# 2 Unix/Linux und Shell-Programmierung

Warum wählen wir Unix/Linux als Einführung im Umgang mit Computern? Es gibt mehrere Gründe:

- Unix ist ein sauberes und transparent entworfenes Betriebssystem und daher für didaktische Zwecke gut geeignet.
- Unix erlaubt es dem Benutzer, sehr direkt mit dem Betriebssystemkern, dem sogenannten *Kernel*, zu interagieren. Man kann so die Funktionsweise des Computers und des Betriebssystem „fühlen“ und verstehen.

Für die Aneignung des Stoffes in diesem Kapitel genügt es *nicht*, sich einfach den Text durchzulesen. Um den Stoff tatsächlich anwenden zu können, ist es erforderlich, das Gelesene zu üben. Das ist vergleichbar mit dem Erlernen einer Fremdsprache, wo das bloße Lesen des Lehrbuchs ebenso nicht genügt, um die Sprache zu lernen. Viele Aspekte der Informatik haben einen ähnlichen Sprach-artigen Charakter und müssen durch Verwendung der Sprache geübt werden. Es wird empfohlen, alle Aufgaben aus diesem (und den nächsten) Kapitel *selbst* am Rechner durchzuspielen.

## 2.1 Grundlegendes

Noch bevor wir die ersten Unix-Kommandos kennen lernen, beschäftigen wir uns mit einigen grundlegenden Eigenschaften von Betriebssystemen im Allgemeinen und Eigenschaften des Unix/Linux-Betriebssytems im Speziellen.

### 2.1.1 Wozu dient ein Betriebssystem?

Um einen Computer nutzen zu können braucht man genau genommen nicht unbedingt ein Betriebssystem. Ohne Betriebssystem müsste aber ein Programmierer direkt mit der „nackten“ Maschine kommunizieren und seine eigenen Anweisungen schreiben unter anderem für …

- die Ansteuerung aller Datenträger, die verwendet werden; also auch der Festplatten, der CD- und DVD-Laufwerke, der USB-Laufwerke, usw. Dass solch eine Programmierung eines *Treibers*, wie man diese Ansteuerungssoftware auch oft nennt, nicht ganz einfach ist, kann man den Anfängen von Linux entnehmen: Anfangs programmierte Linus Torvalds nur einen einzigen Treiber für eine Festplatte,
- das Festlegen der Struktur der Dateien, der Struktur des Dateisystems und, falls es mehrere Nutzer geben kann, den Zugriffsregeln auf das Dateisystem,

– die Ansteuerung eines Bildschirms; man müsste Treibersoftware schreiben, die einzelne Pixel auf dem Bildschirm aufleuchten lassen kann und zwar so, dass ein für den Benutzer nützliches Bild entsteht.

Bis zum Jahre 1967 war genau dies notwendig. Für jeden erworbenen Rechner, der zum damaligen Zeitpunkt so teuer waren, dass er nur durch eine große Institution gekauft werden konnte, musste diese Institution genau diese Art von Steuersoftware selbst schreiben. 1967 führte IBM erstmals ein eigenes Betriebssystem ein, das OS/360, das modellübergreifend eingesetzt werden konnte. Übrigens war die Entwicklung von OS/360 so komplex und IBM auch unerfahren in Softwareprojekten dieser Größe, dass sich die Auslieferung um mehr als 2 Jahre verzögerte. Die Entwicklung soll 50 Millionen Dollar gekostet haben. Auf Codezeilen umgerechnet entspricht das 225 Dollar pro Zeile.

**Aufgabe 2.1**

OS/360 ist heute kostenlos im Internet verfügbar und kann mit Hilfe eines ebenfalls freien Emulators[a] sowohl auf Windows als auch auf Unix ausgeführt und getestet werden. Installieren Sie OS/360 auf Ihrem Rechner und spielen Sie etwas mit dem Betriebssystem.

***a*** Ein Emulator bildet ein bestimmtes Hardware- oder Softwaresystem in bestimmten Aspekten nach.

Um nochmals zusammenzufassen: Ein Betriebssystem ist:
– die Steuerungssoftware eines Computers
– die Schnittstelle zwischen den Rechnerkomponenten und Rechnerhardware (BIOS, Festplatten, Bildschirm-Controller usw.) und den Anwendungen, die auf dem Rechner laufen
– die Software zur Verwaltung und Pflege des Betriebs- und Dateisystems
– die Software zur Ansteuerung der Peripheriegeräte – also Geräte, die sich außerhalb des Rechners befinden, wie Maus, Tastatur, Drucker, usw.

Es gibt gute Gründe, die dafür sprechen, dass „closed source"-Betriebssysteme[1] Nachteile gegenüber „open source"-Betriebssystemen haben. Neal Stephenson, ein bekannter Science-Fiction-Autor und Aktivist für freie Software, schrieb in einem seiner Aufsätze:

> Es liegt in der Natur von Betriebssystemen, dass es keinen Sinn macht, dass sie von einer bestimmten Firma entwickelt werden, deren Eigentum sie dann sind. Es ist sowieso ein undankbarer Job. Applikationen schaffen Möglichkeiten für Millionen von leichtgläubigen Nutzern, wohin-

**1** „closed source" ist zu verstehen als das Gegenteil von „open source".

> gegen Betriebssysteme Tausenden von missmutigen Codern Beschränkungen auferlegen. Daher werden die Betriebssystemhersteller für immer auf der Hassliste aller stehen, die in der Hightech-Welt etwas zählen. Applikationen werden von Leuten benutzt, deren großes Problem es ist, alle ihre Funktionen zu begreifen, wohingegen auf Betriebssystemen Programmierer hacken, die von deren Beschränkungen genervt sind.[11]

### 2.1.2 Unix vs. Linux

Linux ist eigentlich kein Unix-Derivat, d. h. es wurde separat entwickelt; man bezeichnet jedoch Linux oft als ein „unixoides" System; es implementiert genau die für Unix typischen Betriebssystemfunktionalitäten. Da dieser Abschnitt sich nicht auf die Implementierung von Linux und Unix konzentriert, sondern fast ausschließlich auf deren Verwendung, ist es unerheblich, ob wir von Linux oder Unix sprechen; wir sprechen meistens von Unix, obwohl der Leser die meisten Konzepte vermutlich auf Linux ausprobieren wird. Aus Anwendersicht macht dies jedoch keinen Unterschied.

### 2.1.3 Der Aufbau von Linux

Unix ist ein *Multiuser*- und *Multitasking*-Betriebssystem, d. h. es können sowohl mehrere Benutzer (User) gleichzeitig an einem Rechner arbeiten, als auch mehrere Programme (Prozesse, Tasks) quasi gleichzeitig bearbeitet werden. Beide Eigenschaften stellen hohe Anforderungen an das Betriebssystem, denn Konflikte müssen verwaltet werden, die aus folgenden Situationen entstehen:

- gleichzeitiger Zugriff auf eine Datei
- gleichzeitiger Zugriff auf ein Gerät
- gleichzeitiger Zugriff auf den Arbeitsspeicher
- gleichzeitiger Zugriff auf die Rechenleistung des Prozessors

Das Kernstück von Linux, auch *Kernel* genannt, bildet eine „Trennschicht" zwischen der Hardware und einem Anwenderprogramm[2]. Diese Trennschicht sollte für das Anwenderprogramm undurchdringlich sein; benötigt ein Anwenderprogramm eine bestimmte Hardware, beispielsweise den Bildschirm, um Text auszugeben, dann sollte es nie direkt diese Hardware ansprechen, sondern nur den Kernel (über einen sogenannten Systemaufruf).

Der Kernel muss natürlich genau wissen, wie eine bestimmte Hardware angesprochen werden muss; dazu bedient er sich einer Treiberschnittstelle. Über diese Treiberschnittstelle kann er dann den eigentlichen speziellen Treiber für ein bestimmtes Gerät ansprechen. Abbildung 2.1 zeigt diesen Sachverhalt nochmals graphisch.

**2** Ein Anwenderprogramm ist beispielsweise ein Browser, Texteditor, oder ein Mailclient.

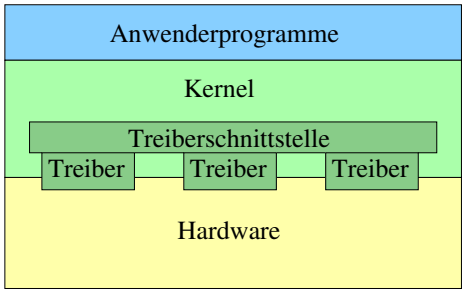



**Abb. 2.1.** Eine grobgranulare Veranschaulichung des Aufbaus von Unix

**Aufgabe 2.2**

Überlegen Sie sich, aus welchem praktischen Grund der Kernel eine sog. Treiberschnittstelle zur Verfügung stellt. Oder anders gefragt: Warum greift der Kernel nicht direkt auf den Treiber einer Hardware zu, sondern indirekt über die sog. Treiberschnittstelle?

### 2.1.4 Die Shell

Ein sehr spezielles Anwendungsprogramm ist die *Shell*. Die Bezeichnung „Shell", das englische Wort für „Schale", soll zum Ausdruck bringen, dass die Shell wie Schale das Betriebssystem umgibt. Sie erlaubt dem Benutzer durch Eingabe von Kommandos die direkte Kommunikation mit dem Betriebssystem. Die Unix-Shell entspricht dem `CMD.EXE`[3] in DOS, die meisten und unter Unix gängigen Shells sind aber in vielerlei Hinsicht mächtiger.

Unter Unix gibt es nicht nur *eine* Shell, wie unter DOS, sondern eine ganze Menge davon. Die historisch gesehen „erste", sehr einfach gestrickte Unix-Shell war die Bourne-Shell (aufrufbar durch `sh`), programmiert von Steven R. Bourne. Später gab es eine Reihe von Weiterentwicklungen, unter anderem die C-Shell (`csh`), die sich eng an der Programmiersprache C orientiert und die Korn-Shell. Die Free Software Foundation schließlich erweckte die alte Bourne-Shell wieder zum Leben und reicherte Sie mit den besten Features der C-Shell, der Korn-Shell und einiger anderer bis zu diesem Zeitpunkt entwickelter Shells an. Das Ergebnis nennt man – ein englisches Wortspiel – die „Bourne Again Shell", oder kurz: Bash.

Die Bash wird mit dem Redstone-Update ab August 2016 auch nativ, d.h. mit exakt den selben ausführbaren Dateien, unter Windows 10 laufen mit dem „Windows Sub-

**3** Man kann diese unter „Programme → Zubehör → Eingabeaufforderung" starten.

system for Linux“ (WSL). Alternativ kann man unter Windows auch Cygwin installieren, eine freie Software, die es erlaubt, auch unter Windows Unix/Linux-Programme laufen zu lassen. Dies wird durch die Bereitsstellung einer Kompatibilitätsschicht erreicht, die eine zu Unix/Linux kompatible Schnittstelle zur Verfügung stellt.

Wir empfehlen im Rahmen dieses Lehrbuchs ausschließlich mit der Bash zu arbeiten. Auch wenn sich heutzutage unter Unix mit Hilfe der modernen graphischen Benutzterschnittstellen wie Gnome oder KDE die allermeisten Aufgaben auch graphisch mit der Maus erledigen lassen, so bietet die Shell doch für fortgeschrittene Benutzer und für administrative Tätigkeiten noch immer die effizienteste Methode Aufgaben zu erledigen. Abbildung 2.2 zeigt ein Shell-Fenster.

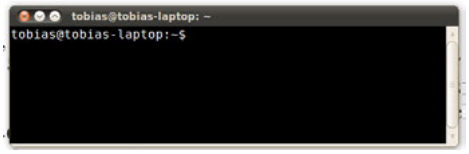



**Abb. 2.2.** Bash mit Eingabeaufforderung

Für alle Unix-Kommandos, die in diesem Kapitel vorgestellt werden, wird die Verwendung der Bash vorausgesetzt. In dieser können alle Befehle auch interaktiv eingegeben und getestet werden.

### 2.1.5 Multitasking

Wenn ein Benutzer ein Programm startet, stellt der Kernel einen Speicherabschnitt zur Verfügung, in den das Programm geladen wird. Zusätzlich werden verschiedene Verwaltungsdaten zu diesem Programm intern in einer Tabelle gespeichert. Sobald das Programm läuft, spricht man nicht mehr von einem „Programm“, sondern von einem „*Prozess*“.

