SHELL

PROGRAMMIERUNG

v2.0.0
29. April 2005

von

Ronald Schaten

ronald@schatenseite.de
http://www.schatenseite.de/

Die aktuellste Version dieses Dokumentes befindet sich auf http://www.schatenseite.de/.

Dieses Dokument ist entstanden, weil ich für mich selbst eine kompakte Übersicht zu diesem Thema haben wollte. Ich beabsichtige nicht, damit in irgendeiner Form Kommerz zu machen. Ich stelle es zur Verfügung, in der Hoffnung, daß andere Leute daraus vielleicht einen Nutzen ziehen können. **Aber ich übernehme keine Garantie für die Korrektheit der hier dargestellten Dinge.**

Ich danke folgenden Personen, die mir bei der Durchsicht behilflich waren und durch ihre konstruktive Kritik zu Verbesserungen beigetragen haben (in chronologischer Reihenfolge ihres Eingreifens):

- Jürgen Ilse (ilse@asys-h.de)
- Christian Perle (christian.perle@tu-clausthal.de)
- Andreas Metzler (ametzler@downhill.at.eu.org)
- Johannes Kolb (johannes.kolb@web.de)
- Falk Friedrich (falk@gmx.de)
- Kai Thöne (kai.thoene@gmx.de)

Und ich bitte alle Leser, auf eventuelle Fehler zu achten und mich darauf aufmerksam zu machen. Auch abgesehen davon freue ich mich über jede Rückmeldung.

Dieses Dokument entstand unter Verwendung von Linux, vim und LaTeX. Dank an deren Entwickler.

Inhaltsverzeichnis

1 Was ist die Shell?

Die Shell ist ein Programm, mit dessen Hilfe das System die Benutzerbefehle verstehen kann. Aus diesem Grund wird die Shell auch oft als Befehls- oder Kommandointerpreter bezeichnet.

1.1 Sinn und Zweck

In einem klassischen Unix-System (ohne die grafische Oberfläche X) greifen die Benutzer über Terminals auf das System zu. Auf diesen Terminals können nur Textzeichen dargestellt werden. Um dem Benutzer die Arbeit mit dem System effektiv möglich zu machen, gibt es die Shell. Die Shell wird dabei für drei Hauptaufgaben benutzt:

- Interaktive Anwendung (Dialog)
- Anwendungsspezifische Anpassung des Unix-Systemverhaltens (Belegen von Umgebungsvariablen)
- Programmierung (Shell-Skripting). Zu diesem Zweck stehen einige Mechanismen zur Verfügung, die aus Hochsprachen bekannt sind (Variablen, Datenströme, Funktionen usw.).

Ursprünglich handelte es sich dabei um ein relativ einfaches Programm, der Bourne Shell (wird oft auch Standard-Shell genannt). Dies ist praktisch die „Mutter aller Shells“. Aus dieser entwickelten sich im Laufe der Zeit mehrere Varianten, die alle ihre eigenen Vor- und Nachteile mit sich bringen. Da es unter Unix kein Problem darstellt den Kommandointerpreter auszutauschen, stehen auf den meisten Systemen mehrere dieser Shells zur Verfügung. Welche Variante ein Benutzer verwenden möchte ist reine Geschmackssache.

1.2 Die Qual der Wahl

Um die Auswahl einer Shell zu erleichtern, werden hier die wichtigsten Varianten kurz vorgestellt. Sie sind aufgeteilt in Einfach- und Komfort-Shells. Die Komfort-Shells zeichnen

sich durch komfortablere Funktionen zur interaktiven Bedienung aus, während die Einfach-Versionen üblicherweise für die Programmierung benutzt werden.

Einfach-Shells:

- Die Bourne- oder Standard-Shell (`sh`) ist die kompakteste und einfachste Form. Sie bietet schon Mechanismen wie die Umlenkung der Ein- oder Ausgaben, Wildcards zur Abkürzung von Dateinamen, Shell-Variablen und einen Satz interner Befehle zum Schreiben von Shell-Prozeduren. Neuere Versionen beherrschen auch das Job-Controlling.

  Für die Entwicklung von Shell-Skripten sollte man sich auf diese Shell beschränken, da sie auf praktisch allen Systemen zur Verfügung steht. So bleiben die Skripte mit kleinen Einschränkungen[1] portabel.

- Die Korn-Shell (`ksh`), eine Weiterentwicklung der Bourne-Shell, erlaubt das Editieren in der Befehlszeile. Außerdem gibt es hier History-Funktionen[2], eine Ganzzahl-Arithmetik, verbesserte Möglichkeiten zur Mustererkennung, Arrays, Aliase[3] und das Job-Controlling[4]. Außerdem bietet die Korn-Shell im Gegensatz zu fast allen anderen Shells die Möglichkeit, Aliase und Shell-Funktionen an Subshells zu vererben.

  Die Korn-Shell existiert in verschiedenen Implementierungen, sowohl kommerziell (ksh88), kostenlos (ksh93) als auch frei (pdksh).

- Die C-Shell (`csh`) bietet ähnliche Annehmlichkeiten wie die Korn-Shell, lehnt sich aber in der Syntax sehr stark an die Programmiersprache C an. Sie sollte nach Möglichkeit nicht zur Shell-Programmierung benutzt werden, da sie an vielen Stellen nicht so reagiert, wie man es erwarten sollte.

Komfort-Shells:

- Die Bourne-Again-Shell (`bash`) ist voll abwärtskompatibel zur sh, bietet aber von allen Shells die komfortabelsten Funktionen für das interaktive Arbeiten. Da die Bash ein GNU-Produkt ist, ist sie die Standard-Shell auf allen Linux-Systemen. Sie steht aber auch auf den meisten anderen Unixen zur Verfügung. Die Bash unterstützt Auto-Completion[5], History-Funktionen, Aliase, eine Ganzzahl-Arithmetik und indizierte Arrays.

[1] Die verschiedenen Implementierungen weisen kleine Unterschiede, z. B. bei der Behandlung von $@ oder den Parametern von `read` (-r) auf.

[2] History-Funktionen ermöglichen es dem Benutzer, einfach auf zurückliegende Befehle zurückgreifen zu können.

[3] Ein Alias ist eine Abkürzung für einen Befehl. Beispielsweise kann man das Häufig benutzte `ls -la` einfach durch `la` ersetzen.

[4] Unter Job-Controlling versteht man einen Mechanismus, mit dessen Hilfe der Benutzer die Ausführung von Prozessen selektiv stoppen oder fortsetzen kann

[5] Mit Auto-Completion ist das automatische Vervollständigen von Dateinamen gemeint.

- Die TENEX-C-Shell (`tcsh`) verhält sich zur C-Shell wie die Bourne-Again-Shell zur Standard-Shell. Sie ist voll kompatibel, bietet aber Komfort-Funktionen wie Kommandozeilen-Editierung, programmierbare Auto-Completion, Rechtschreibhilfen und eine History.
- Die Z-Shell (`zsh`) ähnelt der Korn-Shell, enthält aber viele Erweiterungen. Die Z-Shell unterstützt Kommandozeilen-Editing, programmierbares Auto-Completion, Shell-Funktionen und eine History. Zudem ist eine Rechtschreibprüfung eingebaut.

Exoten:

Desweiteren gibt es noch eine Reihe weiterer Shells, die aber nur selten eingesetzt werden. Dazu gehören die `ash` (Ein Bourne-Shell-Clone für Rechner mit wenig Speicher.), `rc` (Noch ein Bourne-Shell-Clone, ursprünglich aus AT&T Plan 9. Klein, schnell und mit eine C-ähnlichen Syntax.), `esh` (Klein und schnell, bietet eine Lisp-ähnliche Sprache), `sash` (System Administrator's Shell - eine statisch gelinkte Shell mit integrierten Standard-Kommandos.).

Diese Liste ist bei weitem nicht vollständig.

2 Wo sind Unterschiede zu DOS-Batchdateien?

Unter DOS werden Batchdateien oft dazu benutzt, lange Kommandos abzukürzen um die Tipparbeit zu vermindern, oder um sich das Merken von vielen Parametern zu ersparen. Diese Aufgabe überläßt man unter Unix am besten den Shell-Aliasen oder Shell-Funktionen.

Shell-Skripte können viel mehr als Batchdateien.

Wie der Name schon sagt, sind Batchdateien im Wesentlichen nur ein ‘Stapel’ von Anweisungen, die nacheinander ausgeführt werden. Es stehen zwar auch einige einfache Mechanismen zur Verzweigung zu Verfügung, aber das entspricht bei weitem nicht den Möglichkeiten, die man an der Shell hat.

Interaktive Batchdateien sind unter DOS nicht möglich, in der Shell steht dazu das `read`-Kommando zur Verfügung. Ein Mechanismus wie die Befehls-Substitution[1] fehlt völlig.

Ein weiteres interessantes Merkmal ist die Behandlung von Pipes. Es ist unter DOS zwar möglich, zwei Kommandos durch eine Pipe zu verbinden. Aber da es unter DOS keine Möglichkeit gibt, zwei Kommandos parallel laufen zu lassen, wird das erste Kommando vollständig ausgeführt, seine Ausgabe in eine temporäre Datei geschrieben und danach als Eingabe für das zweite Kommando benutzt. Daß so ein Verhalten unter Umständen schnell zu einer vollen Festplatte führen kann, sieht man bei dem Beispiel in Abschnitt 4.15, in dem eine CD kopiert werden soll.

Shell-Skripte kann man dagegen eher mit einer ‘richtigen’ Programmiersprache vergleichen. Es stehen alle Konstrukte zur Verfügung, die eine Programmiersprache auszeichnen (Funktionen, Schleifen, Fallunterscheidungen, Variablen, etc).

[1] Verarbeitung der Ausgabe von Kommandos mittels Backticks (siehe unter Befehlsformen - 4.14)

3 Wofür Shell-Programmierung?

Natürlich stellt sich die Frage, in welchen Situationen ein Shell-Skript der richtige Weg ist, und wann man vielleicht doch besser zu einer interpretierten oder compilierten Sprache greift.

3.1 Wofür?

Die Shell ist der perfekte Baukasten für das Unix-Paradigma ‘small is beautiful’. Die mitgelieferten Unix-Standardkommandos sind einfach gehalten, erledigen aber auf effiziente Weise die Arbeit für die sie programmiert wurden. Mit der Shell, bzw. dem Shell-Skript, wird aus dem Heinzelmännchen ein starker Riese.

Shell-Skripte werden im Wesentlichen aus zwei Gründen geschrieben: Erstens, weil man so ständig wiederkehrende Kommandos zusammenfassen kann, die man dann mit einem einfachen Aufruf starten kann, und zweitens, weil man so einfache Programme schreiben kann, die relativ intelligent Aufgaben erledigen können.

Der erste Aspekt ist wichtig, wenn man beispielsweise regelmäßig auftretende Aufgaben erledigen möchte, wie z. B. das Backup von Log-Dateien. In dem Fall schreibt man sich ein Skript, das die Dateien archiviert, und sorgt dafür, daß dieses Skript in regelmäßigen Abständen aufgerufen wird (per Cron-Job).

Der zweite Fall tritt ein, wenn man eine mehr oder weniger komplexe Abfolge von Befehlen ausführen möchte, die voneinander abhängen. Ein Skript das zum Beispiel eine Audio-CD kopieren soll, sollte das Brennprogramm nur dann aufrufen, wenn der Einlesevorgang erfolgreich abgeschlossen wurde.

3.2 Wofür nicht?

Ein Shell-Skript besteht aus einer Abfolge von System-Tool-Aufrufen. Das heißt, für jeden Schritt in einem Skript wird ein neuer Prozeß gestartet. Das kostet eine Menge Systemzeit,

die Skripte laufen also vergleichsweise langsam. Für komplexe, zeitkritische oder langwierige Aufgaben sollte man also besser zu Perl, Python oder in Extremfällen zu C / C++ greifen.

Shell-Skripte können als imperativ angesehen werden, für viele Aufgaben ist aber ein objektorientierter Ansatz wesentlich geeigneter. Auch hier ist also der Griff zu einer anderen Sprache angeraten.

Es gibt zwar ein paar Tools[1], mit denen auch Shell-Skripte eine grafische oder textorientierte Benutzeroberfläche (GUI) bekommen können, aber das ist trotzdem nicht das natürliche Terrain der Shell-Programmierung.

[1] Zum Beispiel dialog im Textmodus, oder xmessage unter X.

4 Wie sieht ein Shell-Skript aus?

Wie schon erwähnt, kann ein Shell-Skript beinahe alles, was eine ‘richtige’ Programmiersprache auch kann. Bei der Entwicklung sollte man nur bedenken, daß gerade die Ausführung von externen Kommandos – und das ist eine der Standard-Techniken bei der Shell-Programmierung – nur sehr langsam vonstatten geht. Für Anwendungen bei denen z. B. viele Rechnungen oder Stringbearbeitungen gemacht werden müssen, sollte man also ggf. die Benutzung einer anderen Sprache, beispielsweise Perl, in Erwägung ziehen.

In der Shell stehen viele Mechanismen zur Verfügung, die auch aus anderen Sprachen bekannt sind. Um den Umfang dieses Dokuments nicht zu sprengen, werden an dieser Stelle nur die wichtigsten vorgestellt.

4.1 HowTo

Zunächst soll die Frage geklärt werden, wie man überhaupt ein ausführbares Shell-Skript schreibt. Dabei wird vorausgesetzt, daß dem Benutzer der Umgang mit mindestens einem Texteditor (`vi`, `emacs` etc.) bekannt ist.

Zunächst muß mit Hilfe des Editors eine Textdatei angelegt werden, in die der ‘Quelltext’ geschrieben wird. Dabei muß darauf geachtet werden, daß sich keine CR/LF-Zeilenumbrüche einschleichen, wie dies leicht bei der Benutzung von MS-DOS bzw. Windows-Systemen zur Bearbeitung von Skripten über das Netzwerk passieren kann. Wie der Quelltext aussieht, sollte man anhand der folgenden Abschnitte und der Beispiele im Anhang erkennen können. Beim Schreiben sollte man nicht mit den Kommentaren geizen, da ein Shell-Skript auch schon mal sehr unleserlich werden kann.

Nach dem Abspeichern der Datei unter einem geeigneten Namen[1] muß die sie ausführbar gemacht werden. Das geht mit dem Unix-Kommando `chmod` und wird in Abschnitt 5.2.7 ausführlich beschrieben. An dieser Stelle reicht uns ein Aufruf in der Form `chmod 755 name`, um das Skript für alle Benutzer ausführbar zu machen.

[1] Bitte *nicht* den Namen `test` verwenden. Es existiert ein Unix-Systemkommando mit diesem Namen. Dieses steht fast immer eher im Pfad, d. h. beim Kommando `test` würde nicht das eigene Skript ausgeführt, sondern das Systemkommando. Dies ist einer der häufigsten und zugleich einer der verwirrendsten Anfängerfehler. Mehr zu dem `test`-Kommando unter 4.13.6.

Dann kann das Skript gestartet werden. Da sich aus Sicherheitsgründen auf den meisten Systemen das aktuelle Verzeichnis nicht im Pfad des Benutzers befindet, muß man der Shell noch mitteilen, wo sie zu suchen hat: Mit `./name` wird versucht, im aktuellen Verzeichnis (`./`) ein Programm namens `name` auszuführen.

Auf den meisten Systemen befindet sich im Pfad ein Verweis auf das Verzeichnis `bin` unterhalb des Home-Verzeichnisses eines Benutzers. Das bedeutet daß man Skripte die immer wieder benutzt werden sollen dort ablegen kann, so daß sie auch ohne eine Pfadangabe gefunden werden. Wie der Pfad genau aussieht kann man an der Shell durch Eingabe von `echo $PATH` herausfinden.

4.2 Fehlersuche

Es gibt für Shell-Skripte keine wirklichen Debugger, aber trotzdem verfügt man über einige bewährte Methoden zum Aufspüren von Fehlern:

- Debug-Ausgaben: Das wohl einfachste Mittel um herauszufinden was im Skript vor sich geht sind wohl regelmäßige Debug-Ausgaben. Dazu fügt man einfach an 'strategisch wichtigen' Punkten im Skript `echo`-Zeilen ein, die Auskunft über den Status geben.
- Syntax-Check: Wenn man das Skript in der Form `sh -n ./skriptname` aufruft, wird es nicht wirklich ausgeführt. Lediglich die Syntax der Kommandos wird geprüft. Diese Methode findet natürlich keine logischen Fehler, und selbst wenn dieser Aufruf ohne Probleme durchläuft kann sich zur Laufzeit noch ein anderer Fehler einschleichen.
- `set -x`: Wenn in einem Skript der Aufruf `set -x` abgesetzt wird, gibt die Shell jede Zeile nach der Expandierung aber vor der Ausführung aus. Dadurch ist klar ersichtlich wann welche Kommandos mit welchen Parametern ausgeführt werden. Um den Effekt wieder aufzuheben benutzt man `set +x`. Man kann die Option auch auf das komplette Skript anwenden ohne sie in das Skript einbauen zu müssen. Dazu startet man das Skript nicht einfach durch `./skriptname` sondern durch `sh -x ./skriptname`.
- `set -v`: Dies funktioniert genau wie `set -x`, auch der Aufruf von der Kommandozeile über `sh -v ./skriptname` funktioniert. Diese Option gibt jede Zeile vor der Ausführung aus, allerdings im Gegensatz zu `-x` nicht in der expandierten sondern in der vollen Form.
- `set -e`: Alle gängigen Shell-Kommandos liefern einen Rückgabewert, der Auskunft über Erfolg oder Mißerfolg gibt (siehe Abschnitt 4.3). Normalerweise liegt es beim Programmierer, diese Werte zu interpretieren. Setzt man aber mit dem Schalter

`-e` den sogenannten errexit-Modus, beendet die Shell das Skript sobald ein Kommando sich mit einem Rückgabewert ungleich 0 beendet.

Ausnahmen gibt es lediglich, wenn das betroffene Kommando in ein Konstrukt wie `while`, `until` oder `if` eingebunden ist. Auch wenn der Rückgabewert mittels `&&` oder `||` verarbeitet wird, beendet sich die Shell nicht.

- System-Log: Für das direkte Debuggen ist dieser Weg weniger geeignet, aber gerade in unbeobachtet laufenden Skripten sollte man unerwartete Zustände oder besondere Ereignisse im System-Log festhalten. Dies geschieht mit dem Kommando `logger`, das in Abschnitt 5.2.25 beschrieben wird.
- `script`: Mit dem Kommando `script` kann eine Sitzung an der Shell vollständig protokolliert werden, inclusive aller Ein- und Ausgaben. Das umfaßt sogar Drücke auf die Pfeiltasten oder auf Backspace. So kann auch eine längere Sitzung mit vielen Ein- und Ausgaben nach dem Testlauf in aller Ruhe analysiert werden. Das Kommando wird in Abschnitt 5.2.37 beschrieben.
- `tee`: Wenn Ausgaben eines Kommandos durch den Filter `tee` geschoben werden, können sie in einer Datei mitgeschrieben werden. Auch diese Variante bietet einen streßfreien Blick auf unter Umständen sehr lange und komplexe Ausgaben. Abschnitt 5.2.43 gibt weitere Hinweise zu dem Kommando.
- Variablen ‘tracen’: Das Kommando `trap` (Abschnitt 5.2.46) reagiert auf Signale. Die Shell erzeugt nach jedem Kommando das Signal DEBUG, so daß mit dem folgenden Kommando dafür gesorgt werden kann, daß der Inhalt einer Variablen nach jedem Kommando ausgegeben wird:

```
trap 'echo "Trace> \$var = \"$var\""' DEBUG
```

4.3 Rückgabewerte

Wenn unter Unix ein Prozeß beendet wird, gibt er einen Rückgabewert (auch Exit-Code oder Exit-Status genannt) an seinen aufrufenden Prozeß zurück. So kann der Mutterprozeß kontrollieren, ob die Ausführung des Tochterprozesses ohne Fehler beendet wurde. In einigen Fällen (z. B. `grep`) werden unterschiedliche Exit-Codes für unterschiedliche Ereignisse benutzt.

Dieser Rückgabewert wird bei der interaktiven Benutzung der Shell nur selten benutzt, da Fehlermeldungen direkt vom Benutzer abgelesen werden können. Aber in der Programmierung von Shell-Skripten ist er von unschätzbarem Wert. So kann das Skript automatisch entscheiden, ob bestimmte Aktionen ausgeführt werden sollen, die von anderen Aktionen abhängen. Beispiele dazu sieht man bei der Beschreibung der Kommandos `if` (4.13.7),

`case` (4.13.8), `while` (4.13.10) und `until` (4.13.11), sowie in dem Abschnitt über Befehlsformen (4.14).

In der Bourne-Shell wird der Exit-Code des letzten aufgerufenen Programms in der Variable `$?` abgelegt. Üblicherweise geben Programme den Wert 0 zurück, bei irgendwelchen Problemen einen von 0 verschiedenen Wert. Das wird im folgenden Beispiel deutlich:

```
$ cp datei /tmp
$ echo $?
0

$ cp datie /tmp
cp: datie: Datei oder Verzeichnis nicht gefunden
$ echo $?
1
```

Normalerweise wird man den Exit-Code nicht in dieser Form abfragen. Sinnvoller ist folgendes Beispiel, in dem eine Datei erst gedruckt wird, und dann – falls der Ausdruck erfolgreich war – gelöscht wird:

```
$ lpr datei && rm datei
```

Näheres zur Verknüpfung von Aufrufen steht im Kapitel über Befehlsformen (4.14). Beispiele zur Benutzung von Rückgabewerten in Schleifen finden sich im Anhang unter A.1.

Auch Shell-Skripte können einen Rückgabewert an aufrufende Prozesse zurückgeben. Wie das geht, steht in dem Abschnitt zu `exit` (4.13.14).

4.4 Variablen

In einem Shell-Skript hat man – genau wie bei der interaktiven Nutzung der Shell – Möglichkeiten, über Variablen zu verfügen. Anders als in den meisten modernen Programmiersprachen gibt es aber keine Datentypen wie Ganzzahlen, Fließkommazahlen oder Strings[2]. Alle Variablen werden als String gespeichert, wenn die Variable die Funktion einer Zahl übernehmen soll, dann muß das verarbeitende Programm die Variable entsprechend interpretieren[3].

Man muß bei der Benutzung von Variablen sehr aufpassen, wann die Variable expandiert[4] wird und wann nicht. Grundsätzlich werden Variablen während der Ausführung des Skriptes immer an den Stellen ersetzt, an denen sie stehen. Das passiert in jeder Zeile, unmittelbar

[2] Bei einigen modernen Shells (`csh`, `tcsh`, `ksh`, `bash`, `zsh`...) hat man die Möglichkeit, Variablentypen zu vereinbaren. In der Bourne-Shell nicht.

[3] Für arithmetische Operationen steht das Programm `expr` zur Verfügung (siehe Zählschleifen-Beispiel unter 4.13.10)

[4] Mit *Expansion* ist das Ersetzen des Variablennamens durch den Inhalt gemeint

bevor sie ausgeführt wird. Es ist also auch möglich, in einer Variable einen Shell-Befehl abzulegen. Im Folgenden kann dann der Variablenname an der Stelle des Befehls stehen. Um die Expansion einer Variable zu verhindern, benutzt man das Quoting (siehe unter 4.7).

Wie aus diversen Beispielen hervorgeht, belegt man eine Variable, indem man dem Namen mit dem Gleichheitszeichen einen Wert zuweist. Dabei darf zwischen dem Namen und dem Gleichheitszeichen keine Leerstelle stehen, ansonsten erkennt die Shell den Variablennamen nicht als solchen und versucht, ein gleichnamiges Kommando auszuführen – was meistens durch eine Fehlermeldung quittiert wird.

Wenn man auf den Inhalt einer Variablen zugreifen möchte, leitet man den Variablennamen durch ein `$`-Zeichen ein. Alles was mit einem `$` anfängt wird von der Shell als Variable angesehen und entsprechend behandelt (expandiert).

4.5 Vordefinierte Variablen

Es gibt eine Reihe von vordefinierten Variablen, deren Benutzung ein wesentlicher Bestandteil des Shell-Programmierens ist.

Die wichtigsten eingebauten Shell-Variablen sind:

`$`*n*	Aufrufparameter mit der Nummer *n*, $0 <= n <= 9$. `$0` enthält den Namen des gerade laufenden Skripts.
`$*`	Alle Aufrufparameter. `"$*"` enthält alle Aufrufparameter in einem String.
`$@`	Alle Aufrufparameter. `"$@"` enthält alle Aufrufparameter, wobei jeder für sich ein separater String bleibt.
`$#`	Anzahl der Aufrufparameter
`$?`	Rückgabewert des letzten Kommandos
`$$`	Prozeßnummer der aktiven Shell
`$!`	Prozeßnummer des letzten Hintergrundprozesses
`$ERRNO`	Fehlernummer des letzten fehlgeschlagenen Systemaufrufs
`$IFS`	Feldseparator, wird beispielsweise beim Lesen mittels `read` benutzt
`$PATH`	Pfad, in dem nach ausführbaren Kommandos gesucht wird[5]. Mehrere Einträge werden durch Doppelpunkte getrennt angegeben
`$PWD`	Aktuelles Verzeichnis (wird durch `cd` gesetzt[6])

[5] Mit dem Kommando `type` findet man heraus, welches Executable tatsächlich verwendet wird.
[6] Durch das Kommando `cd` wird das aktuelle Verzeichnis gewechselt, siehe Abschnitt 5.2.5.

`$OLDPWD`	Vorheriges Verzeichnis (wird durch `cd` gesetzt)

Die Variable `$IFS` enthält per Default die Blank-Zeichen, also Newline, Space und Tab. Man kann sie aber auch mit anderen Zeichen überschreiben. Diese werden immer dann als Trennzeichen benutzt, wenn ein String in mehrere Teile zerlegt werden soll, also beispielsweise in `for`-Schleifen oder beim zeilenweisen Einlesen mit `read`. Ein gutes Beispiel gibt es in dem Beispielskript zu `printf` (Abschnitt 5.2.32).

`$ERRNO`, `$PWD` und `$OLDPWD` werden nicht von jeder Shell gesetzt.

