Linux-Kurs

Maike Tech

5. Juni 2007

Inhaltsverzeichnis

1 Hardware/Software

1.1 Hardware

Unter Hardware versteht man alle Geräte (*devices*) des Computers. Also alles, was man sehen und anfassen kann.

1.1.1 Ein- und Ausgabegeräte (Tastatur, Maus, Monitor, ...)

Die Ein- und Ausgabegeräte dienen der Kommunikation zwischen Rechner und Benutzer. Dabei ist der Monitor meist als Standardausgabe-Gerät (*standardoutput device*) und die Tastatur als Standardeingabe-Gerät (*standardinput device*) definiert. Auf die Bedeutung dieser Definition wird in Kapitel 2 eingegangen.

1.1.2 Statische Speichermedien (Festplatte, CD-ROM, ...)

Statische Speichermedien wie Festplatten usw. dienen, wie der Name schon sagt, zur dauerhaften Speicherung von Daten. D. h. die Daten bleiben auch nach dem Abschalten des Rechners erhalten, im Gegensatz zum dynamischen Speicher (z. B. Arbeitsspeicher, s. u.). Auf Festplatten (HDD – *harddisk drive*) werden die Daten mit Hilfe eines Schreib-/Lesekopfes magnetisch auf mehreren Scheiben gespeichert. Ähnlich verläuft der Speichervorgang mit dem Diskettenlaufwerk (*floppy disk drive*). Derzeit gängige 3,5"(Zoll)-Disketten haben eine Kapazität von 1,44 MB, während Festplatten auf derzeitigem Stand häufig eine Kapazität von 100 GigaByte und mehr haben.

Daten, die auf einer Festplatte oder einer Diskette gespeichert sind, können beliebig gelöscht oder überschrieben werden. Auf CD-ROMs (*compact disk read only memory*) werden die Daten mit einem Laser permanent eingebrannt und können dann nicht mehr verändert werden. Weitere statische Speichermedien sind ZIP-Disketten (*zip-disk*), CD-RWs (*compact disk rewritable*), MO-Disketten (*magneto optical disk*), DVD (*digital versatile disk*) usw.

Speichereinheiten:
bit (b) – kleinste Speichereinheit (2 Zustände: 0, 1)
Byte (B) = 8 b
KiloByte (kB) = 1024 B
MegaByte (MB) = 1024 kB
GigaByte (GB) = 1024 MB
TeraByte (TB) = 1024 GB

Ein Byte repräsentiert eine 8-stellige Binärzahl. D. h. es können 256 verschiedene Zeichen durch ein Byte dargestellt werden. Graphisch könnte man das folgendermaßen andeuten:

128	64	32	16	8	4	2	1

Die erste »Zelle« repräsentiert den Wert 128, die zweite den Wert 64 usw. Durch das Füllen der Zellen mit 0 oder 1, ergibt sich eine (Dezimal)Zahl, beispielsweise 01100111 = 103.

1.1.3 Rechner (Prozessor, Arbeitsspeicher, ...)

Der Rechner verfügt über eine Reihe von Komponenten, die auf einer zentralen Einheit, dem *Mainboard* (oder *Motherboard*) untergebracht sind. Im Folgenden sollen nur zwei dieser Komponenten vorgestellt werden, die für das Arbeiten am Rechner, insbesondere effektives Programmieren, relevant sind.

Der Prozessor (Hauptprozessor, CPU – *central processing unit*) führt Rechenoperationen wie Addieren oder Multiplizieren auf Daten aus. Die Rechenleistung (Verarbeitungsgeschwindigkeit)

des Prozessors wird u. a. als Taktfrequenz in MHz bzw. GHz angegeben und entspricht der ausgeführten Anzahl an Elementar-Operationen pro Sekunde[1]. Aktuell benötigte Daten werden dabei im Arbeitsspeicher (Hauptspeicher, RAM – *random access memory*) »gehalten«. D. h. nach Beendigung eines Prozesses werden die Daten, die dieser Prozess im Speicher abgelegt hat, gelöscht[2]. Die Kapazität derzeit gängiger Arbeitsspeicher liegt im Bereich von mehreren Megabyte bis zu einigen Gigabyte.

1.2 Software

Software ist der nicht-greifbare Teil, der den Computer durch eine Reihe von Anweisungen steuert. Die Anweisungen sind als ausführbare Daten im System (auf der Festplatte oder einem anderen Speichermedium) gespeichert, also statisch. Beim Aufruf eines Programms durch ein Kommando wird ein Prozess erzeugt, der nur als Zustand von Prozessor und Arbeitsspeicher existiert. Dieser ist im Gegensatz zum Programm nicht-statisch (also dynamisch).

Die Vermittlung zwischen Programmen und Hardware, sowie die Verwaltung und Zuteilung der Ressourcen eines Rechners an verschiedene Prozesse, übernimmt das Betriebssystem.

1.2.1 Betriebssystem (*OS – operating system*)

Direkt über der Hardware befindet sich das Betriebssystem. Der Kern des Betriebssystems (*kernel*), stellt Funktionen zur Verfügung, mit deren Hilfe die Systemkomponenten des Rechners gesteuert werden können (Treiber). Dadurch können Anwendungsprogramme ohne genaue Kenntnis der aktuellen Hardware (beispielsweise was für eine Grafikkarte gerade benutzt wird – Typ, Hersteller...) auf dem Rechner arbeiten. Das Betriebssystem vermittelt quasi zwischen Anwendungsprogrammen und Hardware.

Betriebssysteme, die das gleichzeitige Arbeiten mehrerer Benutzer (*user*) auf einem Rechner ermöglichen, nennt man Mehrbenutzer-System (*multiuser-System*). Ein solches Mehrbenutzer-System ist Linux.

Linux

Ebenso wie Windows ist Linux ein Betriebssystem. Es ist aus dem 1969 entwickelten Unix entstanden. Unix ist das am weitesten verbreitete Netzwerkbetriebssystem. Veränderungen/Neuerungen führten zu verschiedenen Varianten, die alle zur Unix-Familie gehören und auch als Unix-Derivate bezeichnet werden, z. B. Linux oder Solaris.

Der Linux-Kernel selbst wurde ursprünglich von Linus Torvalds entwickelt (erste Veröffentlichung 1991) und ist im Gegensatz zu Windows nicht kommerziell sondern frei verfügbar.[3] Durch die freie Verfüg- und Veränderbarkeit war eine rasante und erfolgreiche Weiterentwicklung möglich, an der viele Programmierer auf der ganzen Welt beteiligt waren und sind.

Es gibt verschiedene Distributionen von Linux (beispielsweise SuSE, RedHat, Debian). Diese basieren alle auf dem gleichen Kernel (möglicherweise in unterschiedlichen Versionen), unterscheiden sich aber durch die mitgelieferte Ausstattung.

Linux ist, wie bereits erwähnt, ein Mehrbenutzer-System, d. h. mehrere Benutzer können gleichzeitig auf einem Rechner arbeiten. Dazu ist eine geeignete Benutzerverwaltung nötig, welche die

[1] Heutige Prozessoren können oft mehrere Befehle pro Takt ausführen, d. h. eine Taktfrequenz von 1 GHz kann theoretisch 3-4 Mrd Prozessorbefehlen pro Sekunde entsprechen.

[2] Unter Linux wird der Arbeitsspeicher erst geleert, wenn er wieder gebraucht wird oder der Rechner heruntergefahren wird.

[3] Das gilt für das Programm und den Quellkode (*open source*).

Daten und Ressourcen jedes Benutzers vor dem Zugriff der anderen Benutzer schützt. Jedem Benutzer wird normalerweise ein eigener Bereich zugewiesen, in dem er schalten und walten kann wie er will, das Heimatverzeichnis (*home*-Verzeichnis).

1.2.2 Anwendungsprogramme

Anwendungsprogramme arbeiten auf dem Betriebssystem (das wiederum selbst ein Programm ist) und sind dadurch weitgehend unabhängig von der aktuellen Hardware. Sie stellen dem Benutzer kompliziertere Funktionen zur Verfügung, die aber letztlich auf der Kombination elementarer Operationen beruhen. Um den korrekten Ablauf zu gewährleisten, kommt es dabei nur darauf an, die Reihenfolge der Anweisungen korrekt zu befolgen.

Ein Programm kann bei seiner Ausführung im Prozessor unterbrochen werden, um andere Programme auszuführen. Dies geschieht viele hundertmal pro Sekunde nach einem genau festgelegten Verfahren, das vom Betriebssystem abhängt, dem Zeitscheibenverfahren (*time slicing* oder *Round-Robin-Verfahren*). Durch die Schnelligkeit neuerer Rechner entsteht der Eindruck des gleichzeitigen Ablaufs mehrerer Programme (*multitasking*). Der Prozessor führt aber immer eine Operation nach der anderen aus. Echtes Multitasking, also paralleles Ausführen mehrerer Operationen, ist nur auf Rechnern mit mehreren Prozessoren möglich.

1.3 Ein bildliches Beispiel

Um die Begriffe Prozess, Programm und Betriebssystem darzustellen, verwendet man oft Analogien aus dem täglichen Leben, in diesem Fall betrachten wir ein Restaurant. Der Benutzer wäre mit einem Gast zu vergleichen, der eine Bestellung macht, sich aber im Weiteren nicht unbedingt damit beschäftigen muss, wie das Kochen abläuft. Die Programme sind vergleichbar mit den Rezepten, statische Anleitungen zum Kochen verschiedener Gerichte. Ein Prozess entspricht dann dem kochenden Essen und das Betriebssystem entspricht dem Koch, der die Zutaten gemäß den Anweisungen im Rezept zubereitet und dabei die Ressourcen (Herdplatte, Kochtopf) an die verschiedenen Gerichte, die gekocht werden, verteilt. Ist das Essen fertig, wird der Prozess beendet und der Gast erhält das gewünschte Ergebnis.

2 Die Shell (Kommandozeileninterpreter)

Über dem Betriebssystem liegt die Shell (Mantel, Muschel). Sie bietet eine Konsole, auf der Kommandos eingegeben werden können, die zeilenweise interpretiert werden, und wird daher auch Kommandozeileninterpreter (*commandline interpreter*) genannt. Es gibt verschiedene Shells, die ein unterschiedliches Repertoire an Kommandos liefern, beispielsweise `ksh` (*korn-shell*). Die hier verwendete Shell ist eine `bash` (*Bourne-again-shell*).

Wird eine Shell aufgerufen, so erhält man zunächst eine Eingabeaufforderung (*prompt*), hinter der Befehle eingegeben werden (in den folgenden Beispielen mit `$` angegeben). Oft steht vor dem Prompt noch eine Pfadangabe (je nach Einstellung der Shell), die in den Beispielen mit der Tilde (~) angegeben wird. Jedes Kommando wird mit `[Enter]` abgeschlossen und damit an das System geschickt.

Kommandos können beim Aufruf sogenannte Argumente übergeben werden. Dabei kann es sich um Parameter (siehe Abschnitt 2.1.1, S. 7) zur Steuerung des Programms oder um Zeichenketten handeln, die dem Programm zur Bearbeitung übergeben werden (beispielsweise Dateinamen). Eingegebene Kommandos und Argumente werden jeweils durch ein Trennzeichen voneinander getrennt, dies ist meist ein Leerzeichen (*whitespace*).

2.1 Arbeiten mit der `bash`

Im Folgenden wird auf einige Kommandos, sowie auf deren Benutzung, eingegangen. Der erste Befehl, den wir kennenlernen, lautet `echo`. Er zeigt die ihm als Argument übergebene Zeichenfolge auf dem Standardausgabe-Gerät (*standardoutput device*, meist der Monitor) an. Nach dem Ausführen des Prozesses kehren wir automatisch auf die Shell zurück und erhalten wieder ein Eingabeprompt.

Beispiel:

```
~$ echo hallo
hallo
~$
```

2.1.1 Parameter

Durch die Übergabe von Parametern können Programmaufrufe modifiziert werden. Sie werden nach dem Kommando selbst angegeben und beginnen meist mit einem - (*dash*).

Durch die Übergabe des Parameters `-n` beim Aufruf von `echo` kann der Zeilenumbruch (*newline*) hinter der Ausgabe unterdrückt werden, so dass der Prompt direkt hinter der Ausgabe von `echo` erscheint.

Beispiel:

```
~$ echo -n hallo
hallo~$
```

Syntax:

`echo [`*Optionen*`]` *Zeichenkette*

`-n` Unterdrücken des Zeilenumbruchs

Viele Parameter sind unter Linux standardisiert. So steht der Parameter `-v` bei vielen Programmen für *verbose* (wortreich) und bewirkt, dass zusätzliche Informationen beispielsweise über den Ablauf des Programms ausgegeben werden.

Schema der Syntax-Angaben
Die Syntax zur Benutzung vorgestellter Kommandos wird im Folgenden immer nach dem gleichen Schema angegeben (s. o.). Das Kommando wird in Nichtproportional-Schrift (`monospace`, `fixed-width`) angegeben. Die Argumente werden in *italic* gesetzt, wobei optionale Parameter außerdem in eckigen Klammern gesetzt sind. Die eckigen Klammern dienen hier nur zur Verdeutlichung und müssen beim Aufruf weggelassen werden.

2.1.2 Dokumentation

Linux bietet eine umfassende Dokumentation zu den systemeigenen Kommandos und Programmen. Diese kann in der Shell abgerufen werden und ist standardmäßig in Englisch verfasst. Zusätzlich bietet das Internet eine große Bandbreite an Informationsquellen.

1. **man-pages**
   Ein Großteil der Kommandos unter Linux ist neben vielen anderen Komponenten in Form von `man`-pages (*manual pages*) dokumentiert. Das Anzeigen der `man`-pages übernimmt der `man-PAGER`. Durch den Aufruf ›`man` *Kommando*‹ wird eine Bedienungsanleitung zu *Kommando* angezeigt. Diese enthält neben einer generellen Beschreibung der Bedienung und einer Beschreibung des Kommandos viel weitere nützliche Hinweise, z. B. auf verwandte Kommandos oder bekannte Fehler des Kommandos (*bugs*).

   Beispiel:

   ```
   ~$ man echo
   ```

   Syntax:

   `man [`*Optionen*`]` *Kommando*

   Wichtige Kommandos zur Bedienung des `man-pagers`:

   `[space]` oder `[z]` Weiterblättern (eine Bildschirmseite)
   `[q]` Verlassen (*quit*)
   `[b]` Zurückblättern (*back*)
   `[/]` Vorwärtssuchen
   `[?]` Rückwärtssuchen

   Der Aufbau von `man`-pages ist mehr oder weniger standardisiert, so dass man sich mit etwas Übung relativ leicht über die Eigenschaften und die Bedienung eines Programms informieren kann.

   Genereller Aufbau einer `man`-page:

   `NAME` – Name und kurze Beschreibung des Kommandos
   `SYNOPSIS` – Syntax der Benutzung
   `DESCRIPTION` – Ausführlichere Beschreibung des Kommandos
   `OPTIONS` – Beschreibung der Parameter
   `FILES` – vom Kommando benötigte Dateien
   `SEE ALSO` – Hinweis auf ähnliche oder verwendete Kommandos
   `DIAGNOSIS` – Fehlerkode
   `BUGS` – bekannte Fehler
   `EXAMPLE` – Beispiele zur Benutzung

   Außerdem enthalten `man`-pages häufig Hinweise auf die Umgebung (`ENVIRONMENT`), die Interaktive Bedienung (`COMMANDS`) des Programmes, Referenzen (`REFERENCES`), Angaben zur Version (`VERSION`), `COPYRIGHT`-Angaben, Angaben zu den Autoren (`AUTHOR`) und viele weitere Informationen.

Konventionen zur Angabe der Syntax (SYNOPSIS):

`bold text`	–Argument muss genauso geschrieben werden wie angegeben.
`italic text`	–Das Unterstrichene ersetzen durch entsprechendes Argument.
`[-abc]`	–Argumente in eckigen Klammern sind optional.
`-a|-b`	–Durch `|` getrennte Argumente, können nicht gleichzeitig in einem Aufruf verwendet werden (exklusives Oder).
`argument ...`	–Das Argument, kann mehrfach übergeben werden (z. B. Dateien).
`[expression] ...`	–Der Ausdruck in eckigen Klammern kann wiederholt werden.

Weiterhin ist in `man`-pages noch angegeben, zu welcher Sektion (*section*) das Dokumentierte gehört, d. h. ob es sich beispielsweise um ein ausführbares Kommando, eine Bibliothek oder etwas anderes handelt. Die Sektion ist in der ersten Zeile der `man`-page als Nummer in Klammern hinter dem Namen angegeben. Sie wird aber auch beim Aufruf von `apropos` (s. u.) mit ausgegben.

- `(1)` – Ausführbare Programme oder Shell-Kommandos
- `(2)` – Systemaufrufe (Kernelfunktionen)
- `(3)` – Funktion in einer Bibliothek, z. B. `print`
- `(4)` – Spezielle Dateien (meist unter `/dev`), z. B. `null`
- `(5)` – Dateiformate und Konventionen, z. B. `protocols`
- `(6)` – Spiele
- `(7)` – Verschiedenes (Macros, Pakete, Konventionen…) z. B. `missing`
- `(8)` – Kommandos, die nur ein Nuter mit Administratorrechten ausführen darf (meist ist dies *root*)
- `(9)` – Kernelroutinen (gehört nicht zum Standard)

2. **Suche nach Stichworten**
   Mit dem Kommando `apropos` kann man nach einem Befehl mit Stichworten suchen, beispielsweise mit welchem Kommando man den aktuellen Kalender anzeigen lassen kann.

   Beispiel:

```
~$ apropos calendar
cal (1)           - displays a calendar and the date of easter
calendar (1)      - reminder service
ncal (1)          - displays a calendar and the date of easter
```

3. **Kurzbeschreibung von Kommandos**
   Mit ›`whatis` *Kommando*‹ erhält man die Kurzbeschreibung von *Kommando* (entspricht der ersten Zeile einer `man`-page).

   Beispiel:

```
~$ whatis apropos
apropos (1)       - search the manual page names and descriptions
```

4. **Weitere Informations-Dokumente**
   Mit dem Kommando `info` können zu den meisten Programmen auch Informationen im

Info-Format eingeholt werden. Die Info-Dokumente enthalten wie die `man`-pages Informationen zur Benutzung von Programmen.

Syntax:

`info [`*Optionen*`]` *Kommando*

2.1.3 Pfade

Das Dateisystem unter Linux ist baumartig aufgebaut (siehe Abbildung 1, S. 11). Das Wurzelverzeichnis (*root*-Verzeichnis) ist `/`. Der Schrägstrich (`/`, *slash*) ist gleichzeitig das Trennzeichen (*separator*) mit dem unter Linux Verzeichnisse (*directory*) und Unterverzeichnisse (*subdirectory*) getrennt werden.[4]

Wichtig ist es zu wissen, dass unter Linux alle angeschlossenen Geräte Teil des Dateisystems sind, d. h. Festplatten und andere Laufwerke sind Dateien, ebenso wie der Monitor und die Tastatur, die unter dem *root*-Verzeichnis `/` liegen (was bedeutet, dass es keine `c:\`-Partition wie unter Windows gibt!). Den Laufwerken wird beim Einhängen ins Dateisystem (*Mounten*) ein beliebiger Name gegeben, der dem Pfad einer »normalen Datei« entspricht. So könnte beispielsweise ein CD-ROM-Laufwerk unter `/cdrom` eingehängt sein.[5]

Das Verzeichnis, in dem man sich aktuell befindet, wird auch als Arbeitsverzeichnis (*working directory*) bezeichnet. Mit dem Kommando `pwd` (*print working directory*) wird die aktuelle Position im Dateisystem (*filesystem*) angegeben. D. h. es wird der Wert der Umgebungsvariable `PWD`[6] ausgegeben, in der die Shell die derzeitige Position des Benutzers im Dateisystem speichert.

Beispiel:

```
~$ pwd
/home/user
```

Im Beispiel befinden wir uns im Verzeichnis `user`, das ein Unterverzeichnis von `home` ist. Dieses ist wiederum ein Unterverzeichnis des *root*-Verzeichnisses `/`. So ergibt sich zusammengesetzt der Pfad `/home/user`.

Jede Datei hat mindestens einen eindeutigen Namen, der als **Pfad** (*path*) bezeichnet wird, dieser sollte keine Leerzeichen enthalten, da der Pfad dann nicht eindeutig ist. Pfade, die Leerzeichen enthalten, müssen in Hochkommata (`' '`) angegeben werden, damit durch Leerzeichen getrennte »Wörter« als Einheit erkannt werden und nicht einzeln interpretiert werden (siehe Quoting, Abschnitt 2.2.3, S. 16).

Unter Linux ist eine maximale Länge von 255 Zeichen für Dateinamen erlaubt. Sonderzeichen (mit Ausnahme von _ und `.`) sollten bei der Namensgebung vermieden werden, da sie z. T. bereits vom Betriebssystem belegt sind (beispielsweise `/`). Zu beachten ist, dass Linux im Gegensatz zu Windows zwischen Groß- und Kleinschreibung unterscheidet (*case sensitive*).

Pfade können jeweils absolut oder relativ angegeben werden. Der absolute Pfad bezieht sich auf das *root*-Verzeichnis, der relative Pfad auf die derzeitige Position des Benutzers im Verzeichnisbaum. Als Vereinfachung können Pfade außerdem mit Hilfe der Tilde `~` angegeben werden.

[4] Unter Windows ist das Standard-Separatorzeichen der Backslash (\), unter MacOS der Punkt.
[5] Auf das Ein- und Aushängen von Geräten wird in Abschnitt 2.2.12, S. 28 näher eingegangen.
[6] Auf die »Umgebung« und Umgebungsvariablen wird in Kapitel 4, S. 52 eingegangen

Graphische Darstellung eines Verzeichnisbaumes

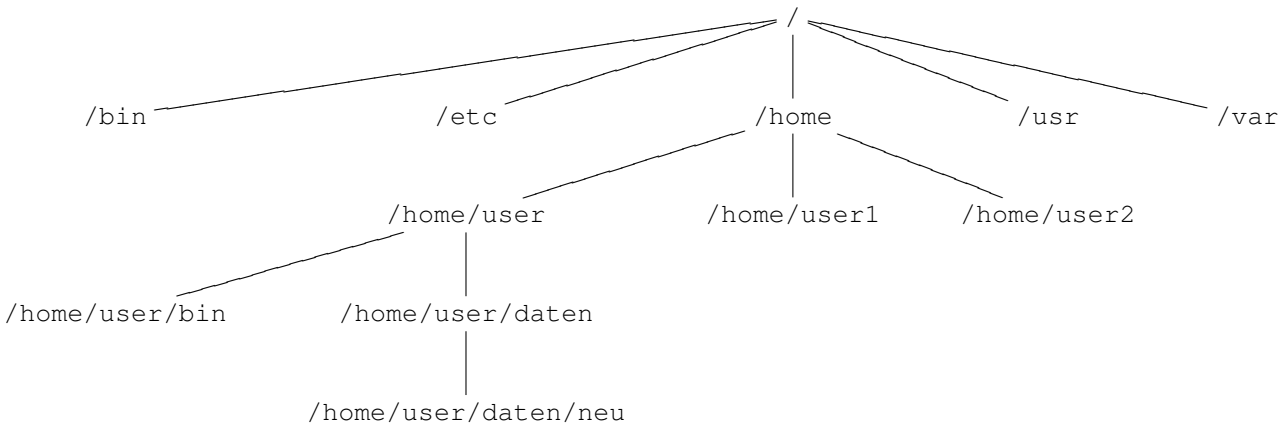

Abbildung 1: *Beispiel zur Verzeichnisstruktur des Dateisystems. Auf den meisten Linux-Rechnern ist der Grundaufbau der Verzeichnisstruktur ähnlich. Die Heimatverzeichnisse der Benutzer sind meist unter* `home` *angelegt (hier mit* `user`*,* `user1` *und* `user2` *bezeichnet), das Verzeichnis* `etc` *enthält häufig Dateien zur Konfiguration des Systems, im* `bin`*-Verzeichnis sind einige Programme zu finden usw.*

Die Tilde stellt unter Linux immer den absoluten Pfad des Heimatverzeichnis des aktuellen Benutzers dar (siehe Wildcards, Abschnitt 2.2.2, S. 15).