Zur Realisierung des Multitasking muss das Betriebssystem tricksen, denn muss auch auf Rechnern laufen, die nur einen einzigen Prozessor haben[4], der zu einem bestimmten Zeitpunkt also auch immer nur einen (Maschinen-)Befehl-Prozess abarbeiten kann. Um den Anschein einer parallelen Ausführung zu erwecken, teilt Unix die verfügbare Rechenzeit des Prozessors in kleine „Zeitscheiben“ auf. Der sogenannte *Scheduler* teilt den einzelnen Prozessen dann bestimmte Zeitscheiben zu. Dieses Modell der Zeitscheiben-Zuteilung nennt man auch Multitasking. Unix verwendet das sogenannte *präemptive Multitasking*; der Unterschied zu anderen Multitasking-

**4** Die neueren Rechner, haben im Allgemeinen jedoch mehrere Prozessoren und könnten prinzipiell Aufgaben wirklich parallel auf mehreren Prozessoren gleichzeitig ausführen.

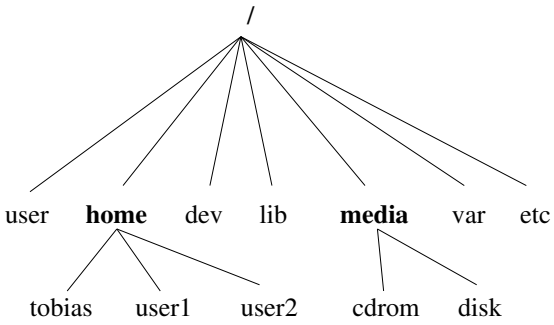



**Abb. 2.3.** Ausschnitt eines typischen Unix-Verzeichnisbaums.

Techniken ist, dass hier eine übergeordnete Instanz, nämlich der Scheduler, die Kontrolle über die Rechenzeit übernimmt.

### 2.1.6 Das Dateisystem von Unix

In Unix existiert ein einziger Verzeichnis„baum“. Das Wort „Baum“ stammt aus der Graphentheorie, und wird auch in anderen Teilgebieten der Informatik als Bezeichnung für sich verzweigende Strukturen benutzt. Bei einer graphischen Darstellung steht im Allgemeinen die „Wurzel“ des Baumes oben. Im Gegensatz zu Unix existieren unter Windows im Allgemeinen mehrere Verzeichnisbäume: Für jedes physikalische Laufwerk wird unter Windows ein eigener Dateibaum anlegt. Abbildung 2.3 zeigt einen Dateibaum eines Unix-Systems, wie er typischerweise durch ein Ubuntu-System angelegt wird. Verzeichnisse können sowohl einfache Dateien also auch wiederum Unterverzeichnisse enthalten. Das „oberste“ Verzeichnis wird unter Unix mit / – gesprochen: *root*[5] – bezeichnet. Abbildung 2.3 stellt graphisch dar, dass beispielsweise `home` ein Unterverzeichnis von / ist, und `user1` ein Unterverzeichnis von `home`. Das Zeichen / wird auch gleichzeitig zum Trennen von Datei- und Verzeichnisnamen verwendet.

Bei Datei- und Verzeichnisnamen unterscheidet Unix Groß- und Kleinschreibung; `README` und `Readme` sind also unterschiedliche Namen.

Der Begriff „Datei“ ist unter Unix weiter gefasst, als es Windows-Benutzer gewohnt sind. Unter Unix gibt es viele Dateiarten, von denen nur eine dem entspricht, was man gewöhnlich unter dem Begriff „Datei“ versteht:

- normale Dateien – Kürzel –
- Verzeichnisse – Kürzel `d`

**5** Der Begriff „root“ wird oft auch synonym zu „Administrator“ verwendet; der Grund ist, dass das „Home“-Verzeichnis des Administrators sich an der Wurzel des Verzeichnisbaumes befindet.

- symbolische Links – Kürzel `l`
- blockorientierte Geräte – Kürzel `b`
- zeichenorientierte Geräte – Kürzel `cb`
- Named Pipes – Kürzel `pcb`

Links verweisen nur auf eine Datei. Blockorientierte bzw. zeichenorientierte Geräte sind physikalische Medien, wie beispielsweise eine Festplatte oder eine parallele Schnittstelle. Jedes dieser Geräte ist über eine Datei in das Unix-System eingebunden. Gerätedateien befinden sich übrigens (meistens) im Verzeichnis `/dev` (= devices = Geräte). Mit *Pipes* können in Unix verschiedene Prozesse kommunizieren.

In Unix hat jede Datei einen Eigentümer sowie Benutzergruppen-Zugehörigkeiten. Eine Benutzergruppe könnte etwa „Student“, „Mitarbeiter“ oder „Professor“ sein. Zusätzlich besitzt jede Datei eine Menge von Attributen, die bestimmen, wer die Datei wie benutzen darf: `r` steht für „readable“, also lesbar; `w` steht für „writeable“, also beschreibbar; `x` steht für „executeable“, also ausführbar.

Mit dem Kommando

`ls` ⟨*Datei1*⟩ ⟨*Datei2*⟩ ...

kann man sich Informationen über eine bzw. mehrere Dateien aus dem aktuellen Arbeitsverzeichnis ausgeben lassen. Eine häufig gebrauchte Option ist `-l`, was eine Ausgabe von ausführlichen Informationen bewirkt. Führe ich beispielsweise das Kommando `ls -l einfinf.tex` in dem Verzeichnis auf meiner Festplatte aus, in dem ich an diesem Buch arbeite, so erhalte ich die folgende Ausgabe:

```
-rw-r-r- 1 tobias tobias 6717 2016-04-23 13:59 einfinf.tex
```

Diese Zeile beschreibt eine Datei, mit dem Namen `einfinf.tex`, die am 23. April 2016 um 13:59 Uhr zum letzten Mal verändert wurde. Sie ist 6717 Bytes groß, gehört dem User `tobias` und der Gruppe `tobias`. Die Zeichenkette `-rw-r-r-` hat die folgende Bedeutung:

| Dateiart | Eigentümer | Gruppe | Rest |
|---|---|---|---|
| - | rw- | r- | r- |
| Normale Datei | Lese- u. Schreibrechte | Nur Leserechte | Nur Leserechte |

## 2.2 Erste wichtige Kommandos

Dieser Abschnitt bietet einen ersten Überblick über die grundlegende Befehle zum Umgang mit Verzeichnissen, mit Dateien und zur Benutzer-Verwaltung.

### 2.2.1 Aufbau von Shell-Kommandos

Ein Kommando, das üblicherweise über die Shell eingegeben wird, hat in Unix die Form:

⟨*kommando*⟩ −⟨*option1*⟩ −⟨*option2*⟩ ... −⟨*longoption1*⟩ −⟨*longoption2*⟩ ... ⟨*file1*⟩ ⟨*file2*⟩ ...

Ein Kommando ist eine Folge von Zeichenketten, auch Strings genannt, die durch ein Leerzeichen oder einen Tabulator getrennt sind. Das erste Wort ist der Name des auszuführenden Kommandos. Die restlichen Wörter werden dem aufgerufenen Kommando als Argumente übergeben. Fehlt die Dateiliste, wird in der Regel die sogenannte *Standardeingabe* – normalerweise die Tastatur – als Eingabe verwendet.

In der Bash kann man zur Beschreibung von Dateinamen auch Wildcards verwenden. Wildcards sind Platzhalter für andere Zeichen. Häufig verwendet werden das Wildcard Stern (∗), das für eine beliebige Folge von Zeichen in einem Dateinamen steht, und das Wildcard Fragezeichen (?), das für genau ein beliebiges Zeichen in einem Dateinamen steht. DOS und Windows haben diese Methode zur Beschreibung von Dateinamen übernommen. Im Gegensatz zu DOS und Windows werden diese Wildcards jedoch nicht vom jeweiligen Programm, sondern von der Shell zu Dateinamen expandiert und erst nach dieser Expansion wird der eigentliche Befehl aufgerufen. Noch bevor ein Unix-Kommando in der Bash ausgeführt wird, ersetzt die Bash alle Wildcardzeichen der gesamten Kommandozeile durch die entsprechenden Dateinamen. Ein bestimmtes Unix-Kommando „sieht“ also diese Wildcards nie, sondern erhält als Argumente immer die von der Bash erzeugten Dateilisten.

Ein wichtiges erstes Kommando, das Sie oft benutzen sollten, ist

`man` *kommandoname*

das das Benutzerhandbuch (engl: manual) eines bestimmten Befehls anzeigt. Wichtige Abschnitte des Benutzerhandbuchs sind:

**Name:** Name und Kurzbeschreibung; diese Kurzbeschreibung erhält man auch durch den Befehl `whatis`.

**Synopsis:** Schema der Argumente, Optionen und oder Parameter. Sind sie optional, werden sie in eckige Klammern eingeschlossen; drei Punkte geben mögliche Wiederholungen an.

**Description:** Hier wird die Funktionsweise des Befehls genau beschrieben.

**Options:** Beschreibung der einzelnen Optionen; in diesem Abschnitt werden Sie vermutlich oft herumstöbern.

**Environment:** Benutzte Umgebungsvariablen (siehe Abschnitt 2.4.4).

**See also:** Querverweise zu Benutzerhandbüchern anderer Befehle.

Mit `q` können Sie das Benutzerhandbuch wieder verlassen.

**Aufgabe 2.3**

Experimentieren sie mit dem `man`-Kommando. Geben Sie in einer Shell das Kommando

```
> man man
```

ein und „durchforsten“ Sie dieses Benutzerhandbuch.

**Aufgabe 2.4**

Mit welchem Befehl können Sie sich …

(a) die Dateien des aktuellen Verzeichnisses sortiert nach Ihrer Größe ausgeben lassen?
(b) die Dateien des aktuellen Verzeichnisses sortiert nach dem Datum der letzten Änderung ausgeben lassen?
(c) alle Dateien – auch die versteckten – des aktuellen Verzeichnisses ausgeben lassen?
(d) die ausführliche Liste aller – auch der versteckten – Dateien sortiert nach dem Datum der letzten Änderung ausgeben lassen?

Hinweis: Verwenden Sie den `man`-Befehl, um herauszufinden, welche Optionen Sie für die jeweiligen Aufgaben benötigen.

### 2.2.2 Befehle für Verzeichnisse

Die folgende Tabelle zeigt wichtige Befehle für den Umgang mit Verzeichnissen.

| | |
|---|---|
| `cd` ⟨*Verzeichnis*⟩ | Wechselt ins angegebene Verzeichnis; es muss sich – wie übrigens bei allen Befehlen – immer ein Leerzeichen zwischen Befehlsname und Argument befinden. |
| `mkdir` ⟨*Verzeichnis*⟩ | Erstellt ein neues Verzeichnis. |
| `rmdir` ⟨*Verzeichnis*⟩ | Löscht ein leeres Verzeichnis. |
| `pwd` | (= print working directory) Gibt das aktuelle Arbeitsverzeichnis aus. |
| `ls` ⟨*Verzeichnis*⟩ | Zeigt an, welche Dateien sich im Verzeichnis befinden. Wird das Argument weggelassen so wird der Inhalt des aktuellen Arbeitsverzeichnisses ausgegeben. (Entspricht übrigens dem `dir`-Befehl von DOS). |

| | |
|---|---|
| `file` ⟨*Datei*⟩ | Gibt an, welchen Typ die als Argument übergebene Datei hat. Hierbei wird nicht die Endung der Datei betrachtet, sondern `file` untersucht den Inhalt der Datei und sucht nach Anhaltspunkten. |

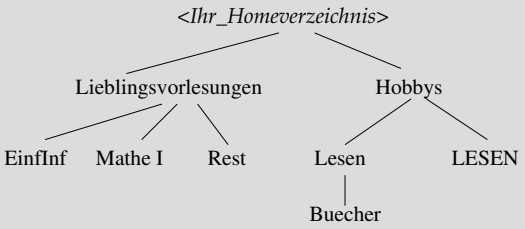
Statt eines Verzeichnisses kann auch immer `..` oder `.` angegeben werden: Dabei steht `..` für das Verzeichnis über dem aktuellen Verzeichnis und `.` steht für das aktuelle Verzeichnis. Beispiele: `cd ..` wechselt ins darüberliegende Verzeichnis und der Befehl `cp ../main.c .` kopiert die Datei `main.c`, die sich im darüberliegenden Verzeichnis befindet, ins aktuelle Verzeichnis. Das Verzeichnis, in dem wir uns aktuell befinden, wird meist als das *Arbeitsverzeichnis* bezeichnet.