4.6 Variablen-Substitution

Unter Variablen-Substitution versteht man verschiedene Methoden um die Inhalte von Variablen zu benutzen. Das umfaßt sowohl die einfache Zuweisung eines Wertes an eine Variable als auch einfache Möglichkeiten zur Fallunterscheidung. In den fortgeschritteneren Shell-Versionen (`bash`, `ksh`) existieren sogar Möglichkeiten, auf Substrings von Variableninhalten zuzugreifen. In der Standard-Shell benutzt man für einfache Aufgaben üblicherweise Tools wie `cut`, `basename` oder `dirname`; komplexe Bearbeitungen erledigt der Stream-Editor `sed`. Einleitende Informationen dazu finden sich im Kapitel über die Mustererkennung (4.9).

Die folgenden Mechanismen stehen in der Standard-Shell bereit, um mit Variablen zu hantieren. Bei allen Angaben ist der Doppelpunkt optional. Wenn er aber angegeben wird, muß die *Variable* einen Wert enthalten.

Variable=*Wert*	Setzt die *Variable* auf den *Wert*. Dabei ist unbedingt darauf zu achten, daß zwischen dem Variablennamen und dem Gleichheitszeichen keine Leerzeichen stehen.
${*Variable*}	Nutzt den Wert von *Variable*. Die Klammern müssen nur angegeben werden, wenn auf die *Variable* eine Zahl, ein Buchstabe oder ein Unterstrich folgen.
${*Variable*:-*Wert*}	Nutzt den Wert von *Variable*. Falls die *Variable* nicht gesetzt oder leer ist, wird *Wert* benutzt.
${*Variable*:=*Wert*}	Nutzt den Wert von *Variable*. Falls die *Variable* nicht gesetzt oder leer ist, wird *Wert* benutzt, und *Variable* erhält den *Wert*.
${*Variable*:?*Wert*}	Nutzt den Wert von *Variable*. Falls die *Variable* nicht gesetzt oder leer ist, wird der *Wert* ausgegeben und die Shell beendet. Wenn kein *Wert* angegeben wurde, wird der Text `parameter null or not set` ausgegeben.

${*Variable*:+*Wert*}	*Wert*, falls die *Variable* gesetzt und nicht leer ist, andernfalls nichts.

Beispiele:

`$ h=hoch r=runter l=`	Weist den drei Variablen Werte zu, wobei *l* einen leeren Wert erhält.
`$ echo ${h}sprung`	Gibt *hochsprung* aus. Die Klammern müssen gesetzt werden, damit *h* als Variablenname erkannt werden kann.
`$ echo ${h-$r}`	Gibt *hoch* aus, da die Variable *h* belegt ist. Ansonsten würde der Wert von *r* ausgegeben.
`$ echo ${tmp-‘date‘}`	Gibt das aktuelle Datum aus, wenn die Variable *tmp* nicht gesetzt ist.
`$ echo ${l:=$r}`	Gibt *runter* aus, da die Variable *l* keinen Wert enthält. Gleichzeitig wird *l* der Wert von *r* zugewiesen.
`$ echo $l`	Gibt *runter* aus, da *l* jetzt den gleichen Inhalt hat wie *r*.

4.7 Quoting

Dies ist ein sehr schwieriges Thema, da hier mehrere ähnlich aussehende Zeichen völlig verschiedene Effekte bewirken. Die Bourne-Shell unterscheidet allein zwischen drei verschiedenen Anführungszeichen. Das Quoten dient dazu, bestimmte Zeichen mit einer Sonderbedeutung vor der Shell zu ‘verstecken’ um zu verhindern, daß diese expandiert (ersetzt) werden.

Die folgenden Zeichen haben eine spezielle Bedeutung innerhalb der Shell:

`;`	Befehls-Trennzeichen
`&`	Hintergrund-Verarbeitung
`( ) { }`	Befehlsfolge
`|`	Pipe
`< > >&`	Umlenkungssymbole

`* ? [ ] ~ + - @ !`	Meta-Zeichen für Dateinamen
‘ ‘ (Backticks oder Single Backquotes[7])	Befehls-Substitution
`$`	Variablen-Substitution
NEWLINE SPACE TAB	Wort-Trennzeichen[8]

Die folgenden Zeichen können zum Quoten verwendet werden:

" "	(Anführungszeichen oder Double Quotes) Alles zwischen diesen Zeichen ist buchstabengetreu zu interpretieren. Ausnahmen sind folgende Zeichen, die ihre spezielle Bedeutung beibehalten: `$ ‘ "`
’ ’	(Ticks oder (Single) Quotes[9]) Alls zwischen diesen Zeichen wird wörtlich genommen, mit Ausnahme eines weiteren ’ oder eines Backslashes (\)
\	(Backslash) Das Zeichen nach einem \wird wörtlich genommen. Anwendung z. B. innerhalb von " ", um ", $ und ‘ zu entwerten. Häufig verwendet zur Angabe von Leerzeichen (space) und Zeilenendezeichen, oder um ein \-Zeichen selbst anzugeben.

Beispiele:

```
$ echo ’Ticks "schützen" Anführungszeichen’
Ticks "schützen" Anführungszeichen

$ echo "Ist dies ein \"Sonderfall\"?"
Ist dies ein "Sonderfall"?

$ echo "Sie haben ‘ls | wc -l‘ Dateien in ‘pwd‘"
Sie haben 43 Dateien in /home/rschaten

$ echo "Der Wert von \$x ist $x"
Der Wert von $x ist 100
```

[7] Man erhält sie üblicherweise durch SHIFT und die Taste neben dem Backspace.

[8] Die Wort-Trennzeichen sind in der vordefinierten Variable `$IFS` abgelegt. Siehe Abschnitt 4.5.

[9] Sie liegen auf der Tastatur über der Raute.

4.8 Meta-Zeichen

Bei der Angabe von Dateinamen können eine Reihe von Meta-Zeichen[10] verwendet werden, um mehrere Dateien gleichzeitig anzusprechen oder um nicht den vollen Dateinamen ausschreiben zu müssen.

Die wichtigsten Meta-Zeichen sind:

*	Eine Folge von keinem, einem oder mehreren Zeichen
?	Ein einzelnes Zeichen
[*abc*]	Übereinstimmung mit einem beliebigen Zeichen in der Klammer
[*a*-*q*]	Übereinstimmung mit einem beliebigen Zeichen aus dem angegebenen Bereich
[!*abc*]	Übereinstimmung mit einem beliebigen Zeichen, das nicht in der Klammer ist[11]
~	Home-Verzeichnis des aktuellen Benutzers
~*name*	Home-Verzeichnis des Benutzers *name*
~+	Aktuelles Verzeichnis
~-	Vorheriges Verzeichnis

~, ~*name*, ~+ und ~- werden nicht von jeder Shell unterstützt.

Beispiele:

```
# Alle Dateien listen, die mit 'neu' anfangen:
$ ls neu*

# 'neuX', 'neu4', aber nicht 'neu10' listen:
$ ls neu?

# Alle Dateien listen, die mit einem Grossbuchstaben zwischen D und R
# anfangen - Natuerlich ist die Shell auch hier Case-Sensitive:
$ ls [D-R]*
```

Hier ist anzumerken, daß Hidden Files (Dateien, deren Name mit einem Punkt beginnt) nicht durch ein einfaches * erfaßt werden, sondern nur durch das Suchmuster .*.

[10] Meta-Zeichen werden auch Wildcards, Joker-Zeichen oder Platzhalter genannt. Meint man die Expansion der Meta-Zeichen zu Dateinamen ist auch von 'Globbing' die Rede.

[11] Bei einigen älteren Versionen der Bash muß an Stelle des Rufzeichens ein ^ geschrieben werden.

4.9 Mustererkennung

Man unterscheidet in der Shell-Programmierung zwischen den Meta-Zeichen, die bei der Bezeichnung von Dateinamen eingesetzt werden und den Meta-Zeichen, die in mehreren Programmen Verwendung finden, um z. B. Suchmuster zu definieren. Diese Muster werden auch reguläre Ausdrücke (regular expression) genannt. Sie bieten wesentlich mehr Möglichkeiten als die relativ einfachen Wildcards für Dateinamen.

In der folgenden Tabelle wird gezeigt, in welchen Unix-Tools welche Zeichen zur Verfügung stehen. Eine ausführlichere Beschreibung der Einträge findet sich auf Seite 19.

	ed	ex	vi	sed	awk	grep	egrep	
`.`	•	•	•	•	•	•	•	Ein beliebiges Zeichen.
`*`	•	•	•	•	•	•	•	Kein, ein oder mehrere Vorkommen des vorhergehenden Ausdrucks.
`^`	•	•	•	•	•	•	•	Zeilenanfang.
`$`	•	•	•	•	•	•	•	Zeilenende.
`\`	•	•	•	•	•	•	•	Hebt die Sonderbedeutung des folgenden Zeichens auf.
`[ ]`	•	•	•	•	•	•	•	Ein Zeichen aus einer Gruppe.
`\( \)`	•	•		•				Speichert das Muster zur späteren Wiederholung.
`\{ \}`	•			•		•		Vorkommensbereich.
`\< \>`	•	•	•					Wortanfang oder -ende.
`+`					•		•	Ein oder mehrere Vorkommen des vorhergehenden Ausdrucks.
`?`					•		•	Kein oder ein Vorkommen des vorhergehenden Ausdrucks.
`|`					•		•	Trennt die für die Übereinstimmung verfügbaren Alternativen.
`( )`					•		•	Gruppiert Ausdrücke für den Test.

Bei einigen Tools (`ex`, `sed` und `ed`) werden zwei Muster angegeben: Ein Suchmuster (links) und ein Ersatzmuster (rechts). Nur die folgenden Zeichen sind in einem Ersatzmuster gültig:

	ex	sed	ed	
\	•	•	•	Sonderbedeutung des nächsten Zeichens aufheben.
n	•	•	•	Verwendet das in \(\) gespeicherte Muster erneut.
&	•	•		Verwendet das vorherige Suchmuster erneut.
~	•			Verwendet das vorherige Ersatzmuster erneut.
\u \U	•			Ändert das (die) Zeichen auf Großschreibung.
\l \L	•			Ändert das (die) Zeichen auf Kleinschreibung.
\E	•			Hebt das vorangegangene \U oder \L auf.
\e	•			Hebt das vorangegangene \u oder \l auf.

Sonderzeichen in Suchmustern:

.	Steht für ein beliebiges *einzelnes* Zeichen, mit Ausnahme des Zeilenendezeichens.
*	Steht für eine beliebige (auch leere) Menge des einzelnen Zeichens vor dem Sternchen. Das vorangehende Zeichen kann auch ein regulärer Ausdruck sein. Beispielsweise steht .* für eine beliebige Anzahl eines beliebigen Zeichens.
^	Übereinstimmung, wenn der folgende Ausdruck am Zeilenanfang steht.
$	Übereinstimmung, wenn der vorhergehende Ausdruck am Zeilenende steht.
\	Schaltet die Sonderbedeutung des nachfolgenden Zeichens ab.
[ ]	Steht für *ein* beliebiges Zeichen aus der eingeklammerten Gruppe. Mit einem Bindestrich kann man einen Bereich aufeinanderfolgender Zeichen auswählen ([a-e]). Ein Zirkumflex (^) wirkt als Umkehrung: [^a-z] erfaßt alle Zeichen, die keine Kleinbuchstaben sind. Ein Bindestrich oder eine schließende eckige Klammer am Listenanfang werden als Teil der Liste angesehen, alle anderen Sonderzeichen verlieren in der Liste ihre Bedeutung.
\(\)	Speichert das Muster zwischen \(und \) in einem speziellen Puffer. In einer Zeile können bis zu neun solcher Puffer belegt werden. In Substitutionen können sie über die Zeichenfolgen \1 bis \9 wieder benutzt werden.
\{ \}	Steht für den Vorkommensbereich des unmittelbar vorhergehenden Zeichens. \{*n*\} bezieht sich auf genau *n* Vorkommen, \{*n*,\} auf mindestens *n* Vorkommen und \{*n*,*m*\} auf eine beliebige Anzahl von Vorkommen zwischen *n* und *m*. Dabei müssen *n* und *m* im Bereich zwischen 0 und 256 liegen.
\< \>	Steht für ein Zeichen am Anfang (\<) oder am Ende (\>) eines Wortes.

`+`	Steht für ein oder mehrere Vorkommen des vorhergehenden regulären Ausdrucks (äquivalent zu {`1,`}).
`?`	Steht für kein oder ein Vorkommen des vorhergehenden Ausdrucks (äquivalent zu {`0,1`}).
`|`	Übereinstimmung, wenn entweder der vorhergehende oder der nachfolgende reguläre Ausdruck übereinstimmen.
`( )`	Steht für die eingeschlossene Gruppe von regulären Ausdrücken.

Sonderzeichen in Ersatzmustern:

`\`	Hebt die spezielle Bedeutung des nächsten Zeichens auf.
`\`*n*	Ruft das *n*te Muster aus dem Puffer ab (siehe oben, unter `\( \)`. Dabei ist *n* eine Zahl zwischen 1 und 9.
`&`	Verwendet das vorherige Suchmuster erneut als Teil eines Ersatzmusters.
`~`	Verwendet das vorherige Ersatzmuster erneut im momentanen Ersatzmuster.
`\u`	Ändert das erste Zeichen des Ersatzmusters auf Großschreibung.
`\U`	Ändert alle Zeichen des Ersatzmusters auf Großschreibung.
`\l`	Ändert das erste Zeichen des Ersatzmusters auf Kleinschreibung.
`\L`	Ändert alle Zeichen des Ersatzmusters auf Kleinschreibung.
`\e`	Hebt das vorangegangene `\u` oder `\l` auf.
`\E`	Hebt das vorangegangene `\U` oder `\L` auf.

Beispiele: Muster

`Haus`	Die Zeichenfolge *Haus*.
`^Haus`	*Haus* am Zeilenanfang.
`Haus$`	*Haus* am Zeilenende.
`^Haus$`	*Haus* als einziges Wort in einer Zeile.
`[Hh]aus`	*Haus* oder *haus*.
`Ha[unl]s`	*Haus*, *Hals* oder *Hans*.
`[^HML]aus`	Weder *Haus*, noch *Maus*, noch *Laus*. Jedoch muß die Zeichenfolge *aus* enthalten sein.

`Ha.s`	Der dritte Buchstabe ist ein beliebiges Zeichen.
`^...$`	Jede Zeile mit genau drei Zeichen.
`^\.`	Jede Zeile, die mit einem Punkt beginnt.
`^\.[a-z][a-z]`	Jede Zeile, die mit einem Punkt und zwei Kleinbuchstaben beginnt.
`^\.[a-z]\{2\}`	Wie oben, jedoch nur in grep und sed zulässig.
`^[^.]`	Jede Zeile, die nicht mit einem Punkt beginnt.
`Fehler*`	*Fehle* (!), *Fehler*, *Fehlers*, etc.
`"Wort"`	Ein Wort in Anführungszeichen.
`"*Wort"*`	Ein Wort mit beliebig vielen (auch keinen) Anführungszeichen.
`[A-Z][A-Z]*`	Ein oder mehrere Großbuchstaben.
`[A-Z]+`	Wie oben, jedoch nur in egrep und awk zulässig.
`[A-Z].*`	Ein Großbuchstabe, gefolgt von keinem oder beliebig vielen Zeichen.
`[A-Z]*`	Kein, ein oder mehrere Großbuchstaben.
`[a-zA-Z]`	Ein Buchstabe.
`[^0-9a-zA-Z]`	Symbole (weder Buchstaben noch Zahlen).
`[0-9a-zA-Z]`	Jedes Alphanumerische Zeichen.

Beispiele: egrep- oder awk-Muster

`[567]`	Eine der Zahlen *5*, *6* oder *7*.
`fuenf|sechs|sieben`	Eines der Worte *fuenf*, *sechs* oder *sieben*.
`80[234]?86`	*8086*, *80286*, *80386* oder *80486*.
`F(ahr|lug)zeug`	*Fahrzeug* oder *Flugzeug*.

Beispiele: ex- oder vi-Muster

`\<The`	Wörter wie *Theater* oder *Thema*.
`ung\>`	Wörter wie *Teilung* oder *Endung*.
`\<Wort\>`	Das Wort *Wort*.

Beispiele: sed- oder grep-Muster

`0\{5,\}`	Fünf oder mehr Nullen in Folge.
`[0-9]-[0-9]\{3\}-[0-9]\{5\}-[0-9X]`	ISBN-Nummern in der Form *n-nnn-nnnnn-n*, das letzte Zeichen kann auch ein X sein.

Beispiele: Suchen und Ersetzen mit `sed` und `ex`. Im Folgenden werden Leerzeichen durch [SPACE] und Tabulatoren durch [TAB] gekennzeichnet. Befehle für ex werden mit einem Doppelpunkt eingeleitet.

`s/.*/( & )/`	Wiederholt die ganze Zeile, fügt aber Klammern hinzu.
`s/.*/mv & &.old/`	Formt eine Wortliste (ein Wort pro Zeile) zu `mv`-Befehlen um.
`/^$/d`	Löscht Leerzeilen.
`:g/^$/d`	Wie oben, im ex-Editor.
`/^[`[SPACE][TAB]`]*$/d`	Löscht Leerzeilen und Zeilen, die nur aus Leerzeichen oder Tabulatoren bestehen.
`/^[`[SPACE][TAB]`]*$/d`	Löscht Leerzeilen und Zeilen, die nur aus Leerzeichen oder Tabulatoren bestehen.
`:g/^[`[SPACE][TAB]`]*$/d`	Wie oben, im ex-Editor.
`/`[SPACE][SPACE]`*/`[SPACE]`/g`	Wandelt ein oder mehrere Leerzeichen in ein Leerzeichen um.
`:%s/`[SPACE][SPACE]`*/`[SPACE]`/g`	Wie oben, im ex-Editor.
`:s/[0-9]/Element &:/`	Wandelt (in der aktuellen Zeile) eine Zahl in ein Label für ein Element um.
`:s`	Wiederholt die Substitution beim ersten Vorkommen.
`:&`	Wie oben.
`:sg`	Wie oben, aber für alle Vorkommen in einer Zeile.
`:&g`	Wie oben.
`:%&g`	Wiederholt die Substitution im ganzen Puffer.

`:.,$s/Wort/\U&/g`	Wandelt von der aktuellen bis zur letzten Zeile das Wort *Wort* in Großschreibung um.
`:%s/.*/\L&/`	Wandelt die gesamte Datei in Kleinschreibung um.
`:s/\<./\u&/g`	Wandelt den ersten Buchstaben jedes Wortes in der aktuellen Zeile in Großschreibung um.
`:%s/ja/nein/g`	Ersetzt das Wort *ja* durch *nein*.
`:%s/Ja/~/g`	Ersetzt global ein anderes Wort (*Ja*) durch *nein* (Wiederverwendung des vorherigen Ersatzmusters).

4.10 Klammer-Expansion

Dieser Mechanismus ist sehr praktisch, aber nur wenigen Programmierern bekannt. Er steht nicht in jeder Shell zur Verfügung.

Über die Klammer-Expansion (Brace Expansion) können automatisch Strings generiert werden. Dabei wird ein Muster angegeben, nach dem neue Strings aufgebaut werden. Dieses Muster besteht aus einem Prefix, der allen erzeugten Strings vorangestellt wird, und einer in geschweifte Klammern eingebundenen und durch Komma getrennten Menge von String-Teilen. Dieses Konstrukt expandiert zu mehreren, durch Leerzeichen getrennten Strings, indem sämtliche möglichen Permutationen generiert werden.

Die durch die Klammern angegebenen Mengen können auch verschachtelt werden. Dabei werden die Klammern von links nach rechts aufgelöst. Die Ergebnisse werden nicht sortiert, sondern in der Reihenfolge ihrer Erstellung zurückgegeben werden.

Die Expansion von Klammern erfolgt vor der Behandlung aller anderen Ersetzungen. Auch eventuell vorhandenen Sonderzeichen bleiben in dem expandierten Text erhalten. So lassen sich auch Variablen durch diese Technik erzeugen.

Beispiele:

```
# Folgende Verzeichnisse erzeugen:
# - /usr/local/src/bash/old
# - /usr/local/src/bash/new
# - /usr/local/src/bash/dist
# - /usr/local/src/bash/bugs
$ mkdir /usr/local/src/bash/{old,new,dist,bugs}
```

```
# Die folgenden Dateien / Verzeichnisse dem Benutzer root zuweisen:
# - /usr/ucb/ex
# - /usr/ucb/edit
# - /usr/lib/ex?.?*
# - /usr/lib/how_ex
# Dabei wird /usr/lib/ex?.?* noch weiter expandiert.
$ chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

4.11 Arithmetik-Expansion

Auch hier werden Klammern expandiert. Allerdings gleich doppelte Klammern. Mit einem Konstrukt in der Form `i=$(($i + 1))` können einfache Berechnungen angestellt werden.

Dabei wird der Ausdruck in den Klammern bewertet als ob er in doppelten Anführungszeichen stehen würde. Das bedeutet zum Einen, daß man auch mit Variablen rechnen kann, zum anderen macht es das Quoten des Sternchens überflüssig.

Für komplexere Berechnungen steht das Tool `bc` (Siehe Abschnitt 5.2.3) zur Verfügung.

4.12 Eltern und Kinder: Prozeßordnung

Jedes laufende Programm auf einem Unixoiden System besteht aus einem oder mehreren Prozessen, die jeweils eine eigene Prozeß-ID (PID) haben. Erzeugt ein Programm mehrere Prozesse, sind die zu einer Prozeßgruppe zusammengefaßt. Jeder laufende Prozeß[12] verfügt über genau eine Parent-Prozeß-ID (PPID). Das ist die ID des Prozesses, der den jeweiligen Prozeß erzeugt hat. Man spricht in diesem Zusammenhang tatsächlich von Eltern- bzw. Kind-Prozessen.

Diese Zusammenhänge lassen sich sehr schön durch die Ausgabe eines Kommandos wie `pstree` oder `ps -efH` darstellen, letzteres zeigt auch gleich die PIDs und die PPIDs an.

Wenn in einer Shell ein Kommando gestartet wird, ist es ein Kind dieser Shell. Wird ein Skript gestartet, öffnet es sich seine eigene Shell (siehe 4.13.2) und führt sich innerhalb dieser aus. Die Shell des Skriptes ist dabei ein Kind der interaktiven Shell, die einzelnen Kommandos des Skriptes sind Kinder der Skript-Shell.

[12] Es gibt eine Ausnahme: der Init-Prozeß, der immer die PID 1 hat, hat keine Eltern. Er stammt direkt vom Kernel ab.

Eine solche Shell-in-Shell-Umgebung wird ‘Subshell’ genannt, dieser Mechanismus – und somit auch der Begriff – tauchen immer wieder auf.

Wichtig in diesem Zusammenhang ist das Verständnis für die Vererbung zwischen den beteiligten Prozessen. Wenn in einer Shell eine Variable definiert und exportiert wird, existiert diese auch für die Kind-Prozesse. Gleiches gilt beispielsweise für einen Verzeichnis-Wechsel. Umgekehrt gilt dies nicht: ein Prozeß kann die Umgebung des Parent-Prozesses nicht verändern. Das geschieht nicht zuletzt aus Sicherheitsgründen so.

Will man die Änderungen eines Skriptes übernehmen – beispielsweise wenn ein Skript die Benutzerumgebung konfigurieren soll (.bashrc, .profile und Konsorten) – muß das explizit angefordert werden. Dazu ruft man es mit einem vorangestellten `source` bzw. in der Kurzform mit einem vorangestellten Punkt auf. Weiteres zu dem Thema findet sich im Abschnitt 4.13.4.

Besonders muß man diesen Umstand im Hinterkopf behalten, wenn mit Pipelines (siehe Abschnitt 4.14) gearbeitet wird. Dabei werden auch Kommandos in Subshells ausgeführt, was dann dazu führt daß Variablen belegt werden die dann nach Ausführung der Pipeline plötzlich wieder leer sind. Die Abschnitte A.2 und B.3.1 widmen sich diesem mitunter recht ärgerlichen Thema.

4.13 Programmablaufkontrolle

Bei der Shell-Programmierung verfügt man über ähnliche Konstrukte wie bei anderen Programmiersprachen, um den Ablauf des Programms zu steuern. Dazu gehören Funktionsaufrufe, Schleifen, Fallunterscheidungen und dergleichen.

4.13.1 Kommentare (#)

Kommentare in der Shell beginnen immer mit dem Nummern-Zeichen (#). Dabei spielt es keine Rolle, ob das Zeichen am Anfang der Zeile steht, oder hinter irgendwelchen Befehlen. Alles von diesem Zeichen bis zum Zeilenende wird nicht beachtet (bis auf eine Ausnahme – siehe unter 4.13.2).

4.13.2 Auswahl der Shell (#!)

In der ersten Zeile eines Shell-Skriptes sollte definiert werden, mit welchem Programm das Skript ausgeführt werden soll. Das System öffnet dann eine Subshell und führt das restliche Skript in dieser aus.

Die Angabe erfolgt über eine Zeile in der Form `#!/bin/sh`, wobei unter `/bin/sh` die entsprechende Shell (in diesem Fall die Bourne-Shell) liegt. Dieser Eintrag wirkt nur dann, wenn er in der ersten Zeile und der ersten Spalte des Skripts steht.

Dieser Mechanismus wurde mit dem Aufkommen modernerer Shells eingeführt um eben durch die Angabe von `#!/bin/sh` die Bourne-Shell für die Ausführung von Shell-Skripten benutzen zu können. Interpretiert wird die Zeile vom Kernel, in der Shell selbst wirkt das führende # als Kommentarzeichen.

4.13.3 Null-Befehl (`:`)

Dieser Befehl tut nichts, außer den Status 0 zurückzugeben. Er wird benutzt, um Endlosschleifen zu schreiben (siehe unter 4.13.10), oder um leere Blöcke in `if`- oder `case`-Konstrukten möglich zu machen.