Beispiel:

Angenommen, es gibt einen Benutzer `user`, dem das Heimatverzeichnis `/home/user` zugeordnet ist. Das Verzeichnis `neu` beispielsweise kann dann auf die folgenden Arten angesprochen werden:

1. **Absoluter Pfad:**
   `/home/user/daten/neu/`

2. **Relativer Pfad** (wenn der Benutzer sich im Verzeichnis `/home/user` befindet):
   `daten/neu`

3. **Mit Tilde** als Wildcard, wenn man der Benutzer *user* ist (auch absolut!):
   `~/daten/neu`

4. **Mit Tilde** als Wildcard, wenn man **nicht** der Benutzer *user* ist:
   `~user/daten/neu`

2.2 Datenverwaltung

Zur Datenverwaltung stellt die Shell einige Kommandos zur Verfügung. Bei der Benutzung der Programme kann jeweils der absolute oder der relative Pfad der Quell- bzw. Zieldatei angegeben werden.

2.2.1 Dateien und Verzeichnisse

1. Für das **Anzeigen von Dateien/Verzeichnissen** gibt es das Kommando `ls` (*list*).

   Syntax:

   `ls [`*Optionen*`] [`*Pfad*`]`

   Wird `ls` beim Aufruf kein Pfad übergeben, so zeigt es den Inhalt des aktuellen Verzeichnisses an.

   Beispiel:

   ```
   ~$ ls
   bin   datei1   prog
   ```

   Das aktuelle Verzeichnis enthält die beiden Unterverzeichnisse `bin` und `prog` und die Datei `datei1`. In diesem Ausgabeformat werden Verzeichnisse und einfache Dateien nicht unterschieden.[7]

   Mit Hilfe der Parameter `-l` (*long* – Ausgabe in langem Format) und `-a` (*all* – alle Dateien ausgeben) kann die Ausgabe erweitert werden, so dass u. a. der Besitzer der Datei, die Größe und das Datum der letzten Änderung ausgegeben werden. Beide Parameter können in diesem Fall gemeinsam als `-la` übergeben werden.

   Beispiel:

   ```
   ~$ ls -la
   total 228k
   drwxr-xr-x    9 user     group      4096 Aug 10 12:50 .
   drwxr-xr-x   20 user     group      4096 May  8 08:39 ..
   -rw-------    1 user     group      5404 Aug 10 16:11 .bash_history
   -rw-r--r--    1 user     group       461 May 10 09:25 .bashrc
   -rw-r--r--    1 user     group       420 Aug 13 22:51 .emacs
   -rw-r--r--    1 user     group       279 Jul 11 21:29 .profile
   prw-r--r--    1 user     group         0 Aug 15 18:37 FIFO
   drw-r--r--    1 user     group       461 May 10 09:25 bin
   -rw-r--r--    1 user     group       115 Aug  7 23:15 datei1
   drwx------    2 user     group       944 Aug 10 10:13 prog
   ```

   Attribute | Nr | Benutzer | Gruppe | Größe | letzte Änderung | Name

   Die Spalten der Anzeige:
   Im Block der **Attribute** (*attributes*) ist zunächst angegeben, ob es sich bei dem Eintrag um ein Verzeichnis, eine Datei, einen Link oder eine benannte Pipeline (*named pipe*) handelt. Weiterhin können hier noch blockorientierte Geräte, zeichenorientierte Geräte und Sockets aufgeführt sein. Auf diese wird hier nicht weiter eingegangen. Genau genommen handelt es

[7] Auf einigen Systemen ist die Option `-color` voreingestellt, wodurch u. a. Verzeichnisse farblich kenntlich gemacht werden.

sich bei allen hier aufgezählten Typen um Dateien (unter Linux sind auch Festplatten, Ein- und Ausgabegeräte usw. Dateien), im Sprachgebrauch werden aber meistens nur »einfache Dateien« also Text- oder Binärdateien (*plain file*) mit »Datei« angesprochen. In diesem Sinne wird der Ausdruck Datei hier weiterhin verwendet.

- `d` steht für ein Verzeichnis (*directory*)
- `-` steht für eine Datei (*file*)
- `l` steht für einen Link (siehe S. 15)
- `p` steht für eine benannte Pipeline (siehe Abschnitt 2.2.10, S. 25)
- `b` steht für ein blockorientiertes Gerät
- `c` steht für ein zeichenorientiertes Gerät
- `s` steht für einen Socket

Als nächstes folgen die **Benutzer-Rechte** jeweils in 3er-Blöcken. Sie geben an, wer welche Rechte auf der entsprechenden Datei hat (mit Ausnahme des Benutzers *root*, der als Administrator alle Rechte hat.[8]) Die ersten drei Zeichen stehen für den Besitzer (**u** – user), die nächsten drei für die Gruppe (**g** – group) und am Schluss alle anderen Benutzer (**o** – other).

`rwx r-x r-x`
u g o

Im Beispiel hat der Nutzer alle Rechte, die Gruppe und alle anderen jeweils Lese- und Ausführrecht.

- `r` steht für Leserecht (*read*)
- `w` steht für Schreibrecht (*write*)
- `x` steht für Ausführrecht (*execute*)

Die **Nummer** (Nr) in der folgenden Spalte gibt bei einem Verzeichnis an, wieviele Unterverzeichnisse es hat. Handelt es sich um eine Datei, gibt die Nummer an, wieviele »Namen« (Hardlinks, siehe S. 15) für diese Datei existieren.

In der nächsten Spalte (Besitzer) steht der Name des **Besitzers** der Datei (in diesem Fall *user*) gefolgt von der **Gruppe** (*group*), der sie zugeordnet ist. Dann folgen **Größe** (*size*), Datum und Uhrzeit der **letzten Änderung** (*last modified*) und schließlich der **Name** (*name*).

Die Verzeichnisse `.` und `..` stehen für das aktuelle Verzeichnis (`.`)[9] und das übergeordnete Verzeichnis (`..`).

Dateien (oder Verzeichnisse), deren Name mit einem `.` beginnt, sind sogenannte »versteckte Dateien«. Sie werden nur beim Aufruf mit `-a` angezeigt. Oft handelt es sich hierbei um System- oder Konfigurationsdateien.

2. **Bewegen im Dateisystem**
   Mit dem Kommando `cd` (*change directory*) kann man sich im Dateisystem bewegen. Als Argument wird das Zielverzeichnis übergeben.
   Durch den Aufruf

```
~$ cd ..
```

[8] Unter Linux gibt es immer mindestens einen *Superuser*, der als Administrator alle Rechte auf dem System hat. Dies ist standardmässig *root*.

[9] Vereinfachung falls ein Programm ein Verzeichnis als Argument erwartet und man das Verzeichnis übergeben will, in dem man sich derzeit befindet. In anderen Fällen wird der `.` auch als synonym für das Schlüsselwort `source` (Quelle) verwendet (siehe Kapitel 4, S. 52).

wechselt man beispielsweise ins übergeordnete Verzeichnis.
Beim einfachen Aufruf von `cd` ohne Pfadangabe wird in das Heimatverzeichnis des aktuellen Benutzers gewechselt.

Syntax:

`cd [`*Pfad*`]`

3. Das **Löschen von Dateien** wird durch das Kommando `rm` (*remove*) aufgerufen.

   Syntax:

   `rm [`*Optionen*`]` *Datei*

   Löscht die Datei *Datei* ohne Vorwarnung. Wird der Parameter `-i` (*interactive*) beim Aufruf von `rm` übergeben, so muss jeder Löschvorgang explizit mit `y` (*yes*) bestätigt werden (in manchen Systemen aus Sicherheitsgründen voreingestellt).

   Um ein Verzeichnis zu löschen, muss der Parameter `-r` (*recursive*) übergeben werden. Das Verzeichnis wird dann rekursiv mit allen Unterverzeichnissen gelöscht. Alternativ können leere Verzeichnisse auch mit dem Kommando `rmdir` gelöscht werden.

4. Zum **Kopieren von Daten** wird das Kommando `cp` verwendet.

   Syntax:

   `cp [`*Optionen*`]` *Quelldatei Zieldatei*

   `cp` kopiert die Datei *Quelldatei* nach *Zieldatei*. Existiert *Zieldatei* bereits, so wird sie ohne Warnung überschrieben (sofern der aufrufende Benutzer Schreibrecht für *Zieldatei* hat). Handelt es sich bei *Zieldatei* um ein Verzeichnis, so wird *Quelldatei* in dieses Verzeichnis kopiert. Wie `rm` kann auch `cp` mit dem Parameter `-i` im interaktiven Modus ausgeführt werden. Dies hat zur Folge, dass der Benutzer gewarnt wird, bevor Dateien überschrieben werden. Es können auch mehrere Dateien als Quelldatei angegeben werden, dann muss die Zieldatei allerdings ein Verzeichnis sein. Um ein Verzeichnis zu kopieren, muss der Parameter `-r` übergeben werden.

5. Das **Verschieben/Umbenennen von Dateien** erfolgt mit dem Kommando `mv` (*move*).

   Syntax:

   `mv [`*Optionen*`]` *Quelle Ziel*

   Ebenso wie `cp` und `rm` kann auch `mv` durch den Parameter `-i` im interaktiven Modus ausgeführt werden. Ist als Zieldatei ein Verzeichnis angegeben, wird die Quelldatei in dieses verschoben. Wie bei `cp` können hier auch mehrere Quelldateien oder Verzeichnisse angegeben werden.

6. Das **Erstellen von Verzeichnissen** erfolgt mit Hilfe des Kommandos `mkdir`.

   Syntax:

   `mkdir [`*Optionen*`]` *Verzeichnisname*

   Legt das Verzeichnis *Verzeichnisname* an. Dabei muss darauf geachtet werden, dass alle übergeordneten Verzeichnisse bereits existieren. Ist dies nicht der Fall kann mit dem Parameter `-p` (*parent*) dafür gesorgt werden, dass alle übergeordneten Verzeichnisse, die noch nicht existieren, angelegt werden.

7. Das **Anlegen von Links** erfolgt mit dem Aufruf des Kommandos `ln`.

   Syntax:

   `ln [`*Optionen*`]` *Quelldatei Linkname*

   Es wird ein Link *Linkname* auf die Datei *Quelldatei* gesetzt. D. h. es wird ein zweiter Name für die Datei angelegt, womit sie zwei gleichwertige Pfade besitzt (*hardlink*). Der Parameter `-s` bewirkt, dass nur ein »symbolischer« Link (*symbolic-/softlink*) erstellt wird. Dieser enthält nur einen Verweis auf die Quelldatei. Dabei ist jedoch das tatsächliche Vorhandensein der Datei, auf die er verweist, nicht garantiert. Dateien, die durch mehrere harte Links referenziert werden, existieren so lange, bis alle Links gelöscht sind. Dennoch sind symbolische Links vorzuziehen, da sie Dateisystem übergreifend benutzt werden können und harte Links im Dateisystem »Verwirrung« verursachen können.[10]

   Beispiel:

   ```
   ~$ ln  -s /opt/blast/blastall ~/bin/blastall
   ```

   Im Beispiel wird ein Softlink auf das Programm `blastall` gesetzt. Mit dem Aufruf `ls -l` würde dieser Link folgendermaßen dargestellt werden:

```
lrwxrwxrwx   1 user    user   19 Jul 10 12:41 blastall -> /opt/blast/blastall
```

8. Auf das **Anlegen von benannten Pipelines** (*named pipes*) wird in Abschnitt 2.2.10, S. 25 eingegangen.

2.2.2 Wildcards (Ersetzungs-/Maskenzeichen)

Um mehrere Dateien gleichzeitig ansprechen zu können, stellt die Shell sogenannte Wildcards zur Verfügung, mit deren Hilfe Gruppen von Zeichen maskiert (d. h. gemeinsam angesprochen) werden können. Eine Art Wildcard haben wir bereits kennengelernt, die Tilde `~`. Die Tilde steht für das Heimatverzeichnis des aktuellen Benutzers. Der Ausdruck ~user bezeichnet das Heinmatverzeichnis des Benutzers »user«. Andere Wildcards sind `?` und `*`. In eckigen Klammern (`[ ]`) kann eine Liste von Zeichen angegeben werden, die als Treffer gewertet werden.

- `?` steht für genau ein Zeichen
- `*` steht für ein, kein oder beliebig viele Zeichen

[10] Name und Hardlinks sind gleichwertig. Das System kann nicht mehr unterscheiden, was zuerst da war.

`[`Liste von Zeichen`]` steht für genau ein Zeichen aus der Liste.

`[!`Liste von Zeichen`]` steht für genau ein Zeichen, das **nicht** in der Liste steht. Das `!` negiert die Liste.

Beispiele:

```
~$ rm b*
```

löscht alle Dateien, die mit einem `b` beginnen.

```
~$ rm [_a-z]*
```

löscht alle Dateien, die mit einem Kleinbuchstaben oder einem Unterstrich beginnen. Zu beachten ist, dass der `-` nicht als normales Zeichen betrachtet wird. Er gehört zu den Metazeichen, einer Gruppe von Zeichen, die von der Shell funktionell belegt sind (wie beispielsweise auch `$`). Auf den Umgang mit Metazeichen wird im nächsten Abschnitt eingegangen.

2.2.3 *Quoting* – Verhindern der Shell-Interpretation

Die Shell interpretiert die Zeichenketten der Eingabe als Kommandos, Parameter, Pfadnamen usw. Dabei haben bestimmte Zeichen eine Sonderbedeutung, die sogenannten Metazeichen. Bei der Interpretation werden diese Zeichen ersetzt. Manchmal ist es allerdings notwendig, die Interpretation durch die Shell zu unterdrücken. Das haben wir bereits bei den Pfadnamen mit Leerzeichen kennengelernt. Hier musste der Pfad in Hochkommata gesetzt werden, damit er als ein Pfad erkannt wird. Ein weiteres Beispiel ist der Umgang mit Wildcards, wenn diese nicht als Metazeichen, sondern als »echte Zeichen« interpretiert werden sollen.

Metazeichen:

```
; $ & ( ) < > [ ] { } ? * " ` ' ~ # % \
```

Die Shell bietet verschiedene Mechanismen, mit denen die Interpretation verhindert werden kann. Diese werden auch als *Quoting*-Mechanismen bezeichnet. Zum Quoting bietet die Shell neben den Hochkommata und Anführungszeichen den Backslash `\`.

Quoting-Mechanismen

Backslash
Der Backslash schützt das folgende Zeichen vor der Interpretation durch die Shell. So wird `\*` als `*` interpretiert und nicht als »beliebiges Zeichen«. Um längere Zeichenketten vor der Interpretation zu schützen, ist der Backslash allerdings etwas umständlich, hier verwendet man besser Hochkommata oder Anführungszeichen.

Hochkommata und Anführungszeichen
Bei der Verwendung von Hochkommata und Anführungszeichen ist zu beachten, dass diese sich nicht gleich verhalten. Hochkommata verhindern jegliche Interpretation durch die Shell, während die Anführungszeichen einige zulassen. So werden beispielsweise Umgebungsvariablen in Anführungszeichen interpretiert, in Hochkommata dagegen wird die Variable als einfache Zeichenkette betrachtet. Wichtig ist außerdem zu wissen, dass in Hochkommata auch Hochkommata, die mit dem Backslash gequotet (`\'`) sind, als Hochkommata interpretiert werden.

Mit dem Kommando `echo` kann das Quoting von Zeichen gefahrlos ausprobiert werden. Mit `echo` können Umgebungsvariablen wie die Variable `PWD`, in der die Shell die aktuelle Position des Benutzers im Dateisystem speichert[11], angezeigt werden. Variablen werden für die Shell durch das `$`-Zeichen kenntlich gemacht, so dass `PWD` als `$PWD` angesprochen wird.

Beispiele:

(1)
```
~$ echo $PWD
/home/user/daten
```

(2)
```
~$ echo \$PWD
$PWD
```

(3)
```
~$ echo "$PWD"
/home/user/daten
```

(4)
```
~$ echo "\$PWD"
$PWD
```

(5)
```
~$ echo '$PWD'
$PWD
```

(6)
```
~$ echo 'zeichenkette \' $PWD
zeichenkette \ /home/user/daten
```

(7)
```
~$ echo 'zeichenkette \' $PWD '
>
```
Die Shell interpretiert in Beispiel (7) `\'` als Hochkomma und betrachtet die (erste) Zeichenkette als beendet. Darauf folgt jedoch eine Variable (`$PWD`) und eine weitere Zeichenkette wird begonnen. Die Shell wartet hier (nach dem Drücken von `[Enter]`) darauf, dass die Eingabe abgeschlossen wird, beispielsweise mit:
```
> hallo'
zeichenkette \ /home/user/daten
hallo
```

2.2.4 Dateiinhalte

Zum **Anzeigen** von Dateiinhalten werden u. a. die Programme `more`, `less`, `head`, `tail` und `cat` zur Verfügung gestellt. Wann man welches der Programme vorzieht, ist eine Frage des Geschmacks und der gewünschten Funktionalitäten.

1. Zur Ausgabe einer ganzen Datei verwendet man `less`, `more` oder `cat` wie im Beispiel gezeigt:

   Beispiel:

   ```
   ~$ less Datei
   ```

[11] Auf Umgebungsvariablen wird in Kapitel 4, S. 52 eingegangen.

Durch den Aufruf wird der Inhalt von `Datei` auf das Standardoutput-Device ausgegeben. Der Aufruf von `more` und `less` tut im Großen und Ganzen dasselbe, mit dem Unterschied, dass Dateien mit `less` vorwärts und rückwärts durchmustert werden können und noch einige weitere Funktionalitäten geboten werden.

`less` und `more` sind seitenbasiert, d. h. sie geben zunächst den Teil der Datei aus, der in das Fenster der Konsole passt, `cat` durchläuft die Datei sofort bis zum Ende.

Wichtige Kommandos zur Bedienung von `less`[12]:

`[space]` oder `[z]` Weiterblättern (eine Bildschirmseite)

`[q]` Verlassen (*quit*)

`[b]` Zurückblättern (*back*)

`[/]` Vorwärts Suchen

`[?]` Rückwärts Suchen

2. **Anzeigen des Anfangs/Endes einer Datei**
Um nur den Anfang bzw. das Ende einer Datei anzuzeigen, gibt es die Kommandos `head` und `tail`. Bei ihrem Aufruf geben sie standardmäßig die ersten bzw. letzten `10` Zeilen einer Datei aus. Diese Programme werden vor allem verwendet, wenn man eine Information benötigt, von der man weiss, dass sie am Anfang oder Ende einer Datei steht, die sehr groß ist.

Syntax:

`head [`*Optionen*`]` *Datei*

`tail [`*Optionen*`]` *Datei*

Mit dem Parameter `-n` kann die Anzahl der auszugebenden Zeilen übergeben werden.

3. **Dateien aneinanderhängen**
Mit `cat` (*concatenate*) können außerdem mehrere Dateien aneinander gehängt werden.

Syntax:

`cat` *Datei1 Datei2*

Datei1 und *Datei2* werden direkt nacheinander ausgegeben, als handele es sich um nur eine Datei. Wird `cat` nur eine Datei übergeben, so wird nur diese ausgegeben (s. o.).

4. **Zählen von Worten**
Ein weiteres nützliches Kommando ist `wc` (*word count*). Beim Aufruf wird die Anzahl der Zeilen, Wörter und Bytes der übergebenen Datei ausgegeben.

Syntax:

`wc [`*Optionen*`]` *Datei*

Beispiel:

[12] Eine Beschreibung der Navigation in `more` und `less` ist in der `man`-page gegeben.

```
~$ wc Datei
     4      12      92 Datei
```

Die Datei `Datei` im Beispiel enthält 4 Zeilen, 12 Wörter und 92 Bytes.

2.2.5 Speicher anzeigen

1. Mit dem Kommando `quota` kann sich der Benutzer den für ihn verfügbaren Speicherplatz und die maximal erlaubte Anzahl an Dateien anzeigen lassen, sofern das Programm installiert ist.

2. Mit `du` (*disk usage*) wird die **Größe eines Verzeichnisses** und seiner Unterverzeichnisse in Blocks ausgegeben. Mit dem Parameter `-h` (*human readable*) wird die Ausgabe in Kilo-, Mega- bzw. Gigabyte umgerechnet.

3. `df` gibt die **verfügbare Kapazität** der gemounteten Festplatten aus. Genauso wie bei `du` kann die Ausgabe durch den Parameter `-h` angepasst werden.

2.2.6 Archivieren von Daten

Mit dem Befehl `tar` (*tape archiver*) können mehrere Dateien oder ein Verzeichnis in ein **Archiv** (eine Datei) gepackt werden. Mit `tar` erstellte Archive werden meist mit dem Suffix `.tar` gekennzeichnet.

Syntax:

`tar -cf` *Archiv Verzeichnis*
legt *Archiv* an und packt *Verzeichnis* (hier kann auch eine Liste von Dateien angegeben werden) hinein.

`~$ tar -xf` *Archiv*
entpackt das Archiv *Archiv*.

Zum **Komprimieren** von Daten stehen die Befehle `gzip` (GNU[13]) und `compress` (Unix, kommerziell) zur Verfügung. Die zu komprimierende Datei wird dabei durch die komprimierte Version, die durch ein Suffix gekennzeichnet wird, ersetzt. Mit `gzip` komprimierte Dateien werden defaultmäßig mit dem Suffix `.gz` gekennzeichnet (d. h. aus *Dateiname* wird automatisch *Dateiname*`.gz`). Mit `compress` komprimierte Dateien werden mit dem Suffix `.Z` versehen.

Syntax:

`gzip [`*Optionen*`] [-S` *Suffix*`]` *Dateiname*
Mit dem Parameter `-S` kann ein alternatives Suffix übergeben werden, das an den Dateinamen angehängt wird.

`compress [`*Optionen*`]` *Dateiname*

[13] Abkürzung für »GNU's Not Unix« – entstanden aus Projekten zur freien Software Entwicklung, das von FSF (Free Software Foundation) koordiniert wird. Viele der hier vorgestellten Programme entstammen GNU-Projekten und stehen damit unter GNU Public Licence (GPL). D. h. diese Programme stehen mit Quellkode (*source code*) frei zur Einsicht/Benutzung und Veränderung zur Verfügung.

Bei der Kompression von Daten ist die Kompressionsrate stark von dem Datentyp abhängig. Mit `gzip` werden im Allgemeinen Kompressionsraten von ca. 60% erreicht.

Das **Dekomprimieren** erfolgt jeweils mit den entsprechenden Kommandos `gunzip` bzw. `uncompress`.

Syntax:

`gunzip [`*Optionen*`]` *Dateiname*.gz
Das Programm `gunzip` entpackt mit `gzip` gepackte Dateien.

`uncompress [`*Optionen*`]` *Dateiname*.Z
Das Programm `uncompress` entpackt mit `compress` gepackte Dateien.

Um ein Verzeichnis zu komprimieren, muss dieses zunächst mit `tar` in ein Archiv gepackt werden. Das Archivieren und Komprimieren kann getrennt oder in einem Schritt ausgeführt werden. Häufig wird auch die Endung `.tgz` verwendet, wenn `tar` und `gzip` verwendet werden.

Beispiel:

`~$ tar -cf` *Verzeichnisname*.tar *Verzeichnisname*

`~$ gzip` *Verzeichnisname*.tar

Analog kann der Parameter `-z` (bzw. `-Z`) beim Aufruf von `tar` übergeben werden. Dieser bewirkt, dass automatisch das Programm `gzip` (bzw. `compress`) mit aufgerufen wird.

Beispiel:

`~$ tar -czf` *Verzeichnisname*.tgz *Verzeichnisname*

Das gleiche gilt für das Entpacken von Dateien/Verzeichnissen, die mit `tar` archiviert und anschließend komprimiert wurden.

Beispiel:

`~$ tar -xzf` *Verzeichnisname*.tgz

Mit dem Parameter `-c` kann eine komprimierte Datei entpackt werden, ohne dass die komprimierte Version gelöscht wird. Die entpackte Datei wird dabei auf Standardout ausgegeben. Die Übergabe des Parameters `-c` beim Aufruf von `gunzip` entspricht der Verwendung des Progammes `zcat`.