**Aufgabe 2.5**

Erstellen Sie unter Ihrem Home-Verzeichnis, also dem Verzeichnis

`/home/`⟨*IhrName*⟩

den in der folgenden Abbildung dargestellten Verzeichnisbaum.




**Aufgabe 2.6**

Finden Sie auf Ihrem Unixsystem einige Gerätedateien; was liefert der Befehl `file` für ein Ergebnis, wenn Sie als Argument eine Gerätedatei übergeben?

### 2.2.3 Befehle für Dateien

Die folgende Tabelle zeigt die wichtigsten Befehle für den Umgang mit Dateien:

| | | |
|---|---|---|
| `cp` | ⟨*Datei1*⟩ ⟨*Datei2*⟩ ... ⟨*Ziel*⟩ | Kopiert die Datei(en) ins Ziel. Das Argument ⟨*Ziel*⟩ kann entweder eine Datei oder ein Verzeichnis sein. Werden dem Befehl mehrere (Quell-)Dateien übergeben, so *muss* das Ziel ein Verzeichnis sein. |

| | | |
|---|---|---|
| `cp -r` | ⟨*Datei1*⟩ ⟨*Datei2*⟩ … ⟨*Ziel*⟩ | Kopiert (rekursiv) ganze Verzeichnisbäume mit, d. h. die Inhalte aller Verzeichnisse des werden mitkopiert. |
| `cp -a` | ⟨*Datei1*⟩ ⟨*Datei2*⟩ … ⟨*Ziel*⟩ | Kopiert (rekursiv) ganze Verzeichnisbäume und lässt zudem alle Attribute (wie Zeitstempel, Zugriffsrechte, …) unverändert. |
| `mv` | ⟨*Datei*⟩ ⟨*Ziel*⟩ | (= move) Verschiebt die Datei nach ⟨*Ziel*⟩; dies kann sowohl ein Verzeichnis als auch ein Dateiname sein. |
| `rm` | ⟨*Datei1*⟩ ⟨*Datei2*⟩ … | (= remove) Löscht Dateien. Optionen: `-r` löscht Dateien und ganze Verzeichnisbäume; `-i` löscht Dateien nach Abfrage. |
| `touch` | ⟨*Name*⟩ | Erzeugt eine leere Datei mit dem als Argument übergebenen Namen. |
| `cat` | ⟨*Datei1*⟩ ⟨*Datei2*⟩ … | Gibt den Inhalt der als Parameter übergebenen Dateien aus. |
| `less` | ⟨*Datei*⟩ | Komfortableres Lesen einer Textdatei, falls diese größer als eine Bildschirmseite ist. Beenden des Lesens erfolgt mit `q`. |

**Aufgabe 2.7**

(a) Erzeugen Sie in Ihrem Home-Verzeichnis ein leeres Verzeichnis `Vom` und ein leeres Verzeichnis `Nach`.
(b) Erzeugen Sie nun im Verzeichnis `Vom` zwei (leere) Dateien `bla` und `blubber`.
(c) Kopieren Sie nun – mit einem einzigen Befehl – die beiden Dateien ins Verzeichnis `Nach`.
(d) Kopieren Sie nun – mit einem einzigen Befehl – die beiden Dateien ins Verzeichnis `Nach`, so dass alle Attribute der beiden Dateien (also Datum der letzten Veränderung, Rechte, …) erhalten bleiben.

**Aufgabe 2.8**

(a) Erstellen Sie mit einem beliebigen Texteditor eine Textdatei, die mindestens so groß ist, dass man deren Inhalt nicht auf einer einzelnen Bildschirmseite betrachten kann.
(b) Betrachten Sie nur die Textdatei mit dem `cat`-Kommando.
(c) Betrachten Sie nun die Textdatei mit dem `less`-Kommando.

(d) Betrachten Sie nun die Textdatei mit dem `more`-Kommando, das ähnlich arbeitet, wie das `less`-Kommando. Was ist der Unterschied zum Betrachten mit `less`?

### 2.2.4 Befehle für die Benutzerverwaltung

Die folgende Tabelle zeigt die wichtigsten Befehle für die Benutzerverwaltung:

| | |
|---|---|
| `who` | Zeigt die Benutzernamen der im System eingeloggten Personen an. |
| `whoami` | Gibt aus, unter welchem Namen man gerade eingeloggt ist. |
| `passwd` | Programm zum Ändern des eigenen Passworts. |
| `chmod` ⟨*Datei*⟩ | Kann die *mode bits* einer Datei, d. h. die Zugriffsrechte der Datei ändern. |

Wie schon oben erwähnt besitzt jede Datei *mode bits*, die die Zugriffsrechte für diese Datei regeln. Es gibt zwei Methoden diese Bits zu setzen. Zum einen mittels der Optionen `+x` (Hinzufügen von Ausführungsrechten), `+w` (Hinzufügen von Schreibrechten) und `+r` (Hinzufügen von Leserechten). Stellt man diesen Optionen statt eines `+` ein `-` voran, so nimmt man die entsprechenden Rechte weg. Vor den Optionen kann noch einer oder mehrere der Buchstaben `u` für „user“ (also der Eigentümer der Datei), `g` für „group“ und/oder `o` für „others“ stehen. Einige Beispiele:

| | |
|---|---|
| `chmod +x` | Ausführungsrechte für alle hinzufügen. |
| `chmod +r` | Leserechte für alle hinzufügen. |
| `chmod o+w` | Schreibrechte für den „others“ hinzufügen |
| `chmod ug-rw` | Schreib- und Leserechte dem „user“ und der „group“ wegnehmen. |

Die andere Methode, die „mode bits“ zu setzen, besteht darin, die Bits für user, group und others jeweils als Oktalzahl anzugeben. Hierbei wird für das Setzen des `r`-Bits eine 4, für das Setzen des `w`-Bits eine 2 und für das Setzen des `x`-Bits eine 1 addiert. Einige Beispiele:

| | |
|---|---|
| `chmod 754 datei.txt` | Rechtebits: `rwx r-x r--` |
| `chmod 740 datei.txt` | Rechtebits: `rwx r-- ---` |

**Aufgabe 2.9**

Sie wollen, dass user und group alle Rechte auf die Datei `bla` haben, others dagegen nur Ausführungsrechte.

(a) Geben Sie den entsprechenden `chmod`-Befehl an, der dies mittels Optionen bewerkstelligt.
(b) Geben Sie den entsprechenden `chmod`-Befehl an, der dies mittels Oktalzahlen bewerkstelligt.

**Aufgabe 2.10**

Geben Sie einen chmod-Befehl mit der jeweils anderen Methode an, der das gleiche macht wie …

(a) …`chmod 731 blubber.txt`
(b) …`chmod +w bla.txt`
(c) …`chmod 526 laber.txt`

### 2.2.5 Befehle des Prozesssystems

Mit den folgenden Befehlen können Sie sich laufende Prozesse auflisten lassen bzw. beenden:

| | |
|---|---|
| `ps` | (=process status) Listet die Prozesse auf, die momentan laufen (`ps` kann dynamische Änderungen nicht darstellen).<br>Parameter: `-l`: Langes Format; `-u`: User Format (d. h. mit User und Startzeit); `a`: Alle Prozesse (auch die anderer User); `-x`: Auch Daemon[6]-Prozesse; `f`: (= forest = Wald) Der ganze Prozessbaum wird dargestellt. |
| `top` | Statt einer Momentaufnahme, wie `ps`, liefert `top` eine sich ständig aktualisierende Liste der laufenden Prozesse. Leider werden aber nur so viele Prozesse aufgelistet, wie auf eine Bildschirmseite passen. |
| `kill` | Mit `kill` kann man Prozessen Signale schicken; es wird oft benutzt, um einen Prozess (evtl. „gewaltsam“) zu beenden. |
| `killall` | Ähnlich wie das `kill`-Kommando, nur können mit1G `killall` die betreffenden Prozesse über deren Namen angesprochen werden. |

**6** Als Daemon-Prozesse bezeichnet man diejenigen Prozesse, die über keine „TTY-Leitung“ verfügen, d. h., die nicht für eine Ausgabe auf dem Bildschirm vorgesehen sind.

**Aufgabe 2.11**

Ruft man den `top`-Befehl auf, so ist eingestellt, dass alle 5 Sekunden die angezeigten Prozesse aktualisiert werden. Finden Sie heraus, wie sie einen Aktualisierungszyklus von 1 Sekunde einstellen können.

**Aufgabe 2.12**

Starten Sie `firefox` und verwenden Sie danach den Befehl `kill`, um den soeben gestarteten `firefox`-Prozess „gewaltsam“ zu beenden.

**Aufgabe 2.13**

Was bewirkt :

```
$ killall killall
```

Das `$`-Zeichen am Zeilenanfang soll das Prompt, also die Eingabeaufforderung der Bash, darstellen.

### 2.2.6 Sonstige Befehle

| | |
|---|---|
| `echo` ⟨*string1*⟩ ⟨*string2*⟩ … | Gibt die übergebenen Strings auf dem Bildschirm aus. |
| `wc` ⟨*datei1*⟩ ⟨*datei2*⟩ … | (= word count) Gibt aus, wie viele Zeilen, Wörter und Zeichen jede der als Parameter übergebenen Dateien hat. Werden `wc` keine Argumente übergeben, so liest `wc` von der Tastatur. |
| `du` | (= disk usage) Gibt an, wie viel Platz auf dem Speichermedium verbraucht ist. |
| `tar` | (= tape archive) Archivierungsprogramm, mit dem Dateien zusammengepackt bzw. wieder extrahiert werden können.<br>Die erste Option muss entweder `x` (Extract - auspacken), `c` (Create - Archiv erzeugen), `r` (Append - Datei ans Ende eines bestehenden Archivs anhängen), `A` (Catenate - Ein Tar-File an ein Anderes anhängen) oder `t` (List - Inhalt eines Archivs auflisten) sein. |

| | |
|---|---|
| | Folgende Optionen können sein: `f` (File - Verwende die folgende Datei als Archiv), `z` - (Zip - die eingepackten Dateien werden zusätzlich komprimiert), `v` - (Verbose - die Namen der eingepackten Dateien werden zusätzlich ausgegeben). |

Beipielsweise bewirkt das Kommando

```
tar xvzf backup.tar Files/
```

dass das Verzeichnis `Files/` komprimiert in das tar-Archiv `backup.tar` gepackt wird.

**Aufgabe 2.14**

Sie wollen wissen, wie viel Platz, gemessen in Gigabyte, ihr Homeverzeichnis in Anspruch nimmt. Verwenden Sie das Kommando `du`, um sich die entsprechende Zahl ausgeben zu lassen.
Vermutlich müssen Sie mit `man` etwas im Benutzerhandbuch blättern.

**Aufgabe 2.15**

Erstellen Sie eine kleine Textdatei `test.txt`.

(a) Verwenden Sie nun den `wc`-Befehl, um sich nur die Anzahl der Zeilen in `test.txt` ausgeben zu lassen.
(b) Verwenden Sie nun den `wc`-Befehl, um sich nur die Anzahl der Zeichen in `test.txt` ausgeben zu lassen.

Vermutlich müssen Sie auch hier im Benutzerhandbuch blättern.

**Aufgabe 2.16**

Der `echo`-Befehl platziert nach Ausgabe des letzten Strings immer einen Zeilenvorschub (newline). Mit welcher Option können Sie diese verhindern?
Vermutlich müssen Sie auch hier im Benutzerhandbuch blättern.

## 2.3 Textdateien erstellen und editieren mit `vi`

In Unix sind alle Konfigurationsdateien reine Textdateien. Das wichtigste Werkzeug der Systemverwaltung ist daher ein Texteditor. Unix bietet dazu verschiedene Editoren an. Ein beliebter Texteditor ist `vi`. Ein weiterer sehr mächtiger Editor ist der

Emacs (Escape Meta Alt Control Shift), dessen Bedienung über verschiedenste Tastenkombinationen läuft. Wir lernen hier den zwar anfangs gewöhnungsbedürftigen aber schnellen, mächtigen und praktischen `vi` kennen. Dieser hat zudem den Vorteil, dass er nicht an eine graphische Benutzeroberfläche gebunden ist, d. h. rein „konsolenorientiert“ ausgeführt wird.

