Michael Kofler
Jürgen Plate

Linux für Studenten

ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Linux-Shell

4

ÜBERBLICK

In diesem Kapitel geht es um ein Programm, das bei der Anwendung einer grafischen Benutzerschnittstelle anfangs oft unbemerkt bleibt: der Kommandointerpreter, „Shell“ genannt, der ein mächtiges Werkzeug zur Automatisierung regelmäßig wiederkehrender Aufgaben ist. Wir haben die Shell auch schon ganz kurz im Abschnitt „Textkonsolen und Shell-Fenster“ erwähnt. Was dort gesagt wurde, gilt natürlich besonders in diesem Kapitel – insbesondere auch die Tastenkürzel.

Die Shell bietet neben ihren Diensten als interaktiver Kommandointerpreter eine mächtige Sprache zum Erstellen so genannter Shell-Skripten (*shell scripts*). So wie bei der Programmiersprache *C* der eigentliche Sprachumfang recht gering und die Mächtigkeit der Sprache durch die zahllosen Bibliotheksfunktionen gegeben ist, „lebt“ die Shell v. a. von den vielen, vielen Linux-Kommandos, die alle kommandozeilenorientiert aufgerufen werden können. Dies ist auch das Ziel dieses Kapitels. Worauf deshalb nicht besonders eingegangen wird, sind die interaktiven Möglichkeiten der Shell (wie z. B. History-Funktion, Aliase, Zeileneditor etc.). Auch beschränkt sich das Buch auf die „Standard-Shell“ von Linux, die *bash*. Die erste Linux-Shell, die nach ihrem Entwickler, Steven R. Bourne, benannte *Bourne-Shell* diente als Vorbild, weshalb „bash“ auch ein Acronym für „Bourne again shell“ ist (wenn Sie das laut aussprechen, wird auch das verborgene Wortspiel deutlich).

Der Vorteil bei der Shell-Programmierung ist auch die Interaktivität der Shell. Sie können alle (!) Sprachkonstruktionen auch auf der Kommandozeile ausführen lassen und natürlich können Sie durch abwechselndes Probieren und Erweitern Ihr Shell-Programm Schritt für Schritt entwickeln – fast so wie früher mit *BASIC* auf dem C64.

4.1 Aufgaben der Shell

Die Hauptaufgaben bzw. Anwendungsmöglichkeiten der Shell lassen sich in folgenden Punkten zusammenfassen:

- Kommandozeile vom Terminal annehmen, das Kommando entschlüsseln und in seine Komponenten (Kommandoname, Optionen, Parameter) aufteilen
- Gewünschtes Programm suchen und starten; der Suchweg wird durch eine Shell-Variable `PATH` vorgegeben.
  Mittels `which Programmname` kann man den Pfad zu einem Programm feststellen bzw. untersuchen, ob sich das Programm im Pfad befindet. Das Kommando `type` arbeitet ähnlich, es findet zusätzlich auch alle internen Kommandos der Shell.
- Steuerung der Ein- und Ausgabe des Terminals; UNIX (und so auch Linux) ist als Dialogsystem konzipiert und war ursprünglich für den Betrieb mit Fernschreibern (*Teletype*) und als Datensichtgerät konzipiert. Diese so genannten Terminals findet man noch in den *logischen* Terminals, z. B.:
  - den Konsolen, die dem Bildschirm und der Tastatur des PCs zugeordnet sind,
  - den Pseudo-Terminals, die einer Telnet-Verbindung zugeordnet werden oder
  - den X-Terminal-Fenstern auf der grafischen Benutzeroberfläche.

Linux behandelt alle Ein- und Ausgabemedien als Dateien, auch Geräte wie Terminals, Drucker, serielle Schnittstellen usw. Daher gibt es drei Standarddateien, die dem aktuellen Terminal zugeordnet sind:

- Standardeingabe (Tastatur, `stdin`)
- Standardausgabe (Bildschirm, `stdout`)
- Standardfehlerausgabe (Bildschirm, `stderr`)

Die Shell kann angewiesen werden, die Ein- und Ausgaben auf Geräte oder auf Dateien umzuleiten. Man kann also beispielsweise Shell-Eingaben auch aus einer Datei lesen oder die Ausgabe der gestarteten Programme in einer Datei speichern.

- Ersetzung von Sonderzeichen und Befehlssubstitution; Kommandoeingaben können durch Sonderzeichen erweitert und der Kommandostring kann selbst durch die Shell expandiert werden.
- Ablaufsteuerung; die Shell beherrscht viele Strukturen, wie man sie von Programmiersprachen her kennt (Test, bedingte Anweisung, Schleifen, Variablen, Rechenanweisungen).
- Erzeugen so genannter Kind- und Hintergrundprozesse. Es besteht die Möglichkeit, Programme „im Hintergrund“ zu starten und am Bildschirm weiterzuarbeiten, während das Programm läuft.

4.2 Grundlegende Eigenschaften der Shell

Folgende Punkte mögen – in zugegeben knapper Form – zusammenfassen, mit welcher Art von Werkzeug man es bei der Shell zu tun hat und nach welchen Prinzipien sie arbeitet:

1. Sobald die Shell bereit ist, Kommando-Eingaben anzunehmen, meldet sie sich mit einem Bereitschaftszeichen (*Prompt*). Im einfachsten Fall ist dies – wie Sie im dritten Kapitel gesehen haben – ein `$`-Zeichenfür den Normaluser und das `#` für den Superuser. Das Zeichen `>` erscheint, wenn noch weitere Eingaben erwartet werden. Wie vieles in der Shell kann der Prompt beliebig modifiziert werden.
2. Die Shell ist ein ganz „normales“ Programm, das von der Standardeingabe (Tastatur) liest und auf die Standardausgabe (Bildschirm) ausgibt. Wenn das Dateiende erreicht wird (*End of File*, `EOF`), z. B. durch Eingabe von Strg - D, terminiert sie. Handelt es sich um die Login-Shell, erfolgt ein Logoff des Benutzers.
3. Die Shell kann wie ein Programm als Subshell aufgerufen werden (Schachtelung). Dies wird beispielsweise benötigt, um Shell-Programme (*shell scripts*) zu testen.
4. Es gibt zwei Möglichkeiten, den Pfad für ein Kommando festzulegen:
   - direkt als absoluter oder relativer Pfadname,
   - indirekt über den Pfad, der in der Shell-Variablen `PATH` gespeichert ist.

Zur Erinnerung: In der Literatur (auch beim Kommando `man`) werden die Kommandos in Kurzform beschrieben. Dabei werden Werte, die optional sind, in eckige Klam-

mern gesetzt. Die (meist aus einem Buchstaben bestehenden) Optionen werden als Kette aufgelistet, z. B. `[-aeGhlz]`. Das bedeutet nichts anderes, als dass eine beliebige Kombination dieser Optionen möglich ist (ob sie sinnvoll ist, wird dabei nicht berücksichtigt).

Ein Kommando wird erst ausgeführt, wenn der Benutzer am Ende der Kommandozeile die ↵-Taste drückt. Eine genauere Kenntnis des dann folgenden Ablaufs erlaubt es zu verstehen, warum etwas nicht zu dem Ergebnis führt, das man erwartet hat. Daher sollen hier die einzelnen Schritte dieses Arbeitsablaufs kurz beschrieben werden (einige der Kommando-Trenner, Jokerzeichen und Variablen werden später behandelt):

1. Die Shell liest die Eingabe bis zum ersten Kommando-Trenner und stellt fest, ob Variablenzuweisungen erfolgen oder die Ein-/Ausgabe umgelenkt werden soll.
2. Die Shell zerlegt die Kommandozeile in Argumente. Sie trennt die einzelnen Argumente durch eines der Zeichen, die in der Shell-Variablen `IFS` (*Internal Field Separator*) stehen, normalerweise Leerzeichen, Tabs und Newline-Zeichen.
3. Variablenreferenzen, die in der Kommandozeile stehen, werden durch ihre Werte ersetzt.
4. Kommandos, die in `` `…` `` oder bei der Bash in `$(…)` stehen, werden ausgeführt und durch ihre Ausgabe ersetzt.
5. `stdin`, `stdout` und `stderr` werden auf ihre „Zieldateien“ umgelenkt (s. unten).
6. Falls in der Kommandozeile noch Zuweisungen an Variablen stehen, werden diese ausgeführt.
7. Die Shell sucht nach Jokerzeichen (Metazeichen) und ersetzt diese durch passende Dateinamen.
8. Die Shell führt das Kommando aus.

4.3 Ein- und Ausgabeumleitung

Die drei dem Terminal zugeordneten Dateikanäle (Pseudodateien) `stdin`, `stdout` und `stderr` können jederzeit auf Dateien oder Geräte umgeleitet werden. Für C-Programmierer: Den drei Standarddateien sind die Filehandles `0` (`stdin`), `1` (`stdout`) und `2` (`stderr`) zugeordnet.

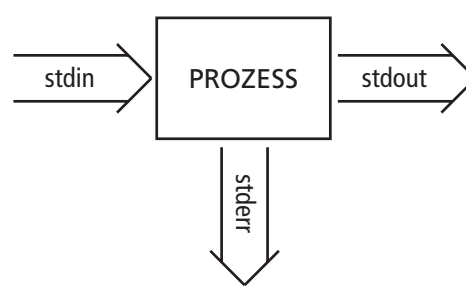

Abbildung 4.1: Standarddateien unter Linux

4.3.1 Eingabeumleitung

Das Programm liest nun nicht mehr von der Tastatur (`stdin`), sondern aus einer Datei, bis das Dateiende erreicht ist.

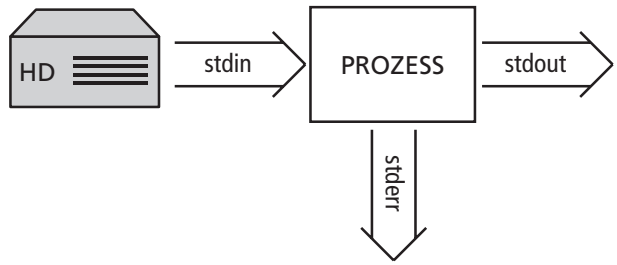

Abbildung 4.2: Umleitung der Standardeingabe

Die Eingabeumleitung erfolgt durch das Zeichen „<“, gefolgt von einem Dateinamen:

```
Kommando < Dateiname
```

Statt z. B. beim `mailx`-Kommando den Text direkt einzugeben, kann eine Datei an den Empfänger gesendet werden:

```
user$ mailx -s "Automatische Nachricht" linus < Nachricht
```

4.3.2 Ausgabeumleitung

Die Ausgabe des Programms wird nicht auf dem Bildschirm (`stdout`) ausgegeben, sondern in eine Datei geschrieben. Die Ausgabeumleitung erfolgt durch das Zeichen „>“, gefolgt von einem Dateinamen.

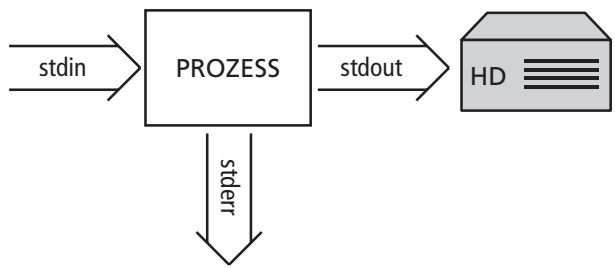

Abbildung 4.3: Umleitung der Standardausgabe

Falls die Datei noch nicht vorhanden war, wird sie automatisch angelegt. Falls die Datei schon vorhanden ist, wird sie überschrieben, d. h. es wird immer ab dem Dateianfang geschrieben:

```
Kommando > Dateiname
```

Beispiel: Ausgabe der Verzeichnisbelegung in eine Datei namens `info`:

```
user$ ls -l > info
```

Fehlermeldungen erscheinen nach wie vor auf dem Bildschirm. Die Umleitung der Fehlerausgabe `stderr` erfolgt ebenso wie die Ausgabeumleitung, jedoch wird hier die Zeichenfolge `2>` verwendet, da `stderr` die Handle-Nummer `2` hat.

```
Kommando 2> Fehlerdatei
```

(Die Umleitung der Standardausgabe ist nur die Kurzform von *`Kommando 1> Datei-name`*). Natürlich ist eine beliebige Kombination von Ein- und Ausgabeumleitung möglich, z. B.

```
Kommando < Eingabedatei > Ausgabedatei 2> Fehlerdatei
```

Es ist darüber hinaus auch möglich, die Ausgabe des Programms an eine bereits vorhandene Datei anzuhängen. Dazu wird das „>“ doppelt geschrieben.

```
Kommando >> Sammeldatei
```

Dazu einige Beispiele:
Dateiliste und aktive Benutzer in eine Datei schreiben:

```
user$ ls -l > liste
user$ who >> liste
```

Die Umleitung von Ein- und Ausgabe lässt sich auch unterdrücken. Für die Ausgabe schreibt man dann

```
Kommando > /dev/null
```

oder für die Fehlerausgabe

```
Kommando 2> /dev/null
```

Beides lässt sich auch kombinieren:

```
Kommando > Ergebnisdatei 2> /dev/null
```

Will man ein Programm mit einem beliebigen Eingabedatenstrom versorgen, schreibt man

```
Kommando < /dev/zero
```

Die Umleitung von `stdout` und `stderr` in dieselbe Datei würde prinzipiell eine zweimalige Angabe der Datei (eventuell mit einem langen Pfad) erfordern. Für die Standarddateien werden in solchen Fällen spezielle Platzhalter verwendet:

```
Kommando > Ausgabe 2>&1
```

Es gibt drei Platzhalter für die Standarddateien: `&0` für die Standard*eingabe*, `&1` für die Standard*ausgabe* und `&2` für die Standard*fehlerausgabe*. Es ist sogar möglich, in Shell-Skripten anonyme Dateien zu öffnen, die nur über die Dateinummer (`&4`, `&5`, . . .) referenziert werden.

Hinweis

Bei der Ein- und Ausgabeumleitung darf nicht die gleiche Datei für Eingabe und Ausgabe verwendet werden – sie wird sonst verwüstet. So liefert beispielsweise `sort < dat > dat` keineswegs eine sortierte Datei `dat`, sondern die Datei wird gelöscht.

4.3.3 Pipes

Eine Pipe verbindet zwei Kommandos über einen temporären Puffer, d. h. die Ausgabe vom ersten Programm wird als Eingabe vom zweiten Programm verwendet. Alles, was das erste Programm in den Puffer schreibt, wird in der gleichen Reihenfolge vom zweiten Programm gelesen. Pufferung und Synchronisation werden vom Betriebssystem vorgenommen. Der Ablauf beider Prozesse kann verschränkt erfolgen. In einer Kommandofolge können mehrere Pipes vorkommen. Der Pipe-Mechanismus wird durch das Zeichen | (senkrechter Strich) aktiviert:

```
Kommando1 | Kommando2
```

Beispiel: Ausgabe der Dateien eines Verzeichnisses mit der Möglichkeit zu blättern:

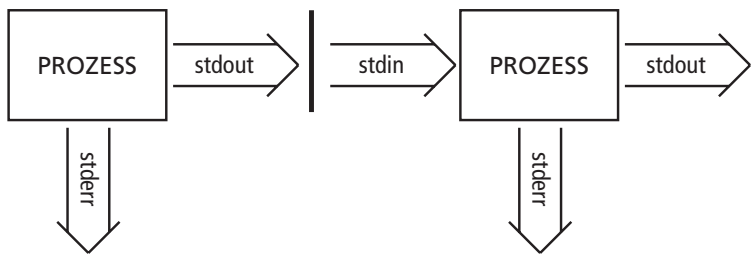

Abbildung 4.4: Pipe: Die Ausgabe eines Programms wird in die Eingabe des nächsten geleitet.

Weil die Zeile *Kommando < Eingabedatei > Ausgabedatei 2> Fehlerdatei* optisch etwas seltsam wirkt, bemühen viele Shell-Programmierer – insbesondere in Verbindung mit den Pipes – das `cat`-Kommando. „cat“ steht für „concatenate“ (= zufügen, aneinderhängen) und genau das macht das Kommando. Es gibt die Inhalte der Datei(en), die als Parameter angegeben wurden, auf der Standardausgabe aus. Aus dem Kommando oben wird dann:

```
cat Eingabedatei | Kommando > Ausgabedatei 2> Fehlerdatei
```

Natürlich können auch mehrere Kommandos hintereinander durch Pipes verbunden werden:

```
Kommando1 | Kommando2 | Kommando3 | Kommando4 | ...
```

Solche Kommandofolgen werden auch als *Filter* bezeichnet. Einige nützliche Filter sind in jedem Linux-System verfügbar. Zum Beispiel:

- `head [-n] [datei(en)]`
  Ausgabe der ersten `n` Zeilen aus den angegebenen Dateien. Voreinstellung ist zehn Zeilen. Wird keine Datei angegeben, liest `head` von der Standardeingabe.
- `tail [-/+n] [bc[f|r]] [datei]`
  Ausgabe der letzten `n` Zeilen einer Datei. Voreinstellung für `n` ist 10. Wird keine Datei angegeben, liest `tail` von der Standardeingabe. Optionen:

  `+n` ab der `n`-ten Zeile ausgeben

  `-n` die letzten `n` Zeilen ausgeben. Wird hinter die Zahl `n` ein `b` gesetzt (z. B. `-15b`), werden nicht `n` Zeilen, sondern `n` Blöcke ausgegeben. Wird hinter die Zahl `n` ein `c` gesetzt (z. B. `-200c`), werden nicht `n` Zeilen, sondern `n` Zeichen (*characters*) ausgegeben.

  `-r` Zeilen in umgekehrter Reihenfolge ausgeben (letzte zuerst). Funktioniert nicht bei GNU-`tail` – stattdessen kann man das Programm `toc` verwenden.

  `-f` `tail` am Dateiende nicht beenden, sondern auf weitere Zeilen warten (Ende des Kommandos mit Strg - C). Damit kann man z. B. Logfiles beobachten, die ständig wachsen.
- `tee [-i] [-a] [datei]`
  Pipe mit T-Stück: Kopiert von `stdin` nach `stdout` und schreibt die Daten gleichzeitig in die angegebene Datei. Optionen:

  `-i` Ignorieren von Interrupts (Unterbrechungs-Taste)

  `-a` Anhängen der Info an die angegebene Datei (Voreinstellung: Überschreiben der Datei)
- `wc [-lwc] [Datei(en)]`
  Dieses Kommando zählt Zeilen, Worte oder Zeichen in einer Datei. Wird kein Dateiname angegeben, liest `wc` von der Standardeingabe. Normalerweise zählt man damit in Skripten irgendwelche Ergebnisse. Optionen:

  `-l` Zähle Zeilen

  `-w` Zähle Worte

  `-c` Zähle Zeichen

Weitere Filter sind `more`, `less`, `tr`, ...

4.4 Metazeichen zur Expansion von Dateinamen

Damit bei der Angabe von z. B. Dateinamen nicht jeder einzelne Name getippt werden muss, sondern die Dateien alle oder nach bestimmten Kriterien ausgewählt werden können, gibt es so genannte *Metazeichen* (auch *Jokerzeichen* oder *Wildcards* genannt).

Im Gegensatz zu anderen Systemen (z. B. MS-DOS) werden diese von der Shell ersetzt. Dies ist eine sehr wichtige Tatsache, die zur Folge hat, dass nahezu jedes Linux-Kommando als Dateiangabe immer eine (im Rahmen der Betriebssystem-Parameter) beliebige Menge von Dateien als Parameter haben kann. Im Programm sind daher auch keine Systemaufrufe notwendig, die auf die Verzeichnisinformation zugreifen; es wird lediglich eine Schleife benötigt, welche die einzelnen Dateien nacheinander bearbeitet. Metazeichen sind Zeichen mit erweiterter Bedeutung. Die Shell ersetzt die

Tabelle 4.1

Tabelle der Shell-Metazeichen

Metazeichen	Bedeutung
`*`	Der Stern steht für eine beliebige Zeichenfolge – oder für überhaupt kein Zeichen. Dazu ein Beispiel: `ab*` steht für alle Dateinamen, die mit `ab` beginnen, auch für `ab` selbst (also `ab`, `abc`, `abcd`, `abxyz` usw.).
`?`	Das Fragezeichen steht für genau ein beliebiges Zeichen. Zum Beispiel: `?bc` steht für alle Dateinamen mit 3 Zeichen, die auf `bc` enden (also `abc`, `bbc`, `1bc`, `vbc`, `xbc` usw.), nicht jedoch für `bc`.
`[ ]`	Die eckige Klammer wird ersetzt durch *eines* der in der Klammer stehenden Zeichen. Auch ein Bereich ist möglich, z. B. `[a-k]` = `[abcdefghijk]`. Beispiel: `a[bcd]` wird ersetzt durch `ab`, `ac` und `ad`.
`[! ]`	Die eckige Klammer mit Ausrufezeichen wird ersetzt durch ein in der Klammer *nicht* stehendes Zeichen, zum Beispiel: `[!abc]` wird ersetzt durch ein beliebiges Zeichen außer `a`, `b` oder `c`.
`\`	Der *backslash* hebt den Ersetzungsmechanismus für das folgende Zeichen auf. Beispiel: `ab\?cd` wird zu `ab?cd` – das Fragezeichen wird übernommen.

Metazeichen durch alle Dateinamen des aktuellen Verzeichnisses, die auf das Muster passen. Dabei können die Metazeichen beliebig oft an beliebiger Stelle im Dateinamen stehen (z. B.: `*abc*def*`). Die Metazeichen sind in der Tabelle 4.1 aufgelistet.

Wichtig: Bei der Umleitung von Ein- und Ausgabe werden Metazeichen in den Dateinamen hinter dem Umleitungszeichen nicht ersetzt. Beispiele für die Anwendung:

```
user$ ls -l a*
```

listet alle Dateien, die mit `a` anfangen.

```
user$ ls test?
```

listet alle Dateien, die mit `test` anfangen und fünf Zeichen lang sind (`test1`, `test2`, `testa` usw.).

```
user$ ls /dev/tty1[1-9]
```

listet alle Terminalbezeichnungen mit einer `1` in der Zehnerstelle (`tty11`, `tty12`, …, `tty19`)

Hinweis

Der `*` ist ein gefährliches Zeichen, denn Tippfehler können zum Fiasko führen, wenn versehentlich ein Leerzeichen zuviel eingegeben wird.

```
user$ rm a*
```

löscht beispielsweise alle Dateien, die mit `a` beginnen.

```
user$ rm a *
```

löscht dagegen erst die Datei `a` und dann alle Dateien im Verzeichnis.