Oft müssen unter Linux auch Dateien entpackt werden, die als `ZIP`-Archive vorliegen. Diese sind meist unter MS-DOS-Systemen gepackt und mit dem Suffix `zip` versehen. Dazu steht das Programm `unzip` zur Verfügung.

Syntax:

`unzip [`*Optionen*`]` *Dateiname*.zip

2.2.7 Auffinden von Daten

Als Hilfe beim Auffinden von Ausdrücken (*expressions*) oder Mustern (*pattern*) bzw. Dateinamen werden die Programme `locate`, `find`, `grep`, `whereis` und `which` vorgestellt.

1. **Suchen in allen Pfaden**
   Mit `locate` wird der Verzeichnisbaum nach Dateien durchsucht, deren Name das übergebene Suchmuster enthält.

   Syntax:

   `locate` *Suchmuster*

2. **Suche im angegebenen Pfad**
   Das Programm `find` sucht im angegebenen Pfad rekursiv in allen Unterverzeichnissen nach Dateinamen, die den übergebenen Ausdruck enthalten.

   Syntax:

   `find [`*Pfad*`] [`*Ausdruck*`]`

   Beispiel:

   ```
   ~$ find / -name 'blast'
   /opt/blast
   ```

   Der gesamte Verzeichnisbaum wird nach Dateien durchsucht, deren Name (Pfad) das Wort `blast` enthält.

3. **Suche in Dateien**
   Mit dem Kommando `grep` wird ein Programm zum Auffinden von Suchmustern in Dateien gestartet. Es wird jeweils die Datei, in der das Muster gefunden wird und die Zeile ausgegeben. Wird beim Aufruf keine Datei übergeben, so durchsucht `grep` die Standardeingabe (*standardinput*). Bei dem Suchmuster kann es sich auch um sogenannte Reguläre Ausdrücke (*regular expressions*) handeln. Dazu muss beim Aufruf von `grep` der Parameter `-E` übergeben werden.

   Syntax:

   `grep [`*Optionen*`]` *Suchmuster Datei*
   `grep [`*Optionen*`] -E` *RegEx Datei* (analog zu `egrep`)

   Reguläre Ausdrücke (*RegEx*, *regular expressions*) sind spezielle Suchmuster, die auf mehrere Zeichenketten passen können (vgl. Wildcards). Sie können mitunter recht komplex sein und bedürfen einiger Aufmerksamkeit. Es gibt ganze Bücher, die sich ausschließlich mit dem Aufbau und der Anwendung von Regulären Ausdrücken beschäftigen. Einige Ausdrücke sind in der `man`-page von `grep` beschrieben. Im `Perl`-Teil des Kurses werden wir uns eingehender mit ihnen beschäftigen.

Einige Parameter zur Bedienung von `grep`:

`i` bedeutet, Groß- und Kleinschreibung werden nicht unterschieden.

`l` bewirkt, dass nur die Dateinamen der Dateien, in denen das Suchmuster gefunden wurde, angegeben werden. Die Ausgabe der Zeile wird unterdrückt.

`w` das angegebene Suchmuster muss als »eigenständiges Wort«, also nicht alt Teilwort, vorkommen.

`v` bewirkt, dass alle Zeilen bzw. Dateien angezeigt werden, die das Suchmuster nicht enthalten.

`x` bedeutet, es wird nur nach Zeilen gesucht, die genau dem Suchmuster entsprechen, also keinerlei sonstige Zeichen enthalten.

Beispiel:

```
~$ grep -i  wort Datei1 Datei2
```

Es werden alle Vorkommen von »wort, Wort, WORT, WOrt, usw.« in `Datei1` und `Datei2` ausgegeben.

4. **Erfragen, welches Programm verwendet wird**
   Mit dem Kommando `which` erfragt man den Pfad eines Kommandos oder Programms. Der Pfad wird ausgegeben, sofern das Kommando im `PATH` (`PATH` ist wie `PWD` eine Umgebungsvariable der Shell) gefunden wurde.

   Syntax:

   `which` *Kommando*

5. **Programme im `PATH` finden**
   Mit dem Befehl `whereis` können Programme im `PATH` gesucht werden (vgl. `which`). Wird das übergebene Kommando gefunden, so werden die Pfade der Binaries (ausführbarer Kode), des Quellkodes und der `man`-page ausgegeben.

   Syntax:

   `whereis` *Kommando*

2.2.8 Benutzerrechte setzen

Jede Datei und jedem Verzeichnis sind unter Linux verschiedene Rechte zugeordnet, die jeweils für den Besitzer, die Gruppe oder alle anderen gelten. Um die Rechte, die einer Datei zugeordnet sind, zu ändern, gibt es das Kommando `chmod` (*change mode*). Dabei kann allerdings nur der Eigentümer einer Datei die Benutzerrechte verändern.

Syntax:

`chmod [`*Optionen*`]` *Gruppe(n)* `[+-=]` *Recht(e) Datei*

Beispiel:

```
~$ chmod ug+wr Datei1 Datei2
```

Im Beispiel wird dem Besitzer und der Gruppe Schreibrecht und Leserecht für die Dateien `Datei1` und `Datei2` gegeben.

***Die Gruppen*:**

- `u` steht für den Benutzer (*user*)
- `g` steht für die Gruppe (*group*)
- `o` steht für alle anderen Benutzer (*other*)
- `a` steht für alle (*all* = ugo)

Veränderung der Rechte:
Die Rechte entsprechen den Benutzerrechten (Siehe 1, S. 13).

- `+` Hinzufügen von Rechten.
- `-` Entzug von Rechten.
- `=` Genaues Zuweisen von Rechten.

Zahlendarstellung der Rechte:
Die Rechte können alternativ auch als Zahlen dargestellt werden. Ein vergebenes Leserecht (r) wird dabei durch die Zahl 4 identifiziert, ein Schreibrecht (w) durch eine 2 und ein Ausführungsrecht (x) durch 1. Die Rechte einer Person/Gruppe ergeben sich aus der Summe der vergebenen Rechte.

Beispiel:

```
~$ chmod 755 Datei
```

755 entspricht der Rechteverteilung:

$$\underbrace{\texttt{rwx}}_{7}\underbrace{\texttt{r-x}}_{5}\underbrace{\texttt{r-x}}_{5}$$

Im Beispiel werden dem Besitzer von `Datei` Lese-, Schreib- und Ausführrechten (4+2+1=7) gegeben, die Gruppe und alle anderen Benutzer werden jeweils nur mit Lese- und Ausführrecht (4+1=5) ausgestattet.

Standard-Benutzerrechte definieren
Um die Rechte beim Schreiben einer Datei nicht immer manuell anpassen zu müssen, kann eine Standardmaske gesetzt werden. Dazu bietet die Shell das Kommando `umask`. `umask` erwartet bei der Eingabe die Rechte als Ziffern dargestellt. Alle Dateien, die der Benutzer nach dem Setzen der Standardmaske erzeugt, werden automatisch **ohne** diese Rechte angelegt. D. h. die Ziffern-darstellung ist genau umgekehrt zu der bei `chmod`. Dabei gelten hier unterschiedliche Regeln für

einfache Dateien und für Verzeichnisse. Dateien erhalten zunächst vom System dem Wert 666, Verzeichnisse den Wert 777. Von diesem Werten wird dann der `umask`-Wert abgezogen:

	Dateien	Verzeichnisse
Voreinstellung	666	777
Standardmaske	022	022
Resultierend Rechte	644	755

Standardmäßig ist die Maske mit `022` voreingestellt. D. h. alle Verzeichnisse, die der Nutzer anlegt werden mit allen Rechten für ihn und mit Lese- und Ausführrecht für die Gruppe und den Rest der Welt ausgestattet. Alle einfachen Dateien, die er erzeugt, werden für ihn mit Lese- und Schreibrecht und für alle anderen mit Leserecht versehen. Um eine andere Maske als die voreingestellte für jedes Einloggen zu definieren, muss sie im der `.profile` bzw. der `.shrc` gesetzt werden (siehe Kapitel 4, 52).

Syntax:

`umask [`*maske*`]`

Beispiel:

```
~$ umask 022
```

(analog für eine Datei: `chmod 644` *Datei*)

Die Maske im Beispiel bewirkt, dass der Benuzter Lese- und Schreibrecht, die Gruppe und alle anderen nur Leserecht erhalten.

Wird `umask` ohne Argumente aufgerufen, wird die vollständige Rechtemaske des Benutzers angezeigt. Es werden hier vier Stellen angegeben. Die erste Stelle bezeichnet spezielle Rechte, auf hier die nicht weiter eingegangen wird. Die folgenden drei Stellen geben die Standardmaske des Benutzers an.

2.2.9 Dateieigentümer ändern

Um den Eigentümer einer Datei und die Gruppe, der sie zugeordnet ist, zu ändern, gibt es das Kommando `chown`. Mit `chown` kann entweder nur ein neuer Eigentümer oder auch eine neue Gruppe gesetzt werden. Beide werden dann durch einen Doppelpunkt voneinander getrennt. Soll nur die Gruppe geändert werden, so kann man dies auch mit dem Kommando `chgrp` erfolgen. Den Eigentümer einer Datei kann nur *root* ändern, um die Gruppe einer Datei zu ändern muss man zum einen Besitzer der Datei sein und zum andern Mitglied der Gruppe.

Syntax:

`chown [`*Optionen*`]` *User*`[:`*Gruppe*`]` *Datei*

`chgrp [`*Optionen*`]` *Gruppe Datei*

Beispiel:

```
chown user1 data.txt
```

Im Beispiel wird die Datei `data.txt` dem Benutzer `user1` übergeben.

2.2.10 Pipen, Ein- und Ausgabeumleitung

Die Ein- und Ausgabe am Rechner werden meist als Ströme bezeichnet. Als Standardinput (stdin, Standardeingabe, Deskriptor: `0`) haben wir die Eingabegeräte Maus und Tastatur kennengelernt, als Standardoutput (stdout, Standardausgabe, Deskriptor: `1`) den Monitor. Es gibt noch einen weiteren Standardstrom, Standarderror (stderr, Standardfehler, Deskriptor: `2`), zur Ausgabe von Fehlermeldungen (dieser wird häufig auf stdout ausgegeben).

Die Ströme können beliebig umgeleitet werden, d. h. die Ausgabe eines Programms kann beispielsweise in eine Datei geschrieben werden oder als Eingabe eines anderen Programms dienen. Der Vorteil ist hier, dass dabei die Daten nicht auf Datenträger (Festplatte o. ä.) geschrieben werden, sondern im Arbeitsspeicher (Puffer) gehalten werden und direkt weiter verwendet werden können.

1. **Umleiten der Ausgabe in eine Datei**
   Umleitung des Datenstromes vom Standardoutput-Device (`1`) in eine Datei erfolgt mit Hilfe des Operators `>`. Im Beispiel wird dem Kommando `cat` eine Datei *Datei1* übergeben, die in *Datei2* umgeleitet wird. Existiert *Datei2* bereits, so wird der Inhalt überschrieben.

   Beispiel:

   `~$ cat` *Datei1* > *Datei2*

   Datei2 enthält nach dem Ausführen den Inhalt von *Datei1*.

   Analog lassen sich auch zwei oder mehr Dateien in *Datei2* speichern:

   Beispiel:

   `~$ cat` *Datei1 Datei3* > *Datei2*

   Nach diesem Aufruf stehen der Inhalt von *Datei1* und *Datei3* hintereinander in *Datei2*.

   Hinweis: Eigentlich muss bei der Umleitung ein Deskriptor mit angegeben werden. Im Fall der Standardausgabe ist dieser `1`. Er muss in diesem Fall nicht mitangegeben werden, da `1` der Standardwert bei der Ausgabeumleitung ist. Das gleiche gilt für die Eingabeumleitung, deren Deskriptor die `0` ist. Bei der Umleitung des Fehlerstroms (s. u.) beispielsweise muss der Deskriptor (`2`) explizit mit angegeben werden.

2. **Anhängen an existierende Dateien**
   Das Anhängen an bereits existierende Dateien erfolgt mit Hilfe des Operators `>>`. Im Beispiel wird das Datum mit dem Kommando `date` erfragt und die Ausgabe in die Datei *Datei* umgeleitet.

   Beispiel:

   `~$ date >` *Datei*

   Mit `cat` zeigen wir den Inhalt der Datei an:

   ```
   ~$ cat Datei
   Tue Oct 21 17:20:04 CEST 2003
   ```

Beim erneuten Aufruf von `date` soll die Ausgabe in *Datei* geschrieben werden, ohne dass deren Inhalt verloren geht.

```
~$ date >> Datei
~$ cat Datei
Tue Oct 21 17:20:04 CEST 2003
Tue Oct 21 17:20:45 CEST 2003
```

3. **Umleiten des Eingabestromes**
   Das Umleiten des Eingabestromes (`0`) erfolgt analog zur Ausgabeumleitung. Hier muss der Operator < verwendet werden.

   Beispiel:

   `grep` *Suchmuster* < *Datei*

   Der Aufruf im Beispiel bewirkt, dass `grep` das angegebene Suchmuster in *Datei* sucht. Wird bei der Eingabeumleitung hinter dem < nichts übergeben, so wartet das Kommando auf Eingaben vom Standardeingabestrom, also der Tastatur.

4. **Umleiten von Fehlerausgaben**
   Das Umleiten der Fehlerausgabe (`2`) in eine Datei erfolgt mit `2>`. Im Beispiel wird beim Aufruf von `less` kein Argument übergeben, worauf eine Fehlermeldung erscheint.

   Beispiel:

   ```
   ~$ less
   Missing filename ("less --help" for help)
   ```

   Die Fehlerausgabe wird in *Datei* umgeleitet:

   ```
   ~$ less 2> Datei
   ```

   Das Umleiten von stderr ist vor allem dann nützlich, wenn die Fehlerausgabe sehr umfangreich ist oder wenn »Ergebnis-« Ausgabe und Fehlermeldungen getrennt werden sollen. Ist man an der Ausgabe von Fehlermeldungen nicht interessiert, kann man sie auch ins »Nirgendwo« (`/dev/null`[14]) umleiten.

   Beispiel:

   Durch den Aufruf:

   ```
   ~$ find / -name 'blast' > Datei
   ```

   erhalten wir das gewünschte Ergebnis in *Datei* und auf dem Monitor zahlreiche Fehlermeldungen, da wir als einfacher Benutzer in vielen Verzeichnissen kein Lese- oder Ausführrecht haben.

   ```
   find: /root/.ssh: Permission denied
   find: /root/.gconfd: Permission denied
   find: /root/.kde: Permission denied
   find: /root/.gnome: Permission denied
   ```

[14] Was nach `/dev/null` kopiert oder umgeleitet wird, ist unwiderruflich weg.

Da wir wissen, dass diese Fehlermeldungen auftreten, aber für unsere Belange nicht von Bedeutung sind, leiten wir den Fehlerstrom um.

Durch den Aufruf:

`~$ find / -name 'blast' >` *Datei* `2> /dev/null`

wird das Ergebnis in *Datei* und die Fehlerausgaben in `/dev/null` umgeleitet.

Um Ausgabe- und Eingabestrom gemeinsam umzuleiten, steht außerdem der Parameter `&>` rameter `&>` (oder `>&`) zur Verfügung.

5. **Pipen von Kommandos**
Die Verkettung von mehreren Programmen erfolgt mit Hilfe der sogenannten Pipeline. Die Ausgabe jedes Programms wird dabei an das folgende weitergereicht. Dabei können im Prinzip beliebig viele Kommandos miteinander verkettet werden. Der Vorteil ist, dass die Ausgaben hierbei weitergereicht werden, ohne dass sie zwichengespeichert werden.

Syntax:

Kommando1 | *Kommando2* | *Kommando3*

Beispiel:

`cat` *Datei* | `grep` *Suchmuster*

Der Aufruf im Beispiel bewirkt, dass `grep` *Datei* nach dem angegebenen *Suchmuster* durchmustert.

6. **Benannte Pipelines (*named pipes*)**

Eine benannte Pipeline ist im Grunde das gleiche wie eine einfache Pipeline mit dem Unterschied, dass sie einen Namen hat und ins Dateisystem eingetragen ist. Analog zur einfachen Pipeline dient sie zur »Kommunikation« von Prozessen. D. h. ein Prozess kann auf die Pipeline gesetzt werden und wartet darauf, dass andere Prozesse etwas hineingeben. Genauso kann ein Prozess seine Ausgabe in die Pipeline schreiben und ein beliebiger anderer Prozess kann sie später als Eingabe verwenden.

Das Anlegen einer benannten Pipeline erfolgt mit dem Kommando `mknod`. Das Kommando `mknod` dient außerdem zum Anlegen von Blocks und weiteren »Spezialdateien«. Durch die Übergabe des Parameters `p` wird `mknod` angewiesen eine benannte Pipeline zu erstellen.

Syntax:

`mknod` *Name* `p`

Kommando1 > *Name*

Kommando2 < *Name*

Beispiel:

```
~$ mknod FIFO p
~$ cat Datei >FIFO
```

Im Beispiel wird die Pipeline `FIFO` (*first in first out*) erzeugt und die Ausgabe von `cat` hineingegeben. Auf einer anderen Konsole wird `FIFO` als Eingabe für `grep` verwendet (vgl. Bsp. aus 5.).

```
~$ grep Suchmuster < FIFO
```

Das Beispiel kann ebenso gut andersherum angewendet werden. Das würde bedeuten, das Kommando `grep` wird auf die Pipeline gesetzt und sobald etwas in `FIFO` umgeleitet wird, werden die Vorkommen von Suchmustern ausgegeben.

2.2.11 Sortieren von Eingaben

Mit dem Kommando `sort` können beliebige Eingaben sortiert werden. Dabei kann es sich um die Ausgabe eines anderen Programms handeln, die mit Hilfe einer Pipe als Eingabe von `sort` umgeleitet wird, um den Inhalt von einer oder mehreren Dateien oder um direkte Eingaben eines Benutzers (diese müssen dann mit `[strg+d]` (= EOF, *end of file*) abgeschlossen werden).

Syntax:

`sort [`*Optionen*`]` *Datei1 Datei2 ...*

Beispiel:

```
~$  grep die /tmp/*.txt | sort -f
```

Listet alphabetisch geordnet alle Zeilen aus Dateien mit dem Suffix `.txt` in dem Verzeichnis `/tmp` auf, welche das Wort »die« enthalten. Durch den Parameter `-f` wird dabei die Groß- und Kleinschreibung ignoriert (sonst würden zunächst alle Zeilen, die mit einem Großbuchstaben beginnen, aufgelistet werden).

Durch die Übergabe des Parameters `-c` kann man mit Hilfe von `sort` überprüfen, ob eine Eingabe bereits sortiert ist:

```
~$ sort -c /tmp/text.txt
```

2.2.12 Ein- und Aushängen von peripheren Geräten

Zum Einhängen (*Mounten*) eines peripheren Gerätes, wie eines CD-ROM- oder Floppy-Laufwerks (oder eines anderen Gerätes, beispielsweise eines USB-Sticks) ins Dateisystem gibt es das Kommando `mount`. Der »normale« Benutzer kann das betreffende Gerät allerdings nur einmounten, wenn dieses bereits in der `/etc/fstab` eingetragen ist. Hier können auch die Einstellungen und der Pfad für das Gerät eingesehen werden.

Nach dem *Mounten* wird das Gerät wie ein Verzeichnis behandelt und ist ebenso anzusprechen. Es ist zu beachten, dass Datenträger, die eingehängt wurden (beispielsweise eine Diskette), nicht entfernt werden sollten, solange sie nicht wieder ausgehängt sind. Das Aushängen (*Unmounten*) erfolgt mit dem Kommando `umount`. Erst wenn der Datenträger ausgehängt wurde, ist garantiert, dass alle Daten, die auf ihm gespeichert werden sollen, wirklich geschrieben werden. Wenn man beispielsweise mit `cp` eine Datei auf eine Diskette speichert, wird diese vorerst im Hauptspeicher gehalten. Wann sie auf die Diskette geschrieben wird, ist Sache des Betriebssystems, erfolgt aber spätestens bevor das Kommando `umount` ausgeführt wird.

Syntax:

`mount` *Gerätename*

Beispiel:

```
~$ mount /floppy
```

Ein Laufwerk kann und sollte nur ausgehängt werden, wenn es nicht in Benutzung ist. Um dies zu testen gibt es das Kommando `fuser`. `fuser` zeigt an, ob eine Datei (im weitesten Sinne) oder ein Socket gerade angesprochen wird. Übergibt man den Parameter `-v`, so wird außerdem angegeben, von wem eine Datei bzw. ein Port benutzt wird und welches Kommando gerade ausgeführt wird. Zu beachten ist, dass das Aushängen jeweils nur durch den Benutzer erfolgen kann, der das Laufwerk eingehängt hat (oder durch *root*).

Syntax:

`fuser [-v]` *Datei-(i.w.s.)/Portname*

Beispiel:

```
~$ fuser -v /floppy
```

Wenn das Dateisystem auf dem Gerät nicht automatisch erkannt wird, müssen die `FAT32`-kompatiblen (MS-DOS-Dateisystem) Kommandos `mcopy` für Binärdateien (bzw. `mcopy -t` für Textdateien) verwendet werden.

Einige FAT32 kompatible Kommandos:

`mdir`	listet den Inhalt der Diskette auf
`mdir` *Verzeichnis*	listet den Inhalt der angegebenen Verzeichnisse auf
`mcd` *Verzeichnis*	wechselt in das angegebene Verzeichnis auf der Diskette
`mcopy` *Quelldatei Zieldatei*	kopiert die angegebene Quelldatei nach Zieldatei
`mmd` *Verzeichnis*	erzeugt das angegebene Verzeichnis
`mrd` *Verzeichnis*	entfernt die angegebenen (leeren) Verzeichnisse
`mren` *Quelldatei Zieldatei*	benennt Datei *Quelldatei* in *Zieldatei* um
`mren` *Dateiname Verzeichnis*	bewegt die angegebenen Dateien in das angegebene Verzeichnis
`mdel` *Datei*	löscht die angegebenen Dateien
`mtype` *Datei*	zeigt den Inhalt der angegebenen Datei an

2.3 Eingabehilfen in der Shell

Im Folgenden wird auf einige Hilfen eingegangen, welche die Shell bei der Eingabe von Kommandos bietet. Dazu gehört das komfortable Durchsuchen bereits eingegebener Kommandos in der Kommandohistorie und die automatische Vervollständigung (Komplettierung) von Kommandos und Pfaden.

2.3.1 Komplettierung von Kommandos und Pfaden

Die `bash` bietet die Möglichkeit, Kommandos oder Pfade mit Hilfe der `[TAB]`-Taste zu komplettieren. D. h. beispielsweise bei Eingabe von `/home/us [TAB]` ergänzt die Shell dies auf `/home/user/`, sofern der Pfad `/home/user/` existiert und es keine weiteren Verzeichnisse gibt, die mit `/home/us` gemeint sein könnten (z. B. `/home/uswusf/`). Erfolgt beim Drücken von `[TAB]` keine Komplettierung, ist der Name nicht eindeutig fortzuführen. Durch zweimaliges Drücken von `[TAB]` erhält man alle Komplettierungen, die zum bereits Eingegebenen möglich sind.

2.3.2 Die Kommandohistorie (*command history*)

Die eingegebenen Kommandos werden von der Shell in der Historie (*history*) gespeichert. Das heisst, alle eingegebenen und mit `[Enter]` an die Shell abgeschickten Kommandos werden in eine Datei geschrieben (meist `.sh_history`, `.bash_history` im Falle der `bash`). Die History kann mit Hilfe der Pfeiltasten vorwärts und rückwärts durchlaufen werden. Mit dem ↑ erhält man das zuletzt eingegebene Kommando.

Durch die Eingabe von `[strg+r]` (bzw. `[ctrl+r]` auf US-Tastaturen) kann die History auch rückwärts nach Eingabemustern durchsucht werden (*reverse search*). Auf Drücken von `[strg+r]` erscheint in der Shell die Zeile:

```
(reverse-i-search)'':
```

In den Hochkommata erscheint das vom Nutzer eingegebene Suchmuster, hinter dem Doppelpunkt das letzte Kommando, auf welches das Suchmuster passt. Dabei kann das Suchmuster irgendwo in der Zeile des Kommandos auftauchen.