Der `vi`-Editor wird einfach durch

`vi` ⟨*Datei1*⟩ ⟨*Datei2*⟩ …

gestartet. Ganz nützlich ist die Option `-p`: mit dieser öffnet `vi` für jede angegebene Datei einen eigenen *Tabulator* – die graphische Metapher einer Registerkarte.

Der `vi`-Editor besitzt mehrere Modi: Die wichtigsten sind der Kommandomodus, in dem sich `vi` nach dem Starten befindet, und der eigentliche Eingabemodus. Im Kommandomodus werden eingegebene Zeichen als Befehle interpretiert; im Eingabemodus werden die eingegebenen Zeichen in die Textdatei geschrieben. Man kann vom Eingabe in den Kommandomodus wechseln, indem man die Esc-Taste drückt.

Der Umgang mit dem `vi`-Editor mag für den Neuling, der bisher nur mit graphischen Oberflächen gearbeitet hat, etwas gewöhnungsbedürftig sein; mit den unten aufgeführten, relativ wenigen wichtigsten Kommandos und wenigen Tagen Übung, werden Sie mit hoher Wahrscheinlichkeit produktiver sein als mit einem herkömmlichen GUI-basierten Editor.

Es folgen nun die (für den Anfang) wichtigsten `vi`-Kommandos (die natürlich nur im Kommandomodus funktionieren):

**Wechsel in den Eingabemodus**

| | |
|---|---|
| `i`, `I` | Einfügen vor dem Cursor, am Anfang der Zeile; |
| `a`, `A` | Einfügen nach dem Cursor, am Ende der Zeile; |
| `r`, `R` | Ersetzen eines Zeichens, vieler Zeichen. |

**Löschen und Einfügen von Text**

| | |
|---|---|
| `x`,`X` | Löscht ein Zeichen rechts, links; |
| `D` | Löscht den Rest der Zeile; |
| `dd` | Löscht die ganze aktuelle Zeile; |
| `dw` | Löscht das nächste Wort; |
| `yy` | Speichert die aktuelle Zeile in die Zwischenablage; |
| `p`, `P` | Einfügen nach, Einfügen vor. |

**Bewegen des Cursors**

| | |
|---|---|
| `w`, `W` | Nächstes Wort, nächstes Leerzeichen-getrenntes Wort; |

| | |
|---|---|
| `h`, `j`, `k`, `l` | links, runter, hoch, rechts; |
| `(`, `)` | Satz zurück, nach vorne; |
| `1G`, `G` | Anfang, Ende der Datei; |
| `nG` | Zeile `n`; |
| `H`, `M`, `L` | Oben, mitte, unten der Bildschirmansicht. |

**Suchen und Ersetzen**

| | |
|---|---|
| `/` ⟨*string*⟩ | Sucht ⟨*string*⟩ nach vorne |
| `?` ⟨*string*⟩ | Sucht ⟨*string*⟩ rückwärts |
| `:s/`⟨*pattern*⟩`/`⟨*string*⟩`/`⟨*flags*⟩ | Ersetzt das ⟨*pattern*⟩ mit ⟨*string*⟩. ⟨*flag*⟩=`g` – alle Vorkommen jeder Zeile; ⟨*flag*⟩=`c` – Vor jeder Ersetzung fragen. |

Das Argument ⟨*pattern*⟩ des `:s`-Befehls ist ein sogenannter „regulärer Ausdruck“; wie dieser aufgebaut ist, wird später genau erklärt. Inzwischen ist es vielleicht ganz nützlich zu wissen, dass ein einfacher String auch ein regulärer Ausdruck ist.

**Dateien**

| | |
|---|---|
| `:w` ⟨*datei*⟩ | Schreibt den editierten Text in ⟨*datei*⟩ bzw. in die aktuelle Datei, falls kein Argument angegeben. |
| `:wq` | Schreibt editierten Text in aktuelle Datei und verlässt `vi`. |
| `:w >>`⟨*datei*⟩ | Hängt editierten Text ans Ende von ⟨*datei*⟩. |
| `:r` ⟨*datei*⟩ | Setzt Inhalt von ⟨*datei*⟩ hinter die aktuelle Zeile. |
| `:r !`⟨*programm*⟩ | Setzt die Ausgabe von ⟨*programm*⟩ hinter die aktuelle Zeile. |

**Sonstiges**

| | |
|---|---|
| `J` | Fügt die nächste Zeile an die aktuelle an. |
| `.` | Wiederholt das letzte (textverändernde) Kommando. |
| `u`, `U` | Macht die letzte Änderung rückgängig. Macht alle Änderungen an der aktuellen Zeile rückgängig. |

## 2.4 Features der Shell

### 2.4.1 Eingabe

Jede Instanz der Shell hält sich einen Kommandozeilenspeicher und merkt sich so die letzten 500 Befehle (oder je nachdem was voreingestellt ist). Mit den Tasten ↑ (Pfeil nach oben) und ↓ (Pfeil nach unten) kann man in diesem Kommandozeilenspei-

cher blättern. Außerdem kann man mit Strg-`R` rückwärts im Kommandozeilenspeicher nach einem Befehl suchen. Jedes danach eingegebene Zeichen verursacht sofort einen Sprung auf die nächste passende Befehlszeile.

Außerdem bietet die Shell eine Kommandoerweiterung an, die es erlaubt, eine Befehlszeile sehr schnell und fehlerfrei einzugeben. Man muss Kommandos und Dateinamen nicht vollständig eintippen, sondern es genügt, den/die ersten Buchstaben einzugeben und danach die Tab-Taste zu drücken. Sollte es mehrere Alternativen geben, so gibt die Shell nach erneutem Drücken der Tab-Taste alle Möglichkeiten von Erweiterungen an.

### 2.4.2 Wildcards

Beginnen wir mit einem Beispiel und nehmen wir an, ein Buchdokument sei über mehrere Textdateien verteilt, sagen wir `ch1.1`, `ch1.2`, ..., `ch1.8` und `ch2.1`, `ch2.2`, ..., `ch2.8`. Man könnte dann das ganze Dokument folgendermaßen ausgeben:

```
cat ch1.1 ch1.2 ch1.3 ... ch2.1 ch2.2 ... ch2.8
```

Es ginge aber leichter durch Eingabe des Kommandos `cat ch[12].*`. Der Stern `*` und die eckigen Klammern `[...]` sind Beispiele für Wildcards.

Fast alle Unix-Befehle können mehrere Dateien gleichzeitig verarbeiten und erwarten eine ganze Liste von Dateien als Parameter. Wird ein Wildcard statt einer Liste angegeben, so ersetzt die Shell, *bevor* der Befehl ausgeführt wird, das Wildcard mit der Liste der dazu passenden Dateinamen; diese Liste wird dann dem Programm zur Ausführung übergeben. Das Programm `cat` im Beispiel oben, bekommt also das Joker-Zeichen nie zu sehen, sondern nur die durch die Shell expandierte Liste der Dateinamen.

Die Bash kann folgende Wildcards verarbeiten:

`*` Der Stern steht für jede beliebige, auch eine leere, Zeichenfolge. So steht der Stern alleine für alle Dateien im aktuellen Verzeichnis; ein Parameter `*.*` steht für jede Datei, die irgendwo einen Punkt im Namen hat.

`?` Das Fragezeichen steht für genau ein beliebiges Zeichen.

`[...]` Die eckigen Klammern stehen für genau eines der in die Klammern eingeschlossenen Zeichen. Möchte man beispielsweise alle Jpeg-Dateien ansprechen, egal ob die Endung groß oder klein geschrieben ist, könnte man das Muster `*.[Jj][Pp][Gg]` verwenden. In den eckigen Klammern sind auch Bereichsangaben möglich: Beispielsweise steht das Muster `[a-d]` für ein beliebiges Zeichen zwischen `a` und `d`, oder das Muster `[0-9]` steht für eine beliebige Ziffer.

`[!...]` Steht direkt nach der öffnenden eckigen Klammer ein Ausrufezeichen, so steht dieses Muster für ein Zeichen, das *nicht* in der nachfolgenden Zeichenmenge vorkommt.

Wichtig zu erwähnen ist noch, dass diese Muster immer nur schon existierende Dateien im aktuellen Verzeichnis erkennen können. Es ist nicht möglich, mit Hilfe von Mustern neue Dateien zu erzeugen.

**Aufgabe 2.17**

(a) Wie können Sie sich alle Dateien in Ihrem Arbeitsverzeichnis anzeigen lassen, die mit einer Ziffer enden?
(b) Wie können Sie sich alle Dateien auf dem Bildschirm ausgeben lassen, die mindestens eine Ziffer enthalten und nicht mit einer Ziffer enden?
(c) Wie können Sie sich alle Dateien im Arbeitsverzeichnis anzeigen lassen, deren Namen aus genau drei Zeichen bestehen?
(d) Wie können Sie sich alle Dateien im Arbeitsverzeichnis anzeigen lassen, die mit einem kleinen Buchstaben anfangen, irgendwo einen Punkt enthalten und mit einem großen Buchstaben enden?

**Aufgabe 2.18**

Angenommen, ihr Arbeitsverzeichnis enthält Buchkapitel mit Namen `kap1.txt`, ..., `kap6.txt`. Sie wollen diese Dateien umbenennen in `Kapitel1.txt`, ..., `Kapitel6.txt`. Was ist falsch an dem Versuch dies folgendermaßen zu tun:

```
$ mv kap*.txt Kapitel*.txt
```

### 2.4.3 Umleitungen und Pipes

**Umleitungen**

Ein- und Ausgabedatenströme eines Programms können umgeleitet werden – die Ausgabe eines Kommandos kann so beispielsweise statt wie üblich auf den Bildschirm in eine Datei umgeleitet werden. Jedes Programm besitzt in Unix automatisch drei Kanäle: Die Standardeingabe (stdin), die Standardausgabe (stdout) und die Standardfehlerausgabe (stderr). Normalerweise ist stdin mit der Tastatur verbunden, stdout und stderr sind mit dem Bildschirm verbunden.

Ein erstes Beispiel: Der Befehl `ls -l >dateien.txt` würde die Liste der Dateien im Arbeitsverzeichnis nicht auf dem Bildschirm ausgeben, sondern in die Datei `dateien.txt` schreiben. Es folgt eine Liste der wichtigsten Umleitungsmöglicheiten.

| | |
|---|---|
| ⟨*programm*⟩ >⟨*datei*⟩ | Die Standardausgabe von ⟨*programm*⟩ wird in die Datei ⟨*datei*⟩ umgelenkt; existiert ⟨*datei*⟩ schon, so wird sie einfach überschrieben und der bisherige Inhalt geht verloren. |
| ⟨*programm*⟩ >>⟨*datei*⟩ | Auch in diesem Fall wird die Standardausgabe von ⟨*programm*⟩ in ⟨*datei*⟩ umgeleitet; existiert ⟨*datei*⟩schon, so wird die Ausgabe ans Ende der Datei angehängt. |
| ⟨*programm*⟩ 2>⟨*datei*⟩ | Die Fehlerausgabe von ⟨*programm*⟩ wird in ⟨*datei*⟩ geschrieben. |
| ⟨*programm*⟩ <⟨*datei*⟩ | Die Standardeingabe von ⟨*programm*⟩ wird aus ⟨*datei*⟩ gelesen statt von der Tastatur. |

Man kann auch die Ein- bzw. Ausgabe eines ganzen Code-Blocks umleiten. Dieser muss dann in geschweiften Klammern eingeschlossen sein. Man kann etwa durch folgendes Kommando zwei Textzeilen in die Datei `hallowelt.txt` schreiben:

```
\{
 echo "hallo"
 echo "Welt"
\} >hallowelt.txt
```

Falls man verhindern möchte, dass ein Kommando eine Ausgabe oder Fehlerausgabe erzeugt, so kann man dies einfach dadurch erreichen, dass man seine Ausgabe bzw. Fehlerausgabe nach `/dev/null` umleitet. Das logische „Gerät“ `/dev/null` schluckt alle Eingaben und liefert nichts zurück; es kann als eine Art Mülleimer für unerwünschte oder uninteressante Ausgaben eines Befehls verwendet werden: Man kann beispielsweise folgendermaßen die Fehlerausgabe des `ls`-Kommandos unterdrücken und dadurch etwa verhindern, dass eine Fehlermeldung ausgegeben wird, wenn sich keine `*.c`-Dateien im aktuellen Verzeichnis befinden.

```
ls -l *.c 2>/dev/null
```

**Aufgabe 2.19**

(a) Erzeugen Sie mit einem Unixkommando eine Datei `test.txt`, die den Text `Ich heisse test` enthält.
(b) Angenommen, sie haben in ihrem aktuellen Arbeitsverzeichnis mehrere C-Dateien (mit Endung `.c`). Geben Sie ein Unix-Kommando an, das alle C-Dateien aneinanderhängt und in der Datei `alleC.c` speichert.
(c) Betrachten Sie die beiden Kommandos `wc -l test.txt` und `wc -l <test.txt`. Wie unterscheiden sich die Ausgaben? Erklären Sie die Unterschiede.
(d) Welches Ergebnis liefert der Befehl `wc -l test.txt >test.txt`?