Wie schon im dritten Kapitel erwähnt, gilt:

- Der Punkt am Anfang von Dateinamen stellt eine Ausnahme dar; er muss explizit angegeben werden, einerseits wegen der Verzeichnisreferenzen . bzw. . . , andererseits weil Dateien, die mit einem Punkt beginnen, normalerweise nicht angezeigt werden.
- Der \ am Zeilenende unterdrückt auch das ↵-Zeichen – das Kommando kann in der folgenden Zeile fortgesetzt werden (es erscheint dann der Prompt > anstelle von `$`).

4.5 String-Ersetzungen (*Quoting*)

Um bestimmte Sonderzeichen (z. B. `*`, `?`, `[ ]`, Leerzeichen, Punkt) zu übergeben, ohne dass sie von der Shell durch Dateinamen ersetzt werden, werden Anführungszeichen verwendet, die auch ineinander geschachtelt werden können. Dabei haben die drei verschiedenen Anführungszeichen *Doublequote* (¨), *Quote* (´) und *Backquote* (`) unterschiedliche Bedeutung:

- ¨ … ¨
  Keine Ersetzung der Metazeichen `* ? [ ]`, jedoch Ersetzung von Shell-Variablen (siehe unten) und Ersetzung durch die Ergebnisse von Kommandos (*Backquote*). Auch \ funktioniert weiterhin. Dazu ein Beispiel:

  ```
  echo Der * wird durch alle Dateinamen ersetzt}
  echo "Der * wird hier nicht ersetzt"
  ```

- ´ … ´
  Das einfache Anführungszeichen unterdrückt jede Substitution. Zum Beispiel:

  ```
  echo 'Weder * noch ‘pwd‘ werden ersetzt'
  ```

- ` … `
  Zwischen *Backquotes* gesetzte Kommandos werden ausgeführt und das Ergebnis wird dann als Parameter übergeben (d. h. die Ausgabe des Kommandos landet als Parameter in der Kommandozeile). Dabei werden Zeilenwechsel zu Leerzeichen. Braucht dieses Kommando Parameter, tritt die normale Parametererersetzung in Kraft. Zum Beispiel:

  ```
  echo "Aktuelles Verzeichnis: ‘ pwd ‘"
  ```

  Weil die verschiedenen `Quotes` manchmal schwer zu unterscheiden sind, wurde bei der `bash` eine weitere Möglichkeit eingeführt: Statt in *Backquotes* wird die Kommandofolge in `$(…)`eingeschlossen., z. B.:

  ```
  echo "Aktuelles Verzeichnis: $(pwd)"
  ```

4.6 Bash-Spezialitäten

Was wir bisher beschrieben haben, ist bereits seit der ersten Bourne-Shell `/bin/sh` gültig. Die Bash ist vollständig kompatibel zu der originalen *Bourne-Shell*, aber sie hat etliche Erweiterungen erfahren.

Wie bereits erwähnt, kann man die Eingaben eines Programms auch aus einer Datei lesen, anstatt sie von Hand einzugeben. Die Eingaben der Shell sind in der Regel Kommandos. Die Shell liest unter gewissen Bedingungen auch automatisch Kommandos aus bestimmten Dateien und arbeitet sie ab. DOS-Benutzer kennen etwas Vergleichbares von der Datei *AUTOEXEC.BAT*.

Im Verlauf des ersten Aufrufs arbeitet jede Shell zunächst die Datei `/etc/profile` ab, in der der Systemverwalter für alle Anwender gültige Kommandos und Variablen eintragen kann. Daran anschließend sucht die Bash nach einer der folgenden Dateien im Heimatverzeichnis des Benutzers und führt diejenige aus, die zuerst gefunden wird:

1. `.bash_profile`
2. `.bash_login`
3. `.profile`

Anders verhält es sich jedoch, wenn die Bash nicht als Login-Shell gestartet wird. Dann führt sie ausschließlich die Datei `.bashrc` aus.

Die Ausführung der Dateien lässt sich über zwei Kommandozeilen-Parameter steuern. Mit dem Parameter `-noprofile` veranlassen Sie, dass die Bash keine der oben genannten Startdateien ausführt, mit `-norc` erreichen Sie, dass die persönliche Konfigurationsdatei `~/.bashrc` ignoriert wird.

4.6.1 Der Prompt

Die Bash gibt einen Eingabe-Prompt aus, häufig in der Form *Username*`:`*Pfad*`$` Der Prompt ist über dieUmgebungsvariable `PS1` konfigurierbar. Ein Prompt in der oben genannten Form resultiert aus der Einstellung

```
PS1=\u:\w\\$
```

Um andere Einstellungen auszuprobieren, müssen Sie die Variable neu belegen. Innerhalb dieser Einstellungen können Sie folgende Metazeichen verwenden:

Metazeichen und ihre Bedeutung

`\h`	Rechnername ohne Domäne
`\H`	Vollständiger Rechnername
`\t`	Aktuelle Uhrzeit
`\u`	User-Name
`\w`	Aktuelles Verzeichnis
`\#`	Fortlaufende Nummer des aktuellen Befehls
`\\`	das Zeichen \
`\$`	# für `root`, `$` für normale User

4.6.2 Editieren der Kommandozeile

Die Hauptaufgabe einer Shell ist die Entgegennahme und Ausführung von Kommandos. Zum Bearbeiten der aktuellen Kommandozeile stehen sowohl *Emacs*- als auch *vi*-kompatible Editiermodi zur Verfügung – voreingestellt ist der *Emacs*-Modus. Wenn Sie lieber im *vi*-Modus arbeiten, stellen Sie ihn durch den Befehl `set -o vi` ein, zurück in den *Emacs*-Modus geht's mit `set -o emacs`.

Die folgende Tabelle stellt die wichtigsten Editierfunktionen des *Emacs*-Modus zusammen:

Tasten und ihre Wirkung

← und →	Bewegen des Cursors in der Kommandozeile
Esc F	springt zum nächsten Wortende
Esc B	springt zum vorigen Wortanfang
Strg + E	springt zum Zeilenende
Strg + A	springt zum Zeilenanfang
Strg + K	löscht den Text ab der Cursorposition bis zum Ende der Zeile
Strg + Y	fügt den zuletzt gelöschten Text nach der Cursorposition ein
Tab	ergänzt ein Eingabefragment zum passenden Dateinamen Ist die Ergänzung nicht eindeutig möglich, ertönt ein Signal und Sie erhalten durch zweimaliges Drücken der Tab-Taste eine Liste der in Frage kommenden Dateinamen.

Abhängig von der eingesetzten Terminalemulation können Sie Kombinationen mit der Esc -Taste oft auch mit der Alt -Taste nachbilden. Statt also nacheinander Esc und F zu drücken, funktioniert meist auch die Kombination Alt + F.

4.6.3 History-Mechanismus

In der Bash können Sie nicht nur die aktuelle Kommandozeile editieren, sie speichert auch alle zuvor eingegebenen Befehle in einer Datei, der so genannten Kommandozeilen-*History*. Auch diese Datei befindet sich im `home`-Directory des Benutzers und heißt `.bash_history`. Mit den Cursortasten ↑ und ↓ kann man in dieser Liste blättern. Darüber hinaus stehen folgende History-Befehle zur Verfügung: Strg + R sucht nach dem letzten Kommando anhand der Anfangsbuchstaben, ← springt an den Anfang der History-Liste und → springt ans Ende der History-Liste. Die Anzahl der Befehlszeilen wird mit der Variablen `HISTSIZE` eingestellt. Wächst die History darüber hinaus, verwirft die Bash die ältesten Zeilen.

4.6.4 Wichtige interne Kommandos

Interne Kommandos sind direkt in die `bash` eingebaut. Es muss also kein externes Programm geladen und gestartet werden und so ist die Kommandoausführung schneller und speicherschonender.

- `alias` weist einem Befehl oder einer Befehlsfolge einen neuen Namen zu. Beispiele für Windows-Umsteiger sind:

  ```
  alias dir='ls -l'
  alias cd..='cd ..'
  alias md='mkdir'
  alias rd='rmdir'
  alias del='rm -i'
  ```

  Ohne Parameter gibt dieser Befehl eine Liste der aktuell definierten Alias-Namen aus. Zum Löschen eines Alias-Eintrags verwendet man den Befehl `unalias` *Name*.

- `fg`, `bg` und `jobs` sind Befehle für die interne Jobkontrolle der Bash. Ein laufendes Programm kann mit (Strg) + (Z) in den Hintergrund verschoben werden; dort wird es zunächst angehalten (*suspend*). Das Kommando `jobs` gibt eine Liste aller derzeit im Hintergrund „schlafenden" Programme aus. Dabei wird für jedes Programm die interne Jobnummer angegeben. Es ist wichtig anzumerken, dass die vergebene Jobnummer in keinem Zusammenhang zu der – etwa von `ps` – angezeigten Prozessnummer steht. Mit `fg` *Jobnummer* holen Sie eine Task wieder in den Vordergrund. Soll ein angehaltenes Kommando dagegen im Hintergrund weiterlaufen, verwendet man den Befehl `bg` *Jobnummer*.

- `dirs`, `pushd`, `popd` verwalten den Stapel der internen Verzeichnisse. Mit `pushd` *Verzeichnis* wird das angegebene Verzeichnis auf dem Stapel abgelegt. Mit `popd` gelangen Sie anschließend zum zuletzt abgelegten Verzeichnis zurück. Den aktuellen Stapelinhalt zeigt der Befehl `dirs` an.

- `echo` wurde in der Bash ebenfalls erweitert: Soll der Zeilenumbruch am Ende der Ausgabe unterdrückt werden, verwendet man den Parameter `-n`. Wer die von der *C*-Funktion `printf` bekannten Ausgabesteuerzeichen verwenden will, gibt den Parameter `-e` an.

- `hash` (ohne Parameter) zeigt die Liste der Pfade zu den Programmen an. Um die Zugriffe auf Programme zu beschleunigen, verwaltet die Bash einen internen Cache der Pfade auf bereits gestartete Programme. Wird ein Programm erneut gestartet, kann die zeitaufwändige Suche entlang des Pfades entfallen. Bei der Gelegenheit wird auch gleich angezeigt, wie oft das Programm gestartet wurde. Wer am Abend wissen will, womit er sich den ganzen Tag über beschäftigt hat, kann mit diesem Kommando zumindest Hinweise bekommen (oder er wirft einen Blick auf die `.bash_history`). `hash -r` verwirft alle gespeicherten Pfade.

- `help` ist die Hilfefunktion der Bash. Ohne Parameter aufgerufen, listet sie alle internen Kommandos auf. Mit einem Parameter wird ein Hilfetext über den angegebenen Befehl ausgeben.

- `logout` führt das Skript `.bash-logout` aus und beendet die Login-Shell.

4.6.5 Zeichenkettenbildung mit geschweiften Klammern

`bash` setzt aus Zeichenketten, die in geschweiften Klammern angegeben werden, alle denkbaren Zeichenkettenkombinationen zusammen. Die offizielle Bezeichnung für diesen Substitutionsmechanismus lautet „Klammererweiterung" (brace expansion). Beispielsweise wird `teil{1,2,3}` zu `teil1 teil2 teil3`. Klammererweiterungen

können den Tippaufwand beim Zugriff auf mehrere ähnliche Dateinamen oder Verzeichnisse reduzieren. Gegenüber Jokerzeichen wie * und *?* haben Sie den Vorteil, dass auch noch nicht existierende Dateinamen gebildet werden können (etwa für `mkdir`).

```
user$ echo {a,b}{1,2,3}
a1 a2 a3 b1 b2 b3

user$ echo {a,b,c}{e,f,g}.{1,2,3}
ae.1 ae.2 ae.3 af.1 af.2 af.3 ag.1 ag.2 ag.3 be.1 be.2 be.3
bf.1 bf.2 bf.3 bg.1 bg.2 bg.3 ce.1 ce.2 ce.3 cf.1 cf.2 cf.3
cg.1 cg.2 cg.3
```

4.6.6 Berechnung arithmetischer Ausdrücke in eckigen Klammern

`bash` ist normalerweise nicht in der Lage, Berechnungen auszuführen. Wenn Sie 2 + 3 eingeben, weiß die Shell nicht, was sie mit diesem Ausdruck anfangen soll. Wenn Sie innerhalb der Shell eine Berechnung ausführen möchten, müssen Sie den Ausdruck in eckige Klammern setzen und ein $-Zeichen voranstellen.

```
user$ echo $[2+3]
5
```

Innerhalb der eckigen Klammern sind die meisten aus der Programmiersprache C bekannten Operatoren erlaubt: `+ - * /` für die vier Grundrechenarten, `%` für Modulo-Berechnungen, `== != < <= >` und `>=` für Vergleiche, `<<` und `>>` für Bitverschiebungen, `! && ||` und `||` für logisches NICHT, UND und ODER etc. Alle Berechnungen werden für 32-Bit-Integerzahlen ausgeführt (Zahlenbereich zwischen +/-2147483648). Wenn einzelne Werte aus Variablen entnommen werden sollen, muss ein `$`-Zeichen vorangestellt werden. Eine alternative Möglichkeit, Berechnungendurchzuführen, bietet das Kommando `expr`. Dabei handelt es sich um ein eigenständiges Linux-Kommando, das unabhängig von `bash` funktioniert.

4.6.7 Ausgabevervielfachung mit tee

Gelegentlich kommt es vor, dass die Ausgaben eines Programms zwar in einer Datei gespeichert werden sollen, dass aber dennoch (parallel) am Bildschirm der Programmverlauf verfolgt werden soll. In diesem Fall ist eine Verdoppelung der Ausgabe erforderlich, wobei eine Kopie am Bildschirm angezeigt und die zweite Kopie in einer Datei gespeichert wird. Diese Aufgabe übernimmt das Kommando `tee`:

```
user$ ls | tee inhalt
```

Das Inhaltsverzeichnis des aktuellen Verzeichnisses wird am Bildschirm angezeigt und gleichzeitig in der Datei `inhalt` gespeichert. Dabei erfolgt zuerst eine Weiterleitung der Standardausgabe an das Kommando `tee`. Dieses Kommando zeigt standardmäßig die Standardausgabe am Terminal an und speichert die Kopie davon in der angegebenen Datei. Dass es sich wirklich um eine Vervielfachung der Ausgabe handelt, bemerken Sie, wenn Sie auch die Standardausgabe von `tee` in eine Datei weiterleiten:

```
user$ ls | tee inhalt1 > inhalt2
```

Das Ergebnis sind zwei identische Dateien `inhalt1` und `inhalt2`. Das obige Kommando hat reinen Beispielcharakter. Etwas schwieriger zu verstehen, dafür aber sinnvoller ist das folgende Beispiel:

```
user$ ls -l | tee inhalt1 | sort +4 > inhalt2
```

In `inhalt1` befindet sich wiederum das „normale“ Inhaltsverzeichnis, das von `ls` automatisch nach Dateinamen sortiert wurde. Die Kopie dieser Ausgabe wurde an `sort` weitergegeben, dort nach der Dateigröße (fünfte Spalte, also Option +4) sortiert und in `inhalt2` gespeichert.

Übersicht Kommando-Ausführung

`kommando1; kommando2`	Führt die Kommandos nacheinander aus
`kommando1 && kommando2`	Führt K. 2 aus, wenn K. 1 erfolgreich war
`kommando1 || kommando2`	Führt K. 2 aus, wenn K. 1 einen Fehler liefert
`kommando &`	Startet das Kommando im Hintergrund
`kommando1 & kommando2`	Startet K. 1 im Hintergrund, K. 2 im Vordergrund
`(kommando1 ; kommando2)`	Führt beide Kommandos in der gleichen Shell aus

Übersicht Substitutionsmechanismen

Jokerzeichen für Dateinamen

`?`	Genau ein beliebiges Zeichen
`*`	Beliebig viele (auch null) beliebige Zeichen (aber keine .*-Dateien!)
`[abc]`	Eines der angegebenen Zeichen
`[a-f]`	Ein Zeichen aus dem angegebenen Bereich
`[!abc]`	Keines der angegebenen Zeichen
`[^abc]`	Wie oben
`~`	Abkürzung für das Heimatverzeichnis
`.`	Aktuelles Verzeichnis
`..`	Übergeordnetes Verzeichnis

Zeichenkettenzusammensetzungen

`ab{1,2,3}`	Liefert `ab1 ab2 ab3`

Arithmetik

`$[3*4]`	Arithmetische Berechnungen

Kommandosubstitution

`` `kommando` ``	Ersetzt das Kommando durch sein Ergebnis
`$(kommando)`	Wie oben, alternative Schreibweise

Auswertung von Zeichenketten

`kommando "zeichen"`	Verhindert die Auswertung von Sonderzeichen
`kommando 'zeichen'`	Wie oben, aber noch restriktiver

4.7 Reguläre Ausdrücke, *grep* und *sed*

4.7.1 Reguläre Ausdrücke (*Regular Expressions*)

Vielfach werden in Shell-Skripten Textdateien manipuliert oder Textzeilen nach bestimmten Zeichenfolgen durchsucht. In solchen Fällen hilft das Kommando `grep` (global regular expression print). So liefert das folgende Kommando eine Liste aller Prozesse, welche die Zeichenkette „apache“ enthalten:

```
user$ ps ax | grep "apache"
```

`grep` sucht also nach einer Zeichenkette im Eingabedatenstrom. Aber das, was da als Suchbegriff stehen kann, geht weit über eine einfache Zeichenkette hinaus – es sind *Textmuster*, so genannte *reguläre Ausdrücke* erlaubt. Worum handelt es sich dabei?

Beim Suchen (und Ersetzen) von Textmustern steht hiermit ein mächtiges Werkzeug zur Verfügung. Es wirkt ein ähnlicher Mechanismus, wie er bereits bei der Auswahl von Dateien in der Shell besprochen wurde – die Möglichkeiten gehen aber sehr viel weiter. Die einfachste Form ist eine einfache Zeichenkette als Suchmuster. Diese Zeichenkette kann spezielle Zeichen (Metazeichen) mit besonderer Bedeutung enthalten. Wichtig sind hier vor allem folgende Metazeichen:

- `^` steht für den Zeilenbeginn
  `/^Meier/` adressiert im ex-Modus des `vi` die Zeile, die mit `Meier` beginnt
- `$` steht für das Zeilenende,
  `/Meier$/` adressiert im ex-Modus des `vi` die Zeile, die mit `Meier` endet.
- `[...]` definiert *einen* Buchstaben aus der Zeichenmenge in `[...]`. Beginnt die Aufzählung der Zeichen mit `^`, wird nach einem Zeichen gesucht, das *nicht* in der Menge enthalten ist (bei den Dateinamen war dies das `!`-Zeichen).

  `[ABC]`: einer der Buchstaben `A`, `B` oder `C`
  `[A-Z]`: Großbuchstaben
  `[A-Za-z]`: alle Buchstaben
  `[^0-9]`: keine Ziffer

  Die Zeichen „]“, „-“ und „^“ benötigen eine Sonderbehandlung, wenn sie als „normales“ Zeichen in einer Menge vorkommen sollen. Mengenangaben bei regulären Ausdrücken können somit enthalten:

  – Zeichen dicht geschrieben: `[abcx123]`
  – Bereich von … bis …: `[a-fu-x]` (alle Zeichen, deren ASCII-Code dazwischen liegt)
  – das Zeichen „^“ (*nicht* als erstes): `[abx^17]`
  – das Zeichen „]“ (*nur* als erstes): `[]abx17]`
  – das Zeichen „-“ (*nur* als erstes oder letztes): `[-+.0-9]`

- `.` der Punkt steht für ein beliebiges Zeichen (wie `?` in der Shell).
- `*` steht für eine beliebige Folge des *vorhergehenden* Zeichens (auch gar kein Zeichen!).

  `a*`: Leerstring oder beliebige Folge von `a`'s
  `aa*`: eine beliebige Folge von `a`'s (mindestens eines)
  `[a-z]*`: Leerstring oder eine beliebige Folge von Kleinbuchstaben

`[a-z][a-z]*`: eine beliebige Folge von Kleinbuchstaben (mindestens einer)
`.*`: jede beliebige Zeichenfolge

- `+` steht wie `*` für eine beliebige Folge des *vorhergehenden* Zeichens, jedoch muss das Zeichen mindestens einmal auftauchen.
  `a+` steht für eine beliebige Folge von `a`'s, jedoch mindestens eines (entspricht `aa*`).
- `\` hebt den Metazeichen-Charakter für das folgende Zeichen auf.
  `a*` steht für Leerstring oder beliebige Folge von `a`'s.
  `a\*` steht für die Zeichenfolge `a*`.
- `\(...\)` Reguläre Ausdrücke können mit Klammern gruppiert werden. Damit die Klammern nicht als Teil der Zeichenkette aufgefasst werden, muss ein `\` davor stehen.
  `\([A-Za-z]*\)` gruppiert beispielsweise ein Wort aus beliebig vielen Buchstaben, wobei auch ein leeres Wort (0 Buchstaben) dazu gehört. Soll das Wort mindestens einen Buchstaben enthalten, muss man `\([A-Za-z][A-Za-z]*\)` schreiben. Die Anwendung solcher Gruppen wird weiter unten gezeigt – es kann in den Editor-Befehlen nämlich Bezug auf die Gruppen genommen werden.
- `\i` Referenzieren des `i`-ten Klammerausdrucks (siehe Beispiele weiter unten)

Reguläre Ausdrücke sind in doppelter Hinsicht von Bedeutung: zur Adressierung von Zeilen und beim Ersetzen von Zeichenketten. Mit regulären Ausdrücken lässt sich der Ersetzungsbefehl erweitert verwenden. Reguläre Ausdrücke basieren auf einem nicht-deterministischen finiten Automaten, der folgendermaßen vorgeht: Er merkt sich die Stellen, an denen mehr als eine Möglichkeit zu kontrollieren ist. Stellt er beim Testen einer Variante fest, dass der Gesamtausdruck nicht mehr zutrifft, geht er zurück zum „Scheideweg" und prüft die Alternative. Erst wenn alle abgehakt sind, entscheidet der NFA, ob der Ausdruck zutrifft oder nicht. Durch dieses „Backtracking" genannte Vorgehen beherrscht ein Programm (z. B. die Editoren `vi` oder `sed` nummerierte Rückbezüge wie in `s/(Eins) (Zwei)/\2 \1/g`. Hier sorgen die Klammern dafür, dass sich der Editor jedes „Eins" und jedes „Zwei" merkt. Im zweiten Teil vertauscht dann `\2 \1` die beiden miteinander.

In den folgenden Beispielen wird von einer Telefonliste ausgegangen, die aus Namen, Vornamen und Telefonnummern besteht:

```
Huber Karl       123
Meier Hans       231
Gaukeley Gundel  781
Schulze Maria    256
Weber Klaus      400
```

Für die Befehle in den Beispielen soll uns der Editor `vi` im Zeilenediter-Modus (ex-Modus) dienen. Bei der Shell-Programmierung kommt natürlich kein interaktiver Editor wie `emacs`, `vi` oder `ed` zum Einsatz (obwohl das auch ginge), sondern der unten beschriebene `sed`, der das/die Editierkommando(s) als alle Zeilen in der Standardeingabe anwendet und das Ergebnis in die Standardausgabe schreibt. Die Zeilen beginnen immer mit „1,$s", was bedeutet:„suche und ersetze in allen Zeilen des Textes". Die Telefonnummern sollen um den Text `Tel.:` ergänzt werden. Dazu wird ein weiteres Feature der `s`-Anweisung verwendet, der oben beschriebene Rückbezug.

```
1,$s/\([0-9][0-9]*\)\$/Tel.: \1/
```

Das Ergebnis sieht dann folgendermaßen aus:

```
Huber Karl      Tel.: 123
Meier Hans      Tel.: 231
Gaukeley Gundel Tel.: 781
Schulze Maria   Tel.: 256
Weber Klaus     Tel.: 400
```

Jetzt sollen Nachname und Vorname vertauscht werden (beachten Sie die Leerzeichen zwischen den Gruppen):

```
1,$s/\{[A-Za-z][A-Za-z]*\) \{[A-Za-z][A-Za-z]*\)/\2/ \1/
```

Das Ergebnis:

```
Karl Huber      Tel.: 123
Hans Meier      Tel.: 231
Gundel Gaukeley Tel.: 781
Maria Schulze   Tel.: 256
Klaus Weber     Tel.: 400
```

Hinweis

Aus Gründen der Übersichtlichkeit wurden die Umlaute und das scharfe S weggelassen – in einer realen Anwendung müssen die natürlich mit in die eckige Klammer. Gegebenenfalls sind auch noch weitere Buchstaben zu berücksichtigen, etwa die Vokale mit Akzent.

In einem letzten Schritt werden die Telefonnummern aus der Liste entfernt:

```
1,$s/([0-9][0-9]*\$//
```

Das Ergebnis:

```
Karl Huber
Hans Meier
Gundel Gaukeley
Maria Schulze
Klaus Weber
```

Reguläre Ausdrücke versuchen normalerweise, den frühesten Treffer im String zu finden. Kommt aber ein Quantifizierer wie * ins Spiel, will der reguläre Ausdruck so viel wie möglich finden, er wird gierig („greedy“). Dabei ist die „Gierigkeit“ stärker als die „Links-Bindung“.

Will man beispielsweise in HTML-Code einen bestimmten Tag erwischen, heißt der erste Versuch vermutlich: `/<.*>/`. übersetzt: „Suche beliebig viele (auch gar kein) Zeichen, umschlossen von spitzen Klammern.“ Was würde Perl nun in der Zeile

```
<B>Wir</B> sind die <B>Champions</B>!
```

finden? Alles von der ersten spitzen Klammer bis zur letzen vor dem Ausrufezeichen. Da ein Quantifizierer dabei ist, gilt nicht mehr „Treffer soweit links wie möglich“ (also das erste `<B>`) sondern „So viel wie möglich“.
Die Gierigkeit lässt sich jedoch durch ein hinter `+` oder `*` gesetztes Fragezeichen beschränken. Benutzt man im obigen Beispiel `<.*?>`, wird es `<B>` finden. In solchen Fällen hilft ebenfalls: `/<[^>]+>/`. Dieser Ausdruck sucht ein <, dann etwas, was kein > ist, davon mindestens eines, schließlich ein >.

Es gibt ein paar Standard-Tools, die reguläre Ausdrücke verwenden. Das bekannteste ist das schon erwähnte Text-Suchprogramm `grep`.

4.7.2 *grep*

Der Aufruf erfolgt über das Kommando

```
grep reg. Ausdruck [Optionen] [Dateiname(n)]
```

grep steht für „global regular expression print“. Das Kommando ist ein Tool zum Durchsuchen von Dateien auf Strings, die mit regulären Ausdrücken definiert sind. Die gefundenen Zeilen werden auf die Standardausgabe ausgegeben. Wegen der Angabe des regulären Ausdrucks in der Befehlszeile muss der reguläre Ausdruck in `" ... "` oder `´ ... ´` eingeschlossen werden, wenn er Leerzeichen oder Sonderzeichen enthält, die von der Shell ersetzt werden. Wurde keine Datei angegeben, liest *grep* von der Standardeingabe. Daneben gibt es einige nützliche Optionen:

`-q` Nur Return-Wert zurückgeben (gefunden/nicht gefunden)
`-n` Zeilennummern mit ausgeben
`-c` Nur die Anzahl der Zeilen ausgeben, für die der reguläre Ausdruck erfüllt ist
`-l` Nur die Namen der Dateien ausgeben, in denen etwas gefunden wurde
`-L` Nur die Namen der Dateien ausgeben, in denen *nichts* gefunden wurde
`-i` Groß-/Kleinschreibung nicht unterscheiden
`-v` Alle Zeilen ausgeben, in denen der reguläre Ausdruck *nicht* erfüllt ist.

Daneben kennt `grep` noch etliche andere Optionen, die aber wesentlich seltener gebraucht werden. Hier einige Beispiele:

- Suchen nach allen „Meiers“, „Maiers“, „Meyers“, „Mayers“ in einer Datei:

  ```
  user$ grep 'M[ae][iy]er' Namen
  ```

- Suchen nach allen Zeilen, die mit dem Wort „Beispiel“ beginnen; Ausgabe der Zeilennummern:

  ```
  user$ grep -n '^Beispiel' Vorlesung
  ```

- Wichtig ist *grep* für den Systemverwalter, z. B. bei der Suche nach Login-Namen in der Benutzerdatei, damit nicht derselbe Name zweimal verwendet wird:

  ```
  user$ grep '^klaus:' /etc/passwd
  ```

  Bei *grep* wird auch vielfach nur der Rückgabewert des Programms (nicht die Ausgabe!) verwendet. In Skript-Konstruktionen (siehe später) kann das beispielsweise so aussehen:

  ```
  user$ grep -q '^klaus:' /etc/passwd && echo "Gibts schon"
  ```

4.7.3 Der Stream-Editor *sed*

sed liest aus der angegebenen Eingabedatei (normalerweise Standardeingabe) Zeile für Zeile in seinen Eingabepuffer, führt die Editieranweisungen auf die Zeile aus und schreibt sie auf die Standardausgabe. *sed* beendet seine Arbeit, wenn das Ende der Eingabedatei erreicht ist oder explizit eine Beendigungsanweisung erreicht wurde. Ein zweiter Puffer, der Haltepuffer, dient zum Zwischenspeichern von Ergebnissen. Aufruf:

sed [-n] [-e '*sed-Befehle*'] [-f *Skriptdatei*] [*Eingabedatei(en)*]

Der *sed* ist ein nicht interaktiver Editor zur Dateibearbeitung. Daher muss die Befehlsfolge zum Editieren bereits beim Aufruf festliegen. Der Inhalt der Eingabedatei(en) oder der Standardeingabe (wenn keine Eingabedatei angegeben wurde) wird durch die Editieranweisungen modifiziert und dann auf die Standardausgabe ausgegeben. Mit der Option `-f Skriptdatei` entnimmt der *sed* die Editieranweisungen dieser Skriptdatei. Beachten Sie, dass die Eingabedatei selbst nicht verändert wird! Soll das Ergebnis der *sed*-Anweisung in einer (anderen) Datei gespeichert werden, muss die Standardausgabe in diese Datei umgelenkt werden. Optionen:

`-n` Meldungen werden unterdrückt (sinnvoll im Zusammenhang mit Pipes, die in den `sed` hineingeleitet werden).

`-e` Es werden die auf die Option `-e` folgenden Anweisungen für die Bearbeitung der Eingabedatei verwendet. Bei mehr als einer Anweisung müssen die Anweisungen durch Semikola voneinander getrennt werden. Das Semikolon muss dabei ohne Leerzeichen direkt hinter der Anweisung stehen! Diese Option kann mehrfach angegeben werden.

Um Fehlinterpretationen der Shell zu vermeiden, sollte die Editieranweisung grundsätzlich in Hochkommata oder „Gänsefüßchen" eingeschlossen werden.

`-f` Wird diese Option angegeben, liest *sed* seine Editieranweisungen aus der angegebenen Datei. Die Anweisungen müssen entweder durch Semikola getrennt werden oder jede Anweisung muss in einer eigenen Zeile stehen. Leerzeichen hinter einer Anweisung sind nicht erlaubt. Die Anweisungen dürfen nicht in Hochkommata eingeschlossen werden. Durch mehrfache Angabe der Option können mehrere Skriptdateien zugewiesen werden.

`-r` Erweiterte reguläre Ausdrücke verwenden.

`sed` kennt dieselben Anweisungen wie der interaktive Editor `ed` oder der `vi` im ex-Modus. Auch die Adressierung (Zeilennummern, reguläre Ausdrücke) erfolgt bei allen auf die gleiche Weise. Editieranweisungen haben die folgende Form:

```
[Adresse1 [,Adresse2]] Anweisung [Argumente]
```

Für jede Zeile der Eingabedatei werden alle Anweisungen ausgeführt. Durch Angabe einer Zeile (*Adresse1*) oder eines Zeilenbereichs (*Adresse1, Adresse2*) kann man deren Wirkung jedoch einschränken. Anstelle von Zeilennummern können auch reguläre Ausdrücke verwendet werden:

```
[/Muster/] Anweisung [Argumente]
[/Muster1/] [/Muster2/] Anweisung [Argumente]
```

Im ersten Fall wird jede Zeile, die die Zeichenkette *Muster* enthält, von der Anweisung bearbeitet, im zweiten Fall alle Bereiche der Eingabedatei, die mit einer Zeile, die *Muster1* enthält, beginnen und mit einer Zeile, die *Muster2* enthält, enden.

Zusätzlich hat der *sed* noch weitere Funktionen (z. B. Test- und Sprungfunktionen, Klammerung, Wiederholungen). Er eignet sich daher besonders für Shell-Programme. Interessant sind beim *sed* die Klammern:

`( )` Die runden Klammern müssen, wie schon erwähnt, mit \ geschützt werden. Sie legen die Gruppierung der einzelnen Komponenten fest.

`{ }` Die geschweiften Klammern müssen ebenfalls mit \ geschützt werden. Sie legen Wiederholungen fest. Im Folgenden stehen `Z` für ein Zeichen und `M` und `N` für Zahlen zwischen 0 und 253.

1. genau `M` Wiederholungen: `Z\{M\}`; z. B. alle Namen mit genau 8 Zeichen Länge: `sed -n '/^.\{8\}/p' namen`
2. mindestens `M` Wiederholungen: `Z\{M, \}`
3. zwischen `N` und `M` Wiederholungen: `Z\{N,M\}`

Am häufigsten kommt die Zeichenketten-Ersetzungsanweisung in Shell-Anwendungen vor. Dazu ein etwas komplexeres Beispiel: Es soll einer Datei mit der Endung `.tex` auf eine Datei gleichen Namens, jedoch mit der Endung `.bak`, kopiert werden. Aus `diplom-kap1.tex` wird beispielsweise `diplom-kap1.bak`. Den Dateinamen *ohne* Endung übergeben wir mal in einer Shell-Variablen:

1. Erst einmal probieren wir die Generierung des neuen Namens. Da `sed` seine Daten auf der Standardeingabe erwartet, pipen wir mittels `echo` den Variableninhalt hinein:

   ```
   user$ DATEI="diplom-kap1.tex"
   user$ echo $DATEI | sed -e 's/tex/.bak/'
   diplom-kap1.bak
   ```

   Da der Punkt ein Metazeichen ist, musste er im ersten Ausdruck, der ja ein regulärer Ausdruck ist, geschützt werden.

2. Jetzt muss der neue Name irgendwie zum Kopieren verwendet werden. Deshalb nützt er auf der Standardausgabe relativ wenig. Aber wir haben ja eine Möglichkeit, den Output wieder in eine Variable zu packen:

   ```
   user$ DATEI="diplom-kap1.tex"
   user$ NEU=` echo $DATEI | sed -e 's/tex/.bak/'`
   user$ echo $NEU
   diplom-kap1.bak
   ```

3. Das hat also geklappt. Nun kann man das Kopierkommando anhängen:

   ```
   user$ DATEI="diplom-kap1.tex"
   user$ NEU=` echo $DATEI | sed -e 's/tex/.bak/'`
   user$ cp $DATEI $NEU
   ```

4. Die Profis packen natürlich alle Aktionen in eine Zeile:

```
user$ DATEI="diplom-kap1"
user$ cp $DATEI ` echo $DATEI | sed -e 's/tex/.bak/'`
```

Wenn wir nun noch eine Schleife um die Zeile mit der Kopieranweisung setzen, in der die Variable `DATEI` nacheinander alle Dateinamen annimmt, die umbenannt werden sollen, haben wir schon ein nettes Tool.

Noch ein Beispiel – Umlaute HTML-gerecht ersetzen:

```
cat textfile | sed -e 's/ä/\\&auml;/g' \
                   -e 's/ö/\\&ouml;/g' \
                   -e 's/ü/\\&uuml;/g' \
                   -e 's/Ä/\\&Auml;/g' \
                   -e 's/Ö/\\&Ouml;/g' \
                   -e 's/Ü/\\&Uuml;/g' \
                   -e 's/ß/\\&szlig;/g' > htmlfile
```

Die folgende Liste fasst die wichtigsten *sed*-Kommandos zusammen:

- `a\ text` (*append lines*)
  text wird nach der aktuellen Eingabezeile auf die Standardausgabe geschrieben. Besteht der Text aus mehreren Zeilen, so muss das Fortsetzungszeichen \ am Zeilenende vor ↵ angegeben werden.
- `b[marke]` (*branch to label*)
  Es wird zu der *marke* (in der Form :*marke* angegeben) des *sed*-Skripts gesprungen und dort die Abarbeitung des Skripts fortgesetzt. Fehlt die Angabe der Marke, so wird die nächste Eingabezeile bearbeitet.
- `c\ [text]` (*change lines*)
  Der Inhalt des Eingabepuffers wird durch *text* ersetzt. Besteht *text* aus mehreren Zeilen, so muss das Fortsetzungszeichen \ am Zeilenende vor ↵ angegeben werden.
- `d` (*delete lines*)
  Der Inhalt des Eingabepuffers wird gelöscht (nicht ausgegeben) und das *sed*-Skript wird sofort wieder von Beginn an mit dem Lesen einer neuen Eingabezeile gestartet.
- `i\ text` (*insert lines*)
  Der *text* wird vor der aktuellen Eingabezeile auf die Standardausgabe geschrieben. Besteht *text* aus mehreren Zeilen, muss das Fortsetzungszeichen \ am Zeilenende vor ↵ angegeben werden.
- `l` (*list pattern space on the standard output*)
  Der Inhalt des Eingabepuffers wird auf die Standardausgabe geschrieben, wobei nicht druckbare Zeichen durch ihren ASCII-Wert (2-Ziffer) ausgegeben werden. Überlange Zeilen werden als mehrere einzelne Zeilen ausgegeben.
- `n` (*next line*)
  Der Eingabepuffer wird auf die Standardausgabe ausgegeben, dann wird die nächste Eingabezeile in den Eingabepuffer gelesen.
- `N` (*Next line*)
  Die nächste Eingabezeile wird an den Eingabepuffer (mit Newline-Zeichen getrennt) angehängt; die aktuelle Zeilennummer wird hierbei weitergezählt.

- `p` (*print*)
  Der Eingabepuffer wird auf die Standardausgabe ausgegeben.
- `P` (*Print first part of the pattem space*)
  Der erste Teil des Eingabepuffers (einschließlich des ersten Newline) wird auf die Standardausgabe ausgegeben.
- `q` (*quit*)
  Nach Ausgabe des Eingabepuffers (nicht bei Option `-n`) wird zum Skriptende gesprungen und die Skriptausführung beendet.
- `r datei` (*read the contents of a file*)
  Es wird die Datei *datei* gelesen und ihr Inhalt auf die Standardausgabe ausgegeben, bevor die nächste Eingabezeile gelesen wird. Zwischen `r` und *datei* muss genau ein Leerzeichen sein.
- `s/rA/txt/[flag]`
  (*substitute*) Im Eingabepuffer werden die Textstücke, die durch den regulären Ausdruck *rA* abgedeckt sind, durch den String *txt* ersetzt. Anstelle des Trennzeichens `/` kann jedes beliebige Zeichen verwendet werden. Die *flags* legen fest, wie der Ersetzungsprozess durchgeführt werden soll. Unter anderem gibt es:
  `n`: Es wird nur `n`-mal in der Zeile ersetzt.
  `g` (*global*): Es werden alle passenden Textstücke, die sich nicht überlappen, ersetzt.
  Ohne *flag* wird nur das erste passende Textstück ersetzt.
- `w datei` (*write to a file*)
  Es wird der Eingabepuffer an das Ende der Datei *datei* geschrieben. Zwischen `w` und *datei* muss genau ein Leerzeichen sein.
- `y/st1/st2/`
  Im Eingabepuffer werden alle Zeichen, die in *st1* vorkommen, durch die an gleicher Stelle in *st2* stehenden Zeichen ersetzt. *st1* und *st2* müssen gleich lang sein.

Die folgende Übersicht in Tabelle 4.2 fasst die verschiedenen Möglichkeiten zusammen.

4.8 Shell-Programmierung

Die Shell dient nicht nur der Kommunikation mit dem Anwender, sondern sie kennt die meisten Konstukte einer Programmiersprache. Es lassen sich Anweisungen in einer Textdatei speichern, die dann wie ein beliebiges anderes Unix-Kommando aufgerufen werden kann. Solche Dateien nennt man *Shell-Skripten* oder *shell scripts*. Ein Shell-Skript kann aufgerufen werden über eine Subshell (`sh Dateiname`) oder über den Namen, wenn die Excecute-Berechtigung gesetzt ist (mittels des Befehls `chmod Dateiname`).

Das Shell-Skript wird mit einem Editor erstellt und kann alle Möglichkeiten der Shell nutzen, die auch bei der interaktiven Eingabe möglich sind. Insbesondere kann auch die Umleitung der Ein-/Ausgabe wie bei einem Binärprogramm erfolgen. Selbstverständlich lassen sich auch innerhalb eines Skripts weitere Shell-Skripten aufrufen. Dem Shell-Skript können Parameter übergeben werden; es ist damit universeller verwendbar.

Tabelle 4.2

Metazeichen bei regulären Ausdrücken

Ausdruck	Beschreibung
^	steht für den Zeilenbeginn qq^Meier adressiert eine Zeile, die mit „Meier" beginnt
$	steht für das Zeilenende „Meier$" adressiert Zeile, die mit „Meier" endet
[ ]	definiert einen Buchstaben aus dem Bereich in `[ ]` `[ABC]`: einer der Buchstaben A, B oder C `[A-Z]`: Großbuchstaben `[A-Za-z]`: alle Buchstaben Beginnt der Bereich mit ^, wird nach einen Zeichen gesucht, das **nicht** im Bereich enthalten ist. `[^0-9]`: keine Ziffer
.	Der Punkt steht für ein beliebiges Zeichen.
*	Der * steht für eine beliebige Folge des **davorstehenden** Zeichens * (auch 0 Zeichen!). `a*`: Leerstring oder beliebige Folge von „a" * `aa*`: eine beliebige Folge von „a" (min. 1) `[a-z]*`: * Leerstring oder Folge von Kleinbuchstaben `.*`: jede beliebige * Zeichenfolge
*?	Der Stern „*" ist recht „gefräßig" (greedy), d. h. es wird versucht, maximal viele Zeichen in den regulären Ausdruck einzuschließen. Bei der Zeichenkette „aaa:bbb:ccc" würde der Ausdruck „.*:" den String „aaa:bbb:" finden. Durch das nachgestellte Fragezeichen wird diese Eigenschaft umgekehrt, es wird die minimale Teilzeichenkette genommen, also im obigen Beispiel „`aaa:`".
+	Das + steht für eine beliebige Folge des **davorstehenden** Zeichens, jedoch mindestens eines. `a+`: a, aa, aaa, aaaa, …
?	Nullmal oder einmal das **davorstehende** Zeichen
\	Metazeichen-Charakter des folgenden Zeichens aufheben `a*` steht für Leerstring oder beliebige Folge von a's. `a\*` steht für die Zeichenfolge „a*".
(…)	Gruppieren regulärer Ausdrücke `([A-Za-z]*)` gruppiert beispielsweise ein Wort aus beliebig vielen Buchstaben, wobei auch ein leeres Wort (0 Buchstaben) dazugehört. Soll das Wort mindestes einen Buchstaben enthalten, muss man `([A-Za-z][A-Za-z]*)` oder `([A-Za-z]+)` schreiben.
\i	Referenzieren des *i*-ten Klammerausdrucks
&	Referenzieren des Suchausdrucks (beim s-Befehl) `s/hallo/& &/` erzeugt „hallo hallo"

4.8.1 Testen von Shell-Skripten

Die Ersetzungsmechanismen der Shell machen es manchmal nicht leicht, auf Anhieb korrekt funktionierende Skripten zu erstellen. Zum Testen bieten sich daher einige Möglichkeiten an:

- Einfügen von `echo`-Kommandos anstelle der vorgesehenen Kommandos:

```
echo [Argumente]
```

Dieses Kommando gibt die Argumente auf dem Bildschirm aus. Für den Einsteiger ist das Kommando wichtig, weil er so die Kommando-Bearbeitung der Shell recht gut verfolgen und studieren kann.

- Aufruf über Subshell mit Optionen:

  `-v` (*verbose*) Die Shell gibt alle bearbeiteten Befehle aus.

  `-x` (*execute*) Die Shell gibt die Ersetzungen aus.

  `-n` (*noexecute*) Die Shell gibt die Befehle zwar aus, sie werden jedoch nicht ausgeführt.

  Typischer Kommando-Aufruf: `sh -vx Dateiname`

- „Kritische" Kommandos (z. B. `rm`) sollten Sie zunächst durch vorangestellte `echo`-Befehle „entschärfen".
- Ein Doppelpunkt vor einer Zeile wirkt wie ein Kommentar: Das Kommando wird nicht ausgeführt, aber die Parametersubstitution erfolgt.

4.8.2 Kommentare in Shell-Skripten

Wie Programme müssen auch Shell-Skripten kommentiert werden. Kommentare werden durch das Zeichen # eingeleitet. Alles was in einer Zeile hinter dem # steht, wird als Kommentar betrachtet (übrigens betrachten auch nahezu alle anderen Unix-Programme das # als Kommentarzeichen in Steuer- und Parameterdateien). Leer- und Tabulatorzeichen können normalerweise in beliebiger Anzahl verwendet werden. Da die Shell Strukturen höherer Programmiersprachen enthält, ist durch Einrücken eine übersichtliche Gestaltung der *scripts* möglich. Auch eingestreute Leerzeilen sind nahezu überall erlaubt.

Bei vielen Skripten findet man eine Sonderform der Kommentarzeile zu Beginn, die beispielsweise so aussieht:

```
#!/bin/sh
```

Durch diesen Kommentar wird festgelegt, welches Programm für die Ausführung des Skripts verwendet wird. Er wird hauptsächlich bei Skripten verwendet, die allen Benutzern zur Verfügung stehen sollen. Da möglicherweise unterschiedliche Shells verwendet werden, kann es je nach Shell (`sh`, `csh`, `ksh`,...) zu Syntaxfehlern bei der Ausführung des Skripts kommen. Durch die Angabe der ausführenden Shell wird dafür gesorgt, dass nur die „richtige" Shell verwendet wird. Die Festlegung der Shell stellt außerdem einen Sicherheitsmechanismus dar, denn es könnte ja auch ein Benutzer eine modifizierte Shell verwenden. Neben den Shells können auch andere Programme zur Ausführung des Skripts herangezogen werden; häufig sind *awk*- oder *Perl*-Skripten.

In Shell-Script-Dateien dürfen die Zeilen auch nicht durch die Windows-typische Kombination aus Carriage Return und Linefeed getrennt sein. Das kann z. B. passieren, wenn die Dateien unter Windows erstellt und dann nach Linux kopiert wurden. In diesem Fall liefert `bash` die Fehlermeldung *bad interpreter*, weil eben das Carriage Return am Zeilenende Teil des Dateinamens ist und es ein Programm „`bash^M`" nicht gibt (das kann ihnen übrigens auch mit awk- oder perl-Programmen passieren). Die Carriage Returns bekommt man mit `vi` oder `recode` weg.

Hinweis

Shell-Script-Programme können nur ausgeführt werden, wenn die Zugriffsbits für den Lesezugriff (r) und die Ausführung (x) gesetzt sind (`chmod a+rx datei`). In der ersten Zeile eines Scripts dürfen keine deutschen Sonderzeichen verwendet werden, auch nicht in Kommentaren. Die `bash` weigert sich sonst, die Datei auszuführen.

4.8.3 Shell-Variable

Variablen sind frei wählbare Bezeichner (Namen), die beliebige Zeichenketten aufnehmen können. Bestehen die Zeichenketten nur aus Ziffern, werden sie von bestimmten Kommandos als Integer-Zahlen interpretiert (z. B. `expr`). Bei Variablen der Shell sind einige Besonderheiten gegenüber anderen Programmiersprachen zu beachten. Im Umgang mit Variablen lassen sich grundlegend drei Formen unterscheiden: Die Variablendeklaration (z. B. `$foo=`), die Wertzuweisung (z. B. `$foo=bar`) und die Wertreferenzierung (z. B. `$echo $foo`).

Im Allgemeinen werden Variablen in der Shell nicht explizit deklariert. Vielmehr ist in der Wertzuweisung die Variablendeklaration implizit enthalten. Wird eine Variable dennoch ohne Wertzuweisung deklariert, so wird bei der Wertreferenzierung ein leerer String zurückgegeben.

Für Variable gilt allgemein:

- Variablen sind frei wählbare Bezeichner (Namen), die mit einem Buchstaben beginnen und bis zu 200 Zeichen lang sein dürfen. Leerzeichen innerhalb von Variablen sind (normalerweise) nicht erlaubt (und schlechter Stil).
- Ist der Wert, d. h der Inhalt einer Variablen gemeint, wird ein `$`-Zeichen vor den Namen gestellt (s. o.).
- Mittels spezieller Funktionen kann eine Variable auch numerisch oder logisch interpretiert werden.
- Shell-Skripten dürfen ihrerseits wieder Shell-Skripten oder Programme aufrufen. Dabei muss die Vererbung einer Variablen ausdrücklich festgelegt werden (exportieren durch den `export`-Befehl), andernfalls kann eine andere Shell – oder ein beliebiges anderes Programm – nicht darauf zugreifen.
- Jede Subshell läuft in einer eigenen Umgebung, d. h., Variablendefinitionen (und die Wirkung verschiedener Kommandos) in der Subshell sind nach deren Beendigung wieder „vergessen“. Es ist nicht möglich, den Wert einer Variablen aus einem Unterprogramm in die aufrufende Ebene zu übergeben.
- Die Zuweisung eines Wertes an die Variable erfolgt mittels Gleichheitszeichen:
  - `VAR=`*`Wert`*
    `Wert` ist ein String (ohne Leerzeichen und Sonderzeichen).
  - `VAR="`*`Wert`*`"`
    `Wert` ist ein String (Ersetzung eingeschränkt, Leerzeichen und Sonderzeichen dürfen enthalten sein).

  - `VAR=`*kommando*` oder `VAR=$(`*kommando*`)`
    Wert der Variablen ist die Ausgabe desKommandos (Newline wird zu Leerzeichen).

- Es hat sich die Konvention eingebürgert, Variablen zur Unterscheidung von Kommandos groß zu schreiben.
- Soll der Wert der Variablen mit einem String konkateniert werden, ist der Name in geschweifte Klammern einzuschließen, damit die Shell erkennen kann, wo der Variablenname endet. Bei der Zuweisung von Zahlen an Shell-Variable werden führende Leerzeichen und führende Nullen ignoriert.

Beispiele (Kommando-Eingaben beginnen mit `user$`):

```
user$ VAR="Hello World!"
user$ echo $VAR
Hello World!

user$ VAR=` pwd`
user$ echo "aktuelles Verzeichnis: $VAR"
aktuelles Verzeichnis: /home/user
user$ echo ${VAR}/bin
/home/user/bin

user$ VAR=/usr/tmp/mytmp
user$ ls > $VAR
```

Das letzte Beispiel schreibt die Ausgabe von `ls` in die Datei `/usr/tmp/mytmp`.

Enthält eine Variable ein Kommando, so kann dieses Kommando durch Angabe der Variablen ausgeführt werden, z. B.:

```
user$ VAR="echo Hallo"
user$ $VAR
Hallo
```

Innerhalb der Anführungszeichen werden Variablen unterschiedlich behandelt. Angenommen, die Variable `VAR` hat die Zeichenkette „abcdef“ zum Inhalt. Dann gilt:

- Einfache Anführungszeichen entwerten alle zwischenliegenden Metazeichen und erlauben auch keine Variablenreferenz:

  ```
  user$ echo '$VAR'
  $VAR
  ```

- Doppelte Anführungszeichen entwerten alle zwischenliegenden Metazeichen, nicht jedoch die Variablen- und Kommandosubstitution:

  ```
  user$ echo "$VAR"
  abcdef
  ```

4.8.4 Vordefinierte Variable

Beim Systemstart und beim Aufruf der Dateien `/etc/profile` (System-Voreinstellungen), `.profile` (benutzereigene Voreinstellungen) und ggf. weiteren Dateien, die ja

auch Shell-Skripten sind, werden bereits einige Variablen definiert. Alle aktuell definierten Variablen können durch das Kommando `set` aufgelistet werden. Einige vordefinierte Variablen sind neben anderen in Tabelle 4.3 aufgeführt. In der nächsten Tabelle 4.4 sind spezielle Variablen definiert.

Tabelle 4.3

Einige vordefinierte Shell-Variablen

Variable	Bedeutung
`HOME`	Home-Verzeichnis (absoluter Pfad)
`IFS`	Trennzeichen zwischen Parametern (in der Regel Leerzeichen, Tabulator, Newline)
`PATH`	Suchpfad für Kommandos und Skripten
`MANPATH`	Suchpfad für die Manual-Seiten
`MAIL`	Mail-Verzeichnis
`SHELL`	Name der Shell
`LOGNAME` bzw. `USER`	Login-Name des Benutzers
`PS1`	System-Prompt (`$` oder *#*)
`PS2`	Prompt für die Anforderung weiterer Eingaben (>)
`IFS`	(*Internal Field Separator*) Trennzeichen (meist `CR`, Leerzeichen und Tab)
`TZ`	Zeitzone (z. B. `MEZ`)

Tabelle 4.4

Spezielle Shell-Variablen

Variable	Bedeutung	Kommando
`$-`	gesetzte Shell-Optionen	`set -xv`
`$$`	PID (Prozessnr.) der Shell	`kill -9 $$` („Selbstmord“)
`$!`	PID des letzten Hintergrundprozesses	`kill -9 $!` („Kindermord“)
`$?`	Exitstatus des letzten Kommandos	`cat /etc/passwd ; echo $?`

4.8.5 Parameterzugriff in Shell-Skripten

Shell-Skripten können mit Parametern aufgerufen werden, auf die über ihre Positionsnummer zugegriffen werden kann. Die Parameter können zusätzlich mit vordefinierten Werten belegt werden (später dazu mehr). Die Trennung zweier Parameter erfolgt durch die in `IFS` definierten Zeichen (Tabelle 4.5).

Der Verdeutlichung soll ein kleines Shell-Skript dienen:

```
#!/bin/sh
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter uebergeben"
echo "1. Parameter = $1"
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "alle Parameter zusammen: $*"
echo "Meine Prozessnummer PID = $$"
```

Nachdem dieses Shell-Skript mit einem Editor erstellt wurde, muss es noch ausführbar gemacht werden (`chmod u+x foo`). Anschließend wird es gestartet und erzeugt die folgenden Ausgaben auf dem Bildschirm:

```
user$ ./foo eins zwei drei vier
Mein Name ist ./foo
Mir wurden 4 Parameter uebergeben
1. Parameter = eins
2. Parameter = zwei
3. Parameter = drei
Alle Parameter zusammen: eins zwei drei vier
Meine Prozessnummer PID = 3212
user$
```

Tabelle 4.5

Zugriff auf Positionsparameter

Positionsparameter	Bedeutung
`$#`	Anzahl der Argumente
`$0`	Name des Kommandos
`$1`	1. Argument
⋮	⋮
`$9`	9. Argument
`$@`	alle Argumente (z. B. für Weitergabe an Subshell)
`$*`	alle Argumente konkateniert (ein einziger String)

Hinweis

So, wie Programme und Skripten des Unix-Systems in Verzeichnissen wie `/bin` oder `/usr/bin` gesammelt werden, ist es empfehlenswert, im Home-Directory ein Verzeichnis `bin` einzurichten, das Programme und Skripten aufnimmt. Die Variable `PATH` wird dann in der Datei `.profile` durch die Zuweisung `PATH=$PATH:$HOME/bin` erweitert. Damit die Variable `PATH` auch in Subshells (d. h. beim Aufruf von Skripten) wirksam wird, muss sie mittels `export PATH` exportiert werden.

Alle exportierten Variablen bilden das Environment für die Subshells. Informationen darüber erhält man mit den Kommandos `set` (Anzeige von Shell-Variablen) und Environment-Variablen oder `env` (Anzeige der Environment-Variablen).

In Shell-Skripten kann es sinnvoll sein, die Variablen in Anführungszeichen zu setzen, um Fehler zu verhindern (eine leere Variable erzeugt bei der Auswertung gar nichts, sind Anführungszeichen vorhanden, steht ein Leerstring da). Beim Aufruf müssen Parameter, die Sonderzeichen enthalten, ebenfalls in Anführungszeichen (am besten die einfachen) gesetzt werden. Dazu ein Beispiel. Das Skript `zeige` enthält folgende Zeile:

```
grep $1 address.dat
```

Der Aufruf `zeige 'Hans Meier'` liefert nach der Ersetzung das fehlerhafte Kommando `grep Hans Meier address.dat`, das nach dem Namen `Hans` in der Adressdatei `address.dat` und einer (vermutlich nicht vorhandenen) Datei namens `Meier` sucht. Die Änderung von `zeige` liefert bei gleichem Parameter die korrekte Version:

```
grep "$1" address.dat
```

Das Skript `zeige` soll nun enthalten:

```
echo "Die Variable XXX hat den Wert $XXX"
```

Nun wird eingegeben:

```
user$ XXX=Test
user$ zeige
Die Variable XXX hat den Wert
```

Erst wenn die Variable „exportiert“ wird, erhält man das gewünschte Ergebnis:

```
user$ XXX=Test
user$ export XXX
user$ zeige
Die Variable XXX hat den Wert Test
```

Das Skript `zeige` enthalte nun die beiden Kommandos:

```
echo "zeige wurde mit $# Parametern aufgerufen:"
echo "$*"
```

Die folgenden Kommando-Aufrufe zeigen die Behandlung unterschiedlicher Parameter:

```
user$ zeige
zeige wurde mit 0 Parametern aufgerufen:
user$ zeige eins zwei 3
zeige wurde mit 3 Parametern aufgerufen:
eins zwei 3
user$ zeige eins "Dies ist Parameter 2" drei
zeige wurde mit 3 Parametern aufgerufen:
eins Dies ist Parameter 2 drei
```

Die Definition von Variablen und Shell-Funktionen (s. u.) kann man mit `unset` wieder rückgängig machen.

4.8.6 Namens- und Parameterersetzung

Die einfache Parameterersetzung (textuelle Ersetzung durch den Wert) wurde oben gezeigt. Es gibt zusätzlich die Möglichkeit, Voreinstellungen zu vereinbaren und auf fehlende Parameter zu reagieren. Bei den folgenden Substitutionen kann bei manchen Shell-Varianten der Doppelpunkt hinter `shellvar` auch fehlen.

- `${shellvar:-`*`neuerwert`*`}`
  Die Variable `shellvar` ist deklariert:
  1. Die Variable `shellvar` hat einen Wert, dann wird auf diesen Wert referenziert.
  2. Die Variable `shellvar` hat keinen Wert, dann wird bei der Referenzierung der Wert *neuerwert* eingesetzt.
- `${shellvar:=`*`neuerwert`*`}` Die Variable `shellvar` ist deklariert:
  1. Die Variable `shellvar` hat einen Wert, dann wird auf diesen Wert referenziert.
  2. Die Variable `shellvar` hat keinen Wert, dann wird der Variablen `shellvar` der Wert *neuerwert* zugewiesen und bei der Referenzierung der Wert *neuerwert* eingesetzt.
- `${shellvar:?`*`neuerwert`*`}` Die Variable `shellvar` ist deklariert:
  1. Die Variable `shellvar` hat einen Wert, dann wird auf diesen Wert referenziert.
  2. Die Variable `shellvar` hat keinen Wert, dann wird die Fehlermeldung *neuerwert* ausgegeben und das Shell-Skript abgebrochen.
- `${shellvar:+`*`neuerwert`*`}` Die Variable `shellvar` ist deklariert:
  1. Die Variable `shellvar` hat einen Wert, dann wird der Wert *neuerwert* referenziert.
  2. Die Variable `shellvar` hat keinen Wert, dann bleibt dieser Zustand erhalten.
- `${#shellvar}` liefert die Anzahl der Zeichen in der angegeben Variablen als Ergebnis (0 falls die Variable nicht existiert).
- `${shellvar#muster}` Vergleicht den Anfang desVariableninhalts mit dem Muster. Wird das Muster erkannt, wird der Variableninhalt abzüglich des Suchmusters zurückgegeben (kürzestmögliche Variante), andernfalls der unveränderte Variableninhalt. Im Muster dürfen die Jokerzeichen für Dateien verwendet werden.
- `${shellvar##muster}` Wie oben, jedoch wird die längstmögliche Variable des Musters eliminiert.

In Kommandodateien können Variablen auch Kommandonamen oder -aufrufe enthalten, da ja die Substitution vor der Ausführung erfolgt.

4.8.7 Bearbeitung einer beliebigen Anzahl von Parametern

Die Positionsparameter `$1` bis `$9` reichen nicht immer aus. Man denke nur an Skripten, die (ähnlich wie viele Kommandos) beliebig viele Dateinamen auf Parameterposition erlauben sollen. Die Shell-Skripten können mit mehr als neun Parametern versorgt werden – es wird dann mit dem Befehl `shift` gearbeitet:

```
shift
```

Eliminieren von `$1`, die weiteren Parameter werden verschoben `$2 ... $n → $1 ... $n-1` Dazu ein Beispiel: Das Skript `zeige` enthält folgende Befehle:

```
echo "$# Argumente:"
echo "$*"
shift
echo "Nach shift:"
echo "$# Argumente:"
echo "$*"
```

Der folgende Aufruf von `zeige` liefert:

```
user$ zeige eins zwei drei
3 Argumente:
eins zwei drei
Nach shift:
2 Argumente:
zwei drei
```

`shift` wird jedoch viel häufiger verwendet, wenn die Zahl der Parameter variabel ist. Es wird dann in einer Schleife so lange mit `shift` gearbeitet, bis die Anzahl der Parameter 0 ist. `zeige` wird nun abgeändert, um die Parameter zeilenweise auszugeben. Es läuft nun mit jeder Anzahl von Parametern:

```
while [ $# -gt 0 ]
   do
     echo $1
     shift
   done
```

4.8.8 Gültigkeit von Kommandos und Variablen

Jeder Kommandoaufruf und somit auch der Aufruf einer Befehlsdatei (Shell-Skript) hat einen neuen Prozess zur Folge. Bekanntlich wird zwar das Environment des Elternprozesses „nach unten“ weitergereicht, jedoch gibt es keinen umgekehrten Weg. Auch der Effekt der Kommandos (z. B. Verzeichniswechsel) ist nur innerhalb des Kindprozesses gültig. Im Elternprozess bleibt alles beim Alten. Das gilt natürlich auch für Zuweisungen an Variablen.

Die Kommunikation mit dem Elternprozess kann aber z. B. mit Dateien erfolgen. Bei kleinen Dateien spielt sich fast immer alles im Cache, also im Arbeitsspeicher ab und ist somit nicht so ineffizient, wie es zunächst den Anschein hat. Außerdem liefert jedes Programm einen Rückgabewert, der vom übergeordneten Prozess ausgewertet werden kann. **Beachten Sie, dass sich die Shell genau umgekehrt verhält wie fast alle Programmiersprachen, bei denen ein Wert gleich 0 „falsch“ und ein Wert ungleich 0 „wahr“ bedeutet.**

- `= 0`: O. K.
- `> 0`: Fehlerstatus

Es gibt außerdem ein Kommando, das ein Shell-Skript in der aktuellen Shell und nicht in einer Subshell ausführt:

```
{\bf.}    (Dot) Skript ausführen
```

Das *Dot*-Kommando erlaubt die Ausführung eines Skripts in der aktuellen Shell-Umgebung, z. B. das Setzen von Variablen usw. Es funktioniert also prinzipiell wie das `include`-Makro in der Programmiersprache C.

Damit die Variable auch in Subshells (d. h. beim Aufruf von Skripten) auch wirksam wird, muss sie exportiert werden:

```
export PATH
```

Alle exportierten Variablen bilden das Environment für die Subshells.

4.8.9 Interaktive Eingaben in Shell-Skripten

Es können auch Shell-Skripten mit interaktiver Eingabe geschrieben werden, indem das `read`-Kommando verwendet wird.

```
read variable [variable ...]
```

`read` liest eine Zeile von der Standardeingabe und weist die einzelnen Felder den angegebenen Variablen zu. Feldtrenner sind die in `IFS` definierten Zeichen. Sind mehr Variablen als Eingabefelder definiert, werden die überzähligen Felder mit Leerstrings besetzt. Umgekehrt nimmt die letzte Variable den Rest der Zeile auf. Wird im Shell-Skript die Eingabe mit < aus einer Datei gelesen, bearbeitet `read` die Datei zeilenweise.

Anmerkung: Da das Shell-Skript in einer Subshell läuft, kann `IFS` im Skript umdefiniert werden, ohne dass es nachher restauriert werden muss. Die Prozedur `zeige` enthält beispielsweise folgende Befehle:

```
IFS=','
echo "Bitte drei Parameter, getrennt durch Kommata, eingeben:"
read A B C
echo Eingabe war: $A $B $C
```

Aufruf:

```
user$ zeige
Bitte drei Parameter, getrennt durch Komma eingeben:
eins,zwei,drei
Eingabe war: eins zwei drei
```

4.8.10 Hier-Dokumente

Die Shell bietet die Möglichkeit, Eingaben für Programme direkt in das Shell-Skript mit aufzunehmen – womit die Möglichkeit einer zusätzlichen, externen Datei wegfällt.