Beispiel:

```
(reverse-i-search)'xe': xeyes
```

Zur Vereinfachung der Eingabe bietet die `bash` weitere nützliche Tastenkürzel:

`[strg+a]` – An den Anfang der Zeile springen

`[strg+e]` – Ans Ende der Zeile springen

`[strg+b]` – Wort nach rechts springen

`[strg+f]` – Wort nach links springen

`[strg+k]` – Zeile löschen

`[strg+t]` – Zeichen tauschen

`[strg+l]` – Bildschirm löschen (gleich der Eingabe von `clear`)

`[strg+r]` – Rückwärtssuche in der History

2.4 Prozessverwaltung

Um die einzelnen Prozesse zu verwalten, legt das Betriebssystem für jeden Prozess einen Kontrollblock an. Dieser wird eindeutig über die Prozess-ID (*pid – process identification*) identifiziert. Die Shell bietet verschiedene Kommandos, mit denen der Benutzer sich Prozesse anzeigen lassen und in die Prozessverwaltung eingreifen kann. Der Benutzer kann dabei nur seine eigenen Prozesse verwalten.

2.4.1 Vordergrundprozesse

Ein Prozess läuft so lange, bis er seine Aufgabe erfüllt hat oder bis er vom Benutzer beendet wird. Beim Aufruf eines Kommandos bzw. Programms, wie wir es bisher gemacht haben, wird ein sogenannter Vordergrundprozess erzeugt. Dieser blockiert die Shell, solange er läuft. Mit der Tastatureingabe `[strg+c]` (bzw. `[ctrl+c]`) können laufende Vordergrundprozesse beendet werden.

Im Beispiel wird das Programm `xeyes` gestartet und mit `[strg+c]` beendet. Der Benutzer erhält dann wieder ein Eingabeprompt und kann wieder Kommandos eingeben.

Beispiel:

```
~$ xeyes [Eingabe von strg+c]
~$
```

2.4.2 Hintergrundprozesse

Mit dem `&`-Operator kann man Prozesse als Hintergrundprozesse starten. D. h. man kehrt nach dem Aufruf des Programms direkt auf die Shell zurück, wobei der Prozess weiterläuft. Dies bietet sich vor allem für Programme an, die länger laufen, wie beispielsweise ein Browser.

Auf der Shell wird dem Benutzer beim Aufruf eines Hintergrundprozesses eine fortlaufende Nummer in eckigen Klammern (Anzahl der von ihm in dieser Shell gestarteten Prozesse) und die Prozess-ID *pid* angezeigt.

Beispiel:

```
~$ mozilla &
[1] 13430
~$
```

Im Beispiel wird der Aufruf des Browsers `mozilla` gezeigt. Die Shell zeigt an, dass der Prozess »mozilla« der erste Hintergrundprozess ist, der von dem Benutzer hier gestartet wurde und dass ihm die Prozess-ID `13430` zugewiesen wurde. Anschließend erhält der Benutzer wieder ein Prompt und kann weitere Kommandos eingeben.

Prozesse können aber auch während sie bereits laufen in den Hintergrund geschickt werden. Dazu wird der Prozess zunächst mit der Tastatureingabe `[strg+z]` (`[ctrl+z]`) gestoppt (in Wartezustand versetzt), gibt eine entsprechende Meldung auf der Shell aus und kann dann mit dem Kommando `bg` (*background process*) in den Hintergrund geschickt werden.

Beispiel:

```
~$ mozilla [Eingabe von strg+z]
[2]  Stopped                 mozilla
~$ bg
[2] 13433
~$
```

Ein Hintergrundprozess kann mit dem Kommando `fg` (*foreground process*) ebenso in den Vordergrund geholt werden. Dabei wird immer der zuletzt gestartete Hintergrundprozess (der mit der höchsten Nummer in eckigen Klammern) in den Vordergrund geholt.

Beispiel:

```
~$ xeyes &

[3] 13438

~$ fg

xeyes
```

Beim Aufruf von `bg` und `fg` kann auch der Name eines bestimmten Jobs übergeben werden. Es wird dann der letzte Job mit dem übergebenen Namen in den Vordergrund bzw. den Hintergrund geschoben.

Beispiel:

```
~$ fg xeyes

xeyes
```

Für `bg` macht das nur Sinn, wenn es mehrere gestoppte Jobs gibt, von denen ein anderer als der letzte in den Hintergrund geschickt werden soll.

2.4.3 Prozesse beenden

Laufende Prozesse (die nicht im Hintergrund sind) können, wie bereits gesagt, durch die Tastatureingabe `[strg+c]` abgebrochen werden. Um Hintergrundprozesse zu beenden bietet die Shell das Kommando `kill`. Mit dem Aufruf ›`kill` *pid*‹ kann der Benutzer den Prozess mit der Prozess-ID *pid* »töten«. Jeder Benutzer (außer *root*) kann dabei nur Prozesse beenden, die ihm selbst gehören.

Syntax:

`kill [`*Signal*`]` *pid*

Beispiel:

```
~$ xeyes &

[4] 13455

~$ kill 13455

~$    [Eingabe von Enter]

[4]-  Killed                    xeyes
```

Im Beispiel wird das Programm `xeyes` mit der Prozess-ID `13455` beendet. Nach der nächsten Eingabe erscheint auf der Shell die entsprechende Nachricht.

Das Kommando `kill` kann mit verschiedenen Signalen aufgerufen werden, die der Shell anzeigen, wie ein Prozess beendet werden soll. Der Standardwert ist `-15`. Mit diesem Signal ist es dem Programm selbst überlassen, wie es sich beendet. Mit dem Signal `-9` werden Prozesse sofort und auf jeden Fall durch den Kernel beendet (einige Prozesse lassen sich durch einfachen Aufruf von `kill` nicht beenden).

2.4.4 Prozesse anzeigen

Mit dem Kommando `ps` (*report process status*) können die aktuellen Prozesse, sowie ihr Status und einige weitere Informationen aus der Prozesskontrolle angezeigt werden.

Beispiel:

```
~$ ps
  PID TTY          TIME CMD
 8670 pts/0    00:00:00 bash
 9314 pts/0    00:00:00 ps
```

Spalten:

`PID` ist die Prozess-ID.

`TTY` *tele type* gibt an, von welchem »Terminal« der Prozess gestartet wurde.

`TIME` ist die Zeit, die der Prozess bereits läuft.

`CMD` ist der Name des Programms.

Durch die Übergabe von Parametern kann man mit Hilfe von `ps` weitere Informationen über die laufenden Prozesse ausgeben lassen, beispielsweise den Status (`STAT`). Im Wesentlichen kommen zwei Zustände vor: `S` (*sleeping*) und `R` (*running*). Ein Prozess, dessen Status mit `S` angegeben ist, »schläft«, d. h. er pausiert gerade und verbraucht keine Prozessorleistung. Ein Prozess mit Status `R` ist gerade aktiv.

Mögliche Zustände eines Prozesses:

```
PROCESS STATE CODES
     D   uninterruptible sleep (usually IO)
     R   running
     S   sleeping
     T   traced or stopped
     Z   a defunct ("zombie") process
```

Das Kommando `jobs` zeigt nur Hintergrundprozesse an, die in der aktuellen Shell laufen. Es wird die laufende Nummer, der Status und der Name des Prozesses ausgegeben.

Syntax:

`jobs[`*Optionen*`]`

Beispiel:

```
~$ jobs -l
[1]+ 14227 Running                 xeyes &
```

Durch die Übergabe des Parameters `-l` erfolgt die Ausgabe im langen Format (*long*).

```
~$ jobs -p
14227
```

Wird der Parameter `-p` übergeben, erfolgt nur die Ausgabe der Prozess-IDs.

2.4.5 Ressourcen anzeigen

Das Programm `top` zeigt die laufenden Prozesse nach den von ihnen in Anspruch genommenen Kapazitäten an. Außerdem werden die auf dem Rechner verfügbaren Kapazitäten angezeigt.

Beispiel:

```
~$ top
top - 17:11:26 up 14 days,  3:39,  1 user,  load average: 0.00, 0.00, 0.00
Tasks:  72 total,   2 running,  70 sleeping,   0 stopped,   0 zombie
Cpu(s):   7.9% user,   1.7% system,   0.0% nice,  90.4% idle
Mem:    515252k total,   506040k used,     9212k free,    25180k buffers
Swap:   979924k total,       40k used,   979884k free,   216120k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  583 root      12 -10 73236  28m 2456 R  5.3  5.8  16:04.78 XFree86
 8669 tech      14   0 55740  15m  11m R  4.0  3.0   0:04.63 kdeinit
 9313 tech      11   0   948  948  748 R  0.3  0.2   0:00.07 top
    1 root       8   0   464  464  408 S  0.0  0.1   0:03.85 init
    2 root       9   0     0    0    0 S  0.0  0.0   0:01.13 keventd
    3 root      19  19     0    0    0 S  0.0  0.0   0:00.10 ksoftirqd_CPU0
    4 root       9   0     0    0    0 S  0.0  0.0   0:00.06 kswapd
    5 root       9   0     0    0    0 S  0.0  0.0   0:00.00 bdflush
    6 root       9   0     0    0    0 S  0.0  0.0   0:00.13 kupdated
    7 root       9   0     0    0    0 S  0.0  0.0   0:00.00 khubd
  122 daemon     9   0   492  492  416 S  0.0  0.1   0:00.00 portmap
  213 root       9   0   772  772  648 S  0.0  0.1   0:00.12 syslogd
  216 root       9   0   524  524  372 S  0.0  0.1   0:00.00 klogd
  232 root       8   0  1760 1760 1284 S  0.0  0.3   0:00.02 cupsd
  254 root       9   0   688  688  616 S  0.0  0.1   0:00.00 inetd
```

`PID` – Prozess-ID

`USER` – Besitzer des Prozesses

`PR` – Priorität des Prozesses

`NI` – *nice*-Wert des Prozesses (Veränderung der Priorität)

`S` – Status des Prozesses

`%CPU` – Anteil der `CPU`-Zeit (in %), die der Prozess in Anspruch nimmt.

`%MEM` – Anteil des Arbeitsspeichers (in%), den der Prozess in Anspruch nimmt.

`TIME+` – bisherige Laufzeit

`COMMAND` – Name des Kommandos/Programms

Die Werte für CPU-Auslastung, Speicherverbrauch usw. werden über die letzten 3.0 Sek gemittelt. Neben den im Beispiel gezeigten Werten können weitere Informationen zu Prozessen und Ressourcen angegeben werden. Das Anpassen der Ausgabe von `top` und die Bedeutung der Werte ist in der `man`-page beschrieben.

2.4.6 Prozesspriorität und Setzen/Verändern des `nice`-Wertes

Wie eingangs erwähnt, laufen Prozesse beim Multitasking meist nicht wirklich gleichzeitig ab, sondern werden mehrere $100\times$ in jeder Sekunde unterbrochen und weitergeführt, wenn sie die Zeitscheibe zurück erhalten. Dieser Vorgang wird als Zeitscheiben-Verfahren (*time slicing*) oder *Round-Robin-Verfahren* bezeichnet (Größenordnung etwa 10 mSek).

Sieht man sich die laufenden Prozesse mit dem Kommando `top` an, so erkennt man, dass unterschiedliche Prozesse unterschiedliche Prioritäten aufweisen. Diese sind entscheidend dafür, welcher Prozess als nächstes die Zeitscheibe erhält. Die Priorität eines Prozesses wird jeweils dynamisch aus mehreren Faktoren unter anderem dem `nice`-Wert berechnet.[15] Der Prozess mit der jeweils höchsten Priorität erhält die Zeitscheibe als nächstes.

Der `nice`-Wert auf den meisten Unix-Derivaten liegt im Intervall $[-20, 20]$. Je negativer der Wert ist, desto höher ist die berechnete Priorität, was dazu führt, dass der zugehörige Prozess häufiger die Zeitscheibe erhält, also schneller abläuft.

Standardmäßig erhalten Prozesse den `nice`-Wert 0. Will ein Benutzer den Wert eines seiner Prozesse anpassen, so kann er diesen mit dem Kommando `nice` beim Aufruf setzen oder mit `renice` den Wert eines laufenden Prozesses verändern. Benutzer dürfen den Wert jedoch nur nach oben verändern (positiver machen), also ihre Prozesse verlangsamen. *root* dagegen darf die Werte beliebig ändern. Wird ein Prozess mit `nice` aufgerufen ohne einen Wert zu übergeben, so wird der Prozess automatisch mit dem `nice`-Wert 10 gestartet.

1. **Setzen eines `nice`-Wertes**

   Syntax:

   `nice [-n` *Wert*`]` *Kommando* `[`*Parameter*`]`

   Beispiel:

   ```
   ~$ nice -n 15 top
   ```

2. **Verändern des `nice`-Wertes eines laufenden Prozesses**

   Syntax:

   `renice [`*Wert*`] -p` *pid*

   Beispiel:

   ```
   ~$ renice 10 -p 13767
   ```

[15] Weitere Faktoren bei der Vergabe der Zeitscheibe sind: wieviel CPU-Zeit der Prozess zuletzt in Anspruch genommen hat, wieviele andere Prozesse laufen und welche Prioritäten sie haben, usw.

2.4.7 Prozesse abgekoppelt von der Shell starten

Um Prozesse von der Shell abgekoppelt zu starten, gibt es das Kommando `nohup`. Dieses bewirkt, dass das gestartete Programm vor dem `HUP`-Signal (*hangup*) der Shell geschützt wird. D. h. der Prozess bleibt am Leben, auch wenn der Benutzer sich ausloggt. Alle Ausgaben eines Programms, das mit `nohup` gestartet wurde, werden in Dateien umgeleitet (z. B. `nohup.out`).

Syntax:

`nohup [`*Kommando*`] [`*Optionen des Kommandos*`]`

Beispiel:

```
~$ nohup bastall -p blastn -d nt -i ecoli.fna &
```

Der Prozess wird dabei nicht automatisch als Hintergrundprozess gestartet, da so bei interaktiven Prozessen noch Eingaben des Benutzers möglich sind. Anschließend kann der Prozess mit `[strg+z]` und `bg` in den Hintergrund geschickt werden, bevor die Shell beendet wird. Um einen Prozess gleich als Hintergrundprozess zu starten, muss er explizit mit `&` aufgerufen werden.

Es können natürlich nur Prozesse abgekoppelt gestartet werden, die kein Terminal brauchen. Für das Kommando `top` oder einen Editor beispielsweise funktioniert das so nicht. Solche Programme können nur in einem *virtuellen Terminal* gestartet werden, wenn sie auch nach dem Ausloggen weiter laufen sollen. Dazu gibt es das Kommando `screen`, dieses ist allerdings nicht auf jedem System verfügbar.

2.5 Verschlüsselte Datenübertragung

Über Netzwerke können Benutzer auf Rechnern an anderen Orten auf der Welt arbeiten. Dabei sollte man allerdings nicht vergessen, dass die Daten dabei durch Leitungen übertragen werden (z. B. Telefonleitungen) und daher abgehört werden können (davon sind auch eMails betroffen). Sicherheitsrelevante Daten wie Passwörter sollten aus diesem Grund nicht unverschlüsselt übertragen werden.

Programme wie `telnet`, `rsh` (*remote shell*) und `rcp` (*remote copy*) tun genau dies und sollten daher nicht verwendet werden. Die Programme `ssh` (*secure shell*) und `scp` (*secure copy*) verschlüsseln dagegen die Daten auf dem lokalen Rechner und schicken sie dann über das Netz. Auf dem Zielrechner (entfernter Rechner – *remote host*) werden die Daten dann wieder entschlüsselt.

2.5.1 Secure-Copy

Das Kommando `scp` verhält sich im Grunde wie `cp` mit dem Unterschied, dass bei `scp` vor dem Dateinamen ein Rechnername angegeben werden kann. Ist der Rechner dieses Namens[16] über das Netzwerk zugängig, so erfolgt der Login (ggf. mit Passwortabfrage) und die Daten werden verschlüsselt übertragen. Ist der Loginname auf dem Zielrechner ein anderer als der lokale Benutzername, so muss er vor dem Rechnernamen, getrennt durch ein `@` angegeben werden.

Syntax:

`scp [[`*Benutzername*`@]`*Rechnername*`:]`*Pfad* `[[`*Benutzername*`@]`*Rechnername*`:]`*Pfad*

[16] Der Name eines Rechners im Netzwerk ist eine IP-Adresse (*ip – internet protocol*). Diese kann unter Umständen durch einen Domänennamen ersetzt werden.

Beispiele:

`scp user@141.75.70.120:`*Verzeichnis/Datei* `.`

`scp` *Verzeichnis/*`*` `user@141.75.70.120:.`

Im ersten Beispiel wird durch den Benutzer, der auf dem entfernten Rechner (`141.75.70.120`) den Loginnamen *user* hat, eine Datei ins lokale Arbeitsverzeichnis kopiert. Die Datei (*Datei*) liegt auf dem entfernten Rechner im Heimatverzeichnis von `user` unter dem Verzeichnis *Verzeichnis*.

Im zweiten Beispiel werden alle Dateien in dem lokalen Verzeichnis *Verzeichnis* in das Heimatverzeichnis von *user* auf dem Rechner `141.75.70.120` kopiert.

2.5.2 Secure-Shell

Das Programm `ssh` versucht sich auf dem Zielrechner einzuloggen und falls ein Kommando beim Aufruf übergeben wurde, dieses auszuführen. Z. T. verlangt der Zielrechner beim Login die Eingabe eines Passwortes.

Syntax:

`ssh [-X] [`*Benutzername*`@]`*Rechnername* `[`*Kommando*`]`

`ssh [-X]` *Rechnername* `[-l` *Benutzername*`] [`*Kommando*`]`

Wird kein Loginname angegeben, so wird der Name des lokalen Benutzers beim Login verwendet. Wenn kein Kommando übergeben wird, öffnet sich eine Shell auf dem Zielrechner, die in der lokalen Shell angezeigt wird. In ihr kann der eingeloggte Benutzer auf dem entfernten Rechner arbeiten wie auf einem lokalen. Dabei ist zu beachten, dass auf dem Zielrechner evtl. ein anderes System (anderes Betriebssystem, andere Shell, ...) arbeitet, das unter Umständen andere Kommandos und Umgebungseinstellungen (Umgebungsvariablen, Pfade, usw.) verwendet. Weiterhin ist zu beachten, dass die Eingabe beim Aufruf von `ssh` von der lokalen Shell interpretiert wird. Dies gilt insbesondere für Ein-/Ausgabeumleitung und Pipelines. Um zu verhindern, dass die lokale Shell Kommandos interpretiert, muss der Aufruf, der auf dem entfernten Rechner ausgeführt werden soll, in Hochkommata gesetzt werden.

Der Parameter `-X` bewirkt, dass die `DISPLAY`-Umgebungsvariable gesetzt wird, so dass die Anzeige von `X11`-Programmen automatisch auf den lokalen Rechner übertragen wird. Genauer gesagt wird die `X11`-Ausgabe umgeleitet, was heisst, dass der entfernte `X11`-Server die Fenster von `X11`-Programmen auf den lokalen Rechner weiterleitet (*forwarded*). Auf diesem werden sie vom lokalen X-Client angezeigt.

Beispiel:

`ssh 141.75.70.120 ls | grep` *wort*

Das Kommando `ls` wird auf dem Zielrechner ausgeführt, `grep` dagegen wird auf dem lokalen Rechner ausgeführt. Um dies zu vermeiden, kann der komplette Aufruf, der auf dem Zielrechner auszuführenden Kommandos, in Hochkommata (`' '`) gesetzt werden.

`ssh 141.75.70.120 'ls | grep` *wort*`'`

Im folgenden Beispiel wird kein Kommando übergeben. Der Benutzer `user` erhält eine Login-Shell.

```
ssh 141.75.70.120 -l user
```

analog zu:

```
ssh user@141.75.70.120
```

Der Name `user` muss, wie gesagt, nur mitübergeben werden, wenn der Benutzer lokal einen anderen Login-Namen hat als auf dem entfernten Rechner.

2.5.3 Schlüsselpaare zur Authentifizierung generieren

Um sich auf einem Rechner via `ssh` einloggen zu können, ohne jedes Mal ein Passwort eingeben zu müssen, kann man sich Paare von Authentifizierungsschlüsseln generieren. Unter Linux kann man dies mit dem OpenSSH-Programm `ssh-keygen` tun.

`ssh-keygen` generiert assymmetrische Schlüsselpaare, d. h. es gibt einen öffentlichen und einen privaten Schlüssel. Der öffentliche Schlüssel muss auf dem Zielrechner liegen, der private auf dem lokalen. Beim Login auf dem Zielrechner wird der öffentlich Schlüssel dann mit dem privaten abgeglichen. Passen beide Schlüssel zusammen, so erfolgt der Login. Andernfalls wird der Benutzer abgewiesen.

`ssh-keygen` bietet zwei Arten von Verschlüsselungsalgorithmen an: Das ältere Verfahren RSA, benannt nach den Entwicklern Rival, Shamir, Adelman und DSA (*Digital Signature Algorithm*) entwickelt vom us-amerikanischen National Institute of Standards and Technology (NIST). DSA-Schlüssel können dabei nur mit `ssh` Version 2 (inzwischen Standard auf den meisten Rechnern) verwendet werden, RSA auch mit Version 1.

Syntax:

`ssh-keygen -t` *type* `[-f` *key_file*`]`

Die Verschüsselung wird mit `-t dsa` bzw. `-t rsa` angegeben. Mit dem Parameter wird `-f` der Name der Dateien festgelegt, in welche die Schlüssel geschrieben werden sollen. Standardmäßig legt `ssh-keygen` die Schlüssel unter `/home/`*user*`/.ssh/` ab. Der private Schlüssel wird in die Datei `id_dsa` (falls es sich um einen DSA-Schlüssel handelt) geschrieben, sofern keine anderer Dateiname spezifiziert ist. Der öffentliche Schlüssel erhält jeweils den gleichen Namen wie der private, mit dem Suffix `.pub`.
Nach dem Erzeugen muss der öffentliche Schlüssel auf dem Zielrechner unter `/home/`*user*`/.ssh/` in die Datei `authorized_keys` (auf manchen Systemen `authorized_keys2`) abgelegt werden. Von nun an kann sich der Benutzer *user* auf dem Zielrechner ohne Eingabe eines Passwortes einloggen, so fern der private Schlüssel auf dem lokalen Rechner liegt. Ist der private Schlüssel in der Standard-Datei abgelegt, so braucht der Benutzer keine zusätzlichen Angaben beim Login zu machen. Ist dies nicht der Fall, so muss er beim Aufruf von `ssh` mit dem Parameter `-i` den Namen der Datei übergeben, in welcher der private Schlüssel zu finden ist.

Beispiel:

```
~$ ssh-keygen -t dsa
Enter file in which to save the key (/home/tech/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_dsa.
```

```
Your public key has been saved in /home/user/.ssh/id_dsa.pub.
The key fingerprint is:
24:d5:a8:1d:a7:18:fd:d1:4b:2c:c3:fe:55:3f:0d:86 user@eisbaer
```

Im Beispiel wird ein Schlüsselpaar mit DSA für den Benutzer `user` auf dem Rechner `eisbaer` erzeugt. Nach dem Aufruf von `ssh-keygen` hat der Benutzer die Möglichkeit einen anderen Dateinamen anzugeben. Es folgt die Aufforderung ein Passwort einzugeben. Diese muss leer gelassen werden, wenn man ein Schlüsselpaar zum Einloggen generiert (sogenannte *host keys*).

Das Programm gibt dann den Fingerabdruck (*fingerprint*) des Schlüssel aus und wo die Schlüssel gespeichert wurden.

3 Editoren

Im Folgenden werden die beiden wichtigsten Editoren unter Linux beschrieben, der Emacs und der Vi. Beide Editoren sind sehr weit verbreitet und auf nahezu jedem System der Unix-Familie verfügbar. Es gibt aber auch Versionen für Windows und MacOs. Die folgenden Beschreibungen beziehen sich nur auf die Linux-Versionen, wobei allerdings vieles auch auf die anderen Versionen übertragbar ist. Die Wahl des Editors ist vor allem Geschmackssache. Andere Editoren, die häufig unter Linux verwendet werden, sind: `pico`, `nano`, `joe`, usw. Diese sind aber zum einen nicht auf jedem System verfügbar und bieten bei weitem nicht die Funktionalitäten wie der Emacs und der Vi. Ein Vorteil insbesondere für das Arbeiten auf entfernten Rechnern ist neben der weiten Verbreitung, dass beide Editoren sowohl im Terminal der Shell als auch mit graphischer Schnittstelle (GUI – *graphical user interface*) benutzt werden können. Weitere Vorzüge werden in den folgenden Abschnitten beschrieben.

