der Datei `exposures.txt` ausgegeben, welche die Zeichenkette `W2m0m0` enthalten. Das `grep`-Kommando kann aber nicht nur nach genau festgelegten Zeichenketten, sondern auch nach sehr allgemeinen *Textmustern* suchen. Bei diesen Mustern haben wir in der Regel keine *genaue* Zeichenkette, die wir finden wollen, sondern wir haben lediglich mehr oder wenig konkrete *Rahmenbedingungen*, welche Suchtreffer erfüllen müssen. Die betreffenden Muster können dann auf viele verschiedene konkrete Zeichenketten passen. Beispiele für allgemeine Textmuster, nach denen wir unter Umständen suchen wollen, sind:

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

1. Zeichenketten oder Worte in verschiedenen Schreibweisen, z. B.: Unix/UNIX oder Fotografie/Photographie.
2. Zeichenfolgen, die in genau definierter Art in einem Text *verankert* sein müssen. Zum Beispiel haben wir in Abschn. 14.1 nach Buchstaben `i` gesucht, die sich nach einem Leerzeichen am Ende einer Zeile befinden.
3. Beobachtungsregionen in unserer Datei `exposures.txt` (siehe Abschn. 12.3), z. B. `D1`, `D4`, `W1m1p2` oder `W3p2m1`. Zeichenketten für gültige Regionen erfüllen folgende Bedingungen: *Entweder* es folgt eine Zahl zwischen 1 und 4 einem großen „`D`“, *oder* es ist eine Folge von (1) einem großen „`W`“; (2) einer Zahl zwischen 1 und 4; (3) einem der Buchstaben `m` oder `p`; (4) einer Zahl zwischen 0 und 4; (5) einem der Buchstaben `m` oder `p`; (6) einer Zahl zwischen 0 und 4.
4. Telefonnummern sind Zahlenreihen, die bestimmte Einschränkungen haben. Wir wollen uns auf deutsche Festnetznummern mit Vorwahl beschränken. Gültige Telefonnummern sind dann z. B.: 089/413641, (0228) 7321496, 040-972591, 022381234567. Telefonnummern können als Textmuster aufgefasst werden, die unter anderem folgende Bedingungen erfüllen: (1) eine öffnende Klammer (oder kein Zeichen); (2) die Zahl 0; (3) eine Zahl zwischen 2 und 9 (eine zweite 0 in der Vorwahl würde eine internationale Vorwahl und eine 1 als zweite Ziffer eine Mobilfunknummer signalisieren!); (4) eine Folge von Zahlen (der Rest der Vorwahl); (5) kein Zeichen oder beliebig viele Leerzeichen; (6) kein Zeichen, eine schließende Klammer `)` oder ein Slash `/`; (7) kein Zeichen oder beliebig viele Leerzeichen; (8) eine Folge von Zahlen (die Hauptnummer). Wir belassen es bei diesen Rahmenbedingungen für ein Telefonzahlensuchmuster. Wenn man möchte, findet man leicht weitere Bedingungen, wie z. B. Unter- und Obergrenzen der Ziffernanzahl bei (4) und (8). Wir vernachlässigen auch weitere übliche Arten zur Telefonnummerndarstellung, z. B. die *internationale Darstellung* +49-89413641, Spezialnummern wie 110 (Polizeiruf) oder Telefonnummern mit *Durchwahl* wie 0228/62-5143.
5. Beim Schreiben eines längeren Textes macht man oft oft den Fehler, dasselbe Wort direkt hintereinander zweimal zu schreiben, so wie hier hier. Ein Textsuchmuster, das automatisch nach Textstellen sucht, an denen ein und dasselbe Wort direkt zweimal hintereinander steht, kann hierbei sehr hilfreich sein.

Die Textmuster, die wir zur Lösung derartiger Probleme verwenden, werden als *Reguläre Ausdrücke* (im Folgenden kurz „RA“) bezeichnet. Die grundlegende Idee zum Aufbau von RA haben Sie schon in Abschn. 5.5 bei den Wildcards kennengelernt:

```
user$ ls *.txt
...
```

Das Muster `*.txt` repräsentiert hier *alle Zeichenketten/Dateien*, die auf `.txt` enden. Wildcards sind eine Folge von *Metazeichen* (hier der `*`), die für eine bestimmte *Menge* an Zeichen stehen (hier: *eine beliebige Anzahl jedes beliebigen* Zeichen) und Zeichen, die für sich selbst stehen (hier „`.txt`“). Reguläre Ausdrücke

zum Finden von Mustern in Texten sind sehr ähnlich aufgebaut. Allerdings sind hier mehr und mächtigere Metazeichen vorhanden. Auch stehen Wildcards jeweils für Dateinamen, womit Probleme wie *Verankerung innerhalb eines Textes* gar nicht erst auftreten.

Exkurs

Dialekte von Regulären Ausdrücken (`grep` ist nicht immer gleich `grep`)

Bei den Regulären Ausdrücken gibt es viele verschiedene Dialekte und Implementationen. Die drei wichtigsten Gruppen sind *Basic Regular Expressions (BRE)*, *Extended regular expressions (ERE)* und *Perl Compatible Regular Expressions (PCRE)*. Diese Vertreter spiegeln im Wesentlichen die Entwicklung Regulärer Ausdrücke wider. Die BRE wurden zuerst entwickelt und haben den kleinsten Funktionsumfang. Sie orientierten sich an den begrenzten Ressourcen für Speicher und Rechenleistung der ersten Computer mit einem *Unix*-Betriebssystem. Die PCRE, welche mit der Programmiersprache *Perl* entwickelt wurden, stellen den neuesten Entwicklungsstand unter Berücksichtigung jahrelanger Erfahrungen mit den BRE und den ERE dar. Auch sind die Grenzen bei vielen Implementationen nicht klar gezogen. So sind z. B. im BRE-Funktionsumfang von `grep` unter *Linux* Elemente enthalten, die *formal* nur den ERE zustehen. Der Funktionsumfang der BRE und ERE ist bei `grep` unter *Linux* effektiv derselbe. Die Dialekte unterscheiden sich allerdings in der Syntax! Es ist daher wichtig, dass Sie gegebenenfalls die Dokumentation eines Programmes studieren, um den genauen Funktionsumfang einer speziellen RA-Implementation zu erfahren. Beachten Sie bitte auch, dass sich der Funktionsumfang von z. B. `grep` selbst unter anderen *Unix*-Systemen von dem von *Linux* unterscheiden kann!

Ich bespreche die RA hier mit der BRE-Syntax von `grep` unter *Linux*. Ich werde hierbei nur den Funktionsumfang abdecken, den ich für eine erste Einführung als notwendig erachte. Damit sollte es Ihnen nicht schwerfallen, nach kurzem Studium auch mit anderen Implementationen zu arbeiten. Beachten Sie auch, dass viele Programme inzwischen mehrere Dialekte direkt unterstützen, z. B. können Sie mit `grep` unter *Linux* alle drei Varianten nutzen (`grep` für BRE, `grep -E` für ERE und `grep -P` für PCRE).

Im Folgenden werden Treffer eines RA in den Ausgaben durch *kursive Schrift* hervorgehoben:

```
user$ grep 'Meyer' meier.txt
Wir haben heute Michael Meyer getroffen.
```

Sie sollten testen, ob Ihre `grep`-Implementation eine Option besitzt, um Treffer auf dem Terminal deutlich zu kennzeichnen. Zum Beispiel besitzt `grep` unter *Linux* die Option `--color=auto`, welche eine farbliche Markierung erlaubt. Wenn es eine solche Option bei Ihnen gibt, können Sie den `grep`-Befehl mit einem geeigneten Alias so umdefinieren, dass er implizit *immer* mit dieser Option aufgerufen wird;

siehe Abschn. 10.4.2. Beachten Sie auch, dass wir von jetzt an RA grundsätzlich in einfache Anführungsstriche einbetten, um Probleme mit Shell-Sonderzeichen zu vermeiden (siehe Kap. 8).

Als Erstes wollen wir schrittweise einen RA konstruieren, der einen Text nach dem Namen *Meier* mit all seinen Schreibweisen durchsucht. In der Datei `meier.txt` finden Sie den Namen in folgenden 16(!) Schreibweisen: Meyer, Meier, Maier, Mayer, Mayr, Mair, Meir, Mejr, Majer, Meyr, Meijer, Mejer, Maijer, Maiyer, Meiyer und Maihr. Unsere Aufgabe lautet also „Finde alle Zeilen in der Datei `meier.txt`, die den Namen Meier in den angegebenen Schreibweisen findet. Es sollen allerdings *nur* Zeilen mit *genauen* Treffern dieser Namen gelistet werden!“

A.1.1 Einfache Zeichenketten

Mit Ihrem bisherigem Wissen könnten Sie die Aufgabe nur so angehen, dass Sie `grep` nacheinander mit den verschiedenen Schreibweisen aufrufen:

```
user$ grep 'Meyer' meier.txt
Wir haben heute Michael Meyer getroffen.

user$ grep 'Meir' meier.txt
Meir Miriam

user$ grep 'Mair' meier.txt
Mair Melanie
Mairrose
Le Maire de Paris
MairDumont GmbH
```

Obwohl diese Herangehensweise bei vielen Alternativen wie hier sicher unpraktikabel ist, verdeutlicht das Beispiel wichtige Punkte: Der einfachste RA ist eine konkrete Zeichenkette ohne Metazeichen. Hier steht jedes Zeichen für sich selbst und, wie erwartet, werden alle Zeilen ausgegeben, welche die konkrete Zeichenkette enthalten. Man beachte aber im letzten Beispiel, dass „Mair“ z. B. auch von „Mairose“ getroffen wird, was wir nicht wollen. Dies führt auf die Notwendigkeit von *Verankerungen* (siehe Abschn. A.1.4).

A.1.2 Zeichenklassen

Das erste und meines Erachtens wichtigste Metazeichen in RA ist ein eckiges Klammerpaar, welches eine *Klasse* von Zeichen einschließt:

```
user$ grep 'M[ae]iyer' meier.txt
Maiyer Marcel
Meiyer Mario
```

Das RA-Konstrukt „[ae]“ besagt, dass an der Stelle seines Auftretens ein a *oder* ein e stehen kann. In ein Klammerpaar können nicht nur zwei, sondern beliebig viele Zeichen geschrieben werden. Des Weiteren steht jedes Klammerpaar für *genau ein* Zeichen in einer Zeichenkette:

```
user$ grep 'M[ae][ijy]r' meier.txt
Mayr Margit
Mair Melanie
Meir Miriam
Meyr Mike
Mayrhofen Mara
Mairose
Le Maire de Paris
Meyrose Fashion
MairDumont GmbH
Mejr Technologies
```

Um eine Menge an Zeichen anzusprechen, die *alles außer* einer bestimmten Klasse enthält, schreibt man in ein eckiges Klammerpaar als Erstes ein *Caret* ^:

```
user$ grep 'Mai[^yrj]' meier.txt
Maihr Maike
Wie geht es Mandy Maierhofer heute?
Mais
```

Hier werden alle Zeichenketten, die mit Mai beginnen, aber nicht mit y, r oder j weitergehen, angesprochen.

Praktisch ist die Möglichkeit, direkt *alphabetische Bereiche* an Zeichen anzugeben. So stehen z. B. „[A-Z]“ für einen beliebigen Großbuchstaben, „[1-5]“ für eine beliebige Ziffer zwischen 1 und 5. Auch Kombinationen aus Bereichen und einzelnen Zeichen, wie „[a-z3-7A]“ (ein Kleinbuchstabe, eine der Ziffern 3–7 oder ein großes A) sind möglich.

Ähnlich wie in der Shell haben wir bei RA Probleme, wenn wir nach Metazeichen selbst suchen wollen. Falls wir einen Text wie [x] *inklusive* der Klammern finden wollen, leistet der RA „[x]“ offensichtlich nicht das Gewünschte:

```
user$ echo "[x] und [y]" | grep '[x]'
[x] und [y]
```

Ähnlich wie bei den Sonderzeichen der Shell muss hier die Metazeichenfunktion durch *Quotierung* aufgehoben werden. Bei den RA wird dies erreicht, indem wir dem zu schützenden Zeichen einen Backslash „\“ voranstellen:

```
user$ echo "[x] und [y]" | grep '\[x\]'
[x] und [y]
```

Ähnliches gilt für andere Metazeichen. Hierauf komme ich aber weiter unten nochmal zurück.

Neben dem eckigen Klammerpaar ist die zweite Zeichenklasse, die man kennen muss, ein einfacher Punkt „.“. Er repräsentiert *ein beliebiges Zeichen*:

```
user$ echo "axa a9a bxb txt t9t t?t t(t" | grep 't.t'
axa a9a bxb txt t9t t?t t(t
```

Der RA „`'t.t'`“ sucht nach Zeichenketten, die aus einem `t`, gefolgt von einem beliebigen Zeichen und wieder einem `t`, bestehen.

Aufgabe A.1

Welche Zeichen werden von folgenden RA abgedeckt:

a) „`[1-59]`“?
b) „`[-a-z]`“ und „`[a-z-]`“?

Lösungsvorschlag auf Seite 212!

Aufgabe A.2

Was ist der Unterschied zwischen den RA „`[-]a`“ und „`-a`“? Erklären Sie das Ergebnis folgender Befehle:

```
user$ echo "Testtext: -a" | grep '[-]a'
Testtext: −a
```

und

```
user$ echo "Testtext: -a" | grep '-a'
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

Lösungsvorschlag auf Seite 212!

A.1.3 Wiederholung von Zeichen und Zeichenklassen

Sieht man sich die 16 Schreibweisen des Namens Meier genauer an, so erkennen wir folgende Struktur: (1) Erster Buchstabe ist ein `M`; (2) zweiter Buchstabe ist `a` oder `e`, der dritte ist in der Menge `ijy`; (3) als Nächstes kommt *optional* ein `y` oder `j` und danach ebenfalls *optional* noch ein `e` oder ein `h`; (4) der Name endet mit einem `r`.

Bisher fehlt uns eine Möglichkeit anzugeben, dass eine Zeichenklasse (oder ein einzelnes Zeichen) *optional* auftreten kann. Ein verwandtes Problem wäre: „Wir wollen alle Zeichenketten bestehend aus 3–6 Ziffern finden“. Hier würden wir gerne kontrollieren, *wie oft* ein Zeichen auftreten darf. Für diese Aufgaben sind die wichtigsten Metazeichen der Asterisk `*` und ein *gequotetes* geschweiftes Klammerpaar `\{m, n\}`, wobei m und n natürliche Zahlen mit $m \leq n$ sind. Beides sind *Modifikatoren*, die direkt hinter einem Zeichen oder einer Zeichenklasse stehen. Der `*` besagt, dass das voranstehende Zeichen (bzw. die Zeichenklasse)

Tab. A.1 RA-Metazeichen für die Wiederholung von Zeichen und Zeichenklassen

Metazeichen	Funktionsweise
\{*m*,*n*\}	Die vorstehende Zeichenklasse muss mindestens *m* und darf höchstens *n*-mal auftreten
\{*m*,\}	Die vorstehende Zeichenklasse muss mindestens *m*-mal auftreten
\{,*n*\}	Die vorstehende Zeichenklasse darf höchstens *n*-mal auftreten
\{*m*\}	Die vorstehende Zeichenklasse muss genau *m*-mal auftreten
*	Die vorstehende Zeichenklasse darf *beliebig oft* oder auch *überhaupt nicht* auftreten. Dies ist eine Kurzform für \{0,\}
\?	Die vorstehende Zeichenklasse ist optional. Dies ist eine Kurzform für \{,1\}
\+	Die vorstehende Zeichenklasse muss *mindestens* einmal auftreten. Dies ist eine Kurzform für \{1,\}

beliebig oft hintereinander auftreten darf. Beliebig oft schließt auch *keinmal* mit ein:

```
user$ echo "Test: a a1 a12 a123" | grep 'a[0-9]*'
Test: a a1 a12 a123
```

Es werden alle Muster gefunden, in denen beliebig viele Ziffern einem a nachfolgen. Das Metazeichen \{*m*, *n*\} besagt, dass das vorstehende Zeichen (bzw. die Zeichenklasse) *mindestens m*-mal und *höchstens n*-mal für einen Treffer auftreten darf:

```
user$ echo "Test: a a1 a12 a123" | grep 'a[0-9]\{1,2\}'
Test: a a1 a12 a123
```

Man darf bei dem RA „\{*m*,*n*\}“ auch *m* oder *n* weglassen. So steht „\{*n*,\}“ für *mindestens n*-maliges Auftreten. Der * ist also eine Abkürzung für \{0,\}. Das Ganze ist in Tab. A.1 zusammen mit weiteren Abkürzungen für häufig auftretende Kombinationen von \{*m*,*n*\} noch einmal zusammengefasst.

Wahrscheinlich fällt Ihnen hier sofort auf, dass die Metazeichen \{*m*, *n*\} oder das \? vorangestellte Backslash-Zeichen aufweisen. Eine geschweifte Klammer oder ein ? ohne Backslash stehen für sich selbst:

```
user$ echo "{Ein Text in Klammern?}" | grep '[{}?]'
{Ein Text in Klammern?}
```

Es erscheint in der Tat sehr unlogisch, dass bei einigen Metazeichen die Sonderfunktion durch einen Backslash aufgehoben, bei anderen dagegen durch dasselbe Zeichen aktiviert wird! Dies ist wieder eine Folge der Entwicklung von RA. Bestimmte Metazeichen kamen erst später hinzu, und man wollte andererseits das Verhalten bereits bestehender RA und Programme nicht verändern. Die Lösung bei den BRE ist wie hier beschrieben. Bei den ERE und den PCRE hat man sich dieses Ballasts entledigt und die Behandlung von Metazeichen durchgehend konsistent implementiert.

Mit unseren neuen Metazeichen ist ein RA, der alle 16 Schreibweisen des Namens Meier erfasst, durch „`M[ae][ijy][jy]\{,1\}[eh]\{,1\}r`“ oder „`M[ae][ijy][jy]\?[eh]\?r`“ gegeben:

```
user$ grep 'M[ae][ijy][jy]\?[eh]\?r' meier.txt
Mayer Michaela
Mayr Margit
Mair Melanie
.
.
.
Meierson Malin
Maierhofer Mandy
Mayrhofen Mara
Mayersche Buchhandlung
Mairose
Le Maire de Paris
Meyrose Fashion
MairDumont GmbH
Mejr Technologies
```

Wir sehen, dass der RA zusätzlich zu allen gültigen Schreibweisen des Namens weitere Zeichenketten anspricht, z. B. Worte, die den Namen enthalten. Darum kümmern wir uns im Folgenden.

Aufgabe A.3

Das Wildcard-Muster `*.txt` trifft auf alle Dateinamen, die auf `.txt` enden. Wie lautet der entsprechende RA zum Aufspüren von Zeichenketten, die auf `.txt` enden? Was sind die RA-Analoga zu den Wildcard-Mustern `?.txt` und `[0-9].txt`?

Lösungsvorschlag auf Seite 213!

A.1.4 Verankerungen

Sie werden oft RA konstruieren, die zwar ihre Aufgabe vollkommen zufriedenstellend erfüllen, aber *formal* auf weitere Textstellen zutreffen, die Sie eigentlich gar nicht finden wollen. Unser RA zum Auffinden der verschiedenen Schreibweisen des Namens Meier trifft auf weitere Zeichenketten zu, die nichts mit einer gültigen Schreibweise des Namens zu tun haben:

```
user$ echo "Testnamen: Meihr Maiyhr | \
grep 'M[ae][ijy][jy]\?[eh]\?r'
Testname: Meihr Maiyhr
```

Hier muss man entscheiden, ob es sich lohnt, einen RA signifikant zu verkomplizieren, um *theoretische* Spezialfälle auszuschließen, oder ob man sich Arbeit sparen

kann, da der RA in *realistischen* Anwendungen seine Arbeit zufriedenstellend erledigt. Im vorliegenden Fall würde ich mir über das Auftreten von verkehrten Schreibweisen des Namens Meier keine allzu großen Sorgen machen. Problematischer ist, dass *korrekte Schreibweisen* als Teilzeichenkette auftreten können, z. B. in *Mairose* oder *Meierson*:

```
user$ grep 'Meier' meier.txt
Mathilde Meier ist unsere neue Studentin.
Meier Madita
Meierson Malin
```

Hier würden wir gerne die zu findende Zeichenkette innerhalb eines Wortes *verankern*, d. h. nur Treffer zulassen, in denen der Name ein komplettes Wort ist. Das Problem ist leider komplizierter, als man anfangs denkt. Ein spontaner Lösungsansatz ist, den RA mit zwei Leerzeichen zu umgeben:

```
user$ grep '␣Meier␣' meier.txt
Mathilde Meier ist unsere neue Studentin.
```

Zwar wird der Name *Meierson* nicht mehr gefunden, aber leider verschwindet auch die Zeile *Meier Madita*. Das Problem ist hier, dass der Name am Anfang der Zeile steht und so kein voranstehendes Leerzeichen existiert. Ähnliche Probleme tauchen auf, wenn der Name direkt vor oder nach einem Satzzeichen steht, wie z. B. „Ich besuche heute Madita Meier.“. Zur Lösung existiert das Metazeichen des gequoteten spitzen Klammerpaares: `\< \>`. Dieses Metazeichen modifiziert an seiner Position kein Zeichen oder eine Zeichenklasse, sondern es *verankert* den eingebundenen RA an den Anfang und das Ende eines *Wortes*. Hierbei werden Satzanfang, Satzende und Satzzeichen korrekt behandelt:

```
user$ grep '\<Meier\>' meier.txt
Mathilde Meier ist unsere neue Studentin.
Meier Madita
```

Angewendet auf unseren RA zum Finden aller Schreibweisen des Namens Meier werden jetzt alle Zeichenketten, die den Namen als Teilkette enthalten, nicht mehr angezeigt:

```
user$ grep '\<M[ae][ijy][jy]\?[eh]\?r\>' meier.txt
Wir haben heute Michael Meyer getroffen.
Mathilde Meier ist unsere neue Studentin.
Meier Madita
Wenden Sie sich am Martha Maier.
.
.
.
Maiyer Marcel
Meiyer Mario
Maihr Maike
Mejr Technologies
```

Tab. A.2 Metazeichen für eine Verankerung von RA

Metazeichen	Funktionsweise
`^`	Ein RA nach dem Caret `^` trifft nur Textmuster am *Anfang* einer Zeile. Das Caret gilt nur als *erstes* Zeiches eines RA als Metazeichen.
`$`	Ein RA vor dem Dollarzeichen `$` trifft nur Textmuster am Ende einer Zeile. Das Dollarzeichen gilt nur als *letztes* Zeichen eines RA als Metazeichen.
`\< \>`	Ein RA innerhalb des gequoteten spitzen Klammerpaars trifft nur auf eigenständige *Worte*. Der RA für das Wort *Unix* wäre „`\<Unix\>`“. Für einen Treffer muss sich vor dem `U` der Zeilenanfang oder ein Zeichen *außer* einem Buchstaben, einer Zahl und dem Unterstrich _ befinden. Ähnlich kann nach dem `x` das Zeilenende oder ein Zeichen *außer* einem Buchstaben, einer Zahl und dem Unterstrich stehen.

Als weitere Möglichkeiten, RA in einem Text zu verankern, gibt es noch die Metazeichen `caret ^` *als erstes Zeichen* eines RA und das Dollarzeichen `$` *als letztes Zeichen* eines RA. Das Caret verankert einen RA am Zeilenanfang, das Dollarzeichen am Zeilenende:

```
user$ echo 'Meier Maier' | grep 'M[ae]ier'
Meier Maier

user$ echo 'Meier Maier' | grep '^M[ae]ier'
Meier Maier

user$ echo 'Meier Maier' | grep 'M[ae]ier$'
Meier Maier
```

Die Metazeichen zur Verankerung von RA sind noch einmal in Tab. A.2 zusammengefasst.

Aufgabe A.4

Geben Sie jeweils einen RA an, der folgende Muster erkennt:

a) Entweder die Ziffern 1–4 oder 7–9.
b) Die Zahlen 1000–9999.
c) Zeichenketten aus Kleinbuchstaben mit *mindestens* zwei Vokalen.
d) Zeichenketten mit beliebigen Zeichen, aber *genau zwei* Ziffern.
e) Einen Namen (erster Buchstabe ist Großbuchstabe, ansonsten nur Kleinbuchstaben; maximale Wortlänge sei 20).

Lösungsvorschlag auf Seite 213!

Aufgabe A.5

Geben Sie RA an, die auf Regionen in unserer Datei `exposures.txt` ansprechen (Problem 3 auf Seite 152).

Zusatzaufgabe: Eine spontane Lösung ist es, zwei RA anzugeben – einen für die `D`-Regionen und den anderen für die `W`-Regionen. Dies ist eine sehr gute Behandlung des Problems. Es gibt im BRE-Funktionsumfang von `grep` unter *Linux* Metazeichen, um einen RA mit *Alternativausdrücken* zu erzeugen. Analog zu den Zeichenklassen könnte man sagen, dass man damit eine Klasse an RA bilden kann. Erkundigen Sie sich bei Interesse im Internet und geben Sie gegebenenfalls *einen* RA zur Lösung des Problems an.

Lösungsvorschlag auf Seite 214!

Aufgabe A.6

Geben Sie einen RA für das Telefonnummernproblem (Problem 4 auf Seite 152) an.

Lösungsvorschlag auf Seite 214!

Aufgabe A.7

Sie sind Editor einer Zeitschrift und geben Ihren Autoren die Vorgabe, Datumsangaben in dem Format `tt/mm/jjjj` zu schreiben. Beispiele wären: `01/05/2010` und `15/11/1970`. Nach kurzem Überfliegen einiger Texte entdecken Sie auch folgende Formate, die nicht Ihren Vorgaben entsprechen: `1/1/1981`, `11/12/70`, `07-06-2003` und `03.5.1999`. Formulieren Sie Rahmenbedingungen für einen RA, um Datumsangaben in Ihren Texten aufzuspüren. Implementieren Sie dann diesen RA.

Lösungsvorschlag auf Seite 215!

Aufgabe A.8

Der Befehl `ls` bietet keine Option, um in einem Verzeichnis *nur* Dateien zu listen. Finden Sie hierfür eine Möglichkeit, indem Sie den Befehl `ls -l` in einer Pipeline mit `grep` verwenden. Geben Sie auch eine Pipeline an, die *nur* Verzeichnisse listet.

Hinweis: Sie können mithilfe des Rechtevektors entscheiden, ob Sie eine Datei oder ein Verzeichnis vor sich haben.

Lösungsvorschlag auf Seite 215!

Aufgabe A.9

Erkundigen Sie sich über den Aufbau gültiger E-Mail-Adressen und entwickeln Sie einen RA zum Auffinden von E-Mail-Adressen in einem Text.

Hinweis: Wie in Abschn. A.1.4 diskutiert, sollte man bei der Konstruktion von RA von der *Realität* ausgehen. Sie werden sehen, dass man *theoretisch* E-Mail-Adressen haben kann, die wohl in der Praxis *so gut*

wie nie auftreten werden. Haben Sie z. B. schon einmal eine Adresse wie `(Kommentar)MaxMuster@beispiel.de` gesehen?
Lösungsvorschlag auf Seite 216!

A.1.5 Zusammenfassung und Bemerkungen

Wir haben hier besprochen, wie man Textmuster mit RA *findet*, und ich möchte das Thema mit einer Zusammenfassung des Erlernten und einigen Zusatzbemerkungen abschließen.

1. Ein RA hat drei wesentliche Elemente: (1) *Zeichen* und *Zeichenklassen* bestimmen, welche Menge an Zeichen an *einer bestimmten Position* stehen muss; (2) *Modifikatoren* bestimmen, *wie oft* ein vorstehendes Zeichen (oder eine Zeichenklasse) an der entsprechenden Textstelle wiederholt wird, und (3) *Verankerungen* bestimmen, wie eine zu findende Zeichenkette in eine Textzeile eingebettet ist. Diese drei grundlegenden Elemente sind in allen Dialekten enthalten, wenn auch oft mit größerem Funktionsumfang und anderer, bzw. erweiterter Syntax.
2. Sie werden mit der Zeit ein Gefühl dafür bekommen, welche Probleme mit RA lösbar sind und welche nicht, bzw. nur mit großem Aufwand. Es ist auch nicht unüblich, dass sich sehr einfach aussehende Probleme als recht komplex entpuppen. In der Lösung von Aufgabe A.7 habe ich z. B. bemerkt, dass das sehr natürlich und einfach wirkende Problem eines RA für „entweder zwei oder vier Ziffern“ mit dem hier Besprochenen schon gar nicht mehr lösbar ist. Man beachte auch, dass es in RA *keine* Möglichkeit gibt, zu zählen, wie oft ein bestimmtes Zeichen (oder eine Zeichenklasse) schon vorgekommen ist. So ist es sehr einfach, einen RA für das Problem „Finde alle Zeichenketten mit mindestens zwei Ziffern“ anzugeben, nämlich `.*[0-9].*[0-9].*`, aber *unmöglich* einen *allgemeinen* für das Problem *höchstens zwei Ziffern in einer Zeichenkette* zu finden (falls die Gesamtzahl der Zeichen bekannt und klein ist, geht es noch).
3. `grep` und seine RA arbeiten eine Textdatei *zeilenweise* ab. Ein Absuchen von Text über mehrere Zeilen (z. B. eine Suche nach einer öffnenden und schließenden Klammer in verschiedenen Zeilen) ist mit dem hier Besprochenen *nicht* möglich.
4. Der Name `grep` steht für *global regular expression print*. Hierbei bedeutet *global*, dass *alle* Vorkommen eines RA-Treffers in einer Zeile berichtet werden:

   ```
   user$ echo 'Meier Maier Meier' | grep 'Meier'
   Meier Maier Meier
   ```

   Der Name Meier kommt hier zweimal vor und wird auch beide Male gefunden. Dies ist nicht selbstverständlich! Die meisten Programme, die mit RA arbeiten, finden standardmäßig *nur das allererste* Vorkommen. *Globale* Suche ist dort eine zusätzliche Option.

5. RA sind *gierig*:

```
user$ echo 'abc (def) abc (def) abc' | grep '(.*)'
abc (def) abc (def) abc
```

   `grep` erkennt hier die *längst mögliche* Zeichenkette, für die der RA gültig ist. Der RA „`(.*)`“ steht für eine Zeichenkette, die mit einer öffnenden Klammer anfängt und *beliebig viele Zeichen* vor einer schließenden enthält. Man mache sich eventuell klar, dass das von `grep` gelieferte Ergebnis diese Bedingung erfüllt! Dass RA die *längst mögliche* Zeichenkette ansprechen, also *gierig* sind, ist der Standard. Im vorliegenden Fall möchte man wahrscheinlich Text innerhalb eines Klammerpaares finden, und ein *genügsames* Verhalten des RA wäre wünschenswert. Auch dies lässt sich bei bestimmten Implementationen optional steuern.
6. Mit dem hier erworbenen Wissen wird es Ihnen sehr leicht fallen, sich schnell in den Komplex des *Bearbeitens und Veränderns* von gefundenen Textstellen einzuarbeiten. *Unix*-Werkzeuge für diese Aufgabe sind das Programm `sed` *(Stream Editor)* oder das Programm `awk`. Auch werden RA von allen gängigen Skript- und Programmiersprachen direkt und sehr umfangreich unterstützt, z. B. von *Perl, Python, C, Java* usw.

Anhang B
Dateien unter *Unix* archivieren und komprimieren

Zusammenfassung

Ihnen werden in der *Unix*-Welt sehr schnell Dateien mit der Endung `.tar` oder `.tgz` begegnen. Hierbei handelt es sich um Dateiarchive (oder `tar`-Archive), ähnlich den `zip`-Dateien unter *Microsoft Windows*. Wie Sie mit `tar`-Archiven umgehen und selbst welche erstellen, erfahren Sie in diesem Kapitel.

Um unter *Unix* ein Archiv von Dateien und Verzeichnissen zu erstellen, wird am häufigsten der Befehl `tar` *(tape archive)* verwendet. Von *Microsoft Windows* her kennen Sie wahrscheinlich die sogenannten `zip`-Archive (oder `zip`-Dateien). Das Programm `zip` archiviert Dateien und komprimiert sie gleichzeitig. Bei `tar` ist das Komprimieren eine Zusatzoption oder ein zweiter Schritt. Die grundlegende Syntax zum Archivieren von Verzeichnissen `VERZ` oder Dateien `DATEIEN` in eine Archiv-Datei `ARCHIV.tar` ist: `tar -cf ARCHIV.tar DATEIEN/VERZ`. Um ein Archiv wieder auszupacken, verwenden wir: `tar -xf ARCHIV.tar`. Gängige Aufrufe des Kommandos sind in Tab. B.1 aufgelistet.

Im Folgenden einige Beispiele und Zusatzbemerkungen:

- Die Argumentbehandlung des `tar`-Befehls unterscheidet sich leider unter verschiedenen *Unix*-Implementationen, und Sie sollten hier gegebenenfalls Ihre Dokumentation zurate ziehen. Für *Linux* gilt Folgendes:

  Fasst man in einem `tar`-Befehl mehrere einbuchstabige Optionen nach einem `-` zusammen, so muss die Option `f` (*archive file name*) zum Schluss gegeben werden, falls sie vorkommt. Grund ist, dass der Archivname *direkt nach* der Option stehen muss:

```
user$ ls
test.txt
user$ tar -cvf test.tar test.txt
test.txt
```


T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

Tab. B.1 Gängige Aufrufvarianten des `tar`-Kommandos

tar-Befehl	Erläuterung
`tar -cf ARCHIV.tar DATEIEN/VERZ`	`tar`-Archiv `ARCHIV.tar` aus `DATEIEN/VERZ` erzeugen (*create archive file*)
`tar -czf ARCHIV.tar.gz DATEIEN/VERZ`	`ARCHIV.tar` erzeugen und komprimieren (*create archive file and zip with* `gzip`)
`tar -xvf ARCHIV.tar`	`ARCHIV.tar` in aktuellem Verzeichnis auspacken (*extract/verbose*)
`tar -tf ARCHIV.tar`	`ARCHIV.tar` testen und Inhalt zeigen (*test*)
`tar -tzf ARCHIV.tar.gz`	Komprimiertes Archiv `ARCHIV.tar.gz` testen (*test/zipped*)

```
user$ ls
test.tar  test.txt   # alles in Ordnung

user$ rm test.tar
user$ tar -cfv test.tar test.txt
tar: test.tar: Cannot stat: No such file ...
tar: Exiting with failure status ...
user$ ls
test.txt  v
```

Bei dem zweiten Befehl erstellt `tar` ein Archiv mit dem Namen `v`, in dem die Dateien `test.tar` und `test.txt` archiviert werden sollen. Da `test.tar` nicht existiert, kommt es zu der Fehlermeldung.

- ```
  user$ tar -cvf thomas.tar /home/thomas
  tar: Removing leading '/' from member names
  /home/thomas/
  /home/thomas/bin/
  /home/thomas/bin/scamp
  /home/thomas/bin/etprofile
  .
  .
  ```

  Dieser Befehl sichert den gesamten Inhalt (Dateien, Verzeichnisse, Unterverzeichnisse usw.) des Benutzers `thomas`. Die erstellte Archivdatei (`thomas.tar`) befindet sich dann im aktuellen Verzeichnis (falls kein expliziter Pfad für sie angegeben wird).
- Falls ein *absoluter Pfad* für die zu sichernde Datei oder das Verzeichnis angegeben wird, so entfernt `tar` den führenden Schrägstrich und gibt eine entsprechende Warnung aus; siehe zweite Zeile oben. Hiermit ist es möglich, die Dateien in einem *beliebigen* Verzeichnis wieder auszupacken; stellen Sie
  ```

sich vor, Sie kopieren die Datei auf einen anderen Rechner, auf dem Sie in `/home/thomas` keine Schreibrechte haben, oder auf dem dieser Pfad erst gar nicht existiert!

- ```
  user$ tar -czvf /export/thomas/thomas.tar.gz \
    /home/thomas
  tar: Removing leading '/' from member names
  /home/thomas/
  /home/thomas/bin/
  /home/thomas/bin/scamp
  /home/thomas/bin/etprofile
  .
  .
  ```

  Dasselbe wie oben; gleichzeitig wird das Archiv aber noch komprimiert und das Ergebnis in `/export/thomas/` abgelegt. Als Archivendung wählen wir hier typischerweise `.tar.gz` oder `.tgz`.

- ```
  user$ pwd
  /export/thomas
  user$ tar -xvf thomas.tar
  ./home/thomas/
  ./home/thomas/bin/
  ./home/thomas/bin/scamp
  .
  .
  user$ ls /export/thomas/home/thomas
  bin
  .
  .
  ```

  Entpackt die Archivdatei `thomas.tar` im aktuellen Verzeichnis.

Eines der meistbenutzen Programme, um Dateien zu komprimieren/dekomprimieren, ist unter *Unix* `gzip/gunzip`. Es wird auch von `tar` in Verbindumg mit der Option `-z` benutzt. Der Befehl `gzip` komprimiert die angegebene Datei, hängt an den Dateinamen `.gz` an und löscht die Originaldatei:

```
user$ ls -l
-rw-r--r-- ... 296960 Oct 27 16:50 Update.tar
user$ gzip Update.tar
user$ ls -l
-rw-r--r-- ... 59258 Oct 27 16:50 Update.tar.gz
user$ gunzip Update.tar.gz
user$ ls -l
-rw-r--r-- ... 296960 Oct 27 16:50 Update.tar
```

Neben `gzip` gibt es unter *Unix* noch weitere Werkzeuge zum Packen und Entpacken von Dateien, z. B. `compress/uncompress` (Archivendung ist hier typischerweise `.Z`), `bzip2/bunzip2` (typische Archivendung ist `.bz2`). Möchte man andere Komprimierungsverfahren als `gzip` auf `tar`-Archive anwenden, so muss das Archivieren und Komprimieren *in der Regel* in zwei Schritten erfolgen. Manche `tar`-Implementationen haben inzwischen mehrere Komprimierungsalgorithmen eingebaut. So erlaubt z. B. `tar` unter *Unix* das Komprimieren mit `bzip2` (Option `-j`, bzw. `-bzip2`).

Hinweis: Ein gutes Programm, um Dateien, die mit allen möglichen *Unix*-Komprimierformaten behandelt wurden, unter *Microsoft Windows* zu bearbeiten, ist `7-zip` (http://www.7-zip.org/).

Anhang C
Deutsche versus englische *Unix*-Umgebungen

Übersicht

Zusammenfassung

In diesem Kapitel beleuchte ich einige wichtige Aspekte von deutschen und englischen *Unix*-Umgebungen. Wir diskutieren zunächst den Gebrauch von deutschen und englischen Tastaturen. Danach weise ich Sie auf potenzielle Probleme einer deutschen Sprachumgebung während Ihrer wissenschaftlichen Arbeit hin.

Falls Sie sich auf Ihrem Rechner ein eigenes *Linux* installiert haben (siehe Abschn. 3.1), so wird dies höchstwahrscheinlich eine deutsche Variante sein. In Universitäten sind dagegen oft englischsprachige Umgebungen (Tastatur, Betriebssystemsprache) üblich, damit auch ausländische Studenten und Mitarbeiter problemlos mit dem System arbeiten können. Man mag glauben, dass der Unterschied der Umgebungen *hauptsächlich* in der Sprache von Systemmeldungen und System- und Programmmenüs liegt. Dem ist nicht so, und ich möchte die für Sie wesentlichen Punkte hier diskutieren. Vor allem, wenn Sie sowohl mit deutschen als auch englischen Umgebungen arbeiten, müssen Sie sich potenzieller Probleme bewusst sein!

Das Thema hat zwei unabhängige Aspekte: Mit welcher Tastatur sollte man arbeiten, und welche *Unix*-Spracheinstellungen sollte man verwenden.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

C.1 Deutsche versus englische Tastatur

Da Sie mit deutschen Texten unter *Unix* arbeiten werden (Textdateien, E-Mails, usw.), werden Sie deutsche Umlaute benötigen und deswegen auch eine deutsche Tastatur verwenden wollen. Andererseits sind auf englischen Tastaturen die unter *Unix* und allen wichtigen Programmiersprachen essenziellen Zeichen „\ [] { } | “ *sehr* einfach zugänglich, während sie auf einer deutschen Tastatur nur umständlich zu tippen sind. Viele Leute bevorzugen daher eine englische Tastatur, sobald sie den größten Teil ihrer Arbeit auf der *Unix*-Kommandozeile oder mit Programmieren verbringen. Da man in Datei- und Verzeichnisnamen von deutschen Umlauten sowieso Abstand nehmen sollte, gibt es hier kein Problem.

Wenn man *das Beste aus beiden Welten* haben möchte, gibt es mehrere Möglichkeiten. Man kann z. B. über USB sowohl eine englische als auch eine deutsche Tastatur an seinen Rechner anschließen. *Unix*-Desktop-Umgebungen lassen sich dann so konfigurieren, dass die einfache Betätigung eines Icons ausreicht, um zwischen den Tastaturen umzuschalten. Diese Lösung erlaubt mir auf sehr einfache Weise die Benutzung meiner deutschen Tastatur, wenn ich sie für deutsche Texte brauche. Ansonsten benutze ich für meine tägliche Arbeit die englische.

Eine weitere Möglichkeit ist die primäre Nutzung einer englischen Tastatur, welche so konfiguriert ist, dass man mit bestimmten Tastenkombinationen kurzfristig auf eine deutsche Tastaturbelegung umschalten kann.

Was genau unter Ihrer konkreten *Unix*-Installation möglich ist und wie Sie bei Bedarf eine bestimmte Lösung realisieren können, schauen Sie am einfachsten im Internet nach.

C.2 Deutsche versus englische Spracheinstellungen

Wir hatten *Unix*-Spracheinstellungen bereits in Abschn. 12.1.3 in Verbindung mit verschiedenen Kodierungen von Textdateien erwähnt:

```
user$ locale
LANG=en_US.UTF-8
.
.
LC_ALL=
```

Die `locale`-Einstellungen geben Auskunft über die Spracheinstellungen und die Textkodierung, welche *Unix*-Programme benutzen sollen. Hier wollen wir uns mit den Spracheinstellungen, dem `en_US`-Teil der `LANG`-Variablen widmen. Wenn Sie ein *Unix* mit deutschen Spracheinstellungen haben, finden Sie wahrscheinlich `LANG=de_DE.UTF-8`.

Hinweis: Der `locale`-Befehl wird Ihnen eine ganze Reihe an Variablen zeigen, die *höchstwahrscheinlich* alle denselben Wert haben. Auf die Zusammenhänge der verschiedenen Variablen möchte ich hier nicht näher eingehen. Damit das Folgende bei Ihnen, so wie hier gezeigt, funktioniert, ist es nur wichtig, dass die Variable

LANG auf einen Wert gesetzt ist (bei mir ist dies en_US.UTF-8) und die Variable LC_ALL *leer* ist.

Die locale-Einstellungen sorgen dafür, dass *Unix*-Programme, welche es unterstützen, mit Ihnen *in der eingestellten Sprache* kommunizieren:

```
user$ echo ${LANG}
en_US.UTF-8
user$ date
Thu Apr 21 01:20:15 CEST 2016
   # 'date' gibt das Datum in
   # 'englischen' Konventionen
user$ date +'%x'
04/21/2016

user$ export LANG=de_DE.UTF8
   # Hier ändere ich die Spracheinstellungen
   # auf Deutsch!
user$ date
Do 21. Apr 01:21:23 CEST 2016
   # und 'date' spricht deutsch mit mir!
user$ date +'%x'
21.04.2016
user$
```

Erklärung: Die Spracheinstellungen können durch Änderung der LANG-Variablen modifiziert werden. Der Befehl locale -a gibt Ihnen Auskunft, welche Sprachpakete auf Ihrem Rechner installiert sind und auf welche Spracheinstellungen Sie wechseln können. Das date-Kommando unterstützt internationale Spracheinstellungen und kann seine Datumsangaben in einem länderspezifischen Format ausgeben. Im deutschen Sprachraum verwenden wir für eine kompakte Datumsangabe das Format TT.MM.YYYY. In den USA ist dagegen MM/TT/YYYY üblich.

Während Internationalisierungen von Programmen im Allgemeinen *sehr* zu begrüßen sind, tauchen unter Umständen ernsthafte Probleme auf, wenn wir Ergebnisse solcher Programme weiterverarbeiten möchten. Stellen Sie sich in obigem Beispiel vor, dass wir das Ergebnis des Befehls date +'%x' mit einer Kommandosubstitution in einen Dateinamen integrieren wollten; siehe Abschn. 15.1.2. Wie Sie in Aufgabe 15.2 gesehen haben, würde dies in einer deutschen Umgebung funktionieren, während das US-Format zu einem Fehler führt!

Besonders problematisch ist das folgende Beispiel. Der printf-Befehl (*print with format*) ist eine Alternative zu echo, welcher eine Formatierung seiner Ausgabe erlaubt:

```
user$ export LANG=en_US.UTF-8
user$ printf '%.2f\n' 3.14159
   # Die Formatierungsangabe '%.2f' gibt
   # eine Zahl mit 2 gültigen Dezimalziffern
```

```
   # aus. Das folgende '\n' druckt einen
   # Zeilenvorschub - im Gegensatz zu echo
   # führt printf keinen automatischen
   # Zeilenvorschub aus!
3.14     # So soll es sein!
user$ export LANG=de_DE.UTF-8
user$ printf '%.2f\n' 3.14159
   # Ist dasselbe wie oben, nicht wahr?
bash: printf: 3.14159: invalid number
0,00     #UPS!
```

Derselbe Befehl zur Ausgabe einer Zahl funktioniert mit US-Einstellungen wie erwartet und scheitert im Deutschen! Ein *genauer* Blick auf die Fehlermeldung zeigt das Problem. Die Zahl 0,00 wird mit einem *Dezimalkomma* anstatt mit einem Dezimalpunkt wie 0.00 geschrieben! In der Tat verwenden wir diese Darstellung im Deutschen und `printf` treibt die Internationalisierung so weit, dass länderspezifische Zahlendarstellungen *respektiert* werden:

```
user$ echo ${LANG}
de_DE.UTF-8
user$ printf '%.2f\n' 3,14159
   # man beachte das 'Komma' nach der 3!
3,14
   # Auch die Ausgabe hat wieder ein Komma!
```

Obwohl `printf` hier *formal* richtig handelt, sind Programme, deren Zahlenverarbeitung länderspezifisch ist, für wissenschaftliche Anwendungen natürlich höchst problematisch und eine vernünftige Arbeit ist damit nicht möglich!

Die Beispiele machen deutlich, wie komplex und vielschichtig das Thema *Internationalisierung* ist. Oben geschilderte Probleme führen auch dazu, dass unterschiedliche Programme hier *unterschiedlich weit gehen* und ihr Verhalten mit der Zeit ändern können. So hat z. B. `awk` bis ungefähr 2013 Zahlen auf *Linux* ebenfalls länderspezifisch verarbeitet. Wegen der zentralen Rolle dieses Programmes wurde das inzwischen geändert, und `awk` verwendet jetzt standardmäßig überall die Dezimalpunktdarstellung.

Um sicherzustellen, dass Sie derartige Probleme nicht irgendwann *sehr* viel Zeit kosten, sollten Sie auf *Unix*-Rechnern, die Sie für wissenschaftliche Arbeiten nutzen, englische Spracheinstellungen verwenden. Falls Sie ein deutsches System installiert haben, ist eine Definition der `LANG`-Variablen in Ihrer Systemkonfiguration die einfachste Möglichkeit, dies zu ändern.

Anhang D
Literaturhinweise

Übersicht

Zusammenfassung

In diesem Anhang gebe ich Ihnen einige Literaturempfehlungen, die Ihnen einen tieferen Einstieg in *Unix* ermöglichen.

Wie im Vorwort erwähnt, gibt es zum Thema *Unix*/*Linux* eine riesige Auswahl an Literatur. Ich möchte Ihnen hier gezielt *einige wenige* Empfehlungen für Referenzmaterialien und Lehrbücher zum weiteren Studium geben. Alle hier aufgelisteten Quellen gefallen mir sehr gut, ich benutze sie aktiv, habe sie bei der Erstellung dieses Buches zurate gezogen und ich halte sie auch für Anfänger für sehr gut lesbar. Meine Empfehlungen sollen aber keinesfalls ein Urteil über andere, auf dem Markt verfügbare, Quellen sein! Fast alle aufgeführten Materialien sind mindestens in Auszügen im Internet verfügbar, sodass Sie sich vor einem eventuellen Kauf ein eigenes Bild über Umfang und Qualität machen können. Ich möchte explizit erwähnen, dass Sie für fortgeschrittene und sehr spezialisierte Themen zunehmend auf englische Literatur zurückgreifen werden müssen. Selbst wenn Sie deutsche Übersetzungen von wichtigen, englischsprachigen Werken erwerben können, so handelt es sich hier oft nur um ältere Auflagen.

Mein allgemeiner Rat ist, dass Sie sich umsehen und Dokumente bzw. Bücher studieren, die Ihnen Freude und Spaß bereiten. Meine Vorlieben sind lediglich Empfehlungen, für Ihre eigene Auswahl an Literatur sind sie aber weniger wichtig.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

D.1 *Unix*/*Linux*-Anfängerliteratur

1. **Wolfinger, C. (2000): *Keine Angst vor UNIX*, 9. Auflage, Springer (Berlin-Heidelberg)**: Das Buch eignet sich als Zusatzlektüre, falls Sie noch sehr wenig Erfahrung mit Computern haben. Es bietet einen *Unix*-Einstieg auf einem *niedrigeren Niveau*, und es werden auch Themen behandelt, die ich hier größenteils voraussetze. Dazu zählen beispielsweise eine Erklärung grundlegender Begriffe wie *Hardware, Software, Dateien, Verzeichnisse*, die An- und Abmeldung auf einem System oder die Bedienung des *Unix*-Desktops und von GUI-Programmen.
2. **Bradman, K. & Korf, I. (2012): *UNIX and PERL to the RESCUE!*, Cambridge University Textbooks (New York)**: Das Buch hat genau dieselben Zielsetzungen wie das vorliegende. Um Naturwissenschaftlern ein effizientes Arbeiten mit *Unix* zu vermitteln, wird hier eine *Kombination* aus *Unix* und der Programmiersprache *Perl* gewählt. Sie lernen hier im Wesentlichen unsere *Kernthemen* zu *Unix*. Die Anwendungsgebiete der Datenpipelines, der Shell-Skripte und der Regulären Ausdrücke werden hauptsächlich mit *Perl* vermittelt. Das Buch ist sehr lesenswert, wenn Sie sich für *Perl* interessieren oder noch einen anderen Blickwinkel auf den Stoff erhalten möchten.

D.2 *Unix*/*Linux*-Referenzmaterialien

Primäre Quelle, um schnell Details zu einem bestimmten Thema nachzuschlagen, ist sicher das Internet. Von den unzähligen guten Quellen möchte ich hier lediglich die Download-Seiten der *Open Source Training and Consulting GmbH* erwähnen. Unter http://www.ostc.de/download.html finden Sie eine große Menge sehr nützlicher Referenzmaterialien, die alle von sehr hoher Qualität sind.

An Büchern nenne ich hier ein *sehr* umfassendes Nachschlagewerk zu *Unix* und eine Kurzreferenz zur *Bash*-Shell:

1. **Plötner, J. & Wendzel, S. (2006): *Linux - Das distributionsunabhängige Handbuch*, Galileo Computing (Bonn)**: Das Werk ist ein *Hammerbuch* (ich bezeichne ein Buch als solches, wenn es so dick und schwer ist, dass man mit ihm bei Bedarf einen Nagel in die Wand schlagen könnte.), welches alles, was Sie über *Unix* wissen müssen, in einer klaren und leicht verständlichen Sprache beschreibt. Es ist als Begleit- und Nachschlagewerk konzipiert und hervorragend dazu geeignet, einzelne Themen im Detail nachzulesen. Eine etwas ältere Version dieses Buches steht unter http://www.heise.de/newsticker/meldung/98062 kostenlos zum Download zur Verfügung!
2. **Günther, K. (2014): *Bash kurz & gut*, O'Reilly Germany (Köln)**: Eine sehr kompakte, aber vollständige Referenz der *Bash*-Shell. Das Buch behandelt sowohl die interaktive Nutzung der Shell als auch alle nötigen Shell-Skript-Konstrukte. Es ist hervorragend als Ergänzung zu einem Lehrbuch über die *Bash* geeignet; siehe z. B. das Buch *Learning the bash Shell* in Abschn. D.4.

D.3 awk-Literatur

Da ich awk für eines der zentralen und wichtigsten *Unix*-Werkezeuge halte, empfehle ich Ihnen zwei Bücher speziell zu diesem Thema:

1. **Aho, A. V., Kernighan, B. W. & Weinberger, P. J. (1987): *The AWK Programming Language*, Addison-Wesley Longman Publishing Co., Inc. (Reading/Massachusetts)**: Das Buch ist von den awk-Entwicklern geschrieben und behandelt das Thema komplett auf ca. 200 Seiten. Es beschränkt sich auf das Wesentliche und vermittelt awk mit sehr vielen Praxisbeispielen, welche die Stärken dieser Programmiersprache klar darstellen. Neben awk lernt man hier auch eine Menge grundlegender Programmiertechniken, die auch heute noch sehr aktuell sind. Das Buch halte ich trotz seines Alters für *sehr* lesenswert.
2. **Herold, H. (1994): *awk und sed*, Addison-Wesley Deutschland (Bonn)**: Das Buch gibt eine sehr gute und kompakte Zusammenfassung zu awk und sed. Es ist meines Erachtens sehr gut als Nachschlagewerk für einzelne Themen geeignet, wenn man die grundlegenden Konzepte von awk bereits meistert. Der *Streameditor* sed ist neben grep eines der wichtigsten Programme für die Arbeit mit Regulären Ausdrücken; siehe Anhang A. Es übernimmt die Aufgabe, mit Regulären Ausdrücken identifizierte, Textstellen zu bearbeiten und zu modifizieren.

D.4 Shell-Skripte

1. **Newbam, C. & Rosenblatt, B. (2005): *Learning the bash Shell*, O'Reilly Media, Inc. (Cambridge)**: Das Buch beschreibt alle notwendigen Details der bash-Shell sehr verständlich. Ich selbst habe eine frühere Auflage benutzt, um mir ein umfangreicheres Wissen über die *Bash*-Shell-Nutzung und die *Bash*-Shell-Programmierung anzueignen. Das Buch zielt auf ein solides und umfangreiches Verständnis der *Bash*-Shell ab, und es sollte *komplett* durchgearbeitet werden. Für Leute, die *schnell loslegen wollen*, ist es meines Erachtens weniger geeignet.
2. **Albing, C., Vossen, J. P. & Newbam, C. (2007): *bash Cookbook*, O'Reilly Media, Inc. (Cambridge)**: Das Buch ist eine Sammlung von Beispielen, welche nach Themenbereichen sortiert sind. Es werden typische Fragestellungen behandelt, die bei der Arbeit mit *Unix* und der bash auftreten. Zur Lösung notwendige Befehle und bash-Konstrukte werden dann ausführlich erklärt. Das Buch ist einerseits eine sehr gute Zusammenstellung typischer Probleme und wie man sie löst. Andererseits erlaubt es das Erlernen der *Bash* anhand von Beispielen. Die Struktur des Buches und umfangreiche Verweise zwischen den Abschnitten erlauben es, die präsentierten Themen in beliebiger Reihenfolge anzugehen.

Anhang E
Lösungsvorschläge zu den Aufgaben

Im Folgenden finden Sie Lösungsvorschläge zu den Aufgaben, die noch nicht im Text selbst besprochen wurden:

Lösungsvorschlag zu Aufgabe 4.1

Wie im Text besprochen, dient der Befehl `ls` zum Auflisten von Dateien und Verzeichnissen. Der Befehl hat hier das Argument `/usr`, welches ein Verzeichnis repräsentiert – dies werden wir nachfolgend ausführlicher im Text darstellen. Das Verzeichnis `/usr` enthält üblicherweise selbst einige Verzeichnisse, und der erste Befehl liefert bei mir:

```
user$ ls /usr
bin  games  ...  NX  sbin  share  src
```

Bei den folgenden Befehlen sollen Sie die Wirkungsweise und den Gebrauch einiger Optionen erfassen. `ls -l` gibt eine *ausführliche* Auflistung:

```
user$ ls -l /usr
drwxr-xr-x   2 root root 147456 Apr 12 03:44 bin
drwxr-xr-x   2 root root   4096 Feb 20  2015 games
.
.
```

Die Bedeutung der ersten vier Spalten werden wir in Abschn. 6.2 kennenlernen. Die fünfte Spalte gibt die Dateigröße in Bytes an und die sechste, siebte und achte Spalte geben das Datum der letzten Dateimodifikation.

Der Befehl `ls` sortiert seine Ausgabe standardmäßig alphabetisch, was wir mit der Option `-r` (*reverse*) umdrehen können:

```
user$ ls -r /usr
src  share  sbin  NX ... games  bin
```

Die wesentlichen Punkte der restlichen Befehle werden im Text besprochen.

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

Lösungsvorschlag zu Aufgabe 4.2

a) Gegeben ist hier eine Datei `namen.txt` mit dem Inhalt:

```
Martha      Schmitz    100
Bernhard    Bar        2000
Ulf         Klein      500
Kerstin     Meier      3800
Leon        Dinter     24
```

b) Wie der Name schon erahnen lässt, sortiert der Befehl `sort` eine Textdatei.

```
user$ sort namen.txt
Bernhard    Bar        2000
Kerstin     Meier      3800
Leon        Dinter     24
Martha      Schmitz    100
Ulf         Klein      500
```

Hier wird der Dateiinhalt *alphabetisch* sortiert.

c) Bei obiger Liste (oder z. B. auch bei Adresslisten) möchte man unter Umständen gerne nach dem Nachnamen, hier also nach der zweiten Spalte, sortieren. `sort` bietet hierfür mehrere Syntaxmöglichkeiten, dies zu realisieren. Ich bevorzuge die Möglichkeit `sort -kn1,n2`, welche `sort` auffordert, die Spalten `n1-n2` als Schlüssel für die Sortierung zu verwenden. Ein Sortieren nach der zweiten Spalte erreichen wir z. B. durch:

```
user$ sort -k2,2 namen.txt
Bernhard    Bar        2000
Leon        Dinter     24
Ulf         Klein      500
Kerstin     Meier      3800
Martha      Schmitz    100
```

Hinweis: Die Beschreibung der `-k`-Option (*sort according to a key*) auf der `man`-page für `sort` finde ich für Anfänger sehr abstrakt. Hier ist ein Nachschlagen im Internet für Sie sicher einfacher.

d) Eine spontane Lösung wie oben liefert für eine Sortierung nach der dritten Spalte:

```
user$ sort -k3,3 namen.txt
Martha      Schmitz    100
Bernhard    Bar        2000
Leon        Dinter     24
Kerstin     Meier      3800
Ulf         Klein      500
```

Dies entspricht wahrscheinlich nicht Ihren Erwartungen. `sort` sortiert standardmäßig *alphabetisch* und nicht *numerisch*, was obiges Resultat erklärt! Das kann mit der Option `-n` (*numeric sort*) geändert werden:

```
user$ sort -n -k3,3 namen.txt
Leon        Dinter     24
Martha      Schmitz    100
Ulf         Klein      500
Bernhard    Bar        2000
Kerstin     Meier      3800
```

`sort` ist ein *sehr* mächtiges *Unix*-Werkzeug, über dessen Einsatzmöglichkeiten Sie in Kap. 14 mehr lernen werden.

Lösungsvorschlag zu Aufgabe 4.3

a) Um Informationen über ein *Unix*-Kommando `command` zu erhalten, geben wir den Befehl `man command`. Um Informationen über das Kommando `man` zu erhalten, müssen wir also `man man` verwenden.

b) Bei etwas genauerem Überfliegen der `man`-page zum Kommando `man` sollten Sie auf

```
man -k printf
    Search the short descriptions ...
    for the keyword printf ....
```

stoßen. Der Befehl `man -k compress` gibt eine lange Liste mit Programmen und Bibliotheken, die *irgendetwas* mit Komprimierung von Dateien zu tun haben. Diese Kurzbeschreibungen sollten genügen, damit Sie sich über potenziell interessante Programme weiter informieren können.

Lösungsvorschlag zu Aufgabe 5.1

Die Idee ist, in unser Heimatverzeichnis zu wechseln und den absoluten Pfad desselben mit `pwd` zu erfragen:

```
user$ cd ~
        # wechselt auf alle Fälle ins Heimatverzeichnis!
user$ pwd
        # erhalte den absoluten Pfad
/home/thomas
```

Lösungsvorschlag zu Aufgabe 5.2

a)
```
user$ pwd
/home/thomas/Vorlesung
user$ cd ../..
user$ pwd
/home
```

Der Befehl `cd ../..` (relative Pfadangabe!) wechselt allgemein in das *Großelternverzeichnis*. Als solcher ist der Befehl *optimal* und sinnvoll. Im gegebenen

Fall ist der Befehl `cd /home` (absolute Pfadangabe!) vollkommen gleichwertig. Letzterer wäre besser, falls man aus einer tieferen Dateibaumebene als hier, z. B. aus `/home/thomas/Vorlesung/Analysis/Uebung_1`, nach `/home` wechseln möchte. In diesem Fall wäre etwa `cd ../../../../` anstatt `cd /home` unnötig umständlich.

b)
```
user$ pwd
/home/thomas/Vorlesung
user$ cd ../../../home/thomas
user$ pwd
/home/thomas
```

Der gegebene `cd ../../../home/thomas`-Befehl ist sehr umständlich und sollte durch das einfache `cd ..` ersetzt werden. Machen Sie sich gegebenenfalls klar, dass der Befehl `cd ../../../home/thomas` durch die zwei Befehle `cd ..` und danach `cd ../../home/thomas` ersetzt werden kann:

```
user$ pwd
/home/thomas/Vorlesung
user$ cd ..
user$ pwd
/home/thomas
user$ cd ../../home/thomas
user$ pwd
/home/thomas
```

Hier ist offensichtlich, dass der zweite `cd`-Befehl in der gegebenen Situation komplett überflüssig ist und nichts bewirkt!

c)
```
user$ cd ~/Vorlesung
user$ pwd
/home/thomas/Vorlesung
```

Der gegebene Befehl wechselt von jedem Verzeichnis aus direkt in das Unterverzeichnis `Vorlesung` unterhalb des Heimatverzeichnisses (durch eine absolute Pfadangabe). Er nutzt hierzu die Abkürzung ~ für das Heimatverzeichnis und ist *fast* immer optimal. Es gibt hier nur einen offensichtlicheren Befehl mit relativer Pfadangabe, falls man sich bereits im Heimatverzeichnis befindet:

```
user$ pwd
/home/thomas
user$ cd Vorlesung
user$ pwd
/home/thomas/Vorlesung
```

Lösungsvorschlag zu Aufgabe 5.4

Die Befehle sind `cp test.txt ..` (relativer Zielpfad) und `cp test.txt ~` (absoluter Zielpfad). Für den zweiten Befehl geht natürlich auch `cp test.txt /home/thomas`, wobei Sie `/home/thomas` durch Ihr Heimatverzeichnis ersetzen müssen.

Lösungsvorschlag zu Aufgabe 5.5

a) Wir befinden uns in ~, und ~/uebung existiert als Verzeichnis: Bei den ersten beiden Befehlen wird die Datei `aufgabe_1.txt` in `./uebung` kopiert, und sie heißt dort wieder `aufgabe_1.txt`. Ähnliches geschieht bei Befehlen 3 und 4. Die Dateien werden, wie erwartet, kopiert und behalten ihre Namen.

b) Wir befinden uns in ~, und das Verzeichnis ~/uebung existiert *nicht*: Der erste Befehl kopiert `aufgabe_1.txt` mit dem neuen Namen `uebung` nach ~! Der zweite Befehl führt zu einer Fehlermeldung. Mit `./uebung/` als Ziel sprechen wir explizit ein Verzeichnis an (abschließender Slash „/“!), das es nicht gibt. Befehle 3 und 4 bleiben ebenfalls erfolglos. Das Ziel einer Kopie mehrerer Dateien muss immer ein Verzeichnis sein. Dieses existiert hier nicht.

c) Wir befinden uns irrtümlicherweise in ~/uebung, sobald wir die Kopierbefehle ausführen: Dies entspricht, mit einem anderen aktuellen Verzeichnis, den gerade diskutierten Fällen unter a) und b). Falls es ein Verzeichnis ~/uebung/uebung gibt, siehe a), ansonsten b).

Lösungsvorschlag zu Aufgabe 5.6

a) Die Aufgabe kann mit einer Reihe von `cd`- und `mkdir`-Befehlen gelöst werden, z. B.:

```
user$ cd ~
user$ mkdir aufgabe
user$ cd aufgabe
user$ mkdir ordner_1 ordner_2
user$ cd ordner_1
user$ mkdir ordner_3 ordner_4
user$ cd ../ordner_2
user$ mkdir ordner_5 ordner_6
```

b) Wir sollten uns die Frage stellen, ob man nicht *komplette* Pfade mit einem einzigen Befehl erstellen kann. Recherche führt hier auf die Option `-p` (*parent oder path*) von `mkdir`. Damit ist eine effektivere Lösung der Aufgabe z. B. folgendermaßen möglich:

```
user$ cd ~
user$ mkdir -p aufgabe/ordner_1/ordner_3
user$ mkdir -p aufgabe/ordner_1/ordner_4
user$ mkdir -p aufgabe/ordner_2/ordner_5
user$ mkdir -p aufgabe/ordner_2/ordner_6
```

c) Zum Beispiel:

```
user$ cd ~/aufgabe/ordner_1
user$ mv ordner_3 ordner_4 ../ordner_2
user$ cd ..
user$ mv ordner_2 ordner_1
user$ cd ordner_1/ordner_2
user$ rmdir ordner_5 ordner_6
```

d) Hierzu gibt es eigentlich nichts zu sagen:

```
user$ cd ~/aufgabe/ordner_1/ordner_2/ordner_4
user$ touch test_1.txt
user$ cp test_1.txt ..
user$ cp test_1.txt ../ordner_3
```

Lösungsvorschlag zu Aufgabe 5.7

Die Befehle

```
user$ mkdir ~/test
user$ cd ~/test
user$ touch "Ein␣Lied.mp3"
user$ ls
Ein␣Lied.mp3
```

führen zu einer leeren Datei mit Namen `~/test/Ein␣Lied.mp3`. Der Dateiname enthält ein Leerzeichen, und ein naiver Versuch, sie wieder loszuwerden:

```
user$ rm Ein Lied.mp3
rm: cannot remove 'Ein': No such file or directory
rm: cannot remove 'Lied.mp3': No such file or directory
```

scheitert, da `rm` die Zeichenkette `Ein␣Lied.mp3` als zwei zu löschende Dateien mit Namen „`Ein`“ und „`Lied.mp3`“ interpretiert. Das zugrundeliegende Problem, „*Korrekter Umgang mit Shellsonderzeichen*“, wird noch genauer in Kap. 8 behandelt. Spätestens nach Studium dieses Kapitels sollten die hier nur angegebenen Befehle zur Lösung der Aufgabe klar sein:

```
user$ rm Ein\ Lied.mp3
        # Hier gibt es mehrere Möglichkeiten

user$ touch \$PATH
        # Man beachte dass hier der Befehl
        # 'touch "$PATH"' in Analogie zu
        # 'touch "Ein Lied.mp3"' nicht funktioniert
        # und eine für Sie sicher sehr kryptische
        # Ausgabe erzeugt!
```

```
user$ mv '$PATH' ..
        # es geht auch 'mv \$PATH ..'

user$ rm ../'$PATH'
```

Die Aufgabe soll primär verdeutlichen, welche Probleme bei der Verwendung von Shell-Sonderzeichen in Dateinamen auftreten können!

Lösungsvorschlag zu Aufgabe 5.8

a)
```
user$ ls datei_[12][34].txt
datei_13.txt  datei_14.txt  datei_23.txt
datei_24.txt
```

Man beachte, dass jedes Klammerpaar für *genau ein* Zeichen steht!

b)
```
user$ ls datei_[12]?.txt
datei_10.txt datei_11.txt .. datei_19.txt
datei_20.txt datei_21.txt .. datei_29.txt
```

Der Befehl listet alle Dateien, die zwischen `datei_` und `.txt` genau zwei Zeichen haben, wobei das erste eine der Ziffern 1 oder 2 ist.

c)
```
user$ ls datei_[12]*.txt
datei_1.txt datei_2.txt
datei_10.txt .. datei_19.txt
datei_20.txt .. datei_29.txt
datei_100.txt .. datei_199.txt
datei_200.txt .. datei_299.txt
datei_1000.txt
```

Der Befehl listet alle Dateien, die zwischen `datei_` und `.txt` *beliebig viele* Zeichen haben, wobei das erste eine der Ziffern 1 oder 2 ist. Man beachte, dass *beliebig viele* auch *gar keine* mit einschließt!

Hinweis: Die Sortierung der Dateien in der Ausgabe von `ls` ist alphabetisch und somit anders als hier zur Demonstration gezeigt!

d)
```
user$ ls datei_*10.txt
datei_10.txt datei_110.txt .. datei_910.txt
```

Der Befehl listet alle Dateien, die zwischen `datei_` und `.txt` *beliebig viele* Zeichen haben, wobei die letzten beiden die Ziffernfolge 10 sein müssen.

Lösungsvorschlag zu Aufgabe 5.9

Mögliche Lösungen sind:

a)
```
user$ ls *01*
```

Der `*` zu beiden Seiten des benötigten Musters leistet das Gewünschte.

b) `user$ ls datei_???.txt`

oder

`user$ ls datei_[0-9][0-9][0-9].txt`

Da die Zahlen 000–999 *alle* Zahlen aus drei Ziffern sind, ist das Konstrukt mit dem `?` möglich und hier einfacher als die Lösung mit dem Klammer-Joker.

c) `user$ ls datei_00[0-59].txt`

Man beachte hier das Konstrukt `[0-59]`, welches für die Ziffern 0–5 *und* die Ziffer 9 steht!

d) `user$ ls datei_0[1-3][0-9].txt`

Man mache sich gegebenenfalls klar, dass das Wildcardkonstrukt `[1-3][0-9]` genau die Zahlen 10–39 abdeckt.

e) `user$ ls datei_*000.txt`

Man beachte, dass der `*` für beliebig viele Zeichen steht. Das schließt auch *gar keines* mit ein!

Lösungsvorschlag zu Aufgabe 5.10

Das grundlegende Problem dieser Aufgabe, „*Sonderzeichen der Shell*“ und der Umgang mit ihnen, wird noch genauer in Kap. 8 besprochen. Dort finden Sie auch die Lösung dieser Aufgabe.

Lösungsvorschlag zu Aufgabe 6.1

Um eine Datei aus einem Verzeichnis kopieren zu dürfen, braucht die Gruppe, der die Datei zugeordnet ist, das Recht, den Inhalt *lesen* zu dürfen, also `r--` als Gruppenrechte für `testdatei.txt`. Außerdem muss auf die Datei innerhalb des Verzeichnisses zugegriffen werden können. Für das Verzeichnis `rechte` bedeutet dies den minimalen Rechtevektor `--x` für die Gruppe. Hierbei ist wesentlich, dass die Gruppenmitglieder wissen, dass sich die Datei `testdatei.txt` in dem Verzeichnis `rechte` befindet. Mit dem Rechtevektor `--x` dürfen Sie sich *kein* Inhaltsverzeichnis via `ls` besorgen. Hierzu wäre zusätzlich das `r`-Flag nötig!

Lösungsvorschlag zu Aufgabe 6.2

Die Lösung der Aufgabe lautet:

a)
```
~/hausaufgaben:                  drwxr-xr-x
~/hausaufgaben/analysis_1.txt:   -rwxr--r--
~/hausaufgaben/getmean.sh:       -rwxr-xr-x
```

Die gezeigten Rechte müssen im Sinne der Aufgabenstellung vorhanden sein. Zusätzliche Rechte für Gruppe und alle anderen Benutzer können natürlich gesetzt werden.

b) ~/hausaufgaben: drwxr-x---
~/hausaufgaben/analysis_1.txt: -rwxr-----
~/hausaufgaben/getmean.sh: -rwxr-x---

Für das Verzeichnis ~/hausaufgaben wäre hier auch der Rechtevektor drwxr-xr- korrekt. Es war nicht spezifiziert, ob *alle Anderen* das Verzeichnis listen dürfen oder nicht. Man beachte, dass die Rechte für *alle Anderen* bei den Dateien eigentlich unerheblich sind. Da sie für das Verzeichnis ~/hausaufgaben kein x-Flag haben, können sie auf die Dateien auch nicht zugreifen!

Lösungsvorschlag zu Aufgabe 6.3

Bei der gegebenen Installation konnten die Studenten sehr wohl die Konfiguration ihrer Shell verändern. Man macht sich zunutze, dass die root-Datei .bashrc zwar *nicht verändert*, aber *gelöscht* werden kann, da jedem Benutzer sein Heimatverzeichnis gehört! Wir können hiermit die .bashrc-Datei effektiv von root auf uns *überschreiben*:

```
user$ cp .bashrc bashrc.copy
        # möglich da wir die Datei 'lesen' dürfen!

user$ ls -l bashrc.copy
-rw-r--r-- 1 thomas users  724 Oct 11 09:52 bashrc.copy
        # Beachten Sie, dass die Datei bashrc.copy
        # 'uns' gehört!

user$ rm .bashrc
        # Da uns das Heimatverzeichnis gehört,
        # geht das!

user$ mv bashrc.copy .bashrc
user$ ls -l .bashrc
-rw-r--r-- 1 thomas users  724 Oct 11 09:52 .bashrc
```

Die neue .bashrc-Datei kann jetzt nach Belieben geändert werden!

Lösungsvorschlag zu Aufgabe 8.1

Die Aufgabe besteht darin, Text zusammen mit Sonderzeichen auszugeben. Viele lassen sich dadurch verwirren, dass Sonderzeichen, welche üblicherweise Sonderfunktionen aufheben, in die Ausgabe involviert sind. Machen Sie sich gegebenenfalls zunächst klar, warum die naiven Versuche

```
user$ echo "Ein Text"
Ein Text
user$ echo 'Mehr Text'
Mehr Text
```

scheitern. Zur Lösung gibt es zwei Möglichkeiten:

- Man benutzt das dritte Zeichen, das zur Quotierung dient, und nicht in die Ausgabe involviert ist:

```
user$ echo \"Ein Text\"
"Ein Text"
user$ echo \'Mehr Text\'
'Mehr Text'
user$ echo \'Noch mehr Text\"
'Noch mehr Text"
```

- Man nutzt aus, dass die Sonderfunktion des `"` innerhalb eines `'...'`-Konstruktes aufgehoben wird und umgekehrt:

```
user$ echo '"Ein Text"'
"Ein Text"
user$ echo "'Mehr Text'"
'Mehr Text'
user$ echo "'Noch mehr Text"'"'
'Noch mehr Text"
```

  Beachten Sie bei dem letzten Befehl, dass `"'Noch mehr Text"'"'` durch eine Verkettung von `"'Noch mehr Text"` und `'"'` zustande kommt! Da dies recht verwirrend ist, wäre hier eine *Kombination* aus `"..."`- und `\`-Quotierung etwas leichter lesbar:

```
user$ echo "'Noch mehr Text"\"
'Noch mehr Text"
```

Lösungsvorschlag zu Aufgabe 8.2

Der Backslash „\“ ist das Shell-Sonderzeichen, welches eine eventuelle Sonderfunktion des folgenden Zeichens aufhebt! In dem Beispiel

```
user$ echo C:\Benutzer\Thomas
C:BenutzerThomas
```

haben die Zeichen „`B`“ und „`T`“ *keine* Sonderfunktion, die aufgehoben werden müsste. Diese Zeichen werden einfach ausgegeben, der Backslash als Sonderzeichen wird bei der Ausgabe aber unterdrückt. Jede Quotierung der Zeichenkette `C:\Benutzer\Thomas` behebt das Problem. Man betrachte insbesondere den Befehl

```
user$ echo C:\\Benutzer\\Thomas
C:\Benutzer\Thomas
```

Lösungsvorschlag zu Aufgabe 8.3

Wir haben bisher folgende Sonderfunktionen für den Punkt kennengelernt: (1) In Pfadangaben steht der Punkt für das *cwd* (siehe Abschn. 5.2); (2) der doppelte Punkt in Pfadangaben steht für das Elternverzeichnis (siehe Abschn. 5.3); (3) am Anfang eines Datei- oder Verzeichnisnamens markiert er eine versteckte Datei, bzw. ein verstecktes Verzeichnis (siehe Abschn. 5.4). Außer an der ersten Stelle kann man den Punkt ohne *Nebenwirkungen* in Datei- und Verzeichnisnamen benutzen.

In Abschn. 10.2 werden wir den Punkt noch als *Alias* für den `source`-Befehl kennenlernen.

Lösungsvorschlag zu Aufgabe 8.4

```
user$ ls
-i  test_1.txt  test_2.txt
user$ rm -rf *
rm: remove regular file 'test_1.txt'? ('y' eingeben)
rm: remove regular file 'test_2.txt'? ('y' eingeben)
user$ ls
-i
```

Der Befehl `rm -rf *` wird hier effektiv als `rm -rf -i test_1.txt test_2.txt` von der Shell ausgeführt. Die Datei `-i` wird also als *Option* `-i` (*interactive*) interpretiert. Die Option bewirkt, dass vor dem Löschen jeder Datei explizit nachgefragt wird. Man kann diesen Trick benutzen, um sich in wichtigen Verzeichnissen vor einem versehentlichen `rm -rf *` mit möglicherweise schwerwiegenden Folgen zu schützen. Eine leere Datei mit Namen `-i` lässt sich beispielweise mit dem Befehl `touch -- -i` erzeugen.

Lösungsvorschlag zu Aufgabe 9.1

Die Aufgabe ist direkt mit dem unter Abschn. 9.2.2 Besprochenen zu lösen. Der Befehl `vi` führt zu einer Startmeldung des Editors `vi` und einem zunächst blokierten Terminal. Wenn Sie sich die Startseite genauer ansehen, so wird erläutert, dass die Tastenkombination `:q<Return>` benutzt werden kann, um den Editor wieder zu verlassen. Ansonsten müssen Sie in einem neuen Terminal die `PID` des Prozesses herausfinden und den `vi`-Prozess mit `kill` beenden. In einer Beispielsitzung bei mir lief es folgendermaßen ab:

```
user$ vi

# In einem neuen Terminal:
user$ ps -a
  PID TTY          TIME CMD
 4703 pts/19   00:00:00 ssh
 4800 pts/20   00:00:04 alpine
 9529 pts/22   00:00:00 vi
 9541 pts/13   00:00:00 ps
11709 pts/0    00:00:00 ssh
```

```
.
.
user$ kill 9529
```

Lösungsvorschlag zu Aufgabe 9.2

Das Problem hier ist, dass unser bisheriger Aufruf von `ps -a` nur Prozesse anzeigt, die in der gerade laufenden Sitzung mit einem Terminal verbunden sind. Nach dem kompletten Ausloggen existiert aber das Terminal, in dem das Programm `endless_process.sh` gestartet wurde, nicht mehr. Hier der Ablauf einer Beispielsitzung:

```
user$ nohup ./endless_process.sh &
nohup: ignoring input and appending output to \
'nohup.out'

user$ ps -a
 2170 pts/23   00:00:00 endless_process
 3561 pts/20   00:00:00 xterm
.
.
.


# Hier habe ich mich komplett aus- und wieder
# eingeloggt:
user$ ps -a
  PID TTY          TIME CMD
 2873 pts/20   00:00:00 xterm
 6826 pts/23   00:00:00 dbus-launch
.
.
```

Der Prozess `endless_process.sh` mit der `PID 2170` taucht nicht mehr auf! Ein Blick in die Dokumentation von `ps` gibt unter anderem folgende Möglichkeiten, um die `PID` des Prozesses zu bekommen:

```
# Liste 'alle' Prozesse des Benutzers thomas (Die
# Option '-u' steht für 'user'):
user$ ps -u thomas
  PID TTY          TIME CMD
.
.
 2170 ?        00:00:00 endless_process
 2802 pts/20   00:00:00 bash
.
.
```

Dies ergibt üblicherweise eine recht lange Liste, in der man den Prozess suchen muss. Das ? in der Spalte TTY besagt, dass der Prozess nicht mit einem Terminal der gerade laufenden Sitzung verbunden ist. Eine andere Möglichkeit ist:

```
# Liste Prozesse mit Namen 'endless_process.sh'
# (Option '-C' steht für 'command'):
user$ ps -C endless_process.sh
  PID TTY          TIME CMD
 2170 ?        00:00:00 endless_process
```

was *nur* Prozesse mit Namen endless_process.sh anzeigt. Der Prozess kann jetzt ohne Probleme entfernt werden:

```
user$ kill 2170
user$ ps -C endless_process.sh
PID TTY          TIME CMD
```

In Aufgabe 14.1 werden Sie noch eine weitere Möglichkeit zur Lösung mit ps kennenlernen. Die dort besprochene Methode ist flexibler als die hier gezeigten. Natürlich können Sie für die Aufgabe anstatt ps auch das Programm top verwenden!

Lösungsvorschlag zu Aufgabe 10.1

Der Befehl

```
echo$ export PATH=${PATH}:.
```

fügt den Programmsuchpfaden das *cwd* hinzu! Hiermit ist es möglich, *aus jedem* Verzeichnis Programme, die sich innerhalb dieses Verzeichnisses befinden, zu starten. Ein expliziter Pfad muss dann nicht mehr angegeben werden. Falls das *cwd* in Ihrer PATH-Variablen enthalten ist, lief das Eingangsbeispiel zur Programmausführung aus Abschn. 9.1 im Wesentlichen wie folgt ab:

```
user$ countdown_process.sh
countdown_process.sh: Permission denied
user$ chmod u+x countdown_process.sh
user$ countdown_process.sh
.
.
Die Variable 'i' hat jetzt den Wert 1
Die Variable 'i' hat jetzt den Wert 0

Mit dem Erreichen des Wertes 0 fuer die Variable 'i'
ist dieses Skript beendet.
```

Obwohl dies auf den ersten Blick attraktiv erscheint, rate ich Ihnen davon ab! Mit dieser Modifikation ist die Liste der direkt ausführbaren Programme nicht mehr *abgeschlossen*, sondern sie hängt vom *cwd* ab. Vor allem, wenn Sie einmal verschiedene Versionen eines Programms in mehreren Verzeichnissen haben, ist es

nur eine Frage der Zeit, bis *versehentlich* ein *verkehrtes* aufgerufen wird. Sobald Sie einmal selbst Programme schreiben, ist dies ein recht wahrscheinliches Szenario. Eine statische Liste an direkt ausführbaren Programmen ist also primär ein Schutz vor solchen potenziellen Problemen bei der Programmausführung. Aus diesem Grund ist das *cwd* in neueren *Unix*-Installationen standardmäßig nicht mehr in der `PATH`-Variablen enthalten.

Lösungsvorschlag zu Aufgabe 11.1

Der erste Befehl

```
user$ find /home/thomas -name '*.txt'
```

wurde im Text besprochen. Er liefert, startend mit dem Verzeichnis `/home/thomas`, alle Dateien und Verzeichnisse, die im Verzeichnisbaum auf `.txt` enden.

Der zweite Befehl

```
user$ find /home/thomas -name *.txt
```

kommt wahrscheinlich durch ein Vergessen der beabsichtigten Quotierung der Zeichenkette `*.txt` zustande. Im vorliegenden Fall kann dies zu sehr unverständlichen und von den konkreten Umständen abhängigen Ergebnissen führen! Die Shell ersetzt vor dem Aufruf des `find`-Befehls das Wildcard-`*.txt` durch alle Dateien/Verzeichnisse, die im *cwd*(!) mit der Endung `.txt` gefunden werden. Das Hauptproblem ist, dass das Ergebnis des Befehls primär davon abhängt, *wie viele* solche Dateien/Verzeichnisse sich im *cwd* befinden. Die zweite wichtige Rolle spielen *die Namen* dieser Dateien/Verzeichnisse:

1. Es befindet sich *genau eine* Datei oder *genau ein* Verzeichnis mit der Endung `.txt` im *cwd*. Nehmen wir als Beispiel eine Datei mit Namen `test.txt`. Dann führt der Befehl

   ```
   user$ find /home/thomas -name *.txt
   ```

   effektiv zum Aufruf des Befehls:

   ```
   user$ find /home/thomas -name test.txt
   ```

   Es werden also, startend von `/home/thomas`, im gesamten Verzeichnisbaum alle Dateien/Verzeichnisse mit dem Namen `test.txt` gelistet. Andere Dateien/Verzeichnisse mit der Endung `.txt` werden *nicht* berücksichtigt!
2. Es befinden sich *mehrere* Dateien/Verzeichnisse mit der Endung `.txt` im *cwd*. Seien dies zur Illustration die zwei Dateien `test1.txt` und `test2.txt`. Hier führt der Befehl

   ```
   user$ find /home/thomas -name *.txt
   ```

   effektiv zum Aufruf von:

   ```
   user$ find /home/thomas -name test1.txt test2.txt
   find: paths must precede expression: test2.txt
   Usage: find [-H] [-L] [-P] [-Olevel] \
   ```

```
[-D help|tree|search|stat|rates|opt|exec] \
[path...] [expression]
```

Dies führt zu einer Fehlermeldung, da eine ungültige Syntax des Befehls `find` vorliegt.

3. Es befinden sich *keine* Dateien/Verzeichnisse mit der Endung `.txt` im *cwd*. Hier kann die Shell beim Aufruf des Befehls

```
user$ find /home/thomas -name *.txt
```

das Muster `*.txt` nicht durch konkrete Dateien ersetzen, es bleibt stehen und kommt korrekt bei `find` an! In diesem Fall würde der Befehl *wie erwartet* funktionieren und man würde den Fehler unter Umständen gar nicht bemerken!

Lösungsvorschlag zu Aufgabe 11.2

a) Oft möchte man Dateien nicht nur finden, sondern mit ihnen danach noch irgendwelche Aktionen durchführen. Die `-exec` (*execute*)-Option des `find`-Kommandos dient genau dazu. Nach der Option steht das Kommando, das man mit jeder gefundenen Datei (oder jedem gefundenen Verzeichnis) ausführen möchte. Hierbei wird jeweils für den *Platzhalter* `{}` die gefundene Datei (bzw. das Verzeichnis) eingesetzt. Abgeschlossen wird die Option mit einem Semikolon, welches vor der Shell durch Quotierung geschützt werden muss. Im vorliegenden Fall:

```
user$ find /home/thomas -type f -name '*.txt' \
   -exec chmod g+r {} \;
```

wird den Gruppenmitgliedern für alle Dateien mit der Endung `.txt` das `r`-Recht gewährt. Das abschließende Semikolon ist nötig, um *das Ende* des Kommandos zu signalisieren. Man kann nämlich mehrere `-exec`-Optionen hintereinander geben, z. B.:

```
user$ find /home/thomas -type f -name '*.txt' \
   -exec chmod g+r {} \; -exec chmod o-r {} \;
```

b) Nach Bearbeitung von Teil a) sollten Sie hiermit keinerlei Probleme haben:

```
user$ cd ~
user$ find . -type f -name '*~' -exec rm {} \;
```

Lösungsvorschlag zu Aufgabe 11.3

Hauptproblem bei dieser Aufgabe ist wahrscheinlich das Aufspüren der korrekten Syntax in den Dokumentationen von `find`. Wir wollen hier alle Dateien finden, für welche die Benutzerklasse *others* das `w`-Recht besitzt. Dabei soll es nicht darauf ankommen, welche anderen Rechte sonst noch gesetzt sind. Die wesentliche Option für `find` ist hier `-perm -o+w`:

```
user$ cd ~
user$ find . -type f -perm -o+w
user$ touch test.txt
user$ ls -l test.txt
-rw-r--r-- 1 thomas users 0 Jun  1 17:00 test.txt
user$ find . -type f -perm -o+w
user$ chmod o+w test.txt
user$ ls -l test.txt
-rw-r--rw- 1 thomas users 0 Jun  1 17:00 test.txt
user$ find . -type f -perm -o+w
./test.txt
```

Man beachte, dass der Befehl `find . -type f -perm o+w` (ohne `-` vor dem `o+w`) die Datei nicht listen würde. Dieser Befehl würde nur Dateien finden, die *nur* das `w`-Recht für *others*, aber ansonsten *keinerlei* Rechte gesetzt haben:

```
user$ find . -type f -perm o+w
user$ chmod u=,g=,o=w test.txt
user$ ls -l test.txt
--------w- 1 thomas users 0 Jun  1 17:00 test.txt
user$ find . -type f -perm o+w
./test.txt
```

Lösungsvorschlag zu Aufgabe 13.1

Programme, die sich nach einem Aufruf ohne Argumente anscheinend *aufhängen*, warten auf Daten aus der Standardeingabe! Siehe dazu das Beispiel mit dem `sort`-Befehl in Abschn. 13.2. Das Phänomen ist für viele Anfänger sehr verwirrend. Oft ruft man ein Programm einfach auf, um eine Hilfe oder Fehlermeldung zu erhalten, welche auf notwendige Optionen und Argumente hinweist.

Lösungsvorschlag zu Aufgabe 13.2

Eine kleine Schwierigkeit bei dieser Aufgabe ist die korrekte Wahl der Optionen für `ls` und `wc`. Eine Liste aller Dateien und Verzeichnisse bekommen wir mit `ls -1` (Beachte: „`-1`“, „Eins“ und nicht „`-l`“, „L“!). Wir verwenden hier die Option `-1`, um genau eine Ausgabespalte pro Zeile zu bekommen. Die Gesamtzahl an Zeilen (bzw. Dateien und Verzeichnissen) bekommen wir dann mit `wc -l`:

```
user$ cd ~
user$ ls -1 | wc -l
149
```

Löst man die Aufgabe mit dem Befehl:

```
user$ cd ~
user$ ls -l | wc -l
150
```

so bekommt man eine um 1 zu hohe Zahl. Grund ist, dass der Befehl `ls -l` eine zusätzliche Zeile `total ...` in die Ausgabe einfügt:

```
user$ cd ~
user$ ls -l
total 77908
-rw-r--r--  1 thomas users 217 Apr 21 19:29 test.txt
.
.
```

Lösungsvorschlag zu Aufgabe 13.3

Der Befehl `head -n 120 exposures.txt` gibt uns die ersten 120 Zeilen der Datei `exposures.txt`. Von dieser Ausgabe greifen wir mit `tail -n 10` die letzten zehn Zeilen ab:

```
user$ head -n 120 exposures.txt | tail -n 10
696578p D3 03AQ02 1552 0.9 1 z
.
.
696587p D3 03AQ02 1552 0.99 1 g
```

Es geht in diesem Fall auch einfach:

```
user$ head -n 120 test.txt | tail
.
.
```

da eine Ausgabe von zehn Zeilen die Standardeinstellung von `tail` ist.

Lösungsvorschlag zu Aufgabe 13.4

In einer aus mehreren Kommandos bestehenden Pipeline hat man üblicherweise nur Zugriff auf die Ausgabe des letzten Kommandos. Hin und wieder ist man aber auch an der Ausgabe eines Pipelinezwischenschritts interessiert:

```
user$ ls -ltr | sort -k5,5 -n | awk '{print $5}'
```

Hier werden *nur* die Dateigrößen sortiert ausgegeben (siehe Abschn. 13.3). Das *Zwischenergebnis* nach dem `sort`-Befehl ist zunächst verloren. Das `tee`-Kommando erlaubt hier, die Ausgabe einer Pipeline an beliebiger Stelle in eine Datei *abzugreifen*:

```
user$ ls -ltr | sort -k5,5 -n | tee file.txt | \
  awk '{print $5}'
```

Hier erfolgt die endgültige Pipelineausgabe wie eben. Jedoch wird zusätzlich der Pipelinezustand nach `ls -ltr | sort -k5,5 -n` in die Datei `file.txt` geschrieben. In Abb. E.1 wird die Namensgebung von `tee` deutlich. Um mitten aus einer Pipeline etwas abzugreifen, kann man ein *T-Stück* verwenden! In dem gegebenen Beispiel:

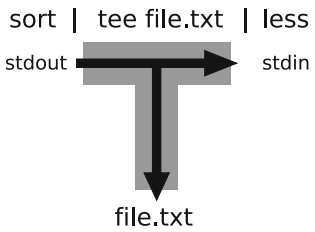

Abb. E.1 Um auf Daten mitten in einer Pipeline zuzugreifen, baut man sich ein *T-Stück* in dieselbe ein

```
user$ ls -ltr | sort -k5,5 -n | tee output.txt | less
```

möchte man offensichtlich eine längere Ausgabe mit `less` genauer betrachten und zusätzlich das Ergebnis zur weiteren Verarbeitung in einer Datei speichern.

Selbstverständlich kann der `tee`-Befehl beliebig oft in einer Pipeline verwendet werden.

Lösungsvorschlag zu Aufgabe 13.5

Das Kommando `xargs` führt seine Argumente als Befehl aus, wobei Daten, die über `STDIN` in `xargs` gepiped werden, als Argumente dieses Befehls dienen. Um den Befehl und den Unterschied zu Pipelines ohne `xargs` zu verdeutlichen, ein Beispiel:

```
user$ ls -1 *.txt | wc -l
15
```

Dieser Befehl zählt die Anzahl aller Textdateien in *cwd*.

```
user$ ls -1 *.txt | xargs wc -l
     74 CODEX.txt
     38 dominik.txt
      7 kitten.txt
     33 konrad.txt
.
.
```

Hier wird über `xargs` der Befehl `wc -l` mit der Argumentliste `CODEX.txt dominik.txt kitten.txt ...` (dem `STDOUT` Ergebnis von `ls -1`) aufgerufen. Er gibt somit die Anzahl an Zeilen in jeder der 15 Textdateien, die sich im *cwd* befinden. Der gezeigte Befehl ist natürlich umständlich und vollkommen identisch zu `wc -l *.txt` – er dient hier lediglich der Demonstration! Die Hauptanwendung von `xargs` ist in Kombination mit `find`. Der Befehl erlaubt es, mit gefundenen Dateien und Verzeichnissen noch eine Aktion auszuführen:

```
user$ find /home/thomas -type f -name '*.txt' | \
  xargs wc -l
```

Der Befehl ist fast wie obiger `ls`-Befehl. Es werden hier aber alle Textdateien im Verzeichnisbaum unter `/home/thomas` berücksichtigt. Das zweite Beispiel lautet:

```
user$ find . -type f -mtime -7 | \
  xargs tar -cvzf backup_der_letzten_Woche.tgz
```

Der Befehl `find . -type f -mtime -7` findet unter dem *cwd* alle Dateien, die in den letzten sieben Tagen modifiziert wurden. Diese werden über `xargs` als Argumente an `tar -cvzf backup_der_letzten_Woche.tgz` weitergereicht. Dieser Befehl wiederum archiviert all seine Argumente in der Datei `backup_der_letzten_Woche.tgz`. Die Pipeline ist somit sehr gut für selektive Backups geeignet!

Wir haben in Aufgabe 11.2 die `find`-Option `-exec` kennengelernt, um Aktionen mit gefundenen Dateien durchzuführen. In der Tat können für viele Aufgaben beide Methoden gleichwertig verwendet werden. Es wird oft argumentiert, dass die Pipelinemethode mit `xargs` *effizienter* ist. So wird in dem Beispiel: `find . -type f -name '*.aux' -exec rm {} \;` für jede gefundene Datei, die auf `.aux` endet, ein *separater* `rm`-Befehl erzeugt. Der Befehl führt also effektiv zu etwas wie:

```
user$ rm test.aux
user$ rm ./test/test1.aux
user$ rm ./test/neu/test2.aux
.
.
```

Hingegen wird bei der `xargs`-Alternative `find . -type f -name '*.aux' | xargs rm` nur ein `rm`-Befehl erzeugt:

```
user$ rm test.aux ./test/test1.aux \
./test/neu/test2.aux ...
```

In den allermeisten praktischen Fällen ist das aber bedeutungslos. Falls man allerdings einmal eine Aktion mit *sehr vielen* Dateien ausführen muss, ist `xargs` oft die einfachste Methode, um dies zu tun. Im folgenden Beispiel hatte ich im *cwd* 40 000(!) Dateien, die ich löschen musste:

```
user$ rm *
/bin/rm: cannot execute [Argument list too long]
```

Hier expandiert der `*` zu einer Argumentenliste, die wegen ihrer Länge von der Shell nicht mehr verarbeitet werden kann:

```
user$ find . -type f -name '*' | xargs rm
```

Dieser Befehl funktioniert in solchen Fällen ohne Probleme. `xargs` splittet hier automatisch zu lange Argumentlisten *intelligent* und ruft `rm` *genauso oft wie absolut nötig* auf! Es ist auch einer der Fälle, wo das obige Effizienzargument zwischen der `xargs`- und der `-exec`-Methode merkbar zum Tragen kommt.

Lösungsvorschlag zu Aufgabe 13.6

a) Der gegebene `find`-Befehl liefert eine Liste aller Dateien. Diese kann mit dem in Aufgabe 13.5 eingeführten `xargs`-Befehl an `ls -ltr` weitergereicht werden, was wiederum mit `sort` auf bekannte Weise bearbeitet wird (siehe Abschn. 13.3):

```
user$ find ~ -type f | xargs ls -ltr | sort -k5,5 -n
```

b) Die vorige Teilaufgabe liefert bereits eine, nach Dateigröße sortierte, Liste aller Dateien. Hier müssen wir lediglich die drei letzten Zeilen, welche die größten Dateien auflisten, mit einem `tail`-Befehl abgreifen:

```
user$ find ~ -type f | xargs ls -ltr | \
  sort -k5,5 -n | head -n 3
```

Lösungsvorschlag zu Aufgabe 14.1

Der Befehl `ps -eaf` gibt eine Liste aller auf dem System laufenden Prozesse. Der Benutzer, der einen Prozess gestartet hat, steht hierbei in der ersten Spalte. Für einen Benutzer `thomas` wäre eine naheliegende Lösung der Aufgabe durch

```
user$ ps -eaf | grep thomas
```

gegeben.

Man beachte jedoch folgende Feinheit: In der letzten Spalte der Ausgabe von `ps -eaf` steht der Name des Prozesses, welcher aufgerufen wurde. Falls es z. B. ein Programm mit Namen `thomas` gäbe, das gerade unter dem Namen eines anderen Benutzers läuft, so wäre die Aufgabe mit obigem Befehl formal nicht korrekt gelöst. Dasselbe gilt, falls es z. B. noch einen Benutzer `thomaserben` geben sollte. Obwohl das im vorliegenden Fall pedantisch erscheinen mag, sind solche Ausnahmefälle unter anderen Umständen wesentlich. Eine bessere Lösung wäre daher der folgende `grep`-Befehl:

```
user$ ps -eaf | grep '^thomas␣'
```

Der *Reguläre Ausdruck* `'^ thomas␣'` spricht nur Zeilen an, die mit der Zeichenkette `thomas␣` beginnen.

Lösungsvorschlag zu Aufgabe 14.2

Wir isolieren aus der Datei `exposures.txt` alle Zeilen, welche die Zeichenkette `D3` enthalten, und zählen sie mit `wc -l` (`-l` steht für *lines*):

```
user$ grep D3 exposures.txt | wc -l
3362
```

Wir haben also insgesamt 3362 Beobachtungen in der Region `D3`.

Lösungsvorschlag zu Aufgabe 14.3

Ich gebe hier nur eine knappe Formulierung der Lösung. Eventuell vorhandene Probleme sollten sich nach Bearbeitung von Aufgabe 13.5 und dem Studium ihrer Lösung erledigen.

Der erste Befehl

```
user$ find . -name '*.txt' -type f | \
  grep 'Test'
./texts/Ein_Test.txt
./Test.txt
.
.
```

liefert zunächst rekursiv unterhalb des *cwd* eine Liste aller Dateien, die auf `*.txt` enden. Der `grep`-Befehl filtert daraus alle Dateien, deren Namen `Test` enthält, heraus und listet die Namen dieser Dateien auf.

Der zweite Befehl

```
user$ find . -name '*.txt' -type f | \
  xargs grep 'Test'
./passwords.txt:Test678
./deutsch.txt:Ein kleiner Test für eine Datei ..
.
.
```

sucht ebenfalls alle Dateien unterhalb des *cwd*, welche auf `*.txt` enden. Diese Dateien werden durch `xargs` als *Argumente* an `grep 'Test'` weitergereicht. Dieser Befehl sucht dann *in* den Dateien nach Zeilen, welche die Zeichenkette `Test` enthalten, und gibt diese auf dem Bildschirm aus!

Lösungsvorschlag zu Aufgabe 14.4

Ein schnelles Betrachten der Datei `emails.txt` und das Anwenden der gegebenen Pipeline liefern:

```
user$ less emails.txt
.
wdqzceqaey329     <wdqzceqaey329@w709.bildundorf.de>,
n259.p798         <n259.p798@mausraub.de>,
p425.x904         <p425.x904@u776.b392.beinpixel.de>,
h368              <h368@mausraub.de>,
z593.o501         <z593.o501@l976.a394.bildundorf.de>,
.
user$ cat emails.txt | tr '<>' '\n' | grep @
.
wdqzceqaey329@w709.bildundorf.de
n259.p798@mausraub.de
p425.x904@u776.b392.beinpixel.de
```

```
h368@mausraub.de
z593.o501@l976.a394.bildundorf.de
.
```

Die Datei enthält in jeder Zeile einen fiktiven Benutzernamen, gefolgt von seiner E-Mail-Adresse, welche in spitzen Klammern eingeschlossen ist. Die gegebene Pipeline *isoliert* die E-Mail-Adressen (Aufgrund der Funktionsweise der Pipeline schreibe ich absichtlich nicht „Es wird alles außer den E-Mail-Adressen gelöscht.“!)

Die Pipeline erledigt diese Aufgabe in folgenden Etappen:

1. Der `cat`-Befehl gibt die Datei einfach aus und dient lediglich als Zulieferer für `tr`, das selbst direkt keine Dateien verarbeiten kann und auf `STDIN` als Eingabemedium angewiesen ist.
2. Der `tr`-Befehl ist leicht zu verstehen, sobald man herausgefunden hat, dass das Ersetzungszeichen `\n` für einen Zeilenumbruch steht! Der Befehl ersetzt also die beiden Klammerzeichen < und > jeweils durch einen Zeilenumbruch:

   ```
   user$ cat emails.txt | tr '<>' '\n' | less
   .
   ,
   wdqzceqaey329
   wdqzceqaey329@w709.bildundorf.de
   ,
   n259.p798
   n259.p798@mausraub.de
   .
   ```

3. Der folgende `grep`-Befehl sammelt alle Zeilen ein, die ein at-Zeichen (`@`) enthalten. Das `@`-Zeichen in einer Zeichenkette ist ein prominentes Alleinstellungsmerkmal für eine E-Mail-Adresse. Daher ist ein `grep` auf dieses Zeichen eine sehr gute Möglichkeit, in einer Textdatei Zeilen mit einer E-Mail-Adresse aufzuspüren. Die vorherigen Pipelineschritte haben praktischerweise diese Adressen in einzelne Zeilen isoliert.

Man präge sich die hier vorgestellte Technik gut ein. Sie wird häufig verwendet, um interessante Textelemente, die in längere Textzeilen eingebettet sind, zu isolieren und weiter zu verarbeiten.

Obwohl die Datei `nochmal_emails.txt` recht verschieden von `emails.txt` aufgebaut ist, liefert die Anwendung der Pipeline dasselbe Ergebnis:

```
user$ less nochmal_emails.txt
.
<wdqzceqaey329@...>, <n259.p798@...>,
<p425.x904@...>, <h368@...>,
<z593.o501@...>,
.
user$ cat nochmal_emails.txt | tr '<>' '\n' | grep @
```

```
.
wdqzceqaey329@w709.bildundorf.de
n259.p798@mausraub.de
p425.x904@u776.b392.beinpixel.de
h368@mausraub.de
z593.o501@l976.a394.bildundorf.de
.
```

Wie das Ergebnis zustande kommt, sollte Ihnen an dieser Stelle klar sein!

Lösungsvorschlag zu Aufgabe 14.5

Der Befehl `grep f | test.txt` versucht, eine Pipeline zwischen den Befehlen `grep` und `text.txt` aufzubauen. Da es kein Kommando mit dem Namen `text.txt` gibt, kommt es zu der gegebenen Fehlermeldung. Hier ist offensichtlich wieder einmal die Maskierung der Zeichenkette `f |` mit dem Sonderzeichen „|“ vergessen worden:

```
user$ grep 'f |' test.txt
.
| d | e | f | g |
.
```

Lösungsvorschlag zu Aufgabe 14.6

Die Aufgabe ist eine Wiederholung von Techniken, die im Text anhand der Datei `exposures.txt` besprochen wurden:

a) Die Liste wird nach der Anzahl der Einwohner (dritte Spalte) numerisch sortiert. Die kleinste und größte Stadt steht dann in der ersten und letzten Zeile der sortierten Datei:

```
user$ sort -n -k3,3 staedte.txt
Stadtlohn          Nordrhein-Westfalen        20005
.
.
Berlin             Berlin                     3421829
```

b) Die Städte mit einer Einwohnerzahl größer als 100 000 werden mit `awk` gefiltert und noch nach der letzten Spalte absteigend numerisch sortiert:

```
user$ awk '($3 > 100000)' staedte.txt | \
  sort -nr -k3,3
Berlin             Berlin                     3421829
.
.
Moers              Nordrhein-Westfalen        103108
```

c) Wir isolieren die Bundesländer und zählen mit `sort` und `uniq`, wie oft jedes Bundesland vorhanden ist. Danach wird die Liste numerisch nach der Häufigkeit absteigend sortiert. Zum Schluss werden die beiden Spalten vertauscht:

```
user$ awk '{print $2}' staedte.txt | sort | \
  uniq -c | sort -nr -k1,1 | \
  awk '{print $2, $1}'
Nordrhein-Westfalen 206
Baden-Württemberg 99
.
.
Hamburg 1
Berlin 1
```

Als Anmerkung gebe ich noch eine Version, in welcher der letzte `awk`-Befehl die Ausgabe *schöner* als Tabelle *formatiert*:

```
user$ awk '{print $2}' staedte.txt | sort | \
  uniq -c | sort -nr -k1,1 | \
  awk '{printf("%-25s %-4d\n", $2, $1)}'
Nordrhein-Westfalen       206
Baden-Württemberg         99
.
.
Hamburg                   1
Berlin                    1
```

Falls Sie dies einmal benötigen, so suchen Sie in der `awk`-Literatur unter dem Stichwort *formatierte Ausgabe*.

d) Es werden die Städte aus dem Bundesland Rheinland-Pfalz mit `grep` isoliert und die Summe der Werte in der dritten Spalte mit dem letzten `awk` Beispielbefehl aus Abschn. 14.2 berechnet:

```
user$ grep 'Rheinland-Pfalz' staedte.txt |\
  awk 'BEGIN {summe = 0} {\
       summe = summe + $3} END {print summe}'
1293716
```

Lösungsvorschlag zu Aufgabe 14.7

Eine Lösung der Aufgabe ist durch:

```
user$ awk '{print $2, $7}' exposures.txt | \
  sort | uniq | awk '{print $1}' | uniq -c | \
  awk '{print $2, $1}'
D1 5
D2 5
D3 5
```

```
D4 5
W1m0m0 5
W1m0m1 5
.
```

gegeben. Wir erstellen zunächst eine Liste, in der jede Kombination aus Beobachtungsregionen und Beobachtungskonfigurationen *genau einmal* vorkommt (die ersten `awk`, `sort` und `uniq`-Befehle):

```
user$ awk '{print $2, $7}' exposures.txt | \
  sort | uniq | less
D1 g
D1 i
.
.
D2 i
D2 r
.
.
W1m0m0 g
W1m0m0 i
W1m0m0 r
.
.
```

Jetzt muss nur noch gezählt werden, wie oft *jede Region* (erste Spalte) in dieser Liste vorkommt. Dies wird durch die Teilpipeline `awk '{print $1}' | uniq -c` erreicht. Man beachte, dass die Regionen bereits sortiert vorliegen und deshalb vor dem `uniq -c`-Befehl kein `sort` mehr nötig ist. Der abschließende `awk '{print $2, $1}'`-Befehl vertauscht lediglich die beiden Ausgabespalten.

Lösungsvorschlag zu Aufgabe 14.8

Der Übersichtlichkeit halber zeige ich hier zunächst eine Lösung in drei Schritten. Die grundlegende Idee ist es, zunächst die Beobachtungen mit den Qualitäts- und Wetteranforderungen zu filtern (Pipeline 1). Danach wird gezählt, wie oft jede Region in der resultierenden Liste vertreten ist (Pipeline 2). Die Anzahl an vorhandenen Beobachtungen wird gegen die nötige Anzahl an Aufnahmen abgeglichen (Pipeline 3):

```
# Filtere alle Beobachtungen mit Konfiguration 'i'
# und den Qualitätsanforderungen; behalte nur
# eine Spalte mit den Regionen:
user$ grep ' i$' exposures.txt | awk '($5 < 0.85)' | \
  awk '($6 < 3) {print $2}' > tmp.txt

# Verifiziere, wie oft jede Region in der Liste ist,
# d.h. wie oft wir sie bereits beobachtet haben:
```

```
user$ sort tmp.txt | uniq -c > tmp1.txt

# Gleiche die Anzahl an vorhandenen Beobachtungen 'N'
# gegen 7 ab. Dazu berechnen wir 7-N und behalten alle
# Zeilen, in denen das Ergebnis positiv ist. 7-N gibt
# in diesem Fall direkt an, wie viele Aufnahmen wir
# noch brauchen:
user$ awk '{print $2, 7 - $1}' tmp1.txt | \
  awk '($2 > 0)' > besorge.txt

# lösche temporäre Dateien:
user$ rm tmp.txt tmp1.txt
```

Beachten Sie, dass die Bedingung „Alle Regionen haben bereits mindestens eine Aufnahme mit den gegebenen Bedingungen." für ein korrektes Ergebnis wesentlich ist! Felder, die nur schlechte Beobachtungen haben, würden in der ersten Pipeline komplett herausgefiltert werden. Mit dem gegebenen Lösungsansatz müsste man diesen Fall bei Bedarf gesondert behandeln.

Wenn man möchte, kann man obige Schritte auch zu einer(!) Pipeline kombinieren:

```
user$ grep ' i$' exposures.txt | awk '($5 < 0.8)' | \
  awk '($6 < 3) {print $2}' | sort | uniq -c | \
  awk '{print $2, 7 - $1}' | awk '($2 > 0)' > \
  besorge.txt
```

Lösungsvorschlag zu Aufgabe 14.9

Zunächst muss man sich einen Überblick verschaffen, *was* die Datei `verfuegbar.txt` überhaupt enthält. Ich würde hier als Erstes etwa Folgendes machen:

```
user$ less verfuegbar.txt
1056957p
769079p
942766p
.
.

user$ wc verfuegbar.txt exposures.txt
 19678 verfuegbar.txt
 17899 exposures.txt
 37577 total
```

Die Datei enthält eine Spalte mit unsortierten Beobachtungs-IDs. Des Weiteren erwarten wir, dass uns 1779 Beobachtungen fehlen (die Differenz an Zeilen von `exposures.txt` und `verfuegbar.txt`). Eine Lösung der Aufgabe könnte jetzt wie folgt aussehen: Wir isolieren aus `exposures.txt` die Beobachtungs-IDs und kombinieren sie mit den IDs aus `verfuegbar.txt`. Diejenigen IDs, die

in der kombinierten Liste *genau einmal* vorkommen, sind Beobachtungen, die wir uns noch besorgen müssen:

```
# Isoliere die Beobachtungs-IDs aus exposures.txt:
user$ awk '{print $1}' exposures.txt > \
  IDs_vorhanden.txt

# Wir brauchen Beobachtungen, die in der kombinierten
# ID-Liste aus exposures.txt und verfuegbar.txt nur
# einmal auftreten:
user$ cat verfuegbar.txt IDs_vorhanden.txt | sort | \
  uniq -c | awk '($1 < 2) {print $2}' > brauche.txt

# Teste, ob die Anzahl der benötigten Aufnahmen der
# Differenz an Zeilen von verfuegbar.txt und
# exposures.txt entspricht:
user$ wc -l brauche.txt
 1779
```

Man mache sich klar, dass in dieser Lösung folgende *implizite* Annahmen enthalten sind: (1) Alle Beobachtungen, die in `exposures.txt` enthalten sind, finden sich auch in `verfuegbar.txt`; (2) die Liste `verfuegbar.txt` enthält jede Aufnahme *genau einmal*. In der Praxis passieren bei der Erstellung solcher Listen oft Fehler, die wir, vor allem bei großen Tabellen wie hier, oft nicht sofort bemerken. Es ist daher immer angebracht, zumindest einfache Tests des Resultats durchzuführen. In diesem Fall habe ich überprüft, ob die Anzahl der fehlenden Beobachtungen meinen Erwartungen entspricht. Bitte vollziehen Sie nach, dass obiger Test bei einer Verletzung einer der beiden impliziten Annahmen *im Allgemeinen* nicht das erwartete Ergebnis liefern würde!

Lösungsvorschlag zu Aufgabe 14.10

Das Kommando `comm` erlaubt es, zwei *sortierte* Dateien zeilenweise miteinander zu vergleichen und gemeinsame und/oder nur in einer Datei vorkommende Zeilen auszugeben:

```
user$ less namen1.txt
Albert
Bernhard
Stefan
Thomas
user$ less namen2.txt
Bernhard
Daniel
Erich
Stefan
user$ comm -12 namen1.txt namen2.txt
   # drucke nur gemeinsame Zeilen
```

```
Bernhard
Stefan
user$ comm -23 namen1.txt namen2.txt
   # drucke Zeilen, die nur in namen1.txt
   # enthalten sind
Albert
Thomas
```

Zur Lösung der Aufgabe können wir `comm` verwenden, um aus den bereits identifizierten Regionen in `g_r_i.txt` diejenigen auszusortieren, welche auch in `u` und/oder `z` beobachtet wurden:

```
# Erstelle, analog zum Text, eine Liste von
# Regionen, die in z beobachtet wurden:
user$ grep ' z$' exposures.txt | \
    awk '{print $2}' | sort | uniq > z.txt
#
# Sortiere in z.txt vorkommende Regionen aus
# g_r_i.txt aus:
user$ comm -23 g_r_i.txt z.txt > tmp.txt
#
# Wiederhole das Ganze mit Konfiguration u,
# wobei jetzt die Regionen in tmp.txt als
# Referenz dienen.
user$ grep ' u$' exposures.txt | \
    awk '{print $2}' | sort | uniq > u.txt
user$ comm -23 tmp.txt u.txt > nur_g_r_i.txt
user$ cat nur_g_r_i.txt
W2m2m0
W2m2m1
W2m2p1
W2m2p2
W2m2p3
```

Beachten Sie, dass alle Dateien, die hier mit `comm` verarbeitet werden, bereits sortiert sind! Sortierte Dateien sind für eine korrekte Funktionsweise des `comm`-Kommandos notwendig.

Lösungsvorschlag zu Aufgabe 14.11

Die Aufgabe sollte keine allzu großen Probleme bereiten, wenn Sie Aufgabe 14.4 bearbeitet und ihre Lösung verstanden haben.

Wenn Sie sich die Datei und die Ausgabe der Pipeline ansehen, stellen Sie fest, dass der Text in seine Worte zerlegt wird und die vorkommenden Worte aufgelistet werden. Die Ausgabeliste enthält in der ersten Spalte eine Zahl, die angibt, wie oft das Wort im Text vorkommt. Die zweite Spalte ist das Wort selbst. Die gesamte Liste ist absteigend nach der Häufigkeit der Worte sortiert.

Die einzelnen Schritte der Pipeline sind wie folgt:

1. Der `cat`-Befehl dient lediglich als *Zulieferer* für `tr`, welches seine Daten über `STDIN` beziehen muss.
2. Die Zeichen, die von `tr` durch einen Zeilenumbruch ersetzt werden, sind das *Komplement* (`tr`-Option `-c`) der Zeichen, die durch `[:alpha:]` abgedeckt werden. `[:alpha:]` repräsentiert sämtliche Buchstaben. Der `tr`-Befehl erzeugt effektiv eine Datei, in der jedes Wort des Eingabestroms auf einer einzelnen Zeile isoliert wird! Die `tr`-Option `-s` *(squeeze repeats)* sorgt dabei dafür, dass im Ausgabestrom keine Leerzeilen entstehen. Falls Ihnen das nicht klar ist, so studieren Sie die `man`-page des Befehls `tr` und sehen Sie sich die Effekte der Pipelines `cat moby_dick.txt | tr -c '[:alpha:]' '\n'` und `cat moby_dick.txt | tr -cs '[:alpha:]' '\n'` an!
3. Die restlichen, bereits bekannten, Konstrukte aus `sort`- und `uniq`-Befehlen sortieren die Worte, zählen ihr Vorkommen und erzeugen die oben beschriebene Endausgabe.

Lösungsvorschlag zu Aufgabe 15.1

Der Befehl `tar -czvf archiv.tgz DATEIEN` erzeugt ein komprimiertes `tar`-Archiv `archiv.tgz` aus allen `DATEIEN`; siehe Anhang B. Bei dem gegebenen Befehl wird in den Namen der Archivdatei durch eine Kommandosubstitution eine Datumsangabe integriert; siehe Abschn. 15.1.2. Die Datei hat letztendlich einen Namen wie `backup_2016-04-07.tgz`. Die zu archivierenden Dateien liefert in einer zweiten Kommandosubstitution der Befehl `find ${HOME} -name \*.txt -type f`, welcher, beginnend mit dem Heimatverzeichnis, rekursiv alle Dateien mit der Endung `.txt` erfasst.

Das Beispiel demonstriert, dass man Variablenauswertungen innerhalb von Kommandosubstitutionen nutzen kann. Wie erwartet, findet die Auswertung der einzelnen `$`-Konstrukte *nacheinander von innen nach außen* statt.

Lösungsvorschlag zu Aufgabe 15.2

Der gegebene Befehl unterscheidet sich von dem in Abschn. 15.1.2 ausführlich diskutierten Beispiel *nur* in dem Argument des `date`-Befehls, der in der Kommandosubstitution verwendet wird:

```
# Mit folgender Form des date-Befehls klappt alles:
user$ date +'%F'
    # Beachte den Formatstring '%F' und die folgende
    # Datumsangabe!
2016-04-07
user$ echo "Text in der Log-Datei" > \
   log_$(date +'%F').txt       # alles klar!

# Mit folgender Variante klappt es nicht mehr:
user$ date +'%D'
    # Der Formatstring '%D' führt zu wichtigen
```

```
    # Änderungen in der Datumsangabe!
04/07/16
user$ echo "Text in der Log-Datei" > \
   log_$(date +'%D_%T').txt       # UPS!
bash: log_$(date +'%D_%T').txt: No such file or ..
```

Die zweite Datumsangabe (`04/07/16`) enthält Slashes (`/`). Dieses Sonderzeichen dient in Datei- und Verzeichnisnamen zum Aufbau von Pfaden! Die Shell versucht, nach der Kommandosubstitution mit der Ausgabeumlenkung eine *Datei* wie `log_04/07/16.txt` anzulegen. Dies wäre eine Datei `16.txt` in dem Kind-Kind-Verzeichnis `log_04/07`. Da das Verzeichnis `log_04/07` nicht existiert, kommt es zu der Fehlermeldung.

```
user$ mkdir -p log_04/07
user$ echo "Text in der Log-Datei" > \
   log_$(date +'%D').txt
user$ ls log_04/07
16.txt
```

Man beachte, dass gerade bei Variablenauswertungen und Kommandosubstitutionen in Shell-Skripten die *unbewusste* Nutzung von Shell-Sonderzeichen in der Regel *sehr* problematisch ist!

Lösungsvorschlag zu Aufgabe 15.3

Das Kommando `who` liefert Informationen über gerade eingeloggte Benutzer auf Ihrer Maschine:

```
user$ who
thomas   :0           ...
thomas   pts/7        ...
.
.
terben   pts/20       ...
terben   pts/21       ...
thomas   pts/22       ...
```

Hieraus können wir die Namen und die Anzahl der Nutzer mit gewohnten Pipeline-Konstruktionen erhalten. Aus der ersten Spalte des `who`-Befehls müssen wir mehrfach vorkommende Einträge aussortieren:

```
user$ who | awk '{print $1}' | sort | uniq
thomas
terben
user$ who | awk '{print $1}' | sort | uniq | wc -l
2
```

Diese Information habe ich folgendermaßen in mein `status.sh`-Skript integriert:

```
user$ cat status.sh
#!/bin/bash
.
.
echo "Hallo ${USER}!"
.
.
echo "Ihr cwd ist $(pwd)"
N_USER=$(who | awk '{print $1}' | sort | uniq | wc -l)

echo "Folgende ${N_USER} Nutzer sind eingeloggt:"
who | awk '{print $1}' | sort | uniq
```

In meiner Lösung habe ich zunächst die Anzahl der Benutzer mithilfe einer Kommandosubstitution in einer Variablen gespeichert und diese danach in die Ausgabe integriert. Es ist hier natürlich auch möglich, die Kommandosubstitution via `echo "Folgende $(who | awk '{print $1}' | sort | uniq | wc -l) Nutzer ....."` direkt in den `echo`-Befehl einzubetten. Bei komplexeren Kommandosubstitutionen wie hier halte ich aber die Zwischenspeicherung in einer Variablen für übersichtlicher. Bei einem gut gewählten Variablennamen übernimmt die Variable zudem eine *Kommentarfunktion*.

Lösungsvorschlag zu Aufgabe 15.4

Das Kommandopaar `basename`/`dirname` erlaubt es, aus einer Pfadangabe das Verzeichnis und den Dateinamen zu isolieren. `basename` bietet darüber hinaus die Möglichkeit, Endungen aus Dateinamen abzuspalten:

```
# Isoliere aus einem Pfad das Verzeichnis:
user$ dirname /home/thomas/test.txt
/home/thomas

# Isoliere aus einem Pfad den Dateinamen:
user$ basename /home/thomas/test.txt
test.txt

# Isoliere aus einem Pfad den Dateinamen und
# die Endung '.txt' (zweites Argument an basename)
user$ basename /home/thomas/test.txt .txt
test
```

Hiermit sollte die geforderte `for`-Schleife keine Probleme bereiten. Eine Möglichkeit ist:

```
user$ cat asc.sh
for DATEI in text*.asc
do
  # Die Variable NAME enthält nach folgendem
  # Befehl den Dateinamen 'ohne' die Endung '.asc'!
  NAME=$(basename ${DATEI} .asc)

  # Benenne die Datei um
  mv ${DATEI} ${NAME}.txt
done
```

Lösungsvorschlag zu Aufgabe 15.5

In Abschn. 14.1 haben wir bereits eine Pipeline kennengelernt, die aus der Datei `exposures.txt` Beobachtungen aus einer bestimmten Region und einer bestimmten Konfiguration extrahiert. Damit können wir z. B. die Beobachtungs-IDs für Region `D1` mit Konfiguration `i` wie folgt erhalten:

```
user$ grep D1 exposures.txt | grep i | \
  awk '{print $1}' > D1_i.txt
```

Beachte, dass wir für eine Lösung unseres Problems diese Pipeline über *zwei* Parameter (Region und Konfiguration) iterieren müssen. Dies kann nicht mit *einer einzigen*, aber mit zwei *ineinander geschachtelten* Schleifen realisiert werden:

```
user$ cat regionen_zweite.sh
for REGION in D1 D2 D3 D4
do
  for KONFIGURATION in u g r i z
  do
    grep ${REGION} exposures.txt | \
      grep ${KONFIGURATION} | awk '{print $1}' > \
      ${REGION}_${KONFIGURATION}.txt
  done
done
```

Machen Sie sich gegebenenfalls die Funktionsweise dieses geschachtelten Konstrukts genau klar! Man beachte, dass die Einrückungen der einzelnen `do`–`done` Befehlsblöcke hier wesentlich für eine einfache Erfassung der, bereits etwas komplexeren, Strukturen sind.

Lösungsvorschlag zu Aufgabe 15.6

In Aufgabe 14.6, Teil d) haben Sie eine Pipeline entwickelt, um zu ermitteln, wie viele Menschen in den aufgeführten Städten des Bundeslandes Rheinland-Pfalz leben. Meine Lösung war:

```
user$ grep 'Rheinland-Pfalz' staedte.txt |\
  awk 'BEGIN {summe = 0} { \
       summe = summe + $3} END {print summe}'
1293716
```

Das folgende Shell-Skript ermittelt zunächst die vorhandenen Bundesländer und konstruiert danach eine `for`-Schleife um obige Pipeline herum. Der veränderliche Parameter ist hier das Bundesland.

```
user$ cat staedte.sh
#!/bin/bash

# Ermittle die in staedte.txt enthaltenen
# Bundeslaender (zweite Spalte) und speichere das
# Ergebnis in eine temporaere Datei:
awk '{print $2}' staedte.txt | sort | uniq > \
  laender.txt

# Jetzt die Iteration ueber die Bundeslaender zur
# Ermittlung der Einwohner:
for LAND in $(cat laender.txt)
do
  EINWOHNER=$(grep ${LAND} staedte.txt |\
    awk 'BEGIN {summe = 0} { \
       summe = summe + $3} END {print summe}')

  echo "Fuer ${LAND} zaehlen wir ${EINWOHNER} Menschen."
done

# Loesche die temporaere Datei wieder:
rm laender.txt
user$ ./staedte.sh
Fuer Baden-Wuerttemberg zaehlen wir 5280176 Menschen.
Fuer Bayern zaehlen wir 4818805 Menschen.
.
.
```

Lösungsvorschlag zu Aufgabe 15.7

Auf Basis der Lösung von Aufgabe 15.5 und den im Text besprochenen Konstrukten zur Behandlung von Skript-Argumenten sollten Sie hier keine Probleme haben. Eine Lösung wäre:

```
user$ cat regionen_dritte.sh
# Die Liste der ersten for-Schleife ist der einzige
```

```
# Unterschied zu regionen_zweite.sh
for REGION in $[->{@}
do
  for KONFIGURATION in u g r i z
  do
    grep ${REGION} exposures.txt | \
      grep ${KONFIGURATION} | awk '{print $1}' > \
      ${REGION}_${KONFIGURATION}.txt
  done
done
```

Der einzige Unterschied zu `regionen_zweite.sh` ist die erste Skriptzeile. Hier wird, anstatt über `D1–D4`, über die Skriptargumente iteriert.

Lösungsvorschlag zu Aufgabe 15.8
Meine Lösung ist wie folgt:

```
user$ cat backup.sh
#!/bin/bash

# Skript zum Backup eines Verzeichnisses

# das zu sichernde Verzeichnis:
VERZ=${1}
# Das Verzeichnis mit meinen Backups:
BACKUPS=${2}

# Wir brauchen das gegenwärtige Datum mehrmals;
# also Zwischenspeicherung in Variable:
ZEIT=$(date +'%F')

# Erstelle das Backup:
tar -czvf ${BACKUPS}/backup_${ZEIT}.tgz \
  $(find ${VERZ})

# Schreibe die Log-Datei:
# Falls wir einen 'relativen Pfad' sichern,
# haetten wir gerne Informationen ueber den
# absoluten. Dieser kann mit realpath ermittelt
# werden:
ABSPFAD=$(realpath ${VERZ})
echo "Es wurde ${ABSPFAD} gesichert." > \
  ${BACKUPS}/backup_${ZEIT}.log
```

Das Verständnis des Skripts sollte keine Probleme bereiten. Es ist etwas umfangreicher und enthält recht viele der im Text besprochenen Shell-Skripte-Elemente. Eine

Feinheit ist die Pfadangabe in der Log-Datei. Wir können mit unserem Skript auch *relative* Pfadangaben nutzen, etwa:

```
user$ ./backup.sh . ..
```

In diesem Fall ist ein Log-Datei-Eintrag wie „`Es wurde . gesichert`." nicht sehr hilfreich, um später nachzuvollziehen, welches Verzeichnis wir gesichert haben. Der `realpath`-Befehl ermittelt, wenn nötig, aus einer Verzeichnisangabe den entsprechenden absoluten Pfad, und diesen schreiben wir in die Log-Datei.

```
user$ pwd
/home/thomas/test
user$ realpath /home/thomas
/home/thomas
user$ realpath .
/home/thomas/test
```

Eventuell müssen Sie dieses Programm bei Ihnen nachinstallieren, falls Sie es nicht bereits auf Ihrem System haben.

Lösungsvorschlag zu Aufgabe 15.9

Das Problem ist, dass eine Überprüfung der Primzahleigenschaft nicht für alle Zahlen *gleich schwer* ist. Für größere Zahlen dauert sie in der Regel länger als für kleine. Eine zyklische Verteilung der Zahlen erzeugt eine Gleichverteilung von kleinen und großen Zahlen in allen Teilprozessen. Ersetzen wir jedoch in `prime_wrapper_dritte.sh` die Zeile `split -n r/${KERNE}` durch `split -n ${KERNE} ...`, so landen die größten Zahlen alle bei *einem einzigen* Prozess. Dieser bestimmt dann letztendlich die *effektive* Laufzeit unseres Programms. Für unsere Programmläufe bis zu einer Obergrenze von einer Million habe ich auf meiner Maschine noch keinen signifikanten Effekt gesehen:

```
# In folgender Variante unseres Wrappers wurde die
# Zeile 'split -n r/${KERNE} ...' durch
# 'split -n ${KERNE} ...' ersetzt:
user$ time ./prime_wrapper_dritte_schlecht.sh \
  1000000 > primzahlen.txt

real    0m6.049s
user    0m16.824s
sys     0m0.133s
```

Das ist zwar etwas langsamer als der Optimalwert, aber nicht weiter tragisch.

Bei deutlich höheren Obergrenzen wird es aber bemerkbar:

```
# Hier lassen wir alle Zahlen bis 10 Millionen(!)
```

```
# auf die Primzahleigenschaft testen:

# Zuerst mit der optimalen Aufteilung der Zahlen:
user$ time ./prime_wrapper_dritte.sh 10000000 > \
  primzahlen.txt

real    1m49.299s
user    6m25.793s
sys     0m0.439s

# Jetzt der Lauf mit der schlechteren
# Zahlenverteilung:
user$ time ./prime_wrapper_dritte_schlecht.sh \
  10000000 > primzahlen.txt

real    2m20.131s
user    6m31.252s
sys     0m0.570s
```

Das Beispiel verdeutlicht, dass bereits *einfache* Parallelisierungen gut durchdacht sein müssen, um eine *gute* Auslastung eines Mehrkernsystems zu gewährleisten.

Lösungsvorschlag zu Aufgabe A.1

a) Der RA „`[1-59]`“ repräsentiert eine Zeichenklasse, die aus den Ziffern 1–5 und der Ziffer 9 besteht. Falls Sie sich hier haben verwirren lassen, und auf Zahlen im Bereich 1–59 getippt haben: Beachten Sie, dass eine Zeichenklasse für *genau ein* Zeichen steht!

b) Beide Zeichenklassen sind eine Kombination des Bereichs `a-z` (Kleinbuchstaben) und eines Bindestrichs. Man beachte, dass man zur Vermeidung von Mehrdeutigkeiten mit Bereichen einen Bindestrich (`-`), der in eine Zeichenklasse integriert werden soll, entweder ganz am Anfang oder ganz am Ende der Zeichenklasse setzen muss!

Lösungsvorschlag zu Aufgabe A.2

Der RA „`[-]a`“ besteht aus einer Zeichenklasse, die nur den Bindestrich enthält, und einem nachfolgendem `a`. Der RA „`-a`“ hat keinerlei Metazeichen und sucht nach Zeichenketten, die aus einem Bindestrich gefolgt von einem `a` bestehen. Beides sind gültige RA, die effektiv dasselbe bewirken!

Der erste Befehl zeigt das erwartete Ergebnis. Im zweiten machen uns wieder einmal Shell-Sonderzeichen zu schaffen. `grep` interpretiert `-a` als Option anstatt als RA. Da `grep` keine Option `-a` besitzt, gibt es die Fehlermeldung. Das Problem hat also nichts mit einem fehlerhaften RA zu tun und ist ein Sonderfall, da ein `-` am Anfang einer Zeichenkette bei *Unix*-Befehlen eine Option markiert (siehe auch Kap. 8). Möchte man mit `grep` eine Zeichenkette suchen, die mit einem `-`

beginnt, so kann man entweder das Minus in eine Zeichenklasse packen oder z. B. den Befehl:

```
user$ echo "Testtext: -a" | grep -e '-a'
Testtext: -a
```

verwenden. Die Option `-e` sagt `grep` explizit, dass die folgende Zeichenkette ein RA und *keine* Option ist! Denselben Effekt kann man mit der in Kap. 8 vorgestellten Sonderoption `--` erreichen:

```
user$ echo "Testtext: -a" | grep -- '-a'
Testtext: -a
```

Lösungsvorschlag zu Aufgabe A.3

Die Shell-Wildcard-Zeichen werden gerne mit RA verwechselt, da sie teilweise dieselben Metazeichen mit ähnlicher, aber *nicht* genau derselben Bedeutung verwenden.

Das Wildcard-Muster `*.txt` ist komplett und steht für alle Dateien/Verzeichnisse, die auf `.txt` enden. Bei den RA ist der `*` lediglich ein *Zeichenmodifikator*, der ohne vorstehendes Zeichen (oder eine Zeichenklasse) alleine nicht nutzbar ist! Ein RA, der auf alle Zeichenketten trifft, die auf `.txt` enden, ist durch „`.*\.txt`“ gegeben. Der erste Punkt (ein RA-Metazeichen!) spricht *jedes beliebige* Zeichen an, das mit dem nachfolgenden `*` beliebig oft auftreten darf. Bei der abschließenden Zeichenkette `.txt` muss der Punkt gequotet werden, um seine Sonderfunktion aufzuheben. Man beachte, dass der RA „`.*.txt`“ alle Zeichenketten finden würde, die mindestens vier Zeichen haben und auf `txt` (ohne Punkt!) enden! Mit ähnlichen Argumenten ist das RA-Pendant zu dem Wildcard `?.txt` durch „`.\.txt`“ gegeben. Hier würde der RA „`.\?\.txt`“ auch die Zeichenkette `.txt` finden, welche im Wildcard-Fall ausgeschlossen ist! Im Fall des Wildcards `[0-9].txt` ist der entsprechende RA bis auf die nötige Quotierung des Punktes identisch: „`[0-9]\.txt`“.

Lösungsvorschlag zu Aufgabe A.4

a) Lösung ist die Zeichenklasse „`[1-47-9]`“.
b) Die Zahlen 1000–9999 sind eine Folge von vier Ziffern. Hierbei muss die erste Ziffer im Bereich 1–9 liegen, und die restlichen drei können eine beliebige Ziffer sein. Der RA „`[1-9][0-9]\{3\}`“ leistet dies. Es funktioniert hier natürlich auch „`[1-9][0-9][0-9][0-9]`“.
c) Der RA „`[a-z]*[aeiou][a-z]*[aeiou][a-z]*`“ bettet zwei zwingende Vokale in drei Zeichenketten aus beliebig vielen Kleinbuchstaben ein. Da *beliebig viele* Kleinbuchstaben auch *gar keinen* mit einschließt, erfüllt der RA die Bedingungen der Aufgabe.
d) Die Idee ist ähnlich wie in voriger Teilaufgabe. Wir betten die zwei erforderlichen Ziffern in drei Zeichenketten aus beliebigen Zeichen *außer Ziffern(!)* ein: „`[^0-9]*[0-9][^0-9]*[0-9][^0-9]*`“.
e) Die Lösung ist: „`[A-Z][a-z]\{,19\}`“

Lösungsvorschlag zu Aufgabe A.5

Zwei getrennte RA für die `D`- und `W`-Regionen anzugeben, bereitet keine großen Schwierigkeiten. Wir setzen direkt die Rahmenbedingungen aus Problem 3 auf Seite 152 um: „`D[1-4]`“ und „`W[1-4][mp][0-4][mp][0-4]`“.

Falls Sie sich über Alternativausdrücke erkundigt haben, so sind Sie sicherlich auf das Metazeichen `\|` (ein gequoteter Querstrich) gestoßen. Er erlaubt es, zwei RA zu verknüpfen und Textmuster zu treffen, die *entweder* dem RA links des „`\|`“ *oder* dem rechts davon entsprechen:

```
user$ grep 'D[1-4]\|W[1-4][mp][0-4][mp][0-4]'
695833 D3 03AQ02 1543 0.79 1 i
.
715802 D1 03BQ02 1703 0.62 1 i
.
715072 W1p2p3 03BQ02 1696 0.64 1 r
.
850000 W4p1m2 06AQ08 2703 0.62 1 r
.
```

Lösungsvorschlag zu Aufgabe A.6

Wir bauen den RA nacheinander auf, wobei wir schrittweise die acht Bedingungen implementieren:

1. Eine öffnende Klammer „(“ oder kein Zeichen:
   `(\?`
2. Die Zahl 0:
   `(\?0`
3. Eine Zahl zwischen 2 und 9:
   `(\?0[2-9]`
4. Eine Folge von Zahlen:
   `(\?0[2-9][0-9]*`
5. Kein Zeichen oder beliebig viele Leerzeichen:
   `(\?0[2-9][0-9]*␣*`
6. Kein Zeichen, eine schließende Klammer „)“ oder ein Slash „/“:
   `(\?0[2-9][0-9]*␣*[)/]\?`
7. Kein Zeichen oder beliebig viele Leerzeichen:
   `(\?0[2-9][0-9]*␣*[)/]\?␣*`
8. Eine Folge von Zahlen:
   `(\?0[2-9][0-9]*␣*[)/]\?␣*[0-9]*`

Das Beispiel verdeutlicht sehr schön, dass es nach einer vernünftigen Formulierung der Rahmenbedingungen recht einfach ist, auch komplexe RA schnell zu entwickeln. Wer allerdings *nur* den endgültigen RA zur Verfügung hat und diesen analysieren soll, tut sich oft sehr schwer. Dies gilt auch, wenn man *eigene* RA

nach längerer Zeit wieder anschauen muss, aber nicht mehr genau weiß, wofür sie gedacht waren. Eine gute Kommentierung komplexer RA ist daher sehr wichtig.

Lösungsvorschlag zu Aufgabe A.7

Spontan formulierte Rahmenbedingungen könnten so aussehen: (1) Ein Datum soll als Wort verankert sein; (2) als Erstes haben wir eine oder zwei Ziffern; (3) danach ein Zeichen aus der Menge `/-.`; (4) als Nächstes wieder eine oder zwei Ziffern; (5) erneut ein Zeichen aus der Menge `/-.`; (6) zwei *oder* vier Ziffern.

Bis auf Punkt (6) macht die Implementation eines RA für diese Aufgabe keinerlei Probleme. Ein RA für *zwei oder vier Ziffern* lässt sich mit dem hier besprochenen Funktionsumfang nicht direkt realisieren. Wie schon an anderer Stelle erwähnt, sollten wir *realistisch* davon ausgehen, dass nur *syntaktisch korrekte* Daten auftauchen und wir uns mit einer Zeichenkette wie `01/01/199` nicht befassen müssen. Beachten Sie bitte auch, dass obige Bedingungen auch Zeichenketten wie `01/01-2000` finden würden. Der Aufwand, einen RA für all diese *theoretisch möglichen* Spezialfälle zu entwerfen, steht in keinem Verhältnis zum Nutzen! Somit können wir es als vollkommen zufriedenstellend betrachten, wenn wir (6) durch die *schwächere* Bedingung „zwischen zwei und vier Ziffern" ersetzen:

```
user$ echo "1/1/99 01-01-1999 1.1.1999" |\
  grep \
  '\<[0-9]\{1,2\}[/.-][0-9]\{1,2\}[/.-][0-9]\{2,4\}'
1/1/99  01–01–1999  1.1.1999
```

Beachten Sie, dass innerhalb einer Zeichenklasse der Punkt *nicht* maskiert wird und für sich selbst steht. Der Bindestrich `-` muss am Anfang oder am Ende der Klasse stehen (Bereichsindikator; siehe auch Aufgabe A.2).

Lösungsvorschlag zu Aufgabe A.8

Wir haben in Kap. 6 gesehen, dass nach dem Befehl `ls -l` die erste Stelle des Rechtevektors angibt, ob es sich um eine Datei oder ein Verzeichnis handelt:

```
user$ ls -l
-rwxr-xr-x 1 thomas users 37 May 22 12:42 datei
drwxr-xr-x 2 thomas users  6 May 22 12:42 verzeichnis
```

Ein `-` am Anfang der Zeile signalisiert eine Datei, ein `d` ein Verzeichnis. Eine Pipeline, um nur Dateien zu listen, ist damit durch:

```
user$ ls -l | grep '^-'
-rwxr-xr-x 1 thomas users 37 May 22 12:42 datei
```

gegeben. Beachte die Verankerung des `-` am Anfang der Zeile! Analog für Verzeichnisse:

```
user$ ls -l | grep '^d' | awk '{print $9}'
verzeichnis
```

Hier wurde noch ein `awk`-Befehl angehängt, um alle Felder außer dem Verzeichnisnamen loszuwerden.

Lösungsvorschlag zu Aufgabe A.9

Ich habe die Seite http://de.wikipedia.org/wiki/E-Mail-Adresse zurate gezogen, um mich über den Aufbau von E-Mail-Adressen zu erkundigen. Ein prominentes Erkennungsmerkmal ist das at-Zeichen (@). Der Teil *vor* dem @ wird als *Lokalteil* bezeichnet und entspricht typischerweise einem Benutzernamen, z. B. `Max.Muster`. Der Teil nach dem @, der Domänenteil, entspricht der E-Mail-Domäne, z. B. `beispiel.de`.

Der Lokalteil ist *formal* eine Zeichenkette bestehend aus Buchstaben, Ziffern und recht vielen Sonderzeichen. Daneben gibt es noch etliche Sonderregeln. Zum Beispiel kann Text innerhalb von doppelten Anführungsstrichen stehen und dort weitere Sonderzeichen enthalten usw. Ich bin mir sicher, dass *formal* erlaubte Sonderzeichen wie Klammern, Dollarzeichen, das Fragezeichen und ähnliche in 99 % aller E-Mail-Adressen niemals auftauchen werden, und ich werde diese bei der Konstruktion meines RA außer Acht lassen. Der Domänenteil besteht aus drei Teilen: einem Hostnamen (z. B. ein Firmenname), einem Punkt und einer *Top-Level-Domain* (TLD; häufig ein Ländercode). Die TLD muss mindestens zwei Zeichen enthalten. Auch beim Domänenteil verzichte ich auf theoretisch mögliche Sonderzeichen und Sonderfälle, die mir in realen E-Mail-Adressen noch nie untergekommen sind.

Für eine einfache Lösung, die sicher den allergrößten Teil aller E-Mail-Adressen identifiziert, würde ich folgende Rahmenbedingungen definieren: (1) Für den Lokalteil erlaube ich eine Folge aus Buchstaben, Ziffern und der Zeichenmenge `-_.`. Ich verlange aber mindestens ein Zeichen; (2) das @-Zeichen; (3) der Hostname besteht aus einer Folge von Buchstaben, Ziffern und den Zeichen `-.`. Ich verlange mindestens ein Zeichen; (4) ein Punkt; (5) die TLD besteht nur aus Buchstaben. Es müssen mindestens zwei sein; (6) Wortverankerung der Adresse.

Direkte Umsetzung dieser Bedingungen führt auf:

1. Eine Folge aus Buchstaben, Ziffern und der Zeichenmenge „`-_.`“ mit mindestens einem Zeichen:
   `[A-Za-z0-9_.-]\+`
2. Das at-Zeichen:
   `[A-Za-z0-9_.-]\+@`
3. Eine Folge von Buchstaben, Ziffern und den Zeichen „`-.`“ mit mindestens einem Zeichen:
   `[A-Za-z0-9_.-]\+@[A-Za-z0-9.-]\+`
4. Ein Punkt:
   `[A-Za-z0-9_.-]\+@[A-Za-z0-9-]\+\.`
5. Mindestens zwei Buchstaben:
   `[A-Za-z0-9_.-]\+@[A-Za-z0-9-]\+\.[A-Za-z]\{2,\}`
6. Wortverankerung:
   `\<[A-Za-z0-9_.-]\+@[A-Za-z0-9-]\+\.[A-Za-z]\{2,\}\>`

Sachverzeichnis

T. Erben, *Einführung in Unix/Linux für Naturwissenschaftler*,
DOI 10.1007/978-3-662-50301-0

T