Eingeleitet werden so genannte Hier-Dokumente mit `<<` und einer anschließenden Zeichenfolge, die das Ende des Hier-Dokuments anzeigt. Diese Zeichenfolge steht dann alleine am Anfang einer neuen Zeile (und gehört nicht mehr zum Hier-Dokument). Bei Quoting der Ende-Zeichenfolge (eingeschlossen in Gänsefüßchen bzw. Apostrophe) werden die Datenzeilen von den üblichen Ersetzungsmechanismen ausgeschlossen. Dazu ein Beispiel:

Das Shell-Skript `hier` enthält folgende Zeilen:

```
cat << EOT
Dieser Text wird ausgegeben, als ob er von
einer externen Datei kaeme - na ja, nicht ganz so.
Die letzte Zeile enthaelt nur das EOT und wird
nicht mit ausgegeben. Die folgende Zeile wuerde
bei der Eingabe aus einer Datei nicht ersetzt.
Parameter: $*
EOT
```

Aufruf:

```
user$ hier eins zwei
Dieser Text wird ausgegeben, als ob er von
einer externen Datei kaeme - na ja, nicht ganz so.
Die letzte Zeile enthaelt nur das EOT und wird
nicht mit ausgegeben. Die folgende Zeile wuerde
bei der Eingabe aus einer Datei nicht ersetzt.
Parameter: eins zwei
```

Außerden wäre bei der Eingabe aus einer Datei die Ersetzung von `$*` durch die aktuellen Parameter nicht möglich. Hier-Dokumentebieten also weitere Vorteile. Durch in die Kommandos eingestreute Variablen wird das Ganze variabel steuerbar. Noch ein Beispiel, diesmal die Simulation des Kommandos `wall`.

```
for  X in ‘who | cut -d’ ’ -f1‘
do
write $X << TEXTENDE
Hallo Leute,
das Wetter ist schoen. Wollen wir da nicht um 17 Uhr Schluss machen
und in den Biergarten gehen?
TEXTENDE
done
```

4.8.11 Verkettung von Kommandos

- **Hintereinanderausführung**: Will man mehrere Kommandos ausführen lassen, braucht man nicht jedes Kommando einzeln einzugeben und mit der Eingabe des nächsten Kommandos auf die Beendigung des vorhergehenden zu warten. Die Kommandos werden, getrennt durch Semikolon, hintereinander geschrieben:

```
Kommando1 ; Kommando2 ; Kommando3
```

- **Sequenzielles UND**: Das zweite Kommando wird nur dann ausgeführt, wenn das erste erfolgreich war:

```
Kommando1 && Kommando2
```

- **Sequenzielles EXOR**: Im Falle des Exklusiven ODER wird das zweite Kommando nur dann ausgeführt, wenn das erste erfolglos war:

```
Kommando1 || Kommando2
```

4.8.12 Zusammenfassung von Kommandos

Kommandofolgen lassen sich – analog der Blockstruktur höherer Sprachen – logisch klammern. Das Problem der normalen Hintereinanderausführung mit Trennung durch Semikolon ist die Umleitung von Standardeingabe und Standardausgabe, z. B.:

```
pwd > out ; who >> out ; ls >> out
```

Die Umleitung lässt sich auch auf die Fehlerausgabe erweitern:

```
echo "Fehler!"      # geht nach stdout
echo "Fehler!" 1>&2 # geht nach stderr
```

Kommandos lassen sich zur gemeinsamen E/A-Umleitung mit {…} klammern:

```
{ Kommando1 ; Kommando2 ; Kommando3 ; ... ; }
```

Wichtig: Die geschweiften Klammern müssen von Leerzeichen eingeschlossen werden!

Die Ausführung der Kommandos erfolgt nacheinander innerhalb der aktuellen Shell, die Ausgabe kann gemeinsam umgelenkt werden, z. B.:

```
{ pwd ; who ; ls ; } > out
```

Die schließende Klammer muss entweder durch ein Semikolon vom letzten Kommando getrennt werden oder am Beginn einer neuen Zeile stehen. Die geschweiften Klammern sind ein ideales Mittel, die Ausgabe aller Programme eines Shell-Skripts gemeinsam umzuleiten, wenn das Skript beispielsweise mit `nohup` oder über `cron` gestartet wird. Man fügt lediglich am Anfang eine Zeile mit der öffnenden Klammer ein und am Schluss die gewünschte Umleitung, z. B.:

```
{
. . .
. . .
. . .
} | mailx -s öutput from Foo" $LOGNAME
```

Eine Folge von Kommandos kann aber auch in einer eigenen Subshell ausgeführt werden:

```
( Kommando1 ; Kommando2 ; Kommando3 ; ... )
```

Das Ergebnis des letzten ausgeführten Kommandos wird als Ergebnis der Klammer zurückgegeben. Auch hier kann die Umleitung der Standard-Handles gemeinsam erfolgen. Auch dazu ein Beispiel. Bei unbewachten Terminals lässt sich schön an der Datei `.profile` des Users „basteln“. Eine Zeile

```
( sleep 300 ; audioplay /home/sounds/telefon.au ) &
```

ist schnell eingebaut. Fünf Minuten nach dem Login rennt dann jemand zum Telefon (geht natürlich nur, wenn der Computer auch soundfähig ist). Noch gemeiner wäre

```
( sleep 300 ; kill -9 0 ) &
```

Abschließend vielleicht noch etwas Nützliches: Wenn Sie feststellen, dass eine Plattenpartition zu klein geworden ist, müssen Sie nach Einbau und Formatierung einer neuen Platte oftmals ganze Verzeichnisbäume von der alten Platte auf die neue kopieren. Auch hier hilft die Kommandoverkettung zusammen mit dem Programm *tar* (*Tape ARchive*), das es nicht nur erlaubt, einen kompletten Verzeichnisbaum auf ein Gerät, etwa einen Streamer, zu kopieren, sondern auch in eine Datei oder auf die Standardausgabe. Man verknüpft einfach zwei *tar*-Prozesse, von denen der erste das Verzeichnis archiviert und der zweite über eine Pipe das Archiv wieder auspackt. Der Trick daran ist, dass beide Prozesse in verschiedenen Verzeichnissen arbeiten. Angenommen, Sie wollen das Verzeichnis `/usr/local` nach `/mnt` kopieren:

```
( cd /usr/local ; tar cf - . ) > ( cd /mnt ; tar xvf - )
```

Der Parameter `f` weist *tar* an, auf eine Datei zu schreiben oder von einer Datei zu lesen. Hat die Datei wie oben den Namen `-`, handelt es sich um `stdout` bzw. `stdin`.

4.8.13 Strukturen der Shell

In diesem Abschnitt werden die Programmstrukturen (Bedingungen, Schleifen etc.) behandelt. Zusammen mit den Shell-Variablen und den E/A-Funktionen `echo`, `cat` und `read` hat man nahezu die Funktionalität einer Programmiersprache. Es fehlen lediglich strukturierte Elemente wie z. B. Arrays und Records, die teilweise in anderen Shells (z. B. Korn-Shell) oder auch in Skript-Sprachen realisiert sind.

4.8.14 Bedingungen testen

Das wichtigste Kommando ist `test`, mit dem man mannigfache Bedingungen testen kann.

`test` *`Argument`* prüft eine Bedingung und liefert `true` (`0`), falls die Bedingung erfüllt ist, und `false` (`1`), falls die Bedingung nicht erfüllt ist. Der Fehlerwert `2` wird nurückgegeben, wenn das Argument syntaktisch falsch ist (meist durch Ersetzung hervorgerufen). Es lassen sich Dateien, Zeichenketten und Integer-Zahlen (16 Bit, bei Linux 32 Bit) überprüfen.

Das *`Argument`* von `test` besteht aus einer Testoption und einem Operanden, der ein Dateiname oder eine Shell-Variable (Inhalt: String oder Zahl) sein kann. In bestimmten Fällen können auf der rechten Seite eines Vergleichs auch Strings oder Zahlen stehen – bei der Ersetzung von leeren Variablen kann es aber zu Syntaxfehlern kommen. Weiterhin lassen sich mehrere Argumente logisch verknüpfen (UND, ODER, NICHT). Beispiel:

```
test -w /etc/passwd
```

Mit der Kommando-Verkettung lassen sich so schon logische Entscheidungen treffen, z. B:

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

Normalerweise kann statt `test` das Argument auch in eckigen Klammern gesetzt werden. Die Klammern müssen von Leerzeichen umschlossen werden:

```
[ -w /etc/passwd ]
```

Die Operationen von Tabelle 4.6 können bei `test` bzw. `[ … ]` verwendet werden.

Tabelle 4.6

Bedingungen für `test`

Ausdruck	Bedeutung
`-e <datei>`	`datei` existiert
`-r <datei>`	`datei` existiert und Leserecht
`-w <datei>`	`datei` existiert und Schreibrecht
`-x <datei>`	`datei` existiert und Ausführungsrecht
`-f <datei>`	`datei` existiert und ist einfache Datei
`-d <datei>`	`datei` existiert und ist Verzeichnis
`-h <datei>`	`datei` existiert und ist symbolischer Link
`-c <datei>`	`datei` existiert und ist zeichenorientiertes Gerät
`-b <datei>`	`datei` existiert und ist blockorientiertes Gerät
`-p <datei>`	`datei` existiert und ist benannte Pipe
`-u <datei>`	`datei` existiert und für Eigentümer s-Bit gesetzt
`-g <datei>`	`datei` existiert und für Gruppe s-Bit gesetzt
`-k <datei>`	`datei` existiert und t- oder sticky-Bit gesetzt
`-s <datei>`	`datei` existiert und ist nicht leer
`-L <datei>`	`datei` ist symbolischer Link
`-t <d.kennzahl>`	`d.kennzahl` ist einem Terminal zugeordnet
`-n <String>`	wahr, wenn `String` nicht leer
`-z <String>`	wahr, wenn `String` leer
`<String1>=<String2>`	wahr, wenn Zeichenketten gleich
`<String1>!=<String2>`	wahr, wenn Zeichenketten verschieden

Ausdruck	Bedeutung
`-eq`	(*equal*) gleich
`-ne`	(*not equal*) ungleich
`-ge`	(*greater than or equal*) größer gleich
`-gt`	(*greater than*) größer
`-le`	(*less than or equal*) kleiner gleich
`-lt`	(*less than*) kleiner
UND-Verknüpfung	`<Bedingung1> -a <Bedingung2>`
ODER-Verknüpfung	`<Bedingung1> -o <Bedingung2>`
Negation	`! <Ausdruck>`
Klammern	`\( <Ausdruck> \)`

4.8.15 Bedingte Anweisung (`if - then - else`)

Wichtig: Als Bedingung kann nicht nur der `test`-Befehl, sondern eine beliebige Folge von Kommandos verwendet werden. Jedes Kommando liefert einen Error-Code zurück, der bei erfolgreicher Ausführung gleich Null (`true`) und bei einem Fehler oder Abbruch ungleich Null (`false`) ist. Zum Testen einer Bedingung dient die `if`-Anweisung. Jede Anweisung muss entweder in einer eigenen Zeile stehen oder durch ein Semikolon von den anderen Anweisungen getrennt werden. Trotzdem verhalten sich bedingte Anweisungen – oder auch die Schleifenkonstrukte, die weiter unten behandelt werden – wie eine einzige Anweisung. Somit ergibt sich eine starke Ähnlichkeit mit der Blockstruktur von *C* oder *Pascal*. Man kann dies ausprobieren, indem man eine `if`- oder `while`-Anweisung interaktiv eingibt. Solange nicht `fi` bzw. `done` eingetippt wurde, erhält man den PS2-Prompt (>).

Einseitiges `if`

```
if kommandoliste
then
   kommandos
fi
```

Zweiseitiges `if`

```
if kommandoliste then    kommandos
else
   kommandos
fi
```

Mehrstufiges `if`

```
if kommandoliste1 then
   kommandos
```

```
elif kommandoliste2    then
      kommandos elif ...
      ... fi
```

Beispiele für die `if`-Anweisung

Zum Anfang ein „if-Test“ (einige der Kommandos in den Beispielen werden erst später behandelt):

```
echo Bitte eine erste Zahl:
read x1
echo Bitte eine zweite Zahl:
read x2
if [ $x1 -lt $x2 ]
  then
  echo $x1 ist kleiner als $x2
else
  if [ $x1 -eq $x2 ]
    then echo "x1 = x2  \(beide sind gleich $x1\)"
  else echo $x1 ist groesser $x2
  fi
fi
```

Das folgende Skript testet, ob eine als Aufrufparameter angegebene Datei vorhanden ist:

```
echo -n "Die Datei $1 ist "
if [ ! \( -f $1 -o -d $1 \) ]
  then echo -n "nicht "
fi
echo "vorhanden"
```

Es soll eine Meldung ausgegeben werden, falls mehr als fünf Benutzer eingeloggt sind:

```
USERS=‘who | wc -l‘  # Zeilen der who-Ausgabe zählen
if test $USERS -gt 5
then
  echo "Mehr als 5 Benutzer am Geraet"
fi
```

Das geht natürlich auch kürzer und ohne Backtics:

```
if [ $(who | wc -l) -gt 5 ] ; then
  echo "Mehr als 5 Benutzer am Geraet"
fi
```

Man sollte bei der Entwicklung von Skripten aber ruhig mit der Langfassung beginnen und sich erst der Kurzfassung zuwenden, wenn man mehr Übung hat und die Langfassungen auf Anhieb funktionieren. Ein weiteres Beispiel zeigt eine Fehlerprüfung:

```
if test $# -ne 2
then
  echo "usage: sortiere quelldatei zieldatei"
```

```
else
  sort +1 -2 $1  > &2
fi
```

Das nächste Beispiel zeigt eine mehr oder weniger intelligente Anzeige für Dateien und Verzeichnisse. `show` zeigt bei Dateien den Inhalt mit `less` an, während Verzeichnisse mit `ls` präsentiert werden. Fehlt der Parameter, wird interaktiv nachgefragt:

```
if [ $# -eq 0 ]                          # falls keine Angabe
then                                     # interaktiv erfragen
   echo -n "Bitte Namen eingeben: "
   read DATEI
else
   DATEI=$1
fi
if  [-f $DATEI ]                         # wenn normale Datei
then                                     # dann ausgeben
   less $DATEI
elif [ -d $DATEI ]                       # wenn aber Verzeichnis
  then                                   # dann Dateien zeigen
     ls -CF $DATEI
   else                                  # sonst Fehlermeldung
     echo "cannot show $DATEI"
fi
```

Das nächste Beispiel hängt eine Datei an eine andere Datei an; vorher erfolgt eine Prüfung der Zugriffsberechtigungen. Unser Kommando `append` *`Datei1 Datei2`*:

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

Beim Vergleich von Zeichenketten sollten möglichst die Anführungszeichen (’’ … ’’) verwendet werden, da sonst bei der Ersetzung durch die Shell unvollständige Test-Kommandos entstehen können. Dazu ein Beispiel:

```
if [ ! -n $1 ] ; then
   echo "Kein Parameter"
fi
```

Ist `$1` wirklich nicht angegeben, wird das Kommando reduziert zu:

```
if [ ! -n ] ; then ....
```

Es ist also unvollständig und es erfolgt eine Fehlermeldung. Dagegen liefert

```
if [ ! -n "$1" ] ; then
    echo "Kein Parameter"
fi
```

bei fehlendem Parameter den korrekten Befehl `if [ ! -n " ... " ]`. Bei fehlenden Anführungszeichen werden auch führende Leerzeichen der Variablenwerte oder Parameter eliminiert.

Noch ein Beispiel: Es kommt bisweilen vor, dass eine *Userid* wechselt oder dass die Gruppenzugehörigkeit von Dateien geändert werden muss. In solchen Fällen helfen die beiden folgenden Skripten:

```
!/bin/sh
# Change user-id
#
if [ $# -ne 2 ] ; then
  echo "usage ‘basename $0‘ <old id> <new id>"
  exit
fi
find / -user $1 -exec chown $2 {} ";"
```

```
#!/bin/sh
# Change group-id
#
if [ $# -ne 2 ] ; then
  echo "usage ‘basename $0‘ <old id> <new id>"
  exit
fi
find / -group $1 -exec chgrp $2 {} ";"
```

4.8.16 Mehrfachauswahl mit `case`

Diese Anweisung wird auch deshalb gerne verwendet, weil sie Muster mit Jokerzeichen und mehrere Muster für eine Auswahl erlaubt.

```
case selector in
   Muster-1) Kommandofolge1 ;;
   Muster-2) Kommandofolge2 ;;
   ...    Muster-n) Kommandofolgen ;;
esac
```

Die Variable `selector` (*String*) wird der Reihe nach mit den Mustern *`Muster-1`* bis *`Muster-n`* verglichen. Bei Gleichheit wird die nachfolgende Kommandofolge ausgeführt und dann nach der `case`-Anweisung (also hinter dem `esac`) fortgefahren.

- In den Mustern sind Metazeichen (`*`, `?`, `[]`) erlaubt, im Selektor dagegen nicht.
- Das Muster `*` deckt sich mit jedem Selektor (Default-Ausgang) und muss als letztes Muster in der `case`-Konstruktion stehen.
- Vor der Klammer können mehrere Muster, getrennt durch `|`, stehen. Das Zeichen `|` bildet eine Oder-Bedingung:

```
case selector in
   Muster-1) Kommandofolge1 ;;
   Muster-2 | Muster3) Kommandofolge2 ;;
   *) Kommandofolge3 ;;
esac
```

Beispiel: Menü mit interaktiver Eingabe:

```
while :  # Endlosschleife (s. später)
do
tput clear  # Schirm löschen und Menütext ausgeben
   echo " +---------------------------------+"
   echo " | 0 --> Ende                      |"
   echo " | 1 --> Datum und Uhrzeit         |"
   echo " | 2 --> aktuelles Verzeichnis     |"
   echo " | 3 --> Inhaltsverzeichnis        |"
   echo " | 4 --> Mail                      |"
   echo "+---------------------------------+"
   echo "Eingabe: \c"  # kein Zeilenvorschub
   read ANTW
   case $ANTW in
    0) kill -9 0 ;; # und tschuess
    1) date ;;
    2) pwd ;;
    3) ls -CF ;;
    4) elm ;;
    *) echo "Falsche Eingabe!" ;;
   esac
done
```

4.8.17 Die `for`-Anweisung

Diese Schleifenanweisung hat zwei Ausprägungen, mit einer Liste der zu bearbeitenden Elemente oder mit den Kommandozeilen-Parametern.

Die `for`-Schleife mit Liste

```
for selector in liste
   do
   Kommandofolge
done
```

Die Selektor-Variable wird nacheinander durch die Elemente der Liste ersetzt und die Schleife mit der Selektor-Variablen ausgeführt. Beispiele:

```
for X in hans heinz karl luise # vier Listenelemente
do
  echo $X
done
```

Das Programm hat folgende Ausgabe:

```
hans
heinz
karl
luise
```

```
for FILE in *.txt # drucke alle Textdateien
```

```
  do              # im aktuellen Verzeichnis
  lpr $FILE
done

for XX in $VAR    # geht auch mit
  do
  echo $XX
done
```

Die `for`-Schleife mit Kommandozeilen-Parametern

```
for selector
  do
  Kommandofolge
done
```

Die Selektor-Variable wird nacheinander durch die Parameter `$1` bis `$n` ersetzt und mit diesen Werten die Schleife durchlaufen. Es gibt also `$#` Schleifendurchläufe. Beispiel:

Das Skript `makebak` erzeugt für die in der Parameterliste angegebenen Dateien eine `.bak`-Datei:

```
for FF
  do
  cp $FF ${FF}.bak
done
```

4.8.18 Abweisende Wiederholungsanweisung (`while`)

Als Bedingung kann nicht nur eine „klassische“ Bedingung (`test` oder `[]`), sondern selbstverständlich auch der Ergebniswert eines Kommandos oder einer Kommandofolge verwendet werden.

```
while Bedingung
  do
  Kommandofolge
done
```

Solange der Bedingungsausdruck den Wert `true` liefert, wird die Schleife ausgeführt. Beispiele:

Warten auf eine Datei (z. B. vom Hintergrundprozess):

```
while [ ! -f foo ]
  do
  sleep 10  # Wichtig damit die Prozesslast nicht zu hoch wird
done
```

Pausenfüller für das Terminal – Abbruch mit Strg-C:

```
while :
  do
  tput clear  # BS löschen
  echo "\n\n\n\n\n" # 5 Leerzeilen
  banner $(date '+ \%T ') # Uhrzeit groß
  sleep 1  # 1s Pause
done
```

Umbenennen von Dateien durch Anhängen eines Suffixes:

```
# Aufruf change suffix datei(en)
if [ $# -lt 2 ] ; then
   echo "usage: ‘basename $0‘ suffix file(s)"
else
   SUFF=$1                 # Suffix speichern
   shift
   while [ $# -ne 0 ]      # solange Parameter da sind
     do
     mv $1 ${1}.$SUFF      # umbenennen
     shift
   done
fi
```

Umbenennen von Dateien durch Anhängen eines Suffixes – Variante 2 mit `for`:

```
 Aufruf change suffix datei(en)
if [ $# -lt 2 ] ; then
   echo "usage: ‘basename $0‘ suffix file(s)"
else
   SUFF=$1                       # Suffix speichern
   shift
   for FILE
     do
     mv $FILE ${FILE}.$SUFF      # umbenennen
     shift
   done
fi
```

4.8.19 `until`-Anweisung

Diese Anweisung ist identisch zu einer `while`-Schleife mit negierter Bedingung. Als Bedingung kann nicht nur eine „klassische“ Bedingung (`test` oder `[]`), sondern selbstverständlich auch der Ergebniswert eines Kommandos oder einer Kommandofolge verwendet werden:

```
until Bedingung
  do
  Kommandofolge
done
```

Die Schleife wird solange abgearbeitet, bis *Bedingung* einen Wert ungleich Null liefert. Beispiele:

```
# warten, bis sich der Benutzer hans eingeloggt hat
until [ ‘who | grep -c "hans" -gt 0 ]
  do
  sleep 10
done
```

4.8.20 Weitere Anweisungen

- `exit`
  Wie die interaktive Shell kann auch ein Shell-Skript mit `exit` abgebrochen werden. Vom Terminal aus kann mit der Taste Strg-C abgebrochen werden, sofern das Signal nicht abgefangen wird (siehe `trap`).
- `break [n]`
  Verlassen von `n` umfassenden Schleifen; Voreinstellung für `n` ist `1`.
- `continue [n]`
  Beginn des nächsten Durchgangs der `n`-ten umfassenden Schleife, d. h. der Rest der Schleife(n) wird nicht mehr ausgeführt; Voreinstellung für `n` ist `1`.
- Interne Kommandos
  Etliche der besprochenen Shell-Kommandos starten nicht, wie sonst üblich, einen eigenen Prozess, sondern sie werden direkt von der Shell interpretiert und ausgeführt. Teilweise ist keine E/A-Umleitung möglich. Einige Kommandos der folgenden Auswahl wurden bereits besprochen, andere werden weiter unten behandelt. Zum Teil gibt es interne und externe Versionen, z. B. `echo` (intern) und `/bin/echo` (extern).
  - `break` Schleife verlassen
  - `continue` Sprung zum Schleifenanfang
  - `echo` Ausgabe
  - `eval` Mehrstufige Ersetzung
  - `exec` Überlagerung der Shell durch ein Kommando
  - `exit` Shell beenden
  - `export` Variablen für Subshells bekannt machen
  - `read` Einlesen einer Variablen
  - `shift` Parameterliste verschieben
  - `trap` Behandlung von Signalen
- `set [Optionen] [Parameterliste]`
  Setzen von Shell-Optionen und Positionsparametern (`$1 … $n`). Einige Optionen:

  `v` gibt die eingelesenen Shell-Eingaben auf dem Bildschirm aus.

  `x` gibt alle Kommandos vor der Ausführung aus (zeigt Ersetzungen).

  `n` liest die Kommandos von Shell-Skripten, führt sie jedoch nicht aus.

  Der Aufruf von `set` ohne Parameter liefert die aktuelle Belegung der Shell-Variablen. Außerdem kann `set` verwendet werden, um die Positionsparameter zu besetzen. `set eins zwei drei vier` besetzt die Parameter mit `$1=eins`, `$2=zwei`, `$3=drei` und `$4=vier`. Da dabei auch Leerzeichen, Tabs, Zeilenwechsel und anderes „ausgefiltert“ wird (genauer alles, was in der Variablen `IFS` steht), ist `set` manchmal einfacher zu verwenden als die Zerlegung einer Zeile mit `cut`. Die Belegung der Parameter kann auch aus einer Variablen (z. B. `set $VAR`) oder aus dem Ergebnis eines Kommando-Aufrufs erfolgen. Beispiel:

```
set ‘date‘          # $1=Thu $2=Nov $3=10 $4=10:44:16 $5=MEZ $6=2005
```

```
echo "Es ist $4 Uhr"
Es ist 10:44:16 Uhr
```

Aber es gibt Fallstricke: Wenn man beispielsweise den Output von `ls` bearbeiten möchte, gibt es zunächst unerklärliche Fehlermeldungen (`set: unknown option`):

```
ls -l > foo
echo "Dateiname Laenge"
while read LINE
  do
  set $LINE
  echo $9 $5
done < foo
rm foo
```

Da die Zeile mit dem Dateityp und den Zugriffsrechten beginnt und für normale Dateien ein „-“ am Zeilenbeginn steht, erkennt `set` eine falsche Option (z. B. `-rwxr-xr-x`). Abhilfe schafft das Voranstellen eines Buchstabens:

```
ls -l > foo
echo "Dateiname Laenge"
while read LINE
  do
  set Z$LINE
  echo $9 $5
done < foo
rm foo
```

Weitere Beispiele: Wenn ein Benutzer eingeloggt ist, wird ausgegeben, seit wann. Sonst erfolgt eine Fehlermeldung.

```
if HELP=‘who | grep $1‘
then
  echo -n "$1 ist seit "
  set $HELP
  echo "$5 Uhr eingeloggt."
else
  echo "$1 ist nicht auffindbar"
fi
```

Ersetzen der englischen Tagesbezeichung durch die deutsche:

```
set ‘date‘
case $1 in
  Tue) tag=Die;;
  Wed) tag=Mit;;
  Thu) tag=Don;;
  Sat) tag=Sam;;
  Sun) tag=Son;;
  *)   tag=$1;;
esac
echo $tag $3.$2 $4 $6 $5
```

Arithmetik in Skripten

Die `expr`-Anweisung erlaubt das Auswerten von arithmetischen Ausdrücken. Das Ergebnis wird in die Standardausgabe geschrieben. Als Zahlen können 16-Bit-Integerzahlen (beim Ur-Unix) oder 32-Bit-Integerzahlen (bei Linux) verwendet werden (bei manchen Systemen auch noch längere Zahlen mit 64 Bit).

Die *Bash* wurde auch an dieser Stelle erweitert. Mit der doppelten Klammerung `$(( Ausdruck ))` kann man rechnen, ohne ein externes Programm aufzurufen. `expr Ausdruck` und `$(( Ausdruck ))` beherrschen die vier Grundrechenarten:

Arithmetische Operationen

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Divisionsrest (Modulo-Operator)

Die Priorität „Punkt vor Strich“ gilt auch hier. Außerdem können Klammern gesetzt werden. Da die Klammern und der Stern auch von der Shell verwendet werden, müssen diese Operationszeichen immer durch den Backslash geschützt werden: `\*` , `\( ... \)`

Damit die Operatoren von der Shell erkannt werden, müssen sie von Leerzeichen eingeschlossen werden, zum Beispiel eine Zuweisung der Summe von `A` und `B` an `X` durch:

```
X=‘expr $A + $B‘
```

oder

```
X=$((expr $A + $B))
```

(Backquotes beachten!) Außerdem sind logische Operationen implementiert, die den Wert `0` für `wahr` und den Wert `1` für `falsch` liefern (siehe folgende Tabelle).

Vergleichsoperationen

`expr1 | expr2`	oder
`expr1 & expr2`	und
`expr1 < expr2`	kleiner
`expr1 <= expr2`	kleiner oder gleich
`expr1 > expr2`	größer
`expr1 >= expr2`	größer oder gleich
`expr1 = expr2`	gleich
`expr1 != expr2`	unleich

Beispiel: Nimm-Spiel, interaktiv:

```
ANZ=0
if test $# -ne 1
then
  echo "usage: $0  Startzahl"
else
  echo "NIM-Spiel als Shell-Skript"
  echo "Jeder Spieler nimmt abwechselnd 1, 2 oder 3 Hoelzer"
  echo "von einem Haufen, dessen Anfangszahl beim Aufruf festgelegt"
  echo "wird. Wer das letzte Holz nimmt, hat verloren."
  echo
  ANZ=$1
  while [ $ANZ -gt 1 ]                # bis nur noch 1 Holz
    do                                # da ist wiederholen
    echo "\nNoch $ANZ Stueck. Du nimmst (1 - 3): \c # Benutzer
    read N
    if [ $N -lt 1 -o $N -gt 3 ] ; then # Strafe bei Fehleingabe
      N=1
    fi
    ANZ=‘expr $ANZ - $N‘              # Benutzer nimmt N weg
    if [ $ANZ -eq 1 ] ; then          # Computer muß letztes Holz nehmen
      echo "\nGratuliere, Du hast gewonnen"
      exit                            # Prozedur verlassen
    else
      C=‘expr \( $ANZ + 3 \) \% 4     # Computerzug berechnen
      if [ $C -eq 0 ] ; then
        C=1                           # Wenn 0 Verlustposition
      fi
      echo "Es bleiben $ANZ Stueck. Ich nehme ${C}.\c"
      ANZ=‘expr $ANZ - $C‘            # Computerzug abziehen
      echo " Rest $ANZ"
    fi
  done                                # Dem Benutzer bleibt
  echo "\nIch habe gewonnen"          # das letzte Holz
fi
```

4.8.21 exec [*Kommandozeile*]

Ähnlich wie beim Dot-Kommando wird keine Subshell erzeugt, sondern die Kommandozeile in der aktuellen Umgebung ausgeführt. Eine erste Anwendung liegt darin, das aktuelle Programm durch ein anderes zu überlagern. Wenn Sie z. B. die *Bourne-Shell* als Login-Shell haben, aber lieber mit der *Bash* arbeiten, können Sie die *Bourne-Shell* durch die Kommandozeile

```
exec /bin/bash
```

als letzte Zeile in der `.profile`-Datei durch die *Bash* ersetzen. (Wenn Sie die *Bash* nur aufrufen würden, müßten Sie beide Shells beenden, um sich auszuloggen.) Das Kommando entspricht also dem Systemcall `exec()`. Wird jedoch kein Kommando

angegeben, kann die E/A der aktuellen Shell dauerhaft umgeleitet werden. Beispiel:

```
exec 2>fehler
```

leitet alle folgenden Fehlerausgaben in die Datei `fehler` um, bis die Umleitung explizit durch

```
exec 2>-
```

zurückgenommen wird. Es können bei `exec` auch andere Dateideskriptoren verwendet werden. Ebenso kann auch die Dateiumleitung einer Eingabedatei erfolgen, z. B.:

```
exec 3< datei
```

Danach kann mit `read <&3` von dieser Datei gelesen werden, bis die Umleitung mit `exec 3<-` wieder zurückgenommen wird. Man kann also in Shell-Skripten durch das Einfügen einer `exec`-Anweisung die Standardausgabe/-eingabe global für das gesamte Skript umleiten, ohne weitere Änderungen vornehmen zu müssen (eine andere Möglichkeit wäre die oben beschriebene Verwendung von { }).

4.8.22 `eval [`*`Argumente`*`]`

Das Kommando `eval` liest seine Argumente, wobei die üblichen Ersetzungen stattfinden, und führt die resultierende Zeichenkette als Kommando aus. Die Argumente der Kommandozeile werden von der Shell gelesen, wobei Variablen- und Kommando- sowie Dateinamenersetzungen durchgeführt werden. Die sich ergebende Zeichenkette wird anschließend erneut von der Shell gelesen, wobei wiederum die oben genannten Ersetzungen durchgeführt werden. Schließlich wird das resultierende Kommando ausgeführt. Beispiel:

```
user$ A='Hello world!'
user$ X='$A'
user$ echo $X
$A
user$ eval echo $X
Hello world!
```

Der Rückgabestatus von `eval` ist der Rückgabestatus des ausgeführten Kommandos oder `0`, wenn keine Argumente angegeben wurden. Ein weiteres Beispiel:

```
user$ cat /etc/passwd | wc -l
76
user$ foo='cat /etc/passwd'
user$ bar=' | wc -l'
user$ $foo $bar
cat: | : No such file or directory
cat: wc: No such file or directory
cat: -l: No such file or directory
user$ eval $foo $bar
76
```

In diesem Beispiel wird zunächst ein einfaches Kommando gestartet, das die Anzahl der Zeilen der Datei `/etc/passwd` bestimmt. Anschließend werden die beiden Teile des gesamten Kommandos in die zwei Shell-Variablen `foo` und `bar` aufgeteilt. Der erste Aufrufversuch `$foo $bar` bringt nicht das gewünschte Ergebnis, sondern lediglich einige Fehlermeldungen, da in diesem Fall der Wert von `bar` als Argument für `foo` interpretiert wird (`cat` wird mit den Dateien `/etc/passwd`, `|`, `wc` und `-l` aufgerufen). Wird jedoch das Kommando `eval` auf die Argumente `$foo` und `$bar` angewendet, werden diese zunächst zur Zeichenkette `cat /etc/passwd | wc -l` ersetzt. Diese Zeichenkette wird dann durch das Kommando `eval` erneut von der Shell gelesen, die jetzt das Zeichen `|` in der Kommandozeile als Pipe-Symbol erkennt und das Kommando ausführt. Das Kommando `eval` wird üblicherweise dazu eingesetzt, eine Zeichenkette als Kommando zu interpretieren, wobei zweifach Ersetzungen in den Argumenten der Kommandozeile vorgenommen werden.

Eine andere Anwendung ist beispielsweise die Auswahl des letzten Parameters der Kommandozeile. Mit `\$$#` erhält man die Parameterangabe (bei fünf Parametern `$5`). Das erste Dollarzeichen wird von der Shell ignoriert (wegen des Backslash), `$#` hingegen ausgewertet. Durch `eval` wird der Ausdruck nochmals ausgewertet, man erhält so den Wert des letzten Parameters:

```
eval \$$#
```

Aber Vorsicht, das funktioniert nur bei 1-9 Parametern, denn z. B. der zwölfte Parameter führt zu `$12` → `${1}2`. Es lassen sich mit `eval` sogar Pointer realisieren. Falls die Variable `PTR` den Namen einer anderen Variablen, z. B. `XYZ`, enthält, kann auf den Wert von `XYZ` durch `eval $PTR` zurückgegriffen werden, z. B. durch

```
eval echo \$$PTR
```

4.8.23 trap *´Kommandoliste´ Signale*

Ausführen der Kommandoliste, wenn eines der angegebenen Signale an den Prozess (= Shell) gesendet wird. Die Signale werden in Form der Signalnummern oder über ihre Namen (`SIGKILL`, `SIGHUP` etc.), getrennt durch Leerzeichen, aufgeführt.

Ist die Kommandoliste leer, werden die entsprechenden Signale abgeschaltet. Bei einfachen Kommandos reichen oft auch die Anführungszeichen, um die Shell-Ersetzung zu verhindern.

Signale sind eine Möglichkeit, über die verschiedenen Prozesse, also laufende Programme, miteinander kommunizieren können. Ein Prozess kann einem anderen Prozess ein Signal senden (der Betriebssystemkern spielt dabei den „Postboten“). Der Empfängerprozess reagiert auf das Signal, z. B. dadurch, dass er sich beendet. Der Prozess kann das Signal auch ignorieren. Das ist beispielsweise nützlich, wenn ein Shell-Skript nicht durch den Benutzer von der Tastatur aus abgebrochen werden soll. Mit dem `trap`-Kommando kann man festlegen, mit welchen Kommandos auf ein Signal reagiert werden soll bzw. ob überhaupt reagiert werden soll. Die Datei `/usr/include/Signal.h` enthält eine Liste aller Signale. Neben anderen können folgende Signalnummern verwendet werden:

Typische Signale

0 (SIGKILL)	Kill: Beenden der Shell
1 (SIGHUP)	Hangup: Beenden der Verbindung zum Terminal/Modem
2 (SIGINT)	Interrupt: Wie Strg - C am Terminal
3 (SIGQUIT)	Quit: Beenden von der Tastatur aus
9 (SIGKILL)	Kill: Kann nicht abgefangen werden; beendet immer den empfangenden Prozess
15 (SIGTERM)	Terminate: Software-Terminate, Voreinstellung

Beispiele:

```
 Skript sperren gegen Benutzerunterbrechung:
trap "" 2 3
```

oder auch

```
# Skript sauber beenden
trap 'rm tmpfile; cp foo xyz; exit' 0 2 3 15
```

Bitte nicht das `exit`-Kommando am Schluss vergessen, sonst wird das Skript nicht beendet. Wiedereinschalten der Signale erfolgt durch `trap [`*Signale*`]`. Ein letztes Beispiel zu `trap`:

```
# Automatisches Ausführen des Shell-Skripts .logoff beim
# Ausloggen durch den folgenden Eintrag in .profile:
trap .logoff 0
```

4.8.24 Shell-Funktionen

Shell-Funktionen bieten eine weitere Strukturierungsmöglichkeit. Funktionen können in Shell-Skripten, aber auch interaktiv definiert werden. Sie lassen sich jedoch nicht wie Variablen exportieren, gelten also nur in der aktuellen Shell. Sie werden nach folgender Syntax definiert:

```
Funktionsname ()
   {
   Kommandofolge
   }
```

Steht die schließende geschweifte Klammer nicht in einer eigenen Zeile, gehört ein Semikolon davor. Die runden Klammern hinter dem Funktionsnamen teilen dem Kommandozeilen-Interpreter der Shell mit, dass nun eine Funktion definiert werden soll (und nicht ein Kommando *Funktionsname* aufgerufen wird). Es kann keine Parameterliste in den Klammern definiert werden.

Der Aufruf der Shell-Funktion erfolgt durch Angabe des Funktionsnamens, gefolgt von Parametern (genauso wie der Aufruf eines Skripts). Die Parameter werden innerhalb der Funktion genauso wie beim Aufruf von Shell-Skripten über `$1` bis `$n` angesprochen. Ein Wert kann mit der Anweisung `return <`*Wert*`>` zurückgegeben werden, er ist über den Parameter `$?` abfragbar. Beispiel:

```
isdir () # testet, ob $1 ein Verzeichnis ist
   {
   if [ -d $1 ] ; then
     echo "$1 ist ein Verzeichnis" # Kontrolle zum Test
     return 0
   else
     return 1
    fi
   }
```

Im Gegensatz zum Aufruf von Shell-Skripten werden Funktionen *in der aktuellen Shell* ausgeführt und sie können bei der *Bourne-Shell* nicht exportiert werden; die *Bash* erlaubt dagegen das Exportieren mit `export -f`. Das folgende Beispiel illustriert die Eigenschaften von Shell-Funktionen.

Die folgende Funktion gibt den Eingangsparameter in römischen Zahlen aus. Dabei wird die Zahl Schritt für Schritt in der Variablen `ZAHL` zusammengesetzt. Würde man bei der Funktion `ZIFF` ein Skript verwenden, ginge das nicht, da sich der Wert von `ZAHL` ja nicht aus dem aufgerufenen Skript heraustransportieren ließe.

```
#
# Ausgabe des Eingangsparameters $1 in roemischen Ziffern
#
ZIFF ()

   # Funktion zur Bearbeitung einer einstelligen Ziffer $1
   # Einer-, Zehner-, Hunderterstelle unterscheiden sich nur
   # durch die verw. Zeichen $2: Einer, $3: Fuenfer, $4: Zehner
   { X=$1
   if test $X -eq 9; then
      ZAHL=${ZAHL}$2$4
   elif test $X -gt 4; then
      ZAHL=${ZAHL}$3
      while test $X -ge 6; do
         ZAHL=${ZAHL}$2 ; X=‘expr $X - 1‘
      done
   elif test $X -eq 4; then
    ZAHL=${ZAHL}$2$3
  else
      while test $X -gt 0; do
          ZAHL=${ZAHL}$2 ; X=‘expr $X - 1‘
      done
    fi
    }
if test $# -eq 0; then
      echo "usage: roem Zahl"; exit
fi
XX=$1
while test $XX -gt 999; do
      ZAHL=${ZAHL}"M"; XX=‘expr $XX - 1000‘
done
```

```
ZIFF ‘expr $XX / 100‘ C D M
XX=‘expr $XX \% 100‘
ZIFF ‘expr $XX / 10‘ X L C
ZIFF ‘expr $XX \% 10‘ I V X
echo "$ZAHL \n"
```

4.8.25 xargs

Das xargs-Programm ist ein eigenständiges Kommando, das hier behandelt wird, weil es – ebenso wie das unten behandelte find-Kommando sehr häufig in Shell-Skripten verwendet wird. xargs übergibt alle aus der Standardeingabe gelesenen Daten einem Programm als zusätzliche Argumente. Der Programmaufruf wird einfach als Parameter von xargs angegeben:

```
xargs Programm [Parameter]
```

Ein Beispiel soll die Funktionsweise klar machen:

```
user$ ls *.txt | xargs echo Textdateien
```

Hier erzeugt ls eine Liste aller Dateien mit der Endung .txt im aktuellen Verzeichnis. Das Ergebnis wird über die Pipe an xargs weitergereicht. xargs ruft echo mit den Dateinamen von ls als zusätzliche Parameter auf. Der Output ist dann

```
Textdateien: kap1.txt kap2.txt kap3.txt h.txt
```

Durch Optionen ist es möglich, die Art der Umwandlung der Eingabe in Argumente durch xargs zu beeinflussen. Mit der Option -n <*Nummer*> wird eingestellt, mit wievielen Parametern das angegebene Programm aufgerufen werden soll. Fehlt der Parameter, nimmt xargs die maximal mögliche Zahl von Parametern. Je nach Anzahl der Parameter ruft xargs das angegebene Programm einmal oder mehrfach auf. Dazu ein Beispiel: Vergleich einer Reihe von Dateien nacheinander mit einer vorgegebenen Datei:

```
user$ ls *.dat | xargs -n1 cmp compare Muster
```

Die Vergleichsdatei Muster wird der Reihe nach mittels cmp mit allen Dateien verglichen, die auf .dat enden. Die Option -n1 veranlasst xargs, je Aufruf immer nur einen Dateinamen als zusätzliches Argument bei cmp anzufügen.

Mit der Option - i <*Zeichen*> ist es möglich, an einer beliebigen Stelle im Programmaufruf, auch mehrfach, anzugeben, wo die eingelesenen Argumente einzusetzen sind. In diesem Modus liest xargs jeweils ein Argument aus der Standardeingabe, ersetzt im Programmaufruf jedes Vorkommen des hinter -i angegebenen Zeichens durch dieses Argument und startet das Programm. In dem folgenden Beispiel wird dies benutzt, um alle Dateien mit der Endung .txt in .bak umzubenennen.

```
user$ ls *.txt | cut -d. f1 | xargs -iP mv P.txt P.bak
```

Das Ganze funktioniert allerdings nur, wenn die Dateien nicht noch weitere Punkte im Dateinamen haben.

xargs ist überall dann notwendig, wenn die Zahl der Argumente recht groß werden kann. Obwohl Linux extrem lange Kommandozeilen zulässt, ist die Länge doch

begrenzt. `xargs` nimmt immer so viele Daten aus dem Eingabestrom, wie in eine Kommandozeile passen, und führt dann das gewünschte Kommando mit diesen Parametern aus. Liegen weitere Daten vor, wird das Kommando entsprechend oft aufgerufen. Insbesondere mit dem folgenden Kommando sollte `xargs` verwendet werden, da `find` immer den gesamten Dateipfad liefert, also schnell recht lange Argumente weitergibt.

4.8.26 `find`

Durchsuchen der Platte nach bestimmten Dateien. Dieses Kommando ist sehr mächtig und besitzt zahlreiche Optionen, welche die Bedingungen für die Dateiauswahl (= Treffer) festlegen. Die angegebenen Pfade werden rekursiv durchsucht, d. h. auch Unterverzeichnisse und Unter-Unterverzeichnisse usw. Aufrufschema:

```
find Pfadname(n) Bedingung(en) Aktion(en)
```

Bei den nachfolgenden Optionen (= Bedingungen) steht das `N` für eine Zahlenangabe (ganze Zahl). Das Vorzeichen dieser Zahl bestimmt die Bedingung:

`N` genau `N`

`+N` mehr als `N`

`-N` weniger als `N`

Bedingungen (es werden nur die wichtigsten Optionen aufgeführt):

- `-name Dateiname`
  Suche nach bestimmten Dateien (bei Verwendung von Metazeichen wie `*` oder `?` den Namen unbedingt in '…' einschließen).
- `-type T`
  Suche nach einem bestimmten Dateityp *T*:
  - `f` normale Datei
  - `d` Directory
  - `b` Block Device
  - `c` Character Device
  - `p` Named Pipe
- `-perm onum`
  Suche nach Dateien mit den durch die Oktalzahl *onum* angegebenen Zugriffsrechten. Steht das Zeichen `-` vor *onum*, werden nicht alle, sondern nur die spezifizierten Rechte geprüft (z. B. `Test SUID: -perm -4000`).
- `-links N`
  Suche nach Dateien mit einer bestimmten Anzahl *N* von Links
- `-user Kennung`
  Suche nach Dateien eines bestimmten Users; es kann der Login-Name oder die UID angegeben werden.
- `-group Kennung`
  Wie `-user`, jedoch für Gruppen

- `-size N`
  Suche nach Dateien mit *N* Blöcken
- `-mount`
  Suche nur auf dem aktuellen Datenträger (wichtig, falls weitere Platten über NFS eingebunden sind)
- `-newer Dateiname`
  Suche nach Dateien, die jünger sind als die angegebene

Aktionen (auch hier nur die wichtigsten):

- `-print`
  Ausgabe der gefundenen Dateinamen auf die Standardausgabe
- `-exec Kommando`
  Ausführen eines Kommandos. Wird innerhalb des Kommandos die leere geschweifte Klammer {} aufgeführt, so wird anstelle der geschweiften Klammern der jeweils gefundene absolute Pfad der Datei eingesetzt. Das per `-exec` aufgerufene Kommando wird immer mit einem geschützten Semikolon (`\;` oder `';'` oder `';'`) abgeschlossen.
- `-ok Kommando`
  Wie `-exec`, jedoch mit Sicherheitsabfrage

Beispiele:

```
user$ find / -name '*.c' -print
```

Beginnt, beim Wurzelverzeichnis, alle *C*-Quellen (`.c`) zu suchen und gibt die Namen (und Pfade) auf dem Bildschirm aus. Die Apostrophe verhindern die Ersetzung der Angabe `*.c` durch die Shell.

```
user$ find -user markus -print
```

Sucht alle Dateien von User markus

```
user$ find . -name dat1 -print
```

Sucht nach allen Vorkommen der Datei `dat1` ab dem aktuellen Verzeichnis

```
user$ find . -print -name dat1
```

Gibt *alle* Dateinamen ab dem aktuellen Verzeichnis aus. *Vorsicht Falle*: Da die Bedingungen von links nach rechts ausgewertet werden und `-print` immer `wahr` ist, hat der Teil `-name dat1` keine Wirkung.

```
user$ find . -name '*.bak' -exec rm  \;
```

Sucht alle `.bak`-Dateien ab dem aktuellen Verzeichnis und löscht sie

```
user$ find /usr -size +2000 -print
```

Gibt alle Dateien der Benutzer mit mehr als 2000 Blöcken aus (z. B. um die Platzverschwender zu mahnen)

```
user$ find . -name '*.bak' -ok rm  \;
```

Sucht alle `.bak`-Dateien ab dem aktuellen Verzeichnis und löscht sie nur, wenn die Nachfrage mit `y` beantwortet wurde.

```
user$ find / -user markus -exec rm  \;
```

Löscht alle Dateien von User markus, wo auch immer sie stehen. Verwendet man `xargs`, sieht das Kommando so aus:

```
user$ find . -user markus -print | xargs rm
```

4.9 Beispiele für Shell-Skripten

Die folgenden Beispiele sollen Ihnen den Weg in die Shell-Programmierung ebnen. Es sind meist relativ kurze Programme, die bestimmte Aspekte illustrieren.

4.9.1 Datei verlängern

… weil immer wieder danach gefragt wird: „Wie hänge ich den Inhalt einer Variablen an eine Datei an?“

```
( cat file1 ; echo "$SHELLVAR" ) > file2
```

4.9.2 Telefonbuch

Man braucht nicht viel Programm, um eine Telefonliste zu verwalten. Sucht man nach einer Nummer, wird das Skript mit einem Namen oder einer Namensliste als Parameter aufgerufen. Damit die Telefonliste nicht verloren gehen kann, hängt sie gleich als Hier-Dokument am Skript dran. Nebenbei – selbst bei etlichen hundert Einträgen ist `grep` immer noch schnell genug:

```
if [ $# -eq 0 ]
then
   echo "usage: ‘basename $0‘ Name [Name ..]"
   exit 2
fi
for SUCH in $*
do
    if [ ! -z $SUCH ] ; then
       grep $SUCH << "EOT"
       Hans  123456
       Fritz 234561
       Karl  345612
       Egon  456123
EOT
   fi
done
```

4.9.3 Argumente mit J/N-Abfrage ausführen

Das folgende Skript führt alle Argumente nach vorheriger Abfrage aus. Mit `j` wird die Ausführung bestätigt, mit `q` das Skript abgebrochen und mit jedem anderen Buchstaben (in der Regel `n`) ohne Ausführung zum nächsten Argument übergegangen. Ein-

und Ausgabe erfolgen immer über das Terminal(-fenster), weil `/dev/tty` angesprochen wird. Das Skript wird anstelle der Argumentenliste bei einem anderen Kommando eingesetzt, z. B. Löschen mit Nachfrage durch `rm $(pick *)`

```
# pick - Argumente mit Abfrage liefern
for I ; do
   echo "$I (j/n)? \c" > /dev/tty
   read ANTWORT
   case $ANTWORT in
     j*|J*) echo $I ;;
     q*|Q*) break ;;
   esac
done </dev/tty
```

4.9.4 Dateien im Pfad suchen

Das folgende Skript bildet das Kommando *which* nach. Es sucht im aktuellen Pfad (durch `PATH` spezifiziert) nach der angegebenen Datei und gibt die Fundstelle aus. An diesem Skript kann man auch eine Sicherheitsmaßnahme sehen. Für den Programmaufruf wird der Pfad neu gesetzt, damit nur auf Programme aus `/bin` und `/usr/bin` zugegriffen wird. Bei Skripten, die vom Systemverwalter für die Allgemeinheit erstellt werden, sollte man entweder so verfahren oder alle Programme mit absolutem Pfad aufrufen.

```
#!/bin/sh
# Suchen im Pfad nach einer Kommando-Datei
OPATH=$PATH
PATH=/bin:/usr/bin
if [ $# -eq 0 ] ; then
   echo "usage: which kommando" ; exit 1
fi
for FILE
do
   for I in `echo $OPATH | sed -e 's/^:/.:/' -e 's/::/:.:/g
                                                    -e 's/:$/:./'`
   do
      if [ -f "$I/$FILE" ] ; then
        ls -ld "$I/$FILE"
      fi
   done
done
```

4.9.5 Berechnung des Osterdatums nach C. F. Gauss

Eigentlich braucht sowas niemand als Shell-Skript. Das Beispiel soll lediglich zeigen, dass man auch komplexere Ganzzahl-Rechnungen mit der Shell erledigen kann.

```
if [ $# -eq 0 ] ; then
    echo "Osterdatum fuer Jahr: \c"; read JAHR
```

```
else
   JAHR="$1"
fi
G=‘expr $JAHR \% 19 + 1‘
C=‘expr $JAHR / 100 + 1‘
X=‘expr \( $C / 4 - 4 \) \* 3‘
Z=‘expr \( $C \* 8 + 5 \) / 25 - 5‘
D=‘expr $JAHR \* 5 / 4 - $X - 10‘
E=‘expr \( $G \* 11 + $Z - $X + 20 \) \% 30‘
if test $E -lt 0; then
   $E=‘expr $E + 30‘
fi
if [ $E -eq 25 -a $G -gt 11 -o $E -eq 24 ] ; then
   E=‘expr $E + 1‘
fi
TAG=‘expr 44 - $E‘
if [ $TAG -lt 21 ] ; then
   TAG=‘expr $TAG + 30‘
fi
TAG=‘expr $TAG + 7 - \( $D + $TAG \) \% 7‘
if [ $TAG -gt 31 ] ; then
   TAG=‘expr $TAG - 31‘
   MON=4
else
   MON=3
fi
echo "Ostern $JAHR ist am ${TAG}.${MON}.\n"
```

Statt des `expr`-Befehls kann bei der *Bash* auch das Konstrukt `$(( ... ))` verwendet werden. Das Programm sieht dann so aus:

```
if [ $# -eq 0 ] ; then
    echo "Osterdatum fuer Jahr: \c"; read JAHR
else
   JAHR="$1"
fi
G=$(($JAHR \% 19 + 1))
C=$(($JAHR / 100 + 1))
X=$((\( $C / 4 - 4 \) \* 3))
Z=$((\( $C \* 8 + 5 \) / 25 - 5))
D=$(($JAHR \* 5 / 4 - $X - 10))
E=$((\( $G \* 11 + $Z - $X + 20 \) \% 30))
if test $E -lt 0; then
   $E=$(($E + 30))
fi
if [ $E -eq 25 -a $G -gt 11 -o $E -eq 24 ] ; then
   E=$(($E + 1))
fi
TAG=$((44 - $E))
if [ $TAG -lt 21 ] ; then
```

```
   TAG=$(($TAG + 30))
fi
TAG=$(($TAG + 7 - \( $D + $TAG \) \% 7))
if [ $TAG -gt 31 ] ; then
   TAG=$(($TAG - 31))
   MON=4
else
   MON=3
fi
echo "Ostern $JAHR ist am ${TAG}.${MON}.\n"
```

4.9.6 Wem die Stunde schlägt …

Wer im Besitz einer Soundkarte ist, kann sich eine schöne Turmuhr, einen Regulator oder Big Ben basteln. Die folgende „Uhr“ hat Stunden- und Viertelstundenschlag. Zuvor ist jedoch ein Eintrag in der `crontab` notwendig:

```
0,15,30,45 * * * * /home/sbin/turmuhr
```

So wird das Skript `turmuhr` jede Viertelstunde aufgerufen. Es werden zwei Sounddateien verwendet, `hour.au` für den Stundenschlag und `quarter.au` für den Viertelstundenschlag. Statt des Eigenbau-Programms *audioplay* kann auch der *sox* verwendet werden oder man kopiert die Dateien einfach nach `/dev/audio`. Die Variable `VOL` steuert die Lautstärke.

```
#!/bin/sh
BELL=/home/local/sounds/hour.au
BELL1=/home/local/sounds/quater.au
PLAY=/usr/bin/audioplay
VOL=60
DATE=‘date +\%H:\%M‘
MINUTE=‘echo $DATE | sed -e ’s/.*://’‘
HOUR=‘echo $DATE | sed -e ’s/:.*//’‘

if [ $MINUTE = 00 ]
then
        COUNT=‘expr \( $HOUR \% 12 + 11 \) \% 12‘
        BELLS=$BELL
        while [ $COUNT != 0 ];
        do
                BELLS="$BELLS $BELL"
                COUNT=‘expr $COUNT - 1‘
        done
        $PLAY -v $VOL -i $BELLS
elif [ $MINUTE = 15 ]
then    $PLAY -v $VOL -i $BELL1
elif [ $MINUTE = 30 ]
then    $PLAY -v $VOL -i $BELL1 $BELL1
elif [ $MINUTE = 45 ]
then    $PLAY -v $VOL -i $BELL1 $BELL1 $BELL1
```

```
else
        $PLAY -v $VOL -i $BELL1
fi
```

4.9.7 Eingabe ohne Enter-Taste

Soll nur eine Taste zur Bestätigung gedrückt werden, z. B. `j` oder `n`, lässt sich das mit dem `read`-Kommando nicht realisieren, da die Eingabe immer mit der ↵-Taste abgeschlossen werden muss. Um eine Eingabe ohne ↵ zu realisieren, sind zwei Dinge notwendig:

- Schalten des Terminals auf ungepufferte Eingabe
- Einlesen der gewünschten Anzahl von Zeichen

Das Umschalten des Terminals wird mit `stty raw` erreicht. Für die Eingabe wird das `dd`-Kommando (*Disk Dump*) zweckentfremdet. `dd` liest von der Standardeingabe und schreibt auf die Standardausgabe. Für die geplante Aktion werden die Parameter `count` (Anzahl zu lesender Blöcke) und `bs` (*blocksize*) verwendet. `count` enthält die Anzahl der einzulesenden Zeichen, `bs` wird auf `1` gesetzt. Der entstehende Programmteil sieht dann so aus:

```
echo "Alles Loeschen (j/n)\c"
stty raw -echo
INPUT=‘dd count=1 bs=1 2> /dev/null‘
stty -raw echo
echo $INPUT
case $INPUT in
  j|J) echo "Jawoll" ;;
  n|N) echo "Doch nicht" ;;
  *)   echo "Wat nu?" ;;
esac
```

4.9.8 Ständig kontrollieren, wer sich ein- und ausloggt

```
#!/bin/sh
# PATH=/bin:/usr/bin
NEW=/tmp/WW1.WHO
OLD=/tmp/WW2.WHO
>$OLD                                # OLD neu anlegen
while :                              # Endlosschleife
do
   who >$NEW
   diff $OLD $NEW
   mv $NEW $OLD
    sleep 60
done
```

4.9.9 Optionen ermitteln

Oft ist es wünschenswert, bei Shell-Skripten – auf die gleiche Weise wie bei Programmen – auch Optionen angeben zu können. Die Optionen bestehen aus einem

Buchstaben mit dem - davor. Bei manchen Optionen folgt auch eine durch die Option spezifizierte Angabe (z. B. beim pr-Kommando der Header). Das folgende Fragment zeigt, wie sich solche Optionen behandeln lassen. Die einzige Einschränkung besteht darin, dass sich mehrere Optionen nicht zusammenziehen lassen (-abc statt -a -b -c geht also nicht). Damit alle Optionen über eine Schleife abgehandelt werden können, wird mit shift gearbeitet. Wie üblich, können nach den Optionen noch Dateinamen folgen. Ein Testaufruf könnte lauten:

```
otest -a -p Parameter -c *.txt

#!/bin/sh
# Bearbeiten von Optionen in Shellskripts
# Beispiel: -a -b -c  als einfache Optionen
#           -p <irgend ein Parameter> als "Spezial-Option"
READOPT=0
while [ $READOPT -eq 0 ] ; do     # solange Optionen vorhanden
  case $1 in
      -a) echo öption a"
          shift ;;
      -b) echo öption b"
          shift ;;
      -c) echo öption c"
          shift ;;
      -p) PARAM=$2 ; shift  # Parameter lesen
          echo öption p: $PARAM"
          shift ;;
       *) if ‘echo $1 | grep -s ’^-’‘ ; then # Parm. beginnt mit ’-’
            echo ünknown option $1"
            shift
          else
            READOPT=1  # Ende Optionen, kein shift!
  esac
done
echo "Restliche Parameter : $*"
```

4.9.10 rename-Kommando

So mächtig das mv-Kommando auch ist, es bietet keine Möglichkeit, eine ganze Reihe von Dateien nach gleichem Schema umzubenennen (z. B. kap??.txt zu kapitel??.txt). Das folgende Skript leistet dies. Die ersten beiden Parameter enthalten den ursprünglichen Namensteil (z. B. kap) und den neuen Namensteil (z. B. kapitel). Danach folgt die Angabe der zu bearbeitenden Dateien. Wenn die Zieldatei schon existiert, wird nicht umbenannt, sondern eine Fehlermeldung ausgegeben.

```
#!/bin/sh
# Alle Dateien umbennen, die durch $3 - $n spezifiziert werden
# dabei wird der String $1 im Dateinamen durch $2 ersetzt,
# wobei auch regulaere Ausdruecke erlaubt sind
if [ $# -lt 3 ] ; then
   echo ’Usage: ren <old string> <new string> files’
```

```
   echo 'Example:  ren foo bar *.foo  renames all files'
   echo '             *.foo ---> *.bar'
   exit 1
fi
S1=$1 ; shift
S2=$1 ; shift
while [ $# -gt 0 ]; do
   for OLDF in $1 ; do
      NEWF=`echo $OLDF | sed -e "s/${S1}/${S2}/\qut{
      if [ -f $NEWF ] ; then
           echo "$NEWF exists, $OLDF not renamed"
       else
        echo "renaming $OLDF to $NEWF"
        mv $OLDF $NEWF
      fi
   done
   shift
done
```

4.9.11 Rekursives Suchen in Dateien

Manchmal muss man auf eine ganze Gruppe von Dateien die gleiche Textmanipulation (Suchen und Ersetzen) anwenden, etwa wenn man sich entschlossen hat alle Dateien der eigenen Web-Präsenz endlich mal mit CSS-Styles zu versehen. Das folgende Skript rattert durch einen Dateibaum und führt die Ersetzung bei jeder Datei durch. Der Aufruf von `rsar` (recursive search and replace) benötigt vier Parameter:

```
rsar <start-dir> <file-expression> <search-pattern> <replace-pattern>
```

Das `start-dir` legt fest, welches Verzeichnis rekursiv durchsucht wird, `file-expression` erlaubt die Auswahl bestimmter Dateien (z. B. *.txt) und das Such- und Ersetzungsmuster entspricht dem, was man auch beim `sed` verwenden würde. Hier wird jedoch nicht der `sed`, sondern der `vi` im Kommandozeilenmodus (`ex`) verwendet.

```
#!/bin/sh
PROGNAME=`basename $0`
TEMPDAT=/tmp/`basename $0.$$`
if test $# -lt 4; then
   echo "$PROGNAME : Recursive search-and-replace-skript."
   echo "usage : $PROGNAME <dir> <file-expr> <search> <replace>"
   echo "Both patterns use ex/vi-syntax !"
else
   find $1 -type f -name "$2" -print > $TEMPDAT
   for NAME in `cat $TEMPDAT`
     do
     echo -n "Processing $NAME.."
     ex $NAME << EOT > /dev/null
1,\$ s/$3/$4/g
wq
EOT
     echo "done."
```

```
    done
    rm $TEMPDAT
fi
```

4.9.12 Das Letzte

Das letzte Shell-Programm in diesem Kapitel heißt „Unix-Camping“ und ist nicht ganz ernst zu nehmen:

```
( unzip; strip; touch; finger; mount; gasp; yes; umount; sleep )
```

4.10 Referenz Shell-Programmierung

Die nachfolgende Referenz liefert in alphabetischer Reihenfolge eine kurze Beschreibung der wichtigsten Kommandos zur Shell-Programmierung (ohne jeden Anspruch auf Vollständigkeit!). Bei vielen der hier besprochenen Kommandos handelt es sich um eingebaute `bash`-Kommandos, zum Teil aber auch um reguläre Linux-Kommandos, die sich besonders zur Anwendung bei der Shell-Programmierung eignen.

Variablenverwaltung und Parameterauswertung

`alias`	definiert eine Abkürzung
`declare`	definiert eine (Umgebungs-)Variable
`export`	definiert eine Umgebungsvariable
`local`	definiert lokale Variablen in einer Funktion
`read`	liest eine Variable ein
`readonly`	zeigt alle schreibgeschützten Variablen an
`shift`	verschiebt die Parameterliste
`unalias`	löscht eine Abkürzung
`unset`	löscht eine Variable

Umgang mit Zeichenketten

`basename`	ermittelt den Dateinamen eines Pfads
`dirname`	ermittelt das Verzeichnis eines Pfads
`expr`	führt einen Mustervergleich durch

Verzweigungen, Schleifen

`break`	beendet eine Schleife vorzeitig
`case`	leitet eine Fallunterscheidung ein
`continue`	überspringt den Schleifenkörper
`exit`	beendet das Shell-Programm
`for`	leitet eine Schleife ein
`function`	definiert eine neue Funktion
`if`	leitet eine Verzweigung ein
`local`	definiert lokale Variablen in einer Funktion
`return`	beendet eine Funktion
`source`	führt die angegebene Shell-Datei aus
`test`	wertet eine Bedingung aus
`until`	leitet eine Schleife ein
`while`	leitet eine Schleife ein

Ausgabe

`cat`	gibt eine Datei auf dem Bildschirm aus
`echo`	gibt eine Zeile Text aus
`printf`	ermöglicht eine formatierte Ausgabe wie unter C
`setterm`	verändert die Schriftart, löscht den Bildschirm

Sonstiges

`dialog`	zeigt eine Dialogbox an
`dirs`	zeigt die Liste der gespeicherten Verzeichnisse an
`eval`	wertet das angegebene Kommando aus
`file`	versucht, den Typ einer Datei festzustellen
`popd`	wechselt in das letzte gespeicherte Verzeichnis
`pushd`	speichert das aktuelle und wechselt in ein neues Verzeichnis
`sleep`	wartet eine vorgegebene Zeit
`trap`	führt beim Eintreten eines Signals ein Kommando aus
`ulimit`	kontrolliert die von der Shell beanspruchten Ressourcen
`wait`	wartet auf das Ende eines Hintergrundprozesses

`alias abkürzung='kommando'`

`alias` definiert eine Abkürzung.

`basename zeichenkette [endung]`

`basename` liefert den Dateinamen des übergebenen Pfads. `basename /usr/man/man1/gnroff.1` führt also zum Ergebnis `gnroff.1`. Wenn als zusätzlicher Parameter eine Dateiendung angegeben wird, so wird diese (falls vorhanden) aus dem Dateinamen entfernt.

`break [n]`

`break` bricht eine `for`-, `while`- oder `until`-Schleife vorzeitig ab. Das Shell-Programm wird beim nächsten Kommando nach dem Schleifenende fortgesetzt. Durch die Angabe eines optionalen Zahlenwerts können *n* Schleifenebenen abgebrochen werden.

```
case ausdruck in
    muster1 ) kommandos;;
    muster2 ) kommandos;;
     ...
esac
```

`case` wird zur Bildung von Mehrfachverzweigungen verwendet, wobei als Kriterium für die Verzweigung eine Zeichenkette angegeben wird (zumeist eine Variable oder ein Parameter, der dem Shell-Programm übergeben wird). Diese Zeichenkette wird der Reihe nach mit den Mustern verglichen, wobei in diesen Mustern die Jokerzeichen für Dateinamen (`*?[]`) verwendet werden können. In einem `case`-Zweig können auch mehrere durch | getrennte Muster angegeben werden.

Sobald ein Muster zutrifft, werden die Kommandos ausgeführt, die zwischen der runden Klammer `)` und den beiden Semikola folgen. Anschließend wird das Programm nach `esac` fortgesetzt.

`cat << ende`

`cat` liest in dieser Syntaxvariante so lange Text der aktuellen Shell-Datei und zeigt diesen auf dem Bildschirm an, bis die Zeichenkette `ende` auftritt. `cat` kann damit zur bequemen Ausgabe größerer Textmengen verwendet werden, ohne für jede Zeile ein `echo`-Kommando ausführen zu müssen. Die Syntax `<< ende` kann aber natürlich auch für alle anderen Kommandos verwendet werden, um Text bis zur Zeichenkette `ende` aus der aktuellen Datei zu lesen.

`continue [n]`

`continue` überspringt den Körper einer `for`-, `while`- oder `until`-Schleife und setzt die Schleife mit dem nächsten Durchlauf fort. Durch den optionalen Zahlenwert kann dieser Vorgang auch für äußere Schleifenebenen durchgeführt werden.

`dialog [--clear] [--title text] [dialogoptionen]`

`dialog` zeigt einen Dialog an. Das Kommando hat genau genommen nichts mit der `bash` zu tun, sondern ist ein eigenständiges Programm zur textorientierten Ein- und Ausgabe von Daten. Es kann dazu verwendet werden, um `bash`-Scripts trotz der Einschränkungen durch den Textmodus ein etwas moderneres Aussehen zu geben.

Die Optionen `--clear` und `--title` können mit jeder der angegebenen Dialogoptionen kombiniert werden. `--clear` bewirkt, dass der Dialog nach seinem Ende von der Konsole entfernt wird. (Die gesamte Konsole ist anschließend blau! Führen Sie eventuell anschließend `setterm -clear` aus, um die Default-Farben wiederherzustellen.) `--title` ermöglicht die Einstellung des Dialogtitels. `dialog` kennt die folgenden Dialogformen:

- **--msgbox text höhe breite** zeigt einen Meldungstext an, der mit ↵ bestätigt werden muss.
- **--infobox text höhe breite** wie oben, aber ohne Bestätigung. Das Programm wird sofort fortgesetzt. Die Dialogbox bleibt so lange auf dem Bildschirm stehen, bis dieser gelöscht wird.
- **--yesno text höhe breite** zeigt eine Dialogbox für eine Ja/Nein-Entscheidung an.
- **--inputbox text höhe breite** ermöglicht die Eingabe einer Textzeile.
- **--textbox datei höhe breite** zeigt die Textdatei an (ohne Editiermöglichkeit, aber mit Scrolling).
- **--menu text h b menühöhe menüpkt1 menütext1 mp2 mt2 ...** ermöglicht die Auswahl einer Option (eines Menüpunkts).
- **--checklist text höhe breite listenhöhe**
- **option1 text1 status1 option2 text2 status2 ...** a ermöglicht die gleichzeitige Auswahl mehrerer Optionen.

`dialog` liefert als Rückgabewert 0, wenn der Dialog mit Ok oder Yes beendet wurde, 1, wenn der Dialog mit Cancel oder No beendet wurde, oder 255, wenn der Dialog mit Esc beendet wurde. Bei den Dialogtypen, die einen Text, Menütitel oder eine Liste von Optionen als Ergebnis liefern, werden die Resultate in die Standardfehlerdatei geschrieben. Im Normalfall ist daher eine Umleitung in eine temporäre Datei mit `2> tmp` erforderlich. Nach dem Ende der Eingabe kann diese Datei ausgewertet werden.

dirname zeichenkette

`dirname` liefert den Pfad eines vollständigen Dateinamens. `dirname /usr/bin/groff` liefert also `/usr/bin`.

dirs

`dirs` zeigt die Liste der durch `pushd` gespeicherten Verzeichnisse an.

echo [optionen] zeichenkette

`echo` gibt die angegebene Zeichenkette auf dem Bildschirm aus. Die Zeichenkette sollte in doppelte oder einfache Hochkommata gestellt werden.

-e beachtet unter anderem die Sonderzeichen `\a` (Beep), `\n` (Zeilenende) und `\t` (Tabulator) in der Zeichenkette. Durch `echo -e "\a"` kann also ein Warnton ausgegeben werden.

-n wechselt beim Ende der Ausgabe nicht in eine neue Zeile. Die Ausgabe kann durch eine weitere `echo`-Anweisung fortgesetzt werden.

eval $var

`eval` interpretiert den Inhalt der Variablen als Kommandozeile, wertet diese Zeile aus (mit allen bekannten Substitutionsmechanismen) und führt das Kommando schließlich aus. `eval` ist immer dann erforderlich, wenn ein in einer Variablen gespeichertes Kommando ausgeführt werden soll und dieses Kommando diverse Sonderzeichen der Shell enthält.

Erst mit der Verwendung von `eval` kann das in der Variablen `kom` gespeicherte Kommando ausgeführt werden. Der erste Versuch, das Kommando auszuführen, scheitert, weil die `bash` das Pipe-Zeichen `|` nicht mehr auswertet, nachdem sie `$kom` durch seinen Inhalt ersetzt hat.

exec kommando

`exec` startet das angegebene Kommando als Ersatz zur laufenden `bash`. Das Kommando kann beispielsweise dazu verwendet werden, eine andere Shell zu starten. Die laufende Shell wird dadurch auf jeden Fall beendet. (Bei einem normalen Kommandostart bleibt die `bash` im Hintergrund aktiv, bis das Kommando beendet ist.)

exit [rückgabewert]

`exit` beendet ein Shell-Programm. Wenn kein Rückgabewert angegeben wird, gibt das Programm 0 (ok) zurück.

export [optionen] variable [=wert]

export deklariert die angegebene Shell-Variable als Umgebungs-Variable. Damit ist die Variable auch in allen aufgerufenen Kommandos und Subshells verfügbar. Optional kann dabei auch eine Variablenzuweisung erfolgen. Wenn das Kommando ohne Parameter aufgerufen wird, werden alle Umgebungs-Variablen angezeigt.

-n macht eine Umgebungs-Variable wieder zu einer normalen Shell-Variablen. Das Kommando hat damit genau die umgekehrte Wirkung wie bei der Verwendung ohne Optionen.

expr zeichenkette : muster

expr kann zur Auswertung arithmetischer Ausdrücke, zum Vergleich zweier Zeichenketten etc. eingesetzt werden. Von test und dem Substitutionsmechanismus $[...] unterscheidet sich expr insofern, als es in der oben angeführten Syntaxvariante Mustervergleiche für Zeichenketten durchführen kann (reguläre Ausdrücke).

file [optionen] datei

file versucht festzustellen, welchen Datentyp die als Parameter angegebene Datei hat. Als Ergebnis liefert file eine Zeichenkette mit dem Dateinamen und dem Typ der Datei. *Achtung*: Textdateien mit deutschen Sonderzeichen werden nicht als Textdateien klassifiziert, sondern als data. Die Klassifizierung basiert auf der Datei /etc/magic. file -z datei versucht den Datentyp einer komprimierten Datei zu erkennen.

```
for var [in liste;] do
    kommandos
done
```

for bildet eine Schleife. In die angegebene Variable werden der Reihe nach alle Listenelemente eingesetzt. Die Liste kann auch mit Jokerzeichen für Dateinamen oder mit {...}-Elementen zur Zusammensetzung von Dateinamen gebildet werden. Wenn auf die Angabe der Liste verzichtet wird, durchläuft die Variable alle der Shell-Datei übergebenen Parameter (also in $*).

```
[ function ] name()
    { kommandos }
```

function definiert eine Subfunktion, die innerhalb der Shell-Datei wie ein neues Kommando aufgerufen werden kann. Innerhalb der Funktion können mit local lokale Variablen definiert werden. Funktionen können rekursiv aufgerufen werden. Den einzelnen Funktionen können Parameter wie Kommandos übergeben werden. Innerhalb der Funktion können diese Parameter den Variablen $1, $2 etc. entnommen werden.

```
if bedingung; then
    kommandos
[elif bedingung; then
    kommandos]
```

```
[else
     kommandos]
fi
```

Das `if`-Kommando leitet eine Verzweigung ein. Der Block nach `then` wird nur ausgeführt, wenn die Bedingung erfüllt ist. Andernfalls werden (beliebig viele, optionale) `elif`-Bedingungen ausgewertet. Gegebenenfalls wird der ebenfalls optionale `else`-Block ausgeführt.

Als Bedingung können mehrere Kommandos angegeben werden. Nach dem letzten Kommando muss ein Semikolon folgen. Als Kriterium gilt der Rückgabewert des letzten Kommandos. Vergleiche und andere Tests können mit dem Kommando `test` durchgeführt werden. Statt `test` ist auch eine Kurzschreibweise in eckigen Klammern zulässig, dabei muss aber nach `[` und vor `]` jeweils ein Leerzeichen stehen.

`local var[=wert]`

`local` definiert eine lokale Variable. Das Kommando kann nur innerhalb einer selbst definierten Funktion verwendet werden (siehe `function`). Vor und nach dem Gleichheitszeichen dürfen keine Leerzeichen angegeben werden.

`popd`

`popd` wechselt in ein zuvor mit `pushd` gespeichertes Verzeichnis zurück. Das Verzeichnis wird aus der Verzeichnisliste entfernt.

`printf format para1 para2 para3 ...`

`printf` erlaubt es, Ausgaben in der Syntax des C-Kommandos *printf* zu formatieren. Detaillierte Informationen zu den Formatierungsmöglichkeiten erhalten Sie mit `man 3 printf`.

`pushd verzeichnis`

`pushd` speichert das aktuelle Verzeichnis und wechselt anschließend in das angegebene Verzeichnis. Mit `popd` kann in das ursprüngliche Verzeichnis zurückgewechselt werden. `dirs` zeigt die Liste der gespeicherten Verzeichnisse an.

`read [var1 var2 var3 ...]`

`read` liest eine Zeile Text in die angegebenen Variablen. `read` erwartet die Daten aus der Standardeingabe. Wenn keine Variable angegeben wird, schreibt `read` die Eingabe in die Variable `REPLY`. Wenn genau eine Variable angegeben wird, schreibt `read` die gesamte Eingabe in diese eine Variable. Wenn mehrere Variablen angegeben werden, schreibt `read` das erste Wort in die erste Variable, das zweite Wort in die zweite Variable ... und den verbleibenden Rest der Eingabe in die letzte Variable. Wörter werden dabei durch Leer- oder Tabulatorzeichen getrennt.

Das `read`-Kommando sieht keine Möglichkeit vor, einen Infotext als Eingabeaufforderung auszugeben. Deswegen ist es zweckmäßig, den Anwender vor der Ausführung von `read`-Kommandos mit `echo -n` über den Zweck der Eingabe zu informieren.

setterm [option]

setterm verändert diverse Einstellungen des Terminals. Wenn das Kommando ohne die Angabe einer Option ausgeführt wird, zeigt es eine Liste aller möglichen Optionen an. Nützliche Optionen zur Shell-Programmierung sind:

- **-bold on** | **off** aktiviert bzw. deaktiviert die fette Schrift. In Textkonsolen erscheint der Text zwar nicht fett, aber immerhin in einer anderen Farbe als der sonstige Text.
- **-clear** löscht den Inhalt des Terminals.
- **-default** stellt Farben und Textattribute auf die Default-Einstellung zurück.
- **-half-bright on** | **off** stellt hervorgehobene Schrift ein/aus.
- **-reverse on** | **off** stellt inverse Schrift ein/aus.
- **-underline on** | **off** stellt unterstrichene Schrift ein/aus.

shift [n]

shift schiebt die dem Shell-Programm übergebene Parameterliste durch die vordefinierten Variablen $1 bis $9. Wenn shift ohne Parameter verwendet wird, werden die Parameter um eine Position verschoben, andernfalls um *n* Positionen. shift ist besonders dann eine wertvolle Hilfe, wenn mehr als neun Parameter angesprochen werden sollen. *Achtung*: Einmal mit shift aus den Variablen geschobene Parameter können nicht mehr angesprochen werden. Sie werden auch aus der Variablen $* entfernt.

sleep zeit

sleep versetzt das laufende Programm für die angegebene Zeit in den Ruhezustand. Das Programm konsumiert in dieser Zeit praktisch keine Rechenzeit. Die Zeitangabe erfolgt normalerweise in Sekunden. Optional können die Buchstaben m, h oder d angehängt werden, um die Zeit in Minuten, Stunden oder Tagen anzugeben.

source datei

source führt die angegebene Shell-Datei so aus, als befänden sich die darin enthaltenen Kommandos an der Stelle des source-Kommandos. Nach der Ausführung der Datei wird das laufende Shell-Programm in der nächsten Zeile fortgesetzt. Zur Ausführung der angegebenen Datei wird keine neue Shell gestartet. Alle Variablen (inklusive der Parameterliste) gelten daher auch für die angegebene Datei. Wenn in dieser Datei exit ausgeführt wird, kommt es *nicht* zu einem Rücksprung in das Programm mit dem source-Kommando, sondern zu einem sofortigen Ende der Programmausführung. Zu source existiert die Kurzform . datei.

test ausdruck

test wird zur Formulierung von Bedingungen verwendet und zumeist in if-Abfragen und Schleifen eingesetzt. Je nachdem, ob die Bedingung erfüllt ist, liefert es den Wahrheitswert 0 (wahr) oder 1 (falsch). Statt test kann auch die Kurzschreibweise

`[ ausdruck ]` verwendet werden, wobei Leerzeichen vor und nach dem Ausdruck angegeben werden müssen.

Wenn *test* oder die Kurzschreibweise *[ausdruck]* als Bedingung in einer Verzweigung oder Schleife verwendet wird, muss die Bedingung mit einem Semikolon abgeschlossen werden, also z. B. `if [ "$1" = "abc" ]; then ...`

if-Abfragen können manchmal durch die Formulierung `test "$1" = "abc" && kommando` ersetzt werden. In diesem Fall ist kein Semikolon erforderlich. Das Kommando wird nur ausgeführt, wenn die vorherige Bedingung erfüllt war.

Zeichenketten

`[ zk ]`	wahr, wenn die Zeichenkette nicht leer ist
`[ -z zk ]`	wahr, wenn die Zeichenkette leer ist (0 Zeichen)
`[ zk1 = zk2 ]`	wahr, wenn die Zeichenketten übereinstimmen
`[ zk1 != zk2 ]`	wahr, wenn die Zeichenketten voneinander abweichen

Die Zeichenketten bzw. Variablen sollten in Hochkommata gestellt werden (z. B. `[ "$1" = "abc" ]` oder `[ "$a" = "$b" ]`). Andernfalls kann es bei Zeichenketten mit mehreren Wörtern zu Fehlern kommen.

Zahlen

`[ z1 -eq z2 ]`	wahr, wenn die Zahlen gleich sind (equal)
`[ z1 -ne z2 ]`	wahr, wenn die Zahlen ungleich sind (not equal)
`[ z1 -gt z2 ]`	wahr, wenn *z1* größer *z2* ist (greater than)
`[ z1 -ge z2 ]`	wahr, wenn *z1* größer gleich *z2* ist (greater equal)
`[ z1 -lt z2 ]`	wahr, wenn *z1* kleiner *z2* ist (less than)
`[ z1 -le z2 ]`	wahr, wenn *z1* kleiner gleich *z2* ist (less equal)

Dateien (auszugsweise)

`[ -d dat ]`	wahr, wenn es sich um ein Verzeichnis handelt (directory)
`[ -e dat ]`	wahr, wenn die Datei existiert (exist)
`[ -f dat ]`	wahr, wenn es sich um eine einfache Datei (und nicht um ein Device, ein Verzeichnis ...) handelt (file)
`[ -L dat ]`	wahr, wenn es sich um einen symbolischen Link handelt
`[ -r dat ]`	wahr, wenn die Datei gelesen werden darf (read)
`[ -s dat ]`	wahr, wenn die Datei mindestens 1 Byte lang ist (size)
`[ -w dat ]`	wahr, wenn die Datei verändert werden darf (write)
`[ -x dat ]`	wahr, wenn die Datei ausgeführt werden darf (execute)
`[ dat1 -ef dat2 ]`	wahr, wenn beide Dateien denselben I-Node haben (equal file)
`[ dat1 -nt dat2 ]`	wahr, wenn Datei 1 neuer als Datei 2 ist (newer than)

Verknüpfte Bedingungen

`[ ! bed ]`	wahr, wenn die Bedingung nicht erfüllt ist
`[ bed1 -a bed2 ]`	wahr, wenn beide Bedingungen erfüllt sind (and)
`[ bed1 -o bed2 ]`	wahr, wenn mindestens eine der Bedingungen erfüllt ist (or)

`trap [kommando] n`

`trap` führt das angegebene Kommando aus, wenn in der `bash` das angegebene Signal auftritt. Wenn kein Kommando angegeben wird, ignoriert das Programm bzw. die `bash` das betreffende Signal. `trap -l` liefert eine Liste aller möglichen Signale und der ihnen zugeordneten Kenn-Nummern.

`unalias abkürzung`

`unalias` löscht eine vorhandene Abkürzung. Wenn das Kommando mit der Option `-a` aufgerufen wird, löscht es alle bekannten Abkürzungen.

`unset variable`

`unset` löscht die angegebene Variable.

`until bedingung; do`

```
    kommandos
done
```

`until` dient zur Bildung von Schleifen. Die Schleife wird so lange ausgeführt, wie die angegebene Bedingung erfüllt ist. Das Schleifenkriterium ist der Rückgabewert des Kommandos, das als Bedingung angegeben wird. Vergleiche und Tests werden mit dem Kommando `test` oder dessen Kurzform in eckigen Klammern durchgeführt.

`wait [prozessnummer]`

`wait` wartet auf das Ende des angegebenen Hintergrundprozesses. Wenn keine Prozessnummer angegeben wird, wartet das Kommando auf das Ende aller laufenden, von der Shell gestarteten Hintergrundprozesse.

`while bedingung; do`

```
    kommandos
done
```

`while` dient zur Bildung von Schleifen. Die Schleife wird so lange ausgeführt, bis die angegebene Bedingung zum ersten Mal nicht mehr erfüllt ist. Das Schleifenkriterium ist der Rückgabewert des Kommandos, das als Bedingung angegeben wird. Vergleiche und Tests werden mit dem Kommando `test` oder dessen Kurzform in eckigen Klammern durchgeführt.

4.11 Referenz aller Sonderzeichen

`;`	trennt mehrere Kommandos
`:`	Shell-Kommando, das nichts tut
`.`	Shell-Programm ohne eigene Subshell starten (`. datei`) (entspricht `source datei`)
`#`	leitet einen Kommentar ein
`#!/bin/sh`	identifiziert die gewünschte Shell für das Shell-Programm
`&`	führt das Kommando im Hintergrund aus (`kom &`)
`&&`	bedingte Kommando-Ausführung (`kom1 && kom2`)

`&>`	Umleitung von Standardausgabe und -fehler (entspricht `>&`)
`|`	bildet Pipes (`kom1 | kom2`)
`||`	bedingte Kommando-Ausführung (`kom1 || kom2`)
`*`	Jokerzeichen für Dateinamen (beliebig viele Zeichen)
`?`	Jokerzeichen für Dateinamen (ein beliebiges Zeichen)
`[abc]`	Jokerzeichen für Dateinamen (ein Zeichen aus `abc`)
`[ ausdruck ]`	Kurzschreibweise für `test ausdruck`
`(...)`	Kommandos in derselben Shell ausführen (`(kom1; kom2)`)
`{...}`	Kommandos gruppieren
`~`	Abkürzung für das Heimatverzeichnis
`>`	Ausgabeumleitung in eine Datei (`kom > dat`)
`>>`	Ausgabeumleitung; an vorhandene Datei anhängen
`>&`	Umleitung von Standardausgabe und -fehler (entspricht `&>`)
`2>`	Umleitung der Standardfehlerausgabe
`<`	Eingabeumleitung aus einer Datei (`kom < dat`)
`<< ende`	Eingabeumleitung aus der aktiven Datei bis zu `ende`
`$`	Kennzeichnung von Variablen (`echo $var`)
`$!`	PID des zuletzt gestarteten Hintergrundprozesses
`$$`	PID der aktuellen Shell
`$0`	Dateiname des gerade ausgeführten Shell-Scripts
`$1` bis `$9`	die ersten neun dem Kommando übergebenen Parameter
`$#`	Anzahl der dem Shell-Programm übergebenen Parameter
`$*` oder `$@`	Gesamtheit aller übergebenen Parameter
`$?`	Rückgabewert des letzten Kommandos (0 = OK oder Fehlernummer)
`$(...)`	Kommandosubstitution (`echo $(ls)`)
`${...}`	diverse Spezialfunktionen zur Bearbeitung von Zeichenketten
`$[...]`	arithmetische Auswertung (`echo $[2+3]`)
`"..."`	Auswertung der meisten Sonderzeichen verhindern
`'...'`	Auswertung aller Sonderzeichen verhindern
blabla	nur als Trenner
`` `...` ``	Kommandosubstitution (`` echo `ls` ``)
`\zeichen`	hebt die Wirkung des Sonderzeichens auf

4.12 Aufgaben

Die Lösungen zu diesen Aufgaben finden Sie im Anhang C.1.

1. Es geht um Kommandosubstitution. Was beinhaltet nach Abarbeitung der Kommandofolge die Datei `datei3`?

```
echo "blau" > kunde
echo "gruen" > kun.de
echo "gelb" >> kunde
echo "rot" > kunde.r
echo kund* >datei2
cat `cat datei2` > datei3
```

2. Mit Pipes kann man recht komfortable Kommandos bilden. Verwenden Sie die Hintereinanderschaltung von `ps -ef` und `grep`, um die Prozesse eine bestimmten Benutzers (z. B. root) oder eines bestimmten Terminals anzuzeigen.
3. Geben Sie einen Aufruf von `find` an, der im Verzeichnis `/home` alle Dateien sucht, auf die innerhalb der letzten 10 Tage zugegriffen wurde.
4. Welches `find`-Kommando müsste man verwenden, um in allen Dateien auf der Platte mit der Endung „.txt“ nach der Zeichenkette „UNIX“ zu suchen.
5. Schreiben Sie ein Shell-Skript, das in einer Endlosschleife alle 10 Sekunden die Uhrzeit (Stunde und Minute) per `banner`-Kommando auf den Bildschirm ausgibt und vorher den Bildschirm löscht.
6. Schreiben Sie ein Shell-Skript „ggt“, das den größten gemeinsamen Teiler der als Parameter übergebenen beiden Zahlen berechnet. Formulieren Sie die Berechnung des GGT als Shell-Funktion. Der Algorithmus lautet folgendermaßen:

```
ggt(x,y):
solange x ungleich y ist, wiederhole
  falls x > y  dann  x = x - y
  sonst              y = y - x;
```

   Denken Sie daran, dass man zum Rechnen das Kommando `expr` braucht.
7. Eine Datei namens `personal` habe folgendes Aussehen:

```
[Name]        [Vorname]     [Wohnort]     [Geb.-Datum]
=========================================================
Meyer         Peter         Berlin         10.10.1970
Schulze       Axel          Hamburg        12.12.1980
Lehmann       Rita          München        17.04.1971
```

   Sortieren Sie die Daten der Datei (ohne Zeile 1 und 2) nach dem Namen und geben Sie das Ergebnis in eine Datei `personal.sort` aus. Die beiden Überschriftszeilen sollen natürlich wieder am Dateianfang stehen.
8. Schreiben Sie ein Shellscript, das an alle Benutzer mit der Gruppennummer 100 eine E-Mail verschickt. Betreff und die Datei, welche den E-Mailtext enthält, werden als Parameter an das Script übergeben.