3.1 Der Emacs

Der Emacs (Akronym von **E**ditor **macros**) ist im Rahmen eines GNU-Projektes entstanden und ist in einem speziellen *Lisp*-Dialekt geschrieben (*eLisp*). Ursprünglich basierte er auf einer Sammlung von Makros des Editors TECO, einem der Urgesteine unter den Editoren. Wie alle GNU-Projekte ist er ein *open source*-Projekt, also frei verfüg- und veränderbar. Er bietet neben vielen anderen Funktionalitäten, vor allem solche, die das Programmieren sehr erleichtern können. Dazu gehören das *Highlighten* von Syntax, automatisches Einrücken (*indent*), spezielle Modi für Programmiersprachen, das Kompilieren und Ausführen in Puffern (so dass man nicht immer zur Shell wechseln muss) und viele mächtige Funktionen zur Textverarbeitung und Datenverwaltung. Außerdem ist der Emacs komplett über (zugegebenermaßen etwas gewöhnungsbedürftige) Tastenkombinationen zu bedienen, von denen hier nur einige vorgestellt werden. Das ist zum einen nützlich, wenn man viel schreibt, da man nicht ständig von der Tastatur zur Maus greifen muss, zum anderen ist es natürlich sehr nützlich, wenn man den Emacs ohne GUI benutzen möchte oder muss. Darüber hinaus kann der Emacs als WWW-Browser, als Email-Client oder zum Lesen von Newsgroups verwendet werden. Die Bedienung dieser Funktionen kann im unten angegebenen Emacs-Manual nachgelesen werden. Hier wird nur auf die Funktionalitäten als Editor eingegangen.

GNU – emacs manual: http://www.gnu.org/software/emacs/manual/.

3.1.1 Frames, Windows und Buffer

Die Namensgebung im Emacs ist aus historischen Gründen vielleicht etwas ungewohnt. Öffnet man den Emacs im eigenen »Fenster« (also nicht in der Konsole), so wird dieses als Rahmen

(*frame*) bezeichnet. In dem Rahmen befindet sich ein Fenster (*window*) und in dem Fenster wird ein Puffer (*buffer*) angezeigt. Eine Datei, die mit dem Emacs angezeigt wird, ist im Puffer geladen und bleibt es über die gesamte (Emacs-)Session, sofern der Benutzer den Puffer nicht »tötet«. Dasselbe gilt auch für Prozesse und Verzeichnisse. Unter dem Menüpunkt »Buffers« kann man sich alle Puffer anzeigen lassen und direkt zu ihnen wechseln (Tastenkommando: `c^x c^b`).

Verändert ein Benutzer eine Datei, so legt der Emacs eine Kopie im gleichen Verzeichnis an, die bis zum Abspeichern (durch den Benutzer) durch Gatterkreuze (`#`) vor und hinter dem Dateinamen gekennzeichnet ist. Speichert der Benutzer seine Änderungen auf der Festplatte, so verschwindet die Zwischenkopie. Beim Abspeichern legt der Emacs außerdem eine Sicherheitskopie an. Diese enthält den Inhalt des Puffers vor der letzten größeren abgespeicherten Veränderung (also eine ältere Version der Datei). Die Sicherheitskopie liegt ebenfalls im gleichen Verzeichnis wie die Originaldatei und ist durch eine Tilde gekennzeichnet. Am unteren Ende eines Windows befindet sich noch ein zusätzliches einzeiliges Window: der *Minibuffer*. Hier werden Kommandos, die gerade ausgeführt werden angezeigt. Weiterhin dient der Minibuffer zur Eingabe von Argumenten, sofern ein Kommando solche erwartet.

3.1.2 Starten des Emacs

Das Starten erfolgt einfach durch Eingabe des Kommandos `emacs` in der Shell. Standardmäßig wird der Emacs im eigenen Fenster geöffnet, sofern dies möglich ist. Möchte man ihn in der Kommandozeile ausühren, so muss man dies beim Start durch den Parameter `-nw` angeben.

Beispiel:

`emacs -nw` *Datei*

Wird beim Starten keine Datei übergeben, so öffnet der Emacs zunächst einen Puffer in den man sich Notizen machen kann, die man nicht sichern möchte. Dieser Puffer kann auch als *Lisp*-Interpreter fungieren, d. h. man kann hier Kommandos und Funktionen der Sprache *Lisp* ausführen. Im eigenen Fenster verfügt der Emacs standardmäßig über eine Menüleiste (die man natürlich »wegkonfigurieren« kann) mit deren Hilfe man Dateien öffnen, schließen, speichern, neue Fenster öffnen usw. kann. Wie bereits gesagt, kann man dies alles auch mit Hilfe von Tastenkommandos tun. Einige dieser Tastenkombinationen werden im Folgenden noch vorgestellt.

3.1.3 Konfiguration des Emacs

Der Emacs kann über die Datei `.emacs` im Heimatverzeichnis des Benutzers individuell konfiguriert und erweitert werden. Hier können neue Funktionalitäten wie beispielsweise `HIPPIE-EXPAND` (siehe Beispiel) oder neue Tastenkommandos eingefügt werden. Zeilen, die mit `;` beginnen, sind Kommentarzeilen.

Beispiel einer `.emacs`-Datei

```
;; Are we running XEmacs or Emacs?
(defvar running-xemacs (string-match "XEmacs\\|Lucid" emacs-version))


;; Set up the keyboard so the delete key on both the regular keyboard
;; and the keypad delete the character under the cursor and to the right
;; under X, instead of the default, backspace behavior.
(global-set-key [delete] 'delete-char)
```

```
(global-set-key [kp-delete] 'delete-char)


;; Enable wheelmouse support by default
(require 'mwheel)


;; Highlight marked region
(transient-mark-mode 1)


;; HIPPIE-EXPAND
(global-set-key [(shift return)] 'hippie-expand)
(setq hippie-expand-try-functions-list
      '(try-expand-dabbrev
        try-expand-dabbrev-all-buffers
        try-complete-file-name-partially
        try-complete-file-name))


;; Turn on font-lock mode for Emacs
(cond ((not running-xemacs)
(global-font-lock-mode t)
))


;; Always end a file with a newline
(setq require-final-newline t)


(custom-set-variables
  ;; custom-set-variables was added by Custom -- don't edit or cut/paste it!
  ;; Your init file should contain only one such instance.
 '(case-fold-search t)
 '(current-language-environment "ASCII")
 '(global-font-lock-mode t nil (font-lock)))
(custom-set-faces
  ;; custom-set-faces was added by Custom -- don't edit or cut/paste it!
  ;; Your init file should contain only one such instance.
 )
```

3.1.4 Einige Emacs-Kommandos

Im Folgenden werden einige Emacs Kommandos angegeben. Wie bereits erwähnt, können alle diese Kommandos auch über das Menü aufgerufen werden. Das Kürzel `c^` steht dabei jeweils für »Gedrückthalten« der `[strg]`- (`[ctrl]`-) Taste, während das folgende Symbol eingegeben wird. `c^x` steht also für das gleichzeitige Drücken von `[ctrl]` und `x`. `c x` dagegen steht für nacheinander Drücken von `[ctrl]` und `x`. `M` bezeichnet die Meta-Taste. Je nach Konfiguration ist dies meist `[Alt]` oder `[ESC]`[17]. Ebenso wie bei `[ctrl]` wird mit dem Dach (^) angegeben, dass die Taste gehalten werden soll, durch ein Leerzeichen, dass beide Eingaben nacheinander erfolgen sollen.

[17] Auf SUN-Tastaturen die Taste `[Meta]`.

Hilfe		
c^h	Hilfe	(help-command)
Dateien und Puffer (Buffer)		
c^x c^s	Speichern des aktuellen Puffers	(save-buffer)
c^x s	Speichern von allen geänderten Puffern	(save-some-buffers)
c^x c^w	Schreiben des Puffers in eine Datei	(write-file)
c^x c^f	Suchen (und Öffnen) einer Datei	(find-file)
c^x c^b	Auflisten aller Puffer	(list-buffers)
c^x k	»Töten« des aktuellen Puffers	(kill-buffer)
c^x b	Wechseln zu Puffer	(switch-to-buffer)
Fenster (Windows) und Rahmen (Frames)		
c^z	Emacs minimieren	(suspend-emacs (minimieren))
c^x c^c	alle Puffer speichern und beenden	(save-buffers-kill-emacs)
TAB	Einrücken einer Zeile	(indent-for-tab-command)
c^x 2	Fenster horizontal splitten	(split-window-horizontally)
c^x 3	Fenster vertikal splitten	(split-window-vertically)
c^x o	ins nächste Fenster wechseln	(other-window)
c^x 0	Fenster (und Rahmen) schließen	(delete-window)
c^x 1	alle Fenster im akt. Rahmen schließen (außer jenem, in dem der *cursor* ist	(delete-other-windows)
M %	Suchen und Ersetzen	(query-replace)
M x	Erweitertes Ausführen	(execute-extended-command)
Textverarbeitung		
c^_	Rückgängig machen	(undo)
c^x u	»angekündigtes« Rückgängig machen	(advertised-undo)
c^g	Abbrechen des akt. Kommandos	(kill-command)
c^a	zum Zeilenanfang springen	(beginning-of-line)
c^e	zum Zeilenende springen	(end-of-line)
M <	zum Pufferanfang springen	(beginning-of-buffer)
M >	zum Pufferende springen	(end-of-buffer)
c^l	Zentrieren mit dem *cursor* als Bezugspkt	(recenter)
c^s	Vorwärtssuche	(isearch-forward)
c^r	Rückwärtssuche	(isearch-backward)
c^d	Buchstaben löschen	(delete-char)
c^k	Zeile löschen	(kill-line)
M d	Wort löschen	(kill-word)
M back	Wort löschen rückwärts	(kill-back-word)
M k	Satz löschen	(kill-sentence)
M u	Wort in Großbuchstaben	(uppercase-word)
M l	Wort in Kleinbuchstaben	(lowercase-word)
M c	Buchstabe in Großbuchstabe	(uppercase-letter)
c^x c^o	Leerzeilen löschen	(delete-blank-lines)
M space	Markierung setzen	(set-mark)
c^w	markierte Region löschen	(kill-region)
c^x c^u	Region in Großbuchstaben	(upcase-region)

3.1.5 Das Programm `ediff`

Für den interaktiven Vergleich von Dateien und Verzeichnissen bietet der Emacs das sehr hilfreiche Tool `ediff`. Mit `ediff` lassen sich auch große Dateien relativ einfach und schnell vergleichen. Der Aufruf des Tools kann über den Minibuffer (mit `M x`) oder über die Menüleiste erfolgen. Im Minibuffer erfolgt nacheinander die Eingabe der zu vergleichenden Dateien. `ediff` öffnet zur Bedienung ein eigenes kleines Fenster, in dem die Nummer des aktuellen Unterschieds, die gesamte Anzahl der Unterschiede und die Hilfe (Eingabe von `?`) angezeigt werden.

Beispiel:

Drücken von `M x`
Eingabe von `ediff` im Minibuffer
Im Minibuffer:

```
File A to compare (default datei1.txt): ~/data/dateiX.txt
File B to compare (default datei1.txt): ~/data/dateiY.txt
```

Es sollen die Dateien `~/data/dateiX.txt` und `~/data/dateiY.txt` verglichen werden. Diese müssen hinter dem Doppelpunkt angegeben werden. In Klammern vor dem Doppelpunkt wird der aktuelle Puffer vorgeschlagen (in diesem Fall `datei1`). Dieser wird ausgewählt, wenn der Benutzer einfach durch Drücken von `[Enter]` bestätigt.

Einige Tastenkürzel für `ediff`

`p` oder `[del]`	vorheriger Unterschied
`n` oder `[space]`	nächster Unterschied
`q` oder `z`	Beenden von `ediff`
`?`	Hilfe an/aus
`C+l`	Zentrieren
`a`	Kopieren von unterschiedl. Region A nach B
`a`	Kopieren von unterschiedl. Region B nach A
`ra`	Rückgängig machen der letzten Änderung in Puffer A
`rb`	Rückgängig machen der letzten Änderung in Puffer B
`wa`	Speichern von Puffer A
`wb`	Speichern von Puffer B

3.1.6 Das `bio-mode`-Paket

Für das Arbeiten mit Sequenzdaten kann das Emacs-Paket `bio-mode` sehr nützlich sein. Dieses ist jedoch nicht standardmäßig installiert und muss separat hinzugefügt werden[18]. Es bietet unter anderem die Möglichkeit, `blast`-Aufrufe für Sequenzen aus dem Emacs zu starten oder sie mit `readseq` in ein anderes Format zu konvertieren, wenn die entsprechenden Programme (`blast` bzw. `readseq`) auf dem System verfügbar sind. Außerdem bietet das `bio-mode`-Paket die Möglichkeit, Nukleotidsequenzen in Aminosäuresequenzen oder in das reverse Komplement zu überführen oder beispielsweise den GC-Gehalt zu bestimmen.

Ist das Paket auf dem System installiert, so sollte es beim Öffnen von Dateien mit der Endung `.embl`, `.seq`, `.fas` oder `.phy` automatisch geladen werden. Ansonsten kann der Modus über Eingabe des Kommandos `bio-mode` im Minibuffer manuell gestartet werden. Die `bio-mode`-Kommandos müssen jeweils auch mit Hilfe der erweiterten Kommandoeingabe im Minibuffer (`M x` *Kommando*) aufgerufen werden. Oft muss dazu zunächst eine Region, für die das jeweilige Kommando ausgeführt werden soll, markiert werden.

[18] Achtung: Das Linux `bio-mode`-Paket ist nur mit dem `xemacs` kompatibel. Eine GNU Emacs-kompatible Version ist von Rechnern im Uninetz aus unter http://doku.gobics.de/ erhältlich.

Einige `bio-mode`-Kommandos

`bio-translate-region`	translatiert markierte Nucleotidsequenz in Aminosäureseq.
`bio-blast-on-region`	`blast` aus dem Emacs starten
`bio-gccontent-on-region`	ermittelt GC-Gehalt der Region
`bio-convert-embl2-gtf`	konvertiert EMBL in Gene Table Format (GTF)
`bio-embl-extract`	extrahiert alle CDS mit Aminosäuresequenz aus EMBL-Datei
`bio-readseq`	`readseq` aus dem Emacs starten
`bio-reverse-region`	bildet reverses Komplement der Region

3.2 Der Vi

Der Vi (*visual editor*) ist ein ebenso mächtiger Editor wie der Emacs und er ist noch weiter verbreitet. Er ist im Grunde die visuelle Betriebsart der zeilenorientierten Editoren `ex` und `ed` und verfügt natürlich über hilfreiche Funktionen wie automatisches Einrücken (*indent*) und Umbrechen (*wrap*). Ebenso wie vom Emacs gibt es auch vom Vi diverse Versionen. Bei der hier vorgestellten handelt es sich um `vim`, die wohl bekannteste unter Linux-Benutzern. Diese Version weist einige sehr nützliche Erweiterung des Vi auf, welche die Bedienung erleichtern, und ist zumindest auf den neueren Linux-Systemen meist voreingestellt. Es gibt aber viele weitere Versionen, beispielsweise `elvis`, `xvi`, `stevie` usw.

Meist wird der Vi direkt im Terminalfenster der Shell benutzt, es gibt aber auch graphische Schnittstellen (GUI), z. B. `gvim` und `kvim`. Ein Vorteil des Vi ist sicherlich seine Schnelligkeit beim Starten. Während der Emacs selbst ohne die GUI einige Zeit (je nach Schnelligkeit des Rechners) bei der Initialisierung braucht, ist der Vi nach dem Aufruf sofort da und auch große Dateien werden verhältnismäßig schnell eingelesen und angezeigt.

Der Vi verfügt über verschiedene Modi, die es ermöglichen, dass viele Kommandos durch das Drücken einer einzigen Taste ausgeführt werden können. Dadurch kann der Benutzer mit einer minimalen Anzahl an Tastenanschlägen Dateiinhalte durchmustern und bearbeiten. Viele Vi-Benutzer schätzen besonders, dass es bei der Eingabe meist nicht nötig ist, mehrere Tasten gleichzeitig zu drücken oder die Steuertasten `[strg]` und `[alt]` (wie beispielsweise beim Emacs) zu verwenden. So kann man den Vi im Grunde nutzen, ohne die Finger von den »normalen« Eingabetasten zu nehmen, um Funktionen von Sondertasten (wie Pfeiltasten, BildAuf und -Ab, `[strg]` usw.) der Tastatur zu nutzen.

Ein Vorzug ist die freie Kombinierbarkeit von Kommandos und von Kommandos und Zahlen. So steht dem Benutzer schon nach dem Lernen einiger grundlegender Tastenkommandos eine recht breite Palette an Operationen zur Verfügung. Zu beachten ist, dass der Vi bei der Eingabe zwischen Groß- und Kleinschreibung unterscheidet. In vielen Fällen ist es aber so, dass der Großbuchstabe eine ähnliches Kommando ausführt wie der Kleinbuchstabe, wie wir noch sehen werden. Das erleichtert es ein wenig sich die Kommandos zu merken.

3.2.1 Starten des Vi

Der Aufruf des Vi in der Kommandozeile kann direkt mit einer Datei erfolgen. Existiert eine Datei mit dem übergebenen Pfad, so wird sie in den Puffer geladen und im Terminalfenster der Shell angezeigt. Andernfalls, wird eine Puffer mit dem übergebenen Namen angelegt und beim Speichern wird die Datei erzeugt, sofern der Benutzer Schreibrecht zu dem angegebenen Pfad hat. Wenn beim Starten kein Dateiname übergeben wird, startet der Vi mit einem unbenannten Puffer.

Beispiel:

```
~$ vi Dateiname
```

Nach dem Starten werden dann die ersten Zeilen der Datei angezeigt, sofern sie existiert und nicht leer ist. Leere Zeilen werden durch die Tilde (~) am Zeilenanfang gekennzeichnet, d. h. im Falle einer leeren oder nicht existierenden Datei wird auf der rechten Seite des Terminalfensters am Zeilenanfang von der ersten bis zur vorletzten Zeile das Zeichen ~ angezeigt.

Die letzte Zeile dient als Status- und Kommandoeingabezeile. Hier wird direkt nach dem Start z. B. der Dateiname, die Position und Anzahl der Zeilen und der Zeichen in der Datei angezeigt (siehe Beispiel). Es können hier aber auch Fehler- und Statusmeldungen gezeigt werden, z. B. wenn man in den Eingabemodus wechselt (s. u.)

Beispiel:

```
"datei.txt" 8L, 79C                                           1,1           All
```

Im Beispiel ist eine Statuszeile gezeigt. Der Reihenfolge nach sind die folgenden Informationen gegeben: Die geöffnete Datei heisst `datei.txt`, sie enthält 8 Zeilen (`L` für *lines*) und 79 Zeichen (`C` für *characters*). Auf der rechten Seite des Fensters ist zunächst die Position des Cursor angezeigt. In diesem Fall steht er in der ersten Zeile an der ersten Position (`1,1`). Als letztes ist angegeben, welcher Anteil der Datei in dem Fenster angezeigt wird. Da die Datei im Beispiel recht kurz ist, kann sie komplett angezeigt werden, was mit `ALL` angegeben wird. Bei Dateien, die nicht vollständig in dem Fenster angezeigt werden können, wird hier der prozentuale Anteil des Gezeigten an der gesamten Datei angegeben.

3.2.2 Modi des Vi

Nach dem Starten ist der Vi standardmäßig im Kommandomodus (*command mode*). D. h. man kann so erstmal keinen Text eingeben. Der Kommandomodus dient zur schnellen Bewegen innerhalb einer Datei, zum Editieren, Kopieren und Einfügen von Texten (Textteilen) und zum Speichern, Öffnen usw. von Dateien. Zum Eingeben von Text, dient der Eingabemodus (*insert mode*) oder Textmodus. Auch in diesem Modus kann man einfache Bewegungen innerhalb der Datei ausführen, wie im folgenden Abschnitt beschrieben. In vielen Fällen ist es allerdings komfortabler, dazu in den Kommandomodus zu wechseln. Für den Wechsel in den Textmodus gibt es mehrere Kommandos, die im folgenden Abschnitt aufgeführt sind. Der Wechseln in den Kommandomodus erfolgt immer durch Drücken der `[esc]`-Taste.

3.2.3 Der Textmodus

Um einen Text einzugeben, muss man in den Textmodus wechseln. Dazu gibt es verschiedene Möglichkeiten, je nachdem, wo man mit der Eingabe beginnen möchte. Will man beispielsweise direkt an der Stelle beginnen, an der sich der Cursor befindet, so kann man dies mit `i` tun (*insert*). In der folgenden Tabelle sind die Kommandos aufgelistet, mit denen man in den Textmodus wechseln kann. Für den Anfang reicht es aber `i` zu kennen, da man anschließend den Cursor mit Hilfe der Pfeiltasten an die gewünschte Position bewegen kann. Nach und nach kann man seine Kenntnisse dann um die unter Umständen komfortableren Kommandos erweitern.

Wechsel in den Textmodus

Kommando	Eingabe beginnt ...
`i`	an der aktuellen Position
`a`	an der nächsten Position
`I`	am Zeilenanfang
`A`	am Zeilenende
`o`	am Anfang der nächsten Zeile
`O`	vom Anfang der vorigen Zeile

Zum Verlassen des Textmodus muss man die Taste `[esc]` (*escape*) drücken. Man gelangt dadurch automatisch wieder in den Kommandomodus.

3.2.4 Schnelles Bewegen im Vi

Es gibt sehr viele Kommandos, um sich im Vi im Kommandomodus schnell und komfortabel an die gewünschte Position zu bewegen. Die hier vorgestellte Version `vim` bietet außerdem die Möglichkeit sich mit Hilfe der BildAuf-, BildAb- und Pfeiltasten zu bewegen. Im Folgenden sind exemplarisch einige der wichtigsten Bewegungsfunktionen aufgeführt. Weitere kann man in der Vi-Referenz oder einem der Vi-Manuals nachlesen, die in der Linkliste angegeben sind.

Zunächst zur einfachsten Bewegung: Mit den Tasten `h` und `l` kann man sich zeichenweise nach links bzw. rechts bewegen, mit `j` und `k` bewegt man sich je eine Zeile runter bzw. rauf. Um an den Zeilenanfang bzw. das Zeilenende zu springen gibt es die Kommandos `^` (oder `0`) und `$`. Will man an den Anfang der vorigen oder nächsten Zeile springen, so kann man dies mit − bzw. + tun.

Weiterhin bietet der Vi die Möglichkeit sich Wortweise zu bewegen. Dabei gibt es zwei Arten von Wörtern, »normale« und *Bigwords*, d. h. es gibt zwei Arten, wie der Vi Wortgrenzen definiert. Als normale Wörter werden alle zusammenhängenden Ketten von Buchstaben definiert. Trennzeichen ist also in dem Falle alles, was kein Buchstabe ist, dazu gehören Leerzeichen, Bindestriche, Punkte, Kommas usw. Das Wort CD-ROM wird dann also als zwei Wörter interpretiert. Mit den Tasten `w`, `b` und `e` kann man ein Wort vor, ein Wort zurück und an das Ende des aktuellen Wortes springen. Bigwords sind dagegen als Zeichenketten definiert, die durch ein Leerzeichen oder einen Zeilenumbruch getrennt sind. Hier wäre CD-ROM also ein Wort. Die Tastenkommandos um sich Bigword-weise zu bewegen entsprechen den Kommandos für normale Wörter, nur dass hier die Großbuchstaben verwendet werden müssen.

Wie bereits erwähnt lassen sich die Kommandos beliebig mit Zahlen kombinieren, d. h. beispielsweise `3h` bedeutet dann drei Zeichen nach links.

Syntax:

`[`*n*`]`*T*
Wobei *T* das Kommando repräsentiert und *n* eine ganze Zahl.

Wird nur eine Nummer, ohne Kommando angegeben, so springt der Cursor die entsprechende Anzahl an Zeilen weiter. Einige Kommandos zur Bewegung innerhalb von Dateien sind in der folgenden Tabelle aufgeführt.

Bewegen in der Datei

Kommando	Bewegung
h	Zeichen zurück
l	Zeichen vor
j	Zeile runter
k	Zeile rauf
w/W	Wort/Bigword vor
b/B	Wort/Bigword zurück
e/E	Ende des Wortes/Bigwords
^/0	Zeilenanfang
$	Zeilenende
+/-	Anfang der nächsten/vorigen Zeile
G	Cursor in letzte Zeile der Datei
*n*G	Cursor in Zeile *n* der Datei
n	*n* Zeilen weiter von der aktuellen Position
(/)	Satzanfang/Satzende
{/}	Anfang des Paragraphen/Ende des Paragraphen
H	Cursor in die erste Zeile auf dem Bildschirm
L	Cursor in die letzte Zeile auf dem Bildschirm
M	Cursor in die Mitte des Bildschirmes
[strg]+F	(Bildschirm-)Seite zurück
[strg]+B	(Bildschirm-)Seite vor
%	springen zur zugehörigen Klammer
/	vorwärts suchen
n/N	nächstes/voriges Vorkommen
?	rückwärts suchen

3.2.5 Texte bearbeiten mit dem Vi

Wie bereits angedeutet, bietet der Vi mächtige Funktionen zur Bearbeitung von Texten. Dazu gehören natürlich Kommandos um Textbereiche zu löschen, zu kopieren und einzufügen. Die Kommandos hierfür sind in der Tabelle (s. u.) aufgeführt. Löscht man eine oder mehrere Zeilen, so wird diese vorerst im Puffer gespeichert, so dass man sie an anderer Stelle wieder einfügen kann.

Besonders nützlich ist dabei, Textabschnitte in verschieden benannten Puffern zu speichern, um sie dann später weiter zu verwenden. Standardäßig beispielsweise beim Löschen wird das »Textobjekt« in durchnummerierte Puffer (1-9) gespeichert. Dabei landet das zuletzt gespeicherte immer in dem Puffer mit der Nummer 1. Um etwas aus dem Puffer herauszuholen und es beispielsweise in der nächsten Zeile einzufügen, gibt man ", die Nummer und p an. Drückt man einfach p ohne Nummer so erhält man automatisch den Inhalt von Puffer 1. Man kann die Puffer aber auch anders benennen, so dass es leichter ist, sich zu merken in welchem Puffer, welches Textobjekt gespeichert ist. Dazu gibt man " und den gewünschten Namen ein, dieser darf allerdings nur ein Zeichen lang sein. Um das so bezeichnet Textobjekt wieder aus dem Puffer herauszuholen, gibt man wiederum ", den Namen ein, gefolgt von dem Kommando, das ausgeführt werden soll, also beispielsweise wieder p zum Einfügen.

Syntax:

`[`*n*`]`*T1 T2*

T1 `[`*n*`]`*T2*
Wobei *T1* das Kommando repräsentiert, *T2* das »Objekt« und *n* eine ganze Zahl.

Beispiele:

`cw`	Das Wort ändern.
`y2$`	Von der aktuellen Position bis zum Ende dieser Zeile und der folgenden zwei Zeilen kopieren.
`3yy$`	Die aktuelle Zeile und die folgenden zwei Zeilen kopieren.
`y2-`	Es werden die aktuelle Zeile und die zwei vorangehenden kopiert.
`2r*`	Zwei Zeichen von der aktuellen Position durch `*` ersetzen.
`10i-`	Zehn mal das Zeichen - einfügen.

Kommandos zur Textbearbeitung

Kommando	Operation
`c`	(*change*)
`C`	bis zum Zeilenende ersetzen
`r`	Zeichen unter dem Cursor durch anschließend eingegebenes ersetzen
`x`	Zeichen unter dem Cursor löschen
`X`	Zeichen vor dem Cursor löschen
`~`	Groß- und Kleinschreibung vertauschen
`[`*n*`]dd`	Zeile bzw. *n* Zeilen löschen (*delete*)
`[`*n*`]yy`	Zeile bzw. *n* Zeilen kopieren (*yank*)
`y[`*n*`]`*move*	aktuelle Position bis *move* kopieren
`d[`*n*`]`*move*	aktuelle Position bis *move* löschen
`D`	Löschen von der aktuellen Position bis zum Zeilenende
`p`	Einfügen in der nächsten Zeile
`P`	Einfügen ab der aktuellen Position
`"`*name* `p`	Einfügen aus Puffer *name* (max. `1` Zeichen!)
`u`	Rückgängig machen der letzten Änderung (*undo*)
`U`	Rückgängig machen aller Änderungen in der aktuellen Zeile
`.`	Wiederholen des letzten Kommandos
`v`	Markieren von Textstücken in visuellen Modus
`[strg]+G`	Dateinamen anzeigen

3.2.6 Visueller Modus

Der visuelle Modus (*visual mode*) bietet die Möglichkeit im Vi Bereiche zu markieren, um sie zu bearbeiten, sie zu durchsuchen oder sie zu kopieren. Dabei gibt es generell drei Arten visueller Modi, den Zeichenmodus (*character mode*), den Zeilenmodus (*line mode*) und den Blockmodus (*block mode*). Sie unterscheiden sich jeweils darin, wie Bereiche des Textes markiert werden. Interessant ist dabei vor allem der Blockmodus, da man Textdateien so spaltenweise bearbeiten kann. Mit `o` kann man jeweils an den Anfang des markierten Bereichs springen (ansonsten ist man beim

Verlassen automatisch am Ende des Bereichs), verlassen kann man den visuellen Modus mit Hilfe von `[esc]`.

Wechseln in visuelle Modi	
`v`	Character Mode
`V`	Line Mode
`[strg]+v`	Block Mode

Spezielle Blockmodi	
`vaw/vaW`	Wort (inkl. Leerzeichen)
`viw/viW`	Wort (exkl. Leerzeichen)
`vas`	Satz (inkl. abschließendes Leerzeichen)
`vis`	Satz (exkl. abschließendes Leerzeichen)
`vap`	Satz (exkl. abschließendes Leerzeichen)
`vip`	Paragraph (inkl. abschließendes Leerzeichen)
`vab`	(...)-Block (inkl. Klammern)
`vib`	(...)-Block (exkl. Klammern)
`vaB`	{...}-Block (inkl. Klammern)
`viB`	{...}-Block (exkl. Klammern)

Viele Kommandos im visuellen Modus sind mit denen des »normalen« Kommandomodus identisch. Im visuellen Modus wird das jeweilige Kommando immer auf den gesamten markierten Bereich angewendet. Bei der Eingabe wird dies durch `'<,'>` am Anfang der Kommandozeile angezeigt. Einige Kommandos unterscheiden sich dabei etwas in jeweiligen visuellen Modus.

3.2.7 ex-Kommandos

Der `ex` ist wie schon erwähnt ein zeilenbasierter Editor, auf dem der Vi basiert. Der `ex`-Modus ist aus dem Vi durch Eingabe des `:` (im Kommandomodus) zu erreichen. Der Vi kann aber auch mit dem Parameter `-e` direkt im `ex`-Modus gestartet werden (oder indem man das Kommando `ex` in der Shell aufruft). In diesem Editor sieht/bearbeitet man jeweils nur eine Zeile, d. h. beim Starten sieht man zunächst gar nichts, außer dem `:`, hinter dem die Kommandos eingegeben werden, da man *vor* der ersten Zeile der Datei steht. Durch Eingabe von `vi` kann man aus dem `ex`-Modus in den `vi`-Modus wechseln.

Durch Eingabe eines `:` (im Kommandomodus) gefolgt von einem Kommando können im Vi `ex`-Befehle ausgeführt werden. Zu diesen gehören beispielsweise das Schreiben einer Datei (`:w`), das Bewegen zu einer bestimmten Zeile (`:`*n*) und die mächtigen Suchen/Ersetzen-Funktion (`:s/`*Muster*`/`*Ersetzungszeichen*`/`). Auch Shell-Kommandos können im `ex`-Modus ausgeführt werden. Angezeigt wird dies durch Eingabe von eines Ausrufezeichens. Teilweise können die Kommandos dabei die aktuell geöffnete Datei als Eingabe verwenden, z. B. `sort` oder `ispell`. Dazu muss nach dem Kommando ein `%` angegeben werden, ggf (z. B. bei `sort`) müsste dabei außerdem die Ausgabe umgeleitet werden, wenn diese nicht auf dem Standardausgabestrom erfolgen soll.

Einige ex-Kommandos

Kommando	Operation
`:q`	Verlassen
`:q!`	Verlassen ohne zu Speichern
`:x` (=`:wq`)	Speichern und Verlassen
`:w`	Speichern
`f` *name*	aktuellen Dateinamen auf *name* setzten
`:cd` *dir*	wie `cd` in der Shell
`:e` *dat*	Datei *dat* bearbeiten
`:`*n*	Bewegen nach Zeile *n*
`:s/`*Suchen*`/`*Ersetzen*`/`*Option*	Suchen/Ersetzen
`:wm`	Zeilen umbrechen (*wrap margin*)
`:sp` *Datei*	Fenster vertikal teilen und *Datei* im oberen Fenster öffnen (Fenster wechseln mit 2x `strg+w`)
`:=`	Anzahl der Zeilen anzeigen
`:set` *Kommando*	Setzen der Einstellungen, z. B. `:set autoindent` (automat. Einrücken, s. u.)
`&`	Wiederholen des letzten `ex`Kommandos
`!` *Kommando*	Shell-Kommandos ausführen
`!` *Kommando* `%`	Shell-Kommandos ausführen mit dem aktuellen Puffer als Eingabe

Kommandos mit `set`

Kommando	Operation
`:help set`	Hilfe zu `set`
`:set tabstop=4W`	»Tab« wird auf 4 Leerzeichen gesetzt
`:set nu/:set nonu`	Zeilennummern einblenden
`:set ic/:set noic`	Groß- und Kleinschreibung nicht berücksichtigen/ berücksichtigen (*case insensitive*/*case sensitive*)
`:set list/:set nolist`	Anzeigen/Ausblenden von Tabs und Zeilenumbruechen

Im Folgenden wird nur auf das **Suchen und Ersetzen** etwas näher eingegangen, da diese Funktion des `ex` sehr komplex sein kann. Unter anderem ist es möglich Reguläre Ausdrücke und Wildcards zu verwenden. Dazu muss die Option `magic` aktiviert sein (`:set magic`). Teilweise ähnelt die Verwendung der Ausdrücke dem, was wir schon bei dem Kommando `grep` (3, 21) kennengelernt haben. Wie auch in der Shell muss die Interpretation der Metazeichen mit dem Backslash unterdrückt werden, wenn das Zeichen selbst gemeint ist.

Da der `ex` ein zeilenbasierter Editor ist, gelten alle Kommandos immer nur für die aktuelle Zeile. Der Parameter `g` steht beispielsweise für globales Ersetzen in der aktuellen Zeile (ohne `g` wird nur das nächste Vorkommen in der aktuellen Zeile ersetzt. Wenn man ein *Muster* in der ganzen Datei ersetzen möchte, muss man dies durch das Zeichen `%` bekannt machen.

Beispiele:

`:s/Hans/Franz/`
Das nächste Vorkommen von `Hans` in der aktuellen Zeile wird durch `Franz` ersetzt.

`:s/Hans/Franz/g`
Das alle Vorkommen von `Hans` in der aktuellen Zeile werden durch `Franz` ersetzt.

`:%s/Hans/Franz/`
Das alle Vorkommen von `Hans` in der Datei werden durch `Franz` ersetzt.

`:s/hans/\u&/`
Aus `hans` wird `Hans`.

`:s/hans/\U&/`
Aus `hans` wird `HANS`

`:s/hans/\U&|s/franz/~`
`hans` wird durch `HANS` ersetzt, dann `franz` durch `FRANZ`.
(Die Tilde ~ steht für das vorherige Ersetzungszeichen.)

Einige Metazeichen, sowie `ex`-Kommandos und Optionen, die beim Suchen und Ersetzen nützlich sein können, sind in der folgenden Tabelle zusammengefasst.

Suchen und Ersetzen mit `ex`	
Ausdruck	Operation
`~`	letztes Ersetzungszeichen
`\( \)`	Klammerung von Ausdrücken
`\u&` bzw. `\U&`	zu Großschreibung (Wortanfang bzw. alle Zeichen des Wortes)
`\l&` bzw. `\L&`	zu Kleinschreibung (Wortanfang bzw. alle Zeichen des Wortes)
Wenn `magic` gesetzt ist:	
`^`*Zeichenkette*	*Zeichenkette* muss am Zeilenanfang stehen.
Zeichenkette`$`	*Zeichenkette* muss am Zeilenende stehen.
`.`	Beliebiger Buchstabe
`[a-z]`	Alle Buchstaben/Ziffern im Bereich (hier alle Kleinbuchstaben)
`[^a-z]`	Alle Buchstaben außer die angegebenen im Bereich
`[`*Zeichenkette*`]`	Alle Buchstaben in *Zeichenkette*
`[^`*Zeichenkette*`]`	Alle Buchstaben außer die in *Zeichenkette*
$X\star$	0 oder mehr Vorkommen von X
`\<`*Zeichenkette*	*Zeichenkette* muss am Anfang eines Wortes stehen
Zeichenkette`\>`	*Zeichenkette* muss am Ende eines Wortes stehen
Optionen	
`i`	Groß-/Kleinschreibung ignorieren (*ignore case*)
`g`	Alle Vorkommen in der Zeile ersetzen (*global*)
`c`	Bei Ersetzung nachfragen (*confirm*)
Bereiche	
`%`	Ersetzen in der ganzen Datei
n1`,`*n2*`s/../../`	zwischen Zeile *n1* und *n2*
`:g/`*Zeichenkette*`/`	alle Zeilen, die *Zeichenkette* enthalten

4 Anpassen der Shell

4.1 Die Umgebung – Voreinstellungen in der Shell

Wenn ein Programm aufgerufen wird, so wird ihm von der Shell die Umgebung (*environment*) zur Verfügung gestellt. D. h. dem Programm wird beim Aufruf eine Liste von Schlüssel-Wert-Paaren übergeben, beispielsweise die Variable `DISPLAY`. Die Schlüssel entsprechen dabei Variablennamen, denen beliebige Werte zugewiesen werden können. Dies geschieht oft direkt beim Öffnen der Shell, sofern die Variable gesetzt ist, kann aber auch später erfolgen. Das automatische Setzen beim Öffnen einer Shell geschieht durch globale und lokale Dateien, wie `.profile` und `.shrc` (siehe Abschnitt 4.2, S. 52).

Solche Shell-Variablen werden aber nicht automatisch an alle in dieser Shell gestarteten Programme (*Kind*-Prozesse) weitergegeben. Dies geschieht nur, wenn die betreffende Variable durch das `export`-Kommando exportiert wird. Wird eine Variable exportiert, so wird sie als Umgebungsvariable bezeichnet, ansonsten gehört sie nur zu den Shell-Variablen.

Um sich die Variablen der Shell anzeigen zu lassen, gibt es die Kommandos `set` und `env`. Mit `set` werden alle Variablen der Shell angezeigt, mit `env` nur die Umgebungsvariablen. Einzelne Variablen kann man sich mit dem Kommando `echo` anzeigen lassen. Zu beachten ist, dass die Variable, wenn sie innerhalb der Shell angesprochen, wird durch das Zeichen `$` kenntlich gemacht werden muss. Beim Setzen dagegen wird sie ohne `$` angegeben.

Beispiel:

```
BLASTDB=/opt/blast/blastdb
export BLASTDB
```

Im Beispiel wird der Variable `BLASTDB` der Wert `/opt/blast/blastdb` zugewiesen. In der folgenden Zeile wird die Variable exportiert. Somit wird jedem Programm, das in dieser Shell gestartet wird, das Schlüssel-Wertpaar ›BLASTDB—/opt/blast/bastdb‹ übergeben. Die Ausgabe der Variable mit `echo` sieht dann folgendermaßen aus:

```
~$ echo $BLASTDB
/opt/blast/blastdb
```

4.2 `.profile` und `.shrc`

Die Dateien `.profile` (in der `bash` oft auch `.bash_profile`) und `.shrc` (bzw. `.bashrc`) dienen zum Speichern von Voreinstellungen (Umgebungsvariablen, Shell-Variablen, Aliasen, Standardmaske, usw.) für das Arbeiten mit der Shell. Die Voreinstellungen in der Datei `.profile` werden bei jedem Einloggen des Benutzers vorgenommen, während die Einstellungen in der `.shrc` nur für `non-login`-Shells (also nur beim lokalen Anmelden) ausgeführt werden.

Es gibt systemweite (globale) Einstellungen und benutzerspezifische Einstellungen. Die benutzerspezifischen Einstellungen kann der Benutzer beliebig für sich anpassen. Dazu liegen im Heimatverzeichnis jedes Benutzers seine eigenen Dateien `.profile` und `.shrc`. Auf die verwendete Syntax wird im nächsten Kapitel (Grundlagen der Shell-Programmierung, S. 55) näher eingegangen.

Beispiel einer `.profile`:
Zeilen, die mit einem Gatterkreuz (`#`) werden vom Interpreter ignoriert. Hierbei handelt es sich um Kommentare oder auskommentierte Kommandos, die nicht ausgeführt werden sollen.

```
# ~/.bash_profile: executed by bash(1) for login shells.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/login.defs
#umask 022
umask 077
# the rest of this file is commented out.

# Get the aliases and functions
# include .bashrc if it exists
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi

# set PATH so it includes user's private bin if it exists
if [ -d ~/bin ] ; then
    PATH="${PATH}":~/bin:.:/usr/local/bin:/opt/blast:
fi
export PATH

#BLASTDB=/opt/blast/db
#export BLASTDB

# do the same with MANPATH
#if [ -d ~/man ]; then
#    MANPATH=~/man:"${MANPATH}"
#    export MANPATH
#fi

#export DISPLAY=172.23.85.116:0.0
```

***Beispiel* einer `.shrc`:**

```
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If running interactively, then:
if [ "$PS1" ]; then
    # don't put duplicate lines in the history. See bash(1) for more options
    # export HISTCONTROL=ignoredups

    # check the window size after each command and, if necessary,
    # update the values of LINES and COLUMNS.
    #shopt -s checkwinsize

    # enable color support of ls and also add handy aliases
    if [ "$TERM" != "dumb" ]; then
        eval `dircolors -b`
        alias ls='ls --color=auto'
        #alias dir='ls --color=auto --format=vertical'
        #alias vdir='ls --color=auto --format=long'
    fi

    # some more ls aliases
    #alias ll='ls -l'
    #alias la='ls -A'
    #alias l='ls -CF'
```

```
    alias di='ls -lah --color |more'

    # set a fancy prompt
    PS1='\u@\h:\w\$ '

    # If this is an xterm set the title to user@host:dir
    #case $TERM in
    #xterm*)
    #    PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"'
    #    ;;
    #*)
    #    ;;
    #esac

    # enable programmable completion features (you don't need to enable
    # this, if it's already enabled in /etc/bash.bashrc).
    #if [ -f /etc/bash_completion ]; then
    #  . /etc/bash_completion
    #fi
fi
```

4.3 Aliase

Mit dem Kommando `alias` können alternative Namen für andere Kommandos der Shell gesetzt werden. Das ist vor allem nützlich, wenn man ein bestimmtes Kommando immer mit den gleichen Parametern aufruft. Diesen Aufruf kann man durch ein Alias ersetzen und damit vereinfachen. Zum Setzen der Aliase wird oft eine Datei `.alias` (im Heimatverzeichnis) verwendet, diese wird aber nicht immer ausgelesen. Aliase können aber auch in die `.shrc` (oder das `.profile`) geschrieben werden, wie im vorherigen Abschnitt im Beispiel der `.shrc` gezeigt oder direkt in der Kommandozeile gesetzt werden. Es ist zu beachten, dass die Syntax in der `.alias` und der `.shrc` (bzw. `.profile`) nicht gleich ist.

Bereits gesetzte Aliase können durch den einfachen Aufruf von `alias` in der Shell erfragt werden und mit dem Kommando `unalias` wieder gelöscht werden.

Beispiel:

Syntax in der `.shrc` oder direkt in der Kommandozeile

```
alias dir='ls -lah --color'
```

Syntax in der `.alias`

```
alias dir 'ls -lah --color'
```

Im Beispiel wird der Zeichenkette `dir` das Kommando `ls -lah --color` zugewiesen. Dabei muss das Kommando in Hochkommata gesetzt werden, wenn es aus mehreren Teilen besteht. Erfolgt in der Shell der Aufruf `dir`, so wird das gesamte in Hochkommata gesetzte Kommando ausgeführt.

5 Grundlagen der Shell-Programmierung

Im Folgenden werden einige Grundlagen der Shell-Programmierung (*Shell-Scripting*) vorgestellt, die beim Arbeiten in der Shell sehr nützlich sein können. Dazu gehören Variablen, einfache Kontrollstrukturen, Schleifen (bedingte Wiederholungen) und das Einlesen von Eingaben und Argumenten.

Alle Kommandos, die auf der Konsole zur Verfügung stehen, können genauso in eine Datei geschrieben werden. So kann man beispielsweise mehrere Kommandos automatisiert hintereinander weg ausführen lassen und sich ein wenig Schreibarbeit sparen. Durch Schleifen zur bedingten Wiederholung hat man ein Werkzeug, mit dem man simple aber oft mühsame Arbeiten schnell erledigen kann, beispielsweise Backup-Dateien suchen und löschen oder Log-Dateien aufräumen.

Aber auch wenn man selbst keine Shell-Skripte schreiben will und die Kommandozeile nur in Ausnahmen benutzt, ist es sinnvoll grundlegende Kenntnisse auf diesem Gebiet zu haben, da ein Großteil der Konfiguration von Programmen unter Linux mit Hilfe von Shell-Skripten erfolgt. Zwei Beispiele zur Konfiguration der Shell-Umgebung haben wir bereits kennengelernt, die `.profile` und die `.shrc`.

5.1 Shell-Skripte schreiben

Die denkbar einfachste Form eines Shell-Skriptes ist ein Kommando oder mehrere Kommandos, die nicht in die Kommandozeile, sondern in eine Datei geschrieben werde, beispielsweise die Kommandos `echo` und `date`, wie im Beispiel gezeigt.

Beispiel:

```
#!/bin/bash

echo -n "Heute ist "
date
```

Im Beispiel wird das Skript `datum` gezeigt. Es wird ein Satz und anschließend das aktuelle Datum ausgegeben. Dabei wird der Zeilenumbruch hinter der Ausgabe von `echo` mit dem Parameter `-n` unterdrückt. Die erste Zeile (`#!/bin/bash`) ist der Interpreteraufruf (siehe 5.1.2, S. 57).

Um ein Shell-Skript auszuführen, muss der Shell durch das Kommando `source` (oder das Synonym von `source`, den `.`) bekannt gemacht werden, dass es sich hier um etwas Ausführbares handelt (*sourcing*). Alternativ kann ein Skript auch direkt mit dem Interpreteraufruf `sh` (bzw. `bash`) ausgeführt werden. Zu beachten ist, dass beim Aufruf des Interpreters aus der Shell die Umgebung nicht mit übergeben wird.

Um ein Skript durch einen »einfachen« Aufruf (also nur durch Eingabe des Skriptnamens) ausführen zu können, müssen die Rechte entsprechend angepasst werden. D. h. es muss das Ausführrecht mit Hilfe des Kommandos `chmod` für alle, die das Skript benutzen sollen, hinzugefügt werden. Betrifft dies beispielsweise nur den Benutzer selbst, so kann er dies mit dem Aufruf `chmod u+x` *Skriptname* tun (siehe 2.2.8, S. 22). Selbstverständlich müssen auch alle Nutzer, die ein Skript benutzen sollen Leserecht habe. Im Falle des Eigentümers erübrigt sich jedoch meist das Anpassen des Leserechts, da er dieses standardmäßig auf seinen Dateien hat.

Nach Hinzufügen des Ausführrechts (`chmod u+x datum`) kann das Skript durch einfachen Aufruf des Skriptnamens ausgeführt werden.

```
~$ ./datum
Heute ist Mon Mar 21 10:40:13 CET 2005
```

Ein Skript, das mit Ausführrecht ausgestattet ist, muss beim Aufruf in einem Verzeichnis stehen, das der Shell bekannt ist, also in der Variable `PATH` enthalten ist. Andernfalls muss der ganze Pfad (absolut oder relativ) angegeben werden. Im Beispiel (s. o.) befinden wir uns in dem Verzeichnis, in dem das Skript gespeichert ist. Beim Aufruf wird der relative Pfad angegeben (`./`). Dies ist nötig, da das Arbeitsverzeichnis standardmäßig nicht im Pfad der Shell enthalten ist (außer unter SuSE-Linux). Um den Pfad entsprechend anzupassen, muss die `PATH`-Variable um den `.` erweitert werden, der ja das aktuelle Verzeichnis repräsentiert.

Standardmäßig werden Programme oder Skripte im `bin`-Verzeichnis (*binary*) gespeichert. Meist gibt es ein globales `bin`-Verzeichnis, das direkt unter `/` liegt (und weitere unter `/usr`), und eines im Heimatverzeichnis des Benutzers. Da man im globalen `bin`-Verzeichnis als einfacher Benutzer normalerweise kein Schreibrecht hat, speichert man eigene Skripte im eigenen `bin`-Verzeichnis und fügt dieses dem Pfad der Shell hinzu. Dabei ist es grundsätzlich egal wie das Verzeichnis heisst, das entscheidende ist nur, dass es im Pfad gefunden wird. Das Anpassen des Pfades kann direkt in der Shell oder mit Hilfe der Dateien `.profile` bzw. `.shrc` erfolgen (siehe 4.2, S. 52).

Versucht man ein Skript mit einfachem Aufruf auszuführen, *ohne* vorheriges Setzen der Ausführrechte (bzw. ggf. Leserechte), wird auf der Konsole die Fehlermeldung der Form ›`bash:` *Skriptname*`: Permission denied`‹ ausgegeben. Mit dem Kommando `echo $?`. kann man sich den Rückgabewert (siehe 5.1.2, S. 58) ausgeben lassen. Wenn ein Prozess erfolgreich ausgeführt wurde, ist der Rückgabewert standardmäßig gleich 0. Beendet sich der Prozess fehlerhaft, so wird ein Wert ungleich 0 zurückgegeben. In diesem Fall ist der Rückgabewert `126`.

Beispiel:

```
~$ ./datum
bash: ./datum: Permission denied
~$ echo $?
126
```

Die beiden Kommandos in dem Skript können natürlich auch gekoppelt direkt in der Shell ausgeführt werden. Dazu werden die Zeilenumbrüche beispielsweise durch `;` ersetzt.

Beispiel:

```
~$ echo -n "Heute ist "; date
Heute ist Mon Mar 21 10:40:13 CET 2005
```

5.1.1 »Ausführbarmachen« eines Skriptes

Um ein Skript ausführbar zu machen, sind wie bereits gesagt einige Anpassungen nötig. Der Nutzer hat dabei verschiedene Möglichkeiten die Ausführbarkeit zu erreichen, die im Folgenden noch einmal zusammengefasst sind.

1. Kontrollieren der `PATH`-Variable (`bin`-Verzeichnis enthalten?)
   (Alternativ: anderes Verzeichnis aus dem Pfad verwenden, in dem man Schreibrecht hat)

2. ggf. anpassen des Pfades
3. Kopieren des Skripts ins `bin`-Verzeichnis
   (Alternativ: Aufruf mit ganzem Pfad (absolut oder relativ) oder Link setzen)
4. Ausführ- und Leserechte mit Hilfe von `chmod` setzen
   (Alternativ nur für Shell-Skripte: Ausführen mit `source`)
5. mit dem Kommando `which` kann abgefragt werden, welches Programm (wo im Pfad?) beim Aufruf ausgeführt wird.

5.1.2 Genereller Aufbau eines Shell-Skriptes

Ein Skript enthält normalerweise in der ersten Zeile den Interpreteraufruf. Dieser zeigt der Shell, welcher Interpreter das Skript ausführen soll. Der Interpreteraufruf beginnt wie ein Kommentar mit einem Gatterkreuz (`#`), auf dieses muss ohne Leerzeichen ein Ausrufezeichen (`!`) und anschließend der Pfad folgen. Ein `bash`-Skript beginnt, wie wir bereits gesehen haben, mit der Zeile `#!/bin/bash`. Der `bash`-Interpreter befindet sich also unter `/bin`. Auch bei anderen Skript-Sprachen wird der Interpreteraufruf im eben beschriebenen Format angegeben, bei `Perl`-Skripten ist dies z. B. standardmäßig `/usr/bin/perl`.

Die Kommandos in einem Skript werden jeweils mit einem Zeilenumbruch abgeschlossen. Alternativ können sie aber auch durch andere Zeichen beispielsweise `;` voneinander getrennt werden. Alle möglichen Zeichen zur Trennung von Kommandos können in der `man`-Page der `bash` nachgelesen werden. Hier finden sich auch viele weitere Hinweise zum Umgang mit der `bash`.

Kommentare sind neben Interpreteraufruf und Kommandos das dritte Element, das Shell-Skripte enthalten können. Ein Kommentar wird dadurch gekennzeichnet, dass er mit einem Gatterkreuz beginnt. Alles was auf das Gatterkreuz folgt wird vom Interpreter ignoriert. Kommentare müssen dabei nicht am Zeilenanfang beginnen, sie können auch auf ein Kommando folgen. Das Kommentieren von Programmen und Skripten mag manchmal überflüssig oder gar lästig erscheinen, man sollte aber nicht unterschätzen, wie hilfreich ein einfacher Kommentar für einen selbst oder andere, die den Kode benutzen sollen, sein kann.

Die Benutzung von Programmen oder Skripten, kann außerdem noch durch eine angemessene Fehlerbehandung mit entsprechenden Meldungen an den Benutzer, deutlich erleichtert werden. Erwartet ein Skript z. B. Argumente, so sollte auf jeden Fall eine Meldung ausgegeben werden, falls keine oder nicht die richtigen Argumente übergeben werden. So spart man sich und anderen Nutzern langwieriges Suchen und Herumprobieren, wenn etwas nicht klappt.

Im Beispiel wird ein Skript gezeigt, das nicht komplett verstanden werden muss, sondern nur einen kleinen Eindruck geben soll, wozu Shell-Skripte verwendet werden können. Auf einige die verwendeten Elemente, wie die `if`-Klausel, die Variable `$#`, das Programm `test`, usw. wird im weiteren Verlauf des Kapitels näher eingegangen.

Das gezeigte Skript beginnt in der ersten Zeile wie erwartet mit dem Interpreteraufruf. Als nächstes folgt ein dreizeiliger Kommentar. Dann beginnt das eigentliche Skript mit der Zeile `if ! test $# -eq 1`, die testet, ob die Anzahl der Argumente, die dem Skript beim Aufruf übergeben wurden, ungleich eins ist. Ist dies der Fall, so werden die Anweisungen hinter `then` ausgeführt. Es wird also die Zeile `usage: check_qstat script` ausgegeben (die dem Benutzer sagen soll, wie er das Skript richtig zu verwenden hat) und das Skript beendet sich mit dem Rückgabewert (s. u.) `-1`. Wird der Ausdruck hinter dem `if` negativ ausgewertet, so springt der Interpreter in die Zeile nach `fi` und setzt die Variable `script` auf den Wert des ersten (und einzigen) Arguments (`$1`). Dann folgt eine Pipeline von Kommandos, die überprüfen, ob ein Skript mit dem Namen, der in der Variable `script` gespeichert ist, in der Warteschleife der Batch-Prozesse steht bzw. gerade ausgeführt wird. Anschließend beendet sich das Skript mit `exit $?`, was bewirkt, dass der Rückgabewert des letzten Programms der Pipeline, also des Programms `awk`, der Rückgabewert des Skriptes selbst ist.

Beispiel:

```
#!/bin/bash

#
# checks qstatus of subprocess with given script name.
#

if ! test $# -eq 1
then
    echo "usage: check_qstat script"
    exit -1
fi
script=$1

qstat |  grep $script  | awk '{if($5 ~ /R|Q/){exit 3}}'
exit $?
```

Der **Rückgabewert** ist ein Wert, den ein Prozess (oder eine Methode) zurückliefert. Im Falle von Prozessen spricht man auch von *exit status*, im Fall von Methoden/Subroutinen eines Programms von *return value*. Hierbei kann es sich um einen Status oder anderen Wert handeln. Der Rückgabewert des letzten Prozesses ist in der Shell in der Variable `$?` gespeichert und kann mit `echo $?` abgefragt werden. Meist wird hierbei die 0 als Standardwert zurückgegeben, ist ein Fehler aufgetreten, so ist der Rückgabewert $\neq 0$.

5.2 Umgang mit Variablen in der Shell

Die Handhabung von Variablen in der Shell ist ja schon an einigen Stellen angedeutet worden. In diesem Abschnitt wird noch einmal zusammengefasst, wie Variablen in der Shell angesprochen und verwendet werden. Außerdem werden einige nützliche Konstrukte und arithmetische Operationen mit Variablen vorgestellt. Alle folgenden Ausdrücke können genauso in der Shell wie innerhalb eines Skriptes (beispielsweise in der `.profile` oder der `.shrc`) verwendet werden. Die Beispiele werden für die Kommandozeile angegeben, wobei das `$`-Zeichen am Anfang wie gehabt den Prompt der Shell symbolisiert.

5.2.1 Zuweisen von Werten

Bei der Zuweisung werden Variablen nur mit ihrem einfachen Namen angesprochen. Die Zuweisung erfolgt mit Hilfe des Gleichheitszeichens. Dabei dürfen zwischen Variablenname und Wert keine Leerzeichen auftreten. Handelt es sich bei dem Wert, der zugewiesen wird, um eine Zeichenkette, die Leerzeichen enthalten soll, so muss die ganze Zeichenkette in Anführungszeichen oder Hochkommata gesetzt werde. D. h. das Setzen der Variable `USERNAME` erfolgt beispielsweise folgendermaßen:

Beispiel:

```
~$ USERNAME="Anneliese Schmidt"
```

Ohne Anführungszeichen würde die Shell der Variable `USERNAME` den Wert `Anneliese` zuweisen und versuchen ein Kommando `Schmidt` auszuführen, was eine Fehlermeldung zur Folge hätte. Soll also nur ein einfacher Wert ohne Leerzeichen zugewiesen werden, so kann man die Anführungszeichen auch weglassen.

5.2.2 Ansprechen des Wertes

Soll der Wert einer Variable angesprochen werden, so wird dies mit dem `$`-Zeichen bekannt gemacht, welches dem Variablennamen vorangestellt wird.

Beispiel:

```
~$ echo $USERNAME
Anneliese Schmidt
```

Die Variable kann dabei auch in geschweifte Klammern gesetzt werden, um sie von evtl. folgenden Zeichen oder Zeichenketten zu trennen.

```
~$ echo ${USERNAME}-Meier
Anneliese Schmidt-Meier
```

Wird eine Variable angesprochen, die nicht definiert oder nicht gesetzt ist, so wird von der Shell an ihre Stelle eine leere Zeichenkette gesetzt.

Beispiel:

```
~$ echo xxx${OTHERNAME}yyy
xxxyyy
```

5.2.3 Exportieren von Variablen

Nach dem Setzen einer Variable ist diese lokal in der Shell verfügbar. Wenn man einen Prozess in der Shell startet hat er jedoch keinen Zugriff auf diese Variable. Um Prozessen die Variable zugänglich zu machen, gibt es das Kommando `export`. Durch den Aufruf `export` *Variablenname*, wird die betreffende Variable dann zu den Umgebungsvariablen (siehe 4.1, S. 52) hinzugefügt und somit an alle *Kind*-Prozesse der Shell vererbt.

Beispiel:

```
~$ export USERNAME
```

Im Beispiel wird die Variable `USERNAME` exportiert. Jedes Programm, das von nun an in dieser Shell gestartet wird, kann auf die Variable zugreifen. Beim Exportieren einer Variable muss diese wieder nur mit dem einfachen Namen **ohne** vorgestelltes `$`-Zeichen übergeben werden.

5.2.4 Freigeben von Variablen

Um Variablen wieder freizugeben, verwendet man das Kommando `unset`. Nach dem Aufruf `unset` *Variablenname* ist die Variable *Variablenname* aus der Symbol-Tabelle der Shell wieder verschwunden, als wäre die Variable nie gesetzt worden. Auch hier wird die Variable ohne `$` übergeben.

Beispiel:

```
~$ unset USERNAME
```

5.2.5 Arithmetische Operationen auf Variablen mit dem Kommando `let`

Mit dem Kommando `let` kann man einfache arithmetische Operationen auf Variablen durchführen. Dies kann man direkt bei der Zuweisung tun oder später. Wichtig ist, dass arithmetische Zeichen beispielsweise + ohne `let` einfach als Teil der Zeichenkette interpretiert werden.

Beispiel:

Zuweisung, beim Setzen der Variable

```
~$ let ZAHL=1+2
~$ echo $ZAHL
3
```

Die gleiche Zuweisung ohne `let`

```
~$ ZAHL=1+2
~$ echo $ZAHL
1+2
```

Zuweisung, wenn die Variable bereits gesetzt ist
(Der Ausdruck `ZAHL+=3` im Beispiel ist eine verkürzte Schreibweise für `ZAHL=ZAHL+3`.)

```
~$ ZAHL=2
~$ let ZAHL+=3
~$ echo ZAHL
5
```

Ist die Variable `ZAHL` bei der Verwendung von `let` nicht oder mit einer Zeichenkette gesetzt ist, so wird von der Shell automatisch der Wert `0` angenommen.

Beispiel:

```
~$ ZAHL="keine Zahl"
~$ let ZAHL+=3
~$ echo $ZAHL
3
```

Alternativ können ganzzahlige Operationen auch mit Hilfe des Konstruktes `$[`*<expr>*`]` ausgeführt werden. ***Beispiel:***

```
~$ echo $[1+2]
3
```

5.2.6 Variablen-Konstrukte

Die Shell bietet weiterhin die Möglichkeit Variablen qualifiziert anzusprechen. Damit kann man beispielsweise einen Standardwert setzen, der zurückgegeben wird, falls die Variable nicht definiert ist. Die folgenden Konstrukte sind möglich:

1. `${VAR:-`*Expr*`}`
   Ist die Variable `VAR` gesetzt und nicht `null`, so wird ihr Wert ausgewertet. Andernfalls wird der Ausdruck *Expr* ausgewertet.

2. `${VAR:=`*Expr*`}`
   Wie im vorigen Konstrukt wird der Wert der Variable `VAR` ausgewertet, falls diese gesetzt und nicht `null` ist. Andernfalls wird *Expr* ausgewertet und `VAR` auf den Wert *Expr* gesetzt.

3. `${VAR:?[`*Expr*`]}`
   Wenn `VAR` gesetzt ist und nicht `null`, wird der Wert von `VAR` ausgewertet. Andernfalls wird der Wert von *Expr* nach Standarderror geschrieben, und der Prozess beendet sich mit einem Fehlerwert ungleich `0`.

4. `${VAR:+`*Expr*`}`
   Es wird eine leere Zeichenkette ausgegeben, wenn `VAR` gesetzt und nicht `null` ist. Andernfalls wird *Expr* ausgewertet.

5. `${#VAR}`
   Gibt die Länge des Wertes von Variable `VAR` aus. Wenn `VAR` nicht gesetzt oder `null` ist, wird der Ausdruck als `0` ausgewertet.

Alle Konstrukte testen, ob die Variable `VAR` gesetzt und nicht `null` ist. Lässt man den Doppelpunkt in den Ausdrücken weg, so wir nur überprüft, ob die Variable gesetzt ist.

5.2.7 Spezielle Variablen der Shell

Variable	Belegung
`$1..$9`	Kommandozeilenargumente (siehe 5.4, S. 69)
`$#`	Anzahl der Kommandozeilenargumente
`$*` und `$@`	Kommandozeilenargumente als Liste
`$?`	Rückgabewert des letzten Prozesses (siehe 5.1.2, S. 58)
`$$`	Prozess-ID des letzten Prozesses
`$!`	Prozess-ID des letzten Hintergrundprozesses

5.3 Flusskontrolle

5.3.1 Die `if`-Klausel

Mit Hilfe der `if`-Klausel können in der Shell Abfragen ausgewertet werden, die im Folgenden mit *Kondition* bezeichnet sind. Generell gilt für die Shell: Ist der Rückgabewert einer Kondition `0`, so ist der Wert wahr (*true*) und die Kommandos im `if`-Block werden ausgeführt. Alle anderen Werten sind unwahr (*false*) und der `if`-Block wird übersprungen. Gibt es einen `else`-Block, so wird dieser ausgeführt, andernfalls werden die Anweisungen nach dem Schlüsselwort `fi` ausgeführt. In vielen anderen Sprachen ist das genau anderes rum. Der Wert `1` bzw. alle Werte ungleich `0` werden als wahr bewertet, während `0` als unwahr bewertet wird.

Um weitere Verzweigungen zu ermöglichen, gibt es außerdem noch das Schlüsselwort `elif` (kurz für *elseif*). Dieses kann auf eine `if`-Klausel folgen und überprüft eine weitere Kondition,

falls die `if`-Bedingung negativ ausgewertet wurde. Die `elif`-Klausel entspricht im Prinzip einer `if`-Klausel innerhalb des `else`-Blockes, mit dem Vorteil, dass keine Verschachtelung nötig ist und so die Leserlichkeit besser ist. Die Syntax von `elif` entspricht der von `if`.

Syntax:

`if` *Kondition*
`then`
    Kommandos
`elif` *Kondition*
    Kommandos
`else`
    Kommandos
`fi`

Die Schlüsselwörter `if`, `then` und `fi` sind obligatorisch, während `elif` und `else` optional sind. Eine `if`-Klausel kann also auch ohne `else` und `elif` verwendet werden. Die Kondition einer `if`-Klausel kann beispielsweise der Vergleich von zwei Werten sein. Für Vergleiche und zum Testen von anderen Bedingungen bietet die Shell zwei syntaktische Varianten, die aber das selbe tun. Die erste Variante wird im nachfolgenden Beispiel verwendet, die zweite (das Kommando `test`) folgt im nächsten Abschnitt.

Die Abfrage der Kondition wird in nachfolgenden Variante in eckige Klammern gesetzt, wie im Beispiel gezeigt. Hier stehen die eckigen Klammern nicht für »optional«, sondern sind ein Teil der Syntax. Zu beachten ist, das die Leerzeichen innerhalb der eckigen Klammern relevant sind. Sie dürfen nicht weggelassen werden.

Beispiel:

```
#!/bin/bash

ROOT_USER=root

if [ $USER = $ROOT_USER ]
then
    echo $USER ist der Administrator.
else
    echo $USER ist nicht der Administrator.
fi
```

Im Beispiel wird mit Hilfe der Umgebungsvariable `USER` überprüft, ob der aktuelle Benutzer *root*, also der Administrator ist. Ist dies der Fall, so wird der Satz ›`root ist der Administrator.`‹ ausgegeben. Andernfalls wird ›*username* `ist nicht der Administrator.`‹ ausgegeben.

Die **Vergleichsoperatoren** für Zeichenketten und Zahlen sind in der Shell etwas anderes als man es vielleicht von anderen Programmiersprachen gewohnt ist. Das kann leicht zu Verwirrung und Fehlern führen. Der Vergleich von zwei Zeichenketten erfolgt, wie im Beispiel gezeigt mit nur einem Gleichheitszeichen, während dies in den meisten anderen Sprachen mit zwei Gleichheitszeichen erfolgt. Es kann hier leicht zur Verwechslung mit der Zuweisung eines Wertes kommen, die ebenfalls standardmäßig (auch in der Shell) mit einem Gleichheitszeichen erfolgt. In der

folgenden Tabelle sind die grundlegenden Unterschiede der Notation in der Shell mit anderen Programmiersprachen (wie Java, C oder Perl) gegenübergestellt.

Vergleichsoperationen

Typen	Operation	Shell	andere Sprachen
Zahlen	ungleich	`-ne`	`!=`
	gleich	`-eq`	`==`
	größer als	`-gt`	`>`
	größer gleich	`-ge`	`>=`
	kleiner als	`-lt`	`<`
	kleiner gleich	`-le`	`<=`
Zeichenketten	ungleich	`!=`	`!=`, `-ne`
	gleich	`=`	`==`, `-eq`

5.3.2 Das Kommando `test`

Die zweite Variante für die Abfrage von Bedingungen, ist das Kommando `test`. Die Verwendung von `test` entspricht der Verwendung der eckigen Klammern, die im vorigen Abschnitt eingeführt wurde. Die Schreibweise `[` *expr* `]` (s. o.) ist ein Synonym für `test`. Alle Bedingungen, die hier vorgestellt werden, können daher sowohl mit `test` als auch mit den eckigen Klammern verwendet werden.

Syntax:

`test` *Ausdruck*

Bei dem *Ausdruck* kann es sich um den Vergleich von zwei Werten (nummerisch oder alphabetisch), um Abfragen für Dateien oder um logische Operationen handeln (siehe Tabelle).

Beispiel:

```
if test $a = $b
then
    Kommandos
fi
```

Im Beispiel werden die Kommandos innerhalb des `if`-Blocks (*Kommandos*) ausgeführt, wenn der Rückgabewert von `test` gleich `0` ist, also `a` und `b` die gleiche Zeichenkette als Wert haben.

Operationen mit `test`

Operation	Ausdruck	Wahr wenn ...
Datei-operationen	`-e` *Datei*	eine Datei (i. w. S.) mit dem Namen *Datei* existiert.
	`-d` *Datei*	*Datei* ein Verzeichnis ist.
	`-f` *Datei*	*Datei* eine »einfach« Datei (*plain file*) ist.
	`-h` *Datei*	*Datei* ein symbolischer Link ist.
	`-r` *Datei*	eine Datei *Datei* existiert und lesbar ist.
	`-w` *Datei*	*Datei* existiert und beschreibbar ist.
Operationen auf Zeichenketten	`-n` *String*	die Länge der Zeichenkette *String* nicht 0 ist.
	`-z` *String*	die Länge der Zeichenkette *String* 0 ist.
	String	die Zeichenkette *String* nicht leer ist.
	s1 = *s2*	wenn die Zeichenketten *s1* und *s2* identisch sind.
	s1 `!=` *s2*	wenn emphs1 und *s2* nicht identisch sind.
Nummerische Operationen	*n1* `-eq` *n2*	die Zahlen *n1* und *n2* gleich sind.
	n1 `-ne` *n2*	*n1* und *n2* nicht identisch sind.
	n1 `-gt` *n2*	*n1* größer als *n2* ist.
	n1 `-ge` *n2*	*n1* größer als oder gleich *n2* ist.
	n1 `-lt` *n2*	*n1* kleiner als *n2* ist.
	n1 `-le` *n2*	*n1* kleiner als oder gleich *n2* ist.
	`!` *expr*	Ausdruck *expr* unwahr ist (Negation).
Logische Operationen	*expr1* `-a` *expr2*	die Ausdrücke *expr1* und *expr2* wahr sind.
	expr1 `-o` *expr2*	die Ausdrücke *expr1* oder *expr2* wahr ist.
	`(` *expr* `)`	Ausdruck *expr* wahr ist.

5.3.3 Die `while`-Schleife

Für die bedingte Wiederholung eines oder mehrerer Kommandos, bietet die Shell mehrere Varianten von Schleifen. Im Prinzip ist es beliebig, welche der Schleifen man für ein Problem auswählt, vor allem in Hinblick auf die Leserlichkeit des Kodes ist aber mal die eine Variante und mal eine andere vorzuziehen ist.

Die syntaktisch einfachste Schleife ist die `while`-Schleife. Ihre Syntax ist vergleichbar mit der Syntax der `if`-Klausel. Der Ablauf ist folgender, zunächst wird die Schleifenbedingung (*Kondition*) überprüft, wird sie als wahr ausgewertet, so werden die Kommandos im Schleifenblock (alles was zwischen `do` und `done` steht) ausgeführt. Anschließend wird überprüft, ob die Bedingung noch wahr ist. Ist dies der Fall, so wird wiederum der Schleifenblock ausgeführt. Wird die Bedingung negativ ausgewertet, so springt der Interpreter hinter das `done` und fährt dort mit der Ausführung fort. Die Kondition wird dabei wie bereits für die `if`-Klausel gezeigt angegeben.

Syntax:

```
while Kondition
do
    Kommandos
done
```

Beispiel:

```
#!/bin/bash
eingabe=

while [ $eingabe != $USER ]
do
```

```
    echo -n "Bitte Namen eingeben: "
    read eingabe
done
```

Im Beispiel wird zunächst die Variable `eingabe` auf eine leere Zeichenkette gesetzt. Dann wird im Schleifenkopf die Bedingung getestet. Da der Inhalt der Variable `eingabe` nicht dem Benutzternamen (Umgebungsvariable) entspricht, wird die Bedingung als wahr ausgewertet und der Interpreter beginnt mit der Abarbeitung des Schleifenblocks. Der Benutzter wird aufgefordert, einen Namen einzugeben. Dieser wird von der Kommandozeile mit Hilfe des Kommandos `read` (5.4.1, S. 70) gelesen. Anschließend wird wieder die Schleifenbedingung getestet. Ist sie noch wahr, so wird erneut der Schleifenblock ausgeführt.

5.3.4 Die `until`-Schleife

Die Syntax der `until`-Schleife entspricht der Syntax der `while`-Schleife mit dem Unterschied, dass sie solange ausgeführt wird, wie die Kondition als unwahr ausgewertet wird. Somit funktioniert die `until`-Schleife also genau *umgekehrt* wie die `while`-Schleife. Sobald die Kondition wahr ist, wird die Ausführung der Schleife abgebrochen und der Kode nach dem `done` ausgeführt.

Syntax:

`until` *Kondition*
`do`
    Kommandos
`done`

Beispiel:

```
#!/bin/bash
eingabe=

until [ $eingabe = $USER ]
do
    echo -n "Bitte Namen eingeben: "
    read eingabe
done
```

Das Beispiel entspricht dem Beispiel der `while`-Schleife. Zu beachten ist hier nur die Umkehrung der Bedingung.

5.3.5 Die `for`-Schleife

Bei der `for`-Schleife handelt es sich um eine sogenannte Zählschleife. Sie ist vor allem geeignet, wenn Listen abgearbeitet werden sollen oder wenn Zählvariablen verwendet werden sollen. Die `bash` bietet zwei Notationen für die `for`-Schleife. Die eine ist eher für die Abarbeitung von Listen geeignet, die andere eher für die Verwendung von Zählvariablen.

Soll eine Liste (siehe 5.3.6, S. 67) vollständig von Anfang bis Ende abgearbeitet werden, ist die folgende Notation am besten geeignet. In eckigen Klammern wird zunächst eine Laufvariable angegeben, der in jeder Schleifenrunde der aktuelle Wert zugewiesen wird. Auf die Variable folgt das Schlüsselwort `in`, gefolgt von einer Liste. In der ersten Schleifenrunde erhält die Variable den

ersten Wert aus der Liste, in der zweiten den zweiten usw. bis alle Elemente der Liste abgearbeitet sind.

Syntax:

```
for Variable in Liste
do
    Kommandos
done
```

Beispiel 1:

```
#!/bin/bash

for i in 1 2 3 4 5
do
    echo -n ${i}" "
done
echo
```

Der Schleifenblock im Beispiel wird fünf mal ausgeführt. Der Variable `i` wird in jedem Schleifendurchlauf der jeweilige Wert aus der Liste zugewiesen. Nach Aufruf des Skriptes erhält man die Ausgabe:

```
$
1 2 3 4 5
$
```

Die Liste ist dabei sehr vielseitig. Man kann sie direkt angeben, wie im Beispiel gezeigt oder sie vorher in einer Variable speichern. Es kann sich hierbei aber auch um Ausgaben eines Kommandos oder Programmes handeln, die iterativ abgearbeitet werden.

Beispiel 2:

```
#!/bin/bash

for i in *.jpg
do
    convert $i -size 230x173 -resize 230x173 tn_${i}
done
```

Im Beispiel werden alle `*.jpg` Dateien im aktuellen Verzeichnis mit dem Kommando `convert` auf die Größe `230x173` skaliert und die konvertierte Version als `tn_*.jpg` gespeichert.

Um ein oder mehrere Kommandos x-Mal wiederholt auszuführen, bietet sich die Zählschleife an. Die Syntax ist etwas gewöhnungsbedürftig, aber in vielen Programmiersprachen verbreitet. Im Schleifenkopf wird eine Zählvariable initialisiert, dann die Abbruchbedingung angegeben und schließlich die Aktualisierung des Zählers. Die Initialisierung wird jeweils nur vor dem ersten Schleifendurchlauf durchgeführt. Anschließend wird die Abbruchbedingung geprüft und der Schleifenblock abgearbeitet. Nach Abarbeiten des Schleifenblocks wird der Zähler aktualisiert,

wieder die Abbruchbedingung überprüft und ggf. die Kommandos im Schleifenblock erneut ausgeführt usw. bis die Abbruchbedingung erfüllt ist (hier die Bedingung als unwahr ausgewertet wird).

Syntax:

```
for ((Initialisierung; Abbruchbededingung; Zählschritt))
do
    Kommandos
done
```

Beispiel:

```
#!/bin/bash

for (($i=0; $i<10; $i++))
do
    echo "Dies ist die ${i}te Wiederholung."
done
```

Im Beispiel wird der Satz 10x ausgegeben, wobei jeweils die aktuelle Zahl aus $\{0 \ldots 9\}$ eingefügt wird.

5.3.6 Listen

Die Elemente einer Liste werden durch das Standard-Eingabe-Separatorzeichen (`IFS` - *input field separator*) getrennt. Dies ist in der `bash` standardmäßig das Leerzeichen. Weist man einer Variable beispielsweise einen Satz zu, so würden die Wörter des Satzes als einzelne Listenelemente betrachtet werden. Man kann Listenelemente auch durch andere Zeichen trennen. Dazu muss man der Variable `IFS` einen neuen Wert zuweisen, beispielsweise den Doppelpunkt, wenn man die Elemente des `PATH` trennen will.

Beispiel:

```
#!/bin/bash

satz="Vor langen Jahren lebte ein Mann im Osten."

for wort in $satz
do
    echo $wort
done
```

Durch den Aufruf der Skriptes erhält man folgende Ausgabe:

```
Vor
langen
Jahren
lebte
ein
Mann
im
Osten.
```

5.3.7 Eindimensionale Arrays

Neuere Versionen der `bash` unterstützen neben den einfachen Listen auch eindimensionale *Arrays* (Reihe, Aufstellung). Arrays sind Datenstrukturen, die auch in vielen höheren Programmiersprachen Verwendung finden. Im Prinzip ist ein Array das gleiche wie eine Liste, mit dem Unterschied, dass man explizit auf die Elemente zugreifen kann. In höheren Programmiersprachen gibt es meist zwei Varianten von Arrays, die einfachen Arrays, deren Elementen fortlaufende Nummern zugeordnet sind, über die sie angesprochen werden können und sogenannte assoziative Arrays (auch *Hash*), deren Elementen Zeichenketten zugeordnet sind.

Die Syntax der Verwendung von Arrays ist meist ähnlich. Um auf Elemente zuzugreifen muss erst der Name und dann der zugordnete Index in eckigen Klammern angegeben werden. Eine Besonderheit der `bash` ist, dass das ganze in geschweifte Klammern gesetzt werden muss.

Syntax:

`${`*arrayname*`[`*n*`]}`
bezeichnet das *n*te-Element des Arrays *arrayname*.

Um in der `bash` Arrays zu verwenden, müssen diese zunächst deklariert werden. Dazu gibt es zwei Möglichkeiten: Zum einen kann dies direkt durch die Zuweisung der Elemente geschehn, zum anderen durch das Schlüsselwort `declare -a`.

Syntax:

arrayname`[`*n*`]=`*Wert*

alternativ:

`declare -a` *arrayname*

Beispiel:

```
declare -a files=( `ls *.txt` )

for ((i=0; i<${#files[@]}; i++))
do
     cat ${files[i]}
done
```

Im Beispiel werden dem Array `files` alle Dateien mit dem Suffix `txt` zugewiesen. In der `for`-Schleife werden die Elemente des Arrays `cat` übergeben. `${#files[@]}` gibt die Anzahl der Elemente des Arrays zurück.

5.3.8 Die Steuerelemente `break` und `continue`

Zur Steuerung der Schleifen gibt es außerdem noch die Kommandos `break` und `continue`. Mit `break` kann eine Schleife beendet werden, bevor die Abbruchbedingung im Schleifenkopf erfüllt ist. Dies dient vor allem der Leserlichkeit, aber auch der Performanz, wenn beispielsweise eine Bedingung, die zum Beenden der Schleifen führen soll, nicht in jedem Durchlauf abfragen werden muss (z. B. wenn sie in einer verschachtelten `if`-Abfrage erfolgt).

Das Kommando `continue` bewirkt, dass alle folgenden Kommandos in einem Schleifedurchlauf nicht mehr ausgeführt werden, sondern der nächste Durchlauf gestartet wird. Dies ist sinnvoll wenn ein Teil der Kommandos unter bestimmten Bedingungen nicht ausgeführt werden soll, die Schleife aber weiter laufen soll und die Kommandos unter Umständen in der nächsten Runde wieder ausgeführt werden sollen.

Beispiel:

```
for (($i=0; $i<10; $i++))
do
    if [ $i -lt 5 ]
    then
        continue;
    fi
    echo "Zahlen groesser 5: ${i}te."
done
```

5.3.9 Die case-Anweisungen

Die `case`-Anweisungen dient als »Schalter«. Sie ist vergleichbar mit dem `switch` in C und Java. Der übergebene *Ausdruck* wird mit den *Mustern* verglichen. Das Muster kann dabei ein einzelnes Zeichen, eine Zeichenkette oder eine Gruppe von Zeichen (*Glob*) darstellen. Wenn der Ausdruck auf eines der Muster passt, werden die nachfolgenden Kommandos ausgeführt. Passt der Ausdruck auf mehrere Muster, so werden jeweils nur die Kommandos nach dem ersten passende Muster ausgewertet.

Syntax:

```
case Ausdruck in
    Muster1)
        Kommandos;;
    Muster2)
        Kommandos;;
esac
```

Beispiel:

```
case $eingabe in
    $name ) "Der Inhalt der Variable name"
    [0-9] ) echo "Ein Kleinbuchstabe";;
    F ) echo "Der Buchstabe F";;
    [a-zA-Z] ) echo "Ein Grossbuchstabe";;
esac
```

5.4 Argumente und Eingaben

Man kann Shell-Skripte wie den meisten Programmen beim Starten Argumente übergeben. Diese werden in den Variablen `$1 ... $9` (siehe 5.2.7, S. 61) gespeichert. Wenn mehr als 9 Argumente

in der Kommandozeile übergeben werden, sind sie nur mit Hilfe von `shift` (siehe 5.4.2, S. 70) erreichbar. Die Anzahl der Kommandozeilenargumente ist in der Variable `$#` gespeichert. In der Variable `$0` ist der Name des Skriptes gespeichert (vgl. C: `argv[0]`, in Java dagegen ist `argv[0]` das erste übergebene Argument).

Beispiel:

```
#!/bin/bash

echo "Das erste Argument ist $1"
```

Die Kommandozeilenargumente können auch als Liste über die Variablen `$*` und `$@` angesprochen werden. Der Unterschied zwischen beiden ist, dass `$*` sich wie eine »normale« Liste (`"$1 $2 ..."`) verhält, während `$@` jedes Argument einzeln in Anführungszeichen setzt (`"$1" "$2" ...`), was aber in den meisten Fällen keinen Unterschied macht.

5.4.1 Das Kommando `read`

Um Skripte interaktiv zu nutzen, kann man Eingaben mit dem Kommando `read` lesen. `read` liest jeweils eine Zeile von der Standardeingabe und speichert sie in der übergebenen Variable.

Syntax:

```
read var
```

Ein Beispiel für die Verwendung von `read` ist unter 5.3.3, S. 64 gegeben.

5.4.2 Das Kommando `shift`

Mit `shift` kann man die Elemente der Argumentenliste verschieben. Dabei löscht `shift` immer das erste Element der Liste und verschiebt alle nachfolgenden Elemente eine Position nach vorne.

Beispiel:

```
#!/bin/bash

until [ -z $1 ]
do
    echo $1
    shift
done
```

Im Beispiel werden die Kommandozeilenargumente ausgegeben. Zunächst wird die Abbruchbedingung der `until`-Schleife überprüft (`-z $1`, wahr wenn `$1` eine Zeichenkette der Länge `0` ist. Ist dies nicht der Fall, so wird `$1` ausgegeben. Mit `shift` wird das Argument aus der Liste entfernt und der Wert von `$2` nach `$1` verschoben. Das geht so weiter bis die Abbruchbedingung als wahr ausgewertet wird.

5.4.3 Backquote (Gravis, Accent Grave)

Ausdrücke, die in Backquotes gesetzt sind, werden von der Shell ausgewertet und durch den Inhalt der Standardausgabe des Ausdrucks ersetzt.

Syntax:

`` `Kommando` ``

Beispiel:

```
#!/bin/bash

path=/var/tmp/foo
name=`basename $path`

echo "Der Dateiname ist: $name"
```

Die Ausgabe des Skriptes lautet ›`Der Dateiname ist: foo`‹. Das gleiche Ergebnis erhält man wenn man den betreffenden Ausdruck mit `$(`*Kommando*`)` angibt.

5.5 Funktionen definieren

Wie in den meisten anderen Sprachen auch, können Funktionen definiert werden, die in einem gesonderten Block stehen. Das ist sinnvoll, wenn man eine Abfolge von Kommandos an verschiedenen Stellen in einem Skript braucht. Indem man die Kommandos in eine Funktion auslagert, kann man sie immer wieder aufrufen, ohne dass man den gleichen Quellkode mehrfach schreiben muss.

Funktionen können im Skript wie Kommandos verwendet werden. Man kann sie mit einem Rückgabewert ausstatten, der als Exit-Status abgefragt werden kann. Im Unterschied zu einem Kommando, öffnet eine Funktion beim Aufruf keine Subshell. Das hat zur Folge, dass der Wert einer Variable, die innerhalb eines Funktionsblocks gesetzt wird, auch ausserhalb der Funktion zu sehen ist.

Syntax:

```
funktionsname () {
    Kommandos
    [return n]
}
```

Der Rückgabewert ist hierbei optional (daher in `[ ]` angegeben). Es handelt sich dabei um einen nummerischen Wert wie bei `exit` (vgl. 5.1.2, S. 58). Der Aufruf der Funktion erfolgt dann wie der Aufruf eines Kommandos mit `funktionsname` *Argumente*.

5.6 Einige nützliche Kommandos

Die Shell bietet einige nützliche Kommandos, die z. T. bereits in den Beispielen verwendet werden, beispielsweise das Kommando `basname`. Dieses und weitere Kommandos sind in der folgenden Tabelle zusammengefasst.

Kommando	Beschreibung
`basename` *Pfad*	Letzter Teil einer Pfandangabe
`dirname` *Pfad*	Pfad ohne letzten Teil
`wait [`*pid*`]`	Warten auf Hintergrundprozess *pid*, wenn keine Prozess-ID übergeben wird, wartet die Shell auf alle Hintergrundprozesse
`sleep` *sekunden*	Pausiert für *Sekunden*
`trap[`*Kommando Sig*`]`	Führt *Kommando* aus, wenn das Signal *Sig* von der Shell geschickt wird
`cmp` *Datei...*	Byteweise Vergleichen von Dateien
`diff` *Datei ...*	Vergleichen von Dateien
`paste` *Datei...*	Spaltenweise Zusammenfügen von Dateien
`seq` *von bis*	Sequenz von Zahlen im übergebenen Range
`file` *Datei*	Gibt den Dateityp aus

5.7 Bearbeiten von Zeichenketten und Texten

5.7.1 Transliteration

Die Transliteration dient dazu Zeichen oder Gruppen von Zeichen aus dem Eingabestrom zu ersetzen und bearbeiteten Strom wieder auszugeben.

Syntax:

```
tr [Optionen] set1 [set2]
```

Beispiel:

```
tr [:digit:] _ < Datei1 > Datei2
```

Im Beispiel werden alle Ziffern aus `Datei1` durch einen Unterstrich ersetzt und das ganze in `Datei2` gespeichert.

5.7.2 `sed` und `awk`

Wenn man in der Shell oder innerhalb von Shell-Skripten komplexere Operationen auf Zeichenketten ausführen will, so kann man dazu die Skriptsprachen `sed` (*Stream EDitor*, 1974, Lee MacMahon) und `awk` (1977, Aho, Weinberger und Kernighan) verwenden. Beide sind sehr mächtig, vor allem da sie die Verwendung von Regulären Ausdrücken ermöglichen, aber insbesondere `awk` ist nicht unbedingt leicht zu handhaben. In vielen Fällen greift man eher auf andere Skriptsprachen wie beispielsweise Perl zurück. Daher werden `sed` und `awk` hier auch nur andeutungsweise vorgestellt.

`sed`
`sed` ist wie bereits gesagt eine Abkürzung für *Stream EDitor*, im Gegensatz zu anderen Editoren ist `sed` aber nicht interaktiv, sondern arbeitet eher wie eine Pipeline. Die Kommandos werden beim Aufruf in Anführungszeichen mit einem Eingabe Strom übergeben. Der

bearbeitete Strom wird dann in die Standardausgabe geschrieben. Die Syntax für das Suchen/Ersetzen wie auch die Regulären Ausdrücke entsprechen der Transliteration (siehe 5.7.1, S. 72) und der Syntax des Editors `ex` (siehe 3.2.7, S. 49).

Beispiel:

`sed "s/`*Muster*`/`*Ersetzung*`/" `*Datei*

`awk`

`awk` war usrprünglich als Erweiterung von `grep` und `sed` gedacht, wurde aber zu einer umfangreichen Skriptsprache weiterentwickelt. Unter Linux steht meist die freie Version `gawk` von der `FSF` zur Verfügung, die aber auch mit dem Kommando `awk` aufgerufen wird.

`awk` ist deutlich umfangreicher und komplexer als `sed`. Es bietet vordefinierte Variablen, Konstrukte zur Flusskontrolle (`if`-Klausel und Schleifen) und Arrays und viele *built-in*-Funktionen wie `print`, `length`, `split` usw. Die Kommandos, die `awk` ausführen soll, werden in Hochkommata gesetzt. Die Hochkommata schließen dabei das gesamte Skript ein, während in geschweifte Klammern einzelne Blöcke definiert werden können. Jedes Kommando in Klammern, wird dabei auf jede Zeile aus dem Eingabestrom angewendet. Außerdem gibt es noch die Bedingungen `BEGIN` und `END` Der `BEGIN`-Block wird nur vorm Einlesen der ersten Eingabe ausgeführt, der `END`-Block nach dem alle Zeilen aus dem Eingabestrom gelesen wurden.

Listen/Arrays	*Name*`[`*Index*`]=`*Wert* Sowohl *Index* aus auch *Wert* können dabei ganze Zahlen (Integer) oder Zeichenketten (Strings) sein.
`if`-Klausel	`if [`*Kondition*`]{`*Kommandos*`}` `else{`*Kommandos*`}`

`for`-Schleife	`for i in `*Array*`{`*Kommandos*`}`
	oder
	`for (`*Initialisierung*`;`*Abbruchbedingung*`;`*Zählschritt*`) {`
	Kommandos`}`

`while`-Schleife	`while (`*Kondition*`) {`*Kommandos*`}` `do{`*Kommandos*`}while (`*Kondition*`)`
Vord. Variablen	`NF` – Anzahl der Felder in einer Zeile `NR` – Anzahl der eingelesenen Zeilen `FS` – Feld-Separatorzeichen (default: Leerzeichen) `FILENAME` – Name der Eingabedatei `$1, $2...` – Elemente einer Zeile
Funktionen	`substr` – Teilausdruck `index` – Position einer Zeichenkette innerhalb einer anderen Zeichenkette `print` – einfache Ausgabe `sprintf` – formatierte Ausgabe `length` – Länge einer Zeichenkette `split` – Zersplitten von Zeichenketten `sqrt` – Quadratwurzel `log` – Logarithmieren `exp` – Exponieren

Ein kurzes Beispiel für die Benutzung von `awk` innerhalb eines Shell-Skriptes ist unter 5.1.2, S. 57 gegeben.

5.8 Debuggen von Shell-Skripten

Die Fehlersuche in Programmen und Skripten wird häufig als *Debuggen* bezeichnet und bezieht sich auf das umgangssprachliche Wort *Bug* für Fehler (zum Ursprung dieser Bezeichnung gibt es einige Legenden, einige davon haben etwas mit Käfern zu tun...). Da für die Shell leider keine Debugger wie für Perl, C, Java und andere Sprachen existiert, muss man auf einfachere Mittel zurückgreifen. Um Fehler in Shell-Skript zu finden, kann man dem Kommando `set` die Optionen `-x/+x` oder `-n` übergeben. `set -n` bewirkt, dass für alle nachfolgenden Kommandos nur die Syntax überprüft wird, die Kommandos aber nicht ausgeführt werden. Durch die `x`-Option werden alle ausgeführten Kommandos, die zwischen `set -x` und `set +x` stehen, auf die Shell ausgegeben.

Ansonsten kann es hilfreich sein, sich Kommandos bevor man sie anwendet mit `echo` ausgeben zu lassen, das gleiche gilt für die Überprüfung des Inhaltes von Variablen.

6 Nützliche Links

1. **Bioinformatik Göttingen:**
   med. BI (UKG) – http://www.bioinf.med.uni-goettingen.de/
   biol. BI (IMG) – http://www.gobics.de/

2. **Linux & Shell:**
   GNU – http://www.gnu.org/
   Numerik Göttingen – http://www.num.math.uni-goettingen.de/Lehre/Lehrmaterial/
   Linuxfibel – http://www.linuxfibel.de/
   Wegweiser zur Installation & Konfiguration – http://www.oreilly.de/german/freebooks/linux_install/
   Linux Documentation Project – http://tldp.org/
   A Beginner's Handbook – http://www.freeos.com/guides/lsst/
   Advanced Bash-Scripting Guide – http://www.tldp.org/LDP/abs/html/
   AWK Tutorial – http://www.vectorsite.net/tsawk.html

3. **Win32:**
   GWDG (SecureShell-Clients) – http://www.gwdg.de/
   Cygwin – http://www.cygwin.com/
   ActivePerl – http://www.activestate.com/

4. **Perl:**
   Perl Doc – http://www.perldoc.perl.org/
   CPAN – http://www.CPAN.org/
   BioPerl – http://doc.bioperl.org/
   Selfhtml – http://selfhtml.teamone.de/
   Perl Monks – http://www.perlmonks.org/ O'Reillys Perl-Seite – http://www.perl.com/

5. **Bioinformatik Tools:**
   NCBI – http://www.ncbi.nlm.nih.gov/
   EMBL – http://www.embl-heidelberg.de/
   EBI – http://www.ebi.ac.uk/
   TIGR – http://www.tigr.org/

6. **Emacs:**
   GNU emacs manual – http://www.gnu.org/software/emacs/manual/

7. **vi:**
   An Introduction to Display Editing with Vi – http://docs.freebsd.org/44doc/usd/12.vi/paper.html
   vi Lovers Home Page – http://thomer.com/vi/vi.html
   The vi Editor – http://unix.t-a-y-l-o-r.com/Vi.html
   vi Reference Card – http://vh224401.truman.edu/ dbindner/mirror/
   vim-Homepage – http://www.vim.org/
   vimdoc – http://vim.sf.net/
   vim Reference Card – http://tnerual.eriogerg.free.fr/vim.html
   http://www.ungerhu.com/jxh/vi.html
   http://www.lagmonster.org/docs/vi.html

8. **Zum selbst Helfen:**
   WIKIPEDIA – http://de.wikipedia.org/
   fireball – http://www.fireball.de/
   google! – http://www.google.de/

7 Index