jetzt lerne ich

Shell-programmierung

PEARSON Education

Markt+Technik

jetzt lerne ich

Shell-programmierung

Effektiv auf der Linux-/Unix-Kommandozeile arbeiten

BETTINA RATHMANN CHRISTA WIESKOTTEN

Markt+Technik

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

10 9 8 7 6 5 4 3 2 1

06 05 04

ISBN 3-8272-6754-4

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Coverkonzept: independent Medien-Design
Coverlayout: adesso 21, Thomas Arlt
Titelillustration: Karin Drexler
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Bosch, Ergolding
Printed in Germany

Übersicht

»A programmer is just a tool which converts caffeine into code«

(anonym)

Im Sinne dieses bekannten und vielsagenden Zitats widmen Ihnen die Autoren und Lektoren der Buchreihe »Jetzt lerne ich« in jeder Ausgabe ein Rezept mit oder rund um das belebende und beliebte Getränk. Sollten Sie gerade ohne Bohnen oder Pulver sein: Über die Adresse *http://www.kaffee.mut.de* können Sie einen eigens entwickelten Markt+Technik Programmiererkaffee bestellen.

Viel Spaß und Genuß!

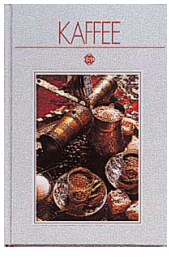
Indischer Kaffeereis

225 g Reis
½ l Milch
60 g Zucker
2 Eigelb
1 Tasse starker Kaffee
1 Likörglas Rum

Den Reis waschen und 3 Minuten in einem großen Topf in Wasser kochen lassen, daneben die Milch zum Kochen bringen. Dann den Reis abtropfen lassen und in die kochende Milch geben. Leicht kochen lassen über ca. 10 Minuten; die Milch darf dabei natürlich nicht überlaufen. Den Topf vom Feuer nehmen, den Zucker, den Kaffee, den Rum und anschließend die Eigelb unter ständigem Rühren zugeben. Den Topf wieder aufs Feuer setzen, 2 Minuten erhitzen, aber nicht mehr zum Kochen bringen. Mit Schlagsahne verziert kalt servieren.

Reis ist das wichtigste landwirtschaftliche Produkt Indiens. Das Land kennt viele Reisgerichte, meist in scharf gewürzter Zubereitung. Da auch Kaffee im Südwesten Indiens in beträchtlichen Mengen kultiviert wird, war es nicht schwer, ein Reisgericht, mit Kaffee zubereitet, zu finden. Der Rum, aus dem in Indien ebenfalls in großen Mengen angebauten Zuckerrohr gewonnen, steuert eine pikante Note bei.

Das Kaffeerezept wurde entnommen aus:



»Kaffee«
Dr. Eugen C. Bürgin
Sigloch Edition, Blaufelden
ISBN: 3-89393-135-X

Inhaltsverzeichnis

Vorwort

Wenn Sie dieses Buch in den Händen halten, fangen Sie mit dem Teil des Buchs an, den wir als Letztes vollendet haben. Das heißt, sämtliche Kapitel und Anhänge sind fertig, viele Klippen umschifft und Fehler ausgebügelt, und nun kommt das unlösbare Problem: Was schreiben wir ins Vorwort?

Wir könnten Sie mit Aussagen langweilen, warum Sie eine gute Wahl getroffen haben, als Sie dieses Buch aus dem Regal gegriffen haben, oder wie toll unsere Skripten sind. Die Wahrheit aber ist: Das müssen Sie selbst entscheiden. Wir können nur unserer Hoffnung Ausdruck verleihen, dass Ihnen das Buch gefällt und Sie etwas daraus mitnehmen können.

Gut, aber damit ist unser Problem immer noch nicht gelöst (wir haben immerhin schon zwei Absätze fertig `:)`). Vielleicht interessiert es ja, wie dieses Buch zustande kam. Da wir zwei Autorinnen haben, wollen wir dies einmal aus zwei verschiedenen Sichtweisen dokumentieren. Tatsächlich geht die Entstehung weiter zurück ...

Bettina

Wir schreiben das Jahr 1993. Ein damaliger Arbeitskollege gibt mir eine CD mit den Worten: »Das ist ein Freeware-Unix, damit kannst du unseren Basicinterpreter laufen lassen. Es ist dabei wesentlich schneller als SCO.« Aha, kostet nichts, schneller als SCO und auch noch kompatibel? Das kann nur ein Scherz auf meine Kosten sein. Also installieren, um diese Aussage zu untermauern. Drei Wochen später war ich überzeugt. Seit diesem Zeitpunkt ist auf allen meinen Privatrechnern ein Linux installiert.

Meine ersten Gehversuche waren Shellskripten, die ich bei der Arbeit einsetzen konnte. Über die Jahre hinweg kamen und gingen die Arbeitgeber, bis ich schließlich in einer Firma in Essen landete …

Christa

… wo wir mittlerweile das Jahr 1995 schreiben. Bis dahin hatte ich die Mathematik im Sinn. Computer und Betriebssystem: ein Buch mit sieben Siegeln. Programmieren ja, aber es musste schon mit Numerik zu tun haben. Alles andere, wozu braucht man das?

Da wurde ich ganz schnell eines Besseren belehrt: »Schreib mal ein paar CGI-Skripten für unseren Webserver.« Und als ob das nicht schon genug war, wurde ich auch in die Programmierung eines Suchdienstes für das Internet integriert. Da war guter Rat teuer.

Aber da war ja noch die neue Kollegin, die mich immer mit Tee belästigte. Hatte die nicht Erfahrung mit Shellprogrammierung?

Bettina

Wie, sie will meinen Tee? Da steckt doch was dahinter!? Eine Einführung in Shellprogrammierung? Nun ja, ich bin ja kein Unmensch. Also in der spärlichen Freizeit noch ein paar Aufgaben zusammengetippt und im Wochenrhythmus an Christa weitergegeben. Über den Lauf eines Jahres hinweg hatten wir die wichtigsten Aspekte der Shellskripturierung abgearbeitet, und das Beste: Ihre Skripten liefen auch auf der Arbeit fehlerfrei!

Christa

Bevor ich allerdings sämtliche Mysterien der Skriptprogrammierung von Bettina gelernt hatte, trennten sich unsere beruflichen Wege, und in meiner neuen Stelle war Shellprogrammierung nicht gefragt. Privat allerdings blieben wir weiter in Kontakt, und das zweite gemeinsame Lernprojekt (Unix-C-Programmierung) lief weiter …

Bettina

Ende 1998 waren wir dort beim Thema Socketprogrammierung angelangt, worauf ich scherzhaft sagte: »Christa, wir haben jetzt so viel gelernt, darüber könnten wir ein Buch schreiben.«

Christa

»Ja, machen wir!«

Bettina

Ich und mein vorlautes Mundwerk: Manchmal gehen meine Scherze echt nach hinten los, und nun sitze ich hier beim Vorwort, und Christa grinst mich diabolisch an.

Essen im Mai 1999

Christa Wieskotten und *Bettina Rathmann*

Rückblende, kurz vor dem Abgabetermin der Neuauflage:

Bettina

Perfekt! Ich bin fertig und Christa muß nur noch 300 Seiten umformatieren. Pech aber auch, dass ich kein Winword habe ... `:o]`

Christa

grummel
Vor dem nächsten Buch schenke ich Bettina Winword. So was gemeines. Oh Gott, schon 2:45 morgens ... Hm, fehlt da nicht was? Oh, die Änderungen zur 2. Auflage hat Bettina vergessen.

greift zum Telefon

Bettina

ring

Was, Christa, oh, hi *gähn* ... Was, eine Änderungsübersicht zur 2. Auflage um 2:50 Uhr??? Was, auch noch sofort??? *grummel* Na gut, lass mal sehen, was wir für die zweite Auflage so alles Neues eingebracht haben.

- Einige Fehler und Unklarheiten behoben, alles auf den Stand der neuesten Versionen von `Bash` und `PDKSH` gebracht
- Kapitel 7 um `FUNCNAME` erweitert
- Kapitel 10 SED deutlich erweitert
- Kapitel 11 um `xargs` erweitert
- Kapitel 12 Portabilität erweitert
- Kapitel 13: Mehr Tipps zur Fehlersuche
- Anhang A um die Änderungen von `Bash` und `PDKSH` ergänzt
- Anhang B und D leicht erweitert und aktualisiert
- plus mehr Beispiele, Aufgaben und Hinweise

Christa

Endlich fertig! Wie ist es eigentlich mit einer Belohnung, Bettina?

Bettina

Warte doch erst einmal ab, ob unser Buch überhaupt bei den Lesern ankommt.

Christa

O.K. Sagen wir mal, wenn unser Buch in die Top 1000 von *Amerson.de* kommt, dann steht uns eine Belohnung zu.

Bettina

Bevor das Buch in die Top 1000 kommt, sind wir eher bis Australien und zurück gereist.

Christa

grins

Bettina

Oh je. Wieder ein Fehler …

Woche 1: Kein Eintrag. Unser Buch taucht überhaupt nicht auf.

Woche 2: Immer noch kein Eintrag zu finden. (Ich wusste es, kein Grund zur Sorge.)

Woche 3: *Amerson* kennt das Buch mittlerweile.

Woche 4: Ein Rang unter den ersten 10.000. (Christa grinst schon wieder.)

[…]

Woche 8: Ein Australienkatalog in meinem Briefkasten. Oh, oh, Amerson bestätigt einen Platz unter den ersten 1000.

Zwei Jahre und eine tolle Reise weiter ein Update. Wie wäre es mit noch einer Wette?

Bettina

grins

Essen im November 2003

Christa Wieskotten und *Bettina Rathmann*

Einleitung

»Der Optimist behauptet, dass wir in der besten aller möglichen Welten leben, der Pessimist befürchtet, dass das stimmt.«

Egal ob Sie nun zu den Pessimisten oder den Optimisten gehören: Wenn Sie dieses Buch in den Händen halten, haben Sie bereits die ersten Unixhürden genommen:

- Sie haben sich für Unix, vielleicht sogar in Gestalt von Linux entschieden.
- Sie haben Unix (Linux) installiert, und es läuft.

Das ist erfreulich und dennoch irgendwie unbefriedigend, schließlich wollen Sie Ihr System auch nutzen.

Stellt sich die Frage: Wie nutze ich mein Unix? Auf diese Antwort gibt es so viele Antworten, wie es Unixanwender gibt (lt. Red Hat gab es Ende 1998 allein ca. 12 bis 15 Millionen Linuxanwender weltweit). Selbst wenn Sie sich zur Programmierung durchgerungen haben, ist die Auswahl schier erdrückend, die Anzahl an HOWTOs erschreckend, C-Programmierung für den Anfang eine zu hohe Hürde und vielleicht für die ersten Schritte die berühmte Kanone, die auf die armen Spatzen schießt. Und die anderen Programmiersprachen sind eh ein Buch mit sieben Siegeln.

Kein Grund, die Flinte ins Korn zu werfen. Dieses Buch soll Ihnen einen möglichen Ausweg aufzeigen: die Shellprogrammierung mit Hilfe der Bash-Shell. Diese bietet gleich mehrere Vorteile:

- Sie ähnelt auf den ersten Blick Ihrer bekannten DOS-Umgebung, ist dabei jedoch ungleich komfortabler und mächtiger.

- Sie lernen wichtige Befehle, die Sie im täglichen Gebrauch unter Unix benötigen werden.
- Sie lernen, wie aus vielen kleinen Programmen ein neues entsteht, das Ihren Bedürfnissen Rechnung trägt.
- Echte Unixgurus streben danach, ihre Probleme durch Skripten und nicht durch C-Programme zu lösen, um dann einem C-Programmierer zu erklären, dass sein Programm total überflüssig ist, weil das Problem durch ein Skript schneller zu lösen wäre ;o)
- Wenn dieses Buch auch meistens auf Linux abzielt: Bash-Skripten sind nicht auf Linux beschränkt, sondern laufen auf fast allen Unices, mit denen wir gearbeitet haben.

Aber selbst wenn Sie kein Linux nutzen, sondern ein anderes Betriebssystem (andere Unixversionen, Mac OS X oder gar *hust* Windows *hust*) und evtl. auch noch eine andere Shell, wie z.B. die Bourneshell `sh` oder die Kornshell `ksh`, kann Ihnen dieses Buch dienlich sein. In Kapitel 12 werden wir uns noch einmal ausführlich mit den Unterschieden zwischen `bash` und `ksh` auseinander setzen, und glauben Sie mir, diese sind nicht groß, aber fein.

Das alles schreckt Sie nicht ab? Dann lassen Sie mich noch ein paar einleitende Worte verlieren, bevor wir endgültig loslegen.

Voraussetzungen

Sie brauchen einen Rechner mit Linux, einem anderen Unix oder Mac OS X und der Shell Ihrer Wahl. Außerdem werden Sie ohne Programme wie `cp`, `mv` oder `rm` nicht sehr weit kommen. Außerdem sollten Sie einen Texteditor Ihrer Wahl beherrschen und einige grundlegende Unixkenntnisse haben (wie meldet man sich an, was sind Dateiberechtigungen, Verzeichnisse etc.).

Zielsetzung

Dieses Buch soll Ihnen über die ersten Hürden der Shellprogrammierung hinweghelfen. Dabei steht weniger die Theorie als die praktische Anwendung im Vordergrund. Erwarten Sie jetzt nicht, dass Sie dieses Buch ohne eine Zeile Theorie durcharbeiten können, die wird sich aber auf die absolut notwendigen Bereiche beschränken.

Das *Durcharbeiten* ist übrigens wörtlich gemeint: Nehmen Sie das Buch, lesen Sie ein Kapitel, und probieren Sie die Beispiele aus. Diese sollten Ihnen im täglichen Umgang mit der Shell einige Arbeit abnehmen. Wenn Sie dabei feststellen, dass etwas fehlt, was Sie dringend brauchen, umso besser: Nutzen Sie Ihr frisch erworbenes Wissen, und erweitern Sie das Skript, denn nur

Übung macht den Meister. Die im Buch besprochenen Skripten können Sie auf der Markt+Technik-Website unter *www.mut.de* herunterladen. Geben Sie unter SUCHE: einfach den Buchtitel ein, um auf die Katalogseite zu diesem Buch zu gelangen. Dort können Sie sich das Archiv über einen Link herunter laden.

Ein letzter Tipp: Fast alle Befehle, mit denen Sie sich in diesem Buch herumquälen müssen, haben mehr Optionen, als im Rahmen dieses Buches aufgeführt werden können. Der `man`-Befehl (gefolgt vom Namen des Befehls, der Ihnen Rätsel aufgibt) erläutert ausführlich den entsprechenden Befehl.

```
buch@koala:/home/buch > man ls
```

gibt also Informationen über den `ls`-Befehl aus. Sie können mit den Pfeiltasten durch den Text blättern, und mit [q] beendet man `man`.

Häufig finden Sie auch Referenzen auf Manpages, die von einer Nummer gefolgt werden: *terminfo(5)*. Die Manpages sind in neun Kategorien unterteilt. Diese Abschnitte decken bestimmte Themenbereiche ab (Tabelle E.1).

Abschnitt	Bedeutung	Beispiel
1	Programme oder Shellbefehle	ls(1), man(1)
2	Systemaufrufe	pause(2), wait(2)
3	Systembibliotheken	strncat(3), getwd(3)
4	Besondere Dateien, z.B. die aus `/dev/`	null(4), zero(4)
5	Dateiformate	fstab(5)
6	Spiele	fortune(6)
7	Makropakete und Vorgaben	ascii(7), man(7)
8	Verwaltungsbefehle, meist für `root`	nslookup(8), inetd(8)
9	Unter Linux: Kernelroutinen	intro(9)

Tabelle E.1: Die Systematisierung der Manpages

`man` findet immer den ersten Eintrag zu einem übergebenen Stichwort. Wenn Sie z.B. auf `man(7)` zugreifen wollen, müssen Sie den Abschnitt angeben: `man 7 man`.

Tipps zur Notation

Wenn Eingaben über die Tastatur beschrieben werden, dann werden wir Ihnen das durch ein Tastenzeichen kennzeichnen, z.B. [Esc] für das Drücken der Escape-Taste oder [Strg]+[C] für das gleichzeitige Drücken zweier Tasten (hier: Steuerung und der Buchstabe c).

Wenn Sie Teile eines Befehls in eckigen Klammern sehen, beispielsweise

```
man <befehl>
```

ersetzen Sie bitte den eingeklammerten Teil durch Ihre Angaben, also z.B. durch

```
man man
```

Die Eingabeaufforderung auf unserem Testsystem (den Benutzer `buch` haben wir ausschließlich für Texte und Skripten zu diesem Buch angelegt) wird immer

```
buch@koala:/home/buch >
```

sein. Auf Ihrem System werden Sie natürlich Ihren eigenen Prompt sehen.

Befehle im Text sind durch eine andere Schrift gekennzeichnet: z.B. `man`.

Tipps zur Handhabung

Wenn etwas schief geht und die Shell nicht so reagiert, wie das Buch es beschreibt: keine Bange. In der Regel haben Sie sich vertippt, was die Shell veranlasst, entweder Fehlermeldungen auszugeben oder weitere Eingaben anzufordern. Ist Letzteres der Fall und Sie wollen die Eingabe wiederholen (ohne Fehler natürlich `:)`), dann drücken Sie Strg+C oder Strg+Pause, und die Shell sollte wieder mit dem gewohnten Prompt reagieren.

Melden Sie sich an, und legen Sie in Ihrem Heimatverzeichnis ein Unterverzeichnis an, in dem Sie unsere Skripten abspeichern und ausprobieren.

```
ihr_prompt:~ > mkdir Skripten
ihr_prompt:~ > cd Skripten
ihr_prompt:~/Skripten >
```

Wie bereits erwähnt, haben wir Autoren auf unserem Testsystem eine Anmeldung und ein Heimatverzeichnis nur für dieses Buch angelegt, Ihre Gegebenheiten dürfen ruhig davon abweichen.

Bevor wir anfangen, noch ein paar allgemeine Tipps zur Handhabung der Shell. Sowohl Kornshell als auch Bash bieten einige Eingabehilfen an, die das Arbeiten mit den Shells sehr komfortabel gestalten. Wenn Sie mit der Bedienung Ihrer Shell vertraut sind, so können Sie gleich mit Kapitel 1 anfangen, ansonsten hier ein paar Tipps. Die Lektüre der Manpages werden sie Ihnen allerdings nicht ersparen können, allein aus der Konfiguration und Bedienung der Bash könnte man leicht ein eigenes Buch erstellen. Aus diesem Grunde beschreiben wir an dieser Stelle die Ausgangskonfiguration unter Linux mit der pdksh 5.2 und der Bash 2.02.

Die Bash

Eine der komfortabelsten Möglichkeiten der Bash (Bash steht für »Bourne Again Shell«) ist das automatische Vervollständigen, welches durch das Drücken von ⇆ (Strg+I) aktiviert wird. Die Bash versucht dabei, den eingegebenen Text als Variablennamen zu interpretieren, falls der Text mit »$« beginnt. Ein »~« am Anfang führt dazu, dass die Bash den Text als Benutzernamen interpretiert, und ein »@« führt zu einer Rechnernamenersetzung. Führen diese Methoden nicht zum Ziel, so versucht die Bash eine Dateinamenersetzung. In diesem Fall wird die Variable `PATH` berücksichtigt und das aktuelle Verzeichnis.

Kann der Name wegen mehrerer Alternativen nicht vervollständigt werden, so wird so weit vervollständigt, wie es möglich ist, und die Bash hupt. Ein erneutes Drücken von zweimal ⇆ listet alle Möglichkeiten auf.

```
buch@koala:/home/buch > echo $PR       # An dieser Stelle TAB drücken
$PRINTER         $PROFILEREAD     $PROMPT_COMMAND
buch@koala:/home/buch > echo $PROM     # und noch einmal TAB gedrückt führt zu
buch@koala:/home/buch > echo $PROMPT_COMMAND
buch@koala:/home/buch > ls k           # TAB ergibt
buch@koala:/home/buch > ls kapitel     # jetzt 2x TAB
kapitel1.txt    kapitel12.txt~  kapitel5.txt    kapitel8.txt~
kapitel10.txt   kapitel13.txt   kapitel5.txt~   kapitel9.txt
kapitel10.txt~  kapitel13.txt~  kapitel6.txt    kapitel9.txt~
kapitel11.txt   kapitel2.txt    kapitel7.txt
kapitel11.txt~  kapitel3.txt    kapitel7.txt~
kapitel12.txt   kapitel4.txt    kapitel8.txt
buch@koala:/home/buch > ls kapitel
buch@koala:/home/buch > ca             # 2x TAB
cal        callbootd  captoinfo  case       cat        catman
buch@koala:/home/buch > ca
```

Stört Sie das Tuten der Shell, wenn sie Ihnen keinen eindeutigen Vorschlag machen kann? Dann tragen Sie in die Datei `.inputrc` in Ihrem Heimatverzeichnis folgende Zeile ein:

```
set bell-style none
```

Sie haben einen Text, den Sie häufiger eintippen müssen? In diesem Falle erweist sich die Verlaufsfunktion (engl. *History*) als nützlich, die sich die letzten eingegebenen Befehlszeilen merkt. Mit den Cursortasten können Sie die Eingaben zeilenweise durchblättern, bei Bedarf editieren und durch ↵ erneut eingeben bzw. ausführen lassen.

Das reicht Ihnen nicht, weil Sie mehr als eine Zeile eingeben müssen? Dann gibt es noch Tastaturmakros, die genau das wiedergeben, was Sie eingegeben haben.

Aufnehmen können Sie ein Makro, wenn Sie Strg+X eingeben. Beendet wird die Aufnahme des Makros durch Strg+X. Danach können Sie mit Strg+X e das so aufgenommene Makro ausführen.

Wenn Sie wissen wollen, wie die Shell Ihre Eingaben wirklich interpretiert und was somit wirklich ausgeführt wird, so hilft Esc+Strg+e:

```
buch@koala:/home/buch > cmd="pwd"
buch@koala:/home/buch > $cmd    # jetzt ESC+CTRL+e drücken
buch@koala:/home/buch > pwd     # Ergebnis
```

Tabelle E.2 informiert Sie über einige der verfügbaren Editierbefehle, die sich am Emacs-Editor orientieren.

Tabelle E.2: Editierbefehle im Emacs-Modus

Tastenkombination	Aktion
Strg+A/Pos 1	An den Anfang der aktuellen Zeile springen
Strg+E/Ende	Ans Ende der aktuellen Zeile springen
Strg+F/→	Cursor ein Zeichen nach rechts
Strg+B/←	Cursor ein Zeichen nach links
Strg+H/←	Zeichen links vom Cursor löschen
Strg+D/Entf	Zeichen unter dem Cursor löschen
	Vorsicht: Wenn die Zeile leer ist und der Cursor in der ersten Spalte nach dem Prompt steht, so führt Strg+D zur Abmeldung.
Strg+U	Alle Zeichen links vom Cursor löschen
Strg+W	Wort links vom Cursor löschen. Wörter werden hierbei durch Leerzeichen oder Tabulatoren getrennt.
Esc+F	Cursor ein Wort nach rechts
Esc+b	Cursor ein Wort nach links
Esc+U	Aktuelles Wort in Großschrift umwandeln
Esc+L	Aktuelles Wort in Kleinschrift wandeln
Esc+←	Wort links vom Cursor löschen. Ein Wort besteht hierbei aus Ziffern oder Buchstaben.
Strg+K	Alles rechts vom Cursor löschen
↑/Strg+P	Einen Eintrag im Verlauf zurück und diesen zur Bearbeitung bereitstellen
↓/Strg+N	Den nächsten Eintrag im Verlauf ausgeben und zur Bearbeitung bereitstellen.
Strg+R	Automatische Ergänzung der Eingabe, ausgehend vom Verlauf bzw. Historie der Eingabe

Wenn Sie diese Funktionen anderen Tasten zuordnen wollen, so benötigen Sie dazu Einträge in die Datei `.inputrc` in Ihrem Heimatverzeichnis, oder Sie nutzen den `bind`-Befehl. Genauere Informationen dazu finden Sie in der Manualpage unter dem Stichwort »*Readline Key Bindings*«.

Wenn Sie das letzte Wort (Wörter sind Kombinationen aus Ziffern und Buchstaben) aus der vorherigen Eingabe direkt wieder eingeben wollen, so hilft Ihnen Esc+.:

```
buch@koala:/home/buch > cp Datei /home/buch/skript/test/     # Normale Eingabe
buch@koala:/home/buch > ls -l                                 # <ESC>+<.>
buch@koala:/home/buch > ls -l /home/buch/skript/test/
```

Wenn Sie ein bestimmtes Wort aus der vorherigen Eingabe benötigen, so geben Sie Esc+n r Esc+. ein. Dabei ist `nr` die Nummer des Wortes, das Sie übernehmen wollen: Das erste Wort ist 0, das zweite 1 usw. Warum das so ist, werden wir in diesem Buch auch noch lernen.

```
buch@koala:/home/buch > echo a b c d e f g h i j k l m n   # Vorherige Eingabe
a b c d e f g h i j k l m n
buch@koala:/home/buch > echo       # <ESC>+<10>
(arg: 10) echo                     # <ESC>+<.>
buch@koala:/home/buch > echo j
```

Falls Sie die Wörter nicht von links, sondern von rechts zählen wollen, so müssen Sie einen negativen Wert eingeben. Bei Esc+ - 1 Esc+. würde `m` übernommen, bei - 2 das `l` usw.

Die Kornshell

Die Kornshell bietet ähnliche Editierungsmöglichkeiten wie die Bash. Sie bietet allerdings zwei Eingabemodi an: den Vi-Modus und den Emacs-Modus. Der Vi-Modus orientiert sich am `vi`-Editor, der zwar nicht sehr komfortabel, aber doch sehr mächtig und vor allem auf allen Unixsystemen vorhanden ist. Dennoch wollen wir hier auf den Emacs-Modus schauen und vor allem die Unterschiede zur Bash ansprechen. Den Emacs-Modus aktivieren Sie durch die Eingabe von `set -o emacs`, während der Vi-Modus durch `set -o vi` aktiviert wird.

Der Emacs-Modus deckt sich zu einem großen Teil mit dem der Bash, was kein Wunder ist, da sich ja beide auf den gleichen Editor beziehen.

Tabelle E.3: Der Emacs-Modus der Kornshell

Tastenkombination	Aktion
Strg+A	An den Anfang der aktuellen Zeile springen
Strg+E	Ans Ende der aktuellen Zeile springen
Strg+F/→	Cursor ein Zeichen nach rechts
Strg+B/←	Cursor ein Zeichen nach links
Strg+H/←	Zeichen links vom Cursor löschen
Strg+D	Zeichen unter dem Cursor löschen
	Vorsicht: Wenn die Zeile leer ist und der Cursor in der ersten Spalte nach dem Prompt steht, so führt Strg+D zur Abmeldung.
Strg+U	Die gesamte Zeile löschen
Strg+G	Editierung abbrechen
Strg+W	Wort links vom Cursor löschen. Wörter werden hierbei durch Leerzeichen oder Tabulatoren getrennt.
Esc+F	Cursor ein Wort nach rechts
Esc+B	Cursor ein Wort nach links
Esc+U	Aktuelles Wort in Großschrift umwandeln
Esc+L	Aktuelles Wort in Kleinschrift wandeln
Esc+←	Wort links vom Cursor löschen. Ein Wort besteht hierbei aus Ziffern oder Buchstaben.
Strg+K	Alles rechts vom Cursor löschen
↑/Strg+P	Einen Eintrag im Verlauf zurück und diesen zur Bearbeitung bereitstellen
↓/Strg+N	Den nächsten Eintrag im Verlauf ausgeben und zur Bearbeitung bereitstellen

Die Kornshell erweitert nur Befehle und Dateinamen. Dazu dient die Tastenkombination Esc Esc. Wenn Sie sehen wollen, welche Erweiterungen nach der aktuellen Eingabe möglich sind, so drücken Sie Esc+?:

```
Kornshell:/home/buch > ls k            # Hier zweimal <ESC>
Kornshell:/home/buch > ls kapitel      # Soweit eindeutig, jetzt <ESC>+<?>
1) kapitel1.txt     7) kapitel12.txt~  13) kapitel5.txt    19) kapitel8.txt~
2) kapitel10.txt    8) kapitel13.txt   14) kapitel5.txt~   20) kapitel9.txt
3) kapitel10.txt~   9) kapitel13.txt~  15) kapitel6.txt    21) kapitel9.txt~
4) kapitel11.txt   10) kapitel2.txt    16) kapitel7.txt
5) kapitel11.txt~  11) kapitel3.txt    17) kapitel7.txt~
6) kapitel12.txt   12) kapitel4.txt    18) kapitel8.txt
Kornshell:/home/buch > ls kapitel
```

Wenn Sie nun alle möglichen Erweiterungen übernehmen wollen, so drücken Sie bitte die Tastenkombination Esc+*.

Ein Beispiel nach obigen Vorgaben:

```
Kornshell:/home/buch > ls kapitel7     # Jetzt <ESC>+<*>
Kornshell:/home/buch > ls kapitel7.txt kapitel7.txt~
```

Auch die Kornshell erlaubt das Kopieren der Wörter aus der vorherigen Eingabe mittels Esc+. Die Auswahl bestimmter Wörter ist ebenfalls mittels Esc+n r Esc+. möglich. Die Möglichkeit, die Wörter von rechts mit einem negativen Wert für `nr` anzusprechen, bietet die Kornshell allerdings nicht.

Weitere Informationen zur Tastaturbelegung finden Sie unter den Stichworten *emacs Editing Mode* und *vi Editing Mode* (SunOS 5.6) in der Manpage. Für die `pdksh` (»Public Domain Kornshell«) finden sich Informationen auch zur eigenen Belegung der Kommandos unter *Emacs Interactive Input Line Editing* bzw. *Vi Interactive Input Line Editing* (Linux).

Viele Manualpages sprechen von Metakey und Controlkey. Der Controlkey entspricht auf der PC-Tastatur in der Regel Ctrl/Strg und der Metakey Esc.

Strg muss dabei immer gleichzeitig mit der zweiten in der Kombination angegebenen Taste gedrückt werden. Beim Metakey reicht es aus, erst den Metakey zu drücken und dann die angegebene Taste.

Genug der Vorrede, es wird Zeit für Kapitel 1.

Kapitel 1

Grundlagen

»Wir dürfen jetzt den Sand nicht in den Kopf stecken« –
Lothar Matthäus

... vor allem nicht vor den Problemen, mit denen wir uns in diesem Buch beschäftigen wollen. Da noch kein Meister vom Himmel gefallen ist, soll dieses Kapitel die wichtigsten Grundlagen der Shellprogrammierung erklären und dabei auch ein wenig Theorie vermitteln. Schließlich brauchen Sie nicht nur praktische Erfahrung, sondern dürfen sich auch nicht durch Fachchinesisch beeindrucken lassen.

1.1 Was ist ein Shellskript?

Eine Shell ist ein Programm, das die Schnittstelle zwischen Ihnen und dem Betriebssystem bildet. Sie nimmt Eingaben von Ihnen entgegen, interpretiert sie, führt die gewünschten Aktionen mithilfe von Betriebssystem und Programmen aus. Die Ausgaben der Programme und einige wenige Ausgaben der Shell selbst werden als Ergebnis an Sie zurückgegeben. Die Eingaben müssen ein bestimmtes Format haben, damit die Shell sie korrekt erkennen und ausführen kann. Wenn Sie Daten eingeben dürfen, druckt die Shell ein Prompt aus und setzt den Cursor direkt dahinter. Ein Prompt ist eine Zeichenkette. Diese Zeichenkette könnte z.B. Informationen über Benutzer, Rechnernamen und das aktuelle Arbeitsverzeichnis ausgeben. Die wichtigste Aufgabe eines Prompts macht allerdings die deutsche Übersetzung klar: Eingabeaufforderung.

Ein Beispiel:


```
buch@koala:/home/buch > pwd
/home/buch
buch@koala:/home/buch >
```

`pwd` ist dabei eine typische Unixabkürzung und steht für *print working directory*, dieses Kommando gibt also das aktuelle Arbeitsverzeichnis aus.

Speichern Sie diese drei Buchdaten in einer Datei namens `skript1.sh` ab. Im Prinzip haben Sie jetzt Ihr erstes Skript geschrieben. Damit Sie Ihr Skript ausführen können, müssen Sie noch einige Vorbereitungen treffen. Denn zunächst wird Ihre Shell den Versuch, das neue Skript zu starten, mit einer Fehlermeldung beantworten. Rufen Sie Ihr Skript dazu einmal mit `./skript1.sh` auf.

```
bash: ./skript1.sh: Permission denied
```

Sollte sich Ihre Shell bei Ihnen mit dem Fehler

```
bash: skript1.sh: command not found
```

beschweren, so haben Sie Ihr Skript versehentlich mit `skript1.sh` statt mit `./skript1.sh` aufgerufen. Die Ursache des Fehlers wird uns in Kapitel 6 klar werden. Vorerst fügen Sie beim Aufruf eines Skripts immer `./` vor dem Dateinamen hinzu. Leider wird auch das so aufgerufene Skript einen `Permission denied`-Fehler ausgeben.

Um zu verstehen, warum dieser Fehler auftritt, müssen wir uns mit etwas Theorie herumschlagen. Unter Unix wird jede Datei mit Berechtigungen zum »lesen«, »schreiben« und »ausführen« ausgestattet. Dafür werden die Kürzel `r` (*read*), `w` (*write*) und `x` (*execute*) vergeben. Schauen Sie sich die Berechtigungen einmal mit dem `ls`-Befehl an, der eine ähnliche Aufgabe erfüllt wie `dir` unter MS-DOS.

```
buch@koala:/home/buch > ls -l skript1.sh
-rw-r--r--   1 buch     users         4 Jan 28 23:29 skript1.sh
```

Wenn Sie die zweite Spalte ignorieren, sehen Sie die Berechtigungen (`rw-r--r--`), den Eigentümer der Datei (`buch`), die Gruppe, der der Eigentümer angehört (`users`), die Größe der Datei in Bytes (`4`), das Datum der letzten Änderung (`Jan 28 23:29`) und den Dateinamen (`skript1.sh`).

Warum wiederholen sich die Berechtigungen nun dreimal? Unter Unix können die Berechtigungen für den Eigentümer (Stelle 2 bis 4, d.h. erstes `rw-`), für alle Gruppenmitglieder (Stelle 5-7) und alle anderen Personen (Stelle 8-10) unterschiedlich vergeben werden. Sollte es sich um ein Verzeichnis handeln,

so steht an Stelle 1 ein `d`. Um die Berechtigungen zu verändern, nutzen Sie den `chmod`-Befehl. Wir wollen den Eigentümern, den Gruppenmitgliedern und anderen Benutzern die Ausführungsberechtigung (Kürzel x) zuteilen:

```
buch@koala:/home/buch > chmod +x skript1.sh
buch@koala:/home/buch > ls -l skript1.sh
-rwxr-xr-x   1 buch     users         119 Jan 28 23:29 skript1.sh
buch@koala:/home/buch > ./skript1.sh
/home/buch
buch@koala:/home/buch >
```

Herzlichen Glückwunsch! Ihr erstes Shellskript funktioniert. Auf diesem Wissen können und werden wir aufbauen.

1.2 Kommentarzeilen

Nachdem Sie die erste Theoriehürde glänzend bewältigt haben, machen wir mit einem einfachen Thema, den Kommentaren, weiter. Kommentare werden eingesetzt, um Informationen zur Verwendung, zur Version etc. zu geben und gegebenenfalls die Parameter des Skripts zu verdeutlichen. Sie werden für jede Information dankbar sein, wenn Sie sich durch ein Skript wühlen müssen, das Sie vor Jahren einmal geschrieben haben (oder noch schlimmer, das Sie von jemand anderem übernommen haben) und nun ändern müssen.

Ein Kommentar fängt mit einem Doppelkreuz (engl. *Hash*) `#` an. Alle folgenden Zeichen in dieser Zeile werden dann bei der Bearbeitung des Skripts durch die Shell ignoriert. Verzieren wir also unser Skript mit ein paar Kommentarzeilen:


```
# JLI Shellprogrammierung: Skript 1
#
# Gibt das aktuelle Arbeitsverzeichnis aus
pwd    # Der einzige Befehl
```

Speichern Sie das Skript ab, und rufen Sie es erneut auf. Ihr Skript verhält sich genauso wie die erste Version.

1.3 Ein Skript beenden

Sie werden sich jetzt sicherlich fragen, wozu dieser Abschnitt notwendig ist. Schließlich sind Sie ja nach dem Aufruf Ihres Skripts wieder in der Shell gelandet. Dies sollte ein klares Indiz dafür sein, das das Skript beendet wurde. Das stimmt schon, allerdings sollte jedes Skript einen Wert an die Shell zurückgeben, der dieser klarmacht, ob das Skript fehlerfrei lief oder nicht. Tra-

ditionell ist ein Rückgabewert (auch Exit-Status genannt) von 0 ein Indiz dafür, dass ein Skript fehlerfrei abgelaufen ist. Jeder andere Wert deutet auf einen Fehler hin. Welche Bedeutungen die einzelnen Werte im Fehlerfall haben, ist allein Ihnen bzw. Ihrem Skript überlassen, allerdings müssen die Werte im Bereich von 0 bis 255 bleiben, da nur ein Byte ausgewertet wird. Deshalb beenden Sie bitte Ihr Skript immer mit einem `exit`, gefolgt von einem Fehlerwert bzw. `0`. Ein `exit` beendet das Skript sofort, unabhängig von seiner Position innerhalb des Skripts.

Sie fragen sich jetzt sicherlich, wofür das gut sein soll? Mal angenommen, Sie schreiben ein Skript, das eine Sicherung Ihrer wichtigsten Daten durchführt. Das Skript kopiert Ihre Daten auf Diskette, Band oder Zip-Disk und löscht sie danach aus dem Quellverzeichnis. Schlägt nun das Kopieren fehl, weil der Zieldatenträger voll ist, und stellt das Skript nicht sicher, dass das Kopieren geklappt hat, bevor es den Löschbefehl abarbeitet, sind Ihre Daten futsch. Also gewöhnen Sie sich das korrekte Beenden eines Skripts mit `exit` gleich an, auch wenn es jetzt noch nicht so sinnvoll erscheint:


```
# JLI Shellprogrammierung: Skript 1
#
# Gibt das aktuelle Arbeitsverzeichnis aus
pwd     # Der einzige Befehl
exit 0  # Diese Zeile ist neu: Alles ok
```

Sollten Sie den Aufruf von `exit` vergessen, so ist der Rückgabewert des Skripts identisch mit dem Rückgabewert des letzten ausgeführten Skriptbefehls. Dies gilt auch, wenn man `exit` ohne Argument aufruft. Geben Sie nun Folgendes ein, um den Rückgabewert des letzten aufgerufenen Befehls zu ermitteln:

```
buch@koala:/home/buch > ./skript1.sh
/home/buch
buch@koala:/home/buch > echo $?
0
buch@koala:/home/buch > false
buch@koala:/home/buch > echo $?
1
buch@koala:/home/buch >
```

Da Ihr Skript zuletzt aufgerufen wurde und mittels `exit 0` beendet wurde, gibt ein anschließendes `echo $?` , den Wert `0` aus. `false` ist ein Befehl, der nur das Ergebnis *falsch* zurückgibt. Im Gegensatz zur Programmiersprache C ist *falsch* in der Shell immer ein Wert ungleich `0` und *wahr* immer `0`.

1.4 Was sind Variablen?

Jede Programmiersprache hat Variablen, und Shellskripten bilden da keine Ausnahme. Variablen sind Platzhalter für variable Werte, mit denen das Skript arbeitet. Der Name kann eine beliebige Anzahl von Zeichen aus dem Bereich A bis Z, 0 bis 9 und »_« sein. Groß- und Kleinschreibung müssen bei der Vergabe der Namen beachtet werden (`Var` ist ungleich `VAR`). Andere Zeichen sind nicht erlaubt, da diese meist eine besondere Bedeutung für die Shell haben.

Das war doch nicht schwer, oder? Um mit den Variablen aber etwas anfangen zu können, sollten Sie ihnen Werte zuweisen:


```
# Skript 2: Variablenzuweisung
# Version 1
EineVar=10
VarZwei=Text
Bruchzahl=10.12
exit 0
```

Solche Variablen sind der Shell so lange bekannt, wie das Skript läuft. Sobald das Skript beendet wird, sind auch die Variablen wieder unbekannt.

Die Tatsache, dass nach dem Beenden eines Skripts der gleiche Zustand wiederhergestellt wird wie vor dem Aufruf des Skripts, gilt übrigens für alle Shelleinstellungen. Nur die Bildschirmausgaben sind von der Wiederherstellung ausgeschlossen. Die Summe aller Variablen nennt der Fachmann übrigens *Shellumgebung* oder neudeutsch *Environment*. Solange Sie die Shell nicht explizit anweisen, etwas in diese Umgebung aufzunehmen, sind die Änderungen nur temporär, d.h. für die Dauer der Skriptausführung gültig.

Was momentan in Ihrer aktuellen Umgebung eingetragen ist, sehen Sie durch den Befehl `env` oder `printenv`.

1.5 Variablen referenzieren

An dieser Stelle können Sie also Variablen Werte zuweisen, aber nützlich waren Variablen bis jetzt jedenfalls nicht. Ein Shellskript sollte Variablen setzen, modifizieren und abfragen können. Die Shell erkennt ein Wort als Variable, wenn vor dem Variablennamen ein $-Zeichen steht. In diesem Fall wird der Variablenname und das führende $ durch den Wert der Variablen ersetzt.

Bevor wir uns auf ein einfaches Beispiel stürzen, braucht das Skript noch eine Möglichkeit, einen Text bzw. den Inhalt einer Variablen auszugeben. Dazu dient der Befehl `echo`. `echo` akzeptiert eine beliebige Anzahl an Parametern und gibt diese auf dem Bildschirm aus.

Ein Beispiel:


```
# Skript 3: Variablen
# Dieses Skript demonstriert die Arbeit mit Variablen
#
tier=Koala
land=Australien
echo Der $tier lebt in $land
exit 0
```

Der Aufruf dieses Skripts bewirkt folgende Ausgabe:

```
buch@koala:/home/buch > ./skript3.sh
Der Koala lebt in Australien
buch@koala:/home/buch >
```

Diese Art der Variablenzugriffe klappt recht gut, ist aber leider nicht ohne Probleme, wie das folgende Beispiel nur zu deutlich macht:


```
# Skript 4: Variablen
# Demonstriert Probleme mit Variablennamen und
# der Wortaufteilung
#
anzahl=4
echo Dies ist das $anzahl. Skript
echo Deshalb haben Sie mindestens $anzahlmal den Editor aufgerufen
exit 0
```

Eigentlich sollte dieses Skript Folgendes ausgeben:

```
buch@koala:/home/buch > ./skript4.sh
Dies ist das 4. Skript
Deshalb haben Sie mindestens 4mal den Editor aufgerufen
```

Ausprobiert? Die erste Zeile wurde wie erwartet ausgegeben. In der zweiten Zeile stand aber:

```
Deshalb haben Sie mindestens den Editor aufgerufen
```

Was ist denn nun schon wieder falsch? Die Shell unterteilt ein Skript immer erst in Zeilen und die Zeilen wiederum in Worte. Einzelne Worte erkennt die

Shell an der Begrenzung durch Leer- und Sonderzeichen. Schauen wir uns die Variablenreferenz in beiden Zeilen unter dieser Voraussetzung an:

```
Dies ist das $anzahl. Skript
```

Da das $ durch den Namen und nicht von Leer- bzw. Sonderzeichen gefolgt wird, erkennt die Shell dies als Variable namens `anzahl` und ersetzt die Variable durch ihren Inhalt. Der Punkt ist ein Sonderzeichen, das in Variablennamen wie gelernt nicht enthalten sein darf und daher als Trennzeichen fungiert.

```
Dies ist das 4. Skript
```

Erst dann wird `echo` aufgerufen und bekommt dadurch folgende Parameter übergeben:

```
»Dies« »ist« »das« »4.« »Skript«
```

Schauen wir auf das zweite `echo` und unterteilen auch diese Zeile in Wörter:

```
… »mindestens« »$anzahlmal« »den«
```

Auch hier haben wir eine Variable, aber diesmal erkennt die Shell eine Variable namens `anzahlmal`. Da diese jedoch nicht definiert wurde, ist deren Wert leer (gleich `""`), und so kommt das für uns unbefriedigende Ergebnis zustande.

Abhilfe naht in Gestalt der geschweiften Klammern {}: Wenn der Variablenname mit diesen Klammern eingerahmt wird, so erkennt die Shell unsere Intention und reagiert so, wie von uns gewünscht.


```
# Skript 5: Variablen
# Demonstriert Probleme mit Variablennamen und
# die Wortaufteilung, diesmal aber korrekt
#
anzahl=4
echo Dies ist das $anzahl. Skript
echo Deshalb haben Sie mindestens ${anzahl}mal den Editor aufgerufen
exit 0
```

Schon besser, oder?

Man sollte sich angewöhnen, Variablen immer in {} zu setzen, wenn sie nicht von Leerzeichen umrahmt sind. Es schadet nicht und macht die Intention klar. Welche Sonderzeichen einen Variablennamen beenden und welche nicht (gemeint ist der Unterstrich »_«), ist nicht sofort erkennbar.

1.6 Quoting

Falls Sie mit den Variablen (d.h. den Skripten 3–5) schon ein wenig experimentiert haben, ist Ihnen sicherlich ein Problem aufgefallen:

```
...
LangerText=Diese dummen Skripten laufen eh nicht!
...
```

gibt folgende Fehlermeldung aus:

```
./skript3.sh: dummen: command not found
```

Sollte Ihnen jetzt jemand über die Schulter schauen, und es ist Ihnen wichtig, dass diese Person den Eindruck erhält, Sie wüssten (schon), was Sie tun, dann reagieren Sie wie folgt:

- Verdrehen Sie wissend die Augen.
- Schütteln Sie ärgerlich den Kopf.
- Murmeln Sie das Stichwort *Quoting* vor sich her.
- Lesen Sie schnell den nächsten Abschnitt, *bevor* diese Person weitere Fragen stellen kann.

Rufen Sie sich die Tatsache noch einmal ins Gedächtnis, wie die Shell Skripten unterteilt, nämlich zunächst in Zeilen und diese wiederum in Worte. So besteht unser Beispiel aus folgenden Worten:

```
»LangerText« »=« »Diese« »dummen« »Skripten« »laufen« »eh« »nicht!«
```

Die ersten drei Worte ergeben eine Variablenzuweisung. Das vierte Wort ist jedoch weder eine Variablenzuweisung noch ein Shellbefehl, und genau das sagt die Fehlermeldung auch aus!

Wenn Sie also einer Variablen einen Wert zuweisen wollen, der aus mehr als einem Wort besteht oder Sonderzeichen beinhaltet, so setzen Sie den Text in Anführungszeichen. Dadurch wird der Text zwischen den Anführungszeichen wie ein einziges Wort behandelt. Die Gänsefüßchen selbst sind nicht Teil des Worts. In diesem Wort werden Sonderzeichen (bis auf wenige Ausnahmen) nicht interpretiert, sondern unverändert ausgegeben:

```
...
LangerText="Diese dummen Skripten laufen eh nicht!"
...
```

Sollten Sie gar eine Fehlermeldung `LangerText command not found` erhalten haben, so müssen Sie die Leerzeichen zwischen Variablennamen und Gleichheitszeichen entfernen, sonst erkennt die Shell nicht, dass es sich um eine Zuweisung handelt!

O.K.! Wunderbar, das Leben ist (wieder) schön. Aber was machen Sie, wenn der Text selbst ein Anführungszeichen enthält?


```
# Skript 6: Version 2
# Variablenzuweisungen mit mehreren Worten
# Probleme mit dem Zeichen "
#
TxtAnf="Dieser Text hat 2 Anführungszeichen: """
echo $TxtAnf
exit 0
```

Die Ausgabe deckt sich möglicherweise nicht mit Ihren Erwartungen:

```
Dieser Text hat 2 Anführungszeichen:
```

Das liegt daran, dass zwei Zeichenketten in der Zuweisung definiert werden:

`"Dieser Text hat 2 Anführungszeichen: "` und `""`. Letzteres ist eine leere Zeichenkette, die nichts bewirkt. Ignorieren wir dieses Problem einen kurzen Moment und halsen uns zunächst noch mehr Ärger auf. Angenommen, Sie möchten folgenden Text durch ein Skript ausgeben lassen:

```
Die Variable $anz hat den Wert 3
```

Das Skript soll eine Variable `anz` enthalten, die vor der Ausgabe des Textes den Wert `3` zugewiesen bekommt (`anz=3`). Die Ausgabe selbst soll zunächst den Variablennamen als reinen Text ausgeben und dann den Inhalt der Variablen selbst.

Ein erster optimistischer Versuch endet in einer Version, die so aussehen könnte:


```
# Skript 7:
# Sonderzeichen und Variablenersetzung in Zeichenketten
#
anz=3
# Versuch 1
echo Die Variable "$anz" hat den Wert $anz
# Versuch 2
echo "Die Variable "$anz" hat den Wert $anz"
exit 0
```

Wenn Sie jetzt einwenden, dass dieses Skript auf jeden Fall zwei Zeilen ausgibt und somit die Aufgabenstellung garantiert verfehlt, haben Sie absolut Recht. Wir sind uns jedoch sicher, dass Sie auf Grund der bisher abgehandelten The-

orie eine der beiden Zeilen als Lösung ermittelt haben. Leider werden Sie aber in beiden Fällen die gleiche (und leider falsche) Ausgabe erhalten:

```
buch@koala:/home/buch > ./skript7.sh
Die Variable 3 hat den Wert 3
Die Variable 3 hat den Wert 3
buch@koala:/home/buch >
```

Auch hier liegt das Problem darin, dass das $-Zeichen, wie das Anführungszeichen auch, eine besondere Bedeutung besitzt. Diese Bedeutung führt in beiden Fällen dazu, dass die Zeichen ersetzt werden und dem `echo`-Befehl erst gar nicht übergeben werden. So weit, so gut. Aber wie lässt sich dieses Problem nun lösen? Die Shell kennt ein so genanntes *Fluchtzeichen*, das dazu führt, dass die Sonderbedeutung des folgenden Zeichens aufgehoben wird. Das Fluchtzeichen ist das »\«-Zeichen (engl. *Backslash*).

Das Fluchtzeichen selbst wird dabei nicht ausgegeben, was gleich zum nächsten Problem führt: Wie gibt man ein einzelnes \ aus? Ganz einfach: Sie geben ein doppeltes \ an. Das erste \ hebt die Sonderbedeutung des folgenden Zeichens auf und wird nicht ausgegeben. Das zweite Zeichen ist erneut das »\«-Zeichen und wird folglich ausgegeben.

Das Fluchtzeichen können Sie an jeder Stelle innerhalb eines Skripts einsetzen. Es ist nicht auf das `echo`-Kommando beschränkt.

Wir hatten bereits oben besprochen, wie die Shell Skripten in Worte aufteilt. Findet die Shell bei diesem Vorgang ein \, so entfernt sie es und hebt die Sonderbedeutung des auf \ folgenden Zeichens auf.

Tabelle 1.1: Zeichen mit Sonderbedeutung in ihrer wörtlichen Bedeutung benutzen

Sequenz	Ergebnis nach	Gültig für Shellinterpretation
\$	ergibt $	gesamtes Skript (Ausnahme Kommentare)
\`	ergibt `	gesamtes Skript, siehe Text
\\	ergibt \	dito
\#	ergibt #	dito

Ein Beispiel sollte die Sache klären: `echo \$anz $anz` wird von der Shell umgeformt zu `echo $anz 3`. Der Backslash verhindert für das erste $, dass die Shell eine Variable erkennt, während das zweite $ die Variable `anz` referenziert. Daher wird diese durch deren Wert (3) ersetzt. Damit wird `$anz` und `3` an `echo` übergeben.

```
# Skript 7a:
# Sonderzeichen und Variablenersetzung in Zeichenketten
#
anz=3
echo Die Variable \$anz hat den Wert $anz
exit 0
```

Kehren wir nun wieder zurück zu unseren Variablen. Bis jetzt waren Sie nur in der Lage, einer Variablen eine Konstante oder den Inhalt einer anderen Variablen (und somit wieder einen konstanten Wert) zuzuweisen. Somit sind Variablen bis jetzt eher unbrauchbar. Schließlich sollten sie Werte beinhalten, die berechnet werden. Auch das können Sie erreichen.

Setzen Sie einen beliebigen Shellausdruck in rückwärtige Anführungszeichen (engl. *Backticks*) ` `, so wird der Teil zwischen den Anführungszeichen als Befehl betrachtet und ausgeführt. Die Ausgabe des Befehls wird nicht auf dem Bildschirm ausgegeben, sondern ersetzt die Zeichenkette zwischen `` . Die Shell unterteilt dann erst das Ergebnis in einzelne Worte und interpretiert diese.

```
# Skript 8:
# Ausführungszeichen
#
# pwd gibt das aktuelle Arbeitsverzeichnis aus (Print Working
# Directory)
akt=`pwd`
cd .. # Ein Verzeichnis zurück: aus /home/buch wird /home
echo "Das aktuelle Verzeichnis ist $akt"
exit 0
```

Dieses Skript gibt aus:

```
buch@koala:/home/buch > ./skript8.sh
Das aktuelle Verzeichnis ist /home/buch
buch@koala:/home/buch >
```

Schauen wir uns noch einmal die zentrale Zeile an:

```
akt=`pwd`
```

Zuerst führt die Shell das Kommando `pwd` aus und ersetzt die Zeichenkette `` `pwd` `` durch die Ausgabe von `pwd`. Da das Arbeitsverzeichnis `/home/buch` ist, steht nach dem ersten Durchlauf in der Zeile

```
akt=/home/buch
```

Diese Zeile wird in Worte unterteilt und interpretiert: »`akt`« »=« »`/home/buch`« und endgültig von der Shell ausgeführt. Damit bekommt die Variable `akt` den Wert `/home/buch` zugewiesen.

Mithilfe des Befehls `cd` (*change directory*) können Sie das Verzeichnis wechseln. Dabei steht »`..`« für das übergeordnete Verzeichnis, Sie wechseln also auf die nächsthöhere Ebene. Dieser Befehl soll zweierlei zeigen:

- `akt` enthält das Verzeichnis, das `pwd` ausgegeben hat, und nicht automatisch immer das aktuelle!
- Nachdem das Skript beendet wurde, stehen Sie wieder im alten Verzeichnis `/home/buch`, obwohl das Skript am Ende in `/home` stand!

Neben den beiden besprochenen Anführungszeichen gibt es noch das Apostroph `'`. Dieses können Sie an der gleichen Stelle wie das Gänsefüßchen setzen, und die Funktionalität ist fast identisch. Allerdings interpretiert die Shell innerhalb der Apostrophe keine Sonderzeichen wie `$` oder `\`. Es werden genau die Zeichen bearbeitet, die zwischen den Apostrophen stehen.


```
buch@koala:/home/buch > echo '$anz'
$anz
buch@koala:/home/buch > echo '\'
\
buch@koala:/home/buch >
```

– Verwechseln Sie nicht das Ausführungszeichen ` mit dem Apostroph '. Das ' hat fast die gleiche Bedeutung wie das Anführungszeichen " und kann auch an dessen Stelle eingesetzt werden (siehe oben). Dadurch wird aber eine Zeichenkette zugewiesen und nichts ausgeführt!