```
if who | grep $1 > /dev/null; then      # who: Liste der Benutzer
                                        # grep: Suche nach Muster
   :                                    # tut nichts
else
   echo "Benutzer $1 ist nicht angemeldet"
fi
```

4.13.4 Source (`.`)

Ein Shell-Skript kann in keiner Weise Einfluß auf die umgebende Shell nehmen. Das heißt, daß es beispielsweise nicht möglich ist, in einem Skript Variablen zu setzen, die dann in der aufrufenden Shell zur Verfügung stehen. Genauso wenig ist es möglich, daß ein Skript den Pfad ändert, in dem man sich befindet. Der Grund für dieses Verhalten ist die Systemsicherheit. Man will verhindern, daß ein Skript unbemerkt Änderungen an der Benutzerumgebung vornimmt.

Wenn es aber doch gewünscht wird, daß ein Skript die Umgebung des Benutzers ändern kann, dann muß es mit dem Source-Kommando aufgerufen werden. Das wird in der Form `source skriptname` bzw. `. skriptname` angegeben. Er wirkt ähnlich wie ein `#include` in der Programmiersprache C.

Die ‘gesourcte’ Datei wird eingelesen und ausgeführt, als ob ihr Inhalt an der Stelle des Befehls stehen würde. Diese Methode wird zum Beispiel beim Login in den Konfigurationsdateien des Benutzers (z. B. `.profile`, `.bashrc`) oder während des Bootvorgangs in den Init-Skripten benutzt, um immer wieder benötigte Funktionen (Starten eines Dienstes, Statusmeldungen auf dem Bildschirm etc.) in einer zentralen Datei pflegen zu können (siehe Beispiel unter A.3).

4.13.5 Funktionen

Es ist in der Shell auch möglich, ähnlich wie in einer ‘richtigen’ Programmiersprache Funktionen zu deklarieren und zu benutzen. Da die Bourne-Shell (`sh`) nicht über Aliase verfügt, können einfache Funktionen als Ersatz dienen.

Der Rückgabewert einer Funktion ist gleich dem Rückgabewert des letzten in der Funktion aufgerufenen Kommandos, es sei denn man gibt mittels `return` (Siehe 4.13.15) explizit einen anderen Wert zurück.

Beispiel: Die Funktion gibt die Anzahl der Dateien im aktuellen Verzeichnis aus. Aufgerufen wird diese Funktion wie ein Befehl, also einfach durch die Eingabe von `count`.

```
count () {
   ls -1 | wc -l  # ls: Liste aller Dateien im Verzeichnis
                  #     Mit Parameter -1 einspaltig
                  # wc: Word-Count, zählt mit Parameter -l Zeilen
}
```

4.13.6 Bedingungen (`[ ]`)

Ursprünglich konnte die Standard-Shell keine arithmetischen oder logischen Ausdrücke auswerten[13]. Für diese Aufgabe mußte ein externes Programm benutzt werden, heutzutage ist der Befehl in die Shell integriert.

Dieser Befehl heißt `test`. Üblicherweise steht er auf allen Systemen auch noch unter dem Namen `[` zur Verfügung. Diese Variante ist fast absolut gleichwertig zu benutzen (in dieser Form wird allerdings eine abschließende Klammer nach der Bedingung erwartet). Dementsprechend ist es auch zwingend erforderlich, nach der Klammer ein Leerzeichen zu schreiben. Das dient dazu, Bedingungen in `if`-Abfragen u. ä. lesbarer zu machen.

`test` bietet sehr umfangreiche Optionen an. Dazu gehören Dateitests und Vergleiche von Zeichenfolgen oder ganzen Zahlen. Diese Bedingungen können auch durch Verknüpfungen kombiniert werden.

Dateitests:

`-b` *Datei*	Die *Datei* existiert und ist ein blockorientiertes Gerät
`-c` *Datei*	Die *Datei* existiert und ist ein zeichenorientiertes Gerät
`-d` *Datei*	Die *Datei* existiert und ist ein Verzeichnis

[13] `if` und Konsorten prüfen nur den Rückgabewert eines aufgerufenen Programmes – 0 bedeutet ‘true’, alles andere bedeutet ‘false’, siehe auch 4.3.

`-f` *Datei*	Die *Datei* existiert und ist eine reguläre Datei
`-g` *Datei*	Die *Datei* existiert, und das Gruppen-ID-Bit ist gesetzt
`-h` *Datei*	Die *Datei* existiert und ist ein symbolischer Link
`-k` *Datei*	Die *Datei* existiert, und das Sticky-Bit ist gesetzt
`-p` *Datei*	Die *Datei* existiert und ist eine Named Pipe
`-r` *Datei*	Die *Datei* existiert und ist lesbar
`-s` *Datei*	Die *Datei* existiert und ist nicht leer
`-t` *[n]*	Der offene Dateideskriptor *n* gehört zu einem Terminal; Vorgabe für *n* ist 1.
`-u` *Datei*	Die *Datei* existiert und das Setuid-Bit ist gesetzt
`-w` *Datei*	Die *Datei* existiert und ist beschreibbar
`-x` *Datei*	Die *Datei* existiert und ist ausführbar

Bedingungen für Zeichenfolgen:

`-n` *s1*	Die Länge der Zeichenfolge *s1* ist ungleich Null
`-z` *s1*	Die Länge der Zeichenfolge *s1* ist gleich Null
s1 `=` *s2*	Die Zeichenfolgen *s1* und *s2* sind identisch
s1 `!=` *s2*	Die Zeichenfolgen *s1* und *s2* sind *nicht* identisch
Zeichenfolge	Die *Zeichenfolge* ist nicht Null

Ganzzahlvergleiche:

n1 `-eq` *n2*	*n1* ist gleich *n2*
n1 `-ge` *n2*	*n1* ist größer oder gleich *n2*
n1 `-gt` *n2*	*n1* ist größer als *n2*
n1 `-le` *n2*	*n1* ist kleiner oder gleich *n2*
n1 `-lt` *n2*	*n1* ist kleiner *n2*
n1 `-ne` *n2*	*n1* ist ungleich *n2*

Kombinierte Formen:

(*Bedingung*)	Wahr, wenn die *Bedingung* zutrifft (wird für die Gruppierung verwendet). Den Klammern muß ein \ vorangestellt werden.
! *Bedingung*	Wahr, wenn die *Bedingung* nicht zutrifft (NOT).
Bedingung1 -a *Bedingung2*	Wahr, wenn beide Bedingungen zutreffen (AND).
Bedingung1 -o *Bedingung2*	Wahr, wenn eine der beiden Bedingungen zutrifft (OR).

Beispiele:

`while test $# -gt 0`	Solange Argumente vorliegen...
`while [ -n "$1" ]`	Solange das erste Argument nicht leer ist...
`if [ $count -lt 10 ]`	Wenn `$count` kleiner 10...
`if [ -d RCS ]`	Wenn ein Verzeichnis RCS existiert...
`if [ "$Antwort" != "j" ]`	Wenn die Antwort nicht "j" ist...
`if [ ! -r "$1" -o ! -f "$1" ]`	Wenn das erste Argument keine lesbare oder reguläre Datei ist...

4.13.7 if...

Die `if`-Anweisung in der Shell-Programmierung macht das gleiche wie in allen anderen Programmiersprachen, sie testet eine Bedingung auf Wahrheit und macht davon den weiteren Ablauf des Programms abhängig.

Die Syntax der `if`-Anweisung lautet wie folgt:

```
if Bedingung1
then Befehle1
[ elif Bedingung2
  then Befehle2 ]
  ⋮
[ else Befehle3 ]
fi
```

Wenn die *Bedingung1* erfüllt ist, werden die *Befehle1* ausgeführt; andernfalls, wenn die *Bedingung2* erfüllt ist, werden die *Befehle2* ausgeführt. Trifft keine Bedingung zu, sollen die *Befehle3* ausgeführt werden.

Bedingungen werden normalerweise mit dem Befehl `test` (siehe unter 4.13.6) formuliert. Es kann aber auch der Rückgabewert[14] jedes anderen Kommandos ausgewertet werden. Für Bedingungen, die auf jeden Fall zutreffen sollen steht der Null-Befehl (`:`, siehe unter 4.13.3) zur Verfügung.

Beispiele: Man achte auf die Positionierung der Semikola[15].

```
# Fuege eine 0 vor Zahlen kleiner 10 ein:
if [ $counter -lt 10 ]; then
  number=0$counter
else
  number=$counter;
fi

# Loesche ein Verzeichnis, wenn es existiert:
if [ -d $dir ]; then
  rmdir $dir     # rmdir: Verzeichnis loeschen
fi
```

4.13.8 case...

Auch die `case`-Anweisung ist vergleichbar in vielen anderen Sprachen vorhanden. Sie dient, ähnlich wie die `if`-Anweisung zur Fallunterscheidung. Allerdings wird hier nicht nur zwischen zwei Fällen unterschieden (Entweder / Oder), sondern es sind mehrere Fälle möglich. Man kann die `case`-Anweisung auch durch eine geschachtelte `if`-Anweisung völlig umgehen, allerdings ist sie ein elegantes Mittel um den Code lesbar zu halten.

Die Syntax der `case`-Anweisung lautet wie folgt:

```
case Wert in
  Muster1 ) Befehle1 ;;
  Muster2 ) Befehle2 ;;
  ⋮
esac
```

Wenn der *Wert* mit dem *Muster1* übereinstimmt, wird die entsprechende Befehlsfolge (*Befehle1*) ausgeführt, bei Übereinstimmung mit *Muster2* werden die Kommandos der zweiten

[14] Siehe unter 4.3.

[15] Und man verzeihe mir einen eventuell falschen Plural...:-)

Befehlsfolge (*Befehle2*) ausgeführt, usw. Der letzte Befehl in jeder Gruppe muß mit `;;` gekennzeichnet werden. Das bedeutet für die Shell soviel wie ‘springe zum nächsten `esac`’, so daß die anderen Bedingungen nicht mehr überprüft werden.

In den Mustern sind die gleichen Meta-Zeichen erlaubt wie bei der Auswahl von Dateinamen. Das bedeutet, daß man durch ein einfaches `*` den Default-Pfad kennzeichnen kann. Dieser wird dann durchlaufen, wenn kein anderes Muster zutrifft. Wenn in einer Zeile mehrere Muster angegeben werden sollen, müssen sie durch ein Pipezeichen (`|`, logisches ODER) getrennt werden.

Beispiele:

```
# Mit dem ersten Argument in der Befehlszeile wird die entsprechende
# Aktion festgelegt:
case $1 in                        # nimmt das erste Argument
   Ja|Nein) response=1;;
     -[tT]) table=TRUE;;
         *) echo "Unbekannte Option"; exit 1;;
esac

# Lies die Zeilen von der Standard-Eingabe, bis eine Zeile mit einem
# einzelnen Punkt eingegeben wird:
while :; do                       # Null-Befehl (immer wahr)
   echo "Zum Beenden . eingeben ==> \c"
   read line                      # read: Zeile von StdIn einlesen
   case "$line" in
       .) echo "Ausgefuehrt"
          break;;
       *) echo "$line";;
   esac
done
```

4.13.9 for...

Dieses Konstrukt ähnelt nur auf den ersten Blick seinen Pendants aus anderen Programmiersprachen. In anderen Sprachen wird die `for`-Schleife meistens dazu benutzt, eine Zählvariable über einen bestimmten Wertebereich iterieren zu lassen (`for i = 1 to 100...next`). In der Shell dagegen wird die Laufvariable nicht mit aufeinanderfolgenden Zahlen belegt, sondern mit einzelnen Werten aus einer anzugebenden Liste[16].

[16] Wenn man trotzdem eine Laufvariable braucht, muß man dazu die `while`-Schleife ‘mißbrauchen’ (siehe unter 4.13.10).

Die Syntax der `for`-Schleife lautet wie folgt:

```
for x [ in Liste ]
do
  Befehle
done
```

Die *Befehle* werden ausgeführt, wobei der Variablen *x* nacheinander die Werte aus der *Liste* zugewiesen werden. Wie man sieht ist die Angabe der *Liste* optional, wenn sie nicht angegeben wird, nimmt *x* der Reihe nach alle Werte aus `$@` (in dieser vordefinierten Variablen liegen die Aufrufparameter – siehe unter 4.5) an. Wenn die Ausführung eines Schleifendurchlaufs bzw der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (4.13.12) bzw. `break` (4.13.13) benutzt werden.

Beispiele:

```
# Seitenweises Formatieren der Dateien, die auf der Befehlszeile
# angegeben wurden, und speichern des jeweiligen Ergebnisses:
for file do
   pr $file > $file.tmp           # pr: Formatiert Textdateien
done

# Durchsuche Kapitel zur Erstellung einer Wortliste (wie fgrep -f):
for item in ‘cat program_list‘    # cat: Datei ausgeben
do
   echo "Pruefung der Kapitel auf"
   echo "Referenzen zum Programm $item ..."
   grep -c "$item.[co]" chap*     # grep: nach Muster suchen
done

# Ermittle einen Ein-Wort-Titel aus jeder Datei und verwende ihn
# als neuen Dateinamen:
for file do
   name=‘sed -n ’s/NAME: //p’ $file‘
                                  # sed: Skriptsprache zur Textformatierung
   mv $file $name                 # mv: Datei verschieben bzw. umbenennen
done
```

4.13.10 while...

Die `while`-Schleife ist wieder ein Konstrukt, das einem aus vielen anderen Sprachen bekannt ist: die Kopfgesteuerte Schleife.

Die Syntax der `while`-Schleife lautet wie folgt:

```
while Bedingung; do
  Befehle
done
```

Die *Befehle* werden so lange ausgeführt, wie die *Bedingung* erfüllt ist. Dabei wird die *Bedingung* vor der Ausführung der *Befehle* überprüft. Die *Bedingung* wird dabei üblicherweise, genau wie bei der `if`-Anweisung, mit mit dem Befehl `test` (siehe unter 4.13.6) formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (4.13.12) bzw. `break` (4.13.13) benutzt werden.

Beispiel:

```
# Zeilenweise Ausgabe aller Aufrufparameter:
while [ -n "$1" ]; do
   echo $1
   shift                          # mit shift werden die Parameter nach
                                  # Links geshiftet (aus $2 wird $1)
done
```

Eine Standard-Anwendung der `while`-Schleife ist der Ersatz für die Zählschleife. In anderen Sprachen kann man mit der `for`-Schleife eine Laufvariable über einen bestimmten Wertebereich iterieren lassen (`for i = 1 to 100...next`). Da das mit der `for`-Schleife der Shell nicht geht[17], ersetzt man die Funktion durch geschickte Anwendung der `while`-Schleife:

```
# Ausgabe der Zahlen von 1 bis 100:
i=1
while [ $i -le 100 ]; do
   echo $i
   i=‘expr $i + 1‘
done
```

Ein weiterer typischer Anwendungsfall ist das zeilenweise Bearbeiten einer Eingabedatei. Dabei kann es sich entweder um eine einfache Textdatei handeln, oder um die Ausgabe eines anderen Kommandos.

Um die Ausgabe eines anderen Kommandos zu verarbeiten kann `while` als Teil einer Pipeline geschrieben werden:

```
# "hallo" suchen und umstaendlich ausgeben:
grep "hallo" datei.txt | while read zeile; do
   echo "Fundstelle: $zeile"
done
```

[17] Auf einigen Systemen steht für diesen Zweck auch das Kommando `seq` (Siehe Abschnitt 5.2.39) zur Verfügung.

Wenn die Eingabe als Textdatei vorliegt ist es verlockend, diese einfach mittels `cat` auszugeben und per Pipe in die Schleife zu schicken. Allerdings sollte an dieser Stelle eine Umleitung benutzt werden. So vermeidet man den überflüssigen Start des Kommandos `cat`:

```
# Zahlen aus einer Datei lesen und aufsummieren:
summe=0
while read zeile; do
   summe=‘expr $summe + $zeile‘
done < datei.txt
echo "Summe: $summe"
```

4.13.11 until...

Die `until`-Schleife ist das Gegenstück zur `while`-Schleife. Allerdings nicht in dem Sinn, wie sie in den meisten anderen Programmiersprachen verstanden wird. Sie arbeitet in der Shell genau wie die `while`-Schleife, mit dem Unterschied daß die Bedingung negiert wird. Es ist also auch eine kopfgesteuerte Schleife, die allerdings so lange läuft wie die angegebene Bedingung nicht zutrifft.

Die Syntax der `until`-Schleife lautet wie folgt:

```
until Bedingung; do
  Befehle
done
```

Die *Bedingung* wird dabei üblicherweise, genau wie bei der `if`-Anweisung, mit mit dem Befehl `test` (siehe unter 4.13.6) formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (4.13.12) bzw. `break` (4.13.13) benutzt werden.

Beispiel: Hier wird die Bedingung nicht per `test` sondern mit dem Rückgabewert des Programms `grep` formuliert.

```
# Warten, bis sich der Administrator einloggt:
until who | grep "root"; do
                                      # who: Liste der Benutzer
                                      # grep: Suchen nach Muster
   sleep 2                            # sleep: warten
done
echo "Der Meister ist anwesend"
```

4.13.12 continue

Die Syntax der `continue`-Anweisung lautet wie folgt:

```
continue [ n ]
```

Man benutzt `continue` um die restlichen Befehle in einer Schleife zu überspringen und mit dem nächsten Schleifendurchlauf anzufangen. Wenn der Parameter *n* angegeben wird, werden *n* Schleifenebenen übersprungen.

4.13.13 break

Die Syntax der `break`-Anweisung lautet wie folgt:

```
break [ n ]
```

Mit `break` kann man die innerste Ebene (bzw. *n* Schleifenebenen) verlassen ohne den Rest der Schleife auszuführen.

4.13.14 exit

Die Syntax der `exit`-Anweisung lautet wie folgt:

```
exit [ n ]
```

Die `exit`-Anweisung wird benutzt, um ein Skript zu beenden. Wenn der Parameter *n* angegeben wird, wird er von dem Skript als Exit-Code zurückgegeben.

4.13.15 return

Die Syntax der `return`-Anweisung lautet wie folgt:

```
return [ n ]
```

Mittels `return` kann eine Funktion (siehe 4.13.5) einen bestimmten Wert zurückgeben. Anderenfalls wird der Exit-Code des letzten in der Funktion ausgeführten Befehls zurückgegeben.

4.14 Befehlsformen

Es gibt eine Reihe verschiedener Möglichkeiten, Kommandos auszuführen. So kommen Verkettungen, Abhängigkeiten und Gruppierungen zustande:

Befehl `&`	Ausführung von *Befehl* im Hintergrund
Befehl1 `;` *Befehl2*	Befehlsfolge, führt mehrere Befehle nacheinander aus
`(`*Befehl1* `;` *Befehl2*`)`	Subshell, behandelt *Befehl1* und *Befehl2* als Befehlsblock und führt sie in einer Subshell aus
`{` *Befehl1* `;` *Befehl2*`; }`	Befehlsblock; alle Befehle werden in der momentanen Shell ausgeführt, aber die Ausgaben werden behandelt, als ob nur ein Befehl ausgeführt würde
Befehl1 `|` *Befehl2*	Pipe, verwendet die Ausgabe von *Befehl1* als Eingabe für *Befehl2*
Befehl1 `‘`*Befehl2*`‘`	Befehls-Substitution, verwendet die Ausgabe von *Befehl2* als Argumente für *Befehl1*
Befehl1 `$(`*Befehl2*`)`	Befehls-Substitution, andere Schreibweise
Befehl1 `&&` *Befehl2*	AND, führt zuerst *Befehl1* und dann (wenn *Befehl1* erfolgreich war) *Befehl2* aus[18]
Befehl1 `||` *Befehl2*	OR, entweder *Befehl1* ausführen oder *Befehl2* (Wenn *Befehl1* nicht erfolgreich war)

Beispiele:

`$ nroff` *Datei* `&`	Formatiert die *Datei* im Hintergrund
`$ cd; ls`	Sequentieller Ablauf
`$ (date; who; pwd) > logfile`	Führt die Befehle in einer Subshell aus und lenkt alle Ausgaben um
`$ { date; who; pwd; } > logfile`	Lenkt alle Ausgaben um
`$ time { date; who; pwd; }`	Summiert die Laufzeit der drei Kommandos
`$ sort` *Datei* `| lp`	Sortiert die *Datei* und druckt sie

[18] Einen ähnlichen Mechanismus kennt man aus verschiedenen Programmiersprachen unter dem Begriff *Kurzschlußauswertung* bzw. *lazy evaluation*.

`$ vi ‘grep -l ifdef *.c‘`	Editiert die mittels grep gefundenen Dateien
`$ grep` *XX Datei* `&& lp` *Datei*	Druckt die *Datei*, wenn sie *XX* enthält
`$ grep` *XX Datei* `|| lp` *Datei*	Druckt die *Datei*, wenn sie *XX* nicht enthält

4.15 Datenströme

Eines der markantesten Konzepte, das in Shell-Skripten benutzt wird, ist das der Datenströme. Die meisten der vielen Unix-Tools bieten die Möglichkeit, Eingaben aus der sogenannten Standard-Eingabe entgegenzunehmen und Ausgaben dementsprechend auf der Standard-Ausgabe zu machen. Es gibt noch einen dritten Kanal für Fehlermeldungen, so daß man eine einfache Möglichkeit hat, fehlerhafte Programmdurchläufe zu behandeln indem man die Fehlermeldungen von den restlichen Ausgaben trennt.

Es folgt eine Aufstellung der drei Standardkanäle:

Dateideskriptor	Name	Gebräuchliche Abkürzung	Typischer Standard
0	Standard-Eingabe	stdin	Tastatur
1	Standard-Ausgabe	stdout	Terminal
2	Fehlerausgabe	stderr	Terminal

Die standardmäßige Eingabequelle oder das Ausgabeziel können wie folgt geändert werden:

Einfache Umlenkung:

Befehl `>` *Datei*	Standard-Ausgabe von *Befehl* in *Datei* schreiben. Die *Datei* wird überschrieben, wenn sie schon bestand.
Befehl `>>` *Datei*	Standard-Ausgabe von *Befehl* an *Datei* anhängen. Die *Datei* wird erstellt, wenn sie noch nicht bestand[19].
Befehl `<` *Datei*	Standard-Eingabe für *Befehl* aus *Datei* lesen.

[19] In einigen sehr alten Versionen der `sh` und in der `csh` wird die Datei nicht erstellt. Um hier sicherzugehen, sollte die Datei vorher mittels `touch` erstellt werden.

Befehl << *Delimiter*	Ein Here-Dokument: Der *Befehl* erhält den folgenden Abschnitt als Standard-Eingabe. Der Abschnitt endet, sobald der Delimiter am Zeilenanfang gefunden wird. Der Delimiter kann beliebig gewählt werden (siehe Beispiel).
Befehl1 | *Befehl2*	Die Standard-Ausgabe von *Befehl1* wird an die Standard-Eingabe von *Befehl2* übergeben. Mit diesem Mechanismus können Programme als 'Filter' für den Datenstrom eingesetzt werden. Das verwendete Zeichen heißt Pipe.

Die Technik eines Here-Dokuments ist sicherlich auf den ersten Blick etwas verwirrend. Man benutzt Here-Dokumente zum Beispiel in einer Situation, in der ein fest vorgegebener Text benötigt wird. Man stelle sich ein Skript vor, das jeden Tag eine Mail mit festem Inhalt und variablem Anhang verschickt.

Oder eine eingebaute Hilfe-Funktion, die bei falschen Parametern einen Hilfetext ausgibt.

Natürlich könnte man zu diesem Zweck eine eigene Datei einrichten, aber das ist eigentlich nicht notwendig. Man handelt sich nur Ärger ein, wenn man das Skript auf einen anderen Rechner portiert und die Datei vergißt. Abgesehen davon - wo legt man eine solche Datei sinnvoll ab?

Um diesem Ärger zu entgehen, sollte man in einer solchen Situation ein Here-Dokument benutzen.

Umlenkung mit Hilfe von Dateideskriptoren:

Befehl >&*n*	Standard-Ausgabe von *Befehl* an den Dateideskriptor *n* übergeben.
Befehl m>&*n*	Der gleiche Vorgang, nur wird die Ausgabe, die normalerweise an den Dateideskriptor *m* geht, an den Dateideskriptor *n* übergeben.
Befehl >&-	Schließt die Standard-Ausgabe.
Befehl <&*n*	Standard-Eingabe für *Befehl* wird vom Dateideskriptor *n* übernommen.
Befehl m<&*n*	Der gleiche Vorgang, nur wird die Eingabe, die normalerweise vom Dateideskriptor *m* stammt, aus dem Dateideskriptor *n* übernommen.
Befehl <&-	Schließt die Standard-Eingabe.

Mehrfach-Umlenkung:

Befehl 2> *Datei*	Fehlerausgabe von *Befehl* in *Datei* schreiben. Die Standard-Ausgabe bleibt unverändert (z. B. auf dem Terminal).

Befehl `>` *Datei* `2>&1`	Fehlerausgabe und Standard-Ausgabe von *Befehl* werden in die *Datei* geschrieben.
Befehl `>` *D1* `2>` *D2*	Standard-Ausgabe erfolgt in die Datei *D1*; Fehlerausgabe in die Datei *D2*.
Befehl `|` `tee` *Dateien*	Die Ausgaben von *Befehl* erfolgen an der Standard-Ausgabe (in der Regel: Terminal), zusätzlich wird sie vom Kommando `tee` in die *Dateien* geschrieben.

Zwischen den Dateideskriptoren und einem Umlenkungssymbol darf kein Leerzeichen sein; in anderen Fällen sind Leerzeichen erlaubt.

Beispiele:

`$ cat` *Datei1* `>` *Neu*	Schreibt den Inhalt der *Datei1* in die Datei *Neu*
`$ cat` *Datei2 Datei3* `>>` *Neu*	Hängt den Inhalt der *Datei2* und der *Datei3* an die Datei *Neu* an
`$ mail` *name* `<` *Neu*	Das Programm `mail` liest den Inhalt der Datei *Neu*
`$ ls -l | grep` *"txt"* `| sort`	Die Ausgabe des Befehls `ls -l` (Verzeichnisinhalt) wird an das Kommando `grep` weitergegeben, das darin nach *"txt"* sucht. Alle Zeilen die das Muster enthalten werden anschließend an `sort` übergeben und landen dann sortiert auf der Standard-Ausgabe.
`$ grep` *"txt"* `* 2>&1 | less`	In allen Dateien wird nach *"txt"* gesucht. Wenn dabei Fehler auftreten (weil z. B. auch Verzeichnisse im aktuellen Verzeichnis sind), werden die Fehlermeldungen an `less` weitergegeben. Dort werden sie dann seitenweise ausgegeben.

Beispiel eines Here-Dokuments:

```
# Ein Here-Dokument: Nach dem << wird ein sogenannter Delimiter
# angegeben. Alle folgenden Zeilen werden an die Standard-Eingabe von
# cat übergeben. Der Text wird durch ein erneutes Vorkommen des
# Delimiters (einzeln und am Zeilenanfang) beendet.
```

```
cat << EOF
Dieser Text wird zeilenweise ausgegeben,
bis ein einzelnes EOF kommt.
EOF
```

Gerade der Mechanismus mit dem Piping sollte nicht unterschätzt werden. Er dient nicht nur dazu, relativ kleine Texte zwischen Tools hin- und herzureichen. An dem folgenden Beispiel soll die Mächtigkeit dieses kleinen Zeichens gezeigt werden:

Es ist mit den passenden Tools unter Unix möglich, eine ganze Audio-CD mit zwei Befehlen an der Kommandozeile zu duplizieren. Das erste Kommando veranlaßt, daß die TOC (Table Of Contents) der CD in die Datei cd.toc geschrieben wird. Das dauert nur wenige Sekunden. Die Pipe steckt im zweiten Befehl. Hier wird der eigentliche Inhalt der CD mit dem Tool ‘cdparanoia’ ausgelesen. Da kein Dateiname angegeben wird, schreibt cdparanoia die Daten auf seine Standard-Ausgabe. Diese wird von dem Brennprogramm ‘cdrdao’ übernommen und in Verbindung mit der TOC ‘on the fly’ auf die CD geschrieben.

```
cdrdao read-toc --datafile - cd.toc
cdparanoia -q -R 1- - | cdrdao write --buffers 64 cd.toc
```

5 Werkzeugkasten

Durch die gezeigten Steuerungsmöglichkeiten stehen dem Shell-Programmierer Möglichkeiten offen, fast alle gängigen Algorithmen zu implementieren. Es ist tatsächlich in der Shell möglich, Sortier- oder Suchfunktionen zu schreiben. Leider kommt aber an dieser Stelle einer der bedeutendsten Nachteile der Shell zum tragen: Die Geschwindigkeit.

In einem Shell-Skript wird für jedes externe Kommando[1] ein eigener Prozeß gestartet. Das kostet natürlich Zeit und Speicher.

Zeitkritische Anwendungen sind also kein Einsatzgebiet für Shell-Skripte. Die schreibt man besser in Perl, Python, oder noch besser in einer ‘compilierten’ Sprache wie C oder C++.

Es stehen jedoch an der Shell viele sehr nützliche externe Kommandos zur Verfügung, die einem die Entwicklung entsprechender eigener Routinen ersparen. Diese externen Kommandos sind zudem in anderen Sprachen geschrieben worden, so daß sie schneller ablaufen als jedes Shell-Skript. Man kommt als Shell-Programmierer nicht sinnvoll um den Einsatz dieser Programme herum.

In diesem Abschnitt sollen einige dieser Programme mit typischen Einsatzmöglichkeiten vorgestellt werden. Eine vollständige Beschreibung wäre (wenn überhaupt möglich) viel zu lang, um an dieser Stelle untergebracht zu werden. **Dies ist also nur ein grober Überblick, nicht mal annähernd eine vollständige Referenz!** Für ausführlichere Beschreibungen empfiehlt sich das Studium der Man-Pages oder der Kauf eines entsprechenden Buches (Siehe Anhang C, ‘Quellen’). Am besten macht man natürlich beides. ;-)

Eine globale Beschreibung aller gängigen Kommandos würde den Rahmen dieses Textes sprengen. Außerdem wäre es nicht leicht, das zu einer Aufgabe passende Werkzeug zu finden. Die Werkzeuge nach Aufgaben zu sortieren fällt allerdings auch nicht leicht. Die Entwickler der Kommandos versuchen, ihre Tools möglichst universell einsetzbar zu halten, also gibt es keine 1:1-Beziehung zwischen Problem und Lösung.

Um sowohl das Finden eines Werkzeugs zu einem gegebenen Problem als auch das Finden einer Beschreibung zu einem gegebenen Werkzeug zu vereinfachen, und um die oben

[1] Externe Kommandos sind solche, die nicht direkt in der Shell enthalten sind, für die also ein eigenes Binary aufgerufen wird.

beschriebene n:m-Beziehung abzubilden, werden hier also zunächst typische Aufgaben beschrieben. Diese enthalten ‘Links’ zu den in Frage kommenden Werkzeugen. Danach gibt es eine alphabetische Aufzählung der wichtigsten Kommandos.

☞ Viele der hier vorgestellten Kommandos stehen in erweiterten Versionen zur Verfügung. Besonders auf GNU-Systemen – und somit auch auf Linux – gibt es viele sehr nützliche Parameter, die man sich auf ‘standardkonformeren’ Systemen nur wünschen kann. Diese Vorteile sind allerdings mit Vorsicht zu genießen: Wenn sie zum Einsatz kommen sind die entstehenden Skripte nicht mehr plattformunabhängig.

Um Überraschungen zu vermeiden wurde versucht, diese Besonderheiten kenntlich zu machen. Stellen mit einer Markierung wie in diesem Absatz sind also besonders zu betrachten.

5.1 Nägel...

5.1.1 Ein- und Ausgabe

Praktisch jedes Skript verwendet in irgendeiner Form die Ein- oder Ausgabe. Sei es in interaktiver Art auf dem Terminal, oder im Hintergrund auf Dateien.

An dieser Stelle sei darauf hingewiesen, daß es auf unixoiden Systemen nicht nur Dateien im Sinne von ‘ein paar Kilobytes Daten, die irgendwo auf der Festplatte rumliegen’ gibt. Vielmehr findet man hier die Geräte des Rechners als Dateien unter /dev. Der Kernel selbst stellt Schnittstellen in Form von virtuellen Dateien unter /proc (ab Kernel 2.6 auch unter /sys) zur Verfügung. Und schlußendlich können Prozesse sich sogenannte Named Pipes anlegen, in die sie schreiben oder aus denen sie lesen.

Diese Kommandos sind also universell nützlich, nicht nur im Zusammenhang mit Dateien auf der Festplatte.

- `cat` (5.2.4): Dateien einlesen und ausgeben
- `date` (5.2.13): Datum oder Zeit ausgeben
- `echo` (5.2.16): Daten ausgeben
- `grep` (5.2.21): In Dateien suchen
- `head` (5.2.22): Dateianfang ausgeben
- `logger` (5.2.25): Text ins System-Log schreiben
- `printf` (5.2.32): Formatierte Datenausgabe

- `read` (5.2.34): Zeilen einlesen
- `tail` (5.2.42): Dateiende ausgeben

5.1.2 Dateiinhalte bearbeiten

Natürlich bietet die Shell eine Reihe von Befehlen, um die Inhalte von Dateien zu bearbeiten. Diese Auflistung ist in weiten Teilen deckungsgleich mit der Liste der Tools zur Manipulation von Pipes, auch diese Kommandos kommen also in mehreren Situationen zum Einsatz.

- `awk` (5.2.1): In einer Pipe editieren
- `cmp` (5.2.10): Binäre Dateien vergleichen
- `cut` (5.2.12): Teile einer Zeile ausschneiden
- `diff` (5.2.14): Textdateien vergleichen
- `paste` (5.2.29): Dateien zusammenführen
- `sed` (5.2.38): In einer Pipe editieren
- `sort` (5.2.41): Zeilenweises Sortieren
- `tr` (5.2.45): Zeichen ersetzen
- `uniq` (5.2.48): Doppelte Zeilen suchen

5.1.3 Pfade und Dateien

Eine der Hauptaufgaben von Shell-Skripten ist natürlich das Hantieren mit Dateien. In diesem Abschnitt geht es allerdings nicht um den Umgang mit Dateiinhalten, sondern vielmehr werden einige nützliche Tools im Umgang mit Dateien an sich vorgestellt.

Auch hier gilt natürlich der Hinweis aus Abschnitt 5.1.1: Eine Datei kann viel mehr sein als nur ein paar Daten im Filesystem.

- `basename` (5.2.2): Den Namen einer Datei (ohne Pfad) ausgeben
- `cd` (5.2.5): Verzeichnis wechseln
- `cp` (5.2.11): Dateien kopieren
- `chgrp` (5.2.6): Gruppen-ID einer Datei ändern
- `chmod` (5.2.7): Zugriffsrechte einer Datei ändern

- `chown` (5.2.8): Eigentümer einer Datei ändern
- `cmp` (5.2.10): Binäre Dateien vergleichen
- `dirname` (5.2.15): Den Pfad zu einer Datei (ohne den Namen) ausgeben
- `find` (5.2.20): Dateien suchen
- `mkdir` (5.2.27): Verzeichnisse anlegen
- `mv` (5.2.28): Dateien verschieben
- `rm` (5.2.35): Dateien löschen
- `rmdir` (5.2.36): Verzeichnisse löschen
- `touch` (5.2.44): Eine leere Datei anlegen, bzw. das Zugriffsdatum einer Datei ändern
- `type` (5.2.47): Art eines Kommandos feststellen
- `which` (5.2.51): Ausführbare Dateien suchen
- `xargs` (5.2.53): Ausgaben eines Kommandos als Parameter eines anderen Kommandos benutzen

5.1.4 Pipes manipulieren

Das Konzept der Pipes (Röhren) wird bereits in dem Kapitel über Befehlsformen (4.14) vorgestellt. Im wesentlichen besteht es darin, daß Daten von einem Programm an ein anderes weitergeleitet werden. Auf diese Weise entsteht eine sogenannte *Pipeline* aus mehreren Kommandos. Einige Kommandos sind für den Einsatz in einem solchen Konstrukt prädestiniert, obwohl die meisten auch alleine eingesetzt werden können.

Übrigens gibt es einen goldenen Merksatz für die Auswahl einiger dieser Tools:

Benutze nicht `awk`, wenn Du `sed` benutzen kannst. Benutze nicht `sed`, wenn Du `grep` benutzen kannst. Benutze nicht `grep`, wenn Du `cut` benutzen kannst.

Der Grund dafür liegt darin, daß diese Programme bei jedem Einsatz gestartet und ausgeführt werden müssen. Und man sollte sich um der Performance willen den kleinsten geeigneten Hammer nehmen.

- `awk` (5.2.1): In einer Pipe editieren
- `cut` (5.2.12): Teile einer Zeile ausschneiden
- `exec` (5.2.18): Dateideskriptoren umhängen
- `grep` (5.2.21): In einer Pipe suchen

- `sed` (5.2.38): In einer Pipe editieren
- `sort` (5.2.41): Zeilenweises Sortieren
- `tee` (5.2.43): Datenstrom in einer Datei protokollieren
- `tr` (5.2.45): Zeichen ersetzen
- `uniq` (5.2.48): Doppelte Zeilen suchen
- `wc` (5.2.50): Zeilen, Wörter oder Zeichen zählen

5.1.5 Prozeßmanagement

Oft werden Shell-Skripte benutzt um Prozesse zu steuern oder zu überwachen. So werden Systemdienste üblicherweise über die Init-Skripte hoch- oder heruntergefahren. Es ist auch nicht sonderlich schwer, mit einem Skript einen ‘Wachhund’ zu implementieren, der regelmäßig kontrolliert ob ein Prozeß noch läuft und ihn bei bedarf nachstartet.

Für Aufgaben in diesem Bereich stehen unter anderem die folgenden Kommandos zur Verfügung.

- `exec` (5.2.18): Kommandos ausführen
- `kill` (5.2.23): Signal an einen Prozeß schicken
- `killall` (5.2.24): Signal an mehrere Prozesse schicken
- `ps` (5.2.33): Prozeßliste ausgeben
- `pgrep` (5.2.30): Bestimmte Prozesse suchen
- `pkill` (5.2.31): Bestimmte Prozesse töten
- `trap` (5.2.46): Auf Signale reagieren
- `wait` (5.2.49): Auf einen Prozeß warten

5.2 ... und Hämmer

Um es noch einmal zu betonen: **Dies ist keine vollständige Kommandoreferenz!** Es werden nur die wichtigsten Kommandos vorgestellt, und deren Funktion wird in den meisten Fällen auch nur kurz angerissen. Für ausgiebigere Informationen empfehle ich entsprechende Bücher (siehe Anhang C, ‘Quellen’) und vor allem die Man-Pages.

5.2.1 awk

Über die Skriptsprache `awk` wurden schon ganze Bücher geschrieben, eine vollständige Beschreibung würde den Rahmen dieses Dokumentes bei weitem sprengen. Hier werden nur ein paar grundlegende Techniken beschrieben, die häufig im Zusammenhang mit Shell-Skripten auftauchen.

Oben wurde `awk` 'Skriptsprache' genannt. Das ist insofern richtig, als daß es eine mächtige und komplexe Syntax zur Verfügung stellt, um Texte automatisiert zu bearbeiten. Es fällt somit in die gleiche Tool-Kategorie wie `sed` (Abschnitt 5.2.38).

Es unterscheidet sich aber in seinen grundlegenden Prinzipien entscheidend von den meisten anderen Programmiersprachen: `awk` arbeitet 'Datenbasiert'. Das bedeutet, daß zunächst die Daten spezifiziert werden mit denen gearbeitet werden soll, dann folgen die auszuführenden Kommandos. Das Prinzip wird schnell klar, wenn man sich einige der Beispiele weiter unten ansieht.

Aufruf

Auch der Aufruf erfolgt analog zu `sed`: Bei einfachen Aufgaben kann das `awk`-Programm direkt an der Kommandozeile mitgegeben werden, komplexere Programme werden in Dateien gespeichert und von dort gelesen.

Eine weitere Gemeinsamkeit ist die Art der Ein- und Ausgabe. Wenn eine Eingabedatei angegeben wird, wird diese verarbeitet. Ansonsten wird die Standard-Eingabe gelesen. Ausgaben erfolgen immer auf der Standard-Ausgabe.

```
# Aufruf als Filter:
kommando1 | awk '{ print $1; print $2 }' | kommando2

# Aufruf mit einer zu bearbeitenden Datei:
awk '{ print $1; print $2 }' datei.txt

# In einem Skript kann das Kommando auch über mehrere Zeilen gehen:
awk '
{
  print $1;
  print $2;
}' datei.txt

# Alternativ können die Kommandos auch in eine eigene Datei gespeichert
# und über den Parameter -f eingebunden werden:
awk -f script.awk datei.txt
```

Neben dem Parameter `-f` zum Einlesen der Programmdatei gibt es noch den Parameter `-F` mit dem der Feld-Trenner angegeben werden kann. Die folgende Zeile gibt beispielsweise alle Benutzernamen und deren User-IDs aus der Doppelpunktseparierten Datei `/etc/passwd` aus:

```
awk -F: '{ print $1" hat ID "$3 }' /etc/passwd
```

Muster und Prozeduren

Die Skripte für `awk` bestehen aus Blöcken von Mustern und Prozeduren. Ein Block hat den folgenden Aufbau:

muster { *prozedur* }

Dabei sind beide Bestandteile des Blockes Optional: Wird das Muster weggelassen, wird die Prozedur auf alle Textbestandteile angewandt. Und wird keine Prozedur angegeben, wird der betroffene Text einfach ausgegeben.

Das Muster kann dabei auf verschiedene Weise angegeben werden:

- Als regulärer Ausdruck (siehe Abschnitt 4.9), eingebettet in Slashes: */muster/*
- Als relationaler Ausdruck, bei dem bestimmte Kriterien auf die Eingabedaten zutreffen müssen. Mit `$2>$1` werden beispielsweise Zeilen angesprochen, deren zweites Feld einen größeren Wert hat als das erste.
- Mit Operatoren für das Pattern-Matching, ähnlich wie in Perl (`~` oder `!~`)
- `BEGIN` kennzeichnet Prozeduren, die vor der Bearbeitung anderer Blöcke zum Tragen kommen sollen.
- Analog dazu gibt es ein `END`, mit dem abschließende Aktionen gekennzeichnet werden.

Abgesehen von `BEGIN` und `END` können die Muster auch durch logische Operatoren (&&, `||` oder `!`) kombiniert werden. Durch Komma getrennt besteht die Möglichkeit, Wirkungsbereiche zu definieren.

Die Prozeduren können Variablen- oder Array-Zuweisungen, Ausgabeanweisungen, Funktionsaufrufe oder Kontrollstrukturen enthalten.

Variablen

Es gibt in `awk` eine Reihe eingebauter Variablen, die in Mustern oder Prozeduren verwendet werden können:

FILENAME	Der Name der aktuellen Eingabedatei, oder ‘-’ wenn von der Standard-Eingabe gelesen wird
FS	Field Separator, das Trennzeichen nach dem die Felder unterteilt werden
OFS	Feldtrennzeichen für die Ausgabe
RS	Record Separator, das Trennzeichen nach dem Datensätze unterteilt werden
ORS	Datensatztrennzeichen für die Ausgabe
NF	Anzahl der Felder im aktuellen Datensatz
NR	Laufende Nummer des aktuellen Datensatzes
OFMT	Ausgabeformat für die Ausgabe von Zahlen und Strings,
$0	Der komplette aktuelle Datensatz
$*n*	Feld *n* des aktuellen Datensatzes

Eigene Variablen können nach Belieben verwendet werden, siehe dazu das Beispiel mit den TEX-Dateien weiter unten.

Beispiele

Hier ein paar Einfache Beispiele für Blocks aus Mustern und Prozeduren:

```
# Das erste Feld jeder Zeile ausgeben:
{ print $1 }

# Alle Zeilen ausgeben, die ’regexp’ enthalten:
/regexp/

# Das erste Feld jeder Zeile ausgeben, die ’regexp’ enthält:
/regexp/ { print $1 }

# Datensätze mit mehr als zwei Feldern auswählen:
NF > 2

# Das dritte und das zweite Feld jeder Zeile ausgeben, deren erstes Feld
# den String ’WICHTIG’ enthält:
$1 ~ /WICHTIG/ { print $3, $2 }

# Die Vorkommen von ’muster’ zählen, und deren Anzahl ausgeben:
/muster/ { ++x }
END { print x }

```

```
# Alle Zeilen mit weniger als 23 Zeichen ausgeben:
length($0) < 23

# Alle Zeilen ausgeben, die mit 'Name:' anfangen und exakt sieben Felder
# enthalten:
NF == 7 && /^Name:/

# Alle Felder der Eingabedaten zeilenweise in umgekehrter Reihenfolge
# ausgeben:
{
  for (i = NF; i >= 1; i--)
    print $i
}

# Die Größe aller TeX-Dateien addieren, die Summe in kB umrechnen und
# ausgeben, verarbeitet die Ausgabe von 'ls -l':
/.*tex/ { summe += $5 }
END { summe /= 1024; print "Die Größe aller TeX-Files:", summe, "kB" }

# Pipe-Separierte Liste aller gemounteten Partitionen und derer
# Füllstände ausgeben, verarbeitet die Ausgabe von 'df':
BEGIN { OFS="|" }
/^\/dev\// { print $1,$5 }

# Alle Hosts aus der /etc/hosts anpingen, muß mit der /etc/hosts als
# Eingabedatei aufgerufen werden:
/^[^#]/ { system("ping -c 1 "$1) }
```

5.2.2 basename

Dem Tool `basename` wird als Parameter ein Pfad zu einer Datei übergeben. Der in der Angabe enthaltene Pfad wird abgeschnitten, nur der Name der eigentlichen Datei wird zurückgegeben. Siehe auch `dirname` (5.2.15).

5.2.3 bc

Bei `bc` handelt es sich, ähnlich wie bei `expr` um einen Taschenrechner. Allerdings verfügt dieses Kommando um eine vergleichsweise komplexe Syntax, die auch Berechnungen mit hoher Genauigkeit zulassen.

Für einfache Grundrechenaufgaben wie das Inkrementieren von Variablen sollte man entweder die eingebaute Arithmetik-Expansion der Shell (Siehe 4.11) oder das wesentlich ressourcenfreundlichere `expr` (Siehe 5.2.19) benutzen.

5.2.4 cat

Auch `cat` ist ein oft unterbewertetes Tool. Seine Aufgabe besteht zwar lediglich darin, etwas von der Standardeingabe oder aus einer Datei zu lesen, und das dann auf der Standardausgabe wieder auszugeben. Allerdings leistet es an vielen, teilweise sehr unterschiedlich gelagerten Aufgaben wertvolle Dienste.

Durch Umlenklung der Ausgabe können Dateien erzeugt und erweitert werden. So können mehrere Dateien per `cat datei1.txt datei2.txt > datei.txt` verkettet werden.

Außerdem kann man mit einem Aufruf in der Art `cat datei.txt | kommando` Daten an ein Programm übergeben, das nur von der Standardeingabe lesen kann (Filter). Das geht zwar auch durch eine Umleitung (siehe Abschnitt 4.15), wird aber in dieser Form von vielen als lesbarer angesehen. Vorteil der Umleitungs-Methode ist, daß nicht erst ein externes Kommando ausgeführt werden muß.

- ☞ GNU-`cat` verfügt über eine Reihe von Parametern, um die Ausgabe zu formatieren, so können mit `-n` bzw. `-b` die Zeilen numeriert werden, oder mit `-s` mehrere Zeilen zu einer einzigen zusammengefaßt werden.

5.2.5 cd

Mit dem Kommando `cd` wird das aktuelle Verzeichnis gewechselt.

5.2.6 chgrp

Jede Datei gehört einem Benutzer und einer Gruppe. Letzteres läßt sich mit `chgrp` einstellen. Als Parameter wird der Name oder die ID der Gruppe, sowie ein oder mehrere Dateinamen übergeben. Verzeichnisse können rekursiv mit dem Parameter `-R` bearbeitet werden.

Der Eingentümer der Datei wird mit `chown` (Abschnitt 5.2.8) festgelegt.

5.2.7 chmod

In unixoiden Systemen verfügt jede Datei über eine Reihe von Attributen. Damit kann eine Menge gemacht werden, für den vollen Funktionsumfang empfiehlt sich das Studium der Man-Page oder einer umfangreicheren Kommandoreferenz. Hier nur das wichtigste in Kürze:

Die Syntax lautet `chmod [options] mode file...`.

Die einzig wichtige Option ist, analog zu `chgrp` und `chown` der Parameter `-R` für die rekursive Bearbeitung von Verzeichnissen.

In der Syntax steht ‘file’ für einen oder mehrere Dateinamen.

Den Modus einer Datei sieht man, indem man `ls -l` darauf ansetzt, die Ausgabe wird im entsprechenden Abschnitt (5.2.26) beschrieben.

Dort ist von den drei ‘rwx-Blöcken’ die Rede, die die Berechtigungen für User (u), Group (g) und Other (o) angeben. Genau die können mittels `chmod` gesteuert werden. Zusätzlich gibt es hier noch die Angabe All (a), mit denen die Rechte für alle Benutzer verändert werden können.

Hier wird der Modus gesteuert, indem direkt angegeben wird für wen welche Rechte gelten sollen. Mit ‘+’ werden die Rechte erweitert, ‘-’ nimmt Rechte und mit ‘=’ werden die Rechte hart gesetzt.

`chmod u+x datei` macht die Datei für den Besitzer ausführbar. Mit dem Parameter `u=rw,go=r` werden die Rechte auf ‘rw-r–r–’ gesetzt, der Besitzer kann lesen und schreiben, alle anderen nur lesen.

Neben dieser Art der Notation gibt es noch eine – wesentlich gängigere – numerische Schreibweise. Dabei werden die Berechtigungen in Form von Zahlen angegeben. Dabei werden drei Zahlen von eins bis sieben benutzt. Deren Bedeutung ergibt sich, wenn man sich die drei Stellen ‘rwx’ als Binärzahl vorstellt. Das x steht an der niederwertigsten Stelle, erhält also den Wert 1. Das w steht für die 2 und r für 4. In Summe ergeben diese Zahlen die Berechtigung. Also ist ‘rwx’ gleichbedeutend mit 4+2+1=7. ‘rw’ entspricht 4+2=6. Die reine Leseberechtigung ‘r’ bleibt als 4 stehen.

Zur Verdeutlichung ein paar Beispiele, wo es möglich ist in beiden Notationen:

a-x		Allen Benutzern werden die Ausführungsrechte genommen
ug+rw		Dem Besitzer und der Gruppe werden Lese- und Schreibrechte gegeben
u=rw,go-rwx	600	Nur der Besitzer kann die Datei lesen und schreiben
u=rw,go=r	644	Der Besitzer darf lesen und schreiben, alle anderen nur lesen
u=rwx,go=rx	755	Der Besitzer darf lesen, schreiben und ausführen, alle anderen nur lesen und ausführen

Am wichtigsten sind also die Aufrufe `chmod 644 datei` und `chmod 755 datei`, je nachdem ob die Datei ausführbar sein soll oder nicht.

5.2.8 chown

Mit `chown` lassen sich Benutzer- und Gruppen-ID von Dateien und Verzeichnissen festlegen. Mit dem Parameter `-R` sogar rekursiv für Verzeichnisse.

Ein einzelner Parameter gibt die User-ID oder den Namen des zukünfigen Benutzers an, in der Form name:gruppe können sowohl User- als auch Gruppen-ID gleichzeitig geändert werden.

Will man lediglich die Gruppen-ID ändern, benutzt man das Kommando `chgrp` (Abschnitt 5.2.6).

5.2.9 chpasswd

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Mit diesem Kommando bietet sich dem Administrator des Systems die Möglichkeit, scriptgesteuert die Paßwörter für neue Benutzer zu vergeben. Manuell ändert man ein Paßwort mit dem Kommando `passwd`, allerdings löscht (flusht) dieses Programm die Standard-Eingabe, bevor es das neue Paßwort erwartet. Somit lassen sich Paßwörter mit `passwd` nur interaktiv ändern[2].

Das Kommando wird in der Form

`echo` *name*`:`*pass* `| chpasswd`

aufgerufen. Es ist auch möglich, dem Programm eine Datei mit vielen Name / Paßwort-Kombinationen an die Standard-Eingabe zu übergeben:

`cat` *passwoerter.txt* `| chpasswd`

Allerdings sollte dabei aus Sicherheitsgründen darauf geachtet werden, daß diese Datei nicht allgemein lesbar ist.

5.2.10 cmp

Mit `cmp` werden zwei Dateien verglichen. Wenn die beiden Dateien identisch sind gibt es keine Ausgabe, ansonsten wird die Position des ersten Unterschiedes ausgegeben.

Einer der beiden anzugebenden Dateinamen kann auch durch `-` ersetzt werden, dann wird die Standard-Eingabe mit der anderen angegebenen Datei verglichen.

[2] Es gibt auch einen anderen Weg: Man kann `passwd` auch mittels `expect` fernsteuern. Allerdings ist diese Methode weniger elegant.

Mit dem Parameter `-l` werden alle abweichenden Bytes aufgelistet, jeweils mit der Position (dezimal) und den beiden Bytes (oktal).

Durch `-s` läßt sich die Ausgabe von Unterschieden unterdrücken, der Exit-Status gibt weiterhin das Ergebnis an.

☞ In der GNU-Version gibt es auch Parameter, mit denen Bereiche der Datei vom Vergleich ausgeschlossen werden können (`-i`), oder mit denen nur die ersten n Bytes der Dateien verglichen werden (`-n`).

5.2.11 cp

Mit `cp` werden Dateien kopiert. Die wichtigsten Optionen im Zusammenhang mit Skripten sind `-f` und `-R`. Ersteres erzwingt (force) das Kopieren, falls an der Zielstelle schon Dateien existieren werden sie überschrieben. Letzteres ermöglicht ein rekursives Kopieren. Verzeichnisse – auch leere – können nur mit `-R` kopiert werden.

5.2.12 cut

Wie der Name (fast) schon sagt, kann man mit diesem Kommando Zeilen zerschneiden. Mit den Parametern `-c` (Character) oder `-f` (Field) wird bestimmt, in welcher Einheit die Schnittstellen abgesteckt werden sollen. Falls die Zeilen in Felder zerteilt werden sollen, kann zusätzlich mittels `-d` der Delimiter, also das Trennzeichen bestimmt werden. Wird das nicht explizit getan, wird der Tabulator benutzt.

Dieser Definition folgt die Angabe des zu behaltenden Bereichs. Dafür kann eins der Formate N, N-, N-M oder -M benutzt werden.

s=”Dies ist ein Test”	
`echo "$s" | cut -c 2-6`	gibt ‘ies i’ aus
`echo "$s" | cut -d " " -f 2`	gibt ‘ist’ aus
`echo "$s" | cut -d " " -f 2-3`	gibt ‘ist ein’ aus
`echo "$s" | cut -d " " -f 2-`	gibt ‘ist ein Test’ aus

Praktisch das Gegenstück zu `cut` ist `paste`, damit werden Dateien in Spalten zusammengeführt. Nährers dazu in Abschnitt 5.2.29.

5.2.13 date

Ein einfacher Aufruf von `date` führt zu einem Ergebnis in der Art:

```
Do Jan 27 00:17:51 CET 2005
```

Wie man sieht, wird das Datum auf dem System entsprechend der lokalen Einstellungen ausgegeben. Auf diesem Rechner ist das deutsch, auf den meisten Rechnern wird Englisch voreingestellt sein.

Neben dieser umfassenden aber starren Ausgabe ist es möglich, das Datum den eigenen Wünschen entsprechend zu formatieren. Ein paar Beispiele:

```
$ date +"%d.%m.%y, %H:%M"
27.01.05, 00:17

$ date +"Heute ist %A."
Heute ist Donnerstag.

$ date +"%A ist der %u. Tag der Woche."
Donnerstag ist der 4. Tag der Woche.
```

Die zur Formatierung bereitstehenden Platzhalter stehen in der Man-Page. Eigentlich sollte `%s` zur Ausgabe der Unix-Systemzeit [3] ein naheliegender Parameter sein, leider steht er nur in der GNU-Version von `date` zur Verfügung.

5.2.14 diff

Mit `diff` werden zwei Dateien verglichen, und die Änderungen auf der Standardausgabe aufgelistet, die nötig sind um die erste an die zweite Datei anzupassen.

Mit dem Parameter `-r` können auch ganze Verzeichnisse rekursiv verglichen werden.

Die ausgegebenen Listen können mit dem `patch`-Kommando auf Dateien angewandt werden um sie auf den geänderten Stand zu bringen.

5.2.15 dirname

Analog zu `basename` (5.2.2) gibt `dirname` nur die Pfad-Komponente einer angegebenen Datei zurück.

[3] Sekunden seit dem 01.01.1970, 0:00 Uhr

5.2.16 echo

Dies ist wohl der grundlegendste Befehl, der in einem Skript verwendet werden kann. Er ist die Voraussetzung, um eines der wichtigsten Werkzeuge der Kybernetik auch mittels eines Shell-Skriptes effizient umzusetzen: Hello World. :-)

Die eigentliche Aufgabe dieses Befehls dürfte jedem bekannt sein, der sich bis zu dieser Stelle durchgearbeitet hat. Allerdings wissen viele nicht, daß auch der echo-Befehl über Parameter verfügt. Zumindest zwei davon erweisen sich in der Praxis oft als sehr hilfreich:

`-n`	Unterdrückt den Zeilenumbruch am Ende der Ausgabe
`-e`	Aktiviert die Interpretation von Steuerzeichen. Diese Steuerzeichen werden mit einem Backslash gequoted, der ganze String muß in Ticks eingeschlossen werden. Beispielsweise gibt `echo -e 'hallo\a'` das Wort hallo und einen Piepton aus.

☞ Die Steuerzeichen, die nach einem `-e` angegeben werden können sind nicht in allen Shells gleich. Das sollte berücksichtigt werden, wenn das Skript portabel bleiben soll.

5.2.17 eval

Die Wirkungsweise von `eval` läßt sich wohl am ehesten durch ein kleines Beispiel erklären:

`foo=42`	Der Variablen namens ‘foo’ wird der Wert ‘42’ zugewiesen.
`a=foo`	Der Variablen namens ‘a’ wird der Wert ‘foo’ zugewiesen.
`b='$'$a`	Der Variablen ‘b’ wird ein $ und der Inhalt der Variablen namens ‘a’ zugewiesen.
`echo $b`	Der Inhalt der Variablen ‘b’ wird ausgegeben.
$foo	
`eval b='$'$a`	Zunächst wird die Zeile nach dem `eval` expandiert, Variablen werden durch Inhalte ersetzt. Es entsteht also ein neuer Ausdruck: ‘b=$foo’. Der wird dann von `eval` ausgeführt.
`echo $b`	Der neue Inhalt der Variablen ‘b’ wird ausgegeben.
42	

Bevor eine Zeile in der Shell tatsächlich ausgeführt wird, wird sie von der Shell expandiert, bzw. evaluiert. Der letzte Begriff deutet schon an was damit gemeint ist: Enthaltene Variablennamen werden durch ihre Werte ersetzt.

Das Kommando `eval` führt die Zeile die durch die Expansion entstanden ist noch einmal aus. So ist es möglich, Variablennamen aus den Inhalten anderer Variablen zu bilden.

Eine wichtige Anwendung für dieses Kommando ist der Fall, wenn eigentlich ein Array gebraucht würde. Wenn `arr` der Name des Arrays und `index` der Name der Variablen ist, die den Index des auszugebenden Elementes enthält, dann kann durch die folgende Zeile der Inhalt eines Array-Elementes ausgegeben werden:

```
eval echo \$arr$index
```

5.2.18 exec

Dieses Kommando hat zwei wesentliche Einsatzgebiete:

Wenn das laufende Skript nur benutzt wird um ein anderes Programm zu starten, beispielsweise um die Umgebungsvariablen geeignet zu belegen, sollte das Programm mit `exec` ausgeführt werden. Der Effekt ist, daß sich die Shell in der das Skript ausgeführt wird beendet und die Kontrolle vollständig an das neue Programm übergibt. So vermeidet man einen überflüssigen Prozeß, der lediglich auf die Beendigung seiner Kinder wartet.

Der zweite Anwendungsfall ist etwas für den fortgeschrittenen Benutzer: Man kann mit `exec` Dateideskriptoren umhängen. Damit ist gemeint, daß zum Beispiel die Standardausgabe mittels `exec >5` auf den Dateideskriptor mit der Nummer 5 gelegt werden kann. Auf diesem Weg kann mit mehreren Datenströmen jongliert werden. Die Beispiele in Anhang B.3 verdeutlichen die Anwendung.

5.2.19 expr

Mit dem Kommando `expr` verfügt die Shell praktisch über einen Taschenrechner für einfache Berechnungen. Für komplexe Aufgaben bietet sich das Tool `bc` an, näheres dazu steht in Abschnitt 5.2.3.

Genau genommen kann man mit `expr` nicht nur Berechnungen durchführen, sondern ganz allgemein ‘Ausdrücke evaluieren’. Damit ist gemeint, daß es zum Beispiel auch Operatoren für Pattern-Matching gibt. Die wichtigsten Operatoren lauten wie folgt:

`|`	logisches ‘oder’
`&`	logisches ‘und’
`<`	kleiner als
`<=`	kleiner als oder gleich
`=`	gleich

`!=`	ungleich
`>=`	größer als oder gleich
`>`	größer als
`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`%`	Modulo (‘Divisionsrest’)
`:`	Pattern-Match mit regulären Ausdrücken
`match`	Analog zu ‘:’
`substr`	Teil einer Zeichenkette zurückgeben
`index`	Position einer Zeichenkette in einer anderen finden
`length`	Länge einer Zeichenkette

Bei einigen Sonderzeichen ist deren Bedeutung in der Shell zu berücksichtigen, sie sind also durch Anführungszeichen oder Backslashes zu quoten: `i=‘expr $i \* 3‘`.

Eine andere Möglichkeit für einfache Rechnungen besteht in der sogenannten Arithmetik-Expansion (Siehe 4.11).

5.2.20 find

Auf einem modernen System sind nicht selten mehrere zehn- oder hunderttausend Dateien vorhanden. Um eine bestimmte Datei anhand komplexer Kriterien ausfindig zu machen benutzt man `find`.

Bei einem Aufruf wird zuerst das zu durchsuchende Verzeichnis, dann die Suchkriterien und eventuell abschließend die durchzuführenden Aktionen angegeben.

Die Angabe der Suchkriterien ist sehr vielseitig, hier werden nur die wichtigsten Optionen beschrieben. Wie immer empfehle ich das Studium der Man-Page oder eines entsprechenden Buches.

Kriterien:	
`-name` *pattern*	Suchmuster für den Dateinamen, auch mit Meta-Zeichen (Abschnitt 4.8)

`-perm [-]`*mode*	Sucht nach Dateien, deren Berechtigungen dem angegebenen Modus entsprechen. Die Notation erfolgt dabei analog zu `chmod` (Abschnitt 5.2.7), wahlweise in der Buchstaben- oder in der Zahlenform
`-type` *c*	Sucht nach Dateien des angegebenen Typs. Dabei kann für *c* ‘b’ (block device), ‘c’ (character device), ‘d’ (directory), ‘l’ (symbolic link), ‘p’ (pipe), ‘f’ (regular file), oder ‘s’ (socket) eingesetzt werden
`-user` *name*	Sucht nach Dateien, die dem angegebenen Benutzer gehören. Die Angabe kann als Name oder als UID erfolgen
`-group` *name*	Sucht nach Dateien, die der angegebenen Gruppe gehören. Die Angabe kann als Name oder als GID erfolgen
`-size` *n*`[c]`	Sucht nach Dateien, deren Größe *n* Blocks[4] oder bei nachgestelltem `c` *n* Bytes ist
`-atime` *n*	Sucht nach Dateien, deren letzter Zugriff (access time) `n`*24 Stunden zurück liegt
`-ctime` *n*	Sucht nach Dateien, deren letzte Statusänderung (change time) `n`*24 Stunden zurück liegt
`-mtime` *n*	Sucht nach Dateien, deren letzte Änderung (modification time) `n`*24 Stunden zurück liegt
`-newer` *file*	Sucht nach Dateien, die jünger sind als *file*
Optionen:	
`-xdev`	Nur auf einem Filesystem suchen
`-depth`	Normalerweise wird eine Breitensuche durchgeführt, dieser Parameter schaltet auf Tiefensuche um
Aktionen:	
`-print`	Dies ist default, die gefundenen Dateinamen werden ausgegeben

[4] Blocks geben die Dateigröße in Bytes dividiert durch 512 und aufgerundet auf den nächsten Integer an. Das hat historische Gründe.

`-exec` *kommando* `{};`	Führt das Kommando für jede Fundstelle aus, an Stelle der geschweiften Klammern wird der gefundene Dateiname eingesetzt. Das abschließende Semikolon muß wegen seiner Sonderbedeutung in der Regel durch einen Backslash gequoted werden

Die verschiedenen Suchkriterien können kombiniert werden. Mit `-a` oder `-o` erreicht man eine logische AND- bzw. OR-Verknüpfung, mit einem vorangestellten `!` können Kriterien negiert werden. Die AND-Verknüpfung muß nicht explizit angegeben werden, wenn mehrere Kriterien verwandt werden. Komplexere Ausdrücke können durch runde Klammern gruppiert werden, dabei ist jedoch deren Sonderbedeutung in der Shell entsprechend zu quoten (Siehe Abschnitt 4.7).

Bei der Angabe numerischer Parameter zu den Suchkriterien wird normalerweise nach dem exakten Wert gesucht. Statt eines einfachen *n* kann jedoch auch *+n* oder *-n* angegeben werden, damit wird dann nach Vorkommen größer bzw. kleiner als *n* gesucht.

Da die reine Beschreibung der Parameter manchmal etwas verwirrend ist, folgen hier ein paar praktische Beispiele:

```
# Suche alle Einträge in bzw. unter dem aktuellen Verzeichnis:
find .

# Suche alle normalen Dateien mit der Endung txt unter /home:
find /home -type f -name \*.txt

# Suche alle Einträge außer symbolischen Links, in die jeder schreiben
# darf:
find / \! -type l -perm 777

# Suche alle Dateien unter dem Homeverzeichnis, deren Größe 10000000
# Bytes übersteigt und gib sie ausführlich aus:
find ~ -size +10000000c -exec ls -l {} \;

# Suche alle Einträge im Homeverzeichnis, die innerhalb der letzten zwei
# Tage geändert wurden:
find ~ -mtime -2
```

Wenn mittels `-exec` weitere Kommandos gestartet werden, sollte beachtet werden daß mindestens ein Prozeß pro Fundstelle gestartet wird. Das kostet sehr viel, unter Umständen macht der Einsatz von `xargs` (Abschnitt 5.2.53) Sinn.

Die Ausführung von `find` erzeugt unter Umständen sehr viel Last auf der Festplatte, bei Netzlaufwerken auch Netzwerkbandbreite. In einigen Fällen bieten sich alternative Suchverfahren an:

Alternative 1: Falls man den Namen der zu suchenden Datei kennt, und das Locate-System installiert ist kann man die Datei auch mittels `locate` suchen. Das ist ressourcenschonender, da nicht 'live' das Filesystem durchforstet wird, sondern nur die Locate-Datenbank. Diese wird allerdings im Regelfall nur einmal täglich aktualisiert, die Suche taugt nicht für schnell wechselnde Bestände.

Alternative 2: Sucht man nach einer ausführbaren Datei, die im Pfad vorhanden ist ('Wo liegt eigentlich Firefox?'), dann sucht man mittels `which` (Abschnitt 5.2.51) oder `type` (5.2.47).

Siehe auch: Abschnitt A.7.2.

5.2.21 grep

Das Tool `grep` stammt aus dem Standard-Repertoire eines jeden Systemadministrators. Mit seiner Hilfe kann in einer oder mehreren Dateien, oder eben auch in einem Datenstrom nach dem Auftreten bestimmter regulärer Ausdrücke (siehe 4.9) gesucht werden.

Die folgende Tabelle stellt einige der vielen Parameter vor:

`-E`	Benutzt erweiterte reguläre Ausdrücke
`-F`	Der Suchbegriff wird nicht als regulärer Ausdruck, sondern als 'fixed String' benutzt
`-c`	Gibt für jede durchsuchte Datei die Anzahl der Fundstellen aus
`-f`	Liest die Suchbegriffe Zeile für Zeile aus einer Textdatei
`-i`	Ignoriert die Unterscheidung zwischen Groß- und Kleinschreibung
`-l`	Gibt nur die Dateinamen aus
`-n`	Gibt zusätzlich die Zeilennummern der Fundstellen aus
`-q`	Gibt die Fundstellen nicht aus, am Exitcode ist zu erkennen ob etwas gefunden wurde
`-v`	Findet alle Zeilen, in denen das Suchmuster *nicht* vorkommt

Siehe auch: Abschnitt A.7.

5.2.22 head

`head` ist das Gegenstück zu `tail` (Siehe 5.2.42). Hier werden allerdings nicht die letzten Zeilen angezeigt, sondern die ersten.

5.2.23 kill

Die landläufige Annahme ist, daß man mit dem `kill`-Kommando Prozesse ‘umbringt’. Das ist zwar wahr, aber nicht die ganze Wahrheit.

Im Prinzip sendet `kill` lediglich ein Signal an einen Prozeß. Ohne weitere Parameter ist das tatsächlich ein SIGTERM, das den Prozeß im Regelfall dazu bewegt sich zu beenden. Jeder Admin kennt das Verfahren, einem hängenden Prozeß mittels `kill -9` den Gnadenschuß zu geben. Die 9 steht dabei für das Signal mit der Nummer 9, SIGKILL. Noch ein gebräuchliches Signal ist SIGHUP (1), der ‘Hangup’. Historisch wurde das gesendet wenn die Leitung zum Rechner aufgelegt wurde, mittlerweile ist es gängige Praxis damit einen Daemon neu zu initialisieren.

Daneben stehen noch eine Reihe weiterer Signale zur Verfügung. Mit `kill -l` kann man sich eine Liste ansehen.

Es gibt verschiedene Wege, das Signal abzusetzen. Welchen man wählt ist Geschmackssache. Hier ein paar Beispiele:

```
# Die folgenden Befehle sind gleichwertig. Alle senden ein HUP an
# Prozeß-ID 42:
kill -1 42
kill -HUP 42
kill -SIGHUP 42
kill -s 1 42
kill -s HUP 42
kill -s SIGHUP 42

# virtueller Selbstmord:
# Alle Prozesse umbringen, die man umbringen kann:
kill -9 -1

# SIGTERM an mehrere Prozesse senden:
kill 123 456 789
```

Siehe auch: Das Beispiel ‘Fallensteller’ in Abschnitt A.5 zeigt, wie ein Skript auf Signale reagieren kann.

5.2.24 killall

Im Abschnitt über `kill` (5.2.23) wird beschrieben, wie man ein Signal an einen Prozeß schickt, dessen ID bekannt ist. Kennt man die ID nicht, oder will man das Signal an mehrere Prozesse schicken, kann dieses Kommando auf vielen Systemen eine große Hilfe darstellen.

Mit dem Parameter `-i` wird vor jedem Signal interaktiv gefragt, ob es geschickt werden soll. Mit `-v` wird angegeben, ob die Signale erfolgreich versandt wurden, `-q` hingegen unterdrückt die Ausgaben.

Da ein Prozeß nach einem Signal nicht notwendigerweise sofort stirbt, gibt es eine Option `-w`. Diese Veranlaßt `killall` zu warten, bis alle Empfänger tot sind. Dieser Parameter ist allerdings mit Vorsicht zu genießen: Wenn der Prozeß sich weigert zu sterben, wartet `killall` ewig.

Eine ähnliche Funktionalität bietet auch das Kommando `pkill` (Abschnitt 5.2.31), allerdings hat `killall` den Vorteil daß es auf mehr Systemen zur Verfügung steht.

5.2.25 logger

Mit `logger` werden Nachrichten an die Log-Mechanismen des Systems geschickt. So können auch unbeobachtet laufende Skripte über ihr tun informieren.

Der zu loggende Text wird einfach als Parameter übergeben.

- ☞ Die GNU-Version verfügt über einige Parameter, unter anderem kann die Nachricht mit `-s` parallel zum System-Log auch auf der Standard-Fehlerausgabe ausgegeben werden.

5.2.26 ls

Den Inhalt von Verzeichnissen im Dateisystem bringt man mit `ls` in Erfahrung. Ein einfacher Aufruf listet lediglich die Dateinamen im aktuellen oder angegebenen Verzeichnis auf, das Kommando hat aber auch sehr viele Parameter mit denen sich die Ausgabe anpassen läßt. Hier sind die wichtigsten, eine vollständige Auflistung bietet wie immer die Man-Page.

- ☞ Einige der folgenden Parameter entsprechen nicht dem allgemeinen Standard:

`-1`	Formatiert die Ausgabe einspaltig
`-a`	Zeigt alle Dateien an, auch solche deren Name mit einem Punkt anfängt
`-A`	☞ Zeigt ‘fast alle’ Dateien an, also auch alle deren Name mit einem Punkt anfängt, allerdings nicht ‘.’ und ‘..’, die in jedem Verzeichnis vorkommen
`-d`	Verzeichnisse werden behandelt wie jede andere Datei auch, ein `ls -ld verzeichnis` gibt also die Eigenschaften des Verzeichnisses aus, nicht dessen Inhalt
`-h`	☞ Gibt bei einer langen Ausgabe mittels `-l` die Größe der Datei ‘human readable’ aus, also nicht zwingend in Bytes
`-l`	Lange Ausgabe, inklusiv der Dateiattribute
`-L`	☞ ‘Dereferenziert’ Links vor der Ausgabe. Es werden nicht die Eigenschaften des Links angezeigt, sondern die der verlinkten Datei
`-r`	Sortierreihenfolge umkehren
`-R`	Rekursiv in Verzeichnisse absteigen und deren Inhalte anzeigen
`-S`	☞ Nach der Größe der Datei sortieren
`-t`	Nach der letzten Änderungszeit sortieren
`-X`	☞ Nach der Extension (also dem Namensteil nach dem letzten Punkt) sortieren

Besonders informativ gibt sich der Parameter `-l`, da damit auch die Eigentümer und die Berechtigungen der Dateien angezeigt werden. Die Ausgabe hat die folgende Form:

```
-rw-r--r-- 1 rschaten users 6252 Nov 19 14:14 shell.tex
```

Die linke Spalte der Ausgabe zeigt die bestehenden Berechtigungen. Es ist ein Block in der Form ‘drwxrwxrwx’. An Stelle des d können auch andere Buchstaben stehen, hier wird der Dateityp angegeben, also ob es sich um eine einfache Datei (-), ein Verzeichnis (d), einen Link (l) oder ähnliches[5] handelt. Die rwx-Blöcke geben die Dateiberechtigungen jeweils für den Besitzer, die Gruppe und andere User an. Dabei steht das r für read, w für write und x für execute. An ihrer Stelle können auch Striche stehen, die repräsentieren nicht gesetzte Attribute. Die Datei im Beispiel ist also für ihren Besitzer les- und schreibbar, für alle anderen nur lesbar. Die Berechtigungen werden mit dem Kommando `chmod` (Abschnitt 5.2.7) gesetzt.

Die nächste Spalte stellt die Anzahl der Links dar, die auf diese Datei verweisen, im Beispiel existiert die Datei an nur einer Stelle im Filesystem.

[5] Siehe Man-Page

Dann folgen der Benutzer und die Gruppe, denen die Datei gehört. Diese Parameter werden mit `chown` (Abschnitt 5.2.8) bzw. `chgrp` (Abschnitt 5.2.6) gesetzt.

Es folgt die Größe der Datei in Bytes, sowie das Datum der letzten Änderung. Liegt dieses mehr als ein halbes Jahr zurück wird an Stelle der Uhrzeit die Jahreszahl angegeben, es gilt also Vorsicht walten zu lassen, wenn dieser Wert in Skripten benutzt werden soll.

Abschließend wird der Name der jeweiligen Datei ausgegeben.

5.2.27 mkdir

Mit diesem Kommando werden Verzeichnisse angelegt. Dabei kann mit `-m` angegeben werden, welche Berechtigungen das Verzeichnis bekommen soll. Mit `-p` werden bei Bedarf auch Parent-Verzeichnisse angelegt, es entsteht also ein kompletter Pfad.

Entfernen lassen sich Verzeichnisse mit `rmdir` (Abschnitt 5.2.36).

5.2.28 mv

Dateien und Verzeichnisse können mit dem Kommando `mv` verschoben werden. Falls am Ziel schon Dateien existieren erzwingt der Parameter `-f` die Aktion, die alten Dateien werden überschrieben. Mit `-i` wird der Vorgang interaktiv, vor jeder Dateibewegung wird nachgefragt.

5.2.29 paste

Während mit `cut` (Abschnitt 5.2.12) Dateien spaltenweise zerlegt werden, werden sie mit `paste` zusammengeführt. Die Dateinamen werden als Parameter übergeben, woraufhin Zeile für Zeile die Inhalte aller Dateien zu einer Tabelle gemacht werden.

Die Spalten werden standardmäßig durch Tabulatorzeichen getrennt, man kann mit dem Parameter `-d` auch ein oder mehrere andere Trennzeichen definieren. Werden mehrere Zeichen angegeben, werden sie der Reihe nach zum trennen der Spalten benutzt.

Mit `-s` wird die Tabelle transponiert, also praktisch um 90 Grad gedreht.

5.2.30 pgrep

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Eine häufig wiederkehrende Aufgabe ist es, zu sehen ob ein bestimmter Prozeß existiert oder nicht. Falls das Kommando `pgrep` zur Verfügung steht, kannn man auf das Konstrukt mit `ps` und `grep` verzichten. Der folgende Aufruf liefert alle Prozeß-IDs, deren Name httpd enthält, inclusive des vollen Kommandos:

```
pgrep -lf httpd
```

Über weitere Parameter läßt sich genauer spezifizieren, wonach gesucht werden soll, hier die wichtigsten:

`-f`	Normalerweise wird nur nach Prozeß-Namen gesucht, mit diesem Parameter wird das volle Kommando (incl. Pfad und Parametern) betrachtet
`-l`	Nicht nur die Prozeß-ID, sondern das volle Kommando wird ausgegeben
`-v`	Findet alle Zeilen, in denen das Suchmuster *nicht* vorkommt
`-x`	Sucht nach Kommandos, deren Name dem Muster *exakt* entspricht

Die Ausgabe enthält per Default nur die Prozeß-IDs der Fundstellen. Diese läßt sich als Parameter für andere Programme benutzen. Das folgende Beispiel liefert detaillierte Informationen über alle xterm-Prozesse:

```
ps -fp $(pgrep -d, -x xterm)
```

Siehe auch: Abschnitt A.7.1.

5.2.31 pkill

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Dieses Kommando ist eng verwandt mit `pgrep` (Siehe Abschnitt 5.2.30), es versteht im Wesentlichen die gleichen Parameter. Allerdings werden die Fundstellen hier nicht ausgegeben. Wie der Name schon andeutet, werden hiermit Prozesse umgebracht. Da man hier mit einem Kommando unter Umständen viele Prozesse beendet, sollten **die verwendeten Parameter genau unter die Lupe** genommen werden, um ‘Kollateralschäden’ zu vermeiden. :-)

Es besteht auch die Möglichkeit, den Prozessen andere Signale zuzuschicken, diese Funktion wird im Abschnitt zu `kill` (5.2.23) näher beschrieben. Das folgende Kommando veranlaßt beispielsweise den Syslog-Daemon, seine Konfiguration neu einzulesen:

```
pkill -HUP syslogd
```

Das Kommando `killall` (Abschnitt 5.2.24) bietet eine ähnliche Funktionalität, allerdings fehlen ihm einige Parameter. Trotzdem sollte im Zweifel `killall` benutzt werden, da es auf mehr Systemen zur Verfügung steht.

5.2.32 printf

Analog zum gleichnamigen Befehl in Programmiersprachen wie Perl oder C dient `printf` der formatierten Ausgabe von Daten. Als Parameter wird ein sogenannter Format-String und eine Liste von auszugebenden Daten mitgegeben. Dabei enthält der Format-String eine Reihe von Platzhaltern, die nach bestimmten Regeln durch die Daten ersetzt werden.

Der Format-String folgt im Wesentlichen den gleichen Regeln wie in der C-Version. Näheres dazu erfährt man mit `man 3 printf`.

Hier die wichtigsten Parameter für den Format-String:

`%b`	Behandelt die Ausgabe von Backslash-Sequenzen. Die Sequenzen sind im Abschnitt über `echo` (5.2.16) beschrieben.
`%s`	Gibt den nächsten Datenstring aus
`%`*n*`$s`	Gibt den *n*-ten Datenstring aus
`%[-]`*m*`[.`*n*`]s`	Gibt den nächsten String aus, dazu wird ein Platz von *m* Zeichen reserviert. Optional können nur die ersten *n* Zeichen ausgegeben oder mit `-` eine Rechtsbündige Formatierung festgelegt werden.

Besonders nützlich ist dieses Kommando bei der tabellarischen Ausgabe von Daten. Im folgenden Beispiel werden alle Benutzernamen, deren Home-Verzeichnisse und Default-Shells aus der Datei `/etc/passwd` extrahiert und übersichtlich ausgegeben:

```
#!/bin/sh
IFS=:
while read user pass uid gid name home shell; do
  printf "%-15s %-25s %s\n" $user $home $shell
done < /etc/passwd
```

Zur Erinnerung: Die vordefinierte Variable `$IFS` ist der Feld-Separator, die Eingabezeilen werden also als Doppelpunkt-separierte Liste gesehen. Näheres dazu steht im Abschnitt über vordefinierte Variablen (4.5).

5.2.33 ps

Mit `ps` gibt man einen Schnappschuß des Zustandes der aktuell laufenden Prozesse aus[6].

Ohne weitere Parameter listet `ps` alle Prozesse auf, die dem aufrufenden Benutzer gehören und die mit dem aktuellen Terminal assoziiert sind. Angezeigt werden dann die Prozeß-ID, das Terminal, die verbrauchte CPU-Zeit und der Name des laufenden Kommandos.

In Skripten möchte man üblicherweise feststellen, ob ein bestimmtes Kommando aktiv ist, ob also zum Beispiel ein bestimmter Serverdienst läuft. Dazu macht man `ps` über Optionen gesprächiger.

☞ Das Kommando versteht in der GNU-Version zwei unterschiedliche Arten von Optionen. Den sogenannten Unix- bzw. Posix-Stil und den BSD-Stil. Zusätzlich gibt es noch ausführliche Parameter, aber die sollen hier nicht beschrieben werden. Die jeweiligen Formen stehen nicht auf allen Systemen zur Verfügung, wenn ein Skript beispielsweise auch unter Solaris benutzt werden soll ist man gezwungen, die Unix-Parametrisierung zu benutzen.

Unix-Parameter zeichnen sich durch die übliche Angabe mit Bindestrich aus. BSD-Parameter werden ohne Bindestrich angegeben, was neben den meisten anderen Kommandos etwas ungewohnt aussieht.

Es gibt sehr viele verschiedene Parameter, die beste Informationsquelle ist wie immer die Man-Page bzw. ein entsprechendes Buch. Hier werden nur ein paar typische Aufrufbeispiele gezeigt, deren Ausgabe sich jeder selber ansehen kann:

```
# Alle Prozesse auflisten, Unix-Syntax:
ps -e
ps -ef
ps -eF
ps -ely

# Alle Prozesse auflisten, BSD-Syntax:
ps ax
ps axu

# Prozeßbaum ausgeben. Das ist in Skripten weniger Sinnvoll, wird hier
# aber angegeben weil es so eine praktische Funktion ist... :-)
ps -ejH
ps axjf

# Alle Prozesse ausgeben, die nicht dem Benutzer ‘root’ gehören:
ps -U root -u root -N

```

[6] Wenn man interaktiv den Zustand der laufenden Prozesse beobachten möchte, benutzt man `top`, das eignet sich jedoch nicht zur Shell-Programmierung und wird deshalb nicht ausführlich beschrieben.

```
# Nur die Prozeß-ID von Syslog ausgeben:
ps -C syslogd -o pid=

# Nur den Namen des Prozesses mit der ID 42 ausgeben:
ps -p 42 -o comm=
```

Für die Suche nach Prozessen bestimmten Namens steht auf manchen Systemen auch das Kommando `pgrep` (Abschnitt 5.2.30) zur Verfügung.

Siehe auch: Abschnitt A.7.1.

5.2.34 read

Mit dem Kommando `read` kann man Eingaben von der Standard-Eingabe lesen. Dabei wird üblicherweise einer oder mehrere Variablennamen übergeben. Dem ersten Namen wird das erste eingegebene Wort zugewiesen, dem zweiten das zweite Wort usw. Dem letzen Variablennamen wird der verbleibende Rest der Eingabezeile zugewiesen. Wenn also nur ein Variablenname angegeben wird, erhält dieser die komplette Eingabezeile. Wenn weniger Worte gelesen werden als Variablen angegeben sind, enthalten die verbleibenden Variablen leere Werte. Als Wort-Trennzeichen dienen alle Zeichen, die in der vordefinierten Variable `$IFS` enthalten sind (siehe Abschnitt 4.5).

Wenn keine Variablennamen angegeben werden, wird die Eingabe in der Variable `REPLY` abgelegt.

Sonderzeichen können während der Eingabe normalerweise mittels eines Backslash vor der Interpretation geschützt werden. Ein Backslash vor einem Newline bewirkt also eine mehrzeilige Eingabe. Dieses Verhalten kann mit dem Parameter `-r` abgeschaltet werden.

☞ Normalerweise wird eine Eingabezeile mit einem Newline abgeschlossen. Mit dem Parameter `-d` ist es möglich, ein anderes Zeilenendezeichen anzugeben. Beispielsweise liest `read -d " " var` alle Zeichen bis zum ersten Leerzeichen in die Variable `var` ein.

Wenn nur eine bestimmte Zahl von Zeichen gelesen werden soll, kann diese durch den Parameter `-n` angegeben werden. Der Befehl `read -n 5 var` liest die ersten fünf Zeichen in die Variable `var` ein. Demzufolge kann ein Skript durch ein `read -n 1` dazu gebracht werden, auf einen einzelnen Tastendruck – nicht zwingend ein Return – zu warten.

Mit dem Parameter `-p` kann man einen Prompt, also eine Eingabeaufforderung ausgeben lassen. `read -p "Gib was ein:" var` schreibt also erst den Text *Gib was ein:*

auf das Terminal, bevor die Eingaben in die Variable `var` übernommen werden. Dieser Prompt wird nur an einem interaktiven Terminal ausgegeben, also nicht in einem Skript das seine Eingaben aus einer Datei oder aus einem Stream erhält.

Wenn die Eingabe von einem Terminal kommt und nicht auf dem Bildschirm erscheinen soll, zum Beispiel bei Paßwortabfragen, kann die Ausgabe mit dem Parameter `-s` (Silent) unterdrückt werden.

Mit `-t` kann ein Time-Out definiert werden, nach dessen Ablauf das Kommando mit einem Fehler abbricht. Dieser Parameter ist nur bei interaktiver Eingabe oder beim Lesen aus einer Pipe aktiv.

Der Rückgabewert des `read`-Kommandos ist 0, es sei denn es trat ein Timeout oder ein EOF auf.

5.2.35 rm

Mit diesem Kommando können Dateien und Verzeichnisse gelöscht werden. Dabei kann man vorsichtig vorgehen, indem man mit `-i` dafür sorgt, daß jeder Löschvorgang bestätigt werden muß. Oder rabiat, indem man mit `-f` das Löschen erzwingt.

Verzeichnisse können mit dem Parameter `-R` entfernt werden, im Gegensatz zu `rmdir` werden dann auch sämtliche enthaltenen Dateien und Unterverzeichnisse gelöscht.

☞ Die GNU-Version von `rm` unterstützt zusätzlich den Parameter `-v`, mit dem jeder Löschvorgang ausgegeben wird.

5.2.36 rmdir

Mit `rmdir` werden Verzeichnisse gelöscht. Das funktioniert nur, wenn sie leer sind. Mit `-p` kann ein kompletter Verzeichnispfad gelöscht werden, will sagen: Alle höher liegenden Verzeichnisse im angegebenen Pfad werden gelöscht. Voraussetzung ist hier natürlich auch, daß die Verzeichnisse nichts außer dem angegebenen Unterverzeichnis enthalten.

Angelegt werden Verzeichnisse mit `mkdir` (Abschnitt 5.2.27), nicht-leere Verzeichnisse können rekursiv mit `rm -r` (Abschnitt 5.2.35) gelöscht werden.

5.2.37 script

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Dieses Kommando eignet sich vorzüglich für das Debuggen fertiger Skripte. Man ruft es in Verbindung mit einem Dateinamen auf. Dieser Aufruf startet eine neue Shell, in der man beliebige Kommandos ausführen kann. Wenn man fertig ist, beendet man den script-Befehl durch die Eingabe von `exit`, `logout` oder Druck der Tastenkombination CTRL+d (EOF).

Script schreibt alle Ein- und Ausgaben die an dem Terminal vorgenommen werden in die angegebene Datei. So kann man auch interaktive Skripte relativ leicht debuggen, da sowohl Ein- als auch Ausgaben in dem Logfile sichtbar sind.

5.2.38 sed

Der ‘Stream Editor’ `sed` stellt, ähnlich wie `awk` (Abschnitt 5.2.1) eigentlich eine eigene Skriptsprache dar. Er wird auch ‘nicht-interaktiver Editor’ genannt. Die Kommandos sind minimalistisch, aber exakt auf die Aufgabe zugeschnitten.

`sed` liest Zeilenweise aus einer Datei, wenn keine Datei angegeben wurde wird von der Standard-Eingabe gelesen. Auf die eingelesenen Zeilen wird dann ein mehr oder weniger kompliziertes `sed`-Skript angewendet, bevor auf der Standard-Ausgabe die Resultate ausgegeben werden.

Eine vollständige Beschreibung von `sed` würde an dieser Stelle den Rahmen sprengen, es gibt aber im Handel gute Bücher zu dem Thema. Hier sollen nur die gängigsten Kommandos und einige Anwendungsbeispiele genannt werden.

Aufruf

```
# Aufruf als Stream-Editor:
kommando1 | sed 's/alt/neu/' | kommando2

# Aufruf mit einer zu bearbeitenden Datei:
sed 's/alt/neu/' datei.txt

# Wenn mehr als ein Kommando ausgeführt werden soll, muß der Parameter
# -e verwendet werden:
sed -e 's/alt/neu/' -e '/loeschen/d' datei.txt

# Man kann auch mehrere Kommandos mit einem -e aufrufen, wenn sie durch
# ein Semikolon getrennt werden:
sed 's/alt/neu/; /loeschen/d' datei.txt
```

```
14
# In einem Skript kann das Kommando auch über mehrere Zeilen gehen:
16 sed '
s/alt/neu/
18 /loeschen/d' datei.txt

20 # Alternativ können die Kommandos auch in eine eigene Datei gespeichert
# und über den Parameter -f eingebunden werden:
22 sed -f script.sed datei.txt
```

Neben den oben erwähnten Parametern kann `sed` auch mit `-n` ruhig gestellt werden. Damit werden die Zeilen nur dann ausgegeben, wenn das mittels 'p' explizit gefordert wird.

☞ Die GNU-Version stellt noch ein paar Parameter zur Verfügung, die Man-Page verrät näheres.

Addressierung

Durch die Adressierung können Befehle gezielt auf bestimmte Zeilen angewandt werden. Dabei können einem Befehl keine, eine oder zwei Adressen mitgegeben werden.

Wenn keine Zeilen adressiert werden, wirkt der Befehl auf alle Zeilen.

Wenn eine Adresse mitgegeben wird, wirkt der Befehl auf alle Zeilen die durch diese Adresse angesprochen werden. Das können, zum Beispiel bei einem regulären Ausdruck, auch mehrere Zeilen sein.

Werden zwei Adressen angegeben, wirkt der Befehl auf die erste betroffene Zeile, sowie auf alle weiteren bis zur zweiten angegebenen Zeile. Die beiden Adressen müssen durch ein Komma getrennt angegeben werden.

Die Auswahl der Zeilen kann durch ein an die Adresse angehängtes Rufzeichen negiert werden, der Befehl wirkt dann also auf alle Zeilen die **nicht** adressiert wurden.

Aber wie sehen solche Adreßangeben aus? Die folgende Tabelle zeigt einige Beispiele anhand des Kommandos 'd', mit dem Zeilen gelöscht werden:

`d`	Keine Adresse angegeben, alle Zeilen werden gelöscht
`23d`	Zeile 23 wird gelöscht
`$d`	Die letzte Eingabezeile wird gelöscht
`/`*pattern*`/d`	Jede Zeile, die den regulären Ausdruck *pattern* enthält wird gelöscht
`23,42d`	Die Zeilen 23 bis einschließlich 42 werden gelöscht
`23,$d`	Von Zeile 23 bis zum Ende wird gelöscht

`1,/^$/d`	Von der ersten bis zur ersten Leerzeile wird gelöscht
`23,42!d`	Alles außer den Zeilen 23 bis 42 wird gelöscht

Adressen können auch vor geschweiften Klammern stehen, dann wirken sie auf die komplette Befehlsfolge innerhalb der Klammern.

Kommandos

Es gibt eine ganze Reihe von Kommandos, diese Beschreibung konzentriert sich aber auf die wichtigsten 'Brot und Butter-Kommandos'. In den Beispielen weiter unten kommen auch andere Kommandos vor, die können bei Bedarf anhand der einschlägigen Quellen nachgeschlagen werden.

Editieren:	
`/`*adresse*`/d`	Zeilen löschen
`s/`*regexp1*`/`*regexp2*`/`	Substitution: Suchen und ersetzen
`y/`*menge1*`/`*menge2*`/`	Zeichen ersetzen
Zeileninformation:	
`/`*adresse*`/p`	Zeilen ausgeben

Mit `s` wird substituiert. Das heißt, in der Eingabezeile wird nach einem Muster gesucht, und im Erfolgsfall wird es ersetzt. Wichtigster Modifikator für dieses Kommando ist `g`, damit wird 'global' ersetzt, falls mehrere Fundstellen in einer Zeile vorkommen. Der Aufruf sieht wie folgt aus:

```
s/Suchmuster/Ersatzmuster/g
```

Im Ersatzmuster können auch Teile der Fundstelle wieder vorkommen, wenn sie durch Klammern in einen Puffer kopiert werden:

```
s/Seite ([0-9]*) von ([0-9]*)/\1 aus \2/
```

Mit `y` hingegen werden einzelne Buchstaben durch andere vertauscht. Das folgende Kommando wandelt alle eingehenden Kleinbuchstaben in Großbuchstaben um[7]:

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

[7] Umlaute und Sonderzeichen ausgeschlossen

Normalerweise werden alle Eingabezeilen nach der Bearbeitung wieder ausgegeben, unabhängig davon ob sie verändert wurden oder nicht. Das Verhalten kann über den Kommandozeilenparameter `-n` abgeschaltet werden. Da dann allerdings nichts mehr ausgegeben wird kann durch ein an ein Kommando angehängtes `p` bestimmt werden, daß die Veränderten Zeilen – und nur die – ausgegeben werden.

Beispiele

Da es in diesem Text nicht um `sed`-Skripte, sondern um Shell-Skripte gehen soll werden hier keine komplexen Sachen vorgestellt, sondern nur ein paar Einzeiler. Nichtsdestotrotz können es auch diese unscheinbaren Aufrufe in sich haben.

```
### SUCHEN UND ERSETZEN

# Im kompletten Text 'rot' durch 'blau' ersetzen:
sed 's/rot/blau/'   # Ersetzt nur das erste Vorkommen in jeder Zeile
sed 's/rot/blau/4'  # Ersetzt nur das vierte Vorkommen in jeder Zeile
sed 's/rot/blau/g'  # Ersetzt nur jedes Vorkommen in jeder Zeile

# 'rot' durch 'blau' ersetzen, aber NUR in Zeilen die auch 'gelb'
# enthalten:
sed '/gelb/s/rot/blau/g'

# 'rot' durch 'blau' ersetzen, AUSSER in Zeilen die auch 'gelb'
# enthalten:
sed '/gelb/!s/rot/blau/g'

# 'rosa', 'hellrot' und 'magenta' durch 'pink' ersetzen:
sed 's/rosa/pink/g;s/hellrot/pink/g;s/magenta/pink/g'
gsed 's/rosa\|hellrot\|magenta/pink/g'                # nur in GNU sed

# Jede Zeile um fünf Leerzeichen einrücken:
# lies: 'ersetze jeden Zeilenanfang durch fünf Leerzeichen'
sed 's/^/     /'

# Führende Blanks (Leerzeichen, Tabulatoren) von den Zeilenanfängen
# löschen:
# ACHTUNG: An Stelle des \t muß der Tabulator gedrückt werden, die
#          Darstellung als \t versteht nicht jedes sed!
sed 's/^[ \t]*//'

# Schliessende Blanks vom Zeilenende löschen, siehe oben:
sed 's/[ \t]*$//'

# Führende und schließende Blanks löschen:
sed 's/^[ \t]*//;s/[ \t]*$//'

# Wenn eine Zeile mit Backslash aufhört den Zeilenumbruch entfernen:
```

```
sed -e :a -e '/\\$/N; s/\\\n//; ta'

### BESTIMMTE ZEILEN AUSGEBEN

# Nur Zeile 42 ausgeben:
sed -n '42p'                # Methode 1
sed '42!d'                  # Methode 2

# Nur die Zeilen 23-42 ausgeben (inklusive):
sed -n '23,42p'             # Methode 1
sed '23,42!d'               # Methode 2

# Von einem regulären Ausdruck bis zum Dateiende ausgeben:
sed -n '/regexp/,$p'

# Den Bereich zwischen zwei regulären Ausdrücken ausgeben (inklusive):
sed -n '/rot/,/blau/p'

# Nur Zeilen mit mindestens 42 Zeichen ausgeben:
sed -n '/^.\{42\}/p'

# Nur Zeilen mit höchstens 42 Zeichen ausgeben:
sed -n '/^.\{42\}/!p'       # Methode 1, analog zu oben
sed '/^.\{42\}/d'           # Methode 2, einfachere Syntax

### BESTIMMTE ZEILEN LÖSCHEN

# Die ersten zehn Zeilen löschen:
sed '1,10d'

# Die letzte Zeile löschen:
sed '$d'

# Alles außer dem Bereich zwischen zwei regulären Ausdrücken ausgeben:
sed '/rot/,/blau/d'

# Alle Leerzeilen löschen:
sed '/^$/d'                         # Methode 1
sed '/./!d'                         # Methode 2

# Alle Leerzeilen am Dateianfang löschen:
sed '/./,$!d'
```

5.2.39 seq

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Oft wird eine auf- oder absteigende Sequenz aufeinanderfolgender Zahlen benötigt, beispielsweise um eine Schleife 100 mal zu durchlaufen. Es ist nicht sehr performant bei jedem Schleifendurchlauf hochzuzählen und dann die entstandene Zahl mit dem Limit zu vergleichen. Daher nimmt man an der Stelle `seq` zur Hilfe, wenn es zur Verfügung steht.

Die zu zählenden Werte werden durch drei unterschiedliche Arten der Parametrisierung definiert: Ein Aufruf in der Form `seq 10` gibt die Zahlen von 1 bis 10 aus. Mit `seq 10 20` wird von 10 bis 20 gezählt, und `seq 20 -2 10` zählt in zweierschritten rückwärts von 20 nach 10.

Per default kommen die Werte zeilenweise, mit dem Parameter `-s` kann aber auch ein anderes Trennzeichen definiert werden. Will man etwas numerieren und später nach den Zahlen sortieren, ist es sinnvoll wenn ‘schmalere’ Zahlen mit führenden Nullen aufgefüllt werden. Das erreicht man mit dem Parameter `-w`.

5.2.40 sleep

Das Kommando `sleep` veranlaßt die Shell, für eine angegebene Zeit zu warten. Die Zeit wird dabei in Sekunden angegeben.

☞ In der GNU-Variante von `sleep` kann die Einheit der angegebenen Zeitspanne durch Suffixe definiert werden: `sleep 10s` schläft zehn Sekunden, `sleep 10m` zehn Minuten. Genauso werden Stunden (h) und Tage (d) definiert.

Außerdem kann die GNU-Variante auch mit nicht-Integer Zeiten arbeiten: `sleep 0.5` schläft eine halbe Sekunde.

5.2.41 sort

Mit diesem Befehl wird wie der Name schon andeutet sortiert. Wenn kein Dateiname als Parameter angegeben wird, liest `sort` von der Standard-Eingabe. Geschrieben wird immer auf der Standard-Ausgabe.

Man kann sich vorstellen, daß ein solches Kommando recht flexibel sein muß, daher stehen eine Menge Parameter zur Verfügung:

`-b`	Ignoriert führende Leerzeichen
`-c`	Prüft nur, ob die Eingabedaten bereits sortiert sind
`-f`	Groß- / Kleinschreibung nicht beachten
`-i`	Nur ‘printable’ Zeichen beachten

`-k`	Sortiert nicht nach der ersten, sondern nach der angegebenen Spalte (siehe unten)
`-n`	Numerische Sortierung
`-r`	Sortierreihenfolge umkehren
`-t`	Normalerweise fängt ein Feld (siehe `-k`) beim Übergang von ‘blank’ nach ‘non-blank’ an, mit diesem Parameter können ander Feld-Trenner definiert werden
`-u`	Gibt bei identischen Zeilen nur die erste Fundstelle aus (unique)

Die Sortierung nach der Spalte (mit `-k`) ist etwas tricky. Die genaue Syntax wird in der Man-Page mit `-k POS1[,POS2]` angegeben, das bedeutet man muß einen Parameter angeben, man kann bei Bedarf einen zweiten angeben. Bei der Sortierung wird dann der Bereich ab POS1, bzw. der Bereich zwischen POS1 und POS2 berücksichtigt.

Dabei lautet die Syntax für POS `F[.C][OPTS]`. Dabei gibt F die Feldnummer an (siehe Parameter `-t`). Wenn nicht nach dem Feld an sich sortiert werden soll, kann C die Position des Zeichens innerhalb des Feldes angeben. Und als ob das noch nicht kompliziert genug wäre, kann man dem ganzen Konstrukt noch einen einbuchstabigen Parameter für die Sortier-Option mitgeben.

Wenn das angegebene Feld nicht existiert wird nach der ganzen Zeile sortiert.

OK, Beispiele:

`-k 3`	Sortiert nach dem dritten Feld
`-k 3.5`	Sortiert nach dem fünften Zeichen des dritten Feldes
`-k 3.5r`	Sortiert rückwärts nach dem fünften Zeichen des dritten Feldes
`-k 3,5`	Beachtet bei der Sortierung nur das dritte bis fünfte Feld
`-k 3.5,3.8`	Beachtet die Zeichen fünf bis acht des dritten Feldes

Weitere Parameter verrät wie immer die Man-Page.

5.2.42 tail

Der Befehl `tail` gibt die letzten zehn Zeilen einer Datei aus. Wenn kein Dateiname (oder ein `-`) angegeben wird, liest `tail` von der Standard-Eingabe. Man kann die Anzahl der ausgegebenen Zeilen mit dem Parameter `-n` steuern.

Mit dem Parameter `-f` (follow) gibt `tail` neue Zeilen aus, sobald sie an die Datei angehängt werden.

☞ Die GNU-Version kann auch das Ende mehrere Dateien ausgeben bzw. verfolgen, wenn mehrere Namen angegeben werden.

5.2.43 tee

Dies ist praktisch ein T-Stück für Pipes. `tee` liest von seiner Standard-Eingabe, und gibt alle Eingaben direkt auf der Standard-Ausgabe wieder aus. Nebenbei werden die Ausgaben in eine oder mehrere Dateien geschrieben.

Wenn die Ausgabedateien schon existieren, werden sie überschrieben. Dieses Verhalten kann mit dem Parameter `-a` geändert werden.

5.2.44 touch

Mit diesem Kommando kann man einerseits Dateien anlegen wenn sie nicht existieren, und andererseits die Änderungs- und Zugriffszeiten einer Datei ändern. Ohne die Angabe weiterer Parameter wird die Datei erzeugt wenn sie nicht existierte, bzw. in ihrer Änderungs- und Zugriffszeit auf die aktuelle Zeit gesetzt.

Mit dem Parameter `-a` wird nur die Zugriffs-, mit `-m` nur die Änderungszeit gesetzt. Mit `-c` kann die Erstellung einer neuen Datei unterdrückt werden.

Die eingesetzte Zeit kann auch durch die Parameter `-t` bzw. `-d` angegeben werden. Mit `-r` kann die Zeit der einer angegebenen Referenzdatei angepaßt werden.

5.2.45 tr

Will man ganze Worte oder komplexe Muster in Dateien oder Pipes suchen und ersetzen, greift man üblicherweise zu `sed` (Abschnitt 5.2.38). Für einzelne Buchstaben nimmt man hingegen `tr`.

Normalerweise wird `tr` mit zwei Zeichenketten als Parametern aufgerufen und übernimmt die zu konvertierenden Daten von der Standard-Eingabe. Jedes Zeichen im eingehenden Datenstrom wird anhand der beiden Zeichenketten ersetzt, dabei wird das erste Zeichen der ersten Kette durch das erste Zeichen der zweiten Kette ersetzt, das zweite durch das zweite, und so weiter.

Ist die zweite Zeichenkette länger als die erste, werden überschüssige Zeichen ignoriert. Ist die zweite Zeichenkette kürzer als die erste, wird ihr letztes Zeichen so lange wiederholt bis sie gleich sind. Durch den Parameter `-t` kann dieses Verhalten abgeschaltet werden, so daß überschüssige Zeichen abgeschnitten werden.

Mit dem Parameter `-c` wird die erste Zeichenkette ‘invertiert’, es werden also alle Zeichen ersetzt die nicht darin vorkommen.

`tr` kann aber auch mit nur einer Zeichenkette aufgerufen werden, wenn die Parameter `-d` oder `-s` benutzt werden. Mit `-d` werden alle Zeichen aus dem Eingabestrom gelöscht, die in der Zeichenkette vorkommen. Mit `-s` werden doppelt vorkommende Zeichen durch ein einzelnes ersetzt.

Die Zeichenketten an sich können übrigens nicht nur Buchstaben oder Zahlen enthalten, sondern auch Sonderzeichen oder Zeichenklassen. Näheres dazu steht in der Man-Page.

Die folgenden Beispiele verdeutlichen die Anwendung:

```
text="Dies ist ein Testtext"

# kleine Umlaute durch grosse ersetzen:
echo "$text" | tr aeiou AEIOU
# -> DIEs Ist EIn TEsttExt

# Kleinbuchstaben durch Großbuchstaben ersetzen:
echo "$text" | tr a-z A-Z
# -> DIES IST EIN TESTTEXT

# alle Vokale durch Unterstriche ersetzen:
echo "$text" | tr aeiouAEIOU _
# -> D__s _st __n T_stt_xt

# Großbuchstaben löschen:
echo "$text" | tr -d A-Z
# -> ies ist ein esttext

# doppelte Buchstaben löschen:
echo "$text" | tr -s "a-zA-Z"
# -> Dies ist ein Testext

# doppelte Buchstaben löschen, mit Zeichenklasse:
echo "$text" | tr -s "[:alpha:]"
# -> Dies ist ein Testext
```

5.2.46 trap

Wie alle anderen Prozesse in einem Unix-System auch, so können auch Shell-Skripte Signale empfangen. Diese können durch Kommandos wie `kill` (Abschnitt 5.2.23) geschickt worden sein, oder zum Beispiel durch einen Tastatur-Interrupt.

Mit `trap` kann ein Skript darauf vorbereitet werden, ein oder mehrere Signale zu empfangen. Beim Aufruf wird eine Aktion mitgegeben, und eine oder mehrere Bedingungen die zum Ausführen der Aktion führen sollen. Das folgende Kommando gibt zm Beispiel eine Fehlermeldung aus wenn sein Skript ein Signal 1 (HUP), 2 (INT) oder 15 (TERM) erhält:

```
trap 'echo "`basename $0`: Ooops..." 1>&2' 1 2 15
```

Die Zeile ist dem Beispiel aus Abschnitt A.5 entnommen, dort findet sich auch nochmal eine ausführliche Erklärung.

Ein weiterer nützlicher Einsatz für `trap` ist es, Signale zu ignorieren. Das kann gewünscht sein, wenn eine Folge von Kommandos in einem Skript auf keinen Fall unterbrochen werden darf. Um zu verhindern daß ein CTRL+C des Benutzers das Skript beendet wird folgendes Konstrukt eingesetzt:

```
trap '' 2  # Signal 2 ist Ctrl-C, jetzt deaktiviert.
kommando1
kommando2
kommando3
trap 2     # Reaktiviert Ctrl-C
```

Vielleicht wäre es aber auch dem Benutzer gegenüber freundlicher, auf das entkräftete Signal hinzuweisen:

```
trap 'echo "Ctrl-C ist ausser Kraft."' 2
```

Eine Sonderbehandlung machen viele Shells, wenn als Signal `DEBUG` angegeben wird. Dann wird nach jedem ausgeführten Kommando der Trap ausgelöst. Dieses Feature wird wie der Name schon erahnen läßt zum Debuggen benutzt, ein Beispiel findet sich in Abschnitt 4.2.

5.2.47 type

Das in die Shell eingebaute Kommando `type` gibt Auskunft über die Art eines ausführbaren Kommandos. So kann man herausfinden ob beim Absetzen des Befehls ein externes Programm gestartet, eine Shell-Funktion ausgeführt oder ein Alias benutzt wird.

Sucht man nur nach einer ausführbaren Datei, hilft `which` (Abschnitt 5.2.51).

5.2.48 uniq

Mit dem Kommando `uniq` werden doppelt vorkommende Zeilen in einer Eingabedatei oder der eingehenden Pipe (Standard-Eingabe) bearbeitet. Per default steht ‘bearbeitet’ an dieser Stelle für ‘gelöscht’, aber durch Parameter kann dieses Verhalten angepaßt werden.

☞ Einige der folgenden Parameter entsprechen nicht dem allgemeinen Standard:

`-c`	Anzahl der Vorkommnisse vor die Zeilen schreiben
`-d`	Nur doppelte Zeilen ausgeben, jede nur einmal
`-D`	☞ Alle doppelten Zeilen ausgeben
`-f` *n*	Die ersten *n* Felder ignorieren
`-i`	☞ Groß- / Kleinschreibung ignorieren
`-s` *n*	Die ersten *n* Zeichen ignorieren
`-u`	Nur einfach vorkommende Zeilen ausgeben
`-w` *n*	☞ Nur die ersten *n* Zeichen betrachten

Achtung: `uniq` betrachtet beim Vergleich nur direkt aufeinander folgende Zeilen. Sollen alle Duplikate Dateiweit betrachtet werden, bietet sich ein ‘vorsortieren’ mit `sort` (Abschnitt 5.2.41) an, vielleicht sogar ausschließlich ein `sort -u`.

5.2.49 wait

Das Kommando `wait` wird benutzt um auf die Beendigung eines Prozesses zu warten. Als Parameter wird eine Prozeß-ID übergeben, `wait` läuft so lange bis sich der Prozeß mit der angegebenen ID beendet hat.

Wenn keine Prozeß-ID angegeben wurde, wartet `wait` auf alle Kind-Prozesse der aktuellen Shell.

Ein Beispiel für die Benutzung findet sich im Kapitel über Wachhunde (Abschnitt B.4).

5.2.50 wc

Wie der Name schon suggeriert[8] kann man mit diesem Kommando Wörter zählen (word count). Gezählt wird entweder in einer Datei, oder – wenn kein Dateiname angegeben wurde – in der Standardeingabe.

Weitaus häufiger wird aber der Parameter `-l` benutzt, mit dem sich die Zeilen zählen lassen. Weiterhin kann man Bytes (`-c`) oder Zeichen (`-m`) zählen lassen.

☞ Der Parameter `-L` gibt in der GNU-Version die Länge der längsten enthaltenen Zeile aus.

5.2.51 which

☞ Dies ist kein Standard-Kommando, es steht nicht auf allen Systemen zur Verfügung.

Sucht im Pfad (vordefinierte Variable `$PATH`, siehe Abschnitt 4.5) nach einer Ausführbaren Datei. Wenn mehrere Dateien auf das Suchwort passen wird die erste Fundstelle ausgegeben, also die Datei die tatsächlich ausgeführt würde. Mit `-a` werden alle Fundstellen ausgegeben.

Einen ähnlichen Zweck erfüllt auch `type` (Abschnitt 5.2.47).

5.2.52 who

Das Kommando `who` gibt eine Liste aller angemeldeten Benutzer, zusammen mit deren aktueller Konsole und der Anmeldezeit aus.

5.2.53 xargs

Bisweilen kommt man in die Verlegenheit, versehentlich zu lange Einzeiler geschrieben zu haben. Neben den Fällen, in denen der Tipp-Eifer überhand genommen hat handelt es sich in aller Regel um Zeilen in der Art `grep 'text' $(find / -name \*.txt)`. Dieses Kommando sucht alle Dateien mit der Endung txt, die im System vorhanden sind. Diese werden ‘in die Kommandozeile eingebaut’. Wenn sehr viele Dateien gefunden werden, wird die Zeile zu lang für die Shell[9].

[8] Oder etwa nicht?!? ;-)

[9] Die maximale Länge der Kommandozeile unterscheidet sich von System zu System

Ein weiterer und in der Praxis mindestens ebenso sinnvoller Einsatzzweck ist das Vermeiden von Schleifen. Das obige Problem ließe sich auch mit einer Zeile in der folgende Form lösen:

```
find / -name \*.txt -exec grep 'text' {} \;
```

Allerdings hätte das den Nachteil, daß für jede gefundene Datei ein neuer `grep` gestartet werden muß. Das kostet Resourcen. Beide Probleme werden durch eine Zeile in der folgenden Form umgangen:

```
find / -name \*.txt | xargs grep 'text'
```

Dabei liest `xargs` aus der Standardeingabe die Parameter, die dann an den `grep`-Aufruf angehängt werden. Sollten zu viele Dateien gefunden werden, wird `grep` mehrfach aufgerufen, allerdings im Gegensatz zum obigen Beispiel nicht einmal pro Fundstelle.

☞ Neben einigen anderen Parametern informiert die Manpage der GNU-Version über die Option `-r`. Damit kann vermieden werden, daß `xargs` das Kommando startet wenn keine Eingabe vorhanden ist. Bezogen auf das angegebene Beispiel würde `grep` ohne Dateinamen gestartet, wenn `find` nichts findet. Es würde auf Input von der Standardeingabe warten, der aber wahrscheinlich nicht kommt. Das Skript würde hängen, wenn der Parameter `-r` nicht angewandt würde.

A Beispiele

A.1 Schleifen und Rückgabewerte

Man kann mit einer `until`- bzw. mit einer `while`-Schleife schnell kleine aber sehr nützliche Tools schreiben, die einem lästige Aufgaben abnehmen.

A.1.1 Schleife, bis ein Kommando erfolgreich war

Angenommen, bei der Benutzung eines Rechners tritt ein Problem auf, bei dem nur der Administrator helfen kann. Dann möchte man informiert werden, sobald dieser an seinem Arbeitsplatz ist. Man kann jetzt in regelmäßigen Abständen das Kommando `who` ausführen, und dann in der Ausgabe nach dem Eintrag ‘`root`’ suchen. Das ist aber lästig.

Einfacher geht es, wenn wir uns ein kurzes Skript schreiben, das alle 30 Sekunden automatisch überprüft, ob der Admin angemeldet ist. Wir erreichen das mit dem folgenden Code:

```
#!/bin/sh
until who | grep "^root "; do
  sleep 30
done
echo "Big Brother is watching you!"
```

Das Skript führt also so lange das Kommando aus, bis die Ausführung erfolgreich war. Dabei wird die Ausgabe von `who` mit einer Pipe (4.15) in das `grep`-Kommando umgeleitet. Dieses sucht darin nach einem Auftreten von ‘`root `’ am Zeilenanfang. Der Rückgabewert von `grep` ist 0 wenn das Muster gefunden wird, 1 wenn es nicht gefunden wird und 2 wenn ein Fehler auftrat. Damit der Rechner nicht die ganze Zeit mit dieser Schleife beschäftigt ist, wird im Schleifenkörper ein `sleep 30` ausgeführt, um den Prozeß für 30 Sekunden schlafen zu schicken. Sobald der Admin sich eingeloggt hat, wird eine entsprechende Meldung ausgegeben.

A.1.2 Schleife, bis ein Kommando erfolglos war

Analog zum vorhergehenden Beispiel kann man auch ein Skript schreiben, das meldet, sobald sich ein Benutzer abgemeldet hat. Dazu ersetzen wir nur die `until`- Schleife durch eine entsprechende `while`-Schleife:

```
#!/bin/sh
while who | grep "^root "; do
  sleep 30
done
echo "Die Katze ist aus dem Haus, Zeit, daß die Mäuse tanzen!"
```

Die Schleife wird nämlich dann so lange ausgeführt, bis `grep` einen Fehler (bzw. eine erfolglose Suche) zurückmeldet.

A.2 Subshell-Schleifen vermeiden

Wir wollen ein Skript schreiben, das die `/etc/passwd` liest und dabei zählt, wie viele Benutzer eine UID kleiner als 100 haben.

Folgendes Skript funktioniert nicht:

```
#!/bin/sh
count=0
cat /etc/passwd | while read i; do
  uid=‘echo $i | cut -f 3 -d:‘
  if [ $uid -lt 100 ]; then
    count=‘expr $count + 1‘
    echo $count
  fi
done
echo Es sind $count Benutzer mit einer ID kleiner 100 eingetragen
```

Was ist passiert?

Dieses Skript besteht im Wesentlichen aus einer Pipe. Wir haben ein `cat`-Kommando, das den Inhalt der `/etc/passwd` durch eben diese Pipe an eine Schleife übergibt. Das `read`-Kommando in der Schleife liest die einzelnen Zeilen aus, dann folgt ein Bißchen Auswertung.

Es ist zu beobachten, daß bei der Ausgabe in Zeile 7 die Variable `$count` korrekte Werte enthält. Um so unverständlicher ist es, daß sie nach der Vollendung der Schleife wieder den Wert 0 enthält.

Das liegt daran, daß diese Schleife als Teil einer Pipe in einer Subshell ausgeführt wird. Die Variable `$count` steht damit in der Schleife praktisch nur lokal zur Verfügung, sie wird nicht an das umgebende Skript ‘hochgereicht’.

Neben der Methode in B.3.1 bietet sich hier eine viel einfachere Lösung an:

```
#!/bin/sh
count=0
while read i; do
  uid=‘echo $i | cut -f 3 -d:‘
  if [ $uid -lt 100 ]; then
    count=‘expr $count + 1‘
    echo $count
  fi
done < /etc/passwd
echo Es sind $count Benutzer mit einer ID kleiner 100 eingetragen
```

Hier befindet sich die Schleife nicht in einer Pipe, daher wird sie auch nicht in einer Subshell ausgeführt. Man kann auf das `cat`-Kommando verzichten und den Inhalt der Datei durch die Umlenkung in Zeile 9 direkt auf die Standardeingabe der Schleife (und somit auf das `read`-Kommando) legen.

A.3 Ein typisches Init-Skript

Dieses Skript dient dazu, den Apache HTTP-Server zu starten. Es wird während des Bootvorgangs gestartet, wenn der dazugehörige Runlevel initialisiert wird.

Das Skript muß mit einem Parameter aufgerufen werden. Möglich sind hier *start*, *stop*, *status*, *restart* und *reload*. Wenn falsche Parameter übergeben wurden, wird eine entsprechende Meldung angezeigt.

Das Ergebnis der Ausführung wird mit Funktionen dargestellt, die aus der Datei `functions` stammen. Ebenfalls in dieser Datei sind Funktionen, die einen Dienst starten oder stoppen.

Zunächst wird festgelegt, daß dieses Skript in der Bourne-Shell ausgeführt werden soll (4.13.2).

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

Dann folgen Kommentare, die den Sinn des Skriptes erläutern (4.13.1).

```
#
# Startup script for the Apache Web Server
#
# chkconfig: 345 85 15
# description: Apache is a World Wide Web server.  It is \
```

```
#              used to serve HTML files and CGI.
# processname: httpd
# pidfile: /var/run/httpd.pid
# config: /etc/httpd/conf/access.conf
# config: /etc/httpd/conf/httpd.conf
# config: /etc/httpd/conf/srm.conf
```

Jetzt wird die Datei mit den Funktionen eingebunden (4.13.4).

```
# Source function library.
. /etc/rc.d/init.d/functions
```

Hier werden die Aufrufparameter ausgewertet (4.13.8).

```
# See how we were called.
case "$1" in
  start)
        echo -n "Starting httpd: "
```

Nachdem eine Meldung über den auszuführenden Vorgang ausgegeben wurde, wird die Funktion `daemon` aus der Funktionsbibliothek ausgeführt. Diese Funktion startet das Programm, dessen Name hier als Parameter übergeben wird. Dann gibt sie eine Meldung über den Erfolg aus.

```
        daemon httpd
        echo
```

Jetzt wird ein Lock-File[1] angelegt.

```
        touch /var/lock/subsys/httpd
        ;;
  stop)
        echo -n "Shutting down http: "
```

Hier passiert im Prinzip das gleiche wie oben, nur daß mit der Funktion `killproc` der Daemon angehalten wird.

```
        killproc httpd
        echo
```

Danach werden Lock-File und PID-File[2] gelöscht.

```
        rm -f /var/lock/subsys/httpd
        rm -f /var/run/httpd.pid
        ;;
  status)
```

[1] Ein Lock-File signalisiert anderen Prozessen, daß ein bestimmter Prozeß bereits gestartet ist. So kann ein zweiter Aufruf verhindert werden.

[2] In einem sogenannten PID-File hinterlegen einige Prozesse ihre Prozeß-ID, um anderen Programmen den Zugriff zu erleichtern (z. B. um den Prozeß anzuhalten etc).

Die Funktion `status` stellt fest, ob der entsprechende Daemon bereits läuft, und gibt das Ergebnis aus.

```
31        status httpd
          ;;
33    restart)
```

Bei Aufruf mit dem Parameter *restart* ruft sich das Skript zwei mal selbst auf (in `$0` steht der Aufrufname des laufenden Programms). Einmal, um den Daemon zu stoppen, dann, um ihn wieder zu starten.

```
          $0 stop
35        $0 start
          ;;
37    reload)
          echo -n "Reloading httpd: "
```

Hier sendet die `killproc`-Funktion dem Daemon ein Signal das ihm sagt, daß er seine Konfiguration neu einlesen soll.

```
39        killproc httpd -HUP
          echo
41        ;;
    *)
43        echo "Usage: $0 {start|stop|restart|reload|status}"
```

Bei aufruf mit einem beliebigen anderen Parameter wird eine Kurzhilfe ausgegeben. Dann wird dafür gesorgt, daß das Skript mit dem Exit-Code 1 beendet wird. So kann festgestellt werden, ob das Skript ordnungsgemäß beendet wurde (4.13.14).

```
          exit 1
45  esac

47  exit 0
```

A.4 Parameterübergabe in der Praxis

Es kommt in der Praxis sehr oft vor, daß man ein Skript schreibt, dem derAnwender Parameter übergeben soll. Wenn das nur eine Kleinigkeit ist (zum Beispiel ein Dateiname), dann fragt man einfach die entsprechenden vordefinierten Variablen (4.5) ab. Sollen aber ‘richtige’ Parameter eingesetzt werden, die sich so einsetzen lassen wie man es von vielen Kommandozeilentools gewohnt ist, dann benutzt man das Hilfsprogramm `getopt`. Dieses Programm parst die originalen Parameter und gibt sie in ‘standardisierter’ Form zurück.

Das soll an folgendem Skript verdeutlicht werden. Das Skript kennt die Optionen `-a` und `-b`. Letzterer Option muß ein zusätzlicher Wert mitgegeben werden. Alle anderen Parameter werden als Dateinamen interpretiert.

```
#!/bin/sh
set -- ‘getopt "ab:" "$@"‘ || {
```

Das `set`-Kommando belegt den Inhalt der vordefinierten Variablen (4.5) neu, so daß es aussieht, als ob dem Skript die Rückgabewerte von `getopt` übergeben wurden. Man muß die beiden Minuszeichen angeben, da sie dafür sorgen, daß die Aufrufparameter an `getopt` und nicht an die Shell selbst übergeben werden. Die originalen Parameter werden von `getopt` untersucht und modifiziert zurückgegeben: `a` und `b` werden als Parameter Markiert, `b` sogar mit der Möglichkeit einer zusätzlichen Angabe.

Wenn dieses Kommando fehlschlägt ist das ein Zeichen dafür, daß falsche Parameter übergeben wurden. Also wird nach einer entsprechenden Meldung das Programm mit Exit-Code 1 verlassen.

```
   echo "Anwendung: ‘basename $0‘ [-a] [-b Name] Dateien" 1>&2
   exit 1
}
echo "Momentan steht in der Kommandozeile folgendes: $*"
aflag=0 name=NONE
while :
do
```

In einer Endlosschleife, die man mit Hilfe des Null-Befehls (`:`, 4.13.3) baut, werden die ‘neuen’ Parameter der Reihe nach untersucht. Wenn ein `-a` vorkommt, wird die Variable `aflag` gesetzt. Bei einem `-b` werden per `shift` alle Parameter nach Links verschoben, dann wird der Inhalt des nächsten Parameters in der Variablen `name` gesichert.

```
   case "$1" in
    -a) aflag=1 ;;
    -b) shift; name="$1" ;;
    --) break ;;
```

Wenn ein `--` erscheint, ist das ein Hinweis darauf, daß die Liste der Parameter abgearbeitet ist. Dann wird per `break` (4.13.13) die Endlosschleife unterbrochen. Die Aufrufparameter enthalten jetzt nur noch die eventuell angegebenen Dateinamen, die jetzt von dem Restlichen Skript wie gewohnt weiterverarbeitet werden können.

```
   esac
   shift
done
shift
```

Am Ende werden die Feststellungen ausgegeben.

```
echo "aflag=$aflag / Name = $name / Die Dateien sind $*"
```

A.5 Fallensteller: Auf Traps reagieren

Ein laufendes Shell-Skript kann durch Druck auf die Interrupt-Taste (CTRL+C) unterbrochen werden. Durch Druck auf diese Taste wird ein Signal an den entsprechenden Prozeß gesandt, das ihn bittet sich zu beenden. Dieses Signal heißt SIGINT (für SIGnal INTerrupt) und trägt die Nummer 2. Das kann ein kleines Problem darstellen, wenn das Skript sich temporäre Dateien angelegt hat, da diese nach der Ausführung nur noch unnötig Platz verbrauchen und eigentlich gelöscht werden sollten. Man kann sich sicher auch noch wichtigere Fälle vorstellen, in denen ein Skript bestimmte Aufgaben auf jeden Fall erledigen muß, bevor es sich beendet.

Es gibt eine Reihe weiterer Signale, auf die ein Skript reagieren kann. Alle sind in der Man-Page von `signal` beschrieben. Hier die wichtigsten:

Nummer	Name	Bedeutung
0	Normal Exit	Wird durch das `exit`-Kommando ausgelöst.
1	SIGHUP	Wenn die Verbindung abbricht (z. B. wenn das Terminal geschlossen wird).
2	SIGINT	Zeigt einen Interrupt an (CTRL+C).
15	SIGTERM	Wird vom `kill`-Kommando gesendet.

Wie löst man jetzt dieses Problem? Glücklicherweise verfügt die Shell über das `trap`-Kommando, mit dessen Hilfe man auf diese Signale reagieren kann. Die Anwendung soll in folgendem Skript beispielhaft dargestellt werden.

Das Skript soll eine komprimierte Textdatei mittels `zcat` in ein temporäres File entpacken, dieses mit `pg` seitenweise anzeigen und nachher wieder löschen.

```
#!/bin/sh
stat=1
temp=/tmp/zeige$$
```

Zunächst werden zwei Variablen belegt, die im weiteren Verlauf benutzt werden sollen. In `stat` wird der Wert abgelegt, den das Skript im Falle eines Abbruchs als Exit-Status zurückliefern soll. Die Variable `temp` enthält den Namen für eine temporäre Datei. Dieser setzt sich zusammen aus `/tmp/zeige` und der Prozeßnummer des laufenden Skripts. So soll sichergestellt werden, daß noch keine Datei mit diesem Namen existiert.

```
trap 'rm -f $temp; exit $stat' 0
trap 'echo "`basename $0`: Ooops..." 1>&2' 1 2 15
```

Hier werden die Traps definiert. Bei Signal 0 wird die temporäre Datei gelöscht und der Wert aus der Variable `stat` als Exit-Code zurückgegeben. Dabei wird dem `rm`-Kommando

der Parameter `-f` mitgegeben, damit keine Fehlermeldung ausgegeben wird, falls die Datei (noch) nicht existiert. Dieser Fall tritt bei jedem Beenden des Skriptes auf, also sowohl bei einem normalen Ende, als auch beim Exit-Kommando, bei einem Interrupt oder bei einem Kill. Der zweite Trap reagiert auf die Signale 1, 2 und 15. Das heißt, er wird bei jedem unnormalen Ende ausgeführt. Er gibt eine entsprechende Meldung auf die Standard-Fehlerausgabe (4.15) aus. Danach wird das Skript beendet, und der erste Trap wird ausgeführt.

```
case $# in
 1) zcat "$1" > $temp
    pg $temp
    stat=0
    ;;
```

Jetzt kommt die eigentliche Funktionalität des Skriptes: Das `case`-Kommando (4.13.8) testet die Anzahl der übergebenen Parameter. Wenn genau ein Parameter übergeben wurde, entpackt `zcat` die Datei, die im ersten Parameter angegeben wurde, in die temporäre Datei. Dann folgt die Seitenweise Ausgabe mittels `pg`. Nach Beendigung der Ausgabe wird der Status in der Variablen auf 0 gesetzt, damit beim Skriptende der korrekte Exit-Code zurückgegeben wird.

```
 *) echo "Anwendung: ‘basename $0‘ Dateiname" 1>&2
esac
```

Wenn `case` eine andere Parameterzahl feststellt, wird eine Meldung mit der Aufrufsyntax auf die Standard-Fehlerausgabe geschrieben.

A.6 Chaoten: Dateien in zufällige Reihenfolge bringen

Wir wollen uns einen MP3-Player programmieren, der Alle MP3-Dateien aus einem bestimmten Verzeichnisbaum in zufälliger Reihenfolge abspielt. Damit dieses Problem für uns eine Herausforderung darstellt[3], wollen wir vor dem Abspielen der jeweiligen Datei etwas mit dem Dateinamen anstellen. Ob das eine einfache Ausgabe per `echo` ist, oder ob der Name per Sprachsynthese oder auf einem externen Display angezeigt werden soll ist an dieser Stelle egal.

Das Problem ist, daß wir in der Shell nur über Umwege an Zufallszahlen kommen können. Auf Systemen, in denen die Datei `/dev/urandom` existiert, liefert uns der Kernel aber schon sehr zufällige Zeichenfolgen. Diese Folgen können alle Zeichen enthalten, daher müssen sie vor der Benutzung für unsere Zwecke noch etwas ‘bereinigt’ werden.

Wie das aussieht, wenn es fertig ist, sieht man im folgenden Skript:

[3] Denn schließlich hat mpg123 schon von Hause aus eine Random-Funktion.

```
#!/bin/sh
for i in ‘find $1 -type f -name "*.[mM][pP]3"‘; do
```

Hier beginnt eine Schleife, die über alle Ausgaben des `find`-Kommandos iteriert. Dabei sucht `find` nach allen normalen Dateien (`-type f`), die die Extension .mp3 tragen (`-name "*.[mM][pP]3"` – wir ignorieren Groß- / Kleinschreibung).

```
  echo ‘tr -dc "[:alpha:]" < /dev/urandom | \
        dd count=8 bs=1 2> /dev/null‘$i
```

Hier ist der ‘magische Teil’. Mit dem `echo` wird die Ausgabe einer Pipe ausgegeben, gefolgt von dem aktuellen Dateinamen. Diese Pipe enthält ein `tr`, der alle ungewollten Zeichen (alles, was kein Textzeichen ist) aus einem Datenstrom entfernt. Die Daten erhält `tr` durch die <-Umleitung aus oben genannter Datei.

Diese Datei liefert ‘ohne Ende’ Zeichen. Wir wollen aber nur acht Zeichen haben, die wir unserem Dateinamen voranstellen können. Dazu benutzen wir das Kommando `dd` mit den angegebenen Parametern. Damit die Erfolgsmeldung von `dd` nicht die Ausgabe verunstaltet, lenken wir sie nach `/dev/null` um.

```
done | sort | cut -b 9- | while read i; do
```

Das Ergebnis der obigen Schleife ist also die Liste der Dateinamen, denen jeweils acht zufällige Zeichen vorangestellt wurden. Die Reihenfolge entspricht allerdings immer noch der Ausgabe von `find`, wird also nach jedem Durchlauf gleich sein.

Um das zu ändern, pipen wir die Ausgabe der Schleife durch ein `sort`. Da die ersten acht Zeichen jeder Zeile zufällig sind, erhalten wir so eine zufällige Reihenfolge der Zeilen. Jetzt müssen wir nur noch durch ein `cut` die zufälligen Zeichen abschneiden, und erhalten so die ursprüngliche Liste von Dateien in einer zufälligen Reihenfolge.

Diese lesen wir jetzt zeilenweise mittels `read` ein. In der `while`-Schleife können wir alle erforderlichen Sachen mit dem Dateinamen anstellen. Hier wird er nur mittels `echo` ausgegeben.

```
  echo "Jetzt wird $i gespielt"
  mpg123 "$i"
done
```

A.7 Wer suchet, der findet

A.7.1 Prozesse suchen

Im Zusammenhang mit grep stößt fast jeder Shell-Skripter früher oder später auf das Problem, daß er irgendwas davon abhängig machen will, ob ein bestimmter Prozeß läuft oder

nicht. Im Normalfall wird er zuerst folgendes ausprobieren, was aber oft (nicht immer) in die Hose gehen wird:

```
ps aux | grep prozessname && echo "läuft schon"
```

Der Grund dafür ist, daß unter Umständen in der Ausgabe von `ps` auch das `grep`-Kommando samt Parameter (*prozessname*) aufgelistet wird. So findet das `grep`-Kommando sich quasi selbst.

Abhilfe schafft entweder `pgrep` (5.2.30) oder das folgende Konstrukt:

```
ps aux | grep "[p]rozessname" && echo "läuft schon"
```

Das p ist jetzt als eine Zeichenmenge (regulärer Ausdruck) angegeben worden. Jetzt sucht `grep` also nach dem String *prozessname*, in der Ausgabe von `ps` erscheint das `grep`-Kommando allerdings mit *[p]rozessname* und wird somit ignoriert.

A.7.2 Dateiinhalte suchen

Ein weiterer wertvoller Trick, diesmal im Zusammenhang mit `find`, ist folgendes Szenario: Es gibt ein Verzeichnis mit vielen Unterverzeichnissen, überall liegen Perl-Skripte und andere Dateien. Gesucht sind alle Dateien, in denen eine Zeile mit dem Inhalt ‘strict’ vorkommt. Man könnte jetzt folgendes versuchen:

```
grep -r strict *
```

Das führt allerdings dazu, daß alle Dateien durchsucht werden, nicht nur die Perl-Skripte. Diese tragen nach unserer Konvention[4] die Extension ‘.pl’. Wir starten also eine rekursive Suche über alle Dateien, die dem Muster entsprechen:

```
grep -r strict *.pl
```

Und wieder führt es nicht zu dem gewünschten Ergebnis. Da die Unterverzeichnisse nicht die Extension ‘*.pl’ tragen, werden sie nicht berücksichtigt. Für die Suche in Unterverzeichnissen ziehen wir `find` (Siehe Abschnitt 5.2.20) heran:

```
find . -name \*.pl -exec grep strict {} \;
```

Dieser Befehl gibt uns zwar die gefundenen Zeilen aus, nicht aber die Namen der Dateien. Es sieht für `grep` so aus als ob nur eine Datei durchsucht würde, da besteht keine Notwendigkeit den Namen anzugeben. Das ginge mit dem Parameter `-l`, allerdings würden uns dann nur noch die Dateinamen angezeigt. Eine Ausgabe mit beiden Informationen erhalten wir mit dem folgenden Konstrukt:

```
find . -name \*.pl -exec grep strict /dev/null {} \;
```

[4] Perl-Skripte müssen keine spezielle Extension haben, es sei aber um des Beispiels Willen mal angenommen.

Hier durchsucht `grep` nicht nur die gefundenen Dateien, sondern bei jedem Aufruf auch `/dev/null`, also den digitalen Mülleimer der per Definition leer ist. Da es für `grep` so aussieht als ob mehr als eine Datei durchsucht würden, wird bei jeder Fundstelle sowohl der Dateiname als auch die gefundene Zeile ausgegeben.

B Schmutzige Tricks :-)

Eigentlich sind diese Tricks gar nicht so schmutzig. Hier ist lediglich eine Sammlung von Beispielen, die genial einfach oder genial gut programmiert sind. Das bedeutet nicht, daß jeder Shell-Programmierer diese Techniken benutzen sollte. Ganz im Gegenteil. Einige Mechanismen bergen Gefahren, die nicht auf den ersten Blick erkennbar sind.

Mit anderen Worten: **Wenn Du diese Techniken nicht verstehst, dann benutze sie nicht!**

Die Intention hinter diesem Abschnitt ist es, dem gelangweilten Skripter etwas interessantes zum Lesen zu geben. Das inspiriert dann vielleicht dazu, doch einmal in den fortgeschrittenen Bereich vorzustoßen, neue Techniken kennenzulernen. Außerdem kann das Wissen über gewisse Techniken eine große Hilfe beim Lesen fremder Skripte darstellen, die eventuell von diesen Techniken Gebrauch machen.

Diese Techniken sind nicht ‘auf meinem Mist gewachsen’, sie stammen vielmehr aus Skripten, die mir im Laufe der Zeit in die Finger gekommen sind. Ich danke an dieser Stelle den klugen Köpfen, die sich solche Sachen einfallen lassen haben.

B.1 Die Tar-Brücke

Eine sogenannte Tar-Brücke benutzt man, wenn eine oder mehrere Dateien zwischen Rechnern übertragen werden sollen, aber kein Dienst wie SCP oder FTP zur Verfügung steht. Außerdem hat die Methode den Vorteil, daß Benutzerrechte und andere Dateiattribute bei der Übertragung erhalten bleiben[1].

Der Trick besteht darin, auf einer Seite der Verbindung etwas mit `tar` einzupacken, dies durch eine Pipe auf die andere Seite der Verbindung zu bringen und dort wieder zu entpacken.

Wenn dem Kommando `tar` an Stelle eines Dateinamens ein Minus-Zeichen als Archiv gegeben wird, benutzt es – je nach der gewählten Aktion – die Standard-Ein- bzw. -Ausgabe. Diese kann an ein weiteres `tar` übergeben werden um wieder entpackt zu werden.

Ein Beispiel verdeutlicht diese Kopier-Fähigkeit:

[1] Vorausgesetzt natürlich, daß der Benutzer auf der entfernten Seite über die nötigen Rechte verfügt.

```
tar cf - . | ( cd /tmp/backup; tar xf - )
```

Hier wird zunächst der Inhalt des aktuellen Verzeichnisses ‘verpackt’. Das Resultat wird an die Standard-Ausgabe geschrieben. Auf der Empfängerseite der Pipe wird eine Subshell geöffnet. Das ist notwendig, da das empfangende `tar` in einem anderen Verzeichnis laufen soll. In der Subshell wird zunächst das Verzeichnis gewechselt. Dann liest ein `tar` von der Standard-Eingabe und entpackt alles was er findet. Sobald keine Eingaben mehr kommen, beendet sich der Prozeß mitsamt der Subshell.

Am Ziel-Ort finden sich jetzt die gleichen Dateien wie am Quell-Ort.

Das ließe sich lokal natürlich auch anders lösen. Man könnte erst ein Archiv erstellen, das dann an anderer Stelle wieder auspacken. Nachteil: Es muß genügend Platz für das Archiv vorhanden sein. Denkbar wäre auch ein in den Raum gestelltes

```
cp -Rp * /tmp/backup
```

Allerdings fehlen einem dabei mitunter nützliche `tar`-Optionen[2], und die oben erwähnte Brücke wäre mit einem reinen `cp` nicht möglich.

Eine Seite der Pipe kann nämlich auch ohne Probleme auf einem entfernten Rechner ‘stattfinden’. Kommandos wie `ssh` oder `rsh` (letzteres nur unter Vorsicht einsetzen!) schlagen die Brücke zu einem anderen System, dort wird entweder gepackt und versendet oder quasi die Subshell gestartet und gelesen. Das sieht wie folgt aus:

```
ssh 192.168.2.1 tar clf - / | (cd /mnt/backup; tar xf - )
```

Hier wird auf einem entfernten Rechner die Root-Partition verpackt, per SSH in das lokale System geholt und lokal im Backup-Verzeichnis entpackt.

Der Weg in die andere Richtung ist ganz ähnlich:

```
tar cf - datei.txt | ssh 192.168.2.1 "(mkdir -p $PWD ;cd $PWD; tar xf -)"
```

Hier wird die Datei verpackt und versendet. Eine Besonderheit gegenüber dem vorigen Beispiel besteht darin, daß das Zielverzeichnis bei Bedarf erstellt wird, bevor die Datei dort entpackt wird. Zur Erklärung: Die Variable `$PWD` wird, da sie nicht von Ticks ‘gesichert’ wird, schon lokal durch die Shell expandiert. An dieser Stelle erscheint also auf dem entfernten System der Name des aktuellen Verzeichnisses auf dem lokalen System.

[2] Mit `-l` verläßt `tar` beispielsweise nicht das File-System. Nützlich wenn nur eine Partition gesichert werden soll.

B.2 Binaries inside

Software wird meistens in Form von Paketen verteilt. Entweder handelt es sich dabei um auf das Betriebssystem abgestimmte Installationspakete (rpm, deb, pkg usw.), gepackte Archive (zip, tgz) oder Installationsprogramme. Unter Unix-Systemen bietet sich für letztere die Shell als Trägersystem an. Shell-Skripte sind mit wenigen Einschränkungen plattformunabhängig, sie können also ohne vorherige Installations- oder Compilier-Arbeiten gestartet werden und die Umgebung für das zu installierende Programm testen und / oder vorbereiten.

Abgesehen davon können Skripte mit den hier vorgestellten Techniken auch andere Daten, z. B. Bilder oder Töne, enthalten.

Doch wie werden die – üblicherweise binären – Pakete auf das Zielsystem gebracht?

Im Prinzip gibt es dafür zwei unterschiedliche Verfahren:

B.2.1 Binäre Here-Dokumente

Eine Möglichkeit ist es, die binäre Datei in Form eines Here-Dokuments mitzuliefern. Da es aber in der Natur einer binären Datei liegt nicht-druckbare Zeichen zu enthalten, kann die Datei mit Hilfe des Tools uuencode vorbereitet werden. Das Tool codiert Eingabedateien so, daß sie nur noch einfache Textzeichen enthalten.

Sehen wir uns das folgende einfache Beispiel an. Es ist etwas wild konstruiert und nicht sehr sinnvoll, aber es zeigt das Prinzip.

```
#!/bin/sh

echo "Das Bild wird ausgepackt..."

uudecode << 'EOF'
begin 644 icon.png
MB5!.1PT*&@H````-24A$4@```!8````6"`8```#$M&P[````"7!(67,```L3
M```+$P$`FIP8````!&=!34$``+&.?/M1DP```"!C2%)-``!Z)0``@(,```/G_
```

Nach einem Hinweis wird also das Here-Dokument als Eingabe für das Tool uudecode benutzt. Erstellt wurde das Dokument mit einer Zeile der folgenden Form:

```
uuencode icon.png icon.png
```

Wie man sieht ist der Name der Datei in dem Here-Dokument enthalten. Die Datei wird entpackt und unter diesem gespeichert. In der ‘realen Welt’ muß an der Stelle auf jeden Fall sichergestellt werden, daß keine existierenden Dateien versehentlich überschrieben werden.

Um diesen Abschnitt nicht allzu lang werden zu lassen überspringen wir einen Teil der Datei.

```
38 M#-""F4%,@%4.GUZ‘‘"(*‘6VW6!S#\>C_?/;__Q<R_S]<P/F7AXDA’I\>@‘‘B
   K!>E;2S-,]5!A7‘,,U’0@GQ6?8H‘‘‘P‘#@&?)O’P’L0‘‘‘‘!)14Y$KD)@@@‘‘
40 ‘
   end
42 EOF

44 if [ $? -ne 0 ]; then
     echo "Fehler beim Auspacken der Datei"
46   exit 1
   fi
48
   display icon.png
```

Nach dem Entpacken wird noch der Exit-Code von `uudecode` überprüft und im Fehlerfall eine Ausgabe gemacht. Im Erfolgsfall wird das Bild mittels `display` angezeigt.

B.2.2 Schwanz ab!

Diese Variante basiert darauf, daß die binäre Datei ohne weitere Codierung an das Shell-Skript angehängt wurde. Nachteil dieses Verfahrens ist, daß das ‘abschneidende Kommando’ nach jeder Änderung der Länge des Skriptes angepaßt werden muß.

Dabei gibt es zwei Methoden, die angehängte Datei wieder abzuschneiden. Die einfachere Methode funktioniert mit `tail`:

```
tail -n +227 $0 > icon.png
```

Dieses Beispiel geht davon aus, daß das Skript selbst 227 Zeilen umfaßt. Die binäre Datei wurde mit einem Kommando wie `cat icon.png >> skript.sh` an das Skript angehängt.

Für die etwas kompliziertere Variante muß die Länge des eigentlichen Skript-Teiles genau angepaßt werden. Wenn das Skript beispielsweise etwa 5,5kB lang ist, müssen genau passend viele Leerzeilen oder Kommentarzeichen angehängt werden, damit sich eine Länge von 6kB ergibt. Dann kann das Anhängsel mit dem Kommando `dd` in der folgenden Form abgeschnitten werden:

```
dd bs=1024 if=$0 of=icon.png skip=6
```

Das Kommando kopiert Daten aus einer Eingabe- in eine Ausgabedatei. Im einzelnen wird hier eine Blockgröße (blocksize, bs) von 1024 Bytes festgelegt. Dann werden Eingabe- und Ausgabedatei benannt, dabei wird als Eingabedatei `$0` und somit der Name des laufenden Skriptes benutzt. Schließlich wird festgelegt, daß bei der Aktion die ersten sechs Block – also die ersten sechs Kilobytes – übersprungen werden sollen.

Um es nochmal zu betonen: Diese beiden Methoden sind mit Vorsicht zu genießen. Bei der ersten führt jede zusätzliche oder gelöschte Zeile zu einer kaputten Ausgabedatei, bei der zweiten reichen schon einzelne Zeichen. In jedem Fall sollte nach dem Auspacken noch einmal mittels `sum` oder `md5sum` eine Checksumme gezogen und verglichen werden.

B.3 Dateien, die es nicht gibt

Eine Eigenart der Behandlung von Dateien unter Unix besteht im Verhalten beim Löschen. Gelöscht wird nämlich zunächst nur der Verzeichniseintrag. Der Inode, also die Markierung im Dateisystem unter der die Datei gefunden werden kann, besteht weiterhin. Er wird erst dann freigegeben, wenn er nicht mehr referenziert wird. Physikalisch besteht die Datei also noch, sie wird lediglich im Verzeichnis nicht mehr angezeigt.

Hat ein Prozeß die Datei noch in irgendeiner Form geöffnet, kann er weiter darauf zugreifen. Erst wenn er die Datei schließt ist sie tatsächlich und unwiederbringlich ‘weg’.

Dieser Effekt der ‘nicht existenten Dateien’ läßt sich an verschiedenen Stellen geschickt einsetzen.

B.3.1 Daten aus einer Subshell hochreichen

Ein immer wieder auftretendes und oft sehr verwirrendes Problem ist, daß Variablen die in einer Subshell definiert wurden außerhalb dieser nicht sichtbar sind (siehe Abschnitt A.2). Dies ist um so ärgerlicher, als daß Subshells auch bei vergleichsweise einfachen Pipelines geöffnet werden.

Ein Beispiel für ein mißlingendes Skriptfragment wäre das folgende:

```
nCounter=0
cat datei.txt | while read VAR; do
  nCounter=‘expr $nCounter + 1‘
done
echo "nCounter=$nCounter"
```

Die Variable nCounter wird mit 0 initialisiert. Dann wird eine Datei per Pipe in eine `while`-Schleife geleitet. Innerhalb der Schleife wird für jede eingehende Zeile die Variable hochgezählt. Am Ende der Schleife enthält die Variable tatsächlich den korrekten Wert, aber da die Pipe eine Subshell geöffnet hat ist der Wert nach Beendigung der Schleife nicht mehr sichtbar. Das `echo`-Kommando gibt die Zahl 0 aus.

Es gibt mehrere Ansätze, diesem Problem zu begegnen. Am einfachsten wäre es in diesem Fall, dem Rat aus Abschnitt A.2 zu folgen und die Subshell geschickt zu vermeiden. Doch das ist leider nicht immer möglich. Wie geht man in solchen Fällen vor?

Bei einfachen Zahlenwerten könnte beispielsweise ein Rückgabewert helfen. Komplexere Informationen können in eine temporäre Datei geschrieben werden, die danach geparst werden müßte. Wenn die Informationen in Zeilen der Form `VARIABLE="Wert"` gespeichert werden, kann die Datei einfach mittels `source` (Abschnitt 4.13.4) oder einem Konstrukt der folgenden Art gelesen werden:

```
eval ‘cat tempfile‘
```

Und genau mit dieser Überlegung kommen wir zu einem eleganten – wenn auch nicht ganz einfachen – Trick.

Anstatt die Daten in eine temporäre Datei zu schreiben, wo sie womöglich durch andere Prozesse verändert oder ausgelesen werden könnten, kann man sie auch in ‘nicht existente’ Dateien schreiben. Das folgende Beispiel demonstriert das Verfahren:

```
#!/bin/sh -x
TMPNAME="/tmp/‘date ’+%Y%m%d%H%M%S’‘$$.txt"
exec 3> "$TMPNAME"
exec 4< "$TMPNAME"
rm -f "$TMPNAME"
```

Bis hierher wurde zunächst eine temporäre Datei angelegt. Die Filehandles 3 und 4 wurden zum Schreiben bzw. Lesen mit dieser Datei verbunden. Daraufhin wurde die Datei entfernt. Die Filehandles verweisen weiterhin auf die Datei, obwohl sie im Dateisystem nicht mehr sichtbar ist.

Kommen wir zum nützlichen Teil des Skriptes:

```
nCounter=0
cat datei.txt | ( while read VAR; do
  while read VAR; do
    nCounter=‘expr $nCounter + 1‘
  done
  echo "nCounter=$nCounter"
) >&3
```

Hier wurde wieder die Variable nCounter initialisiert und in der Subshell die Zeilen gezählt wie im ersten Beispiel. Allerdings wurde explizit eine Subshell um die Schleife gestartet. Dadurch steht die in der Schleife hochgezählte Variable auch nach Beendigung der Schleife zur Verfügung, allerdings immernoch nur in der Subshell. Um das zu ändern, wird in Zeile 11 der Wert ausgegeben. Die Ausgaben der Subshell werden in den oben erstellen Deskriptor umgeleitet.

```
echo "(vor eval)  nCounter=$nCounter"
eval ‘cat <&4‘
echo "(nach eval) nCounter=$nCounter"
```

Das `echo`-Kommando in Zeile 13 beweist, daß der Wert von nCounter tatsächlich außerhalb der Subshell nicht zur Verfügung steht. Zunächst.

In Zeile 14 wird dann die ebenfalls oben schon angesprochene `eval`-Zeile benutzt, um die Informationen aus dem Filedeskriptor zu lesen, die die Schleife dort hinterlassen hat.

Abschließend zeigt die Zeile 15, daß der Transport tatsächlich funktioniert hat, die Variable nCounter ist mit dem Wert aus der Subshell belegt.

B.3.2 Dateien gleichzeitig lesen und schreiben

Es kommt vor, daß man eine Datei bearbeiten möchte, die hinterher aber wieder unter dem gleichen Namen zur Verfügung stehen soll. Beispielsweise sollen alle Zeilen aus einer Datei entfernt werden, die nicht das Wort *wichtig* enthalten.

Der erste Versuch an der Stelle wird etwas in der Form

```
grep wichtig datei.txt > datei.txt
```

sein. Das kann funktionieren, es kann aber auch in die sprichwörtliche Hose gehen. Das Problem an der Stelle ist, daß die Datei an der Stelle gleichzeitig zum Lesen und zum Schreiben geöffnet wird. Das Ergebnis ist undefiniert.

Eine Elegante Lösung besteht darin, einen Filedeskriptor auf die Quelldatei zu legen und diese dann zu löschen. Die Datei wird erst dann wirklich aus dem Dateisystem entfernt, wenn kein Deskriptor mehr auf sie zeigt. Dann kann aus dem gerade angelegten Deskriptor gelesen werden, während eine neue Datei unter dem alten Namen angelegt wird:

```
#!/bin/sh
FILE=datei.txt
exec 3< "$FILE"
rm "$FILE"
grep "wichtig" <&3 > "$FILE"
```

Allerdings sollte man bei dieser Methode beachten, daß man im Falle eines Fehlers die Quelldaten verliert, da die Datei ja bereits gelöscht wurde.

B.4 Auf der Lauer: Wachhunde

Es kommt vor, daß man einen Prozeß startet, bei dem man sich nicht sicher sein kann daß er sich auch in absehbarer Zeit wieder beendet. Beispielsweise kann der Timeout für einen Netzwerkzugriff deutlich höher liegen als erwünscht, und wenn der ‘gegnerische’ Dienst nicht antwortet bleibt einem nur zu warten.

Es sei denn, man legt einen geeigneten Wachhund[3] auf die Lauer, der im Notfall rettend eingreift. In einem Shell-Skript könnte das wie folgt aussehen:

```
#!/bin/sh
timeout=5
ping 192.168.0.254 &
cmdpid=$!
```

Bis hierher nichts aufregendes. Eine Variable wird mit dem Timeout belegt, also mit der Anzahl an Sekunden nach denen der zu überwachende Prozeß unterbrochen werden soll. Dann wird der zu überwachende Prozeß gestartet und mittels & in den Hintergrund geschickt. Die Prozeß-ID des Prozesses wird in der Variablen cmdpid gesichert.

```
(sleep $timeout; kill -9 $cmdpid) &
watchdogpid=$!
```

In Zeile 5 findet sich der eigentliche Watchdog. Hier wird eine Subshell gestartet, in der zunächst der oben eingestellte Timeout abgewartet und dann der zu überwachende Prozeß getötet wird. Diese Subshell wird ebenfalls mit & in den Hintergrund geschickt. Die ID der Subshell wird in der Variablen watchdogpid gesichert.

```
wait $cmdpid
kill $watchdogpid > /dev/null 2>&1
exit 0
```

Dann wird durch ein `wait` darauf gewartet, daß sich der überwachte Prozeß beendet. Dabei würde `wait` bis in alle Ewigkeit warten, wäre da nicht der Watchdog in der Subshell. Wenn dem die Ausführung zu lange dauert, sorgt er dafür daß der Prozeß beendet wird.

Kommt der überwachte Prozeß aber rechtzeitig zurück, sorgt `kill` in Zeile 8 dafür daß der Wachhund ‘eingeschläfert’ wird.

Auf diese Weise ist sichergestellt, daß der `ping` auf keinen Fall länger als fünf Sekunden läuft.

[3] Der englische Begriff ‘Watchdog’ ist in diesem Zusammenhang wahrscheinlich geläufiger...

C Quellen

- Bash Reference Manual (http://www.gnu.org/manual/bash-2.02/bashref.html)
- Die Man-Page der Bash
- Die deutschsprachige Shell-Newsgroup (news://de.comp.os.unix.shell)
- Unix In A Nutshell (http://www.oreilly.com/catalog/unixnut3/)
- Unix Power Tools (http://www.oreilly.com/catalog/upt2/)
- Von DOS nach Linux HOWTO (http://www.linuxhaven.de/dlhp/HOWTO/DE-DOS-nach-Linux-HOWTO.html)
- Bash Guide for Beginners (http://tldp.org/LDP/Bash-Beginners-Guide/)
- Advanced Bash-Scripting Guide (http://tldp.org/LDP/abs/)
- The Open Group Base Specifications
- Single Unix Specifications V2
- ... und eine Menge Skripte, die ich im Laufe der Zeit gelesen habe (das kann ich nur jedem empfehlen - es ist spannender als es sich anhört...).

D Routenplanung - Die Zukunft dieses Dokuments

Ich werde in dieser Version des Dokumentes nicht alles umgesetzt haben, was mir sinnvoll erscheint. An dieser Stelle sammele ich Ideen für eine nächste Version. Kommentare dazu nehme ich jederzeit dankend entgegen.

- **GNU / Posix:** Bessere Abgrenzung der GNU-Erweiterungen gegenüber dem Posix-Standard, damit die Skripte portabel bleiben und nicht nur auf Linux laufen.
- **Performance:** Ein Kapitel mit Hinweisen, wie Skripte nicht unnötig langsam werden.
- **Interaktive Shell:** Exkurs zur interaktiven Benutzung der Bash. Es gibt da eine Reihe hilfreicher Tastenkombinationen, die längst nicht jedem bekannt sind.
- **HTML-Version:** Ich würde gerne eine HTML-Version zur Verfügung stellen, bin mir aber noch nicht sicher ob das aus den LaTeX-Quellen geht. Mir liegt dabei die Indizierung der Stichworte am Herzen, die sollte nicht verloren gehen.
- **Glossar:** Verzeichnis von Begriffen wie PID, Prompt, GID, UID, Cron, usw.

Index