### Pipes

Wollen wir beispielsweise wissen, wie viele Dateien sich im aktuellen Verzeichnis befinden, so könnten wir die entsprechende Anzahl folgendermaßen erhalten:

```
$ ls -l >temp
$ wc -l <temp
43
```

Der Befehl `wc` erhält also das als Eingabe, was `ls -l` als Ausgabe erzeugt hat und bestimmt so die Anzahl der Zeilen, die das `ls`-Kommando als Ausgabe produziert hat. Solche Situationen, in denen man die Standardausgabe eines Befehls mit der Standardeingabe eines anderen Befehls verbinden will, gibt es häufig. Unix stellt hierfür eine elegantere und schnellere Lösung bereit als die Verwendung einer temporären Datei, nämlich die Verwendung sogenannter *Pipes*.

Mit Hilfe von Pipes ist es möglich, die Standardausgabe eines Programms direkt und ohne den Umweg über eine temporäre Datei mit der Standardeingabe eines anderen Programms zu verbinden; diese Verbindung wird mit dem Symbol „`|`“ beschrieben. Um die Anzahl der Dateien im aktuellen Verzeichnis zu zählen, könnte man also genauso die folgende Konstruktion verwenden:

```
ls -l | wc -l
```

Das Ergebnis ist zwar dasselbe, die Betriebssystem-interne Ausführung unterscheidet sich jedoch in zwei entscheidenden Punkten: **1.** Statt einer auf der Festplatte befindlichen temporären Datei verwendet das Betriebssystem einen Speicher-internen Buffer[7]. Die Zugriffszeiten auf die Festplatte sind um mehr als den Faktor 100 größer als die Zugriffszeiten auf den Hauptspeicher. **2.** Das Betriebssystem kann die Ausführung der beiden Befehle `ls` und `wc` teilweise parallel ablaufen lassen, was sich in vielen Fällen auch positiv auf die Laufzeit auswirkt.

Übrigens hat die Pipe der Bash viel gemein mit der Funktionskomposition $\circ$ aus der Mathematik: Die Komposition der beiden Funktionen $g$ und $f$ – geschrieben $g \circ f$, gesprochen „$g$ nach $f$“ – angewandt auf ein Argument $x$ wendet zunächst $x$ auf $g$ an und speist dann den Rückgabewert – also $g(x)$ – in die Funktion $f$ ein. Ersetzen wir „Funktion“ durch „Kommando“ und „Rückgabewert“ durch „Standardausgabe“, so sieht man die Analogie zwischen `g | f` und $f \circ g$. Abbildung 2.4 veranschaulicht diesen Zusammenhang graphisch.

**Aufgabe 2.20**

Was geben die folgenden Anweisungen aus?

(a) `who | wc -l`

7 Man bezeichnet häufig einen kleinen temporären Speicher als „Buffer“.

(b) `ls -l | wc -l | wc -l`
(c) `ls -l | wc -l | cat`
(d) `ls -l | less`

**Aufgabe 2.21**

Verwenden Sie Pipes, um folgende Befehlssequenzen zu verkürzen (das Zeichen `$` ist Prompt einer Unix-Shell):

(a)
```
$ ls -la >temp
$ wc -l temp
```

(b)
```
$ ls -l >temp
$ grep hallo temp >temp2
$ wc -l temp2
```

(c)
```
$ grep hallo file1 >file2
$ grep welt file2
```

### 2.4.4 Shellvariablen

Shellvariablen, auch Umgebungsvariablen genannt, sind Platzhalter für eine bestimmte Zeichenkette. Im Gegensatz zu höheren Programmiersprachen, deren Variablen unterschiedliche Typen wie Integer, Boolean oder Float haben können, kennt die Shell als Datentyp nur Strings, also Zeichenketten. Eine Shellvariable kann man durch

$$\boxed{\langle variablenname\rangle = \langle String\rangle}$$

erzeugen. Man beachte, dass vor und nach dem =-Zeichen kein Leerzeichen stehen darf.

Der Befehl

`set`

zeigt alle mometan definierten Umgebungsvariablen an.

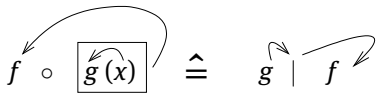



**Abb. 2.4.** Die Pipe der Bash entspricht der Funktionskomposition der Mathematik – bis auf die Tatsache, dass die Funktions-Argumente der Funktionskomposition ∘ in der umgekehrten Reihenfolge angeben werden, als die der Pipe.

Um auf den Inhalt einer Variablen zugreifen zu können, muss dem Variablennamen ein das Zeichen „>“ vorangestellt werden. Ein Beispiel (das Zeichen `$` an jedem Zeilenanfang soll das Prompt[8] sein):

```
$ meinName=Tobias
$ meinAlter=44
$ echo $meinName ist $meinAlter Jahre alt
Tobias ist 44 Jahre alt
```

Übrigens wird ein großer Teil der „Umgebung“ der Shell in Umgebungsvariablen gespeichert. Die Variable `PS1` spezifiziert beispielsweise, wie der Prompt ausschauen soll und kann durch den Benutzer genau wie alle anderen Shellvariablen verändert werden. Das Kommando

```
> PS1="Guten Tag: "
```

verändert das Prompt entsprechend – probieren Sie es aus!

Die Shellvariable `PATH` enthält die Liste der Verzeichnisse, in welchen die Shell nach Kommandos sucht. Der Doppelpunkt „`:`“ dient hier als Trenner zwischen den einzelnen Pfaden.

```
> echo $PATH
/usr/local/bin:/usr/bin:/bin
```

Möchte man die Shell beispielsweise zusätzlich im Verzeichnis `/usr/sbin` nach ausführbaren Dateien suchen lassen, so kann man einfach folgenden Befehl verwenden:

```
> PATH=$PATH:/usr/sbin
```

Bevor die Shell diese Zuweisung tatsächlich ausführt, ersetzt sie den Ausdruck `$PATH` durch den Inhalt der `PATH`-Variablen. Der neue Wert der `PATH`-Variablen ist also fortan der bisherige Wert der Pfad-Variablen, `$PATH`, an dessen Ende der String `:/usr/sbin` angehängt wird.

**Aufgabe 2.22**

Sie wollen, dass zusätzlich zu den in der `PATH`-Variablen angegebenen Pfaden immer auch im aktuellen Arbeitsverzeichnis nach ausführbaren Dateien gesucht wird. Geben Sie ein entsprechendes Unix-Kommando an, das dies ermöglicht.

**8** Zu Deutsch: „Eingabeaufforderung“.

### 2.4.5 Ausblendung von Sonderbedeutungen

Die Shell belegt eine Reihe von Zeichen mit einer bestimmten Bedeutung. Diese Zeichen werden auch *Metazeichen* oder Jokerzeichen genannt. Will man beispielsweise mit dem `echo`-Kommando einen Stern ausgeben, so kann man nicht einfach `echo *` schreiben, denn wir haben ja gelernt: Noch bevor ein Kommando ausgeführt wird, geht die Shell über die Befehlszeile und versucht Metazeichen, also insbesondere das Zeichen „$*$“, zu interpretieren.

**Aufgabe 2.23**

Versuchen Sie folgende Fragen zu beantworten, zunächst ohne die jeweiligen Befehle in der Shell auszuführen. Was liefern die folgenden Kommandos als Ergebnis?

(a) `echo *`
(b) `echo ?`
(c) `echo /`

Man kann bestimmte Anführungszeichen dazu verwenden, Sonderbedeutungen auszuschalten. In diesem Zusammenhang werden die Anführungszeichen auch im Deutschen häufig mit dem englischen Begriff „Quotes“ bezeichnet. Die Bash kennt folgende Arten von Quotes:

| | | |
|---|---|---|
| `"..."` | double quote | Abschalten von Dateinamenersetzungen; andere Ersetzungen, wie etwa die einer Variablen durch ihren Inhalt, nimmt die Bash jedoch vor. |
| `'...'` | tick mark | Abschalten aller Ersetzungen. |
| `\` | backslash | Ausschalten der Sonderbedeutung des folgenden Zeichens. |
| `` `...` `` | back tick | Das in den back ticks eingeschlossene Kommando wird durch seine Ausgabe ersetzt. |

Üblicherweise verwendet man die doppelten Anführungszeichen um ein String-Argument anzugeben, das Leerzeichen enthält, wie in folgendem Beispiel:

```
> ls -l "datei mit leerzeichen.txt"
```

Ohne die doppelten Anführungszeichen würde das `ls`-Kommando das Leerzeichen als Stringtrenner interpretieren und versuchen, die Datei mit Namen `datei`, mit Namen `mit` und mit Namen `leerzeichen.txt` aufzulisten.

**Aufgabe 2.24**

(a) Geben Sie mit Hilfe des `echo`-Kommandos den Text `Hallo Welt` aus.

(b) Geben Sie mit Hilfe des `echo`-Kommandos den Text `"Hallo Welt"` aus.
(c) Geben Sie mit Hilfe des `echo`-Kommandos den Text `",Hallo Welt,"` aus.
(d) Geben Sie mit Hilfe des `echo`-Kommandos den Text `\\` aus.

**Aufgabe 2.25**

(a) Was ist der Wert der Variablen `a` nach Ausführung des Befehls
`a='ls -l | wc -l'`?
(b) Was passiert, wenn Sie nach Ausführung des Befehls `a='ls -l | wc -l'` folgendes in die Kommandozeile eingeben und ausführen (das Zeichen > soll das Prompt sein):
`> $a`
(c) Was ist der Wert der Variablen `a` nach Ausführung des Befehls
`a=`ls -l | wc -l``?

### 2.4.6 Verknüpfungen von Kommandos

Durch Verknüpfung von Kommandos mit dem Strichpunkt `;` werden diese hintereinander ausgeführt:

⟨*kommando1*⟩ ; ⟨*kommando2*⟩ ; ⟨*kommando3*⟩ ...

Bevor wir zu den nächsten beiden Verknüpfungsarten kommen, muss man wissen, dass *jedes* Kommando der Bash einen bestimmten *Exit*-Status zurückliefert. Hierbei gilt:

| | |
|---|---|
| Exit-Status = `0` heißt: | Kommando erfolgreich beendet. |
| Exit-Status ≠ `0` heißt: | Kommando nicht erfolgreich beendet. |

Die Shellvariable `$?` enthält immer den Exit-Status des zuletzt ausgeführten Kommandos.

**Aufgabe 2.26**

Geben Sie ein Kommando in die Shell ein, dessen Ausführung Ihrer Einschätzung nach nicht erfolgreich ist. Lassen Sie sich danach den Inhalt der Shellvariablen `$?` ausgeben.

**Aufgabe 2.27**

In der Programmiersprache C ist es genau umgekehrt: Ein Wert von 0 entspricht immer dem Wahrheitswert „falsch“, Werte $\neq$ 0 entsprechen dem Wahrheitswert „wahr“. Was, denken Sie, ist der Grund, warum die Bash die Wahrheitswerte genau umgekehrt repräsentiert?

Die Shell kann nun den Exit-Status nutzen, um Programme ergebnisabhängig zu verbinden. Hierzu gibt es zwei Operatoren, das „sequentielle Und“ und das „sequentielle XOR“. Der sequentielle Und-Operator wird folgendermaßen verwendet:

⟨*programm1*⟩ `&&` ⟨*programm2*⟩

Hierbei wird das zweite Kommando nur dann ausgeführt, wenn das erste erfolgreich war, d. h., wenn das erste einen Exit-Status von `0` zurückliefert.

Der sequentielle XOR wird folgendermaßen verwendet:

⟨*programm1*⟩ `||` ⟨*programm2*⟩

Hierbei wird das zweite Kommando nur ausgeführt, wenn das erste erfolglos war, d. h. wenn das erste einen Exit-Status ungleich `0` zurückliefert.

**Aufgabe 2.28**

Welche der drei zuvor vorgestellten Verknüpfungen entspricht am ehesten dem if-then-Befehl einer höheren Programmiersprache?

**Aufgabe 2.29**

Geben Sie eine Unix-Kommandozeile an, die …

(a) die Namen aller C-Dateien (Endung: `.c`) im aktuellen Verzeichnis ausgibt, falls welche existieren.

(b) den String `Es gibt keine C-Dateien` ausgibt, falls keine C-Dateien im aktuellen Verzeichnis existieren.

**Aufgabe 2.30**

Geben Sie eine Unix-Kommandozeile an, die …

(a) (nur) den Text `Es gibt keine C-Dateien` auf dem Bildschirm ausgibt, falls im aktuellen Verzeichnis keine C-Dateien (mit Endung `.c`) existieren.

(b) (nur) den Text `Es gibt C-Dateien` ausgibt, falls im aktuellen Verzeichnis sich mindestens eine C-Datei befindet.

## 2.5 Weitere Kommandos

### 2.5.1 Das `find`-Kommando

Das `find`-Kommando kann zur Suche von Dateien verwendet werden und ist sehr mächtig. Es findet Dateien nicht nur im aktuellen Verzeichnis, sondern in einem ganzen Verzeichnisbaum, d. h. es sucht systematisch in allen Unterverzeichnissen nach Dateien mit bestimmten Eigenschaften. Die Syntax lautet

`find` ⟨*Pfadname(n)*⟩ ⟨*Bedingung(en)*⟩ ⟨*Aktion(en)*⟩

Hierbei werden alle angegebenen Pfade vollständig (also inklusive aller darunterliegenden Verzeichnisse) durchsucht.

**Bedingungen**

Das `find`-Kommando findet Dateien, die bestimmten Bedingungen genügen. Folgende Liste zeigt einen Auszug der möglichen Bedingungen, die das `find`-Kommando akzeptiert:

| | |
|---|---|
| `-name` ⟨*datei*⟩ | Sucht nach Dateien, deren Namen auf ein bestimmtes Muster passen. Die Verwendung von Metazeichen wie `*` und `?` ist möglich, allerdings muss dann das Muster in tick marks `'...'` eingeschlossen werden. |
| `-iname` ⟨*datei*⟩ | Funktioniert ähnlich wie `-name`, nur dass der „Match“ nicht zwischen Groß- und Kleinschreibung unterscheidet. |
| `-type` ⟨*T*⟩ | Sucht nach einem Dateityp ⟨*T*⟩; das Argument ⟨*T*⟩ kann hierbei die folgenden Werte annehmen: `f` – normale Datei; `d` – directory; `b` – block device; `c` – character device. |
| `-user` ⟨*name*⟩ | Sucht nach Dateien, die dem Benutzer ⟨*name*⟩ gehören. |
| `-group` ⟨*name*⟩ | Sucht nach Dateien, deren Gruppenzugehörigkeit ⟨*name*⟩ ist. |
| `-size` ⟨*N*⟩ | Sucht nach Dateien mit einer Größe von ⟨*N*⟩ Blöcken. |
| `-newer` ⟨*datei*⟩ | Sucht nach Dateien, die jünger sind als ⟨*datei*⟩. |
| `-mtime` ⟨*N*⟩ | Sucht nach Dateien, die vor ⟨*N*⟩ Tagen geändert wurden. |
| `-mmin` ⟨*N*⟩ | Sucht nach Dateien, die vor ⟨*N*⟩ Minuten geändert wurden. |

Bei Zahlenangaben ⟨*N*⟩ kann optional ein + oder – vorangestellt werden; hierbei bedeutet beispielsweise `5` „genau fünf“, `-2` „zwei oder weniger“ und `+3` „drei oder mehr“.

**Aufgabe 2.31**

Warum muss man bei Verwendung von Metazeichen im Argument der Bedingung `-name` dieses in tick marks einschließen? Was passiert, wenn man die tick marks vergessen würde?

**Aktionen**

Für jede der gefundenen Dateien kann man nun bestimmte Aktionen spezifizieren. Folgende Liste zeigt einen Auszug aller Aktionen, die `find` akzeptiert:

| | |
|---|---|
| `-print` | Gibt die gefundenen Dateien über die Standardausgabe aus. |
| `-exec` ⟨*kommando*⟩ | Führt auf jeder gefundenen Datei ⟨*kommando*⟩ aus. Mit der leeren geschweiften Klammer `{}` kann man die Stellen spezifizieren, an denen die gefundenen Dateien eingesetzt werden. Man muss ⟨*kommando*⟩ immer mit einem geschützten Strichpunkt (`\;` oder `';'`) abschließen. |
| `ok` ⟨*kommando*⟩ | Funktioniert genau wie `-exec`, jedoch wird ⟨*kommando*⟩ immer mit vorheriger Sicherheitsabfrage ausgeführt. |

Beispielsweise versucht das folgende Kommando alle Dateien auf Ihrer Festplatte zu löschen, die mit `A` anfangen und mit `.txt` enden:

```
find . -name 'A*.txt' -exec rm {} \;
```

**Aufgabe 2.32**

Warum muss man den Strichpunkt schützen, mit dem das der `-exec`-Aktion folgende Kommando abgeschlossen wird? Wovor muss er geschützt werden? Was würde passieren, wenn man statt `';'` einfach nur `;` schreibt?

**Aufgabe 2.33**

(a) Mit der Aktion `-printf` hat man viele Möglichkeiten, die gefundenen Dateien nach individuellen Wünschen auszugeben. Informieren Sie sich über das Benutzerhandbuch von `find`, was `-printf` für Argumente erwartet.

(b) Verwenden Sie nun Sie die Aktion `-printf`, um alle Dateien zu finden, die mit `.txt` enden, und sich deren Namen zusammen mit dem Änderungsdatum (in schön lesbarer Form) ausgeben zu lassen.

**Aufgabe 2.34**

Geben Sie ein Unix-Kommando an, …

(a) mit dem Sie alle auf Ihrer Festplatte befindlichen Verzeichnisnamen ausgeben, die mit einem `A` beginnen.
(b) mit dem Sie alle auf Ihrer Festplatte befindlichen Dateien löschen, deren Namen mit `.docx` enden.
(c) mit dem Sie ausgeben können, wie viele Dateien unter Ihrem Homeverzeichnis sich in den letzten zwei Tagen geändert haben.
(d) das von allen C-Dateien in und unterhalb Ihres Homeverzeichnisses eine Backup Kopie erstellt (mit Endung `.bak`).
(e) das bestimmt, wie viele Verzeichnisnamen auf Ihrer Festplatte genau aus vier Buchstaben bestehen.

### 2.5.2 Das `grep`-Kommando

Mit `grep` (= global regular expression print) kann nach Textmustern gesucht werden. Die Sprache, in der diese Muster spezifiziert werden, nennt man Reguläre Ausdrücke (englisch: „regular expressions“). Die theoretischen Grundlagen für reguläre Ausdrücke stammen aus der Linguistik und wurden durch Noam Chomsky formalisiert. Die Anwendungsmöglichkeiten sind vielfältig, denn mit `grep` können nicht nur Dateien, sondern – falls keine Dateiargumente mit übergeben werden – auch die Standardeingabe durchsucht werden. Die Syntax lautet:

`grep` ⟨*option(en)*⟩ ⟨*regExp*⟩ ⟨*datei(en)*⟩

Um ein erstes einfaches Beispiel zu geben: `grep und test.txt` sucht in der Datei `test.txt` alle Zeilen, die das Muster `und` enthalten.

Folgende Liste zeigt die wichtigsten Optionen:

- `-q` (quiet) Keine Ausgabe; nur der Return-Wert wird zurückgegeben.
- `-n` Zeilennummern werden mit ausgegeben.
- `-c` Es wird nur die Anzahl der Zeilen ausgegeben, für die der reguläre Ausdruck zutrifft.
- `-l` Nur die Namen der Dateien werden ausgegeben, in denen der reguläre Ausdruck gefunden wurde.
- `-i` Auf Groß- und Kleinschreibung wird nicht geachtet.
- `-v` Alle Zeilen werden ausgegeben, in denen der reguläre Ausdruck *nicht* erfüllt ist.

Die folgende Liste zeigt beispielhaft wie reguläre Ausdrücke aufgebaut sind, die das `egrep`-Kommando verarbeiten kann. Der Name `egrep` ist ein Synonym für `grep -e` und erlaubt dem Benutzer die regulären Ausdrücke in einer etwas komfortableren Form anzugeben:

| Suchmuster | Passt auf ... |
|---|---|
| `Tiger` | – den String `Tiger`. |
| `^Tiger` | – den String `Tiger` am Zeilenanfang. |
| `Tiger$` | – den String `Tiger` am Zeilenende. |
| `^Tiger$` | – eine Zeile, die nur den String `Tiger` enthält. |
| `[tT]iger` | – den String `Tiger` oder den String `tiger`. |
| `T[aeiou]ger` | – die Strings `Tager`, `Teger`, `Tiger`, `Toger`, `Tuger`. |
| `[0-9]Tiger` | – die Strings `0Tiger`, `1Tiger`, ..., `9Tiger`. |
| `T[^aeiou]ger` | – das 1., 3., 4., 5. Zeichen wie angegeben; das 2. Zeichen kann jedes Zeichen, außer `a`, `e`, `i`, `o`, `u` sein. |
| `.` | – jedes beliebige Zeichen. |
| `^...$` | – jede Zeile, die genau drei Zeichen enthält. |
| `^\.` | – jede Zeile, die mit einem Punkt beginnt; der Backslash `\` unterdrückt die Sonderbedeutung des Zeichens „`.`“. |
| `^[^.]` | – jede Zeile, die nicht mit einem Punkt beginnt. |
| `Tiger.*` | – den String `Tiger` gefolgt von einer beliebig langen (auch leeren) Folge beliebiger Zeichen.. |
| `[A-Z][A-Z]*` | – ein oder mehrere Großbuchstaben. |
| `[A-Z]*` | – eine beliebig lange (evtl. auch leere) Folge von Großbuchstaben. |
| `(Tiger)*` | – eine beliebig lange (auch leere) Folge des Strings `Tiger`. |

Kapitel 4 liefert einen detaillierten Überblick über reguläre Ausdrücke.

Einige einfache Beispiele für die Verwendung von `grep` bzw. `egrep` zeigt die folgende Tabelle:

| Beispiel | Funktionsweise |
|---|---|
| `egrep ^Maier` ⟨*datei*⟩ | Liefert alle Zeilen aus ⟨*datei*⟩, die mit `Maier` beginnen. |
| `egrep -c ^$` ⟨*datei*⟩ | Liefert die Anzahl der Leerzeilen in ⟨*datei*⟩. |
| `egrep [^0-9]$` ⟨*datei*⟩ | Liefert alle Zeilen, die nicht mit einer Ziffer enden. |
| `egrep ^a`⟨*datei*⟩ `\| egrep b$` | Liefert alle Zeilen, die mit `a` beginnen und mit `b` enden. |

Das zweite `egrep` im letzten Beispiel bekommt keine Datei als Argument übergeben, d. h. es liest aus der Standardeingabe. In diesem Fall wird die Standardeingabe aber nicht von der Tastatur gelesen, sondern aus der Pipe, in die das erste `egrep`-Kommando seine Ergebnisse geschrieben hat.

Der Exit-Status des Kommandos `grep` bzw. `egrep` ist `0`, falls in dem angegebenen Muster entsprechende Zeilen gefunden wurden und `1`, falls es keine Treffer gab.

**Aufgabe 2.35**

Geben Sie ein Unix-Kommando an, das den String `Tobias ist da!` ausgibt, falls der Benutzer `tobias` am System angemeldet ist, und den String `Tobias ist nicht da.`, falls der Benutzer `tobias` nicht angemeldet ist. Achten Sie darauf, dass *ausschließlich* die angegebenen Strings ausgegeben werden, insbesondere keine Fehlermeldungen.

**Aufgabe 2.36**

Geben Sie einen regulären Ausdruck an, mit dessen Hilfe das `egrep`-Kommando alle Zeilen ausgibt, die …

(a) die Zeichenkette `Hallo` enthalten.
(b) mit irgend einer Ziffer beginnen.

**Aufgabe 2.37**

Verwenden Sie das `egrep`-Kommando, um …

(a) alle Dateien im aktuellen Verzeichnis aufzulisten, die keine Verzeichnisse sind.
(b) alle Dateien im aktuellen Verzeichnis aufzulisten, auf denen alle Gruppen Schreibrechte haben.
(c) alle angemeldeten Benutzer aufzulisten, deren Benutzername mit `a` endet.

**Aufgabe 2.38**

Geben Sie einen regulären Ausdruck an, mit dessen Hilfe das `egrep`-Kommando alle Zeilen ausgibt, die …

(a) mit `a` oder mit `b` anfangen.
(b) nur Ziffern enthalten.
(c) genau zwei Worte enthalten.

**Aufgabe 2.39**

Geben Sie ein `egrep`-Kommando an, mit dem sie alle Zeilen finden, …

(a) in denen irgendwo `blubber` vorkommt.
(b) die das Wort `blubber` enthalten.
(c) die mit einer Ziffer anfangen und mit einem Buchstaben aufhören.
(d) die genau zweimal den Buchstaben `a` enthalten.
(e) die genau zwei Ziffern enthalten *oder* genau einmal `a` enthalten.

**Aufgabe 2.40**

Verwenden Sie das `egrep`-Kommando, um …

(a) alle Dateien im aktuellen Verzeichnis aufzulisten, die Verzeichnisse sind.
(b) alle Dateien im aktuellen Verzeichnis aufzulisten, auf denen Sie Schreibrechte und Leserechte haben.
(c) alle Dateien im aktuellen Verzeichnis aufzulisten, auf denen Sie entweder Schreibrechte oder Leserechte haben (aber nicht beides).
(d) alle Dateien im aktuellen Verzeichnis aufzulisten, auf denen ausschließlich Sie Schreib- und Leserechte haben.
(e) die Gesamtzahl aller Leerzeilen in allen C-Dateien (Endung `.c`) zurückzugeben, die sich im aktuellen Verzeichnis befinden.

**Aufgabe 2.41**

Geben Sie einen regulären Ausdruck an, mit dessen Hilfe das `egrep`-Kommando alle Zeilen ausgibt, die …

(a) den Text `Hallo Welt` enthalten.
(b) mit dem Text `Hallo Welt` beginnen.
(c) nur den Text `Hallo Welt` enthalten.
(d) mit keiner Ziffer beginnen.
(e) genau eine Ziffer enthalten.

### 2.5.3 Der `cut`-Befehl

Das Kommando

`cut` ⟨*option(en)*⟩ ⟨*datei(en)*⟩

gibt Spalten von Dateizeilen auf der Standardausgabe aus. Wichtige Optionen sind:

| | |
|---|---|
| `-f` ⟨*felder*⟩ | Gibt die angegebenen Felder aus. |
| `-d` ⟨*zeichen*⟩ | Bestimmt das Trennzeichen zwischen den einzelnen Feldern. Default ist das Tabulator-Zeichen. |

**Beispiel**
Die Datei `/etc/passwd` enthält Informationen über die Benutzer; pro Nutzer gibt es eine Zeile, die sieben Felder enthält, die mit `:` getrennt sind. Das Kommando

```
cut -d ':' -f1,6 /etc/passwd
```

druckt das erste und sechste Feld aus, in diesem Fall den Login-Namen und das HOME-Verzeichnis.

**Aufgabe 2.42**

Verwenden Sie das `cut`-Kommando, um sich für jede Datei des aktuellen Arbeitsverzeichnisses den Dateinamen und das Änderungsdatum ausgeben zu lassen.

**Aufgabe 2.43**

Geben Sie ein Unix-Kommando an, das Ihnen den dritten Pfad ausgibt, in dem nach ausführbaren Dateien gesucht wird.

### 2.5.4 Das `sort`-Kommando

Das Kommando

`sort` ⟨*option(en)*⟩ ⟨*datei(en)*⟩

sortiert die Zeilen einer Datei nach bestimmten (Sortier-)Feldern und gibt das Ergebnis auf der Standardausgabe aus. Wichtige Optionen sind:

| | |
|---|---|
| `+` ⟨*pos1*⟩ [*pos2*] | Sortiert nach den Feldern ⟨*pos1*⟩ bis ausschließlich ⟨*pos2*⟩; |
| `-b` | ignoriert Leerzeichen am Anfang von Sortierfeldern; |
| `-n` | sortiert numerisch; |
| `-o` ⟨*datei*⟩ | gibt das Ergebnis in ⟨*datei*⟩ aus; |
| `-r` | dreht die Sortierreihenfolge um; |
| `-t` ⟨*zeichen*⟩ | gibt das Trennzeichen zwischen den Feldern an. |

Beispielsweise gibt das Kommando

```
sort +2 -n -r -t ':' /etc/passwd
```

den Inhalt der Datei `/etc/passwd` aus, numerisch absteigend sortiert nach dem dritten Feld.

**Aufgabe 2.44**

Verwenden Sie das `ls`- und das `sort`-Kommando, um die Dateien des aktuellen Verzeichnisses nach dem Benutzernamen des Besitzers zu sortieren.

### 2.5.5 Die `head`- und `tail`-Kommandos

Das Kommando

`head -n` ⟨*zahl*⟩ ⟨*datei(en)*⟩

gibt die ersten Zeilen der Eingabedateien aus. Dies findet häufig bei sehr großen Dateien Anwendung, wenn man weiß, dass die wichtigen Informationen am Dateianfang stehen.

**Aufgabe 2.45**

Verwenden Sie die Befehle `ls -l`, `sort` und `head` um sich die fünf größten Dateien im aktuellen Verzeichnis ausgeben zu lassen.

Das Kommando

`tail -n` ⟨*zahl*⟩ ⟨*datei*⟩

gibt die letzten Zeilen von `datei` aus. Neben der Option `-n` wird auch die Option `-f` häufig genutzt. Wenn die Datei wächst oder sich verändert, wird die Ausgabe automatisch mit ergänzt. Beispielsweise zeigt

```
tail -f /var/log/messages
```

neue Systemmeldungen sofort an. Das `tail`-Kommando wartet hierbei einfach und gibt die Änderungen aus.

## 2.6 Shell-Programmierung

Bisher haben wir ausschließlich Unix-Kommandos interaktiv über die Tastatur eingegeben und von der Shell direkt auf der Kommandozeile ausführen lassen. Die Shell erlaubt aber auch die „stapelweise“ Ausführung von in einer Textdatei befindlichen Befehlen. Es ist möglich, mehrere Befehlszeilen in eine Textdatei zu schreiben und diese durch die Bash nacheinander ausführen zu lassen. Der Effekt ist so, als ob die Befehle nacheinander eingegeben wurden.

Das folgende Beispiel zeigt, wie man ein in einer Datei befindliches Kommando ausführen kann; das Zeichen „$“ am Zeilenanfang soll das Prompt der Bash darstellen:

```
$ echo 'who | wc -l' >nu
$ bash nu
5
```

Wir schreiben also den Befehl `echo 'who | wc -l'` in die Datei `nu` (= number of users) und können diesen dann einfach mittels `bash nu` ausführen lassen. Wir können das Skript, also eine Textdatei, die Shell-Befehle enthält, in der Datei `nu` auch ausführbar machen und direkt aufrufen:

```
$ chmod +x nu
$ nu
5
```

Übrigens: Enthält ein Skript in der ersten Zeile die folgende Deklaration

`#!`⟨*programm*⟩

dann startet jede Shell automatisch ⟨`programm`⟩ und übergibt diesem als Parameter den Namen des Skripts. Künftig schreiben wir also immer `#!/bin/bash` in die erste Zeile jedes Bash-Skripts.

**Aufgabe 2.46**

Erstellen Sie ein kleines Bash-Skript `anzDat`, das Ihnen die Anzahl der Einträge und die Anzahl der Verzeichnisse im aktuellen Arbeitsverzeichnis folgendermaßen ausgibt (hierbei soll das `$`-Zeichen am Anfang jeder Zeile die Eingabeaufforderung darstellen):

```
$ anzDat
Das aktuelle Verzeichnis enthält 26 Dateien, davon 12 Verzeichnisse
```

### 2.6.1 Dateneingabe

Das Kommando

`read` [`-p` ⟨*prompt*⟩] ⟨*variable1*⟩ ⟨*variable2*⟩ ...

liest Wörter von der Standardeingabe und speichert diese in Umgebungsvariablen. Mit der Option `-p` kann man zusätzlich eine Eingabeaufforderung ausgeben lassen.

**Beispiel**
Nach Eingabe des Kommandos

```
read -p "Bitte Ihren Vor- und Nachnamen eingeben:" VNAME NNAME
```

erscheint der nach `-p` kommende Text auf dem Bildschirm und `read` wartet auf Benutzereingabe; das erste eingegebene Wort wird der Variablen `VNAME`, das zweite der Variablen `NNAME` zugewiesen.

Das Kommand

```
read A <test.txt
```

liest die erste Zeile von `test.txt` in die Variable `A`.

### 2.6.2 Kommandozeilenparameter

Kommandozeilenparameter sind die Parameter, die ein Kommando über die Kommandozeile als String-Argumente übergeben bekommt. Angenommen wir schreiben ein Skript `add`, das zwei Zahlen addiert; wir werden also erwarten, dass der Benutzer beim Aufrufen dem Skript `add` zwei Parameter mit übergibt:

```
add 17 23
```

In diesem Fall wäre der erste Kommandozeilenparameter der String `17`, der zweite Kommandozeilenparameter wäre String `23`. Innerhalb des Skripts können diese beiden Parameter über die speziellen Variablen `$1` und `$2` angesprochen werden. Das Skript `add` könnte also folgendermaßen programmiert werden:

```
#!/bin/bash
ergebnis=$[$1+$2]
echo $1 plus $2 ist gleich $ergebnis
```

Mit Hilfe des Konstruktes `$[...]` kann die Bash einfache Rechnungen ausführen.

Die spezielle Shell-Variable `$#` steht für die Anzahl der übergebenen Kommandozeilenparameter.

**Aufgabe 2.47**

Finden Sie durch Ausprobieren heraus, welchen Wert die Shell-Variable `$0` enthält.

**Aufgabe 2.48**

Schreiben Sie ein Bash-Skript, das den 10. Kommandozeilenparameter wieder ausgibt:

```
$ derZehnte A B C D E F G H I J K L
$ Der 10. Parameter war J
```

### 2.6.3 Bedingungen testen

**Das `test`-Kommando**
Der Befehl

test ⟨*ausdruck*⟩

prüft eine Bedingung und liefert daraufhin einen entsprechenden Exit-Status. Mit dem Argument ⟨*ausdruck*⟩ lassen sich Dateien, Zeichenketten, ganze Zahlen und vieles mehr vergleichen. Als erstes Beispiel vorab betrachten wir die folgende Kommandozeile:

```
test -w /etc/passwd && echo "Du bist root"
```

Diese testet, ob die Datei `/etc/passwd` lesbar ist. Falls sie lesbar ist, d. h., falls der Exit-Status des `test`-Befehls gleich `0` ist, wird der Text `Du bist root`ausgegeben. Die Bezeichnung „root“ bezieht sich ursprünglich auf die Wurzel des Unix-Verzeichnisbaums und damit wird auch gleichzeitig der Administrator bezeichnet, dem üblicherweise kein eigenes Home-Verzeichnis zugeordnet ist. Stattdessen ist er in der Wurzel des Verzeichnisbaums zuhause und hat entsprechend auch grundsätzlich alle Schreib- und Leserechte auf alle Objekte des gesamten Verzeichnisbaums. Obige Kommandozeile macht daher auch Sinn: Nur der Administrator hat Schreibrechte auf die Datei `/etc/passwd`.

Folgende Tabelle zeigt die wichtigsten Komponenten, aus denen sich der Parameter ⟨*ausdruck*⟩ des `test`-Befehls zusammensetzen kann.

**Eigenschaften von Dateien**

| | |
|---|---|
| `-e` | Datei existiert; |
| `-r` | Datei existiert und User hat Leserecht; |
| `-w` | Datei existiert und User hat Schreibrecht; |
| `-x` | Datei existiert und User hat Ausführungsrecht; |
| `-f` | Datei existiert und ist „einfache“ Datei; |
| `-d` | Datei existiert und ist Verzeichnis; |
| `-s` | Datei existiert und ist nicht leer; |
| `-L` | Datei existiert und ist symbolischer Link; |

⟨*datei1*⟩ `-nt` ⟨*datei2*⟩ wahr, wenn ⟨*datei1*⟩ neuer als ⟨*datei2*⟩;
⟨*datei1*⟩ `-ot` ⟨*datei2*⟩ wahr, wenn ⟨*datei1*⟩ älter als ⟨*datei2*⟩;
⟨*datei1*⟩ `-ef` ⟨*datei2*⟩ wahr, wenn ⟨*datei1*⟩ und ⟨*datei2*⟩ die gleiche Inode-Kennung besitzen; diese spezifiziert die Identität einer Datei unter Unix. Das Kürzel `-ef` steht hier für „equal file“.

**Vergleich von Zeichenketten**

`-n` ⟨*str*⟩ Wahr, wenn der String ⟨*str*⟩ nicht leer;
`-z` ⟨*str*⟩ wahr, wenn der String ⟨*str*⟩ leer;
⟨*str1*⟩ = ⟨*str2*⟩ wahr, wenn die beiden Strings gleich sind;
⟨*str1*⟩ `!=` ⟨*str2*⟩ wahr, wenn die beiden Strings verschieden sind.

**Vergleich ganzer Zahlen**

⟨*zahl1*⟩ `-eq` ⟨*zahl2*⟩ Wahr, wenn ⟨*zahl1*⟩ = ⟨*zahl2*⟩;
⟨*zahl1*⟩ `-ne` ⟨*zahl2*⟩ wahr, wenn ⟨*zahl1*⟩ ≠ ⟨*zahl2*⟩;
⟨*zahl1*⟩ `-ge` ⟨*zahl2*⟩ wahr, wenn ⟨*zahl1*⟩ ≥ ⟨*zahl2*⟩;
⟨*zahl1*⟩ `-le` ⟨*zahl2*⟩ wahr, wenn ⟨*zahl1*⟩ ≤ ⟨*zahl2*⟩;
⟨*zahl1*⟩ `-gt` ⟨*zahl2*⟩ wahr, wenn ⟨*zahl1*⟩ > ⟨*zahl2*⟩;
⟨*zahl1*⟩ `-lt` ⟨*zahl2*⟩ wahr, wenn ⟨*zahl1*⟩ < ⟨*zahl2*⟩.

**Logische Verknüpfungen**

⟨*ausdr1*⟩ `-a` ⟨*ausdr2*⟩ Wahr, wenn ⟨*ausdr1*⟩ und ⟨*ausdr2*⟩ gilt;
⟨*ausdr1*⟩ `-o` ⟨*ausdr2*⟩ wahr, wenn ⟨*ausdr1*⟩ oder ⟨*ausdr2*⟩ gilt;
`!` ⟨*ausdr*⟩ wahr, wenn ⟨*ausdr*⟩ nicht gilt;
`\(` ⟨*ausdr*⟩ `\)` Gruppierung des Ausdrucks ⟨*ausdr*⟩.

**Aufgabe 2.49**

Schreiben Sie ein Kommando, das Ihnen den ersten in der `PATH`-Variablen enthaltenen Pfad ausgibt; aber nur dann, wenn die `PATH`-Variable auch tatsächlich definiert ist.

**Aufgabe 2.50**

Schreiben Sie ein Kommando, das die Inhalte der Dateien `test1.txt` und `test2.txt` aneinanderhängt und in der Datei `test3.txt` speichert; aber nur, wenn die Berechtigungen für diese Aktion entsprechend passen.

**Aufgabe 2.51**

Schreiben Sie – aus Ihrem jetzigen Kenntnisstand heraus, also ohne Verwendung von Schleifen – ein Shell-Skript, das genau drei Kommandozeilenparameter einliest (falls mehr oder weniger eingegeben wurde: Fehlermeldung ausgeben) und den größten dieser Kommandozeilenparameter zurückliefert. Falls die Kommandozeilenparameter keine Zahlen sind, soll eine passende Fehlermeldung ausgegeben werden.

Es gibt einen symbolischen Link auf das Kommando `test`, der einfach `[` heißt; Falls `test` aber mittels `[` aufgerufen wurde, wird verlangt, dass das Kommando mit `]` endet. Statt beispielsweise `test -r dat1.txt` könnte man also einfach

```
[ -r dat1.txt ]
```

schreiben.

**Aufgabe 2.52**

Wie könnte `test` programmiert sein, dass es zum einen weiß, ob es als „`test`“ oder als „`[`“ aufgerufen wurde und zum anderen sicherstellt, dass es – falls es mit `[` aufgerufen wurde – auch mit `]` endet?

**Aufgabe 2.53**

Falls `test` als „`[`“ aufgerufen wurde, so ist es wichtig, dass nach „`[`“ und vor „`]`“ ein Leerzeichen steht. Erklären Sie warum das wichtig ist, und was passiert, falls sie diese Leerzeichen vergessen.

### Das `if`-Kommando

Das `if`-Kommando kann mehrere Formen annehmen:

Einseitiges `if`

```
if ⟨kommando⟩
then
    ⟨kommandos⟩
fi
```

Zweiseitiges `if`

```
if ⟨kommando⟩
then
    ⟨kommandos⟩
else
    ⟨kommandos⟩
fi
```

Nach dem Schlüsselwort `if` wird ein Argument ⟨*kommando*⟩ erwartet, ein Unix-Kommando bzw. eine Folge von Kommandos, die mit Kommandoverknüpfungen zu-

sammengefügt wurden. Dieses wird ebenfalls ausgeführt. Ist der Exit-Status dieses Kommandos …

- = 0, d. h. das Kommando schließt erfolgreich ab, so werden die Kommandos, die zwischen dem `then` und `fi`[9] stehen ausgeführt.
- ≠ 0, d. h. das Kommando schließt erfolglos bzw. mit Fehlern ab, so werden die Kommandos zwischen dem `else` und dem `fi`[10] ausgeführt.

Betrachten wir als einfaches Beispiel das folgende Bash-Skript. Es gibt eine Meldung aus, wenn mehr als fünf Benutzer eingeloggt sind:

```
if [ `who | wc -l` -gt 5 ]
then
   echo "Mehr als 5"
fi
```

**Aufgabe 2.54**

Erklären Sie die Funktionsweise des Skripts, und den Sinn der `if`-Abfrage.

```
if [ -r $1 -a -w $2 ]
then
    cat $1 >> $2
else
    "cannot append"
fi
```

**Aufgabe 2.55**

Schreiben Sie ein Shell-Skript, das den Text "`Bitte aufräumen`" auf dem Bildschirm ausgibt, wenn sich mehr als 50 Dateien im aktuellen Verzeichnis befinden.

### 2.6.4 Programmschleifen

#### Die `while`-Schleife

Schleifen nennt man diejenigen Konstrukte einer Programmiersprache, mit denen man eine Folge von Kommandos *wiederholt* ausführen kann, so lange, bis eine be-

9 Man spricht hier auch von dem `then`-Zweig.
10 Man spricht hier auch von dem `else`-Zweig.

stimmte Bedingung erfüllt ist. Diejenige Kommandofolge, die bei einer Schleife evtl. wiederholt ausgeführt wird, nennt man *Schleifenkörper*.

Die `while`-Schleife hat die folgende Struktur:

```
while ⟨kommando⟩
do
    ⟨kommandos⟩
done
```

Das `while`-Kommando prüft vor jedem Schleifendurchgang, ob ⟨*kommando*⟩ wahr ist, d. h. den Exit-Status `0` zurückliefert. Falls ja, dann wird die Schleife nochmals durchlaufen. Erst dann, wenn ⟨*kommando*⟩ falsch ist, wird die `while`-Schleife beendet und die Bash geht zur Ausführung des nächsten Kommandos über. Anders formuliert: Die Kommandofolge ⟨*kommandos*⟩ wird so lange wiederholt ausgeführt, bis ⟨*kommando*⟩ einen Exit-Status > 0 liefert.

Ein einfaches Beispiel: Das folgende Bash-Skript wartet solange, bis die Datei `foo` (möglicherweise von irgendeinem Hintergrundprozess) erzeugt ist:

```
while [ ! -f foo ]
  do
  sleep 10
done
```

(Das `sleep`-Kommando wartet 10 Sekunden).

**Aufgabe 2.56**

Wozu dient das `sleep`-Kommando im vorigen Beispiel; oder anders gefragt: Warum ist es ratsam, nach jeder Abfrage, ob `foo` existiert, eine gewisse Zeit zu warten?

**Aufgabe 2.57**

Verwenden Sie eine `while`-Schleife, um ein Bash-Skript zu schreiben, das …

(a) alle Zahlen von 1 bis 100 auf dem Bildschirm ausgibt.
(b) alle ungeraden Zahlen von 1 bis 100 auf dem Bildschirm ausgibt.
(c) alle Primzahlen zwischen 1 und 100 auf dem Bildschirm ausgibt.

**Die `for`-Schleife**

Die `for`-Schleife hat die folgende Struktur:

```
for ⟨variable⟩ in ⟨liste⟩
do
    ⟨kommandos⟩
done
```

Die `for`-Schleife der Bash unterscheidet sich von der `for`-Schleife, wie sie etwa in C, C++ oder Java verwendet wird; dagegen ist die `for`-Schleife in Python ähnlich. Die `for`-Schleife der Bash ist eine Listenschleife, d. h. die Schleife wird so oft durchlaufen, wie sich Elemente in ⟨*liste*⟩ befinden. Das Argument ⟨*liste*⟩ ist hierbei eine mit Leerzeichen, Tabs oder Zeilentrenner getrennte Liste von Strings. Im *i*-ten Durchlauf der Schleifen nimmt dabei ⟨*variable*⟩ als Wert den *i*-ten String der Liste ⟨*liste*⟩ an.

Ein einfaches Beispiel: Die `for`-Schleife

```
for X in hans ben karl
do
  echo $X
done
```

führt das Kommando `echo` drei mal aus, denn die übergebene Liste besteht aus drei Strings. Im ersten Durchlauf hat `X` den Wert `hans`, im zweiten Durchlauf den Wert `ben` und im dritten den Wert `karl`. Das Skript gibt diese drei Strings einfach aus.

Betrachten wir ein weiteres Beispiel für die Verwendung einer `for`-Schleife:

```
for X in *.c
do
  cp $X ${X}.bak
done
```

Bevor die `for`-Schleife ausgeführt wird, wird das Namensmuster „`*.tex`“ durch die Bash expandiert. Die `for`-Schleife bewirkt also, dass die Variable `X` die Namen aller `.c`-Dateien des aktuellen Verzeichnisses durchläuft. Für jede `.c`-Datei wird dann eine Sicherungskopie mit Endung `.bak` erstellt.

Es gibt noch eine wichtige Besonderheit: Es ist auch möglich, der `for`-Schleife keine Liste aus Strings mit zu übergeben. In diesem Fall ist die `for`-Schleife also folgendermaßen strukturiert:

```
for ⟨variable⟩
do
    ⟨kommandos⟩
done
```

Hierbei läuft die Schleife automatisch über die Kommandozeilenparameter `$1`, `$2` usw. Es gibt dann also `$#` Schleifendurchläufe. Im *i*-ten Schleifendurchlauf nimmt hierbei ⟨*variable*⟩ den Wert `$`*i* an.

**Aufgabe 2.58**

Schreiben Sie ein Shell-Skript, das die Summe der Zeilenlängen aller Dateien im aktuellen Verzeichnis ausgibt.

**Aufgabe 2.59**

Schreiben Sie ein Shell-Skript, das alle C-Dateien (also Dateien mit Endung `.c`) in den darüberliegenden Ordner verschiebt.

**Aufgabe 2.60**

Schreiben Sie ein Shell-Skript, das die Anzahl der Dateien im aktuellen Verzeichnis zählt, die (für Sie) ausführbar sind. Das Shell-Skript soll am Ende (beispielsweise) ausgeben:

```
Im aktuellen Verzeichnis sind 6 Dateien ausführbar.
```

**Aufgabe 2.61**

Schreiben Sie ein Shell-Skript, das alle im aktuellen Verzeichnis befindlichen kurzen, weniger als 10 Zeilen langen, Textdateien (mit Endung `.txt`), für die Sie Leserechte haben, aneinanderhängt und in der Datei `alleKurzen.txt` speichert.