– Sollte die Bash folgende Meldung ausgeben:

  ```
  skript8.sh: line 3: unexpected EOF while looking for matching "' ,
  ```

  dann stimmt die Anzahl der Anführungszeichen (bzw. Apostrophe) innerhalb des Skripts nicht. Zählen Sie alle Anführungszeichen, die nicht mit einem Fluchtzeichen versehen sind. Da Anführungszeichen im Skript paarweise auftauchen, benötigen Sie eine gerade Anzahl. Ist das nicht der Fall, fängt die Bash an zu nörgeln.

Die passenden Anführungszeichen müssen nicht in der gleichen Zeile stehen, obwohl das der Übersicht durchaus zuträglich ist.

Noch ein kleines Beispiel zum Abschluss dieser Thematik. Allerdings brauchen Sie dazu zunächst noch zwei weitere Unixbefehle. Schauen Sie sich also einmal die folgende Erklärung an, und probieren Sie danach das Beispiel aus.

Unter Unix gibt es den Befehl `expr`, der numerische Ausdrücke auswertet und das Ergebnis auf dem Bildschirm ausgibt. So führt die Eingabe

```
buch@koala:/home/buch > expr 1 + 2
```

zur Ausgabe der Zahl

```
3
buch@koala:/home/buch >
```

Die Leerzeichen um das Plus sind wichtig! `expr 1+2` gibt `1+2`, nicht `3`.

Außerdem brauchen Sie noch den Befehl `cat`. Dieser Befehl gibt den Inhalt der als Parameter angegebenen Datei auf dem Bildschirm aus. Ein Beispiel:

```
buch@koala:/home/buch > cat skript1.sh
# Shellprogrammierung: Skript 1
#
# Gibt das aktuelle Arbeitsverzeichnis aus
pwd     # Der einzige Befehl
exit 0  # Diese Zeile ist neu: Alles ok
buch@koala:/home/buch >
```

Das folgende Skript geht davon aus, dass in der Datei `math.txt` eine positive Zahl steht, multipliziert diese mit 4 und gibt das Ergebnis aus.


```
# Skript 9: Ausführungszeichen
#  und expr
#
ausdruck=`cat math.txt`
echo "Das Ergebnis lautet `expr 4 \* $ausdruck`"
exit 0
```

Noch einmal der Hinweis darauf, dass doppelte Anführungszeichen nicht verhindern, dass die Shell Sonderzeichen interpretiert! Zum einen nutzt das Skript dies durch die Ausführungszeichen im `echo` aus, hat aber gleichzeitig Probleme, da auch das * ein Sonderzeichen ist und deshalb mit dem Fluchtzeichen vor der Interpretation geschützt wird.

Nach all Ihrer Mühe nenne ich Ihnen auch noch den Fachbegriff dessen, was Sie gerade in verschiedensten Varianten durchgeführt haben: *Quoting*. Quoting verhindert also, dass Sonderzeichen, die eine besondere Bedeutung für die Shell haben, von dieser erkannt werden.

Warum wir Ihnen erst jetzt diese Fachbegriffe beibringen? Der Unterschied zwischen einem Möchtegern-Skriptprogrammierer und einem Fachmann ist der: Der Angeber streut einen Fachbegriff in eine Fachdiskussion ein, um zu beeindrucken. Er sieht Fachchinesich nur als eine andere Art von Poker an. Verloren hat derjenige, dem zuerst die Fachbegriffe ausgehen oder dessen Fachbegriffe sich weniger imposant anhören. Ein echter Fachmann nutzt Fachbegriffe, um sich langatmige Erklärungen zu sparen und seine Argumentation schnell auf den Punkt zu bringen.

Ausführliche Informationen über Variablen gibt es in Kapitel 6. Dort findet sich auch die genaue Erklärung, warum Variablen nur temporär bekannt sind und wie die Shellumgebung manipuliert und abgefragt werden kann. Bis dahin kommen Sie jedoch ohne weiteres mit den bereits ausgeführten Erklärungen aus.

1.7 Parameter/Argumente

Parameter (häufig auch Argumente genannt) sind Werte, die Sie einem Skript beim Aufruf mitgeben können. In der Eingabeaufforderung der Shell werden solche Werte durch Leerzeichen getrennt:

```
buch@koala:/home/buch > ./skript10.sh Parameter1 Parameter2 Param3
```

Dabei werden die Parameter von der aufrufenden Shell in einzelne Wörter unterteilt, nicht von `skript10.sh`. Diese kann das Skript nun wie Variablen ansprechen, und zwar durch Angabe von `$0` bis `$9`. `$0` steht für den Dateinamen des Skripts, `$1` für den ersten Parameter, `$2` für den zweiten usw. Ein `$10` gibt es nicht ohne weiteres, sondern `$10` wird als `${1}0` interpretiert. Wie Sie über die Parameter `$1` bis `$9` hinaus noch weitere Parameter in einem Skript ansprechen können, werden wir in späteren Kapiteln sehen. Vorerst können Sie zwar eine beliebige Anzahl Parameter übergeben, aber nur die ersten neun ansprechen.

Die Werte der Parameter können Sie übrigens nicht direkt ändern, sie bleiben konstant. Ein Zuweisung `$1=wert` ist nicht möglich und resultiert in einem Fehler. Mit dem Parameter `$#` können Sie ermitteln, wie viele Parameter dem Skript übergeben wurden.

Auch für den Exit-Status des letzten Befehls gibt es einen Parameter: `$?`. Soweit zur Theorie, das folgende Skript zeigt, wie Sie diese Informationen praktisch anwenden:


```
# Skript 10: Parameter
#  Erste Annäherung an die Shellparameter
#
echo "Es wurden $# Parameter dem Skript $0 übergeben"
echo "P1=$1"
echo "P2=$2"
exit 0
```

So, das soll es für dieses Kapitel gewesen sein. War doch gar nicht sooo schlimm, oder?

Noch ein Hinweis, bevor uns unser Lektor wieder mit Korrekturwünschen überschüttet: Tatsächlich wird ein aufgerufenes Skript nicht einfach so ausgeführt, sondern die interaktive Shell (die, in der Sie `skript10.sh` aufgerufen haben) startet eine weitere Shell, die dann das Skript ausführt. Diese zweite Shell bekommt eine Kopie der Umgebung von der interaktiven Shell, die das Skript dann manipulieren kann. Wird das Skript beendet, so geht die Kopie der Shellumgebung verloren und mit ihr sämtliche Änderungen.

1.8 Aufgaben

Aufgaben sollten ja eigentlich ein wenig Praxis in den grauen Theoriealltag bringen, aber leider fällt dieses Kapitel diesbezüglich ein wenig aus dem Rahmen. Die in diesem Kapitel vermittelten Grundlagen müssen Sie unbedingt verstehen, sonst sind Sie in den folgenden Kapiteln verloren. Also Zähne zusammenbeißen:

1. Was passiert in Skript 4, wenn Sie folgende Zeile vor dem ersten `echo` einfügen?

   ```
   anzahlmal
   ```

2. Korrigieren Sie Skript 6 so, dass es läuft wie vorgesehen.

3. Ermitteln Sie doch einmal für Skript 7, welche Worte die Shell für beide `echo`-Befehle erstellt, welche Ersetzungen durchgeführt werden und was schließlich `echo` als Parameter bekommt.

4. Was passiert, wenn Sie in Skript 9 im `echo`-Befehl statt der Gänsefüßchen Apostrophe verwenden?

5. Sicherlich kennen Sie die ASCII-Smilies, mit denen man in öden ASCII-Textwüsten Gefühle zum Ausdruck bringt: `:-)` oder `;)` sind wohl die bekanntesten.

Weniger bekannt sind die Linkshändersmilies: (-: soll als Beispiel genügen.

Jetzt denken Sie mal an die Fehlermeldung von Skript 3. Welche drei Zeichen müssen Sie eingeben, damit die Shell Sie anlacht? Dies ist *kein* echtes Skript und auch nicht sinnvoll, sondern eine einfache Denksportaufgabe (Tipp: Quoting).

1.9 Lösungen

1. Sie bekommen als Ergebnis ausgegeben:

```
buch@koala:/home/buch > ./skript4.sh
Dies ist das 4. Skript
Deshalb haben Sie mindestens 10 den Editor aufgerufen
```

2. Sie müssen Skript 6 folgendermaßen abwandeln:


```
# Skript 6: Version 2
# Variablenzuweisungen mit mehreren Worten
TxtAnf="Dieser Text hat 2 Anführungszeichen: \"\""
echo $TxtAnf
exit 0
```

   Das abgeänderte Skript liefert nun diese Ausgabe:

```
Dieser Text hat 2 Anführungszeichen: ""
```

3. Im ersten `echo`-Befehl wird die Variable `anz` zweimal durch die Zahl `3` ersetzt. Anschließend werden `echo` die Wörter

```
»Die« »Variable« »3« »hat« »den« »Wert« »3«
```

   übergeben. Auch beim zweiten `echo`-Befehl wird zunächst die Variable `anz` durch die Zahl `3` ersetzt. Dieses Mal bekommt `echo` aber folgende Wörter mit auf den Weg:

```
»Die Variable« »3« »hat den Wert 3«
```

4. Der Befehl `echo 'Das Ergebnis lautet `expr 4 * $ausdruck`'` gibt genau die Zeichen zwischen den Literalen wieder, also:

```
Das Ergebnis lautet `expr 4 \* $ausdruck`
```

Kapitel 2

Interaktion von Programmen

»Oh Herr, gib mir Keuschheit und Selbstbeherrschung ...
aber noch nicht, oh Herr, noch nicht« –
Der heilige Augustinus (354–430)

Zumindest Selbstbeherrschung werden Sie aber brauchen, um dieses Kapitel zu überstehen. Auf dem Lehrplan steht jetzt die Interaktion der Shell und der von ihr aufgerufenen Befehle mit der (Computer-)Umwelt. Dazu gehören Befehlslisten, Pipes und Ein-/Ausgabeumlenkung. Zusätzlich enthält dieses Kapitel Informationen über Jokerzeichen, wie z.B. * und ?.

2.1 Ein-/Ausgabeumlenkung

Wenn Sie sich mit Unix schon ein wenig beschäftigt haben, werden Ihnen die Begriffe *Standardeingabe*, *Standardausgabe* und *Standardfehler* (*Standarderror*) schon einmal begegnet sein. Was bedeutet das nun für uns? Jedes in der Shell gestartete Programm hat diese drei Kanäle. Solange einem Befehl nicht explizit gesagt wird, er soll seine Daten aus einer Datei holen, versucht er, diese Daten von der Standardeingabe zu holen. Ausgaben (z.B. per `echo` oder `cat`) gehen immer auf die Standardausgabe. Rufen Sie doch einfach mal den Befehl `cat` ohne Parameter auf, geben Sie ein paar Zeilen ein, und brechen Sie mit Strg+C ab.

Fehlermeldungen werden auf einem gesonderten Kanal ausgegeben, um zwischen normaler Ausgabe und Fehlertexten unterscheiden zu können. Zum

Thema *Standardfehler* kommen wir gleich (ein paar Seiten später) noch ausführlich.

In der Loginshell ist der Eingabekanal identisch mit der Tastatur. Beide Ausgabekanäle sind mit dem Bildschirm verbunden. Jeder Kanal hat eine eindeutige Nummer, anhand deren die Shell die Kanäle identifiziert.

Tabelle 2.1: Die Standard-ein- und -ausgabekanäle

Standardeingabe	0
Standardausgabe	1
Standardfehler	2

Diese vorgegebene Zuordnung der Kanalnummern kann nicht geändert werden. Allerdings ist es jederzeit möglich, z.B. den Eingabekanal von der Tastatur auf eine Datei Ihrer Wahl umzulenken. Gleiches gilt auch für beide Ausgabekanäle. Auch diese sind nicht auf den Bildschirm festgelegt. Ein Umlenken kann jederzeit für jeden einzelnen Befehl innerhalb eines Skripts vorgenommen werden.

Wie diese Zuordnung geändert werden kann, wollen wir jetzt genauer unter die Lupe nehmen.

2.1.1 Ausgabeumlenkung

Wenn Sie die Ausgaben nicht auf den Bildschirm ausgeben wollen, sondern in einer Datei benötigen, fügen Sie ein > und den Dateinamen an den Befehl an:

```
buch@koala:/home/buch > ls -l > tmp.txt
buch@koala:/home/buch >
```

Existiert die Datei nicht, in die die Ausgabe umgelenkt werden soll, so wird sie angelegt. Falls die Datei bereits existiert, wird der alte Inhalt mit den neuen Ausgaben überschrieben.

Dieses Verfahren ist durchaus praktisch, aber nicht immer gewünscht. Manchmal ist es sinnvoller, die neuen Daten an eine Datei anzuhängen. Auch das geht in der Shell. Dazu nutzen Sie den Befehl >>, gefolgt vom Dateinamen, in den die Ausgabe umgelenkt werden soll. Existiert die Datei noch nicht, so wird sie durch die Umleitung angelegt. Existiert sie schon, so werden in diesem Fall die neuen Daten am Ende der Datei angehängt.

Mit dem bisherigen Wissen wollen wir nun ein Skript schreiben, welches zwei Parameter nimmt, diese als Verzeichnisnamen interpretiert und deren Inhalt in eine Datei schreibt. Als Ergebnis gibt das Skript die Summe der Dateien in beiden Verzeichnissen aus. Die Summe ist dabei gleich der Anzahl der Zeilen in der Datei.

Die Anzahl an Zeichen, Wörtern und Zeilen kann man mit dem Befehl `wc` ermitteln. Genauer `wf-cwl <datei>`, wobei die Optionen `c`, `w` und `l` Folgendes bedeuten:

- `-c` zählt die Zeichen in der Datei (engl. *character*)
- `-w` zählt die Wörter in der Datei (engl. *word*)
- `-l` zählt die Zeilen in der Datei (engl. *line*)

`wc` gibt den oder die angeforderten Werte plus Dateinamen aus. Falls Sie sich nun fragen, wofür denn nun wieder `wc` steht: Dies ist eine Abkürzung für *word count*, zählt also Wörter und zusätzlich entgegen der Übersetzung auch noch Zeilen und Zeichen.

Und weil wir so ordentlich sind, löschen wir alles, was wir nur temporär gebraucht haben. Dateien werden dabei mit `rm` (*Remove*) gelöscht. Die Option `-f` erzwingt dabei ein Löschen ohne eventuelle Nachfragen. Falls ein Löschen nicht möglich ist, wird ein Fehler ausgegeben, und der Exitstatus ist ungleich `0`.

Es gibt vor allem zwei Gründe, die Dateien zu löschen, die nicht mehr gebraucht werden:

– Schafft Ordnung und reduziert Platzbedarf.

– Sicherheitsgründe. Es geht ja niemanden etwas an, was Sie gemacht haben.

Aus diesem Grunde empfiehlt es sich übrigens auch, Dateien, in die etwas umgelenkt werden soll, vor der ersten Umlenkung zu löschen. Wer weiß, wo Ihre Daten landen, wenn die Datei schon existierte und ein Link auf eine andere Datei war?

Durch `rm` gelöschte Dateien sind unrettbar verloren. Vor allem als Benutzer `root` überlegen Sie bitte zweimal, bevor Sie eine Datei löschen! Rufen Sie `rm` lieber mit der Option `-i` auf, sodass jeder Dateiname ausgegeben wird und deren Löschung durch Eingabe von `y` bestätigt werden muss. Eine Interaktion ist bei Skripten allerdings nicht immer sinnvoll, weshalb `-i` eher etwas für manuelle Löschvorgänge ist. Legt Ihr Skript Temporärdateien an, so sollte es diese auch löschen!


```
# Skript 11: Die Anzahl an Dateien in
# zwei Verzeichnissen ermitteln
#
tmpfile="/tmp/erg"
ls $1  > $tmpfile
ls $2 >> $tmpfile
echo "Die Verzeichnisse $1 und $2 enthalten `wc -l $tmpfile` Dateien"
rm $tmpfile
exit 0
```

Ein Aufruf des Skripts mit zwei Verzeichnisnamen liefert uns ein korrektes Ergebnis. Schön ist die Ausgabe allerdings nicht, wir werden uns also noch einmal mit diesem Skript beschäftigen, wenn wir mehr Wissen angesammelt haben.

Sowohl `ksh` als auch `bash` haben die Option `clobber/noclobber`. Diese Option beeinflusst, wie sich die Shell verhält, wenn eine Datei durch Ausgabeumlenkung überschrieben werden soll. Normalerweise steht diese Option auf `clobber`, was ein Überschreiben bereits vorhandener Dateien erlaubt. Wurde `noclobber` aktiviert, so führt eine Umlenkung mittels > auf eine bereits existierende Datei zu einem Fehler. Wenn Sie unter diesen Umständen ein Überschreiben erzwingen wollen, nutzen Sie die Umlenkung per >|.

Die Option `noclobber` wird beim Aufruf von `ksh` oder `bash` durch die Option `-C` gesetzt oder in der Shell durch ein `set -C`. `clobber` lässt sich durch die Angabe von `+C` beim Aufruf der Shell einstellen oder in der Shell selbst durch ein `set +C`.

```
buch@koala:/home/buch > echo "Koala" >/tmp/cw
buch@koala:/home/buch > echo "Wallaby" >/tmp/cw
buch@koala:/home/buch > set -C
buch@koala:/home/buch > echo "Eukalyptus" >/tmp/cw

bash: /tmp/cw: cannot overwrite existing file
buch@koala:/home/buch > set +C
buch@koala:/home/buch > echo "Wallaby" >/tmp/cw
buch@koala:/home/buch >
```

2.1.2 Eingabeumlenkung

Natürlich ist es auch möglich, die Eingabe eines Befehls nicht von der Tastatur zu holen. Dazu wird < verwendet. Die Eingabeumlenkung erwartet ebenfalls einen Dateinamen, aus dem der Inhalt ausgelesen und an den vom Umlenken

betroffenen Befehl weitergeleitet wird. So kann eine Datei mit `cat` ausgegeben werden:

```
buch@koala:/home/buch > cat skript1.sh
# JLI Shellprogrammierung: Skript 1
#
# Gibt das aktuelle Arbeitsverzeichnis aus
pwd     # Der einzige Befehl
exit 0  # Diese Zeile ist neu: Alles ok
buch@koala:/home/buch >
```

Das gleiche Ergebnis erhalten Sie auch durch:

```
buch@koala:/home/buch > cat < skript1.sh
...
```

Wie wir gesehen haben, funktionierte Skript 11 zwar, aber toll war die Ausgabe wirklich nicht. Die zusätzliche Ausgabe des Dateinamens ist mehr als störend, wird aber unterdrückt, wenn die Daten über die umgelenkte Standardeingabe kommen. Ändern wir also die Ausgabezeile wie folgt ab:

```
echo "Die Verzeichnisse $1 und $2 enthalten `wc –l < $tmpfile` Dateien"
```

Ich wette, die neue Version ist besser als die alte. Wenn Sie jetzt aber glauben, das wäre cool, sollten Sie erst einmal den Rest des Buches abwarten.

2.1.3 Standardfehler (Standarderror)

Jetzt wird es Zeit, eines der wichtigsten Mysterien innerhalb dieses Kapitels zu enthüllen: die Standardfehlerausgabe. Wie bereits erwähnt, ist die Fehlerausgabe zunächst mit dem Bildschirm verbunden. Da dies auch für die Standardausgabe gilt, stellt sich die Frage, welchen Sinn der Fehlerkanal hat.

Das Verfahren, normale Ausgaben und Fehlerausgaben über getrennte Ausgabekanäle abzuwickeln, gibt dem Programmierer die Möglichkeit, Fehlerausgaben getrennt von den normalen Ausgaben zu behandeln. Vor allem aber bekäme man innerhalb von Pipes die Fehlermeldung niemals zu sehen, sie ginge in der Menge der Daten einfach unter.

Es ist übrigens keine sehr gute Idee, Fehlertexte direkt abzufragen. Ihr Skript käme in schwere Nöte, sollten sich die Fehlermeldungen mal ändern. Diese sind nicht nur von System zu System verschieden, sondern können sich schon auf Ihrem Rechner durch eine neue Version der Programme ändern.

Von daher sollten Sie nur abfragen, ob Fehler auftraten, diese werden ja über den Rückgabewert angezeigt.

Machen wir uns das am Beispiel von Skript 11 nochmals klar. Dazu wollen wir die Aufgabenstellung ein wenig abändern. Das neue Ziel soll es sein, nicht nur die Anzahl an Dateien in beiden Verzeichnissen zu ermitteln, sondern auch die Anzahl an Zeichen, Wörtern und Zeilen in jeder der Dateien. Zusätzlich soll ermittelt werden, wie viele Unterverzeichnisse ignoriert wurden.

Fangen wir mal blauäugig an, denken aber daran, dass `wc` die Dateien im aktuellen Arbeitsverzeichnis sucht oder eine komplette Pfadangabe benötigt. Da wir (noch) nicht in der Lage sind, `wc` mit der kompletten Pfadangabe zu versorgen, setzen wir vor dem Aufruf von `wc` das Arbeitsverzeichnis mittels `cd` (engl. *change directory*) um.


```
# Skript 12: Die Wörter, Zeilen und Zeichen
# in den Dateien aus den
# zwei Verzeichnissen ermitteln
#
tmpfile="/tmp/erg"
ls $1  > $tmpfile
ls $2 >> $tmpfile
echo "Die Verzeichnisse $1 und $2 enthalten `wc -l < $tmpfile Dateien`"
# Ab hier jetzt neu
cd $1
wc `ls`
cd $2
wc `ls`
rm -f $tmpfile
exit 0
```

Ran an den Feind und das Skript aufgerufen. Das Ergebnis ist noch nicht fehlerfrei und je nachdem, ob Sie Optimist oder Pessimist sind, als ermutigend oder niederschmetternd einzustufen. Sollten Sie absolut zufrieden sein, sind Sie entweder sehr genügsam oder beide Verzeichnisse, die dem Skript als Parameter übergeben wurden, haben keine Unterverzeichnisse.

Wenn mindestens ein Unterverzeichnis in einem der angegebenen Verzeichnisse enthalten war, dann haben Sie etwas zu sehen bekommen, welches den folgenden Zeilen ähnelt:

```
...
    243 kapitel2.txt
    218 kapitel3.txt
wc: org: Is a directory
      0 org
wc: skript: Is a directory
      0 skript
    117 toc.txt
...
```

Rufen Sie das Skript noch einmal auf, und leiten Sie die Ausgaben nach `/dev/null` um. Vielleicht gibt das Skript wider Erwarten immer noch etwas aus:

```
buch@koala:/home/buch > ./skript12.sh /tmp/ /home/buch >/dev/null
...
wc: org: Is a directory
wc: skript: Is a directory
buch@koala:/home/buch >
```

Sie ahnen es bestimmt schon, es ist eine Fehlerausgabe. Da diese auf einem eigenen Kanal ausgegeben werden, kann ein Umlenken der Standardausgabe den Meldungen nichts anhaben, und sie erscheinen immer noch auf dem Bildschirm.

`/dev/null` ist ein Gerät (engl. *Device*), das als Datengrab fungiert:

Alles was hier hinein kopiert wird, ist rettungslos verloren und taucht nirgendwo im System wieder auf. Am wichtigsten aber ist: Das Device belegt keinen Speicherplatz für die empfangenen Daten.

Sie sollten diesen Sachverhalt nutzen, wenn Sie Daten probeweise kopieren wollen oder – wie in unserem Fall – wenn Sie bestimmte Ausgaben nicht interessieren. Dies ist wesentlich einfacher, als die Daten wirklich umzukopieren bzw. umzulenken, dadurch eine neue Datei anzulegen und diese danach wieder zu löschen.

Nachdem wir nun wissen, was der praktische Unterschied zwischen Standardausgabe und Standardfehler ist, wollen wir uns dem Thema zuwenden, wie die Fehlerausgabe umgelenkt werden kann.

Dazu rufen wir uns noch einmal die Tabelle über die Nummern der Kanäle in Erinnerung, Sie haben doch nicht geglaubt, ich hätte die nur aufgeführt, um Platz zu schinden, oder?

Bei der Umlenkung können Sie einen Kanal per Nummer ansprechen, indem Sie den umzulenkenden Kanal einfach als Nummer angeben und vor den Zielkanal ein kaufmännisches Und (`&`) stellen.

`&0` ist also die Standardeingabe, `&1` die Standardausgabe und `&2` die Fehlerausgabe. Diese Nummern werden vor oder nach den Umlenkungszeichen angegeben und legen fest, welcher Kanal auf welchen umgelenkt wird.

`2>/dev/null`	Alle Fehlerausgaben werden nach `/dev/null` kopiert.
`2>&1`	Alle Fehlerausgaben werden auf die Standardausgabe umgelenkt.
`>/dev/null 2>&1`	Alle Fehlermeldungen werden auf die Standardausgabe umgeleitet und diese zusätzlich nach `/dev/null`.

Tabelle 2.2: Beispiele für Umlenkungen von Ausgabekanälen

Verbessern wir nun noch einmal das Skript 12. Wir gehen davon aus, dass wc ein Unterverzeichnis gefunden hat, wenn es einen Fehler ausgibt. Dabei ignorieren wir bewusst die Tatsache, dass wc auch Fehler ausgibt, wenn eine Datei wegen fehlender Berechtigungen nicht gelesen werden darf.


```
# Skript 12a: Die Wörter, Zeilen und Zeichen
# in den Dateien aus den zwei Verzeichnissen ermitteln
#
# Parameter $1 $2 entsprechen zwei Verzeichnissen mit
# absoluter Pfadangabe, z.B. /koala/buch oder /tmp
#
tmpfile="/tmp/erg"
ls $1  >$tmpfile
ls $2 >>$tmpfile
echo "Die Verzeichnisse $1 und $2 enthalten `wc -l <$tmpfile` Dateien"
# Ab hier jetzt neu
cd $1
wc `ls` 2>$tmpfile
cd $2
wc `ls` 2>>$tmpfile
anzahl=`wc -l <$tmpfile`
echo "Es waren $anzahl Unterverzeichnisse"
rm $tmpfile
exit 0
```

Um die Sache etwas zu vereinfachen, waren die Erläuterungen zur Umlenkung bewusst etwas ungenau. Am Anfang des Kapitels hatten wir eine Tabelle aufgeführt, in der die Kanalnummern aufgelistet wurden. Im Prinzip führt das Betriebssystem nichts anderes als eine solche Tabelle. Darin wird (u.a.) eingetragen, wohin die Ausgaben gehen. Ohne Umlenkung sähe die Tabelle wie in Abbildung 2.1 aus.

Kanal	Kanalnr.		Gerät
Standardeingabe	0	→	Tastatur
Standardausgabe	1	→	Bildschirm
Standardfehler	2	→	Bildschirm

Abb. 2.1: Die Standardzuordnung von Kanälen zu Ein- und Ausgabegeräten

Nimmt man nun eine Umlenkung vor, so wird der Eintrag des umgelenkten Kanals angepasst.

```
buch@koala:/home/buch > wc verzeichnis datei1 datei2 >liste 2>&1
buch@koala:/home/buch >
```

Das Ergebnis lässt sich folgendermaßen erklären:

Zunächst wird die Standardausgabe in die Datei umgelenkt und danach die Standardfehlerausgabe auf die Standardausgabe. Da diese in die Datei `liste` umgelenkt wird, gilt Gleiches auch für die Fehlerausgabe.

Bitte beachten Sie, dass die Reihenfolge der Umlenkungen in der Bash signifikant ist. Hier lag folgende Umlenkung vor: `>liste 2>&1` (Abbildung 2.2).

Abb. 2.2: Umlenkung der Standardausgabe und des Standardfehlerkanals in eine Datei

Andererseits führt das nächste Beispiel (Abbildung 2.3) dazu, dass nur die Standardausgabe in die Datei umgelenkt wird. Zunächst wird die Fehlerausgabe auf die Ausgabe und diese Ausgabe dann in die Datei umgelenkt.

```
buch@koala:/home/buch > wc verzeichnis datei1 datei2 2>&1 >liste
wc: verzeichnis: Is a directory
buch@koala:/home/buch >
```

Umlenkung: `2>&1 >liste`

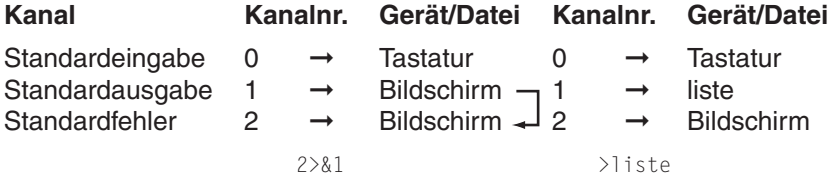


Abb. 2.3: Die Reihenfolge der Umlenkungen ist wichtig

2.2 Pipes

Sie können die Standardein- und -ausgaben nicht nur umlenken, sondern auch die Standardausgabe eines Befehls mit der Standardeingabe eines zweiten Befehls verknüpfen. Dadurch wird die Ausgabe von Befehl 1 die Eingabe von Befehl 2. Diese Technik wird *Pipe* oder *Piping* genannt und ist eine weitere Art der Ein- und Ausgabeumlenkung.

Beschäftigen wir uns also mit den Pipes. Eine Pipe besteht aus (Shell-)Befehlen, die durch | getrennt werden. Dabei werden die Befehle von links nach rechts ausgeführt, und die Standardausgabe des ersten Befehls ist die Stan-

dardeingabe des zweiten Befehls. Dessen Ausgabe wiederum ist die Eingabe des dritten Befehls und so weiter.

Der Exit-Status einer Pipe ist dabei gleich dem Exit-Status des letzten ausgeführten Befehls in der Pipe. Diesen Wert können Sie negieren, indem Sie ein ! vor den ersten Befehl der Pipe stellen. Somit wird ein Status von 0 zu 1 und ein Status von 1 oder größer zu 0.

Auf allen Unixsystemen gibt es so genannte *Pagerprogramme*. Diese nehmen Daten von der Standardeingabe oder einer angegebenen Datei an und geben diese auf dem Bildschirm aus. Werden mehr Daten ausgegeben, als auf eine Bildschirmseite passen, so wartet das Programm auf eine Eingabe und erlaubt so das Durchblättern der Daten.

Solche Programme gibt es viele, und nicht alle Unixsysteme haben den gleichen Pager. Am bekanntesten sind `pg` (uns begegnet unter SCO 3.2 und D-Nix), `more` (auf einem Solaris-System) und `less` (Linux). Im Anhang A und im Kapitel 12 werden wir uns anschauen, wie solche wegen nicht vorhandener oder anders lautender Befehle auftretende Probleme umgangen werden können.

Dennoch stehen die Chancen recht gut, dass Ihr System wenigstens ein `more` hat. Wenn Sie jetzt ein Verzeichnis auflisten wollen und dieses mehr Einträge hat, als Ihr Bildschirm anzeigen kann, dann nutzen Sie eine Pipe und den Pager Ihres Systems:

```
buch@koala:/home/buch > ls -l | more
-rw-r--r--   1 buch     users          52 Feb  5 12:17 anhanga.txt
-rwxrwxrwx   1 buch     users          58 Feb  5 12:14 cds
-rwxr-xr-x   1 buch     users          68 Feb  4 21:12 cds~
-rw-r--r--   1 buch     root         4167 Feb  5 12:34 einleitung.txt
-rw-r--r--   1 buch     users        4275 Feb  2 21:59 einleitung.txt~
-rwxrwxrwx   1 buch     users         126 Feb  4 18:53 ex.sh
-rw-r--r--   1 buch     users       22589 Feb  5 12:12 kapitel1.txt
-rw-r--r--   1 buch     users       22474 Feb  4 20:07 kapitel1.txt~
-rw-r--r--   1 buch     users        1792 Feb  5 12:24 kapitel2.txt
-rw-r--r--   1 buch     users        7206 Feb  4 23:20 kapitel2.txt~
-rw-r--r--   1 buch     users        9075 Feb  4 23:59 kapitel3.txt
drwxr-xr-x   2 buch     users        1024 Jan 30 01:10 org
drwxr-xr-x   2 buch     users        1024 Feb  1 23:41 skript
-rw-r--r--   1 buch     users        3099 Feb  5 11:47 toc.txt
-rw-r--r--   1 buch     users        3096 Feb  4 23:43 toc.txt~
line 1
```

Jetzt können Sie (bei `less` mit den Cursortasten oder bei `more` mit der Leertaste vorwärts und `s` rückwärts) durch die Ausgabe blättern und den Pager mit `q` verlassen. Hier stellt sich die Frage: Was ist passiert?

1. `ls -l` wird ausgeführt und gibt den Verzeichnisinhalt aus.
2. Die Ausgabe geht nicht an den Bildschirm, sondern wegen des | an das Programm `less / more`.
3. `less / more` liest alle Daten zeilenweise aus der Pipe und gibt sie aus. An dieser Stelle findet auch die Eingabe vom Benutzer statt.

Soweit zu den Pagerprogrammen. Sie haben gleich zwei Dinge gelernt: zum einen, wie eine Pipe funktioniert, zum anderen, wie Sie Ausgaben durchblättern können.

Falls Sie sich an die spaltenweise Ausgabe per `ls` gewöhnt haben, werden Sie eine leichte Irritation erleben, wenn Sie `ls | less` aufrufen: Es wird nur eine Spalte ausgegeben. Dies liegt daran, dass `ls` sich anders verhält, wenn es innerhalb einer Pipe aufgerufen wird.

Kommen wir zurück zu den Pipes und stellen uns folgende Frage: Wie kann ich aus einer gegebenen Datei die x-te Zeile anzeigen? Um diese Aufgabe zu bewältigen, benötigen wir noch zwei weitere Befehle:

```
tail -<nr> <datei>
head -<nr> <datei>
```

Dabei gibt `tail` die letzten `<nr>` Zeilen der Datei namens `<datei>` aus, während `head` die ersten `<nr>` Zeilen der Datei liefert.

Um die x-te Zeile aus einer Datei anzuzeigen, lassen wir uns im ersten Schritt die ersten x Zeilen der Datei ausgeben. Von diesen x Zeilen ist nur die letzte Zeile für uns interessant. Mit Pipes könnte die Lösung so aussehen:


```
# Skript 14: Pipes, die erste
# Parameter 1: Dateiname
# Parameter 2: Zeilennummer der Zeile, die angezeigt werden soll.
#
head -$2 $1 | tail -1
exit 0
```

Lassen Sie uns dieses Skript um eine Komponente erweitern: Das Skript soll ausgeben, wie die Zeile lautet und dann, wieviel Zeichen und Wörter in dieser Zeile stehen. Nichts einfacher als das! Sie fügen einfach eine Zeile ein:

```
...
head -$2 $1 | tail -1 | wc
exit 0
```

Wunderbar, es klappt. Aber haben Sie mal eine wirklich große Datei zum Testen genommen und versucht, eine Zeile vom Ende auszugeben. Ja? Hat Ihnen der Kaffee gemundet, den Sie in der Zwischenzeit aufgesetzt und getrunken haben? Nun, das war sicherlich sehr stark überspitzt, aber schnell war die Lösung bestimmt nicht. Das liegt daran, dass das Skript die betroffene Zeile zweimal aus der Datei heraussuchen muss, und das dauert umso länger, je größer die Datei ist.

Eine Lösung wäre es, die Zeile in eine Variable einzulesen und dann mit dieser zu arbeiten. Obwohl elegant und schneller, hat sie doch einen entscheidenden Nachteil: Sie gefällt uns nicht `:-)`

Wir möchten Ihnen nämlich noch einen weiteren Unixbefehl ans Herz legen: `tee` (der Buchstabe T ist im Englischen eine Abkürzung für Abzweig). Der Befehl `tee` erwartet einen Dateinamen als Parameter, nimmt die Standardeingabe, gibt die Eingabe unverändert auf der Standardausgabe wieder aus und kopiert die Standardeingabe gleichzeitig in die angegebene Datei.

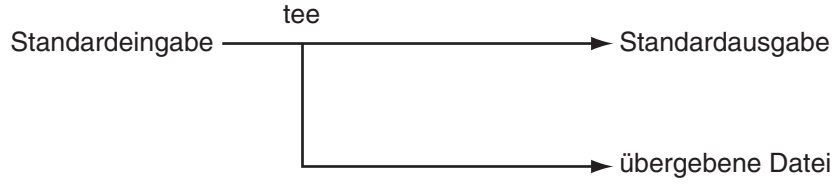


Abb. 2.4: Die Funktionsweise von tee

Also könnte die Lösung so aussehen:


```
# Skript 14: Pipes, Version 3
# Parameter 1: Dateiname
# Parameter 2: Zeilennummer der Zeile, die angezeigt werden soll.
#
tmpfile=/tmp/tmpout
echo "Die Zeile lautet"
head -$2 $1 | tail -1 | tee $tmpfile
echo "und hier die Zeilenstatistik"
wc <$tmpfile
rm -f $tmpfile
exit 0
```

Was den Exit-Status der Pipe betrifft, hatte ich bereits erwähnt, dass dieser identisch mit dem Exit-Status des letzten Befehls aus der Pipe ist. Riskieren wir daher noch einmal einen kurzen Blick auf die Ein- und Ausgabeumlenkung innerhalb einer Pipe.

Auch innerhalb einer Pipe gelten die gleichen Regeln zur Umlenkung wie bei einzelnen Befehlen. Die Umlenkung kann dabei für jeden einzelnen Befehl festgelegt werden.

Was passiert aber, wenn ein Fehler in der Pipe auftritt? Indirekt wurde die Frage schon beantwortet: Der *fehlerhafte* Befehl in der Pipe gibt den Fehler auf die Fehlerausgabe aus, und der nächste Befehl (in der Pipe) wird ausgeführt.

2.3 Wildcards/Ersatzmuster

Ersatzmuster (engl. *Wildcards*) sind Zeichen oder Zeichenfolgen innerhalb eines Worts, welche die Shell nach bestimmten Regeln durch andere Zeichenketten ersetzt. Vereinfacht gesagt verhalten sich Ersatzmuster ähnlich wie Joker im Kartenspiel Rommee: Ein Joker ersetzt dort eine beliebige Karte. Ein Ersatzmuster steht an Stelle eines (oder mehrerer) anderen Zeichens innerhalb eines Wortes.

2.3.1 Allgemeines

Woher kommen die Zeichenketten, die für die Ersatzzeichen eingesetzt werden?

Für die Shell ist ein Wort mit Ersatzzeichen nichts anderes als eine Schablone. Die Shell schaut also nach, welche Dateinamen der vorgegebenen Schablone entsprechen und ersetzt die Schablone durch *alle* Dateinamen, die ins Raster passen. Dadurch wird in der Regel aus einem Wort (also der Schablone) eine Liste von Dateinamen.

Im Folgenden möchte ich Ihnen die Wirkung der einzelnen Wildcards anhand eines fiktiven Verzeichnisinhalts klarmachen. Den `echo`-Befehl kennen Sie ja schon. Er ist gut geeignet, um Wildcards auszuprobieren. Wir nutzen ihn hier, um uns bestimmte Teilmengen aus der Menge der verfügbaren Dateinamen anzuzeigen.

So sieht das Verzeichnis ohne versteckte Dateien aus:

```
buch@koala:/home/buch > ls
Anhangb.txt    Kapitel5.txt   cw              gk.h           toc.txt
Kapitaen       Tst            einleitung.txt  kapitel1.txt   toc.txt~
Kapitel3.txt   anhanga.txt    gk.a            kapitel2.txt   tst
Kapitel4.txt   cds            gk.c            kapitel3.txt
buch@koala:/home/buch >
buch@koala:/home/buch > echo *
Anhangb.txt Kapitaen Kapitel3.txt Kapitel4.txt Kapitel5.txt Tst anhanga.txt
cds cw einleitung.txt gk.a gk.c gk.h kapitel1.txt kapitel2.txt kapitel3.txt
toc.txt toc.txt~ tst
buch@koala:/home/buch >
```

Wie Sie sehen, können Sie ein einfaches `ls` auch mit `echo` erreichen. Wichtiger ist aber die Erkenntnis, dass * als Wildcard zunächst durch passende Dateinamen ersetzt wird. Die Liste aller so erhaltenen Dateinamen wird dann als Parameter an den aufgerufenen Befehl übergeben (hier `ls` bzw. `echo`).

Die Beispiele aus den folgenden Abschnitten werden sich immer auf dieses Verzeichnis beziehen. Falls Sie die Sonderbedeutung der Ersatzzeichen aufheben wollen, denken Sie an Quoting und Fluchtzeichen.

2.3.2 Ein Zeichen ersetzen: »?«

Dieses Ersatzzeichen kennen Sie sicherlich noch von MS-DOS. Es ersetzt genau ein beliebiges Zeichen. Wenn Sie also nur Informationen über `kapitel1.txt`, `kapitel2.txt` und `kapitel3.txt` erhalten wollen, geben Sie Folgendes ein:

```
buch@koala:/home/buch > ls -l kapitel?.txt
-rw-r--r--   1 buch     users       22588 Feb  7 00:38 kapitel1.txt
-rw-r--r--   1 buch     users       23768 Feb  9 19:30 kapitel2.txt
-rw-r--r--   1 buch     users        9075 Feb  6 17:47 kapitel3.txt
buch@koala:/home/buch > echo kapitel?.txt
kapitel1.txt kapitel2.txt kapitel3.txt
buch@koala:/home/buch >
```

Wird das Zeichen `?` in der Schablone genutzt, so muss an dessen Stelle genau ein Zeichen existieren, damit die Schablone passt. Ein `ls toc.txt?` gibt nur toc.txt~ zurück, jedoch nicht toc.txt !

2.3.3 Eine beliebige Anzahl an Zeichen ersetzen: »*«

Auch noch bekannt und beliebt aus MS-DOS Zeiten ist »*«. Es ersetzt eine beliebige Anzahl von Zeichen. Während »?« zwingend durch ein existierendes Zeichen aus dem Dateinamen ersetzt werden muss, kann »*« 0 bis X Zeichen ersetzen. X steht dabei für die maximale Dateinamenlänge auf Ihrem System.

Nehmen wir nochmals das letzte Beispiel aus Abschnitt 2.3.1. Beide Dateinamen – `toc.txt` und `toc.txt` – erhalten Sie durch `ls toc.txt*`.

Unter DOS wurde die Bedeutung von »*« durch ein »`.`« aufgehoben, weil das »*« am Ende der Dateinamen und der Extension durch die entsprechende Anzahl an Fragezeichen »?« ersetzt wurde. Daher führte nur die Angabe von `dir *.*` zur Ausgabe aller Dateien im aktuellen Verzeichnis. Dies gilt nicht für die Bash, weshalb ein `ls *` den gleichen Effekt hat.

2.3.4 Zeichenbereiche angeben

Die Ersatzmuster aus den Abschnitten 2.2 und 2.3 sollten in 90% aller Fälle für eine Bestimmung der Dateinamen ausreichen. Manchmal ist es aber wünschenswert, genau zu definieren, welche Zeichen für eine Ersetzung in Frage kommen, anstatt einfach ? anzugeben. Dafür gibt es den Zeichenbereich. Das Ersatzmuster wird durch eckige Klammern eingerahmt. Alle Zeichen, die innerhalb der Klammern stehen, werden als gültige Zeichen für die Ersetzung akzeptiert, alle anderen ignoriert.

```
buch@koala:/home/buch > ls -l kapitel[12].txt
-rw-r--r--   1 buch     users       22588 Feb  7 00:38 kapitel1.txt
-rw-r--r--   1 buch     users       25023 Feb  9 20:05 kapitel2.txt
buch@koala:/home/buch > echo kapitel[12].txt
kapitel1.txt kapitel2.txt
buch@koala:/home/buch >
```

Schon besser, aber auch das kann in ganz schöne Tipparbeit ausarten, wenn man alle Zeichen von a bis z als gültigen Ersatz haben will. Bevor Sie alle 26 Zeichen plus die Klammern eingeben, ignorieren Sie sicher lieber die paar zusätzlichen Treffer, die das ?-Ersatzzeichen möglicherweise mit sich bringt.

Unixbenutzer sind aber tippfaule Menschen (schauen Sie nur mal nach den Befehlsnamen, die wir bis jetzt eingesetzt haben: `wc`, `cd`, `pwd`, `pg`, `ls`, `tail`). Also gibt es eine elegantere Methode:

```
buch@koala:/home/buch > ls -l Kapitel[1-3].txt
-rw-r--r--   1 buch     users       22588 Feb  7 00:38 Kapitel3.txt
-rw-r--r--   1 buch     users       22588 Feb  7 00:38 kapitel1.txt
-rw-r--r--   1 buch     users       25023 Feb  9 20:05 kapitel2.txt
-rw-r--r--   1 buch     users        9075 Feb  6 17:47 kapitel3.txt
buch@koala:/home/buch >
```

Es geht aber noch besser. Mal angenommen, Sie wollen alle Dateien ausgeben, deren Namen Zeichen enthalten, die *nicht* in dem von Ihnen definierten Bereich vorkommen. Sie können jetzt den Bereich entsprechend anpassen (viel Spaß beim Tippen), oder Sie negieren den angegebenen Bereich, indem Sie als erstes Zeichen innerhalb des Bereichs ein ^ oder ein ! angeben:

```
buch@koala:/home/buch > echo ?apitel[!1-3].txt
Kapitel4.txt
Kapitel5.txt
buch@koala:/home/buch >
```

Sie können natürlich auch mehrere Bereiche in den eckigen Klammern festlegen: `[a-df-z1]` enthält z.B. alle Zeichen von a bis d und von f bis z sowie die 1.

Wenn Sie den Abschnitt aufmerksam mitverfolgt haben, so dürften Ihnen ein paar Probleme aufgefallen sein, die sich aus der Sonderbehandlung der Zeichen], ^, ! und – ergeben.

- Wenn Sie in Ihrem Zeichenbereich ein ^ oder ! benötigen, stellen Sie es frühestens an die zweite Stelle innerhalb der Klammern. Nur an der ersten Stelle negieren diese Zeichen den angegebenen Zeichenbereich.
- Wenn Sie eine] in Ihrem Zeichenbereich benötigen, stellen Sie es direkt hinter die [oder hinter das Zeichen für Negierung.
- Falls Ihr Zeichenbereich ein - beinhaltet, stellen Sie dieses als erstes oder letztes Zeichen in die Klammern.

Tabelle 2.3: Sonderfälle beim Angeben von Bereichen

Zeichenbereich	Notation
]	[]]
alle außer]	[^]]
a-z und ! ^	[a-z!^]
a-z und -	[a-z-]

Zum Abschluss bringen wir Ihnen wie üblich die notwendigen Fachbegriffe und ein ganz klein wenig Theorie bei:

- Das Auswerten der Wildcards wird *globbing* genannt.
- Das *globbing* sortiert die Ausgabe, die *Brace Extension* macht das nicht (siehe auch den nächsten Abschnitt zur Brace Extension).
- Versteckte Dateien, d.h. mit ».« beginnende Dateinamen, werden nicht von den Ersatzmustern erfasst.

Falls Sie versteckte Dateien berücksichtigen wollen, ohne explizit ein ».*« o.ä. anzugeben, müssen Sie für die Bash die Option *glob_dot_filenames* setzen. Dies können Sie durch `shopt -s dotglob` erreichen. Deaktivieren lässt sich die Option wieder durch `shopt -u dotglob`.

Ursprünglich eine Funktionalität der Ksh, bietet die Bash seit der Version 2.05 ebenfalls die Möglichkeit, vordefinierte Zeichenmengen als Parameter für die in diesem Kapitel beschriebenen Ersatzmuster zu nutzen. Es ist ja häufig so, dass Muster auf das Vorkommen von Zahlen, Buchstaben (kleine, große oder auch beide Sorten) prüfen sollen. So ist die wiederholte Angabe von [a-zA-Z] doch etwas lästig. Deshalb bieten Bash und Kornshell folgendes Muster als Ergänzung an: [:alpha:]. Da [:alpha:] aber nur innerhalb der Klammern [] erkannt wird, sieht die korrekte Syntax wie folgt aus:

```
buch@koala:/home/buch > ls kapitel[[:digit:]]*
-rw-r--r--   1 buch     users       22588 Feb  7 00:38 kapitel1.txt
-rw-r--r--   1 buch     users       23768 Feb  9 19:30 kapitel2.txt
-rw-r--r--   1 buch     users        9075 Feb  6 17:47 kapitel3.txt
buch@koala:/home/buch >
```

Die folgende Tabelle bietet eine Zusammenstellung der wichtigsten Zeichenmengen an:

Name	einige Zeichenmengen in Bash (ab Version 2.05)
`alnum`	Alle Buchstaben, Unterstrich und Ziffern
`alpha`	Groß- und Kleinbuchstaben
`digit`	Ziffern
`lower`	Kleinbuchstaben
`print`	druckbare Zeichen
`space`	Leerzeichen, Tabulatoren etc. (Whitespace)
`upper`	Großbuchstaben

2.4 Brace Extension – Erweiterung durch Klammern

Der letzte Abschnitt hat sich ausführlich mit Wildcards beschäftigt, die ein oder mehrere Wörter abhängig von Dateinamen erstellen. Dies ist durchaus nützlich, aber die Beschränkung auf Dateinamen kann manchmal ärgerlich sein. Deshalb bietet die Bash eine Erweiterung auf Basis von geschweiften Klammern. Die dabei entstehenden Wörter müssen nicht als Dateinamen vorliegen.

Eine gültige *Brace Extension* muss mindestens eine { und eine } enthalten sowie mindestens ein Komma »`,`« zwischen beiden Klammern (engl. *Braces*). Sollte eine Brace Extension diesen simplen Bedingungen nicht genügen, so werden die Zeichen unverändert übernommen. Vor dem Klammerausdruck kann ein Präfix angegeben werden und hinter dem Klammerausdruck ein Suffix. Vor jeder Zeichenkette, innerhalb der Klammern, wird das Präfix gesetzt und dahinter wird das Suffix angehängt.

Beispiel:

`Mar{c,k}us` ergibt:

»Markus« und »Marcus«

Ist doch simpel! Also schalten wir noch einmal einen Gang hoch. Die Zeichenketten innerhalb der {} werden wie bereits gesagt durch Kommata getrennt.

Sie dürfen aber:

- Geschweifte Klammern – und damit Brace Extensions – schachteln.
- Die Wildcards aus Abschnitt 2.3 innerhalb der geschweiften Klammern nutzen.

Daher würde ein `mkdir /home/buch/{old{.src,.txt},new{.src,.txt}}` die Verzeichnisse `/home/buch/old.src`, `/home/buch/old.txt`, `/home`, `/buch/new.src` und `/home/buch/new.txt` anlegen. Das sieht zugegebenermaßen etwas wild aus, ist aber einfach.

Die Brace Extension führt im Übrigen zu einer leichten Inkompatibilität zur `sh`:

Die `sh` erweitert geschweifte Klammern innerhalb eines Dateinamens nicht, sondern reicht die Zeichen eins zu eins durch. Deshalb führt die Angabe von `Datei{1,2}` nicht zu den Wörtern »Datei1« und »Datei2«, sondern zu »Datei{1,2}«.

2.5 Aufgaben

1. Gibt es eventuell noch eine andere Lösung für Skript 14, das die gleichen Befehle nutzt?
2. Wie können Sie folgende Ausgabe durch eine Brace Expansion erreichen?

   ```
   Austria Australien Australia
   ```

   Tipp: Ein `echo` genügt! (Aber bitte nicht `echo Austria Australien Australia`.)
3. Schreiben Sie ein Skript, welches zwei Parameter annimmt und dann per `echo` Folgendes ausgibt:

   ```
   buch@koala:/home/buch > ./skriptauf.sh 1 2
   Von 1 bis z Von 2 bis z
   ```

   Dabei sollte die Brace Extension genutzt werden.

4. Zum Abschluss dieses Kapitels haben Sie sich etwas Humor auf Kosten der Shell verdient. Macht absolut keinen Sinn, bringt aber ein Lächeln aufs Gesicht. Geben Sie einmal in der Bash Folgendes ein:

   ```
   buch@koala:/home/buch > set -H
   buch@koala:/home/buch > ^Was ist Süßstoff?
   ```

2.6 Lösungen

1. Das Skript könnte folgendermaßen aussehen:


```
# Skript 14: Pipes, die zweite
# Parameter 1: Dateiname
# Parameter 2: Zeilennummer der Zeile, die angezeigt werden
# soll.
#
# Stellt die Kombination head | tail  um auf
# tail | head
#
total=`wc -l <$1`
vonunten=`expr $total - $2 + 1`
tail -$vonunten $1 | head -1
exit 0
```

2. Die Ausgabe unter Benutzung von Brace Extension erreichen Sie durch den Befehl

   ```
   echo Austr{ia,alien,alia}
   ```

3. Das Skript könnte so aussehen:

   ```
   # skriptauf.sh
   # Brace Expansion
   echo "Von "{$1,$2}" bis z"
   ```

4. Was ist passiert? In der Bash gibt es eine History (deutsch: *Verlauf*), die mittels Cursortasten das Editieren bereits eingegebener Befehle erlaubt. Diese aktivieren wir mittels `set -H`. Für die Shell bedeutet `^str^str2^`, dass die letzte eingegebene Zeile nochmals ausgeführt werden soll, wobei `str` durch `str2` ersetzt werden soll. Da aber das zweite ^ fehlt, kann die Bash keine Ersetzung vornehmen und gibt den Fehler aus.

Kapitel 3

Abfragen und Schleifen

»Gewalt löst keine Probleme« –
Dschingis Khan

... und das gilt nicht nur im wahren Leben, sondern auch bei der Shellprogrammierung. Auch hier ist es nötig, flexibel auf Anforderungen zu reagieren, die an Ihr Skript gestellt werden. Ihre bisherigen Skripten waren ganz nett (Entschuldigung: brillant), aber doch sehr linear. Das Skript führte Ihre Anforderung von der ersten bis zur letzten Zeile aus, eine Änderung des Ablaufs war nur durch eine Änderung des Skripts möglich.

Dies ist kein haltbarer Zustand, und solche Skripten sind nicht viel besser als Handarbeit, die Sie ja gerade durch Skripten reduzieren wollen. Fazit: Sie brauchen Abfragen und Schleifen. Abfragen verzweigen innerhalb des Skripts abhängig von verschiedenen, von Ihnen definierten Bedingungen und führen so unterschiedliche Skriptabschnitte aus.

Schleifen führen dazu, dass bestimmte Abschnitte innerhalb eines Skripts wiederholt werden, solange eine Bedingung zutrifft. Soweit zur Theorie, schreiten wir zur Tat.

3.1 Der test-Befehl

Der `test`-Befehl ist zunächst einmal keine Abfrage und auch keine Schleife. Er ist dennoch so wichtig, dass dieses Kapitel ohne `test` so gut wie sinnlos ist. Wie bereits erwähnt, testen sowohl Schleifen als auch Abfragen Bedingungen und tun etwas, wenn diese Bedingung wahr ist. Wahr bedeutet für ein Skript, dass die Bedingung eine `0` zurückgibt. Falsch ist identisch mit dem Rückgabewert von 1 für die geprüfte Bedingung. Die Prüfung von Bedingungen ist die Aufgabe von `test`, und konsequenterweise gibt `test` entweder `0` oder `1` zurück.

Falls Sie schon Erfahrungen im Umgang mit Sprachen wie C haben, wird Sie das verwundern; schließlich ist dort die interne Darstellung von Wahr und Falsch genau umgekehrt. Halten Sie sich aber bitte vor Augen, dass dies in der Shell nur konsequent ist, wird doch ein Rückgabewert von `0` als OK (Wahr) interpretiert.

`test` kann auf zwei verschiedene Weisen aufgerufen werden. Unterschiede in der Funktion gibt es jedoch nicht:

`[ <Bedingung> ]` oder `test <Bedingung>`

`<Bedingung>` ist ein gültiger Ausdruck, der sich aus mehreren Teilausdrücken zusammensetzen kann. Tabelle 3.1 zeigt einige gängige Möglichkeiten auf.

Tabelle 3.1: Gültige Teilausdrücke zur Verwendung in Bedingungen

Ausdruck	Beispiel	Erklärung
-d verzeichnis	`[-d /tmp ]`	Ist wahr, wenn die Datei existiert und ein Verzeichnis ist.
-f datei	`[ -f math.txt ]`	Ist wahr, wenn die Datei existiert und eine normale Datei ist (also kein Device oder Verzeichnis).
-r datei	`[ -r math.txt ]`	Existiert diese Datei, und erlaubt sie dem Skript mindestens den Lesezugriff?
-w datei	`[ -w txt.txt ]`	Ist wahr, wenn die Datei existiert und den Schreibzugriff erlaubt.
-x datei	`[ -x skript.sh ]`	Ist wahr, wenn die Datei existiert und ausgeführt werden kann.
-z string	`[ -z "$anz" ]`	Ist wahr, wenn der Parameter eine Zeichenkette der Länge `0` ist, also "".
-n string	`[ -n "$anz" ]`	Ist wahr, wenn die übergebene Zeichenkette nicht leer ist.

Ausdruck	Beispiel	Erklärung
str1 = str2	`[ "A" = "a" ]`	Wahr, wenn beide Zeichenketten identisch sind. Das Ergebnis dieses Beispiels ist übrigens falsch, da zwischen Groß- und Kleinschrift unterschieden wird.
z1 -eq z2	`[ 1 -eq 0 ]`	Ist wahr, wenn die erste Zahl gleich der zweiten ist. Im Gegensatz zu = vergleicht `-eq` die Werte arithmetisch:
		So ist folgende Bedingung wahr: `[ "1" -eq "0001" ]`, während `[ "1" = "0001" ]` die Zeichenkette Zeichen für Zeichen vergleicht und somit Falsch zurückgibt. `-eq` steht übrigens für *equal* (Gleichheit).
z1 -lt z2	`[ 1 -lt 2 ]`	Ist wahr, wenn die erste Zahl kleiner ist als die zweite (Beispiel ist wahr). `lt` ist kurz für *less than* (kleiner als).
z1 -le z2	`[ 1 -le 1 ]`	Abfrage auf kleiner oder gleich (*less or equal*).
z1 -ne z2	`[ 1 -ne 1 ]`	Abfrage auf Ungleichheit (*not equal*).
z1 -gt z2	`[ 1 -gt 0 ]`	Abfrage auf größer (*greater than*).
z1 -ge z2	`[ 1 -ge 2 ]`	Abfrage auf größer oder gleich (*greater or equal*)
! ausdruck	`[ ! 1 -eq 2 ]`	Ist wahr, wenn der Ausdruck falsch ist (`1 -eq 2` ist falsch, somit ist `! 1 -eq 2` wahr).
aus1 -a aus2	`[ 1 -eq 1 -a 2 -gt 1 ]`	UND-Verknüpfung. Ist wahr, wenn Ausdruck 1 und Ausdruck 2 wahr sind. Das Beispiel ist wahr.
aus1 -o aus2	`[ 1 -eq 2 -o 2 -gt 1 ]`	ODER-Verknüpfung. Ist wahr, wenn Ausdruck 1 oder Ausdruck 2 wahr ist. Somit ist das Ergebnis des Beispiels wahr.

Es ist auch erlaubt, Klammern `()` einzusetzen, um die Hierarchie der Teilausdrücke zu verändern. Da die Klammern aber eine Bedeutung für die Shell haben, müssen sie mit Fluchtzeichen versehen werden `\(` bzw. `\)`.

Eine komplette Liste aller unterstützten Ausdrücke finden Sie unter `man test`. Übrigens ist `test` bzw. [in der Bash und der Kornshell ein eingebauter Befehl (engl. *Builtin*). Für Shells, die dies nicht bieten, findet sich in `/usr/bin/` das Programm `test`. [liegt im gleichen Verzeichnis, ist aber nur ein Link (Verweis) auf `test`.

Wenn Sie sehen wollen, welches Programm aufgerufen wird, so bieten Bash und Kornshell den eingebauten Befehl `type` an. Unter Berücksichtigung der `PATH`-Variablen gibt `type` aus, wo das Programm gefunden wurde. Die Bash bietet den Parameter `-a`, der alle Vorkommen des übergebenen Programmes innerhalb des Pfads ausgibt.

Den Parameter `-a` kennt die Kornshell nicht, sie bietet dafür den Parameter `-p`, welcher das erste Vorkommen des gesuchten Programms im Pfad ausgibt. Builtins werden bei `-p` ignoriert:

In der Bash:

```
buch@koala:/home/buch > type -a test
test is a shell builtin
test is /usr/bin/test
buch@koala:/home/buch >
```

In der Kornshell:

```
buch@koala:/home/buch > type -p test
test is /usr/bin/test
buch@koala:/home/buch > type test
test is a shell builtin
buch@koala:/home/buch >
```

3.2 Die if-Abfrage


```
if <befehl1> ; then <befehl2> ;
[ elif <befehl3>; then <befehl4> ; ] ...
 [ else              <befehl5> ; ]
fi
```

Die `if`-Abfrage führt `<befehl1>` aus und testet dessen Rückgabewert. In der Regel ist das der Befehl `test`, es kann aber jeder beliebige andere Befehl, eine Liste von Befehlen (durch Semikola **;** getrennt), Befehlsgruppen oder gar eine Pipeline sein. Über die Theorie von Befehlslisten und -gruppen informieren Sie spätere Kapitel. Ist der Rückgabewert gleich `0`, wird `<befehl2>` ausgeführt und der nächste Befehl nach dem abschließenden `fi` ausgeführt. Falls der Rückgabewert `1` war, so schaut Bash nach,

- ob ein `elif` (`else if`) angegeben wurde. Wenn ja, so wird `<befehl3>` ausgeführt und dessen Rückgabewert getestet. Ist dieser `0`, so wird `<befehl4>` ausgeführt und der `if`-Befehl verlassen. Ist der Rückgabewert von `<befehl3>` gleich `1`, macht die Bash weiter beim nächsten `elif`.

- ob ein `else` angegeben wurde, wenn kein weiteres `elif` gefunden wurde. Ist das der Fall, wird der `<befehl5>` ausgeführt und der `if`-Block verlassen.
- Wenn weder `elif` noch `else` angegeben wurde, wird nichts weiter gemacht und der `if`-Block verlassen. Das heißt, dass der nächste Befehl nach dem abschließenden `fi` aufgerufen wird.

Jedes Skript sollte testen, ob die benötigten Parameter beim Aufruf übergeben wurden, im negativen Falle eine Meldung ausgeben und sich beenden. Mit Hilfe des `if`-Befehls und des `$#`-Parameters kann eine solche Abfrage realisiert werden:

```
...
if [ $# -ne 3 ] ; then
   echo "Usage: $0 datei ab bis" 1>&2
   echo "   Gibt die <anz> Zeilen ab Zeile <ab> der Datei <datei> aus" 1>&2
   echo "    datei  -> Datei, aus der die Zeilen angezeigt werden
        sollen" 1>&2
   echo "    ab     -> Die Zeile der <datei>, ab der angezeigt werden
        soll" 1>&2
   echo "    bis    -> Bis zur wievielten Zeile soll die Datei ausgegeben
        werden?" 1>&2
   exit 1
fi
# Hier geht es weiter, wenn drei  Parameter angegeben wurden.
...
```

Das zu schreibende Skript erwartet also drei Parameter: einen Dateinamen, eine Zeilennummer, ab der, und eine Zeilennummer, bis zu der angezeigt werden soll. Als Grundlage zu diesem Skript bietet sich die erste Version von Skript 14 aus Kapitel 2 an. (`$#` enthält die Anzahl an übergebenen Parametern.)


```
# Skript 15: definierten Zeilenbereich ausgeben
# Version 1
if [ $# -ne 3 ] ; then
   echo "Usage: $0 datei ab bis"  1>&2
   echo "   Gibt die Zeilen <ab> bis Zeile <bis> der Datei <datei> aus" 1>&2
   echo "    datei  -> Datei, aus der die Zeilen angezeigt werden sollen" 1>&2
   echo "    ab     -> Zeile, ab der angezeigt werden soll (mindestens 1)" 1>&2
   echo "    bis    -> Bis zur wievielten Zeile (inklusive) ausgeben?"   1>&2
   exit 1
fi
# Variablen zuweisen
datei=$1
ab=$2
bis=$3
#
# Berechne die letzte Zeile, die mit head auszugeben ist:
# anz = bis - ab + 1
#
```

```
anz=`expr $bis - $ab + 1`
#
# Anzeigen der gewuenschten Zeilen
#
head -$bis "$datei" | tail -$anz
exit 0
```

Der Dateiname steht in Anführungszeichen, damit es keine Probleme mit so netten Dateinamen wie `"Christa Wieskotten.txt"` gibt. Ohne Gänsefüßchen würde die Shell dies als zwei eigenständige Wörter interpretieren, und das Skript würde mit einem Fehler enden. Am besten, Sie gewöhnen es sich gleich an, Dateinamen aus diesem Grunde immer in Anführungszeichen zu setzen.

Dieses Skript hat eine Fehlerquelle ausgeschaltet und gibt dem Benutzer im Fehlerfall ein paar rudimentäre Informationen an die Hand. Leider ist das Skript dadurch alles andere als perfekt geworden. Einige wesentliche Probleme existieren immer noch:

- Es wird nicht geprüft, ob die Datei existiert bzw. nicht evtl. ein Verzeichnis oder Gerätetreiber ist.
- Der Zahlenbereich wird nicht auf Plausibilität geprüft: Es ist durchaus möglich, eine `<ab>`-Zeile anzugeben, die größer ist als die letzte Zeile in der Datei. Auch ein `-1` als Wert für `<ab>` wird nicht verhindert, was zu einem `head: --1 illegal option` führen würde.

Wenn Sie einen Blick auf die Tabelle zum `test`-Befehl werfen, werden Sie feststellen, dass das erste Problem recht einfach zu lösen ist. Die Option `-f` testet ab, ob das übergebene Wort eine normale Datei ist. Um es etwas schwieriger zu machen, sollte das Skript ausgeben, ob fehlerhafterweise ein Verzeichnis oder ein Gerätetreiber übergeben wurde. Folgende Abfrage sollten Sie nach der Überprüfung der Parameteranzahl einfügen:

```
...
# Ist es eine normale Datei?
if [ ! -f "$1" ] ; then
  if [ -d "$1" ] ; then
    # Nein, es ist ein Verzeichnis
    # Fehler auf die Fehlerausgabe
    echo "Verzeichnis angegeben" >&2
    exit 1
  elif [ -c "$1" ] ; then
    # Ein Device
    echo "Device angegeben" >&2
    exit 1
  else
    echo "Ungültiger Dateityp" >&2
```

```
    exit 1
  fi
 else
  # Ist eine Datei
  datei=$1
fi
...
```

Jetzt, da wir langsam in Schwung kommen, wird es Zeit für zwei weitere Unixbefehle: `file <datei>`

Der Befehl `file` versucht zu ermitteln, welches Datenformat sich hinter dem Dateinamen verbirgt. So gibt die GNU-Version von `file` für GNU-tar-Archive `GNU tar archive` aus. Andere, weniger effektive Betriebssysteme versuchen, eine ähnliche Funktionalität über die Zuordnung von dreistelligen Dateiendungen zu erreichen (und scheitern kläglich, wenn Sie beispielsweise eine WAV-Datei von `.wav` in `.txt` umbenennen und doppelt draufklicken).

Die Zuordnung, wie `file` die Datenformate erkennt, wird in der Datei `/etc/magic` nachgehalten. Eine Beschreibung dieser Datei ginge über das Ziel des Buches weit hinaus. Hier nur so viel:

In dieser Datei wird festgelegt, an welcher Stelle in einer Datei bestimmte konstante Daten stehen müssen, und welcher Text von `file` dann ausgegeben wird. Diese Regeln beziehen sich auf den *Inhalt* der Dateien und sind somit unabhängig von Dateiendungen. Perfekt sind aber auch sie nicht, aber treffsicherer als Dateiendungen.

Leider ist der Inhalt von `/etc/magic` nicht auf allen Unixsystemen identisch. Das führt dazu, dass `file` nicht unbedingt auf allen Systemen die gleichen Texte für die gleichen Datenformate ausgibt. In letzter Konsequenz bedeutet dies, dass Skripten, die Ausgaben von `file` überprüfen, auf anderen Systemen möglicherweise nicht laufen. Das weiter unten folgende Skript ist aber für ein größeres Projekt, den CW-Commander, gedacht, das wir später im Buch in Angriff nehmen werden. Daher verletzen wir diese goldene Regel in diesem Fall einmal.

Der zweite Befehl ist der Unix Tape Archiver `tar`, der oben bereits kurz erwähnt wurde. Er hat jede Menge Optionen, uns interessieren aber nur drei:

```
tar cvf <datei> <dateiliste>
tar tvf <datei>
tar xvf <datei>
```

Die erste Version erstellt (überschreibt falls nötig) eine Archivdatei namens `<datei>` und kopiert rekursiv alle Dateien hinein, die in `<dateiliste>` aufgeführt werden.

Der zweite Aufruf gibt den Inhalt des Archivs `<datei>` aus, und der dritte kopiert die Dateien aus dem Archiv ins aktuelle Arbeitsverzeichnis. Sollte im dritten Fall noch eine Dateiliste hinter dem Dateinamen des Archivs angegeben worden sein, so werden nur die Dateien aus dem Archiv kopiert, die in dieser Liste aufgeführt wurden.

Was fangen wir nun aber mit diesen Befehlen an? Lassen Sie uns noch ein Skript schreiben, welches einen Parameter erwartet und versucht, dessen Datentyp zu erkennen. Falls es ein Tar-Archiv sein sollte, so geben wir dessen Inhalt aus, ansonsten nur die Ausgabe von `file`.

Testen wir zunächst also die Anzahl der Parameter:


```
# Skript 16: file
# Dateitypen ermitteln. Erste Annäherung
if [ $# -ne 1 ] ; then
   echo "Dateiname erwartet" 1>&2
   exit 1
fi
datei=$1
if [ ! -f "$datei" ] ; then
   echo "Datei nicht gefunden" 1>&2
   exit 1
else
   if [ ! -r "$datei" ] ; then
      echo "Datei nicht lesbar!" 1>&2
   fi
fi
# Noch nicht fertig!
```

Die Fehlermeldungen dürfen Sie gern etwas verbessern, wir haben sie wegen der Übersichtlichkeit (nein, wir sind nicht tippfaul!!! ;)) kurz gehalten.

Im nächsten Schritt versuchen wir, das Dateiformat zu ermitteln und das Ergebnis in einer Variablen einzulesen. Allerdings haben wir ein kleines Problem: Da `file` bei der Ausgabe immer den Dateinamen zuerst ausgibt (im Falle unseres Beispiels gefolgt von `GNU tar archive`), können wir nicht einfach eine Konstante abfragen. Hilfe naht in Gestalt von Parameter `$1`, den wir ja in der Variablen `datei` abgespeichert haben.

```
dattyp=`file $1`
if [ "$1: GNU tar archive" != "$dattyp" ] ; then
   echo $dattyp
  else
   tar tvf $1
fi
exit 0
```

Sehen wir vom folgenden Abschnitt mal ab, war das alles, was Sie in diesem Kapitel über die `if`-Abfrage lernen werden. Also auf zum nächsten Abschnitt.

`test` offeriert Ihnen gleich zwei böse Fallen, die Sie nun wirklich nicht mitnehmen sollten. Zuerst stellen Sie sicher, dass zwischen `[ ]` und den Teilausdrücken mindestens ein Leerzeichen ist, ansonsten wird der `test`-Befehl nicht erkannt, und es erscheint eine Fehlermeldung:

```
Skript.sh: [-d: command not found
```

Wenn Sie mit Zeichenketten arbeiten und den Inhalt von Variablen auf bestimmte Inhalte prüfen wollen, so setzen Sie die Variablen unbedingt in Anführungszeichen `""`. Ein kurzer Skriptauschnitt, der dieses Problem klar macht:

```
# Skript 17: Probleme mit test
#
txt=""
if [ $txt = "Test" ] ; then
   echo "OK"
fi
exit 0
```

Wenn Sie diese Abfrage ausführen, kommt folgender Fehler zurück:

```
./skript17.sh: [: =: unary operator expected
```

Der Fehler ist klar: Die Variable `$txt` ist leer, und somit fällt der erste Parameter der Bedingung weg. Der Rest `[ = "Text" ]` ist kein gültiger Ausdruck, und die Shell beschwert sich. Wenn Sie jedoch die Bedingung wie unten formulieren, erkennt die Bash, dass es sich beim ersten Parameter um einen leeren String handelt. Die Shell erkennt somit auch den ersten Parameter und hat keinen Grund zur Beschwerde.


```
# Skript 17: Probleme mit test
#   Wie Fehler durch leere Zeichenketten
#   vermieden werden können
txt=""
if [ "$txt" = "Test" ] ; then
   echo "OK"
fi
exit 0
```

Es ist für ein Skript sehr wichtig, auf Fehler zu reagieren. Dabei kommt eine Abfrage auf genaue Fehlermeldungen nicht in Frage, weil hier die gleichen Probleme auftreten, die wir weiter oben im Zusammenhang mit `file` besprochen hatten. Deshalb bleiben nur die Rückgabewerte der Befehle übrig.

Angenommen, Sie wollen ein Skript schreiben, das ein Verzeichnis als Parameter entgegennimmt und versucht, den Inhalt des Verzeichnisses anzuzeigen. Nach den bisherigen Beispielen würden Sie die Überprüfung sicherlich so kodieren:

```
# Skript cddir.sh:
#  Listet das übergebene Verzeichnis oder gibt Fehler aus
#  Version 1
#
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 verz" >&2
   echo "   verz  --> Verzeichnis" >&2
   exit 1
fi
# Fehlerausgabe unterdrücken, das machen wir, damit die Fehlermeldung
# auf unserem Mist gewachsen ist, nicht auf dem von cd
cd $1 2>/dev/null
if [ $? -ne 0 ] ; then
   echo "$1 ist kein Verzeichnis!" >&2
   exit 1
fi
# Wir stehen im Verzeichnis, nun ausgeben
ls -l
exit 0
```

Sicherlich nicht falsch, aber weiter oben hatte ich erwähnt, dass `if` jeden beliebigen Befehl ausführt und dessen Exitstatus prüft. Was hindert uns also daran, Folgendes zu schreiben:

```
# Skript cddir.sh:
#  Listet das übergebene Verzeichnis oder gibt Fehler aus
#  Version 2
#
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 verz" >&2
   echo "   verz  --> Verzeichnis" >&2
   exit 1
fi
# Fehlerausgabe unterdrücken, falls nötig :)
if cd $1 2>/dev/null ; then
   # Wir stehen im Verzeichnis, nun ausgeben
   ls -l
  else
   echo "$1 ist kein Verzeichnis!" >&2
   exit 1
fi
exit 0
```

Es ist auch möglich, ein »!« vor den Befehl zu setzen, der in der `if`-Anweisung ausgeführt wird. Dann wird der Rückgabewert negiert (aus `0` wird `1` und umgekehrt). So könnte die Abfrage auch so aussehen:


```
# Skript cddir.sh:
#  Listet das übergebene Verzeichnis oder gibt Fehler aus
#  Version 3: cd im if aufrufen
#
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 verz" >&2
   echo "   verz  --> Verzeichnis" >&2
   exit 1
fi
# Fehlerausgabe unterdrücken, falls nötig
if ! cd $1 2>/dev/null ; then
   echo "$1 ist kein Verzeichnis!" >&2
   exit 1
fi
# Wir stehen im Verzeichnis, nun ausgeben
ls -l
exit 0
```

Falls Sie ein Konstrukt benötigen, das für `if` (aber auch für die noch zu behandelnden `while`- und `until`-Schleifen) immer den Status `0` zurückgibt, so nutzen Sie bitte `true`. Der Name ist Programm: `true` gibt immer `0` zurück. Das Gegenteil dazu ist `false`, welches immer `1` zurückgibt.

Wie man sieht, gibt es in Skripten schon für das kleinste Problem mehr als nur eine Lösung. Lassen Sie sich daher nicht entmutigen, wenn Ihre Lösungen anders aussehen als das, was wir Autoren vorschlagen (wenn Sie wüssten, was unser Fachlektor mit unseren Originalskripten gemacht hat …).

3.3 Die case-Anweisung


```
case <ausdruck1> in
   <le1>) <befehl1> ;;
   <le2>) <befehl2> ;;
   ...
 esac
```

Eine `if`-Abfrage ist ja recht schön, aber wenn eine Variable auf eine Reihe von verschiedenen Inhalten zu prüfen ist, haben Sie eine lange `if-else`- oder `if-elif`-Schlange, was ein wenig unübersichtlich werden kann. Viel schlimmer wiegt aber für einen wahren Unixguru die Tatsache, dass sie (er) viel zu viel

tippen muss. Aber glücklicherweise gibt es da noch die case-Abfrage, unter diesen Umständen ein echtes Schnäppchen.

case verlangt einen Ausdruck <ausdruck1>, der ausgewertet und dann mit einer Liste möglicher Ergebnisse verglichen wird. Hinter jedem Listeneintrag (<le1>, <le2> usw.) stehen ein oder mehrere Befehle (<befehl1>, <befehl2> usw.), die abgearbeitet werden, wenn Ausdruck <ausdruck1> und Listeneintrag übereinstimmen. Die Shell arbeitet die Befehle so lange ab, bis sie auf ein »;;« trifft und führt dann den ersten Befehl nach dem esac aus. Falls die Shell keinen passenden Listeneintrag findet, führt die Shell ebenfalls den ersten Befehl nach esac aus.

Schauen wir uns das einmal in der Praxis an. Dazu wollen wir Skript 16 noch ein wenig verbessern. Neben den Tar-Archiven soll unser Skript jetzt auch normalen Text und gzip-Dateien erkennen und ausgeben können. Die Originaldateien dürfen natürlich nicht durch unser Skript geändert werden. Normale Texte werden mit cat ausgegeben, während durch gzip komprimierte Dateien mit gunzip dekomprimiert werden können. Damit der Benutzer erkennen kann, dass die Datei komprimiert wurde, hängt gzip ein .gz an den Dateinamen an. gunzip entfernt das .gz und stellt die Datei unkomprimiert zur Verfügung.

```
buch@koala:/home/buch > tar cvf backup.tar *.txt
anhanga.txt
einleitung.txt
kapitel1.txt
kapitel2.txt
kapitel3.txt
toc.txt
buch@koala:/home/buch > gzip backup.tar
buch@koala:/home/buch > ls back*
backup.tar.gz
buch@koala:/home/buch > file backup.tar.gz
backup.tar.gz: gzip compressed data, deflated, original filename, last
modified: Sat Feb 13 14:35:03 1999, os: Unix
buch@koala:/home/buch >
```

Versuchen wir also, mit diesem Wissen eine erste Version zu erstellen.


```
# Skript 18: case-Befehl
# Version 1
#
# Dieses Skript versucht, den Dateityp zu ermitteln
# und abhängig davon Aktionen durchzuführen
#
# Wir brauchen genau einen Parameter
```

```
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 <datei>" 1>&2
   exit 1
fi
# Handelt es sich um eine normale, lesbare Datei?
if [ ! -r "$1" ] ; then
   echo "Datei nicht gefunden" >&2
   exit 1
fi
# Bestimmen des Dateityps
typ=`file $1`
case "$typ" in
   "$1: ASCII text")       echo "Normale Textdatei"
                           cat $1 ;;
   "$1: GNU tar archive")  echo "Tar Archiv"
                           tar tvf $1 ;;
   "$1: gzip compressed data,")
          echo "Komprimierte Datei"
          gunzip <"$1" | file -
      ;;
   *)     echo "Unbekannter Typ"
          ls -l $1
          ;;
esac
exit 0
```

Der Vorteil von `case` im Vergleich zu `if` wird hier recht deutlich. Sie können sämtliche Ersatzmuster aus dem letzten Kapitel auch in der `case`-Abfrage zum Einsatz bringen, solange diese nicht in Anführungszeichen stehen. Ersatzmuster in Anführungszeichen werden von der Shell innerhalb des `case`-Befehls nicht beachtet und wie normale Zeichen behandelt. Außerdem beziehen sich Ersatzmuster innerhalb von `case`-Listeneinträgen nicht auf Dateinamen, sondern auf die Werte des abgefragten Ausdrucks (hier also die Variable `$typ`).

Mit diesem Skript handeln wir uns leider ein Problem ein:

- Erkennt `file` eine komprimierte Datei, so gibt es eine Menge Informationen (Datum, Betriebssystem, siehe oben) aus, die wir selbst mit Variablenersetzung nur mit erheblichen Schwierigkeiten lösen können.

Das Problem ist einfach zu lösen. Wir nutzen einfach die Ersatzmuster aus. Hier die angepasste Abfrage:

```
"$1: gzip compressed data,"*
```

Kommen wir noch auf das Konstrukt `gunzip <"$1" | file -` zurück. Wenn `gunzip` die zu dekomprimierenden Daten von der Standardeingabe bekommt, so gibt es das Ergebnis an die Standardausgabe weiter. Wenn Sie die Ausgabe auf die Standardausgabe erzwingen wollen, obwohl die Daten in einer Datei vorliegen, so nutzen Sie bitte die Option `-c`.

Wird `file` mit der Option - aufgerufen, untersucht `file` keine Datei, sondern die Daten, die es über seine Standardeingabe empfängt. Bringen wir nun alle neuen Informationen in der verbesserten Version unter, so könnte das Ergebnis so aussehen:


```
# Skript 18: case-Befehl
# Version 2
# Diesmal erkennen wir den Dateityp und
# nutzen die Ersatzzeichen, um variable
# Ausgaben abzudecken
#
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 <datei>"
   exit 1
fi
if [ ! -r "$1" ] ; then
   echo "Datei nicht gefunden oder nicht lesbar" >&2
   exit 1
fi
# Bestimmen des Dateityps
typ=`file $1`
case "$typ" in
   "$1: ASCII text")       echo "Normale Textdatei"
                           cat $1 ;;
   "$1: GNU tar archive")  echo "Tar Archiv"
                           tar tvf $1 ;;
   "$1: gzip compressed data,"*)
                           echo "Komprimierte Datei"
                           gunzip -c $1 | file -
                           ;;
   *)                  echo "Unbekannter Typ"
                       ls -l $1
                       ;;
esac
exit 0
```

Schon besser, aber leider immer noch nicht fehlerlos. Wenn Sie ein `gzip`-Archiv erwischt haben, haben Sie eventuell folgenden Fehler erhalten:

```
buch@koala:/home/buch > ./skript18.sh backup.tar.gz
Komprimierte Datei
standard input:              GNU tar archive
./skript18.sh: line 27:  2731 Broken pipe          gunzip -c $1
      2732 Done                    | file -
buch@koala:/home/buch >
```

Dieser Fehler ist ärgerlich, aber abhängig von der Implementation von `file` nicht zu verhindern. Ignorieren Sie ihn und leiten Sie die Fehlerausgabe für

`file` und `gunzip` nach `/dev/null` um, oder leiten Sie die Ausgabe `gunzip -c` in eine Datei um, ermitteln davon das Datenformat und löschen die Datei danach.

```
...
rm /tmp/datei
gunzip -c $1 >/tmp/datei
file /tmp/datei
rm /tmp/datei
...
```

Im Kapitel 2 hatten wir die Pipes ja besprochen. In diesem Falle startet `gunzip` und fängt an, die Datei zu dekomprimieren und gleichzeitig auf die Standardausgabe auszugeben. Diese wird von der Pipe aufgenommen und an `file` weitergeleitet, welches nach `gunzip` gestartet wurde. Hat `file` genügend Daten von `gunzip` erhalten, um das Datenformat erkennen zu können, gibt `file` sein Ergebnis aus und beendet sich. Damit wird die Pipe ebenfalls geschlossen. Sollte `gunzip` zu diesem Zeitpunkt noch nicht mit der Dekomprimierung fertig sein, versucht es, weiter Daten in die Pipe zu schicken. Da diese aber geschlossen ist, tritt der obige Fehler auf. Die Nummern, die das Skript ausgibt, sind die Prozessnummern der einzelnen Befehle. Abbildung 3.1 illustriert das Problem.

gunzip	**»Inhalt« der Pipe**	**file**
Startet		
		Startet
Ausgabe Zeile 1		
	Zeile 1	
		Zeile 1
Ausgabe Zeile 2		
	Zeile 2	
		Zeile 2: Dateiformat erkannt, beendet sich
	wird geschlossen	
Ausgabe Zeile 3		
	Pipe zu -> Fehler!	

Abb. 3.1: Das Problem mit gunzip -c $1 | file -

Bevor wir das Thema case zu den Akten legen können, möchte ich Sie noch auf ein Problem hinweisen. Geben Sie einmal einen englischen Testtext ein, speichern Sie ihn als txt.txt, und ermitteln Sie das Dateiformat:

```
buch@koala:/home/buch > cat txt.txt
Hello World,
this is an english test text. Let us see, what the result will
be if we send it through file
Thanks
buch@koala:/home/buch > file txt.txt
txt.txt: English text
buch@koala:/home/buch > ./skript18.sh txt.txt
Unbekannter Typ
-rw-r--r--   1 buch     users         115 Feb 13 15:24 txt.txt
buch@koala:/home/buch >
```

Tolle Analyse von file, aber unser Skript ist an diesem Problem mit wehenden Fahnen eingegangen, sollte es doch Normale Textdatei und den Dateiinhalt ausgeben. Kein Problem, werden Sie sagen, fügen wir einfach noch folgende Bedingung ins case ein:

```
...
"$1: English text")    echo "Normale Textdatei"
                       cat $1 ;;
...
```

Zugegeben, das wird funktionieren, aber die Qualifikation für den Titel *»Unixguru von Kapitel 3«* haben Sie deutlich verpasst, denn ein wahrer Unixguru hätte sich die doppelte Angabe von Befehlen gespart, unterscheiden sich doch beide Ausgaben von Normale Textdatei nur durch die Konstante für die case-Bedingung.

Mit etwas mehr Ausdauer beim Lesen hätte ich Ihnen die Möglichkeit erklärt, wie Sie mehrere Bedingungen in case verknüpfen können. Dazu nutzen Sie das »|«-Symbol, das in diesem Fall keine Pipe aufmacht. Indem Sie zwischen den einzelnen Bedingungen ein »|« setzen, können Sie innerhalb *einer* Zeile beliebig viele Bedingungen durch ein logisches ODER verknüpfen.

```
"$1: ASCII text" | "$1: English text")
                       echo "Normale Textdatei"
                       cat $1 ;;
```

Neben dem »*« können Sie auch alle sonstigen Ersatzzeichen verwenden, allerdings gibt es hier einen kleinen, aber feinen Unterschied:

- Erstens beziehen sich die Ersatzmuster nur auf die nach case angegebene Zeichenkette und nicht auf Dateinamen.
- Zweitens beachtet »*« per Default keine Dateinamen, die mit einem ».« beginnen. Bei den Zeichenketten im case gilt diese Einschränkung allerdings nicht.

Noch ein kleines Beispiel:


```
# Skript case.sh
#   Demonstriert die Ersatzmuster im "case"
#
echo -n "Eingabe: "
read gvEingabe
case "$gvEingabe" in
  "a" | "b" ) echo "Kleines A oder B"
              ;;
  a?        ) echo "a plus <X>"
              ;;
  [d-h]     ) echo "Von d bis h"
              ;;
  ?         ) echo  "Einzelnes Zeichen"
              ;;
  *         ) echo "und der ganze Rest"
              ;;
esac
exit 0
```

Mit dem Befehl `read` bewegen wir das Skript dazu, eine Benutzereingabe entgegenzunehmen und in der Variablen `gvEingabe` abzulegen. Und so sieht das Ergebnis aus:

```
buch@koala:/home/buch > ./case.sh
Eingabe: f
Von d bis h
buch@koala:/home/buch > ./case.sh
Eingabe: .dd
und der ganze Rest
buch@koala:/home/buch > ./case.sh
Eingabe: ax
a plus <X>
buch@koala:/home/buch > ./case.sh
Eingabe: a
Kleines A oder B
buch@koala:/home/buch >
```

Ein letzter Hinweis zum Exitstatus vom `case`-Befehl. Dieser ist `0`, wenn der Ausdruck mit keinem Listeneintrag übereinstimmt, ansonsten ist er identisch mit dem Exitstatus des letzten ausgeführten Befehls.

3.4 Die while-Schleife


```
while <befehl1> ; do
   <befehl2>
   <befehl3>
   ...
done
```

Während Abfragen nötig sind, um innerhalb eines Skripts auf bestimmte Bedingungen reagieren zu können, brauchen Sie auch Konstrukte, die Befehle so lange ausführen, wie eine bestimmte Bedingung erfüllt ist. Dazu setzen fast alle Programmiersprachen `while`-Schleifen ein, und auch die Shell bildet hier keine Ausnahme.

Die `while`-Schleife führt `<befehl1>` aus, und wenn der Exitstatus dieses Befehls gleich `0` ist, werden die Befehle bis zum abschließenden `done` ausgeführt und danach wieder mit `<befehl1>` begonnen. Der Exitstatus der Schleife selbst ist `0`, wenn die Schleife nicht durchlaufen wurde, oder gleich dem Exitstatus des letzten Befehls, der innerhalb der `while`-Schleife ausgeführt wurde. Dabei gilt wie auch beim `if: <befehl1>` ist häufig `test`, kann aber auch eine Pipe, eine Befehlsliste oder eine Befehlsgruppe sein.

Das war schon die ganze Theorie. Also schauen wir uns die Schleife mal in der Praxis an. Stellen wir uns folgende Aufgabe: Es soll ein Skript geschrieben werden, das alle Vorkommen eines Dateinamens innerhalb eines Verzeichnisses (und eventuelle Unterverzeichnisse) ausgeben kann. Als kleinen Bonus wollen wir die Summe der belegten Bytes aller Dateien ermitteln und ausgeben. Der erste Parameter ist dabei das Verzeichnis, in dem gesucht wird, und Parameter zwei der Dateiname, der auch Jokerzeichen enthalten darf.

Bei genauer Betrachtung der Aufgabe stellen wir fest, dass viele der Teilprobleme schon in den bisherigen Skripten aufgetaucht sind:

- Die betroffenen Dateinamen lassen sich mit `find` finden. `find` ist ein Unixbefehl, der bestimmte Datei- oder Verzeichnisnamen auf Grund angegebener Bedingung findet. So gibt

  ```
  find /home/buch -name '*.txt' -print
  ```

  alle Dateien (`-type f`) im Verzeichnis `/home/buch` aus, die auf `.txt` enden (`-name '*.txt'`). `find` bietet noch viele andere Möglichkeiten, die wir im Verlauf des Buches noch genauer unter die Lupe nehmen werden.
- Die Größe der Datei ist identisch mit der Anzahl an Zeichen, die in ihr gespeichert sind. Dazu lässt sich `wc` nutzen.
- Die nötigen Berechnungen lassen sich mit `expr` ausführen.

- Die Ausgabe von `find` wird zweimal benötigt. Einmal, um die Anzahl an Dateien zu ermitteln, und einmal, um die Dateinamen zu ermitteln. Aus diesem Grunde empfiehlt sich der Einsatz von `tee`.

`wc` gibt nicht immer die genaue Dateigröße aus. Unter Unix dürfen Dateien Löcher enthalten. Kurz und knapp sind Löcher Dateibereiche, in die keine Daten geschrieben wurden. Genau wie auf einer Straße nicht alle Hausnummern vergeben sein müssen, müssen Programme auch nicht jedes Byte in der Datei mit Daten füllen.

Mit `dd` (*Diskdump*) kann man so eine Datei namens `Loch` wie folgt anlegen:

```
buch@koala:/home/buch > dd if=/dev/zero bs=1024 count=1 seek=1023 of=Loch
1+0 records in
1+0 records out
buch@koala:/home/buch >
```

Dabei werden Daten aus dem Gerät `/dev/zero` (das so heißt, weil es immer Bytes mit dem Wert `0` zurückgibt) ausgelesen. Die Größe eines Blocks ist 1024, und der Block besteht aus Bytes (count=1). Die Daten werden an den 1023. Block geschrieben (seek=1023), also ans Ende der Datei, weil `seek` von `0` an zählt. Die enthält nur 1024 Byte (= 1 Kilobyte), aber `wc` und `ls -l` geben andere Werte aus:

```
buch@koala:/home/buch > ls -l loch
-rw-r--r--   1 buch  users     1048576 Apr 17 0:21 loch
buch@koala:/home/buch > wc loch
      0       1 1048576 loch
buch@koala:/home/buch > du -k loch
4     loch
buch@koala:/home/buch >
```

`du -k` gibt die tatsächliche Größe der Datei in Kilobyte aus, wie der englische Name schon sagt, den *Disk Usage*. Obwohl `wc` eine Größe von 1 Mbyte angibt, ist die Datei nur 4 Kbyte groß!

Auch wenn Sie die Datei mit `cp` kopieren, ändert sich die Größe nicht. Erst ein `cat` macht aus den Löchern Bytes mit dem Wert `0`:

```
buch@koala:/home/buch > cat loch > keinloch
buch@koala:/home/buch > wc keinloch
      0       1 1048576 keinloch
buch@koala:/home/buch > du -k keinloch
1029  keinloch
buch@koala:/home/buch >
```

Solche Dateien werden wir in diesem Buch aber nicht anlegen, aber früher oder später werden Sie auf ein Problem stoßen, das hieraus resultiert.

Bleibt eigentlich nur die Frage, wie wir die Dateinamen aus der Ergebnisdatei, die `find` angelegt hat, ermitteln können. Auch dieses Problem haben wir schon ansatzweise gelöst, und zwar in den Skripten 14 und 15.

Damit sind fast alle Probleme schon gelöst, ein Befehl fehlt aber noch:

```
cut -c<bereich>
```

schneidet aus jeder Zeile, die dem Befehl übergeben wird, die Zeichen heraus, die durch `<bereich>` definiert werden, und gibt diese Zeichen auf der Standardausgabe aus.

Tabelle 3.2: Beispiele für die Benutzung von cut

`cut -c1`	schneidet das erste Zeichen jeder übergebenen Zeile aus und gibt es aus.
`cut -c-7`	schneidet alle Zeichen bis zum siebten jeder übergebenen Zeile aus und gibt diese Zeichen aus.
`cut -c1-7`	entspricht `-c-7`
`cut -c2-`	schneidet alle Zeichen ab dem zweiten aus und gibt sie aus. Auch das gilt für jede übergebene Zeile.

So gibt

```
echo "Hallo Welt" | cut -c2-
```

Folgendes aus:

```
Hallo Welt
```


```
# Skript 19: Unsere erste while-Schleife
# Sucht alle Dateien mit dem Namen/Muster $2
# im Verzeichnis $1. Keine Prüfungen
TMPFILE=/tmp/count
#
#
anz=`find $1 -name "$2" -type f -print | tee $TMPFILE | wc -l`
nr=0
summe=0
while [ $nr -lt $anz ] ; do
  nr=`expr $nr + 1`
  datei=`head -$nr $TMPFILE | tail -1`
  echo $datei
  #
  # Dieses cut muss angegeben werden
  #
  erg=`wc -c $datei | cut -c-7`
  summe=`expr $summe + $erg`
done
```

```
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $anz Dateien belegen $summe Bytes"
fi
rm $TMPFILE
exit 0
```

Klappt wunderbar, bleiben allerdings zwei Anmerkungen: Einige Versionen von `wc` sind fest darauf fixiert, einen Dateinamen hinter der Anzahl der Zeichen auszugeben. Unter diesen Umständen ist eine Addition der Größen unmöglich. Da die Zahlen auf allen Systemen, auf denen ich gearbeitet habe, nie länger als sieben Stellen waren, schneiden wir diese sieben Zeichen einfach aus. Leerzeichen werden von der Shell ja ignoriert, sodass die führenden Leerzeichen vor der ausgeschnittenen Zahl ignoriert werden können.

Das ist natürlich alles andere als portabel. Was, wenn `wc` auf Ihrem System auf einmal zehn Stellen ausgibt, und wir nur die ersten sieben Stellen berücksichtigen? Kleinere Rundungsfehler wären hier die Folge `:o)`, und das Ergebnis wäre leicht verfälscht. Momentan bleibt Ihnen leider nichts anderes übrig, als den Zeichenbereich für `cut` Ihrem System anzupassen.

Eine Lösung wäre die Umleitung der Eingabe `wc -c <$datei`, wie bereits in Kapitel 2 angesprochen. Weitere Lösungen werden im Abschnitt über die `for`-Schleife, in Kapitel 5 »Parameter zum Zweiten« und Kapitel 10 »sed« aufgezeigt, also haben Sie noch etwas Geduld.

3.5 Die until-Schleife


```
until <befehl1>; do
  <befehl2>;
  <befehl3>;
  ...
done
```

Der folgende Abschnitt könnte eine Wiederholung mit nur minimalen Abweichungen von Abschnitt 3.4 sein. Die wollen wir Ihnen aber ersparen und weisen daher nur auf die Unterschiede zwischen `while` und `until` hin.

`until` ist identisch zur `while`-Schleife mit einem wichtigen Unterschied: Während die `while`-Schleife so lange läuft, wie die Bedingung wahr ist (oder allgemeiner formuliert: solange `<befehl1>` einen Exitstatus von `0` zurückgibt), läuft die `until`-Schleife so lange, *bis* die Bedingung wahr ist (oder genauer: bis

befehl1 einen Exitstatus von 0 zurückgibt). Der Exitstatus von until ist 0, wenn kein Schleifendurchlauf stattfand, ansonsten ist er gleich dem Exitstatus des letzten Befehls, der innerhalb der Schleife ausgeführt wurde.

Nutzen wir diesen Abschnitt einmal dazu, zu demonstrieren, dass befehl1 zwar im Großteil aller Fälle ein test ist, aber dass dies nicht notwendigerweise so sein muss.


```
# Skript 20:
#  Ein simples Skript mit until-Schleife
#
# Läuft, bis Benutzer CTRL-C drückt (Endlosschleife)
#
echo "Abbruch mit CTRL-C"
until false ; do
   date
   sleep 10
done
exit 0
```

Zu diesem Skript ist noch Folgendes zu sagen: date gibt das aktuelle Systemdatum und die aktuelle Systemzeit aus. sleep versetzt das Skript für x Sekunden (Parameter 1) in einen Wartezustand. Dieser läuft nach der angegebenen Anzahl von Sekunden ab oder wenn das Skript abgebrochen wird. false ist ein Kommando, das immer einen Exitstatus von 1 zurückgibt.

Wie Sie sehen: So groß sind die Unterschiede nicht, aber fein.

3.6 Die for-Schleife


```
for <var1> in <wort1> ; do
   <befehl1> ;
   <befehl2> ;
   ...
done
```

<wort1> wird nach den Regeln der Ersatzmuster in eine Wortliste umgewandelt. Danach läuft die Schleife für jedes ermittelte Wort durch, das jeweils in var1 abgespeichert wird. Verwirrt? Dann hilft nur ein simples Skript, um diesen drögen Satz mit Leben zu erfüllen.


```
# Skript 21: Demo für for und Brace-Expansion
#
for land in Austr{ia,alia,alien} ; do
  echo $land
done
exit 0
```

Wenn wir das nun aufrufen, erhalten wir folgende Ausgabe:

```
buch@koala:/home/buch > ./skript21.sh
Austria
Australia
Australien
buch@koala:/home/buch >
```

Sehen wir uns noch einmal kurz die Zeile mit der `for`-Anweisung an. Bevor die Schleife ausgeführt wird, wird `Austr{ia,alia,alien}` in eine Wortliste umgewandelt. Ausgeführt wird also folgende Anweisung:

```
for land in Austria Australia Australien ; do
```

Bestimmt haben Sie jetzt einige wilde Ideen, welche Verbrechen Sie mit dieser Schleife begehen können. Aber seien Sie sich sicher, die haben wir auch. Bevor wir unsere Skripten auf die nichtsahnende Unixgemeinde loslassen, schauen wir noch einmal auf die zweite Version der `for`-Schleife:


```
for <var1> do
   <befehl1> ;
   <befehl2> ;
   ...
done
```

Der einzige Unterschied zur Version 1 liegt darin, dass keine Wortliste angegeben wird. Fehlt diese, so nimmt die Schleife alle angegebenen Parameter des Skripts und weist diese mit jedem Schleifendurchlauf sukzessiv `var1` zu. Wird `for` in einer Funktion benutzt, dann werden die Funktionsparameter und nicht die Shellparameter genommen. Zu den Funktionen kommen wir ausführlich in Kapitel 7.

Auch hier ein simples Beispiel:


```
# Skript 22: for auf Parameter angewendet
#
for param ; do
  echo "Parameter $param"
done
exit 0
```

Rufen Sie dieses Skript mal mit verschiedensten Parametern auf:

```
buch@koala:/home/buch > ./skript22.sh 1 2 3 4 5
Parameter 1
Parameter 2
Parameter 3
Parameter 4
Parameter 5
buch@koala:/home/buch > ./skript22.sh Sydney Perth Canberra
Parameter Sydney
Parameter Perth
Parameter Canberra
buch@koala:/home/buch >
```

Zwar ist diese Methode nicht unpraktisch, aber es stellt sich die Frage, wo der Unterschied zu `for param in $1 $2 $3 $4 $5` liegt. Wie bereits erwähnt, kommen Sie so nur an die ersten neun Parameter (`$1` bis `$9`) heran, die `for`-Schleife jedoch durchläuft *alle* Parameter, also auch die ab Position 10, die bis jetzt unerreichbar waren.

Diese Methode, an Parameter 10 und aufwärts zu gelangen, ist allerdings immer noch ein wenig umständlich, und wir werden uns im Kapitel 5 »Parameter zum Zweiten« dieser Problematik nochmals annehmen.

Nun wird es Zeit für ein wenig Praxis, denn die letzten beiden Skripten können wir nun wirklich nicht mehr als praktische Übung werten. Kommen wir deshalb noch einmal auf Skript 19 zurück, welches ein Portabilitätsproblem durch den Einsatz von `cut` hatte. Dieses Problem lässt sich mit wenig Aufwand durch `for` aus der Welt schaffen.


```
# Skript 19: Verbesserte Version ohne cut
#
# Ermittelt die Anzahl an Dateien mit dem Muster $2 im
# Verzeichnis $1
#
# Hier nutzen wir "for", um die Dateigröße zu ermitteln
#
TMPFILE=/tmp/count
line=`find $1 -name "$2" -type f -print | tee $TMPFILE |wc -l`
#
# Falls Ihr "wc" in der oben angegebenen Anweisung mehr als nur eine
# Zahl zurückgibt (z.B. "   1224 standard input"), passen Sie die
# Abfrage an, wie unten in der while-Schleife aufgezeigt
#
anz=$line
nr=0
summe=0
while [ $nr -lt $anz ] ; do
  nr=`expr $nr + 1`
  datei=`head -$nr $TMPFILE | tail -1`
  echo $datei
```

```
  line=`wc -c $datei`
  #
  # Hier steht in line z.B. ein Wert von
  # "   1234 /tmp/count"
  #
  add="0"
  for wort in $line ; do
     if [ "$add" = "0" ] ; then
        add="1"
        erg=$wort
     fi
  done
  summe=`expr $summe + $erg`
done
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $anz Dateien belegen $summe Bytes"
fi
rm $TMPFILE
exit 0
```

Schon besser, und um die Leerzeichen müssen wir uns auch nicht mehr kümmern, da diese von `for` als Worttrenner interpretiert werden. Diese Version sollte also unabhängig davon funktionieren, wie viele Stellen `wc` für das Ergebnis ausgibt.

Unser Lektor meinte zu dieser Lösung »Man kann es sich wirklich unnötig schwer machen«. Recht hat er, auch dies ist noch nicht der Weisheit letzter Schluß. Schreiben wir deshalb noch ein kleines Skript, das einen Parameter akzeptiert und alle Dateien, die diesem Muster entsprechen, und deren erste Zeile ausgibt.


```
# Skript show.sh
# Demonstriert for
# Gibt für alle durch $1 gefundenen
# Dateien die erste Zeile aus.
# Keine Überprüfung auf Verzeichnisse o.ä.
#
if [ $# -ne 1 ] ; then
   echo "Aufruf: $0 muster" 1>&2
   exit 1
fi
gvEnd=$1
for gvDatei in $gvEnd ; do
  gvZeile=`sed -n -e '1p' <$gvDatei`
  echo "$gvDatei : $gvZeile"
done
exit 0
```

Und weil es bisher so gut lief, haben wir uns schon einen kleinen Vorgriff auf Kapitel 10 erlaubt: `sed` wird erst dort erklärt. Hier soll die Erklärung reichen, dass `sed` die Eingabe filtert und das Ergebnis auf die Standardausgabe druckt. Dieses Beispiel gibt die erste Zeile der Eingabe aus: `-e '1p'`. Dieser Befehl ist wesentlich effektiver als die Kombination von `head` und `tail`.

```
buch@koala:/home/buch > ./show.sh \*.txt
anhanga.txt : - die Pager heißen überall anders:
anhangb.txt : Anhang B: Das letzte Skript
anhange.txt : Anhang D: Ressourcen im Netz
einleitung.txt : Einleitung
kapitel1.txt : Wir dürfen jetzt den Sand nicht in den Kopf stecken - Lothar
Matthäus
kapitel10.txt : Kapitel 10: sed
kapitel11.txt : Kapitel 11: Reste und Sonderangebote
kapitel13.txt : <CHRISTA>
kapitel2.txt : Kapitel 2: Interaktion von Programmen
kapitel3.txt : Kapitel 3: Abfragen und Schleifen
kapitel4.txt : Kapitel 4: Terminal Ein-/Ausgabe
kapitel5.txt : Kapitel 5: Parameter zum Zweiten
kapitel6.txt : Kapitel 6: Variablen und andere Mysterien
kapitel7.txt : Kapitel 7: Funktionen
kapitel8.txt : Kapitel 8: Prozesse und Signale
kapitel9.txt : Kapitel 9: Befehlslisten und sonstiger Kleinkram
toc.txt : #Inhaltsverzeichnis:
tst.txt : Zeile 1
buch@koala:/home/buch >
```

Ksh-93 und Bash ab Version 2.04 bieten auch eine arithmetische for-Schleife an. genauere Informationen finden Sie im Anhang A.

3.7 Die Befehle break und continue

Wie kann eine Schleife vorzeitig beendet werden?

3.7.1 break

Wir wollen jetzt das Skript `show.sh` ein wenig aufbohren. Wir akzeptieren noch einen zweiten Parameter. Findet sich der darin angegebene Text in der ersten Zeile des Textes wieder, so soll die `for`-Schleife abgebrochen werden und sämtliche Informationen über die Datei mittels `ls -l` ausgegeben werden.

Wenn es nur darum gehen würde, die `for`-Schleife abzubrechen, so böte sich `exit` an. An dieser Stelle soll aber die Verarbeitung fortgeführt werden. Es wäre sicherlich kein Problem, den Inhalt zu prüfen und im positiven Falle `ls` aufzurufen und dann das Skript mit `exit` zu beenden.

```
if [ "$zeile" = "$vergl" ] ; then
   ls -l $gvDatei
   exit 0
fi
```

Für einige wenige Befehle, die im entsprechenden `if` aufgerufen werden, ist das sicherlich kein Problem. Wenn sich aber die `if`-Abfrage durch diese Art von Programmierung auf mehr als eine Bildschirmseite aufbläht, ist bei mir die Schmerzgrenze erreicht.

Es muss also noch eine andere Möglichkeit geben, wie eine Schleife abgebrochen werden kann. Ein Flag zu setzen und dieses dann abzufragen, ist auch keine echte Lösung, denn jeder überflüssige Schleifendurchlauf kostet Zeit, die besser genutzt werden könnte. Den gesuchten Befehl gibt es tatsächlich, er lautet `break`.

Trifft die Bash auf ein `break`, so wird die aktuelle `for`/`while`/`until`-Schleife beendet und der nächste Befehl nach dem passenden `done` ausgeführt. Unter diesen Umständen könnte eine neue Version von `show.sh` so aussehen:


```
# Skript show.sh
# Demonstriert for und break
# Version 2
# Erwartet zwei Parameter: ein Muster für Dateinamen
# und ein Stopwort. Wird dieses in der ersten
# Zeile der untersuchten Datei gefunden, so
# wird die Schleife abgebrochen und Infos
# zur Datei ausgegeben.
#
if [ $# -ne 2 ] ; then
   echo "Aufruf: $0 muster Stop" 1>&2
   exit 1
fi
gvEnd=$1
gvStop=$2
for gvDatei in $gvEnd ; do
  zeile=`cat $gvDatei | sed -n -e "1p"`
  if [ "$gvStop" = "$zeile" ] ; then
     break
  fi
  echo "$gvDatei : $zeile"
done
echo
echo "-------- Info zu $gvDatei -------"
echo
ls -l $gvDatei
echo
head -10 $gvDatei
exit 0
```

Ein Aufruf mit den Dateien in unserem Arbeitsverzeichnis gibt momentan Folgendes aus:

```
buch@koala:/home/buch > ./show.sh \*.txt "<CHRISTA>"
anhanga.txt : - die Pager heißen überall anders:
anhangb.txt : Anhang B: Das letzte Skript
anhange.txt : Anhang D: Ressourcen im Netz
einleitung.txt : Einleitung
kapitel1.txt : Wir dürfen jetzt den Sand nicht in den Kopf stecken - Lothar
Matthäus
kapitel10.txt : Kapitel 10: sed
kapitel11.txt : Kapitel 11: Reste und Sonderangebote
-------- Info zu kapitel13.txt -------
-rw-r--r--   1 buch     users       34227 Apr 16 23:07 kapitel13.txt

<CHRISTA>
Bitte beachte, dass die Skriptnummern nicht mehr korrekt sind. Da dieses
Kapitel direkt nach Kapitel 8 geschrieben wurde, kann ich noch nicht
sagen, wieviel Skripten in den restlichen Kapiteln auftauchen. Sorry.
</CHRISTA>
Kapitel 13: Debugging / Fehlersuche
                              "Alles was schiefgehen kann, geht schief"
                                                  Murphys Gesetz
buch@koala:/home/buch >
```

Wenn Sie mehrere Schleifen geschachtelt haben und möchten mehr als nur eine Schleife verlassen, können Sie einen numerischen Parameter an `break` übergeben. Dieser bezeichnet die Anzahl an geschachtelten Schleifen, die Sie verlassen möchten. Der Parameter muss größer oder gleich 1 sein. `break` und `break 1` sind somit funktionell identisch.

Schauen wir uns ein Skriptstück an, welches dies verdeutlicht. Es sucht alle Unterverzeichnisse mit `find` heraus und gibt mittels `ls -ld` Informationen über die ersten vier darin gefundenen Dateien und Verzeichnisse aus.


```
# Skript 23: break
# Demonstriert den Nutzen von "break"
# in geschachtelten "for"-Schleifen.
# Sucht alle Unterverzeichnisse im Verzeichnis $1
# und zählt die Anzahl an Dateien in den Verzeichnissen
#
gvPWD=`pwd`
for gvVerz in `find $1 -type d  -print` ; do
    # Ebene 1
    gvSumme=0
    for gvDatei in $gvVerz/* ; do
        # Ebene 2
        ls -ld $gvDatei 2>/dev/null
        gvSumme=`expr $gvSumme + 1`
```

```
        if [ $gvSumme -gt 4 ] ; then
           echo "$gvVerz enthält mehr als 4 Dateien / Verzeichnisse"
           break    # Beende Durchlauf von "for gvDatei ..."
                       # Zurück zu Ebene 1
        fi
    done
    # Nach break geht es hier weiter
done
exit 0
```

Möchten Sie das Programm komplett abbrechen, wenn das erste Verzeichnis mehr als vier Dateien und Verzeichnisse ausgibt, so tauschen Sie einfach `break` gegen `break 2` aus.

Ist die Anzahl der aktiven Schleifen kleiner als der Parameter für `break`, werden alle aktiven Schleifen verlassen, und das Skript läuft ohne Fehler weiter. In Skript 23 führt deshalb ein `break 2` zu dem gleichen Ergebnis wie ein `break 333`.

Der Exitstatus von `break` ist immer gleich `0`, außer `break` wurde außerhalb einer Schleife aufgerufen, dann ist er `1`.

3.7.2 continue

`continue` hat eine ähnliche Funktion wie `break`. Während `break` jedoch die aktuelle Schleife abbricht, führt `continue` dazu, dass der nächste Durchlauf der aktuellen Schleife angestoßen wird.

Auch hier ein einfaches Beispiel. Erstellen wir ein Skript, das den Inhalt des aktuellen Verzeichnisses ausgibt. Normale Dateien werden einfach durch ein `ls -l` angezeigt, während bei Unterverzeichnissen eine Sonderbehandlung eintritt:

- Mittels `du -k <verzeichnis>` wird ermittelt, wieviel Kilobyte die Dateien und Unterverzeichnisse im angegebenen Verzeichnis belegen (`du` steht für *disk usage*).
- `ls -l` gibt den Inhalt eines Verzeichnisses aus, aber nicht die Informationen (Besitzer, Rechte usw.) über das Verzeichnis. Mit der Option `-ld` kann genau dies erreicht werden.


```
# Skript 24: continue
# gibt die Größe der Unterzeichnisse
# im aktuellen Verzeichnis aus.
# "." und ".." werden nicht berücksichtigt
#
```

```
for var1 in .* * ; do
  if [ "$var1" = "." -o "$var1" = ".." ] ; then
     continue
  fi
  if [ -d "$var1" ] ; then
     echo "Verzeichnisgröße `du -k $var1`"
     ls -ld $var1
     continue
  fi
  ls -l $var1
done
exit 0
```

Unter Unix werden Dateien, die mit ».« anfangen, nicht angezeigt. Sie sind so lange versteckt, wie eine Bearbeitung bzw. Anzeige nicht explizit angefordert wird. Deshalb ist in der `for`-Schleife sowohl »`.*`« als auch »`*`« angegeben, denn dieses Skript soll den gesamten Verzeichnisinhalt beachten. Das zweite `continue` ist eigentlich überflüssig, da ein `else` genau den gleichen Effekt haben würde, aber wir wollten die Funktion von `continue` demonstrieren.

Auch `continue` erlaubt die Angabe eines numerischen Parameters `x`. Haben Sie mehrere Schleifen ineinander geschachtelt und ist `x` gleich `1`, so bewirkt `continue`, dass der nächste Durchlauf durch die aktuelle Schleife ausgeführt wird. Ist `x` gleich `2`, so wird die aktuelle Schleife abgebrochen, und der nächste Durchlauf der vorletzten Schleife wird angestoßen. Dieses Schema wird für alle Schachtelungsebenen durchgehalten. Ist `x` größer als die Anzahl der Schachtelungen, so wird der nächste Durchlauf der äußersten Schleife aktiviert.

Ein einfaches Beispiel soll diesen Sachverhalt verdeutlichen:


```
# Skript democont.sh: continue
#
for a in  1 2  ; do
   echo "For A $a"
   for b in  3 4  ; do
       echo "For B $b"
       for c in  1 2 3 ; do
           echo "continue $c"
#           continue $c
       done
   done
done
exit 0
```

Vergleichen wir einmal die Ausgaben von Skript `democont.sh`, einmal ohne `continue` (also so, wie es da steht: links) mit den Ausgaben des gleichen Skripts, wo `continue` durch Entfernen des `#` aktiviert wurde (rechts):

```
for A 1                          for A 1
for B 3                          for B 3
continue 1                       continue 1
continue 2                       continue 2
continue 3                       for B 4
for B 4                          continue 1
continue 1                       continue 2
continue 2                       for A 2       da die Schleife for B fertig ist
continue 3                       for B 3
for A 2                          continue 1
for B 3                          continue 2
continue 1                       for B 4
continue 2                       continue 1
continue 3                       continue 2    B ist beendet und A ebenfalls
for B 4
continue 1
continue 2
continue 3
```

3.8 Aufgaben

Wenn Sie an dieser Stelle angelangt sind, haben wir Ihnen nichts mehr zu sagen (zumindest was die Themen aus Kapitel 3 betrifft). Jetzt sind Sie am Zuge. Die folgenden Aufgaben sollen Ihnen die Chance geben, Ihr Wissen auf evtl. vorhandene Lücken zu prüfen.

1. Für Skript 15 hatten wir zwei Stellen aufgeführt, die mindestens noch fehlerhaft sind und dann die Lösung vorgestellt, die sicherstellt, dass nur auf Dateien zugegriffen wird. Kodieren Sie die Überprüfung des Bereichs bzw. der Zeilenanzahl, d.h. Sie sollen sicherstellen, dass die Anfangszeile nicht größer als die Anzahl an Zeilen in der Datei ist.
2. Skript 19 bedarf auch noch einiger Verbesserungen.
   a) Zum einen sollten Sie Parameter 1 überprüfen, ob es überhaupt ein Verzeichnis ist.
   b) Nehmen Sie noch einen dritten Parameter an Bord. Ist dieser gleich `-s`, so soll die Dateigröße jeder gefundenen Datei mit ausgegeben wer-

den. Fehlt der Parameter, so soll das Skript so ablaufen wie bisher. Aufruf: `skript19.sh <dir> <datei> <opt>`, wobei `<dir>` und `<datei>` bereits bekannt sind und `<opt>` entweder nicht angegeben ist oder gleich `-s` ist.

c) Wenn Punkt a und b erledigt sind, sollten Sie die Reihenfolge der Parameter umstellen. Typische Unixbefehle geben zuerst die Optionen an und danach erst die nötigen Parameter. Aufruf jetzt:

```
skript19.sh <opt> <dir> <datei>
```

Die Bedeutung ist identisch mit b), aber `<opt>` kann weggelassen werden!

3.9 Lösungen

1. Die Eingabeüberprüfung könnte folgendermaßen aussehen:

```
...
# Variablen zuweisen
datei=$1
ab=$2
bis=$3
#
#Eingabe überprüfen
#
if [ ! -f $datei  ] ; then
   echo "$datei ist keine Datei"
   exit 1
fi
if [ $ab -lt 0 ] ; then
   ab=1
fi
if [ $ab -gt $bis ] ; then
   echo "Bitte geben Sie einen sinnvollen Bereich an"
   exit 1
fi
...
```

2. Auch in Skript 19 sollte die Eingabe besser geprüft werden. Die Ausgabe kann nun über eine Option `-s` genauer gesteuert werden:


```
# Skript 19: Unsere erste while-Schleife
#
TMPFILE=/tmp/count
verz=$1
dateien=$2
opt=$3
#
#Eingabe testen (Aufgabe 2a)
#
if [ ! -d $verz ] ; then
   echo "$0: Usage <Verzeichnis> <Dateinamenmuster> [-s]"
   echo "$verz ist kein Verzeichnis"
   exit 1
fi
#
anz=`find $verz -name "$dateien" -type f -print | tee $TMPFILE | wc -l`
nr=0
summe=0
while [ $nr -lt $anz ] ; do
   nr=`expr $nr + 1`
   datei=`head -$nr $TMPFILE | tail -1`
   #
   # Falls Option -s, Dateigroesse mit ausgeben
   # (Aufgabe 2b)
   if [ "$opt" = "-s"  ] ; then
      echo "`wc -c <$datei` Bytes belegt die Datei $datei"
   else
      echo $datei
   fi
   #
   # Dieses cut muss angegeben werden
   #
   erg=`wc -c $datei | cut -c-7`
   summe=`expr $summe + $erg`
done
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $anz Dateien belegen $summe Bytes"
fi
rm $TMPFILE
exit 0
```

Den Positionsparameter an die erste Stelle zu verschieben und weiterhin optional zu belassen, kann durch eine case-Anweisung erreicht werden:


```
# Skript 19: Unsere erste while-Schleife
#
TMPFILE=/tmp/count
# abhängig von der Anzahl der übergebenen
# Parameter initialisieren
# (Aufgabe 2c)
case $# in
   2) verz=$1
      dateien=$2 ;;
   3) opt=$1
      verz=$2
      dateien=$3 ;;
   *) echo "$0: Usage [-s] <Verzeichnis> <Dateinamenmuster>"
      exit 1 ;;
esac
#
# Eingabe testen (Aufgabe 2b)
#
if [ ! -d $verz ] ; then
   echo "$0: Usage [-s] <Verzeichnis> <Dateinamenmuster>"
   echo "$verz ist kein Verzeichnis"
   exit 1
fi
...
```

Kapitel 4

Terminal-Ein-/Ausgabe

»Man sollte seinen Feinden verzeihen,
aber nicht, bevor sie am Galgen hängen« –
Heinrich Heine (1797–1856)

... unter diesen Umständen halten wir es für sinnvoll, ein einfaches Kapitel einzustreuen, schließlich wollen wir Sie nicht feindlich stimmen. Also widmen wir uns in diesem Kapitel ausschließlich der Terminal-Ein-/Ausgabe und noch ein wenig mit fortgeschrittenen Methoden zur Ein-/Ausgabeumlenkung. Mithilfe der Informationen aus diesem Kapitel werden Sie in die Lage versetzt, auch Skripten zu schreiben, die interaktiv auf Benutzereingaben reagieren.

4.1 Der echo-Befehl

Der Befehl `echo` ist Ihnen schon seit Kapitel 1 ein Begriff. Er gibt die ihm übergebenen Parameter auf die Standardausgabe aus. Dazu will ich auch kein Wort mehr verlieren, konzentrieren wir uns daher auf einige nette Kleinigkeiten, die bisher nicht erwähnt wurden.

Sicherlich haben Sie sich schon darüber geärgert, dass `echo` nach der Ausgabe in die nächste Zeile springt. Häufig ist es allerdings wünschenswert, dass die Ausgabe des folgenden `echo` direkt an die letzte Ausgabe anschließt und nicht eine Zeile tiefer steht. Geben Sie jedoch die Option `-n` an, so wird die Ausgabe nicht in die nächste Zeile springen.

Lassen Sie uns doch einfach Skript 19 nehmen und hinter jedem Dateinamen die Dateigröße ausgeben.


```
# Skript 19: Verbesserte Version ohne cut mit break
#            gibt Dateigröße aus
# Nutzt "break" in der "for"-Schleife, ansonsten
# keine Veränderungen
#
TMPFILE=/tmp/count
line=`find $1 -name "$2" -type f -print | tee $TMPFILE |wc -l`
anz=$line
nr=0
summe=0
while [ $nr -lt $anz ] ; do
  nr=`expr $nr + 1`
  datei=`head -$nr $TMPFILE | tail -1`
  echo -n $datei
  line=`wc -c $datei`
  for wort in $line ; do
     erg=$wort
     break
  done
  echo "    $erg"
  summe=`expr $summe + $erg`
done
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $anz Dateien belegen $summe Bytes"
fi
rm $TMPFILE
exit 0
```

Falls Sie Ihre Version genommen haben, die aus den Aufgaben von Kapitel 3 resultiert: Umso besser!

Ärgerlich ist jetzt eigentlich nur noch die Tatsache, dass die Ausgabe der Dateigrößen so unschön flattert. Besser wäre es, wenn die Ausgaben untereinander stehen würden. Die Ausgabe eines Tabulators würde da sicher helfen. Enthält ein Parameter für `echo` ein »\«, gefolgt von einem Zeichen aus Tabelle 4.1, so wird diese Zeichenkombination in ein Sonderzeichen umgesetzt.

Einige Versionen von `echo` ignorieren solche Zeichenkombinationen und überlassen der Shell eine eventuelle Interpretation bzw. geben die Zeichen eins zu eins aus. Falls z.B. `echo '\a'` nicht hupt, sondern "\a" ausgibt, nutzen Sie die Option `-e`. Soll zusätzlich auch noch der Zeichenumbruch unterbunden werden, so kombinieren Sie die Optionen zu `-ne`.

Es kann aber durchaus passieren, dass Sie an ein `echo` geraten, welches diese Option großzügig ignoriert (D-Nix und Ultrix kommen mir da so in den Sinn) oder nur sehr fehlerhaft umsetzt. In diesem Falle versuchen Sie es bitte mit `printf` (nächster Abschnitt), oder besorgen Sie sich ein `echo`, das funktioniert (Sourcen von GNU, siehe Anhang D).

Denken Sie auch an die Quotingregeln:

Ein `echo -e \a` gibt nur ein `a` aus, da das \ von der Shell geschluckt wird, bevor es an `echo` übergeben wird. Setzen Sie also besser Zeichenfolgen mit \ in Anführungszeichen.

`\a`	Hupe bzw. Glocke
`\b`	*Backspace*, gleiche Funktion wie die gleichnamige Taste
`\c`	Zeilenumbruch unterdrücken, identisch zur Option `-n`
`\e`	Gibt ein Escapezeichen (ASCII-Wert = 27) aus
`\f`	form feed
`\n`	Springt an den Zeilenanfang der nächsten Zeile
`\r`	Springt an den Zeilenanfang der aktuellen Zeile zurück
`\t`	Tabulatorausgabe
`\v`	Vertikaler Tabulator. Der Cursor springt genau eine Zeile tiefer, behält die Position innerhalb der Zeile allerdings bei.
`\\`	*Backslash*, gibt ein einfaches »\« aus
`\nnn`	Gibt das Zeichen aus, dessen ASCII-Wert `nnn` in oktaler Repräsentation (Basis 8) ist

Tabelle 4.1: Darstellung von Sonderzeihen für echo

Verbessern wir die Ausgabe also durch ein

```
echo "\t\t$erg"
```

Das ist allerdings noch nicht perfekt, da bei langen Dateinamen die Ausgabe der Dateigrößen zu weit nach rechts wandert. Solange wir aber nicht in der Lage sind, die Länge der Ausgabe zu bestimmen oder die Ausgabe genauer zu beeinflussen, müssen wir uns mit dem jetzigen Ergebnis bescheiden.

4.2 Der Befehl printf


```
printf <form1> <arg1> ...
```

Der Befehl `printf` wird Ihnen vielleicht schon begegnet sein, falls Sie Ihr geneigtes Auge jemals über ein C-Programm haben schweifen lassen. `printf` steht für *print formatted output*, also für die formatierte Zeichenausgabe.

`printf` nutzt einen Formatstring, der beschreibt, wie die Ausgabe der dem Formatstring `<form1>` folgenden Parameter `<arg1>` und evtl. weitere zu formatieren sind. Im Gegensatz zu C unterstützt die Shellversion nur die Formatierung von einzelnen Zeichen, von Zeichenketten, ganzen Zahlen (Integer) und reellen Zahlen in unterschiedlichen Formaten.

Wie sieht nun ein Formatstring aus?

Ein Formatstring ist nichts anderes als *eine* Zeichenkette, die als Ausgabeschablone für die weiteren Parameter von `printf` dient. Der Formatstring wird von links nach rechts nach bestimmten Zeichenmustern durchsucht. Das erste gefundene Zeichenmuster wird durch den Wert des zweiten Parameters von `printf` ersetzt, evtl. formatiert und ausgegeben. Das zweite Muster wird durch Parameter 3 ersetzt usw. Ist die Anzahl der gefundenen Zeichenmuster größer als die Anzahl der Parameter, die dem Formatstring folgen, so werden diese Zeichenketten durch Leerstrings `""` ersetzt.

Soweit, so gut, aber wie sehen diese mystischen Zeichenmuster nun aus? Jedes Muster wird durch ein `%` eingeleitet und durch einen Buchstaben aus der Menge `diuxXfeEgGcs` abgeschlossen. Dazwischen kann optional noch eine Breitenangabe folgen.

Wie das im Einzelnen aussieht, zeigt Tabelle 4.2.

Tabelle 4.2: Parameter für den Formatstring

Muster	Typ	Bemerkung
`%c`	ein Zeichen	Gibt das erste Zeichen des entsprechenden Parameters aus.
`%s`	Zeichenkette	Gibt eine Zeichenkette beliebiger Länge aus.
`%b`	Zeichenkette	Wie `%s`, nur werden hier Zeichenkombinationen wie `\n \v` usw. interpretiert. Diesen `printf`-Parameter gibt es nicht in C!
`%d`	Ganzzahl	Gibt eine Ganzzahl mit Vorzeichen aus.
`%i`	Ganzzahl	siehe `%d`

Muster	Typ	Bemerkung
%u	Ganzzahl	Gibt eine positive Ganzzahl aus. Negative Werte werden positiv. -2 wird z.B. zu 4294967294, da (bei 32 Bit pro Ganzzahl) beide Zahlen die gleiche (CPU-) interne Darstellung haben.
%f	reelle Zahl	Gibt eine Gleitpunktzahl aus. Solange nicht durch die Breitenangabe geändert, werden sechs Nachkommastellen ausgegeben (evtl. mit 0 aufgefüllt).
%e oder %E	reelle Zahl	Gibt eine Gleitpunktzahl in Exponentialschreibweise aus. Auch hier sechs Nachkommastellen, falls nicht anders definiert. Das Format ist: [-]<Mantisse mit sechs Nachkommastellen>e±<Exponent> bzw. [-]<Mantisse mit sechs Nachkommastellen>E±<Exponent>
%x	Ganzzahl	Gibt die Ganzzahl in hexadezimaler Schreibweise aus.
%X	Ganzzahl	Bei %x werden die Buchstaben a-f klein, bei %X groß ausgegeben.
%g oder %G	reelle Zahl	Falls der Exponent kleiner ist als -4, wird das Format %e bzw. %E verwendet, ansonsten %f
%%	kein Parameter	Gibt ein normales %-Zeichen aus.

Zeichenketten wie \a können ebenfalls im Formatstring aufgeführt werden. Sie haben die gleiche Bedeutung wie bei echo -e, benötigen aber keinen Parameter. Ihre Bedeutung wurde schon beim echo erklärt.

Kommen wir nun zur Breitenangabe, die zwischen »%« und dem Formatzeichen angegeben werden kann. Widmen wir uns zuerst der Breitenangabe bei Zeichen (%c) und Zeichenketten (%s). Die Breitenangabe kann sich aus folgenden optionalen Teilen zusammensetzen:

- Einem Minuszeichen, wenn die Parameterausgabe linksbündig durchgeführt werden soll.
- Einer Ganzzahl, die die minimale Breite der Ausgabe beschreibt. Der Parameter wird in einem Feld ausgegeben, dessen Breite mindestens diese Anzahl an Zeichen umfasst oder bei Bedarf auch mehr. Werden weniger Zeichen ausgegeben, als die minimale Breite verlangt, so wird abhängig vom ersten Punkt entweder links oder rechts mit Leerzeichen aufgefüllt.
- Einem Punkt, der die Feldbreite von der Genauigkeit trennt. Dies ist notwendig, wenn die Genauigkeit definiert werden soll.

- Der Genauigkeit als Ganzzahl. Definiert bei Zeichen und Zeichenketten die Zeichenanzahl, die maximal ausgegeben werden soll. Bei reellen Zahlen wird damit die Anzahl der Ziffern hinter dem Komma definiert, bei Ganzzahlen die minimale (!) Anzahl an Ziffern.

Was bedeutet dies nun praktisch? Schauen wir uns einmal die Ausgaben für `printf <form1> Australien` an. `<form1>` ersetzen wir durch die Angaben aus der folgenden Tabelle:

Form1	Ausgabe
`">%s<"`	`>Australien<`
`">%5s<"`	`>Australien<`
`">%.5s<"`	`>Austr<`
`">%20s<"`	`>          Australien<`
`">%-20s<"`	`>Australien          <`
`">%20.5s<"`	`>               Austr<`

Die Tatsache, dass die Schablone (Formatstring) eine Zeichenkette sein muss, hat zur Folge, dass Sie ohne Anführungszeichen bzw. Fluchtzeichen keine Leer- oder Sonderzeichen (wie hier das < und >) in den Formatstring einsetzen können. Vergessen Sie dies, so bricht die Shell den Formatstring in mehrere Wörter auf, und diese werden dann als Parameter interpretiert.

4.3 Der Befehl tput

Bevor wir das Thema Ausgabe endgültig zu den Akten legen können, möchten wir Sie noch auf einen Befehl aufmerksam machen, mit dem Sie die Attribute des Terminals (bzw. Bildschirms) manipulieren und ermitteln können.

Das bedeutet, dass Sie bestimmte Eigenschaften eines Terminals ermitteln können, z.B.:

- die Anzahl an Zeilen und Spalten,
- die Anzahl an Farben,
- die Anzahl an Leerzeichen, die durch einen Tabulator übersprungen werden,
- die Breite und Höhe des Bildschirms; das kann nützlich sein, wenn Ihre Skripten interaktiv Ein- und Ausgaben vornehmen sollen.

Zusätzlich können Sie natürlich auch Attribute ändern:

- die Vorder- und Hintergrundfarbe,
- die Cursorposition.

Weiterhin können Sie

- die aktuelle Zeile löschen,
- den gesamten Bildschirm löschen,
- Zeichen invers ausgeben,
- die Helligkeit der Zeichen (hell, dunkel) bestimmen.

Das sind nur einige der Attribute, die Sie ermitteln bzw. setzen können, aber für unsere Zwecke soll es reichen. Schauen wir uns nun die Parameter in Tabelle 4.3 in der Übersicht an. Die Schreibweise ist abhängig von Groß-/Kleinschreibung!

Tabelle 4.3: Attribute für tput

Parameter	Wirkung
`cols`	Gibt die Anzahl an Spalten des aktuellen Bildschirms (physikalisch) auf der Standardausgabe aus.
`lines`	Gibt die Anzahl an Zeilen des aktuellen Bildschirms (physikalisch) aus.
`it`	*ident tab*. Gibt die Anzahl an Zeichen (x) aus, die ein Tabulator maximal überspringt. Für x=8 springt der Cursor nach Spalte 1, 9, 17, 25 usw. Ziel ist dabei immer die nächstgrößere Spalte aus der so berechneten Serie.
`clear`	Löscht den Bildschirm, stellt den Cursor in die linke obere Ecke.
`el`	Löscht den Inhalt der aktuellen Zeile von der aktuelle Cursorposition ab.
`bold`	Ausgabe erfolgt heller.
`sgr0`	Die Ausgabe wird auf normal zurückgesetzt. Hebt `bold` und `rev` auf. Steht für *Set Graphics Rendition*.
`rev`	Ausgabe erfolgt invers.
`cup y x`	Setzt den Cursor an die Position `X/Y`, dabei ist `0 <= x < tput cols` und `0 <= y < tput lines`. Vorsicht! Jeder vernünftige Mensch gibt Koordinaten als X/Y-Position an. Unix macht da eine Ausnahme. Ich erinnere mich noch sehr gut an die ersten Gehversuche mit C und der (N)Curses-Bibliothek, aber das ist eine andere (wenig ruhmreiche) Geschichte.
`colors`	Gibt die Anzahl der Farben zurück, die der Bildschirm im Textmodus darstellen kann.

Parameter	Wirkung
`setaf x`	Setzt die Vordergrundfarbe `0<= x <= tput colors`. Steht für *Set ANSI Foreground.*
`setab x`	Setzt die Hintergrundfarbe `0<= x <= tput colors`. Steht für *Set ANSI Background.*
	x Farbe 0 Schwarz 1 Rot 2 Grün 3 Orange/Gelb (eine Frage der Interpretation :o)) 4 Blau 5 Lila 6 Hellblau (Cyan) 7 Grau

Diese Befehle sollten Sie sich unbedingt einprägen. Wenn Sie jemanden mit Unixwissen beeindrucken wollen, dann sollten

- der Befehl kryptisch
- die Optionen reichhaltig und verwirrend
- und die Effekte nützlich und beeindruckend

sein. Sicherlich werden Sie uns zustimmen, wenn wir behaupten, dass `tput` alle drei Bedingungen locker erfüllt.

Damit Sie nicht ins Schleudern kommen, wenn Sie jemand nach dem *Wieso* fragt:

Unix hat eine Datenbank, in der Beschreibungen über Fähigkeiten und Attribute einer *großen* Anzahl verschiedener Bildschirme abgelegt sind. Auf alten Systemen ist dies die `/etc/termcap`, während neuere Systeme eine verbesserte Version namens `terminfo` nutzen. Die Zuordnung, welches der Terminals aus der `terminfo` Ihren Bildschirm bzw. Terminal beschreibt, erfolgt über die Shellvariable `TERM`.

Steht in `TERM` z.B. `linux`, so wird die Beschreibung eines Terminals namens »`linux`« gesucht. Diese enthält eine Liste, was das Terminal kann und wie man bestimmte Funktionen ausführen kann. Diese Funktionen bekommen kurze, möglichst kryptische Namen (z.B. `cup` für Cursorpositionierung), die wiederum von `tput` erkannt und angesprochen werden.

Wie Sie eventuell eigene Funktionen einbauen können und welche Parameter es noch gibt, finden Sie mit `man terminfo` und/oder `man infocmp` heraus.

Es wird Zeit für eine Aufgabe, um das Gelernte auch praktisch anzuwenden. Erstellen wir also ein Skript, das einen Verzeichnisnamen als einzigen Parameter erwartet. Dieses soll mit `tar` in eine Datei mit festgelegtem Namen gesichert werden, und es soll ein Fortschrittsbalken inklusive der Prozentzahl ausgegeben werden. Der Prozentbalken soll in einer Box in der Mitte des Bildschirms erscheinen.

Bevor wir loslegen, müssen wir einige Fragen beantworten:

- Darf `tar` Ausgaben auf den Bildschirm machen?

  Nein, das zerstört unsere Positionierung, wir lenken die Ausgabe in eine Datei um.

- Wie ermittelt man die Prozentangaben?

  Mittels `find` erhält man alle Dateien im angegebenen Verzeichnis. Die Anzahl der Zeilen, die ausgegeben werden, wären also 100 Prozent. Dagegen könnte man die aktuelle Anzahl an Zeilen stellen, die `tar` bis zum Berechnungszeitpunkt übergeben wurden.

- Was machen wir mit Fehlern, die `tar` ausgibt?

  Wir lenken Sie um nach `/dev/null` und ignorieren sie erst einmal.

- Was ist sonst zu beachten?

  Wenn das Verzeichnis leer ist, besteht die Gefahr, dass durch 0 dividiert wird.


```
#!/bin/bash
# (Auf die Bedeutung der ersten Zeile werden
#  wir in 13.6.5 näher eingehen)
# Skript 25: tput und printf
# Sichert die Dateien im Verzeichnis $1 nach
# /tmp/bettina und gibt den Fortschritt aus.
#
# Box ausgeben
tput clear
tput cup 10 12
echo "+--------------------------------------------------+"
tput cup 11 12
printf "! %50s!" " "
tput cup 12 12
printf "! %50s!" " "
tput cup 13 12
echo "+--------------------------------------------------+"
# Anzahl Dateien im Verzeichnis ermitteln
verz=$1
if [ ! -d "$1" ] ; then
   echo "Aufruf: $0 Verzeichnis" >&2
```

```
   echo "  Verzeichnis  --> Das zu sichernde Verzeichnis " >&2
   exit 1
fi
anz=`find $verz -type f -print | tee /tmp/tar.cnt | wc -l`
# tar soll beim 1. Aufruf Archiv anlegen
taropt="cvf"
akt=0
while [ $anz -gt $akt ] ; do
  # Aktuelle Zeile ermitteln, entspricht Dateinamen
  akt=`expr $akt + 1`
  dnam=`head -$akt /tmp/tar.cnt | tail -1`
  # Ins Archiv damit
  tar $taropt /tmp/bettina.tar $dnam >/dev/null 2>&1
  # Beim nächsten Aufruf Option Datei anhängen
  taropt="rvf"
  tput cup 10 38
  erg=`expr  $akt \* 100 / $anz`
  echo "$erg%"
  tput cup 11 14
  printf "%-40.40s" $dnam
  pos=`expr $erg / 2 + 14`
  tput cup 12 $pos
  echo -n ":"
  sleep 1
done
rm /tmp/tar.cnt
tput cup 24 0
exit 0
```

Noch einige Worte zu dieser Lösung, bevor wir mit dem nächsten Abschnitt fortfahren. Dieses Skript hat nämlich mehrere Pferdefüße:

- Es ist extrem langsam. Das liegt daran, dass das Archiv zunächst mit nur einer Datei als Inhalt angelegt wird. Danach werden die Dateien durch Umstellung der Option von `cvf` auf `rvf` immer angehängt, was langsam ist.
- Die Methode mit `head | tail` ist äußerst aufwändig und führt ebenfalls zu einer Verlangsamung.

Falls Sie mit einem Vorgriff auf Kapitel 10 leben können, ersetzen Sie doch einfach

```
head -$akt /tmp/tar.cnt | tail -1
```

durch

```
sed -n -e "${akt}p" /tmp/tar.cnt.
```

Dies sollte schneller funktionieren.

In den folgenden Abschnitten und späteren Kapiteln werden wir dieses Skript noch gravierend verbessern, aber vorerst soll es reichen.

Wenn Sie wissen wollen, wieviel Zeit Ihr Skript wirklich verbraucht, so setzen Sie vor dem Aufruf des Skripts einfach ein `time`. Wird das Skript beendet, so werden drei Zeiten ausgegeben: die tatsächlich verstrichene Zeit und die Zeit, welche die CPU im Usermodus (also in Ihrem Skript) und im System (das sind Systemaufrufe im Kernel) verbracht hat.

```
buch@koala:/home/buch > time skript25.sh
/home/buch/skript
...
real    1m8.093s
user    0m1.970s
sys     0m1.900s
buch@koala:/home/buch >
```

Dies sind die Zeiten vom letzten Skript für ein Datenvolumen von ca. 140 Kbyte auf einem Pentium 2 400 MHz mit 128 Mbyte RAM, was nur die Aussage unterstreicht, dass dieses Skript suboptimal ist.

Dass die Summe der Zeiten von System und User nicht die Realzeit ergibt, hat mehrere Gründe, an dieser Stelle nur die offensichtlichsten:

- Zum einen ist Unix ein Multiuser-Multitaskingsystem. Das heißt, Ihr Skript ist nicht das einzige Programm, das zu einer gegebenen Zeit läuft. Alle gestarteten Programme im System laufen abwechselnd, jeweils nur für wenige Sekundenbruchteile.
- Zum anderen wartet Ihr Programm z.B. auf Daten von der Platte. Der Kernel gibt diese Anforderung an die Platte weiter, die ein wenig braucht, bis sie die Daten zur Verfügung stellen kann. In dieser Zeit wird Ihr Programm angehalten, und andere Programme kommen zur Ausführung.

Diese Verzögerungen führen zu der Differenz zwischen realer Zeit und den Zeiten für User und System.

4.4 Der Befehl read

Wenn ich mir so die Kapitelüberschrift anschaue, so fällt auf, dass bisher nur die Ausgabe behandelt wurde. Um bei Ihnen nicht in Ungnade zu fallen (betrachtet man sich das Kapitelmotto, so scheint das nicht empfehlenswert zu sein), beschäftigen wir uns in den restlichen Abschnitten mit Eingabemethoden in der Shell.

Die einfachste Methode ist der Einsatz von `read`. Wie bereits kurz in Kapitel 3 erwähnt, erwartet `read` als Parameter eine Variable, in der die eingelesenen Daten von der Standardeingabe abgespeichert werden. Dabei wird eine eventuelle Eingabeumlenkung beachtet, sodass `read` auch genutzt werden kann, um zeilenweise aus einer Datei zu lesen. Wie das genau geht, werden wir uns später in diesem Kapitel noch ansehen.

Nutzen wir nun dieses Wissen, um in Skript 25 die Dateinamen für den `tar`-Befehl mittels `read` aus der temporären Datei zu lesen. Dazu fügen wir eine Zeile zwischen `find` und der `while`-Schleife ein, mit der wir einen weiteren Eingabekanal definieren:

```
4</tmp/tar.cnt
```

Die Variable `dnam` bestimmen wir nun wie folgt:

```
read dnam <&4
```

Lassen wir das verbesserte Skript einmal laufen und messen die Zeit:

```
real    1m8.242s
user    0m1.550s
sys     0m1.610s
```

Das Skript ist scheinbar langsamer, aber berücksichtigen Sie dabei, dass Linux ein Multitasking/Multiusersystem ist. Diese Tatsache nutze ich auf meinem System zur Genüge aus. Deshalb fällt die Systemauslastung unterschiedlich aus, wodurch die Differenz noch in den Toleranzbereich fällt.

Nutzen wir nun `read`, um interaktive Skripten zu erstellen, die Eingaben per Tastatur vom Benutzer erwarten und darauf reagieren. Wenn Sie Eingaben verlangen, so ist es sinnvoll, einen Prompt auszugeben, der dem Benutzer klarmacht, dass das Skript eine Eingabe erwartet. Dabei ist es vorteilhaft, wenn der Cursor direkt hinter dem Prompt steht und nicht eine Zeile tiefer. Bisher würden Sie dieses Problem wie folgt lösen:

```
...
echo -n "Eingabe:"
read ein
...
```

Lassen Sie uns nun ein interaktives Skript entwickeln, welches wir über den Rest des Buches hinweg weiter verbessern werden. Wenn es fertig ist, soll es eine vereinfachte Version des *Norton-Commander (R)* darstellen. Nennen wir es `cwc.sh`.

Fangen wir ganz einfach an und geben eine Box aus von Position 0,1 mit einer Breite von 40 Zeichen und einer Höhe von 20 Zeilen und darin unformatiert den Inhalt des aktuellen Verzeichnisses. Wenn mehr Dateien vorhanden sind, als Zeilen verfügbar sind, so sollte man mit den Eingaben `v` und

z seitenweise vor- und zurückblättern können. q steht für Quit und beendet das Skript. Das aktuelle Verzeichnis sollte in der ersten Zeile der Box (sprich auf der Umrandung) erscheinen.

Wie könnte das Skript also aussehen? Wir schlagen Ihnen folgendes Vorgehen vor:

1. Variablen initialisieren und aktuellen Verzeichnisinhalt in temporäre Datei schreiben
2. Ausgabe der Box

   Skript läuft, solange Eingabe nicht q (`until`-Schleife)
3. Ausgabe Verzeichnis (Titel und Inhalt)
4. Eingabe entgegennehmen
5. Eingabe bearbeiten (außer q)
6. Wenn Eingabe ungleich q, dann weiter bei 2.
7. Temporäre Datei löschen

Problematisch könnte Punkt 4 werden und zwar dann, wenn mehr als die 18 Einträge ausgegeben werden sollen, die in der Box auf einmal angezeigt werden können. Teil 1 des Problems besteht darin, eine Ausgabe von mehr als 18 Einträgen vorzunehmen. Hatten wir »Problem« geschrieben? Einfach einen Zähler für die Anzahl der ausgegebenen Zeilen mitlaufen lassen, und schon können wir getrost das Gegenteil behaupten.

Teil 2 ist schon etwas kniffliger, sollen doch die Ausgaben seitenweise durchzublättern sein. Eine Lösung zeichnet sich ab, wenn Sie sich überlegen, welche Zeilen auf welcher Seite angezeigt werden sollen:

Seite	**Zeile**
1	1–18
2	19–36
3	37–54

Deutlich zu erkennen, dass für jede Seite ein Versatz von 18 Zeilen einzuplanen ist. Da wir aber keine Seiten direkt anspringen, reicht es aus, den Versatz (Offset auf Englisch) mit 1 zu initialisieren und bei der Eingabe von v und z um jeweils 18 Zeilen zu erhöhen bzw. zu erniedrigen. Zu beachten ist dabei lediglich, dass der Offset nicht kleiner als 1 oder größer als die Anzahl der anzuzeigenden Dateinamen werden kann.

Ihr Lösungsansatz könnte also so aussehen:


```
# cwc.sh (Skript 26)
#
# CW-Commander. Erste Miniversion. Hier fehlt mehr als
# funktioniert, aber wir wollen die Hoffnung nicht aufgeben,
# schließlich kommen noch einige Kapitel
#
# Box und Verzeichnis ausgeben
tput clear
tput cup 1 0
echo "+--------------------------------------+"
i=2
while [ $i -lt 21 ] ; do
  tput cup $i 0
  printf "! %38s!" " "
  i=`expr $i + 1`
done
tput cup 21 0
echo "+--------------------------------------+"
tput cup 1 2
printf "%.35s" `pwd`
#
# Zeilen Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
offset=1
tmpfile="/tmp/cwc.tmp"
anz=`ls |tee $tmpfile | wc -l | cut -c-7`
akt=1
until [ "$ein" = "q" ] ; do
  # Ausgabe der Dateien
  i=0
  akt=0
  while [ $i -lt 19 -a $akt -le $anz ] ; do
    akt=`expr $i + $offset`
    tput cup `expr $i + 2` 2
    zeile=`head -$akt $tmpfile | tail -1`
    printf "%-38.38s" "$zeile"
    i=`expr $i + 1`
  done
  # evtl. Leerzeilen tatsächlich löschen
  while [ $i -lt 19 ] ; do
    tput cup `expr $i + 2` 2
    i=`expr $i + 1`
    printf "%38.38s" " "
  done
  ein=""
```

```
  # Eingabeschleife
  until [ "$ein" != "" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
                 offset=`expr $offset + 18` ;
                 if [ $offset -gt $anz ] ; then
                    offset=`expr $offset - 18`
                 fi ;;
      "z" | "Z")
                 if [ $offset -gt 1 ] ; then
                    offset=`expr $offset - 18` ;
                 fi ;;
       *) ein="" ;;
     esac
  done
done
exit 0
```

Ein letzter Hinweis zu diesem Skript: Da der Benutzer beliebige Zeichen eingeben kann, würde ohne die zweite `until`-Schleife jede Fehleingabe zu einem erneuten Aufbau des Bildschirms führen. Da dies unschön ist, fangen wir ungültige Eingaben ab.

Falls Sie beim Blättern ein Überlappen der Ausgaben bevorzugen, d.h., dass jeweils eine Datei beim Umblättern erhalten bleibt, dann sollten Sie den Offset auf 17 verändern.

Die Option -p wird Ihnen in in Kapitel 12.1.6 noch genauer erklärt. Hier nur so viel. Diese Option funktioniert so nur in den neueren Bash Versionen.

4.5 Eingabe mit select

OK, das funktioniert ja schon recht gut, aber diese `until`-Schleife für die Eingaben nagt doch ein wenig an Ihnen, geben Sie es ruhig zu! `:)` Es wäre doch viel eleganter, der Eingabe vorzuschreiben, welche Eingaben akzeptabel sind. Dies geht mit dem `select`-Befehl.


```
select <var1> in <wortliste> ; do
  <befehl1>
  <befehl2>
  ...
done
```

Dabei druckt `select` alle Wörter aus `wortliste` untereinander aus und numeriert sie von 1 bis x durch. Der Benutzer kann dann nur die Zahlen von 1 bis x als gültige Eingaben vornehmen. Dabei bekommt `<var1>` den Wert des ersten Wortes zugeordnet.

Schauen wir uns mal an, wie die Eingabe mit `select` aussieht:

```
# Skript 27: select
#
# Bietet die Dateien, die Parameter 1 definiert, zur Bearbeitung
# an. Die Dateien werden dann mit dem Editor joe bearbeitet.
# Wird dieser verlassen, werden erneut alle Dateien zur Auswahl
# gestellt.
#
echo "Welche Datei soll bearbeitet werden?"
select wort in $1 ; do
     joe $wort
done
echo "Ende"
exit 0
```

Rufen wir das Skript einmal mit dem Parameter `*.txt` auf, und nehmen wir an, es existieren die Dateien `kapitel1.txt`, `kapitel2.txt`, `kapitel3.txt`, `toc.txt`, `anhanga.txt` und `kapitel4.txt`. Dann sieht die Ausgabe wie folgt aus:

```
buch@koala:/home/buch > ./skript27.sh \*.txt
1) anhanga.txt     3) kapitel1.txt    5) kapitel3.txt    7) toc.txt
2) einleitung.txt  4) kapitel2.txt    6) kapitel4.txt
#? 1
```

`<joe anhanga.txt` wird ausgeführt>

```
1) anhanga.txt     3) kapitel1.txt    5) kapitel3.txt    7) toc.txt
2) einleitung.txt  4) kapitel2.txt    6) kapitel4.txt
#?
```

Stellt sich nur die Frage, wie sich die `select`-Anweisung abbrechen lässt, ohne dass das Skript ebenfalls beendet wird (damit scheidet Strg+C wohl aus). Die ↵-Taste hilft uns offensichtlich nicht aus dem Dilemma, gibt `select` doch einfach die Auswahl nochmals aus und besteht auf einer gültigen Auswahl.

Wenn Sie jedoch bei der Eingabeaufforderung Strg+D drücken (EOF-(End of file-)Zeichen), so läuft das Skript weiter und beendet sich korrekt. Was aber, wenn das `select` verlassen werden soll, weil eine Bedingung im Skript erfüllt wurde? Auch dies geht, und hier begegnet uns ein alter Bekannter: `break`. Wird dieser Befehl innerhalb von `select` ausgeführt, so wird der erste Befehl nach dem `done` ausgeführt und somit die `select`-Anweisung verlassen. Den gleichen Effekt erreichen Sie auch durch `return`.

Im Übrigen setzt `select` neben `var1` noch eine Variable namens `REPLY` automatisch. Sie enthält den Wert, den Sie ursprünglich eingegeben hatten. Falls Sie etwas Ungültiges eingegeben hatten, so ist `var1` leer, und nur `REPLY` enthält diese fehlerhafte Eingabe.

Dies kann sich ein Skript auf vielfältige Weise zunutze machen. Beispielsweise könnten Sie die Eingabe als Shellbefehl interpretieren und ausführen. Ändern wir das Skript 27 leicht ab:


```
# Skript 27: Select #2
#
# Bietet die Dateien, die Parameter 1 definiert, zur Bearbeitung
# an. Die Auswahl von "<ENDE>" beendet das Skript
echo "Welche Datei soll bearbeitet werden?"
select wort in $1 "<ENDE>"; do
  if [ "$wort" != "" ] ; then
     if [ "$wort" = "<ENDE>" ] ; then
        break
     fi
    joe $wort
    else
     $REPLY
  fi
done
echo "Ende"
exit 0
```

Sicherlich haben Sie neben den Vorteilen auch einen klaren Nachteil ausfindig gemacht:

Die Auswahl wird immer an der Cursorposition ausgegeben, wodurch früher oder später der Bildschirminhalt nach oben geschoben wird. Für Skripten wie Skript 26 ist dies weniger erfreulich, zerstört das doch eventuell den Bildschirmaufbau.

Sinnvoll ist der Einsatz von `select` also nur, wenn Sie eine Eingabe aus einer Liste definierter Vorschläge haben wollen und der Bildschirmaufbau unwichtig ist.

4.6 Die Anwort auf die unausweichliche Frage: Tastatur und ihre Abfrage

Wir hofften ja, Sie hätten keine Fragen mehr. Aber es ist wohl unvermeidlich, dass dieses Thema im Rahmen dieses Kapitel auftauchen würde. Keine Bange, Sie stellen sich mit dieser Frage nicht ins Abseits, und eine Rüge von uns

bekommen Sie auch nicht, schließlich gibt es keine dummen Fragen, nur dumme Antworten. Also, nur Mut!

Wie kann ich einzelne Tasten abfragen?

Na bitte, das war doch nicht so schwer, oder? Wir fürchten allerdings, so wie uns die Frage nicht gefiel, so wird Ihnen die Antwort nicht gefallen:

Das geht im Skript nicht ohne Aufwand!

Schade, aber wirklich nicht zu ändern. So angenehm das auch wäre, aber weder Bash noch Bourne-Shell bieten einen Befehl à la `getkey` an, der genau eine Taste abfragt und deren Code zurückgibt. Am ehestens kommen Sie an diese Funktion noch wie folgt heran:


```
# Getkey: Eine Annäherung in der Shell
#
gvKey=''
while test "$gvKey" != "q" ; do
   echo -n "Drück mich :) "
   gvKey=`stty raw -echo ; dd bs=1c count=1 2>/dev/null ; stty -raw echo`
   echo " Das war ein ($gvKey)";
done
exit 0
```

Interessant dürfte wohl primär die zweite Zeile innerhalb der Schleife sein. Das `stty raw -echo` schaltet für die Eingabe den *Rawmodus* ein (`raw`) und das `echo` (sprich die Wiedergabe der eingetippten Zeichen) aus. Im Rawmodus können Sie die Tasten einzeln abfragen, und Strg+C, Strg+S usw. haben keine Funktion.

Mit `dd` (*Disk Dump*) können Sie Daten von der Standardeingabe auf die Standardausgabe kopieren und dabei auf Wunsch bestimmte Umwandlungen vornehmen. Wir wollen genau einen Datensatz (`count=1`) mit der Blockgröße von einem Byte (`bs=1c` --> Blocksize = 1 Character), also einen Tastendruck. Eventuelle Fehlermeldungen werden ignoriert (`2>/dev/null`).

Im letzten Schritt stellen wir die normale Tastaturabfrage wieder ein (`-raw echo`).

Wenn dies ein C-Buch über Unixprogrammierung wäre, könnten wir hier lang und breit erklären, wie ein C-Programm aussehen müsste, welches diese Funktion anbietet, aber das geht über den Rahmen des Buches weit hinaus.

Damit Sie aber nicht auf diese Funktion verzichten müssen, haben wir ein *kleines* C-Programm erstellt, das Sie in Anhang C finden. Dieses fragt genau eine Taste ab und gibt deren Code zurück.

Dieses Programm hat allerdings ein paar Nachteile:

- Wir können leider nicht garantieren, dass es auf allen Systemen läuft, da uns zum Testen nur ein Linuxsystem zur Verfügung steht. Allerdings sollten alle Routinen auf POSIX-konformen Systemen verfügbar sein.
- Strg+S, Strg+Q, Strg+C und Strg+Z werden nicht abgefangen.
- Der größte Nachteil: Es ist nicht terminalunabhängig. Tasten wie F1 geben mehr als ein Zeichen zurück. Das Programm geht von bestimmten Zeichenfolgen aus und fragt nicht die `terminfo` ab. Selbst mit diesen Nachteilen haben Sie ein Programm, welches jede Taste abfragt und alle druckbaren Zeichen wieder ausgibt (`a-z`, `0-9` usw.).
- ← gibt 127 zurück.
- Strg+X, wobei x ein Zeichen von `a-z` (ohne `c`,`q`,`s` und `z`) ist, geben die Position des Buchstabens im Alphabet zurück (Strg+A gibt 1 zurück, Strg+D ergibt 4 usw.)

Mehr Informationen und den Quellcode finden Sie in Anhang C. Falls Sie das Programm nicht zum Laufen bekommen: Überall, wo wir in Zukunft `gk` einsetzen werden, tauschen Sie es einfach durch `read variable` oder durch Ihr Skript `GetKey.sh` aus. Sämtliche Skripten werden wir so kodieren, dass sowohl einzelne Tastencodes als auch normale Eingaben zum Ziel führen.

4.7 Ein-/Ausgabeumlenkung für Experten

In Kapitel 2 haben wir uns ja bereits mit der Ein-/Ausgabeumlenkung beschäftigt. Einige der fortgeschrittenen Möglichkeiten hätten dort aber nur verwirrt, deshalb haben wir sie bis jetzt aufgespart.

4.7.1 Eingabeumlenkung durch Kanalduplizierung

Falls Sie schon versucht haben, mittels < und `read` mehr als eine Zeile zu lesen, so dürften Sie Probleme bekommen haben. Wahrscheinlich haben Sie immer nur die erste Zeile der Datei lesen können. Um auch auf die folgenden Zeilen zuzugreifen, gibt es eine Möglichkeit, die Eingabe auf einen neuen Kanal mit einer Nummer größer 2 zu kopieren. Die Syntax dazu lautet: `nr<datei`. Eine Umlenkung von diesem Kanal hat dann die Syntax: `<&nr`.

Was bringt Ihnen das nun ein? Wenn Sie den Eingabekanal duplizieren und sukzessiv die Eingabeumlenkung mit der neuen Kanalnummer vornehmen, so lesen Sie eine Zeile nach der anderen ein, während ein Umlenken via `<datei` immer nur die erste Zeile einliest.

Ein Kanal mit einer höheren Nummer als 2, den Sie wie oben beschrieben geöffnet haben, muss am Ende wieder geschlossen werden durch den Befehl `nr <&-`. Zum Thema Kanalduplizierung möchten wir Sie nochmals auf Kapitel 2 hinweisen, wo wir die Ein-/Ausgabekanäle genauer betrachtet haben.

Zum Einlesen von Daten der Standardeingabe gibt es den Befehl `read`. Dieser Befehl erwartet eine Variable als Parameter. Die eingelesene Zeile von der Standardeingabe wird der Variablen zugewiesen. Eventuelle Eingabeumlenkungen werden von `read` natürlich beachtet.

Ein Beispiel hierzu: Die Datei `tst.txt` enthält zwei Sätze in zwei Zeilen.

```
buch@koala:/home/buch > cat tst.txt
Ein Koala ist ein Beuteltier
Delphine sind Säugetiere
```

Das folgende Skript zeigt zwei Versuche, beide Zeilen hintereinander auszugeben.


```
# tst.sh: Eingabeumlenkung und Kanalduplizierung
#
read zeile <tst.txt
echo "1. $zeile"
read zeile <tst.txt
echo "2. $zeile"
exec 4<tst.txt
read zeile <&4
echo "3. $zeile"
read zeile <&4
echo "4. $zeile"
exit 0
```

Erst der zweite Versuch über `&4` führt zum Erfolg:

```
buch@koala:/home/buch > ./tst.sh
1. Ein Koala ist ein Beuteltier
2. Ein Koala ist ein Beuteltier
3. Ein Koala ist ein Beuteltier
4. Delphine sind Säugetiere
buch@koala:/home/buch >
```

In der Kornshell funktioniert eine Umlenkung nur mittels exec. Sie übernimmt Änderungen an den Dateidiskreptoren nur mittels `exec` in die aktuelle Shellumgebung.

```
...
exec 4<tst.txt
read zeile <&4
...
```

Haben Sie es auch zuerst mit Kanal 3 versucht? Auch hier dürfte es nicht geklappt haben, denn dieser Kanal wird von der bash für eigene Zwecke intern verwendet.

4.7.2 Here-Documents

Kommen wir nun zur nächsten Art der Eingabeumlenkung, den Here-Documents.

Mit dieser Art der Eingabeumlenkung wird die Shell angewiesen, Ihre Standardeingabe bis zu einem Stoppwort aus dem aktuellen Skript zu lesen. Sonderzeichen innerhalb des Stoppworts (wie $, * und `) werden *nicht* interpretiert, sondern eins zu eins übernommen. Alle folgenden Zeilen bis zum Stoppwort werden eingelesen und dem Befehl als Standardeingabe zugeführt. Bevor die gelesenen Zeilen allerdings weitergegeben werden, interpretiert die Shell die Zeilen auf bekannte Weise, was sehr hilfreich ist.

Die Umlenkung wird durch ein <<, gefolgt vom Stoppwort eingeleitet. Die folgenden Zeilen bis zum Stoppwort werden interpretiert und danach umgeleitet. Das Stoppwort selbst *muss* identisch sein zur Angabe hinter dem << und in der ersten Spalte anfangen. Schon ein Leerzeichen am Ende führt dazu, dass es nicht gefunden werden kann. Ein einfaches Beispiel:


```
# Here-Dokumente: Skript 13
#
cat <<EOF
Koala
Sydney
Melbourne
`pwd`
EOF
echo "Ende"
exit 0
```

Skript 13 liefert folgendes Ergebnis:

```
Koala
Sydney
Melbourne
/home/buch
Ende
buch@koala:/home/buch >
```

Natürlich kann diese Art der Umlenkung sehr gut genutzt werden, um Skript 12 aus Kapitel 2.1.3 eleganter zu lösen. Dazu benötigen Sie aber zusätzlich den `find`-Befehl. Dieser Befehl sucht in einem angegebenen Verzeichnis und den eventuell darunter liegenden Verzeichnissen nach allen Dateien, die den angegebenen Kriterien entsprechen, und gibt sie aus. Die Syntax lautet:

```
find <dir> <Optionen> -print
```

wobei für `<Optionen>` eine oder beide der folgenden Möglichkeiten eingesetzt werden kann bzw. können:

`-name <name>`	Sucht alle Dateien bzw. Verzeichnisse aus, deren Name gleich `<name>` ist.
`-type f`	Sucht alle normalen Dateien heraus. Wenn `-name` angegeben wurde, werden Dateien mit dem angegebenen Namen gesucht.
`-type d`	Sucht alle Verzeichnisse. Mit `-name` werden dann Verzeichnisse mit dem angegebenen Namen gesucht.

Es gibt auch einen Befehl namens `locate`. Dieser Befehl erlaubt (zumindest in GNU-Versionen) nur das Suchen nach Namen, eine Unterscheidung nach `-type` ist nicht möglich. In Kapitel 12 werden wir uns diesen Befehl noch einmal anschauen, um ihn anstelle von `find` einsetzen zu können.

Wenden wir diese Informationen nun auf eine neue Version von Skript 12 an:


```
# Skript 12: (Vorerst) Letzte Version
#
tmpfile="/tmp/erg"
ls $1  > $tmpfile
ls $2 >> $tmpfile
echo "Die Verzeichnisse $1 und $2 enthalten `wc -l <$tmpfile` Dateien"
# Jetzt mit Here-Dokument
anz=`wc -l <<ENDE
\`find $1 -type d -print\`
\`find $2 -type d -print\`
ENDE`
echo "Insgesamt existieren genau" $anz "Unterverzeichnisse"
rm $tmpfile
exit 0
```

Warum ist jetzt das `find` nochmals mit dem Fluchtzeichen versehen? Weil innerhalb der ersten Befehlsersetzung mittels »`« eine weitere Ersetzung vorgenommen werden soll, muss das Fluchtzeichen verwendet werden. Denn zunächst ermittelt die Bash die Zeichenkette, die ausgeführt werden soll. Da die beiden »`« des `find`-Befehls mit Fluchtzeichen versehen sind, werden sie (natürlich ohne Fluchtzeichen) in diese Zeichenkette übernommen:

```
wc -l <<ENDE
`find $1 -type d -print`
`find $2 -type d -print`
ENDE
```

Wenn die Shell nun diesen Befehl ausführen will, werden zuerst die beiden `find`-Befehle ausgeführt und die Ergebnisse an die Standardeingabe von `wc` weitergegeben.

Abschließend noch zwei Funktionalitäten, die mittlerweile sowohl in der Ksh als auch in der Bash (ab 2.05) vorkommen. Da wäre zuerst das Here-Dokument `<<-`. Dies hat im wahrsten Sinne des Wortes nur kosmetische Auswirkungen: Alle führenden Tabulatoren werden vom Zeilenanfang entfernt und der Rest wird wie bei den bereits besprochenen Here-Dokumenten bearbeitet. Dies ist sinnvoll, wenn Sie Ihre Skripte einrücken, um die Übersicht zu behalten, und dann in einer tiefen Einrückung ein Here-Dokument nutzen wollen. Dieses können Sie nun ebenfalls einrücken.

Bitte beachten Sie, dass ausschließlich Tabulatoren vom Zeilenanfang entfernt werden, keine Leer- oder sonstigen Kontrollzeichen!

Im folgenden Shellskript `here.sh` sind die Einrückungen der beiden Here-Dokumente durch Tabulatoren erreicht worden.

```
#!/bin/bash
echo "<<-"
cat <<-EOF
1
       2
       3
EOF
echo "<<"
cat << EOF
1
       2
              3
EOF
exit 0
```

Ein Aufruf des Skriptes zeigt uns die unterschiedliche Wirkung von << und <<-:

```
buch@koala:/home/buch > ./here.sh
<<-
1
2
3
<<
1
        2
                3
buch@koala:/home/buch >
```

Als letzten Punkt hätten wir da noch die Here-Zeichenkette. Das Format lautet:

```
<<<word
```

Die Shell interpretiert `word` und nimmt das Ergebnis der Interpretation als Eingabewert(e) des Here-Dokumentes. Nehmen wir das oben beschriebene `here.sh`, um ein einfaches Beispiel zu konstruieren:

```
buch@koala:/home/buch > cat <<<`here.sh`
1 2 3 1 2 3
buch@koala:/home/buch >
```

4.8 Aufgaben

1. Nehmen wir noch einmal Skript 13 und ändern es wie folgt ab:


```
# Here-Dokumente: Skript 13
#
cmd=pwd
cat <<$cmd
Zeile 1
`echo "Hallo Welt"`
pwd
$cmd
$cmd
echo "Ende"
exit 0
```

Da die Shell ja zunächst Variablen durch ihre Werte ersetzt und dann erst die Wörter als Befehle interpretiert, führt `$cmd` in diesem Beispiel zu `pwd`. Dieser Befehl gibt ja bekanntermaßen das Arbeitsverzeichnis aus. Die Frage: Wie oft gibt das obige Skript das Arbeitsverzeichnis aus? Ein-, zwei- oder gar dreimal?

2. Sehen Sie sich die letzte Version von Skript 12 an, und erweitern Sie es so, dass die Anzahl der Unterverzeichnisse für jedes übergebene Verzeichnis aufgeschlüsselt wird.
3. Korrigieren Sie die letzte Version von Skript 12 so, dass Fehler von `find` unterdrückt werden. Der Befehl `find` gibt Fehler aus, wenn er keine Berechtigung hat, ein Verzeichnis zu durchsuchen.
4. Nutzen Sie die Kanalduplizierung, um in Skript 25 (Kapitel 4.3) die Dateinamen für den `tar`-Befehl mittels `read` aus der temporären Datei zu lesen. Dazu fügen wir eine Zeile zwischen `find` und der `while`-Schleife ein, mit der wir einen weiteren Eingabekanal definieren.
5. Lösen Sie das Problem von Skript 19 mit `printf`.
6. Setzen Sie Skript 25 so um, dass die Ausgabebox zentriert auf den Bildschirm ausgegeben wird und nicht auf 80x25 festgelegt ist.
7. Es war ja zu befürchten: `cwc.sh` verlangt einige Verbesserungen. Hier soll uns aber erst einmal eine reichen:
   Geben Sie abhängig vom Typ der Datei den Namen in unterschiedlichen Farben aus (Verzeichnisse blau, Tar-Archive rot, Zip-Archive lila.
8. Erstellen Sie ein Skript mit dem `select`-Befehl, das Ihnen vom aktuellen Verzeichnis alle Verzeichnisse zur Auswahl anbietet. Wird ein Verzeichnis ausgewählt, so soll der Inhalt ausgegeben werden. Wird kein Verzeichnis ausgewählt, dann soll versucht werden, die Eingabe als Befehl abzusetzen.

4.9 Lösungen

1. Das Arbeitsverzeichnis wird genau einmal ausgegeben. Was ist passiert? Die Zuweisung `cmd=pwd` führt zum Ersetzen der Variablen namens `cmd`, nicht aber beim Stoppwort. Der Befehl `cat` wird bis zum ersten Erscheinen der Zeichenkette `$cmd` ausgeführt. Daher werden die Zeilen

   ```
   Zeile 1
   Hallo Welt
   pwd
   /home/buch
   Ende
   ```

   ausgegeben. `"Hallo Welt"` wird ausgedruckt, da der `echo`-Befehl in Ausführungszeichen gefasst ist. Erst das dritte `$cmd` ist nach der Variablenersetzung der Shell der Befehl `pwd` und führt zur Ausgabe des aktuellen Arbeitsverzeichnisses.

2. Um für jedes übergebene Verzeichnis die Anzahl seiner Unterverzeichnisse zu ermitteln, könnte folgendes Skript verwendet werden:


```
# Lösung 12:
# Anzahl an Unterverzeichnissen mittels
# HERE-Document ermitteln
tmpfile="/tmp/erg"
ls $1 > $tmpfile
echo "Das Verzeichnis $1 enthält `wc -l <$tmpfile` Dateien"
ls $2 > $tmpfile
echo "Das Verzeichnis $2 enthält `wc -l <$tmpfile` Dateien"
# Jetzt mit Here-Dokument
anz=`wc -l <<ENDE
\`find $1 -type d -print\`
ENDE`
echo "In Verzeichnis $1 existieren genau" $anz "Unterverzeichnisse"
anz=`wc -l <<ENDE
\`find $2 -type d -print\`
ENDE`
echo "In Verzeichnis $2 existieren genau" $anz "Unterverzeichnisse"
rm $tmpfile
exit 0
```

3. Skript 12 mit Fehlerunterdrückung:


```
# Skript 12: mit Fehlerunterdrückung
#
anz=`wc -l <<ENDE
\`find $1 -type d -print 2>/dev/null\`
\`find $2 -type d -print 2>/dev/null\`
ENDE`
echo "Insgesamt existieren genau" $anz "Unterverzeichnisse"
exit 0
```

4. Vor der `while`-Schleife fügen wir noch die Zeile

   `4</tmp/tar.cnt`

   ein. Die Variable `dnam` in der Schleife bestimmen wir nun wie folgt:

   `read dnam <&4`

5.


```
# Skript 19:
#   Sucht im Verzeichnis $1 und dessen Unterverzeichnissen
#   nach Dateien mit dem Muster $2 und gibt die Summe
#   der gefundenen Dateigrößen aus.
#
# 1. Initialisierung
TMPFILE=/tmp/count
line=`find $1 -name "$2" -type f -print | tee $TMPFILE |wc -l`
anz=$line
nr=0
summe=0
#
# 2. Für jede gefundene Datei:
#    Größe bestimmen und summieren
#
while [ $nr -lt $anz ] ; do
  nr=`expr $nr + 1`
  datei=`head -$nr $TMPFILE | tail -1`
  echo -n $datei
  line=`wc -c $datei`
  for wort in $line ; do
     erg=$wort
     break
  done
  echo "    $erg"
  summe=`expr $summe + $erg`
done
#
# 3. Ergebnis in die Welt hinausposaunen
#
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   printf "Insgesamt %5d Dateien belegen %12d Bytes\n" $anz $summe
fi
rm $TMPFILE
exit 0
```

6. Im Skript 25 muss zunächst für eine zentrierte Ausgabe des Statusfensters die Anzahl der Spalten und Zeilen des aktuellen Bildschirms (bzw. Fensters unter X-Windows) ermittelt werden. Wenn die ermittelte Anzahl der Spalten (bzw. Zeilen) um die Anzahl Spalten (bzw. Zeilen) des Fensters selbst verringert und das Ergebnis anschließend halbiert wird, dann erhält man die Anfangsposition (linke obere Ecke des Statusfensters). Anschließend müssen alle fixierten Ausgaben entsprechend verschoben werden.


```
# Skript 25: tput und printf
#  tar-Archiv mit Fortschrittsbalken ausgeben.
#  Die Box wird dabei zentriert ausgegeben.
#
# Box ausgeben
tput clear
#
# Bildschirmzeilen und -spalten merken und so
# verschieben, dass es zentriert ausgegeben wird
# (Aufgabe 6)
#
zeilen=`tput lines`
spalten=`tput cols`
x=`expr \( $spalten - 54 \) / 2`
   y=`expr \( $zeilen - 4 \) / 2`
#
# Alle fixierten tputs entsprechend abändern:
#
tput cup $y $x
echo "+--------------------------------------------------+"
tput cup `expr $y + 1` $x
printf "! %50s!" " "
tput cup `expr $y + 2` $x
printf "! %50s!" " "
tput cup `expr $y + 3` $x
echo "+--------------------------------------------------+"
# Anzahl Dateien im Verzeichnis ermitteln
verz=$1
if [ ! -d "$1" ] ; then
   echo "Aufruf: $0 Verzeichnis"
   echo "  Verzeichnis  --> Das Verzeichnis welches gesichert werden soll"
   exit 0
fi
anz=`find $verz -type f -print | tee /tmp/tar.cnt | wc -l`
# tar soll beim 1. Aufruf Archiv anlegen
taropt="cvf"
akt=0
while [ $anz -gt $akt ] ; do
  # Aktuelle Zeile ermitteln, entspricht Dateinamen
  akt=`expr $akt + 1`
  dnam=`head -$akt /tmp/tar.cnt | tail -1`
  # Ins Archiv damit
  tar $taropt /tmp/bettina.tar $dnam >/dev/null 2>&1
  # Beim nächsten Aufruf Option Datei anhängen
  taropt="rvf"
  #
  # Spalte, so dass erg in der Mitte des
  # Fensters im Titel erscheint
  #
```

```
  tput cup $y `expr $x + 26`
  erg=`expr  $akt \* 100 / $anz`
  echo "$erg%"
  #
  # Anzeige der Dateinamen in der
  # linken Ecke des Fensterinneren
  #
  tput cup `expr $y + 1` `expr $x + 2`
  printf "%-40.40s" $dnam
  #
  # Ausgabe des Fortgangs in der mittleren Zeile
  # des Statusfensters
  #
  pos=`expr $erg / 2 + 14`
  tput cup `expr $y + 2` $pos
  echo -n ":"
  sleep 1
done
rm /tmp/tar.cnt
tput cup 24 0

exit 0
```

7. Achtung! Das Lösungsskript ist nicht vollständig abgedruckt:


```
# cwc.sh (Lösung 26)
# Achtung nicht vollständig!!
#
...
until [ "$ein" = "q" ] ; do
   # Ausgabe der Dateien
   i=0
   akt=0
   while [ $i -lt 19 -a $akt -lt $anz ] ; do
     akt=`expr $i + $offset`
     tput cup `expr $i + 2` 2
     zeile=`head -$akt $tmpfile | tail -1`
     #
     # (Aufgabe 7)
     # Verzeichnis blau darstellen
     # TarArchiv rot
     # Zip Archiv lila
     #
     typ=`file $zeile`
      if [ -d $zeile ] ; then
        tput setaf 4
     elif [ "$typ" = "$zeile: GNU tar archive" ] ; then
        tput setaf 1
```

```
    else
       case "$typ" in
           "$zeile: gzip compressed data,"*)
                 tput setaf 5 ;;
       esac
    fi
    printf "%-38.38s" "$zeile"
    tput sgr0
    i=`expr $i + 1`
  done
...
done
exit 0
```

Da im Testbefehl beim Vergleich von Zeichenketten keine Wildcards verwendet werden können, wurde die Abfrage für `gzip`-komprimierte Dateien über einen `case`-Befehl formuliert.

8. Das folgende Skript erfüllt die Aufgabenstellung:


```
# Skript 27 (Lösung)
#   Demonstriert "select" in dem es alle
#   Unterverzeichnis in $1 sucht und zur
#   Auswahl stellt. Wird keine gültige
#   Auswahl getroffen, wird die Benutzereingabe
#   (in REPLY) als Befehl ausgeführt
#
if [ ! -d "$1" ] ; then
    echo "Aufruf: $0 <Verzeichnis>" >&2
    exit 1
fi
select wort in `find "$1" -type d -print` ; do
   if [ -n "$wort" ] ; then
      echo "Verzeichnis $wort"
      ls "$wort"
   else
      echo "Versuche Befehl $REPLY auszuführen..."
      $REPLY
   fi
done
exit 0
```

Kapitel 5

Parameter zum Zweiten

»Ich habe nichts zu bieten, außer Blut, Schweiß und Tränen« – W. Churchill

... unfair von mir, das erst jetzt zu erwähnen, aber dieser Spruch in Kapitel 1 hätte doch leicht demotivierend gewirkt. Ehrlich gesagt, Sie werden noch einige Zeit opfern müssen, bevor Sie als Experte durchgehen können. Als kleiner Trost sei gesagt, dass auch in den restlichen Kapiteln dieses Buchs kein Blut vergossen wird `:-)`

Auch wenn wir Shellfunktionen erst in Kapitel 7 besprechen werden, hier der Hinweis, dass sämtliche in diesem Kapitel besprochenen Ersetzungen genauso mit den Parametern der Shellfunktionen funktionieren und nicht auf die Skriptparameter beschränkt sind.

5.1 Der Stand der Dinge

In diesem Kapitel wollen wir uns mit den letzten Mysterien der Parameter für Skripten befassen. Sie erinnern sich sicherlich noch an das erste Kapitel, welches sich mit den Parametern `$0` bis `$9` befasste und den Parameter `$#` erklärte, mit dem sich feststellen lässt, wie viele Parameter dem Skript insgesamt übergeben wurden.

Dabei ist der Parameter `$0` der Name des Skripts. Die `for`-Schleife aus Kapitel 3 zeigte dann einen Weg auf, wie die Parameter ab Position 10 abgefragt werden können. Elegant war das allerdings nicht.

5.2 Parameter jenseits $9

Die Methode, mittels for-Schleife auf die Parameter zuzugreifen, funktioniert zwar, ist aber eher umständlich. In diesem Abschnitt möchten wir daher auf zwei Alternativen zu sprechen kommen.

In Kapitel 1 tauchte das Problem auf, wie Variablen zu referenzieren sind, wenn die Shell Variablennamen und folgenden Text nicht in zwei Wörter aufteilen kann. Skript 4 machte das Problem deutlich. Die Lösung war, den Variablennamen in geschweifte Klammern zu setzen: `${anz}mal` statt `$anzmal`.

Ein ähnliches Problem trat auch bei Parameter `$10` auf, den die Shell als `${1}0` interpretierte und somit eine Null hinter den ersten Parameter hing. Die Lösung für dieses Problem sind ebenfalls geschweifte Klammern: `${10}` greift somit auf den zehnten Parameter zu.

Diese Möglichkeit bieten nur die Kornshell und die Bash. Die `sh` quittiert diesen Versuch mit `Syntax error: Bad substitution`. Bitte schauen Sie auch noch in Anhang A nach, welche Unterschiede es zwischen den drei Shells gibt.

Es gibt jedoch noch eine dritte Methode, um an Parameter jenseits der `$9` heranzukommen: den Befehl `shift n`. Dabei muss `n` ein positiver Wert kleiner gleich `$#` sein. Ist `n` nicht angegeben, so wird `n` gleich `1` angenommen. Ein `shift` verschiebt die Positionsparameter des Skripts (oder die Funktionsparameter, wenn `shift` innerhalb einer Funktion ausgeführt wird) um eine Position nach links. Parameter `$0` bleibt unverändert, und der alte Parameter `$1` fällt unwiderruflich weg. Ist `n` größer als `1`, so wird nicht um eine, sondern um `n` Positionen nach links verschoben.

Nehmen wir an, wir rufen ein Skript wie folgt auf:

```
buch@koala:/home/buch > ./skript.sh A B C D E F G H I J K L M N
```

So haben wir folgendes Bild bei den Parametern:

```
Position:     1  2  3  4  5  6  7  8  9 10 11 12 13 14
Wert:         A  B  C  D  E  F  G  H  I  J  K  L  M  N
Parameter:  $1 $2 $3 $4 $5 $6 $7 $8 $9 ...
$#          14
$*          "A B C D E F G H I J K L M N"
```

Führt das Skript ein `shift 2` aus, so verändern sich die Parameter wie folgt:

```
Position:     1  2  3  4  5  6  7  8  9 10 11 12 13 14
Wert:         A  B  C  D  E  F  G  H  I  J  K  L  M  N
Parameter:  $1 $2 $3 $4 $5 $6 $7 $8 $9 ...
$#          12
$*          "C D E F G H I J K L M N"
```

Wenn Sie ein n angeben, welches größer als `$#` ist, so bleiben die Parameter unverändert. Nachdem `shift` ausgeführt wurde, werden alle anderen Parameter (`$#`, `$*` und `$@`) angepasst.

Fragen Sie zunächst alle Parameter ab, die durch `shift` links aus dem Fenster hinausrutschen. Das Fenster lässt sich nicht nach links schieben, sondern nur nach rechts. Sind die Parameter erst einmal aus dem Fenster herausgerutscht, so sind die Daten für Ihr Skript verloren.

5.3 Spezielle Parameter

Das bisherige Wissen erlaubt natürlich schon eine sehr angenehme Bearbeitung aller übergebenen Parameter. In diesem Abschnitt wollen wir uns allerdings noch mit einigen zusätzlichen Parametern mit spezieller Bedeutung beschäftigen. Einige dieser Parameter haben Sie bereits kennen gelernt und mehrfach benutzt:

`$0`	Name des Skripts, das gerade ausgeführt wird. Wird von `shift` nicht beeinflusst!
`$#`	Die Anzahl der übergebenen Parameter. Wird nach einem `shift` angepasst!
`$?`	Der Exitstatus des letzten Befehls.

Tabelle 5.1: Wichtige eingebaute Variablen

Das sind aber nicht alle. Die Wichtigsten finden Sie in Tabelle 5.2 aufgeführt.

`$$`	Ist identisch mit der Prozess-ID der Shell, die das Skript ausführt. Mit Prozessen werden wir uns ausführlich in Kapitel 7 beschäftigen. Jetzt soll die Aussage reichen, dass vom Systemstart an jedes ausgeführte Programm (Shell, Daemon, Skript etc.) eine Prozess-ID bekommt, die systemweit eindeutig ist. Dadurch ist das Betriebssystem fähig, die Prozesse zu unterscheiden und zu verwalten. Diese ID startet beim Systemstart bei 1 und wird mit jedem weiteren ausgeführten Programm um eins erhöht.

Tabelle 5.2: Weitere eingebaute Variablen

`$*`	Wird ersetzt durch eine Liste aller dem Skript übergebenen Parameter. Wird `$*` genutzt, so bricht die Shell den Parameter *nicht* in einzelne durch Leerzeichen getrennte Worte auf, sondern interpretiert ihn als ein einzelnes Wort.
`$@`	Ähnlich wie `$*`, nur bricht die Shell hier den Parameter in einzelne Worte auf. Das heißt, `$@` ist gleichbedeutend mit `$1 $2 $3`.

Der Unterschied zwischen `$*` und `$@` erschließt sich nicht sofort. Ein einfaches Skript allerdings macht den Sachverhalt schnell klar:


```
buch@koala:/home/buch > cat skript28.sh
# Skript 28: Parameter
#
# Simple Demo: Unterschied zwischen $@ und $*
# Macht den Unterschied anhand der übergebenen
# Parameter klar.
#
if [ -z "$*" ] ; then
   echo "Keine Paramter" >&2
   exit 1
fi
echo "Anzahl Parameter: $#"
for wort in "$@" ; do
   echo "-->"$wort"<--"
done
for wort in "$*" ; do
   echo "<--"$wort"-->"
done
exit 0

buch@koala:/home/buch > skript28.sh Darwin Adelaide Alice Queenstown
Anzahl Parameter: 4
-->Darwin<--
-->Adelaide<--
-->Alice<--
-->Queenstown<--
<--Darwin Adelaide Alice Queenstown-->
buch@koala:/home/buch >
```

Stellt sich natürlich die Frage, was sich mit diesen Parametern anfangen lässt. Kommen wir zu diesem Zweck noch einmal auf das Skript 26 zurück, unseren CW-Commander. Besonders beeindruckend ist dieses Skript noch nicht (und wird es in Ihren Augen vielleicht selbst am Ende des Buches nicht sein `;o)`), aber ein krasses Problem wollen wir an dieser Stelle schon einmal ausmerzen.

Vielleicht haben Sie den `cwc` (oder jedes andere Skript) bereits zweimal auf verschiedenen Terminals aufgerufen. Wenn Sie zwei verschiedene Anmel-

dungen (Logins) zum Testen genutzt haben, bekommen Sie eine Fehlermeldung:

```
bash: /tmp/cwc.tmp: Permission denied
```

Falls Sie die gleiche Anmeldung genutzt haben, tritt dieser Fehler zwar nicht auf, allerdings ist die Ausgabe scheinbar völlig unsinnig.

Die Ursache ist sehr schnell ausgemacht, greift unser Skript in beiden Fällen doch auf dieselbe Datei namens `/tmp/cwc.tmp` zu. Im Falle von zwei gleichen Anmeldungen überschreiben sich beide Skripten abwechselnd in Ihre Arbeitsdatei. Während bei unterschiedlichen Anmeldungen der erste die Datei anlegen kann, kann der zweite die Datei nicht mehr anlegen und bekommt stattdessen den obigen Fehler.

Ob sich die Temporärdatei von einer anderen Anmeldung aus überschreiben lässt, hängt von den vergebenen Zugriffsrechten ab. In diesem Zusammenhang spielt die `umask` eine wichtige Rolle. Genauere Informationen finden sich dazu in Kapitel 11.

Wundern Sie sich nicht, wenn Sie `rw-rw-rw-`-Rechte vergeben und Ihre Datei von einer anderen Anmeldung überschrieben wird.

Unsere Rettung naht in Gestalt des `$$`-Parameters. Da dieser die Prozess-ID des aktuellen Skripts enthält und diese systemweit nur einmal vergeben werden kann, hat `$$` bei jedem Aufruf einen unterschiedlichen Wert. Fließt dieser Wert nun in den Dateinamen der Temporärdatei, so wird auch der Dateiname systemweit eindeutig! Problem gelöst.

Nun, das ist nicht völlig korrekt. Die Prozess-ID ist nicht auf immer und ewig eindeutig. Eine Prozess-ID ist eine Ganzzahl. Diese kann auf einem Computer nur einen bestimmten Zahlenbereich darstellen. Irgendwann ist es soweit, dass die größte Zahl erreicht ist (Unixsysteme laufen durchaus solange!), dann beginnt die Prozess-ID wieder von vorne. Wie die ID genau vergeben wird, soll nicht Gegenstand dieses Buches sein. Wichtig ist nur, dass eine Prozess-ID nur für die Laufzeit des Prozesses garantiert eindeutig ist.

Wenn Sie ganz sichergehen wollen, nehmen Sie in den Dateinamen auch weitere Informationen wie Datum oder Zeit mit auf.

Verzichten Sie aber auf `$0`, da darin durchaus eine Pfadangabe enthalten sein kann, womit die Erstellung der Temporärdatei fehlschlägt oder plötzlich in den falschen Verzeichnissen Temporärdateien erstellt werden und dort evtl. sogar wichtige Dateien überschreiben.

Sicherlich stimmen Sie mit uns überein, dass diese Änderung zu gering ist, um dafür erneut das cwc-Skript aufzulisten, stellen wir uns deshalb noch eine weitere Aufgabe:

Wenn dem Skript Parameter übergeben werden, so prüft das Skript, ob diese Parameter Dateinamen im aktuellen Verzeichnis entsprechen, und summiert sie in einer Liste auf, die durch Doppelpunkte getrennt wird. Bei der Anzeige der Verzeichnisse sollen alle Dateinamen invers ausgegeben werden, die in dieser Liste auftauchen.

Sieht beim ersten Mal schlimmer aus, als es ist. Die Ausgaben wurden ausführlich in Kapitel 4, die benötigten Parameter in diesem Kapitel behandelt. Wie Sie feststellen können, ob eine Datei existiert, wurde in Kapitel 3 erläutert, und so sollte nur ein Problem übrig bleiben: Wie lässt sich feststellen, ob ein Dateiname in der Markierungsliste enthalten ist?

Es wundert Sie sicherlich nicht mehr im Geringsten, wenn wir in dieser Situation mal wieder auf einen Befehl hinweisen, der Ihnen noch unbekannt ist: `grep`.

`grep` erwartet mindestens einen Parameter: die gesuchte Zeichenkette. Der zweite optionale Parameter ist der Name einer Datei, in der `grep` suchen soll. Wird keine Datei angegeben, so sucht `grep` in den Daten, die von der Standardeingabe kommen. Mittels des Parameters `-i` sucht `grep` unabhängig von der Groß- bzw. Kleinschreibung.

Findet `grep` die Zeichenkette, wird der Dateiname ausgegeben, gefolgt von der kompletten Zeile, in der die Zeichenkette gefunden wurde. Konnte die Zeichenkette nicht gefunden werden, so wird nichts ausgegeben. Falls die Standardausgabe durchsucht wurde, wird kein Dateiname ausgegeben. Der Rückgabewert von `grep` ist `0`, wenn die Zeichenkette gefunden wurde, ansonsten größer als `0`.

Ein paar Beispiele:

```
buch@koala:/home/buch > echo "Australien" | grep "aus"
buch@koala:/home/buch > echo $?
1
buch@koala:/home/buch > echo "Australien" | grep -i "aus"
Australien
buch@koala:/home/buch > echo $?
0
buch@koala:/home/buch >
```

Erstellen wir also zunächst die Liste in einer Variablen namens "`Mark`":

```
Mark=":"
for wort in "$@" ; do
  if [ -f $wort ] ; then
     Mark="$Mark$wort:"
  fi
done
```

Schauen wir uns nun noch den Code an, der feststellt, ob die aktuelle Zeile invers ausgegeben werden soll:

```
...
  while [ $i -lt 18 -a $akt -le $anz ] ; do
   akt=`expr $i + $offset`
   tput cup `expr $i + 2` 2
   zeile=`sed -n -e "${akt}p`
   # Falls eingetragen, dann invers ausgeben
   revers=`echo "$Mark" | grep ":$zeile:" `
   if [ -n "$revers" ] ; then
      tput rev
   fi
   printf "%-38.38s" $zeile
   tput sgr0
   i=`expr $i + 1`
  done
  ...
```

Die Doppelpunkte beim `grep` werden eingesetzt, damit zufällige Treffer aufgrund von Teilmengen vermieden werden. So ergibt die Suche nach `"Euro"`

```
echo ":Eurora:Europa:" | grep "Euro"
```

im obigen Beispiel ein Ergebnis, während

```
echo ":Eurora:Europa:" | grep ":Euro:"
```

sicherstellt, dass der Suchbegriff als eigenständiges Wort existiert und nicht als Untermenge eines längeren Eintrags. Allerdings ist auch diese Methode nicht wasserdicht. Enthält der Dateiname ein »`:`«, so misslingt die Suche möglicherweise auch.

5.4 Parameter trickreich genutzt

Das ist alles schon recht erfreulich, aber je mehr Parameter Ihr Skript hat, die zusätzlich auch noch optional angegeben werden können, desto mehr sieht Ihr Skript am Anfang nach einer gierigen `if then elif else`-Schlange aus, welche sich an der Übersichtlichkeit Ihres Skripts labt.

Das ist auch den Entwicklern der Bash aufgefallen, und deshalb gibt es auch hier Alternativen, von denen ich die wichtigsten im Folgenden aufführen möchte. Sie können diese an jeder Stelle einsetzen, wo Sie Parameter oder Variablen nutzen würden. Tatsächlich sind diese Parametererersetzungen nicht nur auf Parameter beschränkt. Es ist genauso möglich (und sinnvoll), diese auf Variablen einzusetzen.

Im Folgenden werden wir nur die wichtigsten dieser Ersetzungen besprechen und anwenden. Allerdings werden wir dieses Thema nochmals in Kapitel 12 aufnehmen, wenn die portable Skriptprogrammierung auf der Tagesordnung steht.

Noch ein Hinweis, bevor wir loslegen: Die Ersetzung wird hier zwar für die Parameter beschrieben, funktioniert aber genauso für Variablen. Sie können also in diesem Kapitel überall, wo <parameter> steht, auch einen Variablennamen eintragen. Der Funktion tut dies keinen Abbruch.

5.4.1 Vorgabewerte nutzen (Use Default Value)


```
${<parameter>:-<wert>}
```

Ist <parameter> leer oder noch nicht gesetzt worden, so ersetzt die Bash dieses Konstrukt durch <wert>, ansonsten durch den Inhalt von <parameter>. Dabei bedeutet leer, dass der <parameter> nicht existiert oder den Leerstring "" enthält.

Nehmen wir noch einmal das letzte Skript. Nehmen wir an, dass der erste Parameter ein Verzeichnis ist und ab dem zweiten Parameter Dateinamen angegeben werden, die im Arbeitsverzeichnis markiert werden sollen. Sie könnten natürlich die Anzahl der Parameter feststellen und den ersten Parameter der Variablen zuordnen, in der Sie das aktuelle Arbeitsverzeichnis speichern wollen. Das erfordert allerdings eine if-Abfrage, die wir ja unterbinden möchten.

Das Arbeitsverzeichnis speichern wir jetzt aus zwei Gründen ab:

- Eine Verwaltung des Arbeitsverzeichnisses erspart uns den wiederholten Aufruf von pwd. Und wer weiß, was wir in den künftigen Versionen dieses Skripts noch alles machen?
- Die Parameterersetzung lässt sich daran gut erklären.

Also sieht das Skript jetzt wie folgt aus:


```
# Skript 29: cwc.sh
#
# Komplettes Skript mit Markierung und Parameterersetzung
#
#
Mark=":"
#
# Parameter $1 ist das Startverzeichnis oder Default "."
#
verz=${1:-`pwd`}
if [ -d $verz ] ; then
   cd $verz
fi
#
# Markierung erst ab Parameter 2 nötig, also schieben
```

```
#
shift 1
for wort in "$@" ; do
  if [ -f $wort ] ; then
     Mark="$Mark$wort:"
  fi
done
tput clear
i=2
tput cup 1 0
echo "+--------------------------------------+"
while [ $i -lt 20 ] ; do
  tput cup $i 0
  printf "! %38s!" " "
  i=`expr $i + 1`
done
tput cup 20 0
echo "+--------------------------------------+"
echo "z = PgUpw = PgDn  q = Quit"
tput cup 1 2
printf "%.35s" "$verz"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
offset=1
tmpfile="/tmp/cwc$$.tmp"
anz=`ls "$verz"|tee $tmpfile | wc -l | cut -c-7`
akt=1
until [ "$ein" = "q" ] ; do
  # Ausgabe der Dateien
  i=0
  akt=0
  while [ $i -lt 18 -a $akt -le $anz ] ; do
   akt=`expr $i + $offset`
   tput cup `expr $i + 2` 2
   zeile=`sed -n -e "${akt}p" <$tmpfile`
   revers=`echo "$Mark" | grep ":$zeile:" `
   if [ -n "$revers" ] ; then
      tput rev
   fi
   printf "%-38.38s" $zeile
   tput sgr0
   i=`expr $i + 1`
  done
  while [ $i -lt 18 ] ; do
    tput cup `expr $i + 2` 2
    i=`expr $i + 1`
    printf "%38.38s" " "
  done
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
```

```
    case $ein in
     "q" | "Q") ein="q";;
     "w" | "W")
         # Erstmals addieren
         offset=`expr $offset + 17` ;
         if [ $offset -gt $anz ] ; then
            # Offset größer als Anzahl Zeilen
            offset=`expr $offset - 17`
         fi ;;
     "z" | "Z")
         if [ $offset -gt 1 ] ; then
            offset=`expr $offset - 17` ;
         fi ;;
      *) ein="" ;;
    esac
  done
done
exit 0
```

Die Schreibweise `verz=${1:- `pwd`}` ist korrekt, eine erneute Angabe des $-Zeichens für die Referenz von Parameter $1 ist verboten, es sei denn, Sie möchten einen Fehler `bad substitution` bekommen.

5.4.2 Vorgabewerte setzen (Assign Default Value)


```
${<parameter>:=<wert>}
```

Wenn wir uns jetzt Skript 29 nochmals anschauen, so stellen wir fest, dass die erste `while`-Schleife noch die Zuweisung von `i=2` benötigt. Effizient wie Sie als angehender Skriptexperte nun einmal sind (nein, das Thema Tippfaulheit legen wir endgültig zu den Akten `:)`), wollen Sie diese Zuweisung möglichst einsparen.

Mit der Ersetzung aus Abschnitt 5.4.1 haben Sie leider nichts gewonnen, schließlich wird `<parameter>` kein `<wert>` zugewiesen, sondern nur bei Bedarf ein Defaultwert generiert. Dieser Defaultwert ersetzt dann den Ausdruck `${<parameter>:-<wert>}`.

Mit der Ersetzung `${<parameter>:=<wert>}` erzwingen Sie jedoch, dass `<parameter>` der `<wert>` zugewiesen wird, falls `<parameter>` leer bzw. unbenutzt ist. Danach wird dieses Konstrukt durch den Inhalt von `<parameter>` ersetzt. Sie können also die Zeile `i=2` wegfallen lassen und die `while`-Schleife wie folgt optimieren:

```
...
while [ ${i:=2} -lt 20 ] ; do
...
```

Und schon haben Sie eine weitere Zeile eingespart ...

Bei der zweiten `while`-Schleife innerhalb der `until`-Schleife liegt es nahe, diese Technik nochmals zu nutzen:


```
...
 until [ "$ein" = "q" ] ; do
   # Ausgabe der Dateien
   akt=0
   # anstelle von i=0
   # hier ${i:=0}
   while [ ${i:=0} -lt 18 -a $akt -le $anz ] ; do
...
```

Leider ist das keine gute Idee. Da die Variable `i` bereits benutzt wurde, hat sie einen definierten Wert, und deshalb wird kein Default mehr vergeben. Klappen würde es erst wieder, wenn vor der Whileschleife `i=''` gesetzt worden wäre. Und wenn die Variable schon initialisiert werden muss, kann sie gleich mit dem korrekten Wert beschickt werden.

5.4.3 Fehlermeldung ausgeben, falls Variablen leer sind


```
${<parameter>:?<text>}
```

Variante Nummer drei beglückt Sie mit der Möglichkeit, eine Fehlermeldung `<text>` auf die Standardfehlerausgabe auszugeben, wenn `<parameter>` leer oder noch unbenutzt ist. Falls die Shell nicht interaktiv ist (d.h. ein Skript läuft), wird das Skript beendet.

5.4.4 Alternative Werte setzen


```
${<parameter>:+<wert>}
```

Ist `<parameter>` leer bzw. unbenutzt, so wird nichts ersetzt, ansonsten wird der Ausdruck durch den `<wert>` ersetzt. Diese Ersetzung ist also das Gegenteil von `${<parameter>:-<wert>}`.

Sie dürfen in den beschriebenen Parameterersetzungen auch den Doppelpunkt : bei der Zuweisung weglassen. In diesem Fall wird ein Defaultwert aber nur zugewiesen, wenn der Parameter nicht gesetzt ist. Bei eine Zuweisung mit : hingegen wird der Wert zugewiesen, wenn der Parameter ungesetzt oder gleich "" ist.

5.5 Bash und Kornshellvarianten

Die folgenden Ersetzungen funktionieren leider nicht mehr in der Bourneshell, sondern stehen nur in der Bash und der Kornshell zur Verfügung. Wenn Ihr Skript diese Funktionen verwendet, so läuft es nicht mehr unter der /bin/sh (es sei denn, dies ist ein Link auf die Bash etc.). Im Anhang A finden Sie weitere Informationen nicht nur zu den Ersetzungen, sondern auch zu sonstigen Unterschieden zwischen den drei Shells.

5.5.1 Variablenlänge ermitteln


```
${#<parameter>}
```

Dieser Ausdruck wird ersetzt durch die Länge des Variableninhalts in Zeichen. Falls <parameter> »*« oder »@«, so wird der Ausdruck durch die Anzahl an Parametern ersetzt, die dem Skript übergeben wurden (auch *Positionsparameter* genannt).

```
buch@koala:/home/buch > cw="Christa"
buch@koala:/home/buch > echo ${#cw}
7
buch@koala:/home/buch >
```

Wird anstelle eines Parameters oder einer Variablen ein »*« oder »@« angegeben, so gibt die Ersetzung die Anzahl der übergebenen Parameter zurück.

Falls Sie die Länge einer Zeichenkette in der /bin/sh benötigen, so hilft Ihnen awk. awk ist eine Skriptsprache, die die ihr übergebenen Daten in Seiten aufteilt, die awk automatisch mit Kopf- und Fußzeilen versehen kann. Leider ist awk zu komplex, um es hier ausführlich zu erklären, deshalb nur kurz der Code, um die Länge einer Zeichenkette zu bestimmen:

```
buch@koala:/home/buch > echo "Christa" | awk '{ print length($1) }'
7
buch@koala:/home/buch >
```

Dabei benennt awk (der Name ergibt sich aus den Initialen der drei Autoren) seine Positionsparameter genauso wie die Shell: $1 $2 ... $10 $11 usw.

5.5.2 Suffix entfernen


```
${<parameter>%<wert>}
${<parameter>%%<wert>}
```

Für den Inhalt von <parameter> wird geprüft, ob er mit der Zeichenkette <wert> endet. <wert> kann dabei die Ersatzmuster aus Kapitel 2.3 verwenden, die sich hier jedoch auf <parameter> und nicht auf Dateinamen beziehen. Ist dies der Fall, so wird die kürzeste (Angabe von %) bzw. längste Zeichenkette (Angabe %%) entfernt, die diesem Muster entspricht. Das Ergebnis ersetzt dann den angegebenen Ausdruck.

Ist in Bash (Version 2.0 oder höher) <parameter> »*« oder »@«, so wird die oben beschriebene Ersetzung für jeden einzelnen der Positionsparameter durchgeführt. Der Ausdruck wird dann ersetzt durch die Liste aller so ersetzten Positionsparameter.

Nehmen wir einmal folgendes Skript:


```
# paramsp.sh: Demonstriert die Parameterersetzung
#  mit % und %% anhand von $1 und $@
if [ $# -eq 0 ] ; then
   echo "Mindestens einen Parameter angeben." 1>&2
   echo "Ersetzt das Muster 'a*b'" 1>&2
   exit 1
fi
echo '${1%a*b}'  ${1%a*b}
echo '${1%%a*b}' ${1%%a*b}
if [ $# -gt 1 ] ; then
   # Dieser Teil läuft nur ab Bash 2.0
   echo '${@%%a*b}' ${@%a*b}
   echo '${@%%a*b}' ${@%%a*b}
fi
exit 0
```

So erhalten wir folgende Ausgabe:

```
buch@koala:/home/buch/skript > paramsp.sh Aberabcdab "Saebel gab"
${1%a*b} Aberabcd
${1%%a*b} Aber
${*%a*b} Aberabcd Saebel g
${*%%a*b} Aber S
buch@koala:/home/buch/skript >
```

5.5.3 Präfix entfernen


```
${<parameter>#<wert>}
${<parameter>##<wert>}
```

Diese Ersetzung funktioniert genau wie unter 5.5.2 beschrieben mit einem wichtigen Unterschied: Die Muster werden am Anfang des in `<parameter>` enthaltenen Wertes ersetzt. Auch hier gilt, dass die Ersetzung mit »*« oder »@« nur in der Bash 2.0 oder neuer funktioniert.


```
# paramsh.sh: Demonstriert die Ersetzung
#  per # und ##
#  Nimmt den ersten Parameter und führt die
#  verschiedenen Ergebnisse der Parameterersetzung vor
#  für $1 und $@
if [ $# -eq 0 ] ; then
   echo "Mindestens einen Parameter angeben." 1>&2
   echo "Ersetzt das Muster 'a*b'" 1>&2
   exit 1
fi
echo '${1#a*b}'  ${1#a*b}
echo '${1##a*b}' ${1##a*b}
if [ $# -gt 1 ] ; then
   # Dieser Teil läuft nur ab Bash 2.0
   echo '${@#a*b}'  ${@#a*b}
   echo '${@##a*b}' ${@##a*b}
fi
exit 0
```

Rufen wir `paramsh.sh` jetzt einmal auf:

```
buch@koala:/home/buch/skript > ./paramsh.sh aber aberabc
${1#a*b} er
${1##a*b} er
${@#a*b} er erabc
${@##a*b} er c
buch@koala:/home/buch/skript >
```

Bitte überlegen Sie sich den Einsatz dieser Ersetzung gut, denn sonst könnten Sie über eine solche harmlose Anweisung ins Schleudern geraten, wenn sich die Ausgaben möglicherweise nicht mit Ihren Erwartungen decken:

```
buch@koala:/home/buch/skript > txt="Sydney hat ein Opernhaus"
buch@koala:/home/buch/skript > echo ${txt#S}
ydney hat ein Opernhaus
buch@koala:/home/buch/skript > echo ${txt##S}
ydney hat ein Opernhaus
```

```
buch@koala:/home/buch/skript > echo ${txt##S*}
buch@koala:/home/buch/skript > echo ${txt#S*}
ydney hat ein Opernhaus
buch@koala:/home/buch/skript >
```

Gerade der letzte Punkt verwundert vielleicht auf den ersten Blick etwas, aber # steht für den kürzesten Treffer in der Zeichenkette, und der ist für S* nun mal das »S«.

Falls Sie noch eine Gedächtnisstütze brauchen, welches Zeichen nun für die Präfixe und welches für die Suffixe steht, denken Sie an Prozentangaben, z.B. 35%. Dabei steht das % immer am Ende, und »%« entfernt Suffixe.

5.5.4 Bereiche eines Parameters


```
${<parameter>:<offset>}
${<parameter>:<offset>:<länge>}
```

Ab Bash Version 2 können Sie mittels dieser Ersetzung Teilbereiche aus dem <parameter> heraus kopieren. Dabei werden alle ab dem <offset>-Zeichen ermittelt. Ist <länge> nicht angegeben (1. Version), so werden alle Zeichen bis ans Ende der Zeichenkette kopiert. Ist <länge> angegeben, so werden nur <länge> Zeichen ab <offset> kopiert. Dabei muss <länge> größer oder gleich 0 sein. Ist <offset> kleiner als 0, so fängt der Offset von rechts (sprich Ende der Zeichenkette) an zu zählen.

Ein Beispiel:

```
buch@koala:/home/buch > a="Das ist ein Text"
buch@koala:/home/buch > echo ${a:4}"-"${a:2:3}
ist ein Text-s i
buch@koala:/home/buch >
```

Lassen Sie sich nicht davon irritieren, dass das erste Zeichen als Offset 0 und nicht als Offset 1 angesprochen wird. Dies liegt daran, wie der Computer Zeichenketten im Speicher ablegt:

Das erste Zeichen wird dabei z.B. an Adresse 1000 abgelegt. Da eine Adresse genau ein Zeichen abspeichern kann, liegt das zweite Zeichen an Adresse 1001, das dritte an Adresse 1002 usw. Das erste Zeichen hat damit einen Offset von 0 zur ersten Adresse der Zeichenkette, das zweite Zeichen einen Offset von 1 usw.

5.6 Parameter neu setzen

Okay, weiter oben haben wir noch behauptet, dass Positionsparameter (also Parameter, die dem Skript übergeben wurden) nicht mehr neu zu belegen sind. Nun, das ist nicht völlig korrekt. Ein bekannter Staatsmann hat sich einmal zu einer ähnlichen Problematik wie folgt geäußert:

»Was kümmert mich mein Geschwätz von gestern?«

Zugegeben, mit dieser Haltung gewinnt man als Autor kaum neue Sympathien bei seinen Lesern, doch zum einen ist die Aussage »Ein `shift` nach links bzw. zurück ist und bleibt unmöglich« absolut korrekt.

Auf der anderen Seite erlaubt die Bash eine komplette Neubelegung der Positionsparameter mithilfe des Befehls `set --`. Alle Worte, die auf diesen Befehl folgen, werden der Reihe nach den Parametern `$1 $2` usw. zugeordnet. Dabei dürfen auch mehr als neun Parameter zugeordnet werden. Auch die Parameter `$#`, `$*` und `$@` werden entsprechend den neuen Werten angepasst und spiegeln die Änderungen wider.

Zwar muss das »`--`« nicht angegeben werden, um die Parameter neu zu belegen, allerdings können Sie keine Parameter setzen, die mit einem »`-`« anfangen. In diesem Falle benötigen Sie unbedingt das »`--`«. Aus diesem Grunde gehen wir gleich auf Nummer Sicher und nutzen `set -- .`

`set a b c`	o.k.
`set -c b a`	Fehler, da Parameter mit »`-`« anfängt: `Unknown Option: c`

Tabelle 5.1: Neubelegung von Positionsparametern

Befehl `$*$#$1 $2 $3 $4 $5`
(Aufruf des Skripts)
`Skript.sh 1 2 3 A B"1 2 3 A.B"5 1 2 3 A B`
(Befehl im Skript)
`set -- Emu NSW"Emu NSW"2Emu NSW`

Kommen wir noch einmal auf Skript 12 aus Kapitel 4 zurück. Die letzte Version lief mit Here-Dokumenten und ermittelte die Anzahl an Dateien und Unterverzeichnissen in zwei dem Skript übergebenen Verzeichnissen. Lassen Sie uns diese Version ein wenig verbessern, indem wir die Summe der belegten Bytes in beiden Verzeichnissen getrennt ausgeben.

Kurze Überlegung:

- Wir verzichten auf jede Prüfung (Sind die Parameter Verzeichnisse? Sind genau zwei Parameter übergeben worden?) und sind optimistisch.
- Die `while`-Schleifen müssen so lange laufen, wie `read` etwas aus der Datei lesen kann (Exitstatus von `read` ist 0). Daher kann `read` direkt in der Schleife aufgerufen werden.


```
# Skript 30: Verzeichnisbelegung ausgeben
#
# Zwei Verzeichnisse werden übergeben. Keine
# Sicherheitsüberprüfungen, damit das Skript kurz bleibt
#
# 1. Parameter ermitteln und tmp-Dateinamen setzen
tmpfile="/tmp/erg$$"
rm -f $tmpfile
verz1=$1
verz2=$2
# 2. Dateien aus Verzeichnis 1 ermitteln
find $verz1 -type f -print 2>/dev/null >$tmpfile
# 3. Aus Ergebnisdatei lesen und addieren
exec 4<$tmpfile
erg1=0
while read zeile <&4 ; do
  set -- `ls -l "$zeile"`
  erg1=`expr $erg1 + $5`
done
# 4. Dateien aus dem 2. Verzeichnis anhängen
find $verz2 -type f -print >> $tmpfile 2>/dev/null
erg2=0
# 5. Die 2. Schleife mal mit awk
while read zeile <&4 ; do
  datgr=`ls -l "$zeile" | awk '{ print $5 }'`
  erg2=`expr $erg2 + $datgr`
done
4<&-
printf "%-20.20s   %10d Bytes\n" $verz1 $erg1
printf "%-20.20s   %10d Bytes\n" $verz2 $erg2
echo "Die Verzeichnisse $verz1 und $verz2 enthalten `wc -l <$tmpfile` Dateien"
anz=`find $verz1 $verz2 -type d -print 2>/dev/null | wc -l`
echo "Insgesamt existieren genau" $anz "(Unter-)Verzeichnisse"
rm $tmpfile
exit 0
```

Dieses Skript nutzt gleich zwei Techniken aus: Das Setzen der Positionsparameter war zu erwarten. In der zweiten Zeile läuft die gleiche Schleife ab, hier ermittelt das Skript die Größe zur Abwechslung mal mit `awk`. Ungewohnt dürfte aber das Lesen aus der tmp-Datei sein: Die erste While-Schleife bricht

ab, nachdem die letzte Zeile eingelesen wurde (`$zeile` ist `""` und damit Exitstatus von `read` ungleich 0).

Der zweite `find` hängt nun weitere Zeilen an die Datei an. Dadurch führt das nächste `read` dazu, dass die erste neu hinzugekommene Zeile eingelesen wird. Zwar ist das Setzen der Parameter eine gültige Lösung, allerdings können Sie das Skript mit dem weiter oben aufgeführten `awk` noch kürzen. Anstatt von

```
...
read zeile <&4
while [ -n "$zeile" ] ; do
  datei=`ls -l $zeile`
  set -- $datei
  erg1=`expr $erg1 + $5`
  read zeile <&4
done
...
```

können wir direkt `read` in die `while`-Schleife einbauen und anstatt von `set awk` nutzen:

```
...
while read zeile <&4 ; do
   datgr=`ls -l $zeile | awk '{ print $5 }' `
   erg1=`expr $erg1 + $datgr`
done
...
```

5.7 getopts für Positionsparameter


```
getopts <opts> <var>
```

Bisher haben Sie sich immer noch persönlich um die Überprüfung und Abfrage der Positionsparameter gekümmert. Das funktionierte sicherlich ziemlich gut, aber die Shell bietet Ihnen zusätzlich noch einen Befehl, um diese Parameter abzufragen: `getopts`. Mit diesem Befehl betreten Sie das letzte Neuland, zumindest was dieses Kapitel betrifft, denn Sie haben es geschafft: Der letzte Abschnitt von Kapitel 5 ist erreicht.

Die Schreibweise ist absolut richtig: `getopts`. Es gibt für C-Programme auch noch eine Routine `getopt`, und auf einigen Unixsystemen existiert auch noch ein Befehl `getopt`, der sich aber vollkommen anders als `getopts` verhält. Also nicht verwechseln!

Fast jeder Unixbefehl hat Optionen, die häufig mit einem Minuszeichen »-« eingeleitet werden. Als Beispiel sollen hier nur `ls` und `wc` erwähnt werden. Solche Optionen sind für ein Skript nichts anderes als Positionsparameter, die allerdings bestimmten Regeln gehorchen müssen, damit sie korrekt erkannt werden können:

1. Optionen können, müssen aber nicht angegeben werden.
2. Werden Optionen angegeben, so stehen sie vor den Dateinamen bzw. Parametern, die der Befehl unbedingt benötigt.

```
ls -l kapitel*
ls -l
tail -1 kapitel1.txt
```

aber nicht

```
ls kapitel* -l     (Interpretiert -l als Dateinamen)
tail kapitel1.txt -1
```

3. Die Reihenfolge der Optionen ist gleichgültig.
4. Optionen können zusammengefasst werden, falls Sie das möchten.

Schauen wir uns diese Regeln anhand von `wc` an:

```
wc skript1.sh       --> Nach Regel 1 korrekt
wc -c skript1.sh    --> Ok
wc skript1.sh -c    --> Fehler nach Regel 2
```

Die folgenden vier Beispiele sind funktional identisch:

```
wc -c -w skript1.sh    --> Ok
wc -w -c skript1.sh    --> Ok
wc -cw skript1.sh     --> Ok
wc -wc skript1.sh     --> Ok
```

Vor allem die letzten vier Zeilen machen deutlich, welche Probleme bei der Abfrage der Parameter lauern können. Sie können sich gern an einer entsprechenden Routine versuchen, aber der freundliche Autor von nebenan empfiehlt in solchen Fällen den Einsatz von `getopts <opts> <var>`.

Bei jedem Aufruf von `getopts` prüft dieses, ob es eines der in `<opts>` angegebenen Zeichen (eingeleitet von einem »-«, s.o.) in den Positionsparametern finden kann. Dabei durchläuft `getopts` die Positionsparameter von links nach rechts. Welcher Parameter dabei als Nächstes bearbeitet wird, legt die von der Shell automatisch verwaltete Variable `OPTIND` fest. Diese wird beim Aufruf des Skripts auf 1 gesetzt, und jeder weitere Aufruf von `getopts` erhöht diese automatisch. Findet `getopts` eine Option, so wird das gefundene Zeichen der Variablen `<var>` zugewiesen.

Falls eine der in <opts> angeführten Optionen ein zusätzliches Argument benötigt, so sollte dem entsprechenden Zeichen in <opts> ein Doppelpunkt folgen. Um dies ein wenig zu verdeutlichen, hier ein paar Beispiele:

```
getopts ABC:D var
```

würde u.a. folgende Optionen als Parameter akzeptieren:

```
-A
-AB
-A -B
-C argument -A
-C      ergibt einen Fehler, da -C ein zusätzliches Argument verlangt (1)
-Q      ebenfalls ein Fehler, da illegale Option (2)
```

Im Falle von `-C` im oberen Beispiel wird `argument` ebenfalls automatisch erkannt. Da in `$var` jedoch schon die erkannte Option (also `C`) abgespeichert wurde, muss das Argument an einer anderen Stelle abgelegt werden. Dazu dient die von der Shell verwaltete Variable `OPTARG`. Diese enthält also in diesem Beispiel `argument`.

An dieser Stelle können wir es uns nicht verkneifen, auf Murphys Gesetz hinzuweisen: »Alles was schief gehen kann, geht schief.« Was hat dies nun mit `getopts` zu tun? Ganz einfach: Wir müssen uns an dieser Stelle mit der Fehlerbehandlung beschäftigen, denn es ist durchaus nicht unwahrscheinlich, dass der Benutzer bei der Angabe der Optionen einen Fehler begeht.

`getopts` hat zwei Fehlermodi: *normal* und *leise*. Im Modus *normal*, der standardmäßig eingestellt ist, gibt `getopts` Fehlermeldungen aus, die im Beispiel von oben so aussähen:

```
buch@koala:/home/buch > ./skript.sh -C
./skript.sh: option requires an argument -- C
buch@koala:/home/buch > ./skript.sh -Q
./skript.sh: illegal option -- Q
```

Da solche Fehlermeldungen nicht immer erwünscht sind, bietet sich der *Leise-Modus* an. Er wird aktiviert, wenn das erste Zeichen von <opts> ein Doppelpunkt »`:`« ist. In diesem Fall wird die Ausgabe der Fehlermeldungen unterdrückt. Die Unterdrückung der Fehlermeldungen lässt sich ebenfalls erreichen, wenn die Variable `OPTERR` auf 0 gesetzt wird.

`getopts` setzt die Variablen `OPTARG` und <var> auf bestimmte Werte, die abhängig vom Modus sind.

Fall 1: Illegale Option

	Modus Normal	Modus Leise
OPTARG	""	gefundenes Zeichen, welches als illegal erkannt wurde
<var>	wird auf ? gesetzt	wird auf ? gesetzt
Fehlermeldung	wird ausgegeben	wird unterdrückt

Fall 2: Fehlendes Argument

	Modus Normal	Modus Leise
OPTARG	""	:
<var>	wird auf ? gesetzt	Option, deren Argument fehlt
Fehlermeldung	wird ausgegeben	wird unterdrückt

Der Rückgabewert von `getopts` ist 1, wenn alle Parameter abgearbeitet wurden oder ein Fehler aufgetreten ist und somit ein weiterer Aufruf von `getopts` keinen Sinn mehr macht. Ansonsten ist der Rückgabewert gleich 0.

Noch ein Hinweis zum Schluss: Wenn `getopts` auf den ersten Parameter trifft, der weder Option noch Argument einer Option sein kann, so gibt es 1 zurück. Wenn Sie dann auf die restlichen Parameter zugreifen wollen, so ist ein `shift` `` `expr $OPTIND - 1` `` sehr hilfreich. Danach steht in $1 der erste Parameter, der nicht mehr von `getopts` bearbeitet wurde. Warum `OPTIND - 1`? `OPTIND` steht auf dem ersten Parameter, der nicht mehr von `getopts` bearbeitet wurde. Verschieben wir aber um `OPTIND` Parameter, so geht auch der erste benötigte Parameter verloren.

5.8 getopts für eigene Parameter

```
getopts <opts> <var> <arg1> ...
```

Normalerweise arbeitet `getopts` die Parameter ab, die dem Skript übergeben wurden. Falls Sie jedoch eigene Parameter überprüfen wollen, ohne dass `set --` zum Einsatz kommt, so können Sie nach <var> beliebig viele Parameter angeben, die dann von `getopts` abgearbeitet werden.

Bis auf die Tatsache, dass die Argumente <arg1> usw. anstelle der Positionsparameter abgearbeitet werden, verhält sich `getopts` in beiden Fällen absolut identisch.

Falls Sie mittels `set --` die Positionsparameter neu belegen, wird die Shellvariable `OPTIND` nicht automatisch auf `1` zurückgesetzt. Dies müssen Sie selbst sicherstellen.

Bevor wir zu den Aufgaben kommen, möchten wir Ihnen abschließend noch einen nützlichen Befehl zur Manipulation von Textdateien vorstellen:


```
tr -d <Zeichen>
tr -s <Zeichen>
tr <Menge1> <Menge2>
tr -c <Menge1> <Zeichen>
```

Der Befehl `tr` steht für *Translate Characters*. Durch `tr` können Sie Zeichen umsetzen, Wiederholungen reduzieren oder Zeichen löschen. `tr` nimmt dazu Daten ausschließlich von der Standardeingabe entgegen und gibt sie auf der Standardausgabe gegebenenfalls verändert wieder aus.

`-d` löscht das angegebene `<Zeichen>` (*delete*)

`-s` löscht Wiederholungen des Zeichens (*squeeze repeats*)

`-c` Ersetzt alle Zeichen, die nicht in `<Menge1>` stehen, durch das `<Zeichen>` (*complement*)

```
buch@koala:/home/buch > echo "abcdef" | tr -c "abc\n" "#"
abc###
buch@koala:/home/buch >
```

Der Befehl

```
echo "AABB  CC  DD ABCD"|tr -s " "
```

ergibt also

```
AABB CC DD ABCD
```

Werden `<menge1>` und `<menge2>` angegeben, so wird jedes Zeichen aus `<menge1>` durch das entsprechende Zeichen aus `<menge2>` ersetzt. `tr` verhält sich analog zum `sed`-Befehl `y/<menge1>/<menge2>/`, falls in `<menge1>` bzw. `<menge2>` kein Zeichenbereich angegeben wird. Der `sed`-Befehl `y/A-Z/a-z/` tauscht `A` gegen `a`, `-` gegen `-` und `Z` gegen `z` aus und nicht alle Großbuchstaben gegen Kleinbuchstaben.


```
echo "MarkuS"|tr "MS" "m\$"
```

liefert die Ausgabe

```
marku$
```

Die Ersetzungen erfolgen also zeichenweise.

Der Befehl `tr` ist einer der wenigen Befehle, der nicht mit Dateien direkt arbeiten kann, wohl aber von der Standardeingabe oder einer Pipe lesen kann. Ein nützliches Beispiel formatiert Ihnen die Zeilenumbrüche unter DOS (Kombination der ASCII-Zeichen 10 und 15 bzw. oktal 12 und 15) in Zeilenumbrüche unter Unix (nur ASCII-Zeichen 10):

```
tr -d "\015" <dostext >unixtext
```

Ein GNU-`tr` kommt auch mit dem Zeichen `NUL` (Character `0`) klar. Normalerweise ist unter C ein String dann beendet, wenn er auf ein `NUL`-Zeichen trifft. Dies führt dazu, dass die meisten `tr`-Versionen ebenfalls bei einem `NUL` die Bearbeitung einstellen. Wenn allerdings noch Zeichen folgen, kann dies sehr ärgerlich sein. GNU-`tr` hat damit keine Probleme:

```
buch@koala:/home/buch > echo -e "ab\000cd" | tr "\000" "?"
ab?cd
buch@koala:/home/buch >
```

Andere `tr` brechen die Ausgabe nach dem »b« ab.

5.9 Aufgaben

1. Erweitern Sie Skript 29 so, dass alle markierten Dateien in ein Tar-Archiv `/tmp/backup$$` geschrieben werden. Dazu müssen zunächst alle »`:`« in der Variablen `Mark` durch Leerzeichen ersetzt werden.
2. Was sagen Sie zu folgender Anweisung?

   ```
   if [ -z $@ ] ; then
      echo "Alles klar"
   fi
   ```

   Funktioniert diese Abfrage immer, manchmal oder nie? Begründen Sie Ihre Antwort!
3. Nehmen Sie Skript 19 aus Kapitel 3, und nutzen Sie die Parameterersetzung aus Abschnitt 5.4, um eine einfache Fehlerprüfung einzuführen, ob mindestens zwei Parameter übergeben wurden.
4. Wie können Sie auf Parameter nochmals zugreifen, die durch `shift` nicht mehr verfügbar sind?
5. Gehen wir die Aufgaben doch einmal etwas anders an. Wir haben hier ein Skript, welches sich mit der Abarbeitung von Optionen mittels `getopts` beschäftigt.

Das Skript soll eine beliebige Anzahl an Optionen von MRCW akzeptieren. Die Option -C verlangt ein zusätzliches Argument. Am Ende der Optionen soll eine weitere Zeichenkette angegeben werden, die mit einem »-«-Zeichen versehen als neue Positionsparameter gesetzt werden. Diese sollen dann einfach mit getopts abgearbeitet werden. Akzeptiert werden dabei Optionen -A -B -C und -D oder Kombinationen davon. Auf Argumente wird verzichtet.

Folgende Ausgabe soll das Skript liefern:

```
buch@koala:/home/buch > ./fehler.sh -WM AC
Opt=<W> OptArg=><
Opt=<M> OptArg=><
Neue Parameter setzen =(-AC)
Opt=<A> OptArg=><
Opt=<C> OptArg=><
buch@koala:/home/buch >
```

Ausgegeben wird jedoch:

```
buch@koala:/home/buch > ./fehler.sh -WM AC
Opt=<W> OptArg=><
Opt=<M> OptArg=><
Neue Parameter setzen =(-)
buch@koala:/home/buch >
```

Das Skript sieht so aus:


```
#!/bin/sh
# Skript fehler.sh
#
# Dieses Skript hat 2 Fehler!
while getopts MRC:W opt ; do
  echo "Opt=<$opt> OptArg=>$OPTARG<"
done
shift $OPTIND
optneu="-$*"
echo "Neue Parameter setzen =($optneu)"
set -- "$optneu"
while getopts ABCD opt ; do
  echo -n "Opt=<$opt>"
  echo " OptArg=>$OPTARG<"
done
exit 0
```

Woran liegt es? Als kleiner Tipp: Dieses Skript hat zwei Fehler!

5.10 Lösungen

1. Skript 29 muss um die Befehle `tarfiles=`echo $Mark | tr \":\" " " `` und `tar cvf /tmp/backup$$ $tardateien` erweitert werden:


```
# Skript 29: cwc.sh (Lösung Kapitel 5)
#
# Komplettes Skript mit Markierung und
# Parameterersetzung
# Die Dateien ab Parameter 2 im Verzeichnis $1 werden
# markiert. Wird "a" eingegeben, so werden diese Dateien
# gesichert.
#
Mark=":"
#
# Parameter $1 ist das Startverzeichnis oder Default "."
#
verz=${1:-`pwd`}
if [ -d $verz ] ; then
   cd "$verz"
fi
#
# Markierung erst ab Parameter 2 nötig, also schieben
#
shift 1
for wort in "$@" ; do
  if [ -f $wort ] ; then
     Mark="$Mark$wort:"
  fi
done
tput clear
i=2
tput cup 1 0
echo "+----------------------------------------+"
while [ $i -lt 20 ] ; do
  tput cup $i 0
  printf "! %38s!" " "
  i=`expr $i + 1`
done
tput cup 20 0
echo "+----------------------------------------+"
echo "z = PgUpw = PgDn  q = Quit  a = Tar"
tput cup 1 2
printf "%.35s" "$verz"
#
# Zeilen Offset setzen, Dateien ermitteln, Anzahl
# ermitteln
#
offset=1
tmpfile="/tmp/cwc$$.tmp"
anz=`ls "$verz"|tee $tmpfile | wc -l | cut -c-7`
```

```
akt=1
until [ "$ein" = "q" ] ; do
  # Ausgabe der Dateien
  i=0
  akt=0
  while [ $i -lt 18 -a $akt -le $anz ] ; do
   akt=`expr $i + $offset`
   tput cup `expr $i + 2` 2
   zeile=`sed -n -e "${akt}p" <$tmpfile`
   revers=`echo "$Mark" | grep ":$zeile:" `
   if [ -n "$revers" ] ; then
      tput rev
   fi
   printf "%-38.38s" $zeile
   tput sgr0
   i=`expr $i + 1`
  done
  while [ $i -lt 18 ] ; do
    tput cup `expr $i + 2` 2
    i=`expr $i + 1`
    printf "%38.38s" " "
  done
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
          # Erstmals addieren
          offset=`expr $offset + 17` ;
          if [ $offset -gt $anz ] ; then
             # Offset größer als Anzahl Zeilen
             offset=`expr $offset - 17`
          fi ;;
      "a" | "A") #
                 # Aufgabe 1
                 #
                 tarfiles=`echo $Mark | tr \":\" " " ` 
                 archiv=/tmp/backup$$
                 tar cf $archiv $tarfiles
                 tput cup 0 0
                 echo "Sicherung gestartet $archiv   "
                 ;;
      "z" | "Z")
          if [ $offset -gt 1 ] ; then
             offset=`expr $offset - 17` ;
          fi ;;
       *) ein="" ;;
     esac
  done
done
exit 0
```

2. Wenn Sie keinen oder mehrere Positionsparameter übergeben oder wenn ein Positionsparameter ein Leerzeichen enthält, wird der Testbefehl `[ -z $@ ]` Probleme bekommen, da `$@` von der Shell in Worte unterteilt wird. Daher besser:

```
if [ -z "$@" ] ; then
  echo "Alles klar"
fi
```

3. Eine Überprüfung der Übergabeparameter könnte durch

```
{$1:? "$0: Usage [-s] <Verzeichnis> <Dateinamen>"}
{$2:? "$0: Usage [-s] <Verzeichnis> <Dateinamen>"}
```

   erreicht werden.

4. Merken Sie sich zu Beginn des Skripts die Positionsparameter in einer Variablen. Dadurch können Sie nach einem `shift`-Befehl wieder zurück auf die ursprünglichen Positionsparameter umstellen:

```
sicher=$@
...
shift 2
...
# Positionsparameter zurücksetzen
set -- $sicher
...
```

5. Der erste Fehler wurde bei dem Befehl `shift $OPTIND` gemacht, da `OPTIND` auf dem ersten Parameter steht, der weder Option noch Argument einer Option ist. Der zweite Fehler erklärt sich, wenn wir uns daran erinnern, dass nach einem `set --` die Shellvariable `OPTIND` nicht automatisch auf 1 zurückgesetzt wird.


```
#!/bin/sh
# Skript fehler.sh
#
# Dieses Skript hat 2 Fehler!
while getopts MRC:W opt ; do
  echo "Opt=<$opt> OptArg=>$OPTARG<"
done
shift `expr $OPTIND - 1`   #(1. Fehler)
optneu="-$*"
echo "Neue Parameter setzen =($optneu)"
set -- "$optneu"
OPTIND=1   #(2. Fehler)
while getopts ABCD opt ; do
  echo -n "Opt=<$opt>"
  echo " OptArg=>$OPTARG<"
done
exit 0
```

Dieses Skript soll nur die Problematik demonstrieren, es ist nicht wasserdicht! Es geht davon aus, dass nur korrekte Optionen angegeben werden. Ein Aufruf `./fehler.sh -WM AC -W` wirft das Skript total aus dem Tritt.

Kapitel 6

Variablen und andere Mysterien

»Der Trainer steht teilweise voll und ganz hinter mir« –
J. Wegmann (Fußballer)

... und wenn Sie jetzt überlegen, dass wir Ihre Trainer und Sie der Spieler sind, sollte Ihnen das doch ein gutes Gefühl geben, oder etwa nicht? Wie auch immer, in diesem Kapitel beschäftigen wir uns ausschließlich mit Variablen. Diese haben viel mehr Möglichkeiten, als Sie bisher wissen. Viele der Probleme, die wir bisher umständlich gelöst haben, werden Sie am Ende dieses Kapitels viel eleganter lösen können. Nun denn, fangen wir einfach einmal an.

6.1 Typen setzen für Benutzervariablen

Benutzervariablen sind Variablen, die Sie setzen und manipulieren, um darin Daten zu speichern, die für Ihr Skript von Bedeutung sind. Diese Variablen sind zunächst einmal immer vom Typ String, was bedeutet, dass in ihnen beliebige Zeichenketten gespeichert werden können. Damit sind auch nur Manipulationen von Zeichenketten erlaubt. Selbst wenn wir in den Variablen Zahlen speichern, können wir mit ihnen nicht rechnen, sondern müssen den Umweg über `expr` gehen. Außerdem wäre es doch ganz nett, wenn es möglich wäre, Konstanten zu definieren, und der Versuch, diesen Konstanten einen neuen Wert zuzuweisen, zu einem Fehler führen würde. All dieses und noch mehr können Sie mithilfe des Befehls `typeset` erreichen, dessen Syntax wie folgt aussieht:


```
typeset <opt> <var>[=<wert>]
declare <opt> <var>[=<wert>]
```

Der Befehl `typeset` definiert die Eigenschaft `<opt>` für die Variable `<var>` und setzt gleichzeitig den `<wert>`, falls dieser angegeben wurde. `typeset` ist dabei der Befehl, den die Kornshell zur Verfügung stellt. Aus Gründen der Kompatibilität bietet auch die Bash diesen Befehl an. Leider unterstützt die Bash nicht alle Optionen, die die Kornshell anbietet. Da die Kornshell den Befehl `declare` nicht kennt, ist es sinnvoller, `typeset` zu benutzen, um die Kompatibilität der Skripten zu gewährleisten. Tabelle 6.1 zeigt die erlaubten Optionen.

Tabelle 6.1: Optionen zu typeset

`-i`	Definiert die Variable `<var>` als Integervariable (Ganzzahl). Dadurch können Sie mit diesen Variablen ohne den Umweg von `expr` rechnen. Weisen Sie dieser Variablen eine Zeichenkette zu, so gibt es weder einen Fehler bei der Zuweisung noch bei der Berechnung, sondern der Inhalt der Variablen wird als `0` gewertet, und mit diesem Wert wird weiter gerechnet.
`+i`	Hebt die Definition von `-i` wieder auf. Die Variable `<var>` verhält sich nach diesem `typeset` wieder wie eine Stringvariable.
`-r`	Setzt die Variable `<var>` auf readonly, d.h. nur lesend. Das führt dazu, dass dieser Variablen während der Laufzeit des Skripts kein neuer Wert mehr zugewiesen werden kann. Diese Variable ist somit eine Konstante. Solange das Skript läuft, kann das Attribut readonly nicht mehr aufgehoben werden.
`-a`	Definiert die Variable `<var>` als Feldvariable (Array). Mehr zum Thema Arrays findet sich später in diesem Kapitel. Diese Option existiert nur in der Bash. In der Kornshell ist es nicht möglich, Arrays mittels `typeset` zu definieren.
`-x`	Exportiert Variablen in die Shellumgebung (siehe Kapitel 6.5).

Konstanten sollten Sie direkt am Anfang eines Skripts setzen. Konstanten enthalten, wie der Name schon deutlich macht, konstante Werte, die innerhalb des Skripts mehrfach benötigt werden. Als bestes Beispiel dazu dient unsere Temporärdatei, die wir häufig als `tmpfile` bezeichnet haben. Anstatt jedes Mal den Namen zu schreiben, setzen wir diesen einmal und geben danach ausschließlich den Namen der Konstanten an. Muss der Name später einmal geändert werden (z.B. weil die alten Dateinamen für die Temporärdaten noch nicht eindeutig durch den Einsatz von `$$` waren), so müssen Sie nur eine Stelle ändern, was eindeutig pflegeleichter ist.

6.2 Arithmetische Ausdrücke

Bisher haben wir immer `expr` genutzt, wenn es etwas zu berechnen gab. Es wird langsam Zeit, diese umständliche Vorgehensweise durch etwas Eleganteres zu ersetzen. Bedenken Sie aber, dass diese Vorgehensweise nur in der Kornshell und der Bash funktioniert (siehe auch Anhang A). Wurde eine Variable mittels `typeset -i` als Integer definiert, so können Sie ganz einfach damit rechnen:

```
...
typeset -i erg=0
typeset -i var
var=10
erg=11*var
echo "Ergebnis=$erg"
...
```

In arithmetischen Ausdrücken brauchen Sie tatsächlich kein `$`, um Variablen zu dereferenzieren. Anders sieht es allerdings bei Positionsparametern aus. Da diese ja durch `$1` bis `$9` angesprochen werden, würde der Verzicht auf das `$`-Zeichen zu korrekten Zahlen (nämlich 1 bis 9) im Ausdruck führen. Das Ergebnis wäre sicherlich unerwünscht. Nutzen wir das Wissen dazu, die letzte Version von Skript 30 aus Kapitel 5 nochmals zu verbessern. Wir verzichten auf den Einsatz von `expr` und setzen arithmetische Ausdrücke ein:

Speichern Sie das Skript ab, und rufen Sie es erneut auf. Ihr Skript verhält sich genauso wie die erste Version.


```
# Skript 30: Verzeichnisbelegung ausgeben
#
# Zwei Verzeichnisse werden übergeben. Keine
# Sicherheitsüberprüfungen, damit das Skript kurz bleibt
#
# 1. Parameter ermitteln und tmp-Dateinamen setzen
typeset -r tmpfile="/tmp/erg$$"
rm -f $tmpfile
verz1=$1
verz2=$2
# 2. Dateien aus Verzeichnis 1 ermitteln
find $verz1 -type f -print 2>/dev/null >$tmpfile
# 3. Aus Ergebnisdatei lesen und addieren
4<$tmpfile
typeset -i erg1=0
while read zeile <&4 ; do
  set -- `ls -l "$zeile"`
  erg1=erg1+$5
done
```

```
# 4. Dateien aus 2. Verzeichnis anhängen
find $verz2 -type f -print >> $tmpfile 2>/dev/null
typeset -i erg2=0
# 5. Die 2. Schleife mal mit awk
while read zeile <&4 ; do
  typeset -i datgr=`ls -l "$zeile" | awk '{ print $5 }'`
  erg2=erg2+datgr
done
4<&-
printf "%-20.20s   %10d Bytes\n" $verz1 $erg1
printf "%-20.20s   %10d Bytes\n" $verz2 $erg2
echo "Die Verzeichnisse $verz1 und $verz2 enthalten `wc -l <$tmpfile`
Dateien"
anz=`find $verz1 $verz2 -type d -print 2>/dev/null | wc -l`
echo "Insgesamt existieren genau" $anz "(Unter-)Verzeichnisse"
rm $tmpfile
exit 0
```

Die Typisierung wird in der Shell nicht konsequent durchgehalten. Es ist durchaus möglich, eine Variable vom Typ Ganzzahl zu definieren und ihr dennoch eine Zeichenkette zuzuweisen:

```
...
buch@koala:/home/buch > typeset -i ohje=123
buch@koala:/home/buch > ohje="Down Under"
buch@koala:/home/buch > echo $ohje
0
buch@koala:/home/buch >
...
```

In Hochsprachen wie Pascal oder C würde eine solche Zuweisung mit einem Fehler bestraft werden.

In Tabelle 6.2 sind alle Operatoren aufgeführt, die die Shell innerhalb von arithmetischen Ausdrücken erkennt.

Tabelle 6.2: Gültige Operatoren in arithmetischen Ausdrücken

Operator	Bedeutung
- +	Vorzeichen
! ~	Logische und bitweise Negierung
* / %	Multiplikation, Division, Restwert bei ganzzahliger Division
+ -	Addition, Subtraktion
<< >>	Bitweises Schieben nach links bzw. rechts
<= >= < >	Vergleich

Operator	Bedeutung
`== !=`	Gleichheit und Ungleichheit
`&`	Bitweises UND
`^`	Bitweises XOR (exklusives Oder)
`|`	Bitweises ODER
`&&`	Logisches UND
`||`	Logisches ODER

`<aus1>?<aus2>:<aus3>`	Bedingte Auswertung. Ist identisch mit der Abfrage:
	`if [ <aus1> ] ; then`
	`  <aus2>`
	`else`
	`  <aus3>`
	`fi`
	Der einzige Unterschied liegt darin, dass if-Abfragen nicht innerhalb von arithmetischen Ausdrücken erlaubt sind.

`= += &=`
`*= -= ^=`
`/= <<= |=`
`%= >>=`
	Zuweisungen: Steht ein Operator vor der Zuweisung, so wird die Variable links vom Gleichheitszeichen = mit dem Ausdruck rechts durch den Operator verbunden.
	`a+=2` ist somit identisch mit `a=a+2`

Die Reihenfolge der Operatoren von oben nach unten und von links nach rechts ist identisch mit der Reihenfolge, in der die Operatoren ausgewertet werden. Bei der Auswertung eines solchen Ausdrucks wird nicht auf einen Überlauf des Integerbereichs geprüft (d.h., die Shell prüft nicht, ob das Ergebnis einer Operation zu groß oder zu klein ist, um in einer Ganzzahl darstellbar zu sein). Nur bei der Division durch Null wird ein Fehler erkannt und ausgewiesen. Klammern `()` innerhalb der Ausdrücke sind ebenfalls erlaubt und ermöglichen wie in der Mathematik das Bilden komplexer Terme. Dabei werden geklammerte Ausdrücke schrittweise von innen nach außen ausgewertet.

Falls Sie mit einer anderen Zahlenbasis als 10 (Dezimalsystem) arbeiten wollen, haben Sie die Möglichkeit, eine 0 voranzustellen, um oktale (Basis 8) Werte anzugeben. Sowohl ein `0X` als auch `0x` stellen Werte im Hexadezimalsystem dar (Basis 16, Werte 10–15 werden durch die Buchstaben A–F dargestellt). Möchten Sie eine andere Basis als 8, 10 oder 16 nutzen, so stellen Sie die Basis gefolgt von # direkt vor die Zahl. Ist die Basis größer als 10, so werden die Ziffern durch die Zeichen `a-z`, `A-Z` , _ und `@` in dieser Reihenfolge herangezogen. (Kommt eine Darstellung des Zahlensystems noch mit Buchstaben von `a-z` aus, so ist Groß- und Kleinschreibung egal.)

Wenn Sie Ausdrücke an Stellen einsetzen wollen, in denen es nicht um eine Zuweisung geht, so müssen Sie diese Ausdrücke in `$(())` setzen:

```
buch@koala:/home/buch > echo $((0xA))
10
buch@koala:/home/buch >
```

6.3 Feldvariablen/Arrays

Sowohl Bash (ab Version 2.0) als auch die Kornshell bieten Feldvariablen (Arrays), die mittlerweile schon an mehreren Stellen angesprochen wurden. Nun ist es an der Zeit, auch dieses Mysterium zu enthüllen.

Feldvariablen können Sie mit Häusern in einer Straße vergleichen: Jedes Haus hat eine eindeutige Hausnummer, mit der sich ein Haus finden lässt. Die kleinste Nummer ist 1, und die höchste Nummer ist unbestimmt. Nicht alle Nummern müssen auf der Straße belegt sein, aber keine Nummer ist zweimal vergeben.

Eine Feldvariable ist nichts anderes als eine Straße, bei dem jedes Haus einen Wert speichern kann. Auch hier kann eine Hausnummer (im Fachjargon *Index* genannt) nur einmal vergeben werden, aber nicht alle Indices müssen belegt werden. Die maximale Anzahl an Hausnummern ist nicht beschränkt. Im Unterschied zur Straße ist die erste Hausnummer (der erste Index) `0` und nicht `1`.

Feldvariablen können Sie auf zwei verschiedene Weisen definieren. Eine Deklaration erfolgt durch

`typeset -a <name>` oder `typeset -a <name>[<nr>]`,

wobei ein gegebenenfalls eingetragenes `<nr>` ignoriert wird. Diese Methode funktioniert nur in der Bash, da die Kornshell die Definition von Arrays mittels `typeset` nicht unterstützt.

Die zweite Möglichkeit besteht darin, eine Zuweisung zu machen:

`<var>[<index>]=<wert>` `z.B.: feld[10]=10`

Zusätzlich können Sie auch die Attribute von Feldvariablen mittels `typeset` verändern und auf readonly oder Integer setzen. Wie auch immer, die Änderung eines Attributs für einen Eintrag des Arrays bewirkt, dass die Attribute des gesamten Arrays entsprechend angepasst werden.

Die Zuweisung Eintrag für Eintrag funktioniert zwar, ist aber nicht immer effizient. Effizient wäre es, die Inhalte aller Elemente in einem Vorgang zuzuweisen. Aus diesem Grunde schauen wir uns einmal an, wie so etwas in der Bash bzw. in der Kornshell durchzuführen ist.

Betrachten wir zunächst die Version für die Bash: Wenn Sie in der Bash mehrere Einträge zuweisen wollen, so sieht die Zuweisung wie folgt aus: Links vom Gleichheitszeichen steht der Name des Arrays (ohne []!), gefolgt vom = und dann in runden Klammern die zuzuweisenden Werte, getrennt durch Leerzeichen. Vor und nach dem = dürfen aber keine Leerzeichen stehen, sonst erkennt die Bash nicht, dass es sich um eine Zuweisung handelt.

```
...
typeset -a feld
feld=(1 10 ba 4)
...
```

Obiges Minibeispiel führt dazu, dass `feld[0]=1`, `feld[1]=10`, `feld[2]=ba` usw. ist.

Die Kornshell beschwert sich bei dieser Methode über eine ihrer Meinung nach deplatzierte Klammer »(«. Die `ksh` akzeptiert nur diese Syntax:

```
...
set -A feld 1 10 ba 4
...
```

Wie Feldvariablen angelegt und Werte zugewiesen werden, sollte an dieser Stelle klar sein, nur wurde bisher nicht ein Wort darüber verloren, wie diese Werte wieder abgefragt werden können. Ein Eintrag einer Feldvariablen wird angesprochen durch

```
${<var>[<index>]}
```

Dabei sind die geschweiften Klammern notwendig, um Konflikte mit den Ersatzzeichen aus Kapitel 2 zu verhindern.

Ist `<index>` gleich `*` oder `@`, so werden alle Einträge des Arrays durch Leerzeichen getrennt ausgegeben. Der einzige Unterschied zwischen `*` und `@` ist die Art und Weise, wie sich die Auflistung verhält, wenn der Array in Anführungszeichen gesetzt wird. `${<var>[*]}` wird in diesem Falle zu einem einzigen Wort erweitert, während `${<var>[@]}` zu einer Liste von Wörtern erweitert wird. Siehe zu diesem Thema auch Kapitel 5, wo wir `$*` und `$@` besprochen haben. Wenn Sie feststellen wollen, wie lang ein Eintrag innerhalb des Arrays in Zeichen ist, so stellen Sie ein `#` vor den Namen:

```
...
echo ${#feld[0]}
echo ${#feld[2]}
...
```

Bezogen auf die Zuweisung von weiter oben, würde das `echo` 1 und 2 ausgeben. Wenn für den Index ein `*` oder `@` angegeben wird, so wird die Anzahl der Einträge der Feldvariablen ausgegeben. Dabei zählt die Shell die Anzahl

der wirklich belegten Indizes oder, auf die Straßenmetapher bezogen, die Anzahl der gebauten Häuser auf der Straße, nicht die Anzahl der Grundstücke.

```
buch@koala:/home/buch > typeset -a wert
buch@koala:/home/buch > wert[10]=12
buch@koala:/home/buch > echo ${wert[*]}
12
buch@koala:/home/buch > echo ${#wert[@]}
1
buch@koala:/home/buch >
```

Wird der Befehl `read` mit der Option `-a` angegeben, so liest `read` eine Zeile ein, unterteilt die Eingabe in Wörter und weist diese der angegebenen Feldvariablen zu. Das erste Wort wird in Index `0` gespeichert, das zweite Wort in Index `1` usw. (Die Option `-a` für `read` gibt es in der `ksh` nicht.)

Nach der ganzen Theorie ist es an der Zeit, das Ganze praktisch einzusetzen.

Der Befehl `grep` hat übrigens noch einige nützliche Optionen:

- `-c` gibt aus, wie oft der Suchbegriff in der Datei gefunden wurde.
- `-v` invertiert die Suche. Das heißt, es werden nur die Zeilen ausgegeben, die den Suchbegriff nicht enthalten.
- `-i` sucht unabhängig von der Groß-/Kleinschreibung.
- `-n` gibt die Zeilennummer aus, in der der Text gefunden wurde, und dann die Zeile.

Schreiben wir also ein Miniskript, das eine beliebige Anzahl Parameter akzeptiert und in allen Dateien, die auf `*.txt` enden, die übergebenen Parameter sucht und für jede Datei ausgibt, wie oft alle Suchbegriffe in ihr gefunden wurden.


```
# Skript 31: Arrays
# #
# Sucht in allen Dateien die auf ".txt" enden
# die übergebene Wörter (Parameter $1 ...)
# und gibt aus, wie oft die Wörter (als Summe)
# in jeder Datei gefunden wurden.
#
# ACHTUNG: Dieses Skript geht davon aus, dass
#          sich die Anzahl an ".txt" Dateien
#          während der Laufzeit nicht ändert
#
# 1. Alle Dateien in den Array und
#    gvAnzahl als Arrays anlegen
gvDateien=(`echo *.txt`)
typeset -ia gvAnzahl
```

```
# 2. Falls keine Parameter angegeben, mittels
#    read einlesen
if [ $# -eq 0 ] ; then
   echo -n "Bitte geben Sie die zu suchenden Wörter ein: "
   read gvWorte
  else
   typeset -a gvWorte
   gvWorte="$@"
fi
# 3. Jedes einzelne Wort mit grep suchen
for gvWort in $gvWorte ; do
   # 4. Jede einzelne Datei durchsuchen.
   typeset -i gvInd=0
   for gvDatei in ${gvDateien[*]} ; do
       # 5. Anzahl der Treffer bestimmen
       typeset -i gvTreffer=`grep -ci "$gvWort" $gvDatei`
       gvAnzahl[$gvInd]=${gvAnzahl[$gvInd]}+gvTreffer
       gvInd=gvInd+1
   done
done
# 6. Und jetzt die Daten ausgeben
gvInd=0
while [ $gvInd -lt ${#gvDateien[*]} ] ; do
   echo "${gvDateien[$gvInd]}  Treffer: ${gvAnzahl[$gvInd]}"
   gvInd=gvInd+1
done
exit 0
```

Schauen wir uns an, was das Skript jetzt genau macht:

1. Es werden zwei Arrays angelegt. Einer enthält alle Dateinamen, die durchsucht werden sollen. Der andere enthält die Summe, wie oft die Wörter gefunden wurden. Dabei werden die Treffer für `gvDatei[0]` in `gvAnzahl[0]` gespeichert usw.
2. Die Suchworte werden mittels `read` eingelesen oder von den Parametern übernommen.
3. Jetzt wird die eigentliche Arbeit durchgeführt: Für jedes Wort wird die äußere `for`-Schleife durchlaufen. Diese setzt den Index für den Array mit den Summen jeweils auf 0 zurück, damit die korrekten Einträge in `gvAnzahl` summiert werden.
4. Die innere `for`-Schleife durchläuft alle in `gvDateien` eingetragenen Dateinamen. Durch `${gvDateien[*]}` wird jeder Dateiname als einzelnes Wort interpretiert:

```
for gvDatei in anhanga.txt anhangb.txt ... ; do
```

5. Das grep sucht die Wörter unabhängig von Groß-/Kleinschreibung und gibt nur eine Zahl aus. Bei mehreren Dateien wird zusätzlich noch der Dateiname ausgegeben:

```
buch@koala:/home/buch > grep -ci Array anhanga.txt
1
buch@koala:/home/buch > grep -ci Array anhan*
anhanga.txt:1
anhangb.txt:0
anhange.txt:0
buch@koala:/home/buch >
```

   Die Anzahl der Treffer wird in gvAnzahl summiert, und gvInd wird für den nächsten Durchlauf erhöht.

6. Schließlich werden die Ergebnisse ausgegeben. Hier ist nur interessant, dass durch ${#gvDateien[*]} die Anzahl der Einträge im Array ermittelt wird.

Kommen wir noch einmal kurz zu den Ersetzungen aus dem letzten Kapitel zurück. Die Bash bietet für Arrays zusätzliche Funktionalitäten an. Die Ersetzungen ${<var>%<muster>}, ${<var>%%<muster>}, ${<var>#<muster>} und ${<var>##<muster>} funktionieren auch mit Arrayvariablen. Ist <var> eine Arrayvariable, die mit @ oder * indiziert wird, so werden die Ersetzungen für alle Einträge innerhalb der Feldvariablen durchgeführt.

Die Ersetzung ${<var>:<offset>:<länge>} gibt bei Arrayvariablen, die durch * oder @ indiziert werden, die passenden Felder aus dem Array zurück. Für alle Ersetzungen gilt: Folgt hinter dem Variablennamen kein [*] oder [@], so wird der Eintrag [0] herangezogen.

```
buch@koala:/home/buch > demo=(abc ade abacb)
buch@koala:/home/buch > echo ${demo:1:2}
bc
buch@koala:/home/buch > echo ${demo[*]:1:2}
ade abacb
buch@koala:/home/buch > echo ${demo[*]#a*b}
c ade acb
buch@koala:/home/buch > echo ${demo#a*b}
c
buch@koala:/home/buch >
```

Andere Sprachen kennen weitere Versionen von Feldvariablen, wie assoziative Arrays oder mehrdimensionale Arrays. Diese sind in der aktuellen Version der Bash nicht verfügbar. Immerhin erlauben die Arrays in der Bash (und auch in der Kornshell) *Löcher*, d.h., nicht jeder Index muss auch wirklich mit Werten belegt werden.

6.4 Variablen löschen

Bis zu diesem Abschnitt haben Sie jede Menge Variablen angelegt und mit ihnen gearbeitet. Jedoch kommt der Zeitpunkt im Leben eines Programmierers, an dem er eine von ihm definierte Variable nicht mehr braucht und löschen will. An diesem schicksalsschweren Punkt sind wir nun angekommen :)

Stellt sich nur die Frage, warum Variablen überhaupt wieder gelöscht werden sollten. Schließlich bieten andere Sprachen dies auch nicht.

- Funktionalität:

  Wie bereits erwähnt, enthält die Shellumgebung Variablen, die von anderen Programmen/Skripten ausgewertet und benötigt werden. Beispiele hierzu wären HOME und TERM. Es kann genauso gut sein, dass ein Befehl eine bestimmte Verhaltensweise nur an den Tag legt, wenn eine Variable (nicht) existiert. Ein ganz simples Beispiel wären die Ersetzungen via: `${TERM:-"linux"}`

- Ordnung halten:

  Ihr Skript braucht weniger Speicherplatz, um die Daten zu speichern, wodurch der Zugriff auf die Daten schneller wird. Außerdem sollten alle Ressourcen – und damit auch Daten – wieder freigegeben werden, wenn sie nicht benötigt werden.

```
unset <var>
unset <array>[<index>]
```

Die übergebene Variable `<var>` wird gelöscht und ist danach der Shell nicht mehr bekannt. Wird ein Array mit Indexangabe an `unset` übergeben (Version 2), so wird nur der Eintrag aus dem Array entfernt, der Array selbst existiert weiter, nur `${#<array>[*]}` wird um eins reduziert. Ist `<var>` ein Array, so wird der komplette Array gelöscht.

Der Befehl `unset` gibt eine 0 als Exitstatus zurück, wenn die Variable gelöscht werden konnte, sonst eine 1. Als Fehler wird dabei das Löschen einer Variablen mit dem *readonly*-Attribut angezeigt oder der Versuch, eine bereits gelöschte (oder noch nicht genutzte) Variable nochmals zu löschen.

Ein `unset txt[0]` bewirkt nur, dass der Array `txt` keinen Eintrag mit dem Index 0 mehr hat. Es bewirkt nicht, dass alle Einträge ab Index 1 um eins nach unten »rutschen«. Das heißt, `${txt[0]}` ist danach leer und enthält *nicht* den Wert des alten Eintrags mit dem Index 1!

Dabei bedeutet *leer*, dass eine Zeichenkette mit der Länge 0 abgespeichert wurde (`""`). `unset` oder nicht existent ist eine Variable, in der keine Werte abgespeichert wurden. Krass formuliert: Eine Variable ist `unset`, wenn sie noch nicht benutzt wurde (bzw. mittels `unset` wieder in diesen Status versetzt wurde).

6.5 Umgebung/Environment

Wird ein Programm aufgerufen, so wird ihm ein Array von Zeichenketten übergeben. Dieser Array wird (Shell-)Umgebung oder Environment genannt. Jeder Eintrag in der Umgebung besteht aus einer Zeichenkette im Format `<name>=<wert>`. Ein Skript kann auf den `<wert>` eines solchen Eintrags durch `$<name>` zugreifen, und damit unterscheidet sich der Zugriff auf die Einträge nicht von den Ihnen bisher bekannten Variablen. Aus diesem Grunde werden die Einträge ins Environment häufig auch Umgebungsvariablen genannt.

Umgebungsvariablen beeinflussen das Verhalten der Shell und Befehle. So steht in der Variablen `PAGER` das Pagerprogramm, welches `man` nutzt. Ändern wir das einfach ab:

```
buch@koala:/home/buch > apg=$PAGER
buch@koala:/home/buch > export PAGER="wc"
buch@koala:/home/buch > man ksh
   3894   19696  182661
buch@koala:/home/buch > export PAGER=$apg
buch@koala:/home/buch >
```

`Man` nutzt `$PAGER`, um die Manpages seitenweise auszugeben. Wird `$PAGER` geändert, so wird der neu eingetragene Befehl per Pipe mit den Seiten beschickt. Welche Variablen in der Umgebung eingetragen sind, erfahren Sie durch den Aufruf von `printenv`.

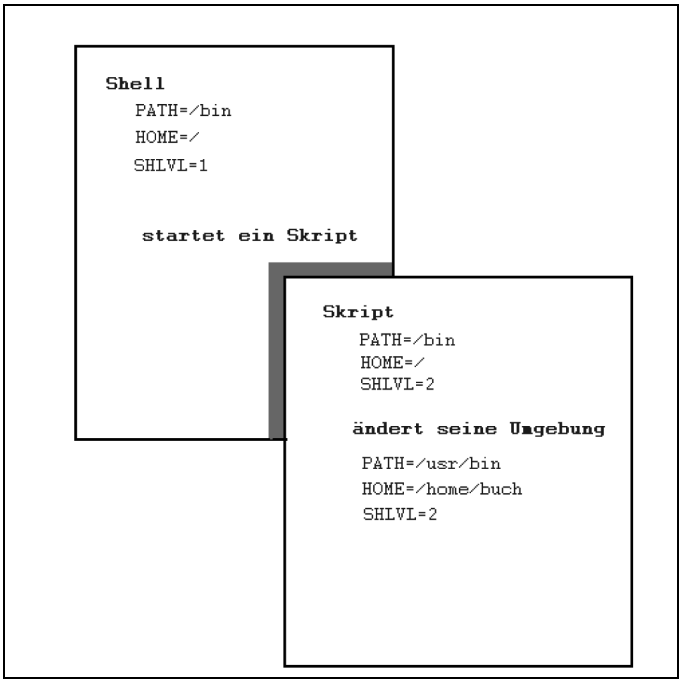


Abb. 6.1: Das Skript erbt die Umgebung von der aufrufenden Shell (SHLVL steht für Shell-Level)

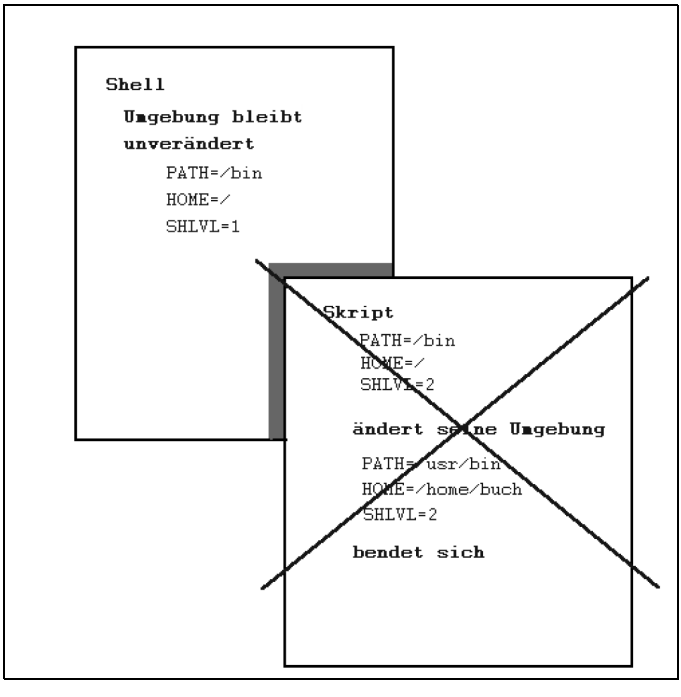


Abb. 6.2: Nach Beendigung des Skripts gehen alle Änderungen seiner Umgebung verloren

Startet die Shell nun ein Skript bzw. Programm, so wird die Umgebung der Shell kopiert, wobei nur exportierte Variablen beachtet werden. Mit der Kopie der Umgebung arbeitet dann das Skript bzw. Programm. Änderungen in dieser Kopie gehen mit der Beendigung des Skripts bzw. Programms verloren. Dies gilt selbst dann, wenn die betroffenen Variablen exportiert werden.

Legt Ihr Skript Variablen an, so muss es diese explizit über ein `typeset -x` oder ein `export` in die Umgebung exportieren. Variablen, die nicht exportiert wurden, sind nur für Ersetzungen innerhalb des Skripts bekannt. Befehle oder Skripten, die aus diesem Skript heraus aufgerufen werden, kennen diese Variablen schon nicht mehr.

Es ist allerdings auch möglich, Variablen genau für *einen* Aufruf in die Umgebung aufzunehmen. Um auf das Beispiel von weiter oben mit der Variablen `PAGER` zurückzukommen: Dies funktioniert für genau einen Aufruf von `man` wie folgt:

```
buch@koala:/home/buch > PAGER="wc" man ksh
   3894   19696  182661
buch@koala:/home/buch >
```

Vielleicht wenden Sie jetzt ein, dass unsere Skripten ja z.B. ein `find` oder `wc` mit Variablen ausgeführt haben. Auf den ersten Blick haben Sie da Recht, bedenken Sie allerdings, was die Shell aus folgender Zeile macht:

```
find $verz -type f -print
```

wird zunächst in Wörter aufgeteilt:

```
»find« »$verz« »-type« »f« »-print«
```

dann wird `$verz` als Variable von der Shell (!) erkannt und ausgetauscht

```
»find« »/home/buch« »-type« »f« »-print«
```

… und erst jetzt wird `find` von der Shell aufgerufen.

Die Skripten 32 und 33 sollen uns diese Problematik noch einmal praktisch zeigen. Skript 32 exportiert eine Variable, gibt diese aus und ruft Skript 33 auf. Skript 33 gibt den Inhalt der Variablen aus, belegt sie mit einem neuen Wert, exportiert die Variable, gibt den Inhalt aus und beendet sich. Daraufhin gibt Skript 32 nochmals den Inhalt der Variablen aus.


```
# Skript 32: Export
#  Demonstriert die Auswirkungen
#  von "export" beim Aufruf von
#  weiteren Skripten
#
a=Bettina
export a
```

```
echo "Name=/$a/"
skript33.sh
echo "a="$a
exit 0
```


```
# Skript 33: Export
# Dieses Skript übernimmt die Umgebung von Skript 32 als Kopie
# und manipuliert sie. Die nichtvorhandenen Auswirkungen auf
# Skript 32 gibt dieses nach dem Aufruf von Skript 33 aus
echo "in 33 #1: ($a)"
export a=Christa
echo "in 33 #2: ($a)"
exit 0
```

Der Aufruf ergibt:

```
buch@koala:/home/buch > skript32.sh
Name=/Bettina/
in 33 #1: (Bettina)
in 33 #2: (Christa)
a=Bettina
buch@koala:/home/buch >
```

Die Tatsache, dass die Shell aufgerufenen Programmen nur Kopien der Originalumgebung zur Verfügung stellt, hat durchaus Vorteile. So müssen Sie nicht alle Variablen löschen oder sicherstellen, dass einige Variablen wieder ihren alten Wert erhalten, den sie vor dem Aufruf Ihres Skripts hatten.

Bleibt nur noch die Frage offen, wie Sie ermitteln können, welche Variablen in der Umgebung abgelegt sind und welche Werte sie haben. Dazu gibt es zwei Möglichkeiten: `env` oder `printenv` gibt alle Variablen und ihre Werte aus, die in die Umgebung exportiert wurden. `set` ist ein Befehl, der zusätzlich zu den Ausgaben von `env` auch noch die Shellvariablen, Shellfunktionen und Ihre Werte ausgibt.

Feldvariablen können nicht in die Umgebung exportiert werden. Der Versuch, einen Array in die Umgebung aufzunehmen, scheitert leider ohne Fehlermeldung:

```
buch@koala:/home/buch > typeset -a oz
buch@koala:/home/buch > export oz
buch@koala:/home/buch > printenv | grep "oz"
buch@koala:/home/buch >
```

6.6 Shellvariablen

Neben den vom Benutzer definierten Variablen und den Umgebungsvariablen gibt es noch Variablen, die von der Shell automatisch angelegt und verwaltet werden oder von besonderer Bedeutung für die Shell sind. Drei von diesen Shellvariablen haben wir bereits bei `getopts` kennen gelernt: `OPTIND`, `OPTERR` und `OPTARG`. Die Tatsache, dass die Shellvariablen von besonderer Bedeutung sind, schließt nicht aus, dass die Variablen nicht in die Umgebung exportiert werden. So sind z.B. `SHLVL`, `HOME` und `PATH` in der Umgebung zu finden. Genug der langen Vorrede, legen wir los.

6.6.1 RANDOM

`RANDOM` ist ein Zufallszahlengenerator. Jeder Zugriff auf `$RANDOM` gibt eine andere, zufällige positive Ganzzahl zurück. Zufallszahlen werden gebraucht, wenn ein Skript nicht mit fest verdrahteter Vorgehensweise zum Ziel kommen kann, sondern sich bei jedem Aufruf unterschiedlich verhalten soll. So könnten Sie bestimmte Stichproben durchführen, z.B. ob bestimmte Benutzer Vorgaben (wie Plattenplatz) verletzen. Anstatt jeden einzelnen Benutzer zu testen, könnten Sie per Zufall einen bestimmen und diesen dann überprüfen. Das beste Beispiel für eine Zufallszahlenanwendung ist jedoch der Sprücheklopfer `fortune`. Jeder Aufruf fördert einen anderen Spruch zu Tage, mal witzig, mal dumm, mal nachdenklich. Die Datenbank der Sprüche ist riesig und leider auf Englisch.

Nehmen wir doch einfach einmal die Sprüche aus diesem Buch und noch einige mehr, die Ihnen einfallen, und packen diese, durch # getrennt, in eine Datei `spruch.txt`. Aus dieser Datei bestimmt unser Skript einen Spruch zufällig und gibt ihn einfach aus. Nehmen wir an, folgendes steht in der Datei `spruch.txt`, die wir als Speicher für unsere Sprüche nutzen wollen:

```
buch@koala:/home/buch > cat spruch.txt
#
Die Optimisten behaupten, dass wir in der besten aller möglichen
Welten leben, die Pessimisten befürchten, dass dies stimmt
#
Murphy's Gesetz: Alles was schief gehen kann geht schief
#
O'Tooles Zusatz zu Murphys Gesetz: Murphy war Optimist
#
Gewalt löst keine Probleme.
#
Und aus dem Chaos sprach eine Stimme zu mir: Lächele und sei froh, es
könnte schlimmer kommen. Da lächelte ich und war froh und es kam schlimmer.
#
Dunkel war's, der Mond schien helle, als ein Wagen blitzeschnelle langsam um
die Ecke fuhr.
```

```
#
Die Chancen stehen 70:40. Fußballer Abramzcik (Schalke) zu den Siegchancen im
nächsten Spiel
#
buch@koala:/home/buch/skript >
```

Unser Skript bestimmt die Anzahl der Sprüche in der Datei, ermittelt eine Zufallszahl in dem Bereich 0 bis Anzahl Sprüche minus 1 und gibt den Spruch mit dieser Nummer aus.

1. Die Anzahl der Sprüche ist identisch mit der Anzahl der Zeilen, die mit # anfangen, minus 1. Da in unserer Datei ein # sowohl am Anfang als auch am Ende steht, ist ein # mehr vorhanden, als Sprüche existieren.
2. Wir teilen die Zufallszahl durch die Anzahl Sprüche und nehmen den ganzzahligen Rest dieser Division. Der ist im Bereich `0<= x <` Anzahl.
3. `grep -n` gibt Dateiname, Zeilennummer und Zeile aus. Wenn wir nach # suchen, so finden wir die Anfangszeilen aller Sprüche. Diese Informationen speichern wir in einem Array. Dann gilt, dass die Endzeile des ersten Spruches gleich der Anfangszeile des zweiten Spruches ist usw.
4. Die Zeilen aus dem Array von Punkt 3. verweisen ja auf die Zeilen, in denen das # abgelegt wurde. Da diese nicht ausgegeben werden sollen, verschieben sich Anfangs- und Endzeile jeweils um 1 nach vorne bzw. nach hinten.
5. Die Zeile wird ausgegeben. Dies können wir wie bereits bekannt mit einer Kombination aus `head` und `tail` erreichen. Es wäre zwar besser, `sed` zu nutzen, aber da wir nicht zu sehr vorgreifen wollen, möchten wir erst im Kapitel über `sed` darauf zu sprechen kommen, wie Zeilenbereiche von `sed` ausgegeben werden können.


```
# Skript 34: Ein simpler Sprücheklopfer
#
# 1. Die Anzahl an Sprüchen ermitteln
typeset -i anz=`grep '^#' spruch.txt | wc -l`
anz=anz-1
# 2. Ein Spruch per Zufall ermitteln, nr enthält
#    Arrayindex der Startzeile und ende den Index der
#    Endezeile
typeset -i nr=$RANDOM
nr=nr%anz
typeset -i ende=nr+1
# 3. Zeilennummern mittels grep ermitteln
#    in $zeile steht "1:# 4:# 6:# 8:# 10:# 13:# 16:# 19:#"
#    Das weisen wir jetzt dem Array name zu.
#    name[0]="1:#"  name[1]="4:#" usw.
#   ^#  sucht nur am Zeilenanfang
zeile=`grep -n '^#' spruch.txt`
name=($zeile)
```

```
st=${name[$nr]}
# 4. Aus den Arrayeinträgen nur die Zahlen berücksichtigen
#    und Start/Endezeilen berechnen
st=`echo $st|cut -d: -f1`
enorg=${name[$ende]}
typeset -i en=`echo $enorg|cut -d":" -f1`
en=en-1
echo $en
az=`expr $en - $st`
# 5. Zeilenbereich ausschneiden
head -$en spruch.txt|tail -$az
exit 0
```

Das Skript hält sich an die Überlegungen, die wir weiter oben angestellt hatten, und unterteilt es entsprechend in die Punkte 1 bis 5. Ein Punkt bleibt jedoch noch zu klären und das ist der Befehl `cut -d":" -f1`. Mit der Option `-d` (Delimiter = Feldtrenner) wird `cut` angewiesen, nicht Zeichenbereiche aus der Standardeingabe auszuschneiden, sondern Spalten. Das Zeichen, welches nach dem `-d` folgt, trennt die einzelnen Spalten voneinander ab. Mittels `-f` wird dann die Spalte bestimmt, die `cut` ausschneiden soll. Die erste Spalte ist demnach `-f1`. Unsere Einträge in `name` sehen ja wie folgt aus: `1:#` oder `19:#`. Nutzen wir als Spaltentrenner den Doppelpunkt, so haben wir jeweils zwei Felder, mit den Werten `1` und `#` bzw. `19` und `#`. So kommen wir schnell und einfach an die Zahlen innerhalb der Zeichenkette.

Sollte die Variable `RANDOM` durch ein `unset` gelöscht werden, so verliert sie ihre besondere Fähigkeit. Ein späteres Anlegen durch ein `set` oder `export` führt nur dazu, dass eine Variable `RANDOM` angelegt wird. Zufallszahlen generiert diese Variable aber nicht mehr.

6.6.2 SHLVL

Diese Shellvariable gibt an, wie viele Shells ineinander verschachtelt sind. Nach der Anmeldung wird die Loginshell gestartet, die Ihre Eingaben entgegennimmt und darauf z.B. durch Starten von Skripten reagiert. Wenn Sie in der Loginshell sind, so ist der Wert dieser Variable 1.

Rufen Sie aus der Shell ein Skript auf, so wird eine neue Shell gestartet, welche die Umgebung der Loginshell als Kopie übernimmt (siehe Kapitel 6.5), und die Skriptausführung übernimmt. In diesem Moment sind zwei Shells aktiv:

- Die Loginshell, deren gesamte Funktion sich zu diesem Zeitpunkt darauf beschränkt, auf die Beendigung der zweiten Shell zu warten.
- Die zweite Shell, welche das Skript ausführt. In dieser Shell hat `SHLVL` den Wert 2. Wird das Skript beendet, so beendet sich die aufgerufene Shell, und die Loginshell übernimmt wieder die Kontrolle.

Sollte das Skript ein weiteres Skript aufrufen, so wird erneut eine Shell gestartet. Diese Shell übernimmt als Kopie die Umgebung der zweiten Shell und führt das Skript aus. Währenddessen wartet die Loginshell auf Shell 2 und diese wiederum auf die Beendigung von Shell 3. In der dritten Shell ist `SHLVL` dann gleich 3.

Vereinfacht ausgedrückt, definiert `SHLVL` die Verschachtelungstiefe der aufgerufenen Shells, wobei die Loginshell `SHLVL = 1` hat.

Das gleiche Prinzip kommt auch zur Anwendung, wenn Shells im Hintergrund gestartet werden. Auch in diesem Fall wird `SHLVL` erhöht, wenn eine neue Instanz der Bash aktiviert wird.

Wie bereits mehrfach erwähnt, führt ein `exit` zur Beendigung der Shell, die das Skript ausführt. In einer Loginshell ist ein `exit` aber gleichbedeutend mit einer Abmeldung, was nicht unbedingt erwünscht ist. Von daher kann `SHLVL` genutzt werden, um ein `exit` in solchen Fällen zu vermeiden.

6.6.3 PIPESTATUS

Wie wir bereits in Kapitel 2 festgestellt haben, ist der Rückgabewert einer Pipe der Rückgabewert des letzten Befehls innerhalb der Pipe. Sollte jedoch der letzte Befehl einen Exitstatus `0` (also OK) zurückgegeben haben und ein Fehler vor dem letzten Befehl aufgetreten sein, so hatten Sie bis jetzt nur die Möglichkeit, die Standardfehlerausgabe zu überprüfen, was mit jedem weiteren Befehl einer Pipe immer umständlicher wird.

Mit der Variablen `PIPESTATUS` hat dieser Zustand ein Ende. Diese Variable ist ein Array, der die Exitstati sämtlicher Befehle der zuletzt ausgeführten Pipe enthält. Der Rückgabewert des ersten Befehls in der Pipe steht in `${PIPESTATUS[0]}`, der Rückgabewert des zweiten Befehls steht in `${PIPESTATUS[1]}` usw.

Die Anzahl der Befehle, die innerhalb der Pipe ausgeführt wurden, lässt sich einfach durch ein `${#PIPESTATUS[*]}` ermitteln. Sie können diese Variable nutzen, um zu prüfen, ob und wenn ja welche Befehle in einer Pipe auf einen Fehler gefallen sind.

6.6.4 IFS

In den bisherigen Kapiteln haben wir uns schon häufiger mit dem Problem befasst, wie die Shell eine gegebene (Skript-)Zeile in Wörter aufteilt. Bis jetzt habe ich Ihnen suggeriert, dass alle Zeichen, die nicht im Bereich `a-z`, `A-Z`, `0-9` und _ liegen, automatisch zur Worttrennung führen. Dies ist nicht die ganze Wahrheit.

Schon die Ersatzmuster dürften unter dieser Betrachtungsweise eigentlich kein einzelnes Wort ergeben, sondern mehrere:

```
?apitel.txt -> "?" "apitel" "." "txt"
```

Dennoch ist dies offensichtlich nicht der Fall, denn die Shell erkennt eindeutig `?apitel.txt` und führt danach auf dieses Wort die Ersetzung für das Fragezeichen aus.

Tatsächlich ist die Aufteilung nach Worten etwas komplizierter. Zum einen führen Leerzeichen (ASCII-Wert 32), Tabulatoren (9) und Zeilenumbrüche (10) dazu, dass eine Zeile in Wörter aufgetrennt wird. Auf diese Wörter wird dann die Shellsyntax angewandt, sodass z.B. bei `${name[10]}` nicht eine Variable mit dem Namen `{name[10]}` angesprochen wird, sondern ein Eintrag (mit Index 10) der Feldvariablen `name` ausgelesen wird.

Die Beschränkung auf die ASCII-Werte 9, 10 und 32 ist nicht willkürlich und unveränderlich festgelegt, sondern in der Shellvariablen `IFS` abgelegt. Dabei steht `IFS` für *Internal Field Separator*. Ein Wort ist abgeschlossen bzw. beendet, wenn die Shell bei der Ermittlung der Wörter auf ein Zeichen trifft, welches in `IFS` abgelegt ist.

`IFS` kann natürlich auch von Ihren Skripten manipuliert werden, was zu einer vereinfachten Unterteilung von Wörtern nach *unseren* Regeln führen kann. So wäre es absolut kein Problem, in Skript 34 auf das `cut` zu verzichten und `IFS` zu nutzen. Dies ist auch eine der Übungen am Ende des Kapitels, wir wollen uns eine andere Aufgabe stellen, die unserer gemeinsamen Fähigkeiten würdig ist `:)`

Wir wollen ein Skript schreiben, welches als Parameter ein Verzeichnis erwartet, dieses Verzeichnis rekursiv nach allen Unterverzeichnissen durchsucht und diese alphabetisch sortiert ausgibt. Gleichzeitig soll das Skript die Anzahl an Verzeichnissen und die maximale Verzeichnisebene ermitteln und ausgeben.

Wie üblich legen wir uns eine Taktik zurecht, bevor wir das Skript erstellen. Also erst noch ein paar Überlegungen, bevor wir loslegen:

1. Wenn es nur darum geht, alle Unterverzeichnisse eines gegebenen Verzeichnisses zu ermitteln, so ist `du` schneller eingetippt als ein `find`.
2. Die Ausgabe von `du` umfasst eine Spalte mit Größenangaben (Blöcke oder Kilobytes) und durch einen Tabulator getrennt den Verzeichnisnamen in der zweiten Spalte.
3. Wird beim `cut` kein `-d`, aber ein `-f` angegeben, so nimmt `cut` Tabulatoren und Leerzeichen als Feldtrenner.
4. Mittels `sort` kann die Standardeingabe alphabetisch sortiert (bestehen Sie bitte nicht auf deutschen Umlauten!) und auf der Standardausgabe ausgegeben werden.

5. Die Ebene eines Verzeichnisses ist gleich der Anzahl der Verzeichnisnamen, die durch ein »/« getrennt werden.


```
# Skript 35: IFS nutzen, um Schachtelungstiefe zu ermitteln
#
# 1. Sichere den alten Wert der Trennzeichen ...
afs="$IFS"
tmpfile=/tmp/du$$
#
# 2. Schneide Größe weg und sortiere
#
du $1 2>/dev/null | cut -f2 |sort >$tmpfile
anz=`wc -l < $tmpfile`
typeset -i nr=1
typeset -i c=0
typeset -i max=0
#
# 3. Schleife für alle Zeilen der Ergebnisdatei ...
#
while [ $anz -gt 0 ] ; do
   #
   # 4. Zeile ermitteln ...
   #
   line=`head -$nr $tmpfile |tail -1`
   echo $line
   c=0
   #
   # 5. Wörter werden durch / getrennt und mit wc -w gezählt ...
   #
   IFS="/$IFS"
   c=`echo $line | wc -w `
   IFS="$afs"
  #
  # Oder alternativ
  #     for wort in $line ; do
  #       if [ "$wort" != "" ] ; then
  #         c=c+1
  #       fi
  #   done
  if [ $c -gt $max ] ; then
     max=$c
  fi
  nr=nr+1
  anz=`expr $anz - 1`
done
rm $tmpfile
# nr durch die Schleife eins zu groß
nr=nr-1
echo "$nr Verzeichnisse insgesamt, verschachtelt bis Ebene $max"
exit 0
```

Wenn Sie IFS ändern, so sichern Sie sich auf jeden Fall den alten Wert. Sie werden mit Sicherheit den alten Wert irgendwann wieder in IFS benötigen. Angenommen, Sie erweitern IFS um den Doppelpunkt und versuchen danach eine Zeichenkettenersetzung ${1:1:4}. Ergebnis: Die Shell wird sich sperren. Da der Doppelpunkt als Worttrenner aktiviert wurde, wird er von der Shell auch nur noch dazu genutzt. Sie erkennt den Doppelpunkt, trennt die Wörter und macht weiter mit dem nächsten Wort, *ohne* den Doppelpunkt noch weiter zu beachten.

Unglücklicherweise benötigt die Syntax aber bei der Ersetzung den »:«, den die Shell unter diesen Umständen nie zu sehen bekommt. Ein echtes Dilemma, wenn Sie nicht vorher den IFS auf den alten (Standard-)Wert zurückgesetzt haben.

Und noch etwas: Da IFS Zeichen enthält, welche von der Shell als »Leerzeichen« interpretiert werden, ist es nicht sehr intelligent, den alten Wert von IFS wie folgt zu speichern: afs=$IFS. Durch die bereits ausführlich beschriebene Interpretation der Shell führt diese Art der Zuweisung garantiert zum Verlust von Zeichen. Deshalb immer Anführungszeichen nutzen: afs="$IFS"

6.6.5 PS1, PS2, PS3 und PS4

Diese vier Variablen werden unter verschiedenen Umständen von der Shell als Prompt ausgegeben. PS1 ist dabei der Prompt, der im interaktiven Modus ausgegeben wird, um Sie zu einer Eingabe aufzufordern. In diesem Buch ist das immer buch@koala:/home/buch >. Wie so vieles in der Bash lässt sich auch dies nach Ihrem Geschmack anpassen. Dabei gibt es einige Zeichenketten, die eine besondere Bedeutung haben (Tabelle 6.3).

Tabelle 6.3: Sonderzeichen beim Setzen des Prompts

Zeichen	Ausgabe	Bedeutung
\a	(Piept, hupt)	Gibt ein Alarmsignal aus
\h	koala	Gibt den Rechnernamen ohne die Domain an
\u	buch	Name des aktuellen Benutzers (Username bei der Anmeldung)
\t	20:12:13	Zeit im 24-Stunden-Format
\w	~ oder ~/skript	Gibt das aktuelle Verzeichnis aus. Dabei wird das $HOME-Verzeichnis als »~« angegeben.
\W	buch	Gibt den Basisnamen ohne »/« des aktuellen Skript-Verzeichnisses aus: /home/buch wird zu buch.

Dies sind bei weitem nicht alle, reicht aber an dieser Stelle. Es kann nicht schaden, wenn Sie mal einen Blick auf die Manpage der Bash, Stichwort *Prompting* werfen.

Falls Sie nun PS1 setzen, und es tut sich nichts, dann liegt das vielleicht daran, dass PROMPT_COMMAND (ebenfalls eine Shellvariable) gesetzt ist. Ist dies der Fall, so wird vor *jeder* Ausgabe des Prompts der Inhalt der Variablen als Befehl ausgeführt.

```
buch@koala:/home/buch > PROMPT_COMMAND='PS1=`pwd` > '
/home/buch > PS1="Mein Prompt-> "
/home/buch >
```

Der PROMPT_COMMAND wird ausgeführt und setzt jedesmal PS1 auf das aktuelle Verzeichnis, gefolgt von » > «. Da hilft es auch nichts, PS1 umzudefinieren. Vor dem nächsten Prompt wird PROMPT_COMMAND ausgeführt, und PS1 erhält wieder den alten Wert.

PS2 ist der Prompt, der ausgegeben wird, wenn Sie einen Befehl eingeben, der mehr als eine Zeile umfasst, und Sie die Folgezeilen eingeben sollen. Zum Test geben Sie einfach mal `if` ein und drücken ↵. Mit Strg+C können Sie abbrechen.

PS3 ist schon interessanter. Dieser Prompt wird beim `select` ausgegeben (PS3 gibt es nicht in der sh).

PS4 ist der Prompt, der beim Tracing ausgegeben wird. Er wird genauso erweitert wie PS1. Tracing (*Nachverfolgung*) ist eine Technik zum Programmtesten. Wie das genau funktioniert, werden wir uns in Kapitel 13 ansehen (PS4 gibt es nicht in der sh).

6.6.6 HOME

Diese Variable wird von `/bin/login` gesetzt. Dieses Programm kümmert sich um die Anmeldung der Benutzer, nimmt Benutzernamen und Kennwort entgegen und überprüft diese auf Gültigkeit. Darüber hinaus setzt das Programm bei Erfolg der Anmeldung das Arbeitsverzeichnis laut Benutzerdatenbank. Dieses Verzeichnis gehört Ihnen allein, und Sie können dort nach Lust und Laune arbeiten. Dieses Verzeichnis wird als Heimatverzeichnis (oder »Home Directory«) bezeichnet, und daher stammt auch der Name der Variablen.

Die Shell nutzt den Inhalt von HOME, wenn Sie ein `cd` ohne Verzeichnis angeben. In diesem Fall wird das aktuelle Verzeichnis auf $HOME gesetzt. Auch die in Kapitel 11 behandelte Tildeexpansion berücksichtigt den Inhalt von HOME.

6.6.7 PATH

Diese Variable enthält eine Liste von einzelnen Verzeichnissen, die durch »:« getrennt sind. Die Shell nutzt diese Variable als Pfad, um Befehle zu suchen. Von links nach rechts wird der eingegebene Befehl in den entsprechenden Verzeichnissen gesucht. Wird eine Übereinstimmung gefunden, für die der Benutzer Ausführungsrechte besitzt, so wird dieser Befehl ausgeführt. Gleichlautende Befehle in einem später aufgeführten Verzeichnis der PATH-Variablen werden ignoriert.

Ein ».« im Pfad ist zwar sehr bequem, in Sachen Sicherheit aber mehr als nur bedenklich, vor allem wenn Sie als Benutzer root unterwegs sind: Ein ».« sucht die Befehle immer aus dem aktuellen Verzeichnis. Wenn im aktuellen Verzeichnis ein Befehl mit dem gleichen Namen steht wie in einem der Systemverzeichnisse (/bin und /usr/bin), so startet das Programm, welches im Pfad zuerst gefunden wird. Wenn das Programm aus dem ».«-Verzeichnis stammt, so kann es mit Ihren Berechtigungen beliebige Aktionen ausführen, die es mit den *normalen* Rechten nicht könnte. Handelt es sich um einen sehr böswilligen Zeitgenossen, kann das Schlimmste passieren.

Wenn es schon unbedingt ein ».« im Pfad sein muss, so sollte er wenigstens als letzter Eintrag im PATH stehen. Stellen Sie auch sicher, dass bei der Addition von Einträgen zum PATH kein »::« eingetragen wird, dieses wird genau wie ein ».« behandelt, womit sich die bereits angesprochenen Probleme wieder auftun.

Wenn Sie wissen wollen, welcher Befehl von der Shell wirklich aufgerufen wird, so geben Sie einfach type <befehl> ein. type durchsucht den Pfad und gibt den ersten passenden Befehl mit kompletter Verzeichnisangabe aus.

```
buch@koala:/home/buch > type cat
cat is /bin/cat
buch@koala:/home/buch >  echo $PATH
/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/openwin/bin:
/usr/lib/java/bin:/var/lib/dosemu:/usr/games/bin:/usr/games:/opt/kde/bin:.
buch@koala:/home/buch >
```

In der Bash gibt es zum type noch die Option -a. Diese bewirkt, dass alle Übereinstimmungen ausgegeben werden.

```
buch@koala:/home/buch > type -a test
test is a shell builtin
test is /usr/bin/test
buch@koala:/home/buch >
```

Die Kornshell hat diese Option leider nicht.

6.6.8 TERM

Diese Variable wird von der Shell und Programmen genutzt, um festzustellen, welcher Bildschirmtyp die Ein- und Ausgaben betreut. Wenn Ihr Bildschirm zum Beispiel ein vt100-Terminal ist, so hat es ganz andere Methoden, den Bildschirm revers zu schalten oder zu löschen als z.B. ein PC-Bildschirm. Aus diesem Grunde gibt es die `terminfo`, die diese Methoden für alle (ihr) bekannten Bildschirme verwaltet. Genauere Informationen dazu finden sich in Kapitel 4.

Unsere Skripten aus Kapitel 4, die den Bildschirm mittels `tput` manipulierten, werten diese Variable indirekt mit dem `tput` aus. Passen die Steuerungssequenzen nicht zum Bildschirm, so ist die Ausgabe total durcheinander, und es lässt sich nichts mehr erkennen.

6.6.9 Sonstige Variablen

Dies sind bei weitem nicht alle Variablen, die die Shell zur Verfügung stellt. Tabelle 6.4 enthält eine kleine Auflistung von zusätzlichen Variablen.

Tabelle 6.4: Sonstige Shellvariablen

Variable	Bedeutung
`MACHTYPE`	Eine Definition des Rechners, auf dem die Shell läuft.
`LINENO`	Enthält die Nummer der Zeile, welche die Shell gerade abarbeitet Wird LINENO mittels `unset` gelöscht, so verliert sie ihre besondere Bedeutung, sollte sie später wieder mittels `set` angelegt werden.
`OSTYPE`	Betriebssystemname
`GROUPS`	Eine Feldvariable, welche die Gruppen enthält, in welchen der aktuelle Benutzer Mitglied ist. Seit Bash 2.04 ist diese Variable zwar nicht mehr nur-lesbar (read-only), aber Zuweisungen werden ohne Fehlermeldung ignoriert. Ein `unset` hebt die Sonderbedeutung der Variablen auf, die selbst ein `set` nicht mehr restaurieren kann.
`FUNCNAME`	Der Name der aktuellen Funktion, die im Skript ausgeführt wird, genauere Informationen finden sich im Anhang A und in Kapitel 7.

Weitere Informationen und Variablen finden sich in der Manpage zur Bash unter dem Stichwort *Shell Variables*.

`LINENO` zählt, die wievielte Zeile gerade ausgeführt wird, und zwar bezogen auf den Start der aktuellen Funktion oder des aktuellen Skripts! Dies muss nicht identisch sein mit der Zeilennummer, wie sie ein Editor vergibt, da dieser vom Dateianfang zählt!

Bitte beachten Sie, dass ein großer Teil der Variablen aus Kapitel 6.5 auf die Bash beschränkt sind. Einige werden auch von der `ksh` unterstützt, aber eben nicht alle. Eine Übersicht der Variablen finden Sie in Anhang A.

6.7 Variablen indirekt

Kommen wir zum abschließenden Abschnitt dieses Kapitels. Und weil wir uns nicht mit Kleinigkeiten aufhalten wollen, haben wir Ihnen ein Bonbon ganz bis zum Schluss aufbewahrt: den indirekten Variablenzugriff.

Hm, keine Begeisterungsstürme, rein gar kein Enthusiasmus? Was, erst wenn wir Ihnen erklärt haben, was das ist?! Wir denken, wir sollten Ihnen einmal Kapitel 14 nahe legen: »Meine Autoren und ich: Warum Vertrauen wichtig ist«. `:o)` Ehrlich gesagt, Sie haben Recht, schauen wir uns also an, was Ihnen dieses Kapitel noch bieten kann.

Die Problemstellung ist ganz einfach: Wie gehe ich vor, wenn ich den Inhalt einer Variablen ausgeben möchte, deren Name in einer zweiten Variablen abgespeichert wurde?

Mittels `${!<var>}` spricht die Shell den Wert der Variablen an, deren Name in `<var>` gespeichert ist.

```
...
varind="koala"
koala="Beuteltier"
echo ${!varind}
...
```

gibt `Beuteltier` aus. Falls Sie die `sh` nutzen, die diese Art der Ersetzung nicht bietet, so können Sie den gleichen Effekt mit `eval` erreichen:

```
...
varind="koala"
koala="Beuteltier"
eval echo \$$varind
...
```

Nutzen wir dies für ein kleines Skript, das einen Buchstaben als Parameter übernimmt und alle Variablen ausgeben soll, welche mit dem Buchstaben beginnen und in der Shellumgebung eingetragen sind.


```
# Skript 36: Environmentvariablen mit
#    Namen und Inhalt ausgeben
#
line=`printenv |grep "=" | cut -d"=" -f1|sort`
```

```
for var in $line ; do
    if [ ${var:0:1} = "$1" ] ; then
        echo $var "=" ${!var}
    fi
done
exit 0
```

Skript 36 sollte demonstrieren, wie `${!var}` genutzt werden kann. Das Problem lässt sich aber auch mit einem Einzeiler lösen:


```
# Skript 36: Die kurze Version
#
printenv | grep "^$1"
exit
```

6.8 Aufgaben

Bevor wir mit den Aufgaben loslegen, möchten wir noch erwähnen, dass J. Wegmann nach der Aussage, die dieses Kapitel einleitete, wegen schlechter Leistung von seinem Trainer auf die Tribüne gesetzt wurde. Da Sie sich bis hierhin durchgebissen haben, brauchen Sie keine Angst um Ihren Platz zu haben, aber seien Sie sich der Gefahren bewusst.

1. Schreiben Sie ein Skript (`./felder.sh <Suchwort> <Dateinamen>`), das nach einem Begriff in allen Dateinamen des übergebenen Musters (z.B. `\*.sh`) sucht und die Treffer zählt. Benutzen Sie für die Dateinamen, in denen der Begriff gefunden wird, ein Feld: z.B.

   ```
   name=(`grep -c $wort $muster 2>/dev/null`)
   ```

   Geben Sie als Erstes aus, wie viele Dateien dem übergebenen Muster entsprechen und somit durchsucht werden, und anschließend, wie viele Treffer in einer Datei ermittelt wurden, wenn das Suchwort gefunden wurde.

2. Bitte ändern Sie Skript 34 so ab, dass es mit der Shellvariablen `IFS` die Einträge `1:#` anstelle von `cut -d":" -f1` unterteilt.

3. Schreiben Sie Skript 36 so um, dass es auch in der `sh` läuft. (Ziehen Sie ggf. Anhang A zu Rate.)

4. Und wieder mal etwas total Überflüssiges. Dennoch zeigt uns die Bash erneut ihre Weisheit:

```
buch@koala:/home/buch > typeset -r buch
buch@koala:/home/buch > buch="hinein schreiben"
bash: buch: readonly variable
```

6.9 Lösungen

1. Das Skript könnte folgendermaßen aussehen:


```
#!/bin/bash
#
# Nutze Arrays, um die Anzahl der Treffer
# eines Suchwortes in allen Dateien im
# übergebenen Verzeichnis zu zählen
#
# Aufruf: felder.sh <Suchwort><Dateinamen>
wort=$1
muster=$2
name=(`grep -c $wort $muster 2>/dev/null`)
gesamt=${#name[*]}
typeset -i sum=0
typeset -i ind=0
echo "$gesamt Dateien werden durchsucht..."
echo
while [ $ind -lt $gesamt ] ; do
   anz=`echo ${name[ind]} | cut -d: -f2`
   datei=`echo ${name[ind]} | cut -d: -f1`
   if [ $anz -ne 0 ] ; then
      echo "$anz mal wurde der Begriff in $datei gefunden"
   fi
   ind=ind+1
   sum=sum+anz
done
echo
echo "Insgesamt wurde der Begriff $sum mal gefunden"
exit 0
```

Die Umlenkung `2>/dev/null` der Fehler von `grep` bei der Zuweisung des Feldes `name` ist eingebaut, damit nicht unerwünschte Fehler erzeugt werden durch Unterverzeichnisse.

2. Um cut durch die Manipulation der Variablen IFS auszutauschen, sollte zunächst der Wert von IFS festgehalten werden:


```
# Skript 34: Ein simpler Sprücheklopfer
# Dieses Skript versucht, eine eigene Version
#  von fortune auf die Beine zu stellen
#
# 1. Die Anzahl an Sprüchen ermitteln
afs="$IFS"
typeset -i anz=`grep '^#' spruch.txt | wc -l`
anz=anz-1
# 2. Ein Spruch per Zufall ermitteln, nr enthält
#    Arrayindex der Startzeile und ende den Index der
#    Endezeile
typeset -i nr=$RANDOM
nr=nr%anz
typeset -i ende=nr+1
# 3. Zeilennummern mittels grep ermitteln
#    in $zeile steht "1:# 4:# 6:# 8:# 10:# 13:# 16:# 19:#"
#    Das weisen wir jetzt dem Array name zu.
#    name[0]="1:#"  name[1]="4:#" usw.
#   ^#  sucht nur am Zeilenanfang
zeile=`grep -n '^#' spruch.txt`
name=($zeile)
stpaar=${name[$nr]}
# 4. Aus den Arrayeinträgen nur die Zahlen berücksichtigen
#    und Start/Endezeilen berechnen

#
#Lösung über IFS statt cut
#
IFS=":$IFS"
#
# Belegt einen Array st mit Feldern [0] und [1]
# in [0] --> Zeilennr
#    [1] --> "#" (nicht gebraucht)
#
st=(`echo $stpaar`)
IFS="$afs"  #IFS wieder zurücksetzen

enorg=${name[$ende]}
typeset -i en=`echo $enorg|cut -d":" -f1`
en=en-1
#
# $st gibt ${st[0]} zurück
#
az=`expr $en - ${st[0]}`
# 5. Zeilenbereich ausschneiden
head -$en spruch.txt|tail -$az
exit 0
```

3. Das Skript 36 muss an zwei Stellen leicht abgeändert werden, damit es auch unter sh ablaufen kann:


```
#!/bin/sh
# Skript 36: Environmentvariablen mit
#    Namen und Inhalt ausgeben
#
line=`printenv |grep "=" | cut -d"=" -f1|sort`
for var in $line ; do
    if [ `echo $var | cut -c1` = "$1" ] ; then
        echo $var "=" `eval "\$"$var`
    fi
done
exit 0
```

Kapitel 7

Funktionen

»Lernen, ohne zu denken, ist eitel; denken, ohne zu lernen, ist gefährlich« – Konfuzius

… die Tatsache aber, dass Sie es bis zu diesem Kapitel geschafft haben, deutet darauf hin, dass Sie im schlimmsten Falle eitel sind, und wenn Sie zumindest einige der gestellten Aufgaben gelöst haben, entbehrt auch diese Unterstellung jedweder Grundlage.

An dieser Stelle haben Sie bereits etwa die Hälfte des Buches durchgearbeitet. Damit sollten Sie in der Lage sein, schon einige Shellskripten selbst zu schreiben. Aber eigene Skripten haben eine Eigenart, die nicht zu ändern ist: Sie werden grundsätzlich länger als ursprünglich vorgesehen, und mit jeder Zeile mehr verfällt Ihr Skript der dunklen Seite der Macht: der Unübersichtlichkeit. Soll der klassische Ausspruch »*Live long and prosper*« als der Leitfaden Ihres Skripts verstanden werden, brauchen Sie Methoden und Funktionen, die eine gewisse Übersichtlichkeit sicherstellen.

Funktionen sind dabei das Stichwort für dieses Kapitel. Denn neben den Befehlen, die Ihnen die Shell zur Verfügung stellt, bietet sie Ihnen auch die Möglichkeit, eigene Funktionen zu definieren.

Dabei stellen Funktionen eine Unterroutine dar, die eine beliebige Anzahl und Art von Shellbefehlen enthält. Diese Unterroutine bekommt dabei einen Namen, mit dem die Funktion dann an anderer Stelle innerhalb Ihres Skripts aufgerufen werden kann. Dieser Name sollte möglichst nicht mit Namen von Befehlen oder Shellbuiltins kollidieren. Hilfreich ist an dieser Stelle, den Namen

der Funktion mit einem Präfix zu versehen: `CW_` z.B. für Funktionen im CW-Commander.

Wann Sie Funktionen einsetzen sollten, lässt sich nicht an eindeutige Bedingungen knüpfen, sondern hängt von den Skripten und Ihrem persönlichen Stil ab. Dennoch hier einige allgemeine Vorschläge, wann Sie Funktionen nutzen sollten:

- Teilen Sie Ihr Skript in sinnvolle Blöcke auf (Eingabe, Eingabeprüfung, Ausgabe als Beispiel). Ist ein solcher Block wesentlich länger als zwei Bildschirmseiten, sollten Sie den Block als Funktion anlegen. Damit wird Ihre Funktion übersichtlicher, da Sie nicht so lange blättern müssen, um die Übersicht über die Funktion zu bekommen.
- Wenn sich bestimmte Anweisungen (die schon drei bis vier Zeilen oder länger sein sollten) mehrfach an verschiedenen Stellen wiederholen, dann sollte man daraus eine Funktion erstellen.
- Der Teil des Skripts, in dem die Logik des Skripts programmiert wird, sollte nicht länger als zwei Bildschirmseiten sein und idealerweise nur aus Schleifen, Funktionsaufrufen und eventuell noch einigen wenigen(!) `if`-Abfragen bestehen. Sie verstehen die Logik viel schneller, wenn die Steuerungslogik übersichtlich ist.
- Werden bestimmte Funktionalitäten mehrfach innerhalb des Skripts benötigt, so empfiehlt es sich, diese in Funktionen abzulegen. Kleinere Unterschiede können durch die der Funktion übergebenen Parameter angepasst werden.

Nachdem Sie jetzt eine Vorstellung über das *Warum* und *Wann* haben, wird es Zeit, dass wir zum *Wie* kommen.

7.1 Gruppenbefehl


```
{ <Befehlsliste> ; }
```

Eine Funktion besteht aus einem Kopf (zu dem wir in Kapitel 7.2 kommen) und einem Befehlsblock. Befehlsblöcke werden in geschweiften Klammern eingefasst und enthalten eine beliebige Anzahl von Befehlen, die durch ein Semikolon oder einen Zeilenumbruch abgeschlossen werden. Diese Art von Befehlen wird als *Gruppenbefehl* (engl. *group command*) bezeichnet.

Der Exitstatus einer Gruppe ist identisch mit dem letzten Befehl, der in der Gruppe ausgeführt wurde. Er verhält sich also ähnlich wie der Exitstatus einer Pipe.

Gruppen können Sie überall dort einsetzen, wo auch normale Shellbefehle aufgerufen werden können.

Die Leerzeichen sind wichtig, da es sonst zu Brace Extensions kommen kann.

7.2 Funktionen

Definition

```
function <name>() { <befehlsliste> ; }
```

oder

```
<name>() { <befehlsliste> ; }
```

Aufruf:

```
<name>
<name> <par1> ...
```

Eine Funktion enthält eine Serie von Shellbefehlen, die zu einem späteren Zeitpunkt vom Skript aufgerufen werden. Wird eine Funktion aufgerufen, so wird sie in der originalen Shellumgebung ausgeführt und nicht in einer Kopie. Dies führt dazu, dass sämtliche von der Funktion vorgenommenen Änderungen in der Umgebung nach Beendigung der Funktion erhalten bleiben.

Parameter werden beim Aufruf durch Leerzeichen getrennt hinter dem Funktionsnamen angegeben. Wird die Funktion ausgeführt, so werden die übergebenen Parameter zu den Positionsparametern innerhalb der Funktion. `$#`, `$1`, `$2` usw. werden angepasst. `$0` gibt weiterhin den Namen des Skripts zurück und nicht den Funktionsnamen. Wird die Funktion verlassen, so werden die Positionsparameter wieder so hergestellt, wie sie vor dem Aufruf der Funktion belegt waren. Dies sind die einzigen Unterschiede zwischen Funktionen und dem aufrufenden Skriptteil, zumindestens was die Shellumgebung betrifft.

Leider stellen nicht alle Shells die Positionsparameter nach dem Aufruf einer Shellfunktion wieder her. Dies gilt z.B. für `/bin/sh` unter HP-UX. Falls Sie mit solchen Shells arbeiten, hilft folgendes:

```
...
 # 1. Alte Parameter sichern
 gvArgs=$@
 # 2. Funktion aufrufen
 NE_Funktion "Christa" "mag" "Delphine"
 # 3. und alte Parameter wieder setzen
 set -- $gvArgs
...
```

Wird der Befehl `return` in einer Funktion ausgeführt, so wird die Funktion beendet, die Positionsparameter werden wieder hergestellt, so wie sie vor dem Aufruf waren, und das Skript fährt mit der Ausführung des nächsten Befehls nach dem Aufruf fort.

Funktionen können sich rekursiv aufrufen. Eine Beschränkung hinsichtlich der Anzahl der Rekursionsebenen ist nicht vorhanden (außer dem Speicher).

Rekursion ist eine Umschreibung für Selbstaufruf. Nehmen wir als Beispiel mal die Fakultätsberechnung. Die Fakultät berechnet sich für eine positive Ganzzahl n mit n größer 0 durch eine Multiplikationskette n * (n-1) * (n-2) ... * 2 * 1. Also ist die Fakultät von 5 -> 5 * 4 * 3 * 2 * 1 = 120. Eine Möglichkeit, die Fakultät zu berechnen, sieht also so aus:


```
# Skript 37: Fakultät, iterativ
#
# Nimmt den Parameter $1 und
# berechnet dazu die Fakultät
typeset -i zahl=${1:-1}
if [ $zahl -lt 1 ] ; then
   echo "$0: Fakultät nur für Ganzzahlen größer 0" >&2
   exit 1
fi
typeset -i fak=1
while [ $zahl -gt 1 ] ; do
  fak=fak*zahl
  zahl=zahl-1
done
echo "Fak($1)=$fak"
exit 0
```

Diese Methode wiederholt die Multiplikation mit dem vorherigen Ergebnis so lange, wie `zahl` größer als 1 ist. Iteration ist ein vornehmes Wort für Wiederholung, weshalb diese Methode als iterativ bezeichnet wird.

Die Fakultät lässt sich aber auch anders berechnen. Wenn Sie sich Tabelle 7.1 einmal genauer anschauen, so stellen Sie fest, dass die Fakultät von n gleich n mal der Fakultät von n-1 entspricht. Die Fakultät für ein n kleiner oder gleich 1 ist per Definition gleich 1.

Tabelle 7.1: Fakultätsberechnung

Parameter	Fakultät	Iterativ	Rekursiv
1	1	1	1
2	2	2*1	2*Fak(1)
3	6	3*2*1	3*Fak(2)
4	24	4*3*2*1	4*Fak(3)

Versuchen wir uns nun gleich einmal an der rekursiven Version. Diese nutzt eine Funktion `Fak`, um die Fakultät zu berechnen. Wichtig bei jeder Rekursion ist die Tatsache, dass die Rekursion eine Abbruchbedingung hat. Fehlt diese, so kommt die Funktion nicht mehr zum Ende, da sie sich bis in alle Ewigkeit selbst aufruft.


```
# Skript 38: Fakultät, rekursiv
#
# Nimmt den Parameter $1 und
# berechnet dazu die Fakultät
#
# 1. Hier wird die Funktion definiert, aber noch nicht ausgeführt
#
function Fak()
{
  if [ $1 -lt 1 ] ; then
     fak=1
    else
     Fak $(($1-1))
     fak=$1*fak
  fi
}
#
# 2. Hier startet das Skript nach dem Aufruf
#
typeset -i zahl=${1:-1}
typeset -i fak=1
if [ $zahl -lt 1 ] ; then
   echo "Fehler: Fakultät nur für Ganzzahlen größer 0" >&2
   exit 1
fi
#
# 3. Erst hier wird die Funktion aufgerufen. Jetzt läuft das
#    Skript weiter in der Funktion mit typeset -i zahl=$1
Fak $zahl
echo "Fak($1)=$fak"
exit 0
```

Möglicherweise sind Sie etwas irritiert, warum wir so sehr auf der Fakultät herumreiten, dass wir gleich zwei verschiedene Versionen für eine im Zusammenhang mit Skriptprogrammierung wenig sinnvolle Funktion verbreiten. Nun, es gibt einige gute Gründe dafür:

1. Es zeigt in einer einfachen Art und Weise, wie Funktionen geschrieben werden können.
2. Es zeigt deutlich den Unterschied zwischen Rekursion und Iteration. Genau wie »Hallo Welt« *das* erste Programm ist, welches in einer neu er-

lernten Sprache erstellt wird, so wird der Unterschied zwischen Rekursion und Iteration anhand der Fakultät erläutert.

3. Vielleicht haben Sie ja bemerkt, dass Christa Wieskotten ein Dipl. Math. hat. Was glauben Sie, was sie mit ihrem Co-Autor macht, wenn wir nicht mindestens ein mathematisches Skript in dieses Buch stellen? Kleiner Tipp: Schauen Sie sich mal das Motto von Kapitel 4 an.

Wir können also feststellen, dass die Punkte 1 und 2 wichtig für Ihre Erfahrungen bei der Shellprogrammierung sind, während Punkt 3 naturgemäß für den männlichen Teil des Autorenteams unglaublich wichtig ist.

Die meisten Hochsprachen ermöglichen es, dass Funktionen ein Ergebnis eines bestimmten Typs zurückgeben (müssen). Dies ist so leider nicht bei der Skriptprogrammierung möglich.

Kommen wir nun nach langer Zeit mal wieder zum CW-Commander-Skript zurück. Als erste Verbesserung sollte die Box mithilfe einer Funktion ausgegeben werden. Dabei sollten vier Parameter verlangt werden:

- Die Startkoordinate (Y,X) sowie die Breite und Höhe. Wird der Funktion noch ein fünfter Parameter übergeben, so wird dieser Text zentriert über der Box ausgegeben.
- Außerdem erstellen wir Funktionen, welche die Bildschirmattribute (revers, hell, normal usw.) setzen. Dies ist zwar nicht notwendig, ist aber übersichtlicher und verständlicher als die kryptischen `tput`-Befehle.
- Das ist aber immer noch nicht alles. Als Letztes erstellen wir eine Druckfunktion, die als Parameter die Koordinate (Y,X) und den zu druckenden Text erwartet. Mit diesen Funktionen bauen wir dann nicht mehr nur ein, sondern zwei gleich große Fenster auf. Im linken Fenster geben wir wie bisher die Dateinamen aus, und ein Blättern sollte auch wieder möglich sein. Im rechten Fenster passiert vorläufig nichts.

Noch ein Hinweis, bevor wir endgültig zum Skript kommen. Sie hatten das ursprüngliche Skript ja bereits in verschiedenen Aufgaben verbessert. Diese Verbesserungen sind in dieser Version unter den Tisch gefallen. Es werden noch einige Versionen dieses Skripts mit zusätzlicher Funktionalität im Laufe der restlichen Kapitel auftauchen. Wenn Sie auf Ihre Änderungen nicht verzichten möchten, so können Sie einerseits die Änderungen in meinen Versionen vornehmen. Sinnvoller wäre aber der andere Weg, meine Änderungen in Ihr Skript einzubauen.

Wie auch immer, die letzte Version dieses Skripts finden Sie in Anhang B. Dort können Sie dann alle Änderungen einbauen, ohne Gefahr zu laufen, dass wir in einer neuen Version die Verbesserungen aus den Aufgaben mal wieder ignorieren.


```
# Skript 39:
#
#  Die neueste Version des CW-Commanders
function CW_SetBold ()
{  # Setzt den Modus Bold
   tput bold
}
function CW_SetNormal ()
{  # Setzt den Modus zurück
   tput sgr0
}
function CW_SetRev ()
{  # Setzt Inversmodus
   tput rev
}
function CW_Print  ()
{  # Setzt an $1, $2 den Text $3
  tput cup $2 $1
  echo -n "$3"
}
function CW_Box ()
{  # Setzt das Fenster an Pos $1, $2 mit $3
   # Zeichen Breite und $4 Zeilen Höhe
   # Falls $5 gesetzt ist, so wird dies
   # als Titel der Box ausgegeben
   typeset -i anz=2
   typeset -i xp=$1
   typeset -i yp=$2
   titel=${5:-""}
   bt="+"
   while [ $anz -lt $3 ] ; do
     bt="$bt-"
     anz=anz+1
   done
   bt="$bt+"
   anz=1
   CW_Print $xp $yp $bt
   yp=yp+1
   leze=`printf "!%$(($3-2)).$(($3-2))s!"`
   while [ $anz -lt $(($4-1)) ] ; do
      CW_Print $xp $yp "$leze"
      anz=anz+1
      yp=yp+1
   done
   CW_Print $xp $yp $bt
   if [ -n "$titel" ] ; then
      typeset -i len=${#titel}
      if [ $len -gt $3 ] ; then
         typeset -i rest=len-$3+6
         titel="..."`echo "$titel" | cut -c${rest}-`
         len=len-rest+4
      fi
```

```
      CW_SetBold
      xp=$3-len
      xp=xp/2+$1
      CW_Print $xp $2 "$titel"
      CW_SetNormal
   fi
}

Mark=":"
CW_Clear
verz=${1:-"."}
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp  w = PgDn"
#
# Zeilen-Offset setzen, Dateien ermitteln,
# Anzahl ermitteln
#
offset=1
tmpfile="/tmp/cwc$$.tmp"
anz=`ls $verz|tee $tmpfile | wc -l`
akt=1
until [ "$ein" = "q" ] ; do
  # Ausgabe der Dateien
  i=0
  akt=0
  while [ $i -lt 18 -a $akt -le $anz ] ; do
   akt=`expr $i + $offset`
   tput cup `expr $i + 2` 1
   zeile=`sed -n -e "${akt}p" $tmpfile`
   revers=`echo "$Mark" | grep ":$zeile:" `
   if [ -n "$revers" ] ; then
       CW_SetRev
   fi
   printf "%-38.38s" $zeile
   CW_SetNormal
   i=`expr $i + 1`
  done
  while [ $i -lt 18 ] ; do
    tput cup `expr $i + 2` 1
    i=`expr $i + 1`
    printf "%38.38s" " "
  done
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
           offset=`expr $offset + 17` ;
```

```
          if [ $offset -gt $anz ] ; then
             offset=`expr $offset - 17`
          fi ;;
     "z" | "Z")
          if [ $offset -gt 1 ] ; then
             offset=`expr $offset - 17` ;
          fi ;;
   *) ein="" ;;
    esac
  done
done
exit 0
```

Diese Version enthält schon den Code für das Invertieren von markierten Einträgen. Die Markierung selbst nehmen wir uns in einer der Aufgaben am Kapitelende vor.

7.2.1 return

```
return [<intwert>]
```

Funktionen sind Unterroutinen, die Ergebnisse zurückgeben können. Diese Eigenschaft wollen wir uns jetzt auch zu Nutze machen. Dazu benötigen wir den Befehl `return`.

Innerhalb einer Funktion aufgerufen, beendet er diese Funktion mit dem Exitstatus (Rückgabewert) von `<intwert>`. Ist `<intwert>` nicht angegeben, so wird die Funktion beendet, und der Rückgabewert ist identisch mit dem Rückgabewert des zuletzt in der Funktion ausgeführten Befehls.

Somit könnte eine simple Funktion, die den größeren von zwei übergebenen Werten zurückgibt, inklusive Aufruf wie folgt aussehen:

```
...
function Max()
{ if [ $1 -gt $2 ] ; then
     return $1
   else
     return $2
  fi
}
...
Max 12 14
echo "Max(12,14)=$?"
...
```

7.3 Lokale Variablen


```
local <var>
local <var>=<wert>
```

Bis jetzt sind alle Variablen in allen Funktionen und im Hauptprogramm bekannt. Solche Variablen werden überraschend :-) als *globale* Variablen bezeichnet, da sie global an allen Stellen des Programms bekannt sind. Nun führen Funktionen in der Regel Aktionen aus, für die sie ebenfalls Variablen brauchen. Diese Variablen werden außerhalb der Funktionen meist nie mehr benötigt. Deshalb wäre es sinnvoll, sie nur in der Funktion zu kennen und am Ende der Funktionen automatisch wieder zu löschen.

Wenn Sie eine Variable mittels des Befehls `local` innerhalb einer Funktion definieren, so ist diese Variable nur für alle Befehle in der Funktion und in allen Befehlen von ihr aufgerufener Funktionen bekannt. Mit der zweiten Variante weisen Sie der Variablen `<var>` sofort einen `<wert>` zu.

Falls Sie ein `typeset` innerhalb einer Funktion einsetzen, wird die betroffene Variable ebenfalls als lokal definiert. Ein `typeset` in einer Funktion verhält sich also genau wie der Befehl `local`. Genauere Informationen zum Thema `typeset` finden Sie in Kapitel 6.

Ein `local` ist nur in einer Funktion zulässig. Wird dies nicht beachtet, so wird eine Fehlermeldung ausgegeben, und der Rückgabewert von `local` ist ungleich `0`. Gleiches gilt auch für den Fall, dass der Variablenname `<var>` nicht zulässig ist. Hat alles geklappt, ist der Exitstatus `0`. Wird `local` ohne Parameter aufgerufen, so gibt er eine Liste der lokalen Variablen auf die Standardausgabe aus.

Wird eine lokale Variable definiert, die den gleichen Namen hat, wie eine globale Variable, so überdeckt die lokale Variable die globale. Zugriffe erfolgen immer auf die lokale Variable. Dabei wird die globale Variable aber nicht gelöscht, sie steht am Ende der Funktion wieder zur Verfügung, da zu diesem Zeitpunkt die lokale Variable wieder gelöscht wird.

Nachdem Bettina Nscho-Tschi und ihr treuer Gefährte LeserIn Halef Omar ben Hadschi Abul Abbas die große Wüste der Theorie durchschritten hatten, kamen sie an der Oase Prack Sihs an. Gerade als sie ihren Durst stillen wollten, trat ihnen eine wunderschöne Blume entgegen und sprach: »Um euch an diesem Wasser zu laben, müsst ihr erst eine Aufgabe lösen, denn sehet, mein Name ist Kries Da Dipel Maht Wies Cod Den, und ich bin die Hüterin dieser Oase.« Halef verzweifelte »Ja mussihbe, ia za'al – Oh Unglück, oh Verdruss.« Doch Bettina Nscho-Tschi schulterte ihre 102-tastige Henrytastatur und

sprach: »Wohl denn, Hüterin, nennt mir eure Aufgabe.« Und so sprach Kries: »Da …«

Kries' Rede war lang und blumig, aber da wir hier keinen Reisebericht schreiben wollen, fasse ich die Aufgabe einmal kurz zusammen:

- Die Eingabeschleife mit den Abfragen, welche Aktionen auszulösen sind, soll in eine Funktion ausgelagert werden.
- Gleiches gilt für die Ausgabe des Verzeichnisses in den Boxen. Übergeben wird der Funktion die Position der ersten Zeile (`x,y`) auf dem Bildschirm, die aktuelle Zeile und die Datei, in der die Informationen abgelegt sind. Zusätzlich sollen Berechtigungen und Dateigrößen ausgegeben werden.
- Ein Durchblättern des Verzeichnisinhalts sollte auch zeilenweise möglich sein. Die aktuelle Zeile sollte hell hervorgehoben werden. Mit den Tasten `l` und `n` kann die letzte bzw. nächste Zeile angesprungen werden.

Während die ersten beiden Punkte nur alter Wein in neuen Schläuchen sind, müssen wir uns über Punkt drei doch ein wenig den Kopf zerbrechen. Klar sollte sein, dass wir eine Variable benötigen, in der die Nummer der aktuellen Zeile abgespeichert wird. Mit der Eingabe `n` wird diese Nummer um eins erhöht, durch `l` um eins erniedrigt. Dabei sollte beachtet werden, dass die Nummer nie größer als die Anzahl Zeilen und nie kleiner als 1 werden darf.

Das nächste Problem ist die Ausgabe der aktuellen Zeile in heller Schrift. Während die Zeilennummer von 1 bis Zeilenanzahl geht, haben wir für die Ausgabe nur den `offset` (der immer die Nummer der ersten auf der Seite ausgegebenen Zeile darstellt) und einen Zähler von 0 bis Zeilenanzahl pro Seite.

Demnach gilt: Ist die aktuelle Zeilennummer gleich `offset` + Zähler, dann ist die Zeile hell auszugeben. Bleibt noch ein Problem: Es ist schon wünschenswert, dass die aktuelle Zeile immer angezeigt wird. Das heißt, springt die Zeile oben oder unten aus der aktuell angezeigten Seite raus, so muss die Seite entsprechend neu ausgegeben werden. In einer für den Computer verständlichen Version bedeutet dies, ist die aktuelle Zeile kleiner als `offset` oder größer `offset` + Zeilenanzahl pro Seite, dann ist der Offset anzupassen und die Seite neu auszugeben.

Schauen wir uns dies einmal als Skript an:


```
# Skript 40:
#
# CW-Commander mit Zeilenauswahl
#
# Achtung! Dieses Skript ist nicht komplett ausgedruckt,
# fehlende Routinen entnehmen Sie vorherigen Skripten
#
```

```
# Die Eingabe wurde in eine eigene Funktion gepackt
# Das Blättern geht jetzt auch zeilenweise
#  Dazu wurde CW_Eingabe und CW_PrintDir angepasst
...
function CW_Eingabe()
{
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
               offset=offset+17
               if [ $offset -gt $anz ] ; then
                  offset=offset-17
               else
                  # Seite geblättert, akt. Zeile nachziehen
                  zakt=zakt+17
                  if [ $zakt -gt $anz ] ; then
                     # akt. Zeile ist größer als Anzahl Zeilen,
                     # also auf Zeilenanzahl setzen
                     zakt=anz
                  fi
               fi
               ;;
      "z" | "Z")
               if [ $offset -gt 2 ] ; then
                  offset=offset-17 ;
                  zakt=zakt-17
               fi
               ;;
      "n" | "N") if [ $zakt -lt $anz ] ; then
               zakt=zakt+1
               if [ $zakt -gt $((offset+17)) ] ; then
                  offset=offset+17
                fi
             fi ;;
      "l" | "L") if [ $zakt -gt 2 ] ; then
               zakt=zakt-1
               if [ $zakt -lt $offset ] ; then
                  offset=offset-17
               fi
            fi ;;
       *) ein="" ;;
     esac
  done
}
```

```
function CW_PrintDir()
{
  # Ausgabe der Dateien
  typeset -i xp=$1
  typeset -i hoehe=$2
  ausdatei=$3
  typeset -i i=0
  typeset -i akt=0
  while [ $i -lt $hoehe -a $akt -le $anz ] ; do
   akt=i+offset
   zeile=`head -$akt $ausdatei | tail -1`
   set -- $zeile
   revers=`echo "$Mark" | grep ":$9:" `
   if [ -n "$revers" ] ; then
      CW_SetRev
   fi
   if [ $zakt -eq $akt ] ; then
      CW_SetBold
   fi
   local datei=`echo $9 | cut -c-15`
   local text="`printf "$1|%9i | %-15s" $5 $datei`"
   CW_Print $xp $((i+2)) "$text"
   CW_SetNormal
   i=i+1
  done
  while [ $i -lt 18 ] ; do
    CW_Print $xp $((i + 2)) "`printf \"%10s|%10s|%16s\"`"
    i=i+1
  done
}
Mark=":"
tput clear
verz=${1:-"`pwd`"}
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp w = PgDn  l = CuUp  n = CuDn"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=2
tmpfile="/tmp/cwc$$.tmp"
cd $verz
typeset -i anz=`ls -ld .* * | tee $tmpfile | wc -l | cut -c-7`
typeset -i zakt=2    # Zähler akt. Zeile
until [ "$ein" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
exit 0
```

Da wir jetzt alle Informationen über eine Datei einlesen wollen, benötigen wir ein `ls -l`. Dies hat jedoch den Nachteil, dass zunächst ein »`total 123`« ausgegeben wird. Dazu kommt noch die Tatsache, dass Inhalte von Unterverzeichnissen ebenfalls ausgegeben werden. Dies kann unterbunden werden, indem `ls` zusätzlich mit der Option `-d` aufgerufen wird. In diesem Fall gibt `ls` nur den Eintrag für das Verzeichnis aus, nicht aber dessen Inhalt. Um alle Dateien auszugeben, inklusive der versteckten, können wir `ls` entweder die Option `-a` oder `-A` übergeben. Der Unterschied liegt darin, dass `-a` auch die Einträge »`.`« und »`..`« ausgibt, die bei `-A` unterdrückt werden.

```
buch@koala:/home/buch > ls -ad ..*|pg
-rw-r--r--   1 buch     users          46 May  7  1996 .Xmodmap
drwxr-xr-x   3 buch     users        1024 Mar  6 12:13 .kde
-rw-r--r--   1 buch     users         492 Aug  8  1997 .profile
-rw-r--r--   1 buch     users          99 May 16  1997 .susephone
-rw-r--r--   1 buch     root         4178 Feb  6 17:42 einleitung.txt
-rwxr-xr-x   1 buch     users        5047 Feb 21 01:16 gk2
-rw-r--r--   1 buch     users        1322 Feb 21 01:16 gk2.c
-rw-r--r--   1 buch     users       22869 Mar  3 21:49 kapitel1.txt
-rwxr-xr-x   1 buch     users       33436 Mar  8 21:18 kapitel2.txt
-rwxr-xr-x   1 buch     users       41562 Mar  8 21:18 kapitel3.txt
-rw-r--r--   1 buch     users       30048 Feb 23 20:55 kapitel4.txt
-rw-r--r--   1 buch     users       31716 Mar  5 22:53 kapitel5.txt
-rw-r--r--   1 buch     users       41530 Mar  6 12:36 kapitel6.txt
-rw-r--r--   1 buch     users       20211 Mar  7 14:50 kapitel7.txt
-rw-r--r--   1 buch     users        3113 Mar  8 20:12 toc.txt
buch@koala:/home/buch >
```

Der Punkt »`.`« steht in Unix für das aktuelle Verzeichnis und »`..`« für das übergeordnete Verzeichnis, welche für uns an dieser Stelle nicht nutzbringend sind.

Hinsichtlich der aktuellen Zeile ist noch anzumerken, dass diese immer auf der angezeigten Seite sein sollte und somit beim seitenweisen Blättern angepasst werden muss. Beim Zurückblättern ist dies einfach: Da `zakt` immer größer oder gleich `offset` ist, kann `zakt` ebenfalls um eine Seite reduziert werden, wenn `offset` reduziert wird. Beim Vorwärtsblättern muss `zakt` ebenfalls erhöht werden, kann jedoch größer werden als die Anzahl Zeilen. In diesem Fall muss `zakt` auf die Zeilenanzahl gesetzt werden.

Die Tatsache, dass in einer Zeile mehr als der reine Dateiname enthalten ist, erzwingt die Änderung der Markierungsabfrage. Zunächst ermitteln wir nur den Dateinamen und `grep`'pen ihn dann erst in `Mark`. Diese Routine ist momentan noch nutzlos, da wir keine Routinen haben, die eine Markierung durchführen.

Haben Sie sich übrigens einmal die Hauptroutine angesehen? Sie passt auf eine Bildschirmseite! Dies ist doch wesentlich übersichtlicher als die alte Version, die mehr als drei Bildschirmseiten benötigte.

In den restlichen Kapiteln werden wir diese Version als Basis für unsere Verbesserungen nehmen. Alle neuen Versionen werden nur die Änderungen zu dieser Version aufzeigen. Unverändert gebliebene Routinen werden nicht mehr aufgeführt, um Platz zu sparen. Die endgültige Version des CW-Commanders (zumindestens, was den Rahmen dieses Buches betrifft) finden Sie noch einmal als komplettes Listing in Anhang B.

Schauen wir uns dieses Vorgehen anhand eines letzten Beispiels an. Dazu fügen wir eine Abfrage auf die ↵-Taste hinzu. Drückt der Benutzer ↵, während die aktuelle Zeile auf einem Verzeichnisnamen steht, so wechselt unser Skript in dieses Verzeichnis und gibt dessen Inhalt aus.


```
# Skript 41:
#
# CW-Commander mit Verzeichniswechsel
#
# Achtung! Dieses Skript ist nicht komplett ausgedruckt,
# fehlende Routinen entnehmen Sie vorherigen Skripten
...
function CW_Eingabe()
 {
   gvEingabe=""
   local line=""
   until [ -n "$gvEingabe" ] ; do
      tput cup 22 0
      read -p "Eingabe:" gvEingabe
      case $gvEingabe in
       "q" | "Q") gvEingabe="q";;
       "w" | "W")
           offset=offset+17
           if [ $offset -gt $anz ] ; then
             offset=offset-17
           else
              zakt=zakt+17
              if [ $zakt -ge $anz ] ; then
                 zakt=anz
              fi
           fi
           ;;
       "z" | "Z")
           if [ $offset -gt 17 ] ; then
              offset=offset-17 ;
              zakt=zakt-17
           fi
           ;;
       "n" | "N") if [ $zakt -lt $anz ] ; then
           zakt=zakt+1
           if [ $zakt -gt $((offset+17)) ] ; then
              offset=offset+17
           fi
       fi ;;
```

```
      "l" | "L") if [ $zakt -gt 1 ] ; then
         zakt=zakt-1
         if [ $zakt -lt $offset ] ; then
            offset=offset-17
         fi
      fi ;;
       "")   line=`sed -n -e "${zakt}p" $tmpfile`
             set -- $line
             local ch=`echo  ${1:0:1}`
             if [ "$ch" = "d" ] ; then
                gvEingabe="CR"
                CW_ReadDir "$9";
             else
                CW_NormalFile "$9"
             fi ;;
       *)    gvEingabe="" ;;
     esac
   done
 }
function CW_ReadDir ()
 {
  offset=1
  zakt=1
  cd $1
  verz=`pwd`
  anz=`ls -lAd .. *|tee $tmpfile | wc -l | cut -c-7`
  CW_Box 0 1 40 20 "$verz"
 }
function CW_Clear ()
{
 # Löscht den kompletten Schirm
 tput clear
}

 Mark=":"
 CW_Clear
 verz=${1:-"`pwd`"}
 CW_Box 0 1 40 20 "$verz"
 CW_Box 40 1 40 20
 CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn m = Markieren"
 #
 # Zeilenoffset setzen, Dateien ermitteln, Anzahl ermitteln
 #
 typeset -i offset=1
 tmpfile="/tmp/cwc$$.tmp"
 rm -f $tmpfile
 cd $verz
 typeset -i anz=`ls -lAd * .. | tee $tmpfile | wc -l`
 typeset -i zakt=1   # Zähler akt. Zeile
```

```
until [ "$gvEingabe" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
exit 0
```

In CW_Eingabe setzen wir im Falle von "" die Variable ein auf einen Pseudowert (ungleich "" und q), wenn wir eine neues Verzeichnis lesen. Damit erreichen wir, dass die Eingabeschleife in CW_Eingabe verlassen wird und gleichzeitig die Schleife im Hauptteil des Skripts nicht abbricht.

7.4 FUNCNAME

Ab Version 2.04 bietet die Bash eine Variable namens FUNCNAME an. Diese wird innerhalb einer Funktion automatisch mit dem Namen der gerade ausgeführten Funktion beschickt. Zuweisungen auf diese Variable werden stillschweigend (sprich ohne Fehlermeldung) ignoriert. Im Hauptteil des Skripts ist diese Variable leer (""). Wird diese Variable mittels unset gelöscht, dann verliert sie ihre besondere Bedeutung, auch wenn sie danach wieder angelegt werden sollte.

Und damit der Stoff nicht so trocken ist, hier ein Beispiel mit einem leicht abgewandelten Skript 38:

```
# Skript 38: Fakultät, rekursiv
#
# Nimmt den Parameter $1 und
# berechnet dazu die Fakultät. Demonstriert FUNCNAME
#
# 1. Hier wird die Funktion definiert, aber noch nicht ausgeführt
#
function Fak()
{
  echo $FUNCNAME
  if [ $1 -lt 1 ] ; then
     fak=1
    else
     Fak $(($1-1))
     fak=$1*fak
  fi
}

#
# 2. Hier startet das Skript nach dem Aufruf
#
echo "Start: $FUNCNAME"
```

```
typeset -i zahl=${1:-1}
typeset -i fak=1
if [ $zahl -lt 1 ] ; then
   echo "Fehler: Fakultät nur für Ganzzahlen größer 0" >&2
   exit 1
fi
#
# 3. Erst hier wird die Funktion aufgerufen. Jetzt läuft das
#    Skript weiter in der Funktion mit typeset -i zahl=$1
Fak $zahl
echo "Fak($1)=$fak"
exit 0
```

Und so sieht es aus, wenn das Skript ausgeführt wird:

```
buch@koala:/home/buch > skript38.sh 4
Start:
Fak
Fak
Fak
Fak
Fak
Fak(4)=24
buch@koala:/home/buch >
```

7.5 Aufgaben

1. Wenn in Skript 39 Fehler auftreten, geben Sie bitte eine Fehlermeldung in der ersten Zeile aus. Verwenden Sie dazu eine ODER-Liste (||), und schreiben Sie sich eine neue Funktion `CW_Error`.
2. Diese Version der Fakultät gibt immer 0 aus, warum? Tip: Achten Sie besonders auf die Definition der lokalen Variablen.


```
# Skript 38: Fakultät, rekursiv
#
# Nimmt den Parameter $1 und
# berechnet dazu die Fakultät
#
# 1. Hier wird die Funktion definiert, aber noch nicht ausgeführt
#
function Fak()
{ zahl=$1
  if [ $zahl -lt 1 ] ; then
     typeset -i fak=1
  else
     Fak $((zahl-1))
     fak=$zahl*fak
  fi
}
```

```
#
# 2. Hier startet das Skript nach dem Aufruf
#
typeset -i zahl=${1:-1}
typeset -i fak=-1   # Dummywert vordefinieren
if [ $zahl -lt 1 ] ; then
   echo "Fehler: Fakultät nur für Ganzzahlen größer 0" >&2
   exit 1
fi
#
# 3. Erst hier wird die Funktion aufgerufen. Jetzt läuft das
#    Skript weiter in der Funktion mit typeset -i zahl=$1
Fak $zahl
echo "Fak($1)=$fak"
exit 0
```

3. Skript 41 braucht noch Erweiterungen:

   - Die Frage zum Abbruch der Eingabeschleife in `CW_Eingabe` mit `gvEingabe='CR'` ist sicherlich nicht das Optimum. Wie könnte der Abbruch besser formuliert werden?
   - Mittels `m` markieren wir die aktuelle Zeile, d.h., wir nehmen den Dateinamen in die Variable `Mark` auf. Ein erneutes `m` einer markierten Zeile hebt den Eintrag auf. Sinnvoll wäre hier eine Funktion `CW_Mark`.
   - Ein ↵ auf ein Tararchiv sollte den Inhalt des Archivs in der zweiten Box auf der rechten Seite ausgeben. Mit a sollte das rechte Fenster aktiviert werden bzw. wenn dieses aktiv ist, das linke Fenster wieder den Fokus bekommen. Alle Eingaben beziehen sich auf das jeweils aktive Fenster. Ein Tipp: Speichern Sie die Information in `Mark`, `anz`, `zakt`, `akt` und `offset` beim Fensterwechsel in anderen Variablen ab.

7.6 Lösungen

1. Die folgenden Skriptteile demonstrieren eine Möglichkeit der Fehlerbehandlung mit einer ODER-Liste:

```
...
function CW_Error()
{
echo "\a"
CW_Print $1
}
...
```

```
#
# Fehlerbehandlung durch eine ODER-Liste
#
[ -d $verz ] || CW_Print "Verzeichnis wurde nicht gefunden"
anz=`ls $verz 2>/dev/null | tee $tmpfile | wc -l`
...
```

2. Die Variable `zahl` ist global angelegt. Das heißt, bei `zahl=2` passiert Folgendes:

`Fak 2`	
`zahl=2`	`zahl=$1`
`zahl -lt 1`	nein
`Fak 2-1`	
`zahl=1`	
`zahl -lt 1`	nein
`Fak 1-1`	
`zahl=0`	`zahl=$1`
`zahl -lt 1`	ja
`fak=1`	Ebene hoch
`fak=fak*zahl`	also `fak=1*0`, und hier kracht es!

Das Skript muss also folgendermaßen abgewandelt werden, damit es wieder korrekt die Fakultät berechnet:


```
# Skript 38: Fakultät, rekursiv
#
# Nimmt den Parameter $1 und
# berechnet dazu die Fakultät
#
# 1. Hier wird die Funktion definiert, aber noch nicht ausgeführt
#
function Fak()
{
  typeset -i zahl=$1
  if [ $zahl -lt 1 ] ; then
     fak=1
  else
     Fak $((zahl-1))
     fak=$zahl*fak
  fi
}

#
```

```
# 2. Hier startet das Skript nach dem Aufruf
#
typeset -i zahl=${1:-1}
typeset -i fak=-1   # Dummywert vordefinieren
if [ $zahl -lt 1 ] ; then
   echo "Fehler: Fakultät nur für Ganzzahlen größer 0" >&2
   exit 1
fi
#
# 3. Erst hier wird die Funktion aufgerufen. Jetzt läuft das
#    Skript weiter in der Funktion mit typeset -i zahl=$1
Fak $zahl
echo "Fak($1)=$fak"
exit 0
```

3. Entnehmen Sie die Lösung bitte dem nachfolgenden Skript:


```
# Skript 41: (Lösung)
#
# CW-Commander mit Verzeichniswechsel
#
# Achtung! Dieses Skript ist nicht komplett ausgedruckt,
# fehlende Routinen entnehmen Sie vorherigen Skripten
function CW_Mark ()
{
 # 1. Schon eingetragen?
 local line=`head -$zakt $1 |tail -1`
 set -- $line
 local name="$9"
 if [ -z "`echo $Mark | grep \"/$name/\"`" ] ; then
     Mark="$Mark$name/"
   else
    aifs=$IFS
    IFS="$IFS/"
    local neu="/"
    for wort in $Mark ; do
      if [ "$wort" != "$name" ] ; then
        neu="$neu$wort/"
      fi
    done
    IFS="$aifs"
    Mark=$neu
 fi
}

function CW_Archiv ()
{
  fnamen=$1
  sleep 1
```

```
  if [ -n "$fnamen" ] ; then
     fnamen=`echo "$fnamen" | tr "/" " "`
     typeset -i pro=`echo "$fnamen" |wc -w`
     typeset -i anz=100*pro
     tar cvfz $tmptar $fnamen >$tmptarcnt 2>/dev/null &
     typeset -i akt=0
     typeset -i erg=1
     prid=$!
     while [ $anz -gt $akt ] ; do
       akt=`cat $tmptarcnt |wc -l`
       akt=akt*100
       tput cup 10 38
       erg=akt/pro
       CW_Print 0 0 "$erg%"
     done
     rm $tmptarcnt
     prid=""
  fi
}

function CW_Eingabe()
 {
   gvEingabe=""
   local line=""
   until [ -n "$gvEingabe" ] ; do
      tput cup 22 0
      read -p "Eingabe:" gvEingabe
      case $gvEingabe in
       "q" | "Q") gvEingabe="q"
                  break;; # Verbesserter Abbruch
       "w" | "W")
           offset=offset+17
           if [ $offset -gt $anz ] ; then
          offset=offset-17
        else
           zakt=zakt+17
           if [ $zakt -ge $anz ] ; then
              zakt=anz
           fi
        fi
        ;;
       "z" | "Z")
           if [ $offset -gt 17 ] ; then
              offset=offset-17 ;
              zakt=zakt-17
           fi
           ;;
       "a" | "A") CW_Archiv "$Mark" ;;
```

```
       "n" | "N") if [ $zakt -lt $anz ] ; then
           zakt=zakt+1
           if [ $zakt -gt $((offset+17)) ] ; then
              offset=offset+17
           fi
       fi ;;
       "l" | "L") if [ $zakt -gt 1 ] ; then
          zakt=zakt-1
          if [ $zakt -lt $offset ] ; then
             offset=offset-17
          fi
       fi ;;
        "")   line=`sed -n -e "${zakt}p" $tmpfile`
              set -- $line
              local ch=`echo  ${1:0:1}`
              if [ "$ch" = "d" ] ; then
                 gvEingabe="CR"
                 CW_ReadDir "$9";
              else
                 CW_NormalFile "$9"
              fi ;;
        *)    gvEingabe="" ;;
      esac
   done
 }

Mark=":"
CW_Clear
verz=${1:-"`pwd`"}
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn  m = Markieren"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=1
tmpfile="/tmp/cwc$$.tmp"
rm -f $tmpfile
cd "$verz"
gvpp="/usr/bin/"
typeset -i anz=`ls -lAd * .. | tee $tmpfile | wc -l`
typeset -i zakt=1       # Zähler akt. Zeile
until [ "$gvEingabe" = "q" ] ; do
   CW_PrintDir 1 18 $tmpfile
   CW_Eingabe
done
exit 0
```

Kapitel 8

jetzt lerne ich

Prozesse und Signale

»Gegen Angriffe kann man sich wehren,
gegen Lob ist man machtlos« – Sigmund Freud

... aber ein wenig Lob ist schon angebracht, wenn Sie es bis hierher geschafft haben. Die meisten wichtigen Konzepte der Shellprogrammierung haben Sie schon kennen und beherrschen gelernt. Fehlt nur noch das Thema *Prozesse und Signale*, dem wir uns in diesem Kapitel widmen wollen. Da dieses Thema eng mit Unix verwoben ist, müssen wir noch einmal eine Textwüste mit jeder Menge Theorie durcharbeiten, und die folgt auf dem Fuße ...

8.1 Prozesse: Ein wenig Theorie

Ein Betriebssystem hat jede Menge zu tun. Es kontrolliert alle Ressourcen des Rechners, auf dem es läuft. Ressourcen sind Sachen wie Speicher, Festplattenplatz, CPU, Netzwerkkarte, serielle Ports und so weiter und so fort. Die alleine korrekt zu betreiben, ist schon ein wenig Aufwand, aber es kommt noch schlimmer: Programme wollen gestartet, Daten mittels Netzwerk empfangen und übertragen werden, Benutzer melden sich an, Zugriffe auf Geräte wie Soundkarte und Festplatte werden von Programmen durchgeführt und Hintergrundprozesse (im Unix-Slang *Daemons* genannt) fordern Rechenzeit und Speicher an. Und als wenn dies noch nicht genug wäre, tauschen viele Programme miteinander Daten aus, und der Zugriff auf die Geräte muss auch geregelt werden. Schließlich wäre es nicht sehr toll, wenn z.B. mehrere Pro-

gramme gleichzeitig auf die Soundkarte zugreifen würden (ja, ich weiß, es gibt Ausnahmen!).

Außerdem merkt sich ein vernünftiges Betriebssystem, welche Programme welche Ressourcen belegt haben und gibt sie am Programmende automatisch wieder zur Benutzung durch andere Programme frei, sollte das Programm es (unkorrekterweise) nicht selbst gemacht haben.

Es fallen also jede Menge Informationen an, und das Betriebssystem muss in der Lage sein, die Informationen genau einem Programm zuzuordnen. Unter Unix ist jedes Programm, das gestartet wird, ein eigener Prozess. Das gilt für die Shell genauso wie für alle aus der Shell heraus gestarteten Programme. So gesehen ist ein Skript ein steuernder Prozess (die Shell selbst), der andere Prozesse (die Skriptbefehle) startet, deren Ergebnisse verwaltet und bei Bedarf an andere Prozesse weitergibt. Für jeden Prozess werden u.a. Informationen abgelegt, wieviel Speicher er belegt, welche Rechte er hat und welche Dateien geöffnet wurden.

Um diese Verwaltungsaufgaben bewältigen zu können, muss das Betriebssystem in der Lage sein, die Prozesse eindeutig zu identifizieren. Unter Unix bekommt deshalb jeder Prozess eine eindeutige ID (*Identifier*). Dies ist eine positive Zahl größer 0. Der erste Prozess bekommt also die 1, der zweite Prozess die 2 usw. Diese Nummer kann nicht verändert werden. Diese Tatsache haben wir uns mit dem Parameter `$$` in Kapitel 5 zu Nutze gemacht, um Dateinamen eindeutig zu machen. Die Shell ersetzt `$$` durch ihre aktuelle Identifikationsnummer, die *Prozess-ID*.

Außerdem merkt sich das Betriebssystem, mit welchen Benutzerberechtigungen ein Programm läuft. In einem Multiuser/Multitasking-System hat nicht jeder Benutzer alle Rechte. Als normaler Anwender dürfen Sie beispielsweise keine Systemdateien editieren oder löschen. Da Ihre Programme wie oben ausgeführt nur Prozesse für das Betriebssystem sind, müssen diese Berechtigungen gerade für Prozesse kontrolliert werden. Dazu wird unter Unix festgehalten, welcher Benutzer welchen Prozess gestartet hat, und die Rechte des Benutzers bekommt auch jeder Prozess zugeteilt.

Natürlich können Sie sich alle angesprochenen Informationen jederzeit ausgeben lassen, dazu dient der Befehl `ps`. Dies ist wieder einmal eine typische Unxiabkürzung, die offensichtlich `:o)` für Prozessstatus (genau genommen ist es ein englisches Kürzel: *Process Status*) steht. `ps` ohne Parameter gibt alle Prozesse aus, die Sie unter Ihrem aktuellen Benutzernamen (eventuell auch auf mehreren Bildschirmen) gestartet haben. Informationen die ausgegeben werden, sind die Prozess-ID (Spalte `PID`), der Bildschirm, von dem aus der Prozess gestartet wurde (`TTY`), der Status des Prozesses (`STATUS`), Information über effektiv verbrauchte Zeit (`TIME`) und der ausgeführte Befehl.

```
buch@koala:/home/buch > ps
  PID TTY STAT TIME COMMAND
  184   1 S    0:00 -bash
  185   2 S    0:00 -bash
  209   1 S    0:00 joe kapitel8.txt
  467   2 R    0:00 ps
buch@koala:/home/buch >
```

ps ist eines der Programme, die sich auf fast jedem System anders verhalten oder andere Parameter nutzen. Eigen ist allen nur, dass Informationen über Prozesse ausgegeben werden.

Aus diesem Grunde mögen sich die Ausgaben auf Ihrem System durchaus von den Ausgaben in diesem Kapitel (Linux 2.0.35) unterscheiden.

Im obigen Beispiel war der Benutzer `buch` auf `tty2` und `tty1` (sprich Bildschirm 1 und Bildschirm 2) angemeldet. Auf beiden lief seine Bash (Prozesse 184 und 185). Auf `tty2` wurde der `ps`-Befehl ausgeführt, während auf `tty1` dieses Kapitel im Entstehen war.

Geben wir zusätzlich die Option `u` an, so gibt `ps` Informationen über den Benutzer (`USER`) aus, der die Prozesse aktiviert hat.

```
buch@koala:/home/buch > ps u
USER     PID %CPU %MEM  SIZE   RSS TTY STAT START   TIME COMMAND
buch     184  0.0  0.8  1752  1072   1 S    20:00   0:00 -bash
buch     185  0.0  0.8  1760  1096   2 S    20:00   0:00 -bash
buch     209  0.0  0.5  1232   660   1 S    20:02   0:00 joe kapitel8.txt
buch     480  0.0  0.2   872   356   2 R    21:25   0:00 ps u
```

Möchten Sie jetzt noch die Prozesse sehen, die von anderen Benutzern gestartet wurden, so geben Sie bei den Optionen ein `a` an.

```
USER    PID %CPU %MEM  SIZE   RSS TTY STAT START   TIME COMMAND
buch    184  0.0  0.8  1752  1072   1 S    20:29   0:00 -bash
buch    185  0.0  0.8  1760  1084   2 S    20:29   0:00 -bash
buch    186  0.0  0.8  1752  1068   3 S    20:29   0:00 -bash
buch   1245  0.0  0.5  1240   684   1 S    20:33   0:00 joe kapitel8.txt
buch   3455  0.0  0.2   880   364   3 R    21:51   0:00 ps ua
html    189  0.0  0.8  1752  1044   6 S    20:29   0:00 -bash
html   1169  0.0  0.5  1540   696   6 S    20:30   0:00 sh /usr/X11R6/bin/sta
html   1170  0.0  0.2   824   272   6 S    20:30   0:00 tee /home/html/.X.err
html   1171  0.0  0.4  2032   592   6 S    20:30   0:00 xinit /home/html/.xin
html   1173  0.0  2.2  4172  2844   6 S    20:30   0:00 kwm
html   1182  0.0  1.2  4428  1656   6 S    20:30   0:00 kaudioserver
html   1185  0.0  1.8  4280  2344   6 S    20:30   0:00 kwmsound
html   1188  0.0  3.6  6500  4616   6 S    20:30   0:01 kfm -d
html   1194  0.0  1.9  4416  2484   6 S    20:30   0:00 kbgndwm
html   1197  0.0  1.9  4364  2472   6 S    20:30   0:00 krootwm
html   1200  0.0  3.4  6012  4436   6 S    20:30   0:01 kpanel
```

```
html   1201  0.0  1.3  4476  1696   6 S    20:30   0:00 maudio -media 4
root    164  0.0  0.2   828   316  a0 S    20:29   0:00 /usr/bin/gpm -t ms -m
root   1172  0.0  6.3 17636  8088   6 S    20:30   0:01 X :0
root   3121  0.0  0.2   808   276   5 S    21:16   0:00 /sbin/mingetty tty5
root   3434  0.0  0.2   808   288   4 S    21:41   0:00 /sbin/mingetty tty4
```

Obige Ausgabe wurde ein wenig gekürzt, um nicht zu viel Platz zu verbrauchen. Sie sehen jetzt, dass drei Benutzer auf meinem Rechner aktiv sind bzw. aktive Prozesse haben: `html`, der offensichtlich nichts Besseres zu tun hat, als ein Spiel unter KDE zu spielen. `root`, der u.a. auf Anmeldungen auf den Bildschirmen `tty4` und `tty5` wartet, und dann noch der Benutzer `buch`, dessen Kapitel 8 scheinbar immer noch nicht fertig gestellt ist.

Im Zusammenhang mit der Shellumgebung und der Variablen `SHLVL` hatten wir bereits darauf hingewiesen, dass die Shell ein Programm startet und dann darauf wartet, dass es beendet wird. Dieser Prozess *schläft*, er macht nichts anderes als warten, daher der Status `S` (siehe auch Kapitel 6.5). Es ist übrigens vollkommen korrekt, dass die Optionen von `ps` ohne Minuszeichen angegeben werden. Der Grund liegt in zukünftiger Kompatibilität mit dem neuen Unix98-Standard, dessen Parameter mit dem – eingeleitet werden.

Unglücklicherweise ist der Benutzer `root` nicht angemeldet, sondern es laufen lediglich Prozesse mit `root`-Berechtigung. Zum einen sind das die Logins (`mingetty`), der X-Server (gestartet von `startx` mit Prozess-ID 219) und der Daemon für die Mausabfrage namens `gpm`. Welche Benutzer wirklich angemeldet sind, können Sie mit dem Befehl `who` oder mit dem Befehl `users` ermitteln:

```
buch@koala:/home/buch > users
buch buch buch html
buch@koala:/home/buch > who
buch     tty1     Mar 10 21:55
buch     tty2     Mar 10 22:04
buch     tty3     Mar 10 22:05
html     tty6     Mar 10 21:55
buch@koala:/home/buch >
```

Deutlich zu sehen ist die Tatsache, dass `who` mehr Informationen liefert als `users`, welches eine einfache Liste der angemeldeten Benutzer ausgibt. `who` gibt dabei den Benutzernamen, den Bildschirm, auf dem sich der Benutzer angemeldet hat, und das Datum inklusive Anmeldezeit aus.

Schreiben wir uns ein kleines Skript, das alle an Terminals oder Bildschirmen angemeldeten Benutzer ausgibt und für jeden die Anzahl an Prozessen und Bildschirmen, an denen er angemeldet ist, ermittelt.


```
# Skript 42: Dem Lektor sei Dank
# Ermittelt die Anzahl der Benutzer und
# die Anzahl der Prozesse pro Benutzer
#
who | cut -d' ' -f1 | sort | uniq -c | while read lvNum lvUser ; do
    echo -n "User: $lvUser hat $lvNum ttys und "
    echo     `ps ua | grep "^$lvUser" | wc -l` "Prozesse"
done
exit 0
```

Zunächst einmal ermittelt `who` alle angemeldeten Benutzer und gibt Informationen über Benutzernamen (Spalte 1), Bildschirm (Spalte 2) und Zeitpunkt der Anmeldung aus (Spalte 3). Falls noch eine vierte Spalte ausgegeben wird, so enthält diese entweder den Rechnernamen oder den Namen des X11-Displays, auf dem der Benutzer gerade arbeitet.

```
buch@koala:/home/buch > who
buch     tty1     Apr 29 20:28
buch     tty2     Apr 29 20:17
html     tty6     Apr 29 20:17
html     ttyp0    Apr 29 20:35
buch     ttyp1    Apr 29 20:55 (:0.0)
buch@koala:/home/buch >
```

Von dieser Ausgabe wird jeweils nur die erste Spalte durch das `cut` berücksichtigt, sodass nur die Benutzernamen ausgegeben werden. Da nicht garantiert ist, dass `who` die Ausgaben sortiert ausgibt, stellen wir das mit `sort` sicher.

`uniq` wiederum nimmt sortierte (!) Daten von der Standardeingabe und gibt Daten, die mehrfach in der Eingabe vorkommen, nur einmalig aus. Die Option `-c` führt dann dazu, dass vor jeder Zeile ausgegeben wird, wie häufig die Daten in der Eingabe gefunden wurden. Für unser obiges Beispiel bekommen wir am Ende von `uniq` folgende Ausgaben:

```
buch@koala:/home/buch > who | cut -d" " -f1 | sort |uniq -c
      2 buch
      2 html
buch@koala:/home/buch >
```

Diese Zeilen lesen wir nun mittels `while read` in die Variablen `lvNum` (die erste Spalte der Ausgabe) und `lvUser` (die zweite Spalte) ein. Die Daten geben wir mit `echo -n` aus. Das zweite `echo` sucht in der Prozessliste alle Zeilen, die für den Benutzer in `lvUser` zu finden sind, und zählt sie. Das Ergebnis der Zählung wird dann ausgegeben.

Noch einige abschließende Worte zu `ps`, bevor wir mit den Signalen weitermachen. `ps` hat jede Menge Optionen, mit denen Sie viel mehr Informationen über Ihre Prozesse in Erfahrung bringen können. Sie können fast alle

Optionen miteinander kombinieren, um die Informationen zu erhalten, die Sie benötigen. Als letzte Option möchte ich an dieser Stelle noch das `ps f` anführen. Dies zeigt auf, welche Befehle von welchen Befehlen aufgerufen wurden. Geben Sie noch die Option `w` an, und die Ausgabe der Informationen wird nicht nach 80 Zeichen abgeschnitten (*wide line*: Zeilen nicht abschneiden). Leider gilt dies nur für die GNU-Version, so gibt die Option `f` auf System-V-Unices die Umgebung der Prozesse aus.

Schauen wir uns noch einmal die Benutzer `html` und `root` an. Von Letzterem hatte ich behauptet, er wäre nicht direkt angemeldet. Stattdessen liefe ein Prozess von `html` mit den Berechtigungen von `root`. Die Begründung war zwar logisch, aber nur teilweise von den `ps`-Ausgaben untermauert.

Schauen wir uns nun die Ausgabe von `ps uafw` an. Ich habe einige Spalten der Ausgabe entfernt, damit die Ausgabe nicht breiter als die Buchseite wird, und die Ausgaben beziehen sich nur auf den interessanten Bereich:

```
USER  PID ... STAT COMMAND
html  189     S     -bash
html  1169    S      \_ sh /usr/X11R6/bin/startx
html  1171    S      |   \_ xinit /home/html/.xinitrc --
html  1173    S      |       \_ kwm
root  1172    S      |       \_ X :0
html  1170    S      \_ tee /home/html/.X.err
html  1188    S     kfm -d
html  1203    S      \_ kshisen
```

Diese Ausgabe belegt, dass Prozesse vom Betriebssystem nicht einfach eingetragen werden, sondern auch noch die Abhängigkeiten beachtet werden. Hier ist `bash` die Shell, von der mittels `startx` das X-Window-System gestartet wurde. Stark vereinfacht ausgedrückt, ist Prozess 189 ein Kontrollprozess, weil alle weiteren Prozesse direkt oder indirekt unter seiner Kontrolle gestartet wurden. Stirbt ein Kontrollprozess (in diesem Beispiel Prozess 189), so werden alle von ihm gestarteten Prozesse ebenfalls beendet, weil das Betriebssystem an die betroffenen Prozesse ein Signal schickt, welches sie zur Beendigung auffordert (`SIGHUP`). Was Signale sind, werden wir jetzt genauer unter die Lupe nehmen.

8.2 Signale: Noch ein wenig mehr Theorie

Wie im letzten Abschnitt schon angedeutet, laufen Programme nicht nur so vor sich hin, sondern kommunizieren auch mit anderen Programmen. In den letzten Kapiteln hatten wir schon einige dieser Methoden angewandt und besprochen (Exitstatus, Pipes, Eingabeumlenkung).

Eine weitere Methode werden wir in diesem Abschnitt kennen lernen, die Signale. Signale sind kurze Nachrichten, die ein Prozess an einen zweiten Prozess schickt. Diese Nachrichten haben besondere Bedeutungen, und jeder Prozess kann die für ihn bestimmten Signale abfangen und darauf reagieren. Tut er das nicht, so reagiert das Betriebssystem für den betroffenen Prozess mit einer Standardaktion.

Reagieren kann eine von drei Aktionen bedeuten:

- Signal ignorieren.

  Das erhaltene Signal wird nicht beachtet, und das adressierte Programm läuft ohne Reaktion weiter. Die Signale `SIGKILL` (Prozess beenden) und `SIGSTOP` (Prozess anhalten) können unter keinen Umständen abgefangen werden. Das Betriebssystem übernimmt die Behandlung dieser Signale und den von ihnen angesprochenen Prozessen.

- Die Standardaktion ausführen.

  Abhängig vom geschickten Signal, reagieren Prozesse mit Standardaktionen. Siehe dazu Tabelle 8.1.

- Das Signal abfangen.

  Dazu setzen die Prozesse Routinen für Signale ihrer Wahl auf, die aktiviert werden, falls der Prozess entsprechende Signale erhält. Mit diesem Thema wird sich dieser Abschnitt primär beschäftigen.

Signale sind asynchrone Nachrichten, die in unseren Skripten auftreten können und Aktionen auslösen. Hört sich wahnsinnig kompliziert an, bedeutet aber schlicht Folgendes:

Unsere Skripten sind für den Computer das, was Wegbeschreibungen für einen Autofahrer sind. Sie sagen dem Computer, wie eine Aufgabe zu lösen ist. Sie beschreiben die Ausgangssituation, verändern Variablen, rufen Befehle auf und lösen so unser Gesamtproblem Schritt für Schritt. Dabei enthält das Skript eine zeitliche Komponente, denn es hat absolut keinen Sinn, den zweiten Schritt vor dem ersten auszuführen. Bleiben wir bei der Wegbeschreibung. Was passiert, wenn eine Ampel auf Rot schaltet oder eine Umleitung eingerichtet wurde? Ein Autofahrer wird vor der Ampel halten und der Umleitung folgen. Solche Aktionen können auf der Fahrt zum Ziel an jeder Stelle auftreten, müssen es aber nicht. Wenn Ihre Wegbeschreibung alle diese Eventualitäten berücksichtigen würde, wäre sie lang und unverständlich. Einfacher wäre die Aussage: »Wenn du auf eine Umleitung triffst, so folge ihr«. Dieser Satz handelt alle möglichen Umleitungen auf der gesamten Strecke ab. Mit »Warte vor einer roten Ampel, bevor du deinen Weg fortsetzt« wird auch die Ampelproblematik erledigt.

Was hat das nun mit unseren Skripten und den Signalen zu tun? Ein Skript ist für den Computer nichts anderes als eine Wegbeschreibung zur Lösung eines Problems. Umleitung bzw. Ampel sind Signale, welche an unser Skript geschickt werden können.

Jetzt müssen Sie eigentlich nur noch wissen, welche Signale es gibt, was sie bedeuten und welche Standardaktion für die Signale vorgesehen sind. Diese Informationen können Sie für die wichtigsten Signale aus Tabelle 8.1 entnehmen. Eine Übersicht über alle Signale finden Sie unter *signal(7)* in den Manual Pages.

Tabelle 8.1: Die wichtigsten Signale

Signal	Standardaktion	Bedeutung
`SIGKILL`	Prozess beendet	Dieses Signal kann nicht abgefangen werden. Das Betriebssystem beendet diesen Prozess sofort ohne Rückfrage.
`SIGSTOP`	Prozess anhalten	Auch dieses Signal kann nicht abgefangen werden. Allerdings wird der Prozess nur angehalten. Er kann mit `SIGCONT` weiterlaufen.
`SIGPIPE`	Prozess abbrechen	Broken Pipe: Erinnern Sie sich an Kapitel 3.3 und Skript 18, welches einen `Broken Pipe`-Fehler ausgab. Dessen Ursache war dieses Signal. Es tritt auf, wenn in eine Pipe geschrieben wird, nachdem der aus der Pipe lesende Prozess beendet wurde.
`SIGUSR1`	Prozess abbrechen	User defined Signal 1: ein Signal, dessen Bedeutung vom Benutzer für sein Skript/Programm selbst festgelegt werden kann.
`SIGUSR2`	Prozess abbrechen	User defined Signal 2: wie `SIGUSR1`. Beide Signale haben für Unix keine Bedeutung und dürfen vom Benutzer nach seinem Gutdünken verwendet werden.
`SIGTERM`	Prozess abbrechen	TERMINATE: die Aufforderung an den Prozess, sich so schnell wie möglich zu beenden. Erhält Ihr Prozess dieses Signal, so sollte er alle von ihm belegten Ressourcen freigeben und sich beenden. So wird beim Shutdown des Systems kurz vor dem Ende an alle Prozesse ein `SIGTERM` geschickt und einige Sekunden später ein `SIGKILL`.
`SIGHUP`	Prozess abbrechen	HANG UP: Prozesse, die von der Shell in den Hintergrund gelegt wurden, bekommen dieses Signal zugeschickt, wenn die Loginshell, aus der sie gestartet wurden, beendet wird (siehe Kapitel 8.1).

Signal	Standardaktion	Bedeutung
		Der Name des Signals kommt daher, weil er in der Regel dann an einen Kontrollprozess geschickt wird, wenn sich der Benutzer vom Bildschirm abmeldet, von dem der Kontrollprozess gestartet wurde. Da früher Terminals Geräte waren, die Bildschirm und Tastatur stellten und die Kommunikation mit dem Unxirechner über serielle Leitungen (durchaus auch Telefonleitungen) regelten, führte eine Abmeldung dazu, dass der bei einer Abmeldung *aufgelegt* wurde, engl. *Hang up*. In diesem Fall konnten die Prozesse des Benutzers, die Ausgaben auf dieses Terminal machten, ebenfalls beendet werden.
`SIGINT`	Prozess abbrechen	INTERRUPT: wird von der Tastatur generiert. Ist meistens ein Strg+C oder ein Entf. Soll das aktuelle Programm abbrechen.
`SIGQUIT`	Prozess abbrechen	QUIT: wie Interrupt, nur wird zusätzlich noch ein Speicherabzug (core) auf der Platte abgelegt. Ist meistens Strg+\ oder Strg+Druck.
`SIGSEGV`	Prozess abbrechen	Auf alten Systemen war der Speicher in kleinere Bereiche, so genannte Segmente, unterteilt. Vereinfacht gesagt, durfte jedes Programm nur Speicher in seinem Bereich (Segment) nutzen. Griff es auf ein anderes Segment zu, so war das eine `Segmentation Violation` (kurz `SEGV`). Moderne Rechner verwalten ihren Speicher mittlerweile wesentlich effektiver, sodass es solche Segmente nicht mehr gibt (oder sie so groß sind, dass es nicht mehr wichtig ist). Dennoch wird bei illegalen Speicherzugriffen auch heute noch `SIGSEGV` generiert. Bei diesem Signal wird der aktuelle Speicherinhalt des Prozesses in einer Datei namens `core` abgelegt, aus der ein Programmierer entnehmen kann, was genau schiefgegangen ist. Aber dies und die Namensgebung ist eine andere Geschichte ...

Alle Signale haben auch eine eindeutige Nummer, die anstatt des Namens eingesetzt werden kann, die Sie unter *signal(7)* finden. So ist `SIGQUIT` identisch mit `3` oder `SIGKILL` mit `9`.

8.2.1 kill oder: Wink mit dem Zaunpfahl


```
kill -<signal> <prozessid>
```

Richten wir nun eine Umleitung ein oder setzen eine Ampel auf Rot: Wir generieren Signale. Dazu dient der Befehl `kill`. Trotz des martialischen Namens beendet er Prozesse nicht grundsätzlich, sondern schickt ein `<signal>` an den Prozess mit der ID `<prozessid>`. Wie Sie an eine Prozess-ID herankommen, wissen Sie bereits. `<signal>` ist dabei der Name des Signals aus der Tabelle weiter oben. Ein kleines Beispiel, zu dem Sie allerdings zweimal mit dem gleichen Namen angemeldet sein müssen: Auf Bildschirm eins geben Sie einfach nur `cat` ein und schalten dann auf die zweite Anmeldung um. Dort ermitteln Sie mittels `ps` die Prozess-ID vom gerade gestarteten `cat` und schicken diesem Prozess dann ein Signal:

```
buch@koala:/home/buch > ps
  PID TTY STAT TIME COMMAND
  184   1 S    0:00 -bash
  185   2 S    0:00 -bash
  186   3 S    0:00 -bash
  262   1 S    0:00 joe kapitel8.txt
  272   2 T    0:00 pg -
  277   2 S    0:00 cat
  285   3 R    0:00 ps
buch@koala:/home/buch > kill -SIGPIPE 277
buch@koala:/home/buch >
```

Auf Bildschirm eins sehen Sie das Ergebnis:

```
buch@koala:/home/buch > cat
Broken pipe
buch@koala:/home/buch >
```

Obwohl keine Pipe genutzt wurde, haben Sie einen Fehler `Broken Pipe` erhalten, womit ich meine Beweisführung abgeschlossen habe.

`kill` sendet zwar Signale an Prozesse, aber auch dabei werden Berechtigungen beachtet. Es ist z.B. nicht erlaubt, Signale an Prozesse zu senden, die nicht von Ihnen gestartet wurden. Hätten Sie den Befehl `cat` mit einem anderen Benutzernamen gestartet, so hätten Sie eine Fehlermeldung erhalten:

```
buch@koala:/home/buch > ps ua
...
html       320  0.0  0.2   828   284   4 S    18:26   0:00 cat
...
buch@koala:/home/buch > kill -SIGPIPE 320
bash: kill: (320) - Not owner
buch@koala:/home/buch >
```

Die Ausnahme von der Regel ist mal wieder der Benutzer `root`. Dieser darf als einziger auch Prozesse von anderen Benutzern mit Signalen belästigen. Sie haben es sicherlich schon gemerkt: Immer, wenn es um Ausnahmen bei Rechten geht, dann hat `root` seine Hand im Spiel. Aus diesem Grunde sollten Sie so wenig wie möglich mit dieser Anmeldung arbeiten. Wenn Sie auf Ihrer privaten Unixstation arbeiten, dann würden Sie nur sich selbst schaden, aber auf einem Unternehmensrechner darf diese »Macht« nur in den Händen verantwortlicher und vertrauenswürdiger Benutzer liegen (kurz gesagt, beim Systemadministrator).

Wenn Sie einen Prozess mittels `SIGKILL` beenden wollen, so schicken Sie lieber zuerst ein `SIGTERM` an den Prozess. Dies erlaubt es dem Prozess, seine angeforderten Ressourcen freizugeben und sich korrekt selbst zu beenden. Erst wenn diese Methode kein Erfolg zeitigt (warten Sie nach dem `SIGTERM` noch ein paar Sekunden), sollten Sie den Prozess brutal abschießen.

Der Rückgabewert von `kill` ist `0`, wenn der angegebene Prozess das Signal erhalten hat, und sonst 1 (`false`). Der Rückgabewert von normalen Befehlen ist `128+<signr>`, wobei `<signr>` die Nummer des Signals ist, welches den Befehl abgebrochen hatte. Eine Liste der Nummern und Signalnamen erhalten Sie durch die Eingabe von `kill -l`.

8.2.2 trap


```
trap -l
trap -p    (erst ab Bash 2.0)
trap '<befehle>' <signal>
trap -<signal>
trap <signal>
```

Und damit bleibt nur ein einziges Themengebiet offen: wie diese Signale von der Shell abgefangen werden können. In der Shell können Sie dies mithilfe des `trap`-Befehls erreichen. Mit der Option `-l` gibt der Befehl eine Liste aller verfügbaren Signale und ihrer Nummern aus. Wie bereits erwähnt, können Sie beim `kill` oder `trap` den Namen des Signals auch durch die Nummer des Signals ersetzen. Mit Hilfe `trap -l` bekommen Sie eine Zuordnung Signalnummer zu Signalname.

Falls Sie hinter dem `trap` nur den Namen oder die Nummer eines Signals angeben, so wird für dieses Signal die Standardaktion ausgeführt, wenn dieses

Signal an die Shell gesendet wird. Möchten Sie das Signal ignorieren, so rufen Sie `trap` mit dem Befehl `''` auf:

```
trap  '' SIGINT      # Ignoriert SIGNINT
```

Wollen Sie einen Befehl ausführen, wenn Ihr Skript ein Signal erhält, so müssen Sie den Befehl in Gänsefüßchen oder Apostrophe einschließen und das entsprechende Signal dahinter angeben. Dies ist notwendig, damit der Befehl oder die Befehle nicht in Wörter aufgeteilt werden, sondern in einem Stück an `trap` übergeben werden.

Wenn Sie jetzt eine Übersicht brauchen, welche Befehle bei welchem Signal aufgerufen werden, so können Sie auch das erreichen. Ein `trap -p` führt Sie zum Ziel.

Zu guter Letzt kann es auch passieren, dass Sie Ihre Signalbehandlung wieder abstellen wollen und die gleichen Einstellungen vornehmen wollen, wie zu dem Zeitpunkt, als das Skript gestartet wurde. Unter diesen Umständen geben Sie »-« anstelle des Befehls an: `trap – SIGPIPE`.

Ohne praktisches Beispiel war alles umsonst. Daher ein Minibeispiel, welches erneut nur mit zwei gleichen Anmeldungen funktioniert. Wir schreiben ein Skript, das die Signale `SIGALARM`, `SIGTERM`, `SIGQUIT`, `SIGINT` und `SIGHUP` abfängt und den Namen des Signals ausgibt. Wurde `SIGTERM` geschickt, so beendet sich das Skript nach der Ausgabe des Signalnamens.


```
# Skript 43: Traps
#
trap 'echo " QUIT  "' 3
trap 'echo " INT   "' 2
trap 'echo "HUP "' 1
trap 'echo "TERM   " ; exit 0' 15
trap 'echo "ALARM  "' 14
while true ; do : ; done
exit 0
```

Der Befehl »:« macht nichts, außer eventuell angegebene Parameter auszuwerten (z.B. Ersatzmuster, Brace Extension) und gegebenenfalls definierte Umlenkungen durchzuführen.

In der Bash können Sie statt `INT` auch `SIGINT` verwenden. In der Kornshell müssen Sie aber `INT` verwenden. Das heißt, hier müssen Sie bei der Angabe eines Signals jeweils »`SIG`« vorne weglassen.

Falls Sie aus einem Skript Nr. 1 ein weiteres Skript Nr. 2 starten, so übernimmt dieses die Umgebung von Skript1. Gleiches gilt auch für die `trap`-Einstellungen. Wenn Sie die Einstellungen mittels `trap - <signal>` in Skript2 zurücksetzen, wird die Einstellung wiederhergestellt, die in Skript1 zum Zeitpunkt des Aufrufs von Skript2 vorlag.

Damit besitzen Skript1 und Skript2 andere Traps als Voreinstellung: Skript1 übernimmt die Einstellungen der Shell, Skript2 die geänderten Einstellungen aus Skript1.

Leider gibt es wohl kaum eine Regel ohne Ausnahme:


```
# Skript trapok.sh
# Fängt SIGINT ab (<CTRL>+<C>)
#
trap "echo INT" INT
echo "mit kill -SIGTERM $$ beenden"
while true ; do sleep 2 ; echo sleep 2 ; done
exit 0
```

Dieses Skript läuft so wie erwartet: nämlich bis es durch ein anderes Signal als `SIGINT` beendet wird. Der gleiche Aufbau nur mit einer Subshell bricht leider nach dem ersten `SIGINT` ab:


```
# Skript trapko.sh
# Fängt SIGINT ab (<CTRL>+<C>), aber leider nur einmal
# Zeigt trap-Verhalten in Subshells
trap "echo INT" INT
echo "mit kill -SIGTERM $$ beenden"
( while true ; do sleep 2 ; echo sleep 2 ; done )
exit 0
```

Aber wenigstens verhalten sich `bash` und `ksh` an dieser Stelle gleich. Die Subshells betrachten wir in Abschnitt 8.6 noch genauer.

8.3 Programme im Hintergrund: &

Unix ist ein Multiuser/Multitaskingsystem. Unsere Skripten waren jedoch immer auf einen Prozess oder die Befehle (und damit Prozesse) innerhalb einer Pipe beschränkt.

Denken Sie einmal zurück an die graue Steinzeit von Kapitel 4. Dort haben wir ein Skript geschrieben, das ein Tar-Archiv erstellte und gleichzeitig einen Fortschrittsbalken mit prozentualen Angaben ausgegeben hat. Die Erstellung des Archivs mit gleichzeitiger Ausgabe des Fortschritts war ein wenig umständlich, musste doch jede Datei einzeln an das Archiv angehängt und dann der Fortschritt ausgegeben werden.

Es wäre doch wesentlich schöner, wenn `tar` und Fortschrittsermittlung parallel nebeneinanderher liefen. Gerade in einem Mehrprozess-System sollte dies doch möglich sein.

Diese Funktion können Sie erreichen, wenn Sie ein kaufmännisches Undzeichen »&« hinter den Befehl stellen. Wenn das Undzeichen hinter eine Pipe gestellt wird, so werden alle Befehle der Pipe in den Hintergrund gelegt. Ein im Hintergrund laufender Prozess blockiert die Shell nicht weiter und kann keinerlei Eingaben mehr von der Tastatur entgegennehmen. Die Shell muss nicht darauf warten, dass der Prozess beendet wird, sondern arbeitet den nächsten Befehl sofort ab. Wird die Standardausgabe des Hintergrundprozesses nicht umgelenkt, so gehen alle Ausgaben weiterhin auf die Standardausgabe des Skripts. Dies kann dazu führen, dass sich Ausgaben von Vordergrund und Hintergrund vermischen, was nicht immer wünschenswert ist. Auf Prozesse, die von der Shell im Vordergrund gestartet werden, muss die Shell so lange warten, bis der Prozess beendet ist.

Versucht ein im Hintergrund laufender Prozess von der Tastatur zu lesen, so hält der Prozess an. Um dieses Problem zu lösen, muss der Prozess in den Vordergrund geholt werden. Damit werden wir uns in Abschnitt 8.8 *Jobverwaltung* genauer beschäftigen.

Die Prozess-ID des zuletzt gestarteten Hintergrundprozesses können Sie durch `$!` abfragen. Beendet sich ein Skript, welches einen Hintergrundprozess gestartet hat, bevor der Hintergrundprozess beendet wurde, so wird der Hintergrund nicht beendet, sondern läuft weiter. Wird allerdings der Kontrollprozess beendet, so wird auch an diesen Prozess `SIGHUP` geschickt.

Kommen wir nun noch zu einem Beispiel, welches Hintergrundprozesse und Signale nutzt. Das bereits angesprochene Skript mit `tar` und Fortschrittsbalken hebe ich mir für die Aufgaben auf. Wir wollen lieber an dieser Stelle unseren CW-Commander erweitern. Dieser war bisher bereits in der Lage, Dateien zu markieren. Jetzt wollen wir diese Möglichkeit auch zu etwas Sinnvollem nutzen. Und zwar soll das Skript durch Eingabe von [a] alle markierten Dateien im aktuellen Verzeichnis als `backup.tar` sichern. In der Statuszeile (Zeile 1) wollen wir dabei den aktuellen Status ausgeben. Wird während der Sicherung auf [Strg]+[C] gedrückt, so soll die Sicherung abgebrochen werden. Läuft keine Sicherung und es wird [Strg]+[C] gedrückt, dann wird das Skript abgebrochen, allerdings nicht ohne vorher die eventuell vorliegenden temporären Dateien zu löschen.

Auch an dieser Stelle soll nicht das gesamte Skript ausgedruckt werden.


```
# Skript 44:
#
# CW-Commander mit Archivierung
#
# Dieses Skript ist ebenfalls unvollständig, hier nur
# die Neuigkeiten und Änderungen im Vergleich zur letzten Version!
# Demonstriert Markierung, Archivierung und Signalbearbeitung
#
function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     # Tarsicherung abbrechen
     kill -SIGTERM $prid 2>/dev/null
     rm -f $tmptar
     rm -f $tmptarcnt
     CW_Print 0 0 "Sicherung abgebrochen!"
    else
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     echo
     exit 0
  fi
}
function CW_Archiv ()
{
  local fnamen=$1
  sleep 1
  if [ -n "$fnamen" ] ; then
     fnamen=`echo "$fnamen" | tr ":" " "`
     typeset -i pro=`echo "$fnamen" |wc -w`
     typeset -i anz=100*pro
     tar cvf $tmptar $fnamen >$tmptarcnt 2>/dev/null &
     typeset -i akt=0
     typeset -i erg=1
     prid=$!
     while [ $anz -gt $akt ] ; do
       akt=`cat /tmp/tar.cnt |wc -l`
       akt=akt*100
       tput cup 10 38
       erg=akt/pro
       CW_Print 0 0 "$erg%"
     done
     rm $tmptar
     prid=""
  fi
}
function CW_Eingabe()
{
  # Diese Funktion wurde leicht erweitert
  #
```

```
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
                 offset=offset+17
                 if [ $offset -gt $anz ] ; then
                    offset=offset-17
                   else
                    zakt=zakt+17
                    if [ $zakt -gt $anz ] ; then
                       zakt=anz
                    fi
                 fi
                 ;;
      "z" | "Z")
                 if [ $offset -gt 2 ] ; then
                    offset=offset-17 ;
                    zakt=zakt-17
                 fi
                 ;;
      "n" | "N") if [ $zakt -lt $anz ] ; then
                    zakt=zakt+1
                    if [ $zakt -gt $((offset+17)) ] ; then
                       offset=offset+17
                    fi
                 fi ;;
      "l" | "L") if [ $zakt -gt 2 ] ; then
                    zakt=zakt-1
                    if [ $zakt -lt $offset ] ; then
                       offset=offset-17
                    fi
                 fi ;;
      "m" | "M") CW_Mark $tmpfile
                ;;
      "a" | "A") CW_Archiv "$Mark"
                 ;;
      "")        local line=`head -$zakt $tmpfile|tail -1`
                 set -- $line
                 local ch=`echo  $1 | cut -c1`
                 if [ "$ch" = "d" ] ; then
                    ein="CR"
                    Mark=":"    # NEU !
                    CW_ReadDir $9;
                   else
                   CW_NormalFile $9
                 fi ;;
       *) CW_Print 0 0 ">>$Mark<<"
          ein="" ;;
     esac
  done
}
```

```
Mark=":"
tput clear
verz=${1:-"`pwd`"}
trap "CW_TrapSIGINT" SIGINT
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp   w = PgDn  l = CuUp  n = CuDn"
CW_Print 0 22 "a = archivieren   m = markieren"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=2
tmpfile="/tmp/cwc$$.tmp"
tmptar="/tmp/tar.cnt$$"
cd "$verz"
typeset -i anz=`ls -ld .* * | tee $tmpfile | wc -l | cut -c-7`
typeset -i zakt=2         # Zähler akt. Zeile
until [ "$ein" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
rm -f $tmpfile
rm -f $tmptar $tmptarcnt
exit 0
```

Noch ein paar Anmerkungen zu diesem Skript.

`rm -f` erzwingt die Löschung der angegebenen Datei, falls die Rechte es zulassen. Dabei werden keine Fehlermeldungen auf die Standardfehlerausgabe ausgegeben und auch Rückfragen (wie z.B. `Override Mode 644?`) unterlassen: Entweder es kann gelöscht werden oder nicht. Nur anhand des Rückgabewertes lässt sich ermitteln, ob die Datei gelöscht wurde (Wert =0) oder nicht (Wert >0).

Die Variable `Mark` wird beim Verzeichniswechsel wieder zurückgesetzt, da wir uns nur die Dateinamen ohne Pfadangabe merken. Wird das Verzeichnis gewechselt, so wären diese Dateien nicht mehr auffindbar und würden `tar` zu Fehlern veranlassen. Aus diesem Grunde setzen wir `Mark` lieber wieder zurück.

8.4 wait – Warten auf Godot?

```
wait [prozessid]
```

Nein, auf Godot warten wir sicherlich nicht. Vor allem bestünde dann die Gefahr, dass wir endlos warteten. Wie der Name schon sagt, wartet `wait` auf die Beendigung eines Prozesses mit der ID `<prozessid>`. Dadurch läuft das Skript

erst dann weiter, wenn der angegebene Prozess beendet wurde. Wie der Prozess beendet wurde, lässt sich aus dem Rückgabewert von `wait` entnehmen. Ist dieser gleich 127, so wurde auf einen Prozess gewartet, der nicht (mehr) existiert. Ansonsten ist der Rückgabewert gleich dem Rückgabewert des Prozesses `<prozessid>`.

Kehren wir noch einmal zurück zum Skript 43 und dessen Verhalten beim Drücken von Strg+C. War eine Sicherung aktiv, so wurde sie mit `kill SIGTERM` höflich aber bestimmt abgeschossen. Ob dies bei Sicherungen so geschickt ist, bleibt zweifelhaft. Zumindestens im Zusammenhang mit dem Thema dieses Kapitels wäre es sicherlich besser, die Sicherung ablaufen zu lassen und dann das Skript kontrolliert zu beenden. Wie sähe also die geänderte Routine mit einem `wait` aus?


```
# Skript 44: Version mit wait
#
function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     wait $prid
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     echo
     exit 0
  fi
}
```

Natürlich ist ein Skript nicht darauf beschränkt, nur einen Prozess im Hintergrund zu starten, sondern es kann beliebig viele Prozesse starten. Wenn Sie warten wollen, bis alle von Ihrem Skript gestarteten Hintergrundprozesse beendet sind, so rufen Sie `wait` ohne Parameter auf. Dann ist der Rückgabewert immer gleich `0`.

8.5 Prioritäten

Wenn Sie praktisch veranlagt sind, so gehen Ihnen lange Theorieergüsse sicherlich auf die Nerven. Unter diesen Umständen können wir Ihre Nerven nur bewundern, denn Sie haben allein in diesem Kapitel einiges erdulden müssen, um so weit zu kommen. Aber es kommt noch schlimmer. Vielleicht ist Ihnen folgende Programmierweisheit ein Trost:

»Diejenigen, die es können, machen es. Diejenigen, die es nicht können, lehren es. Und diejenigen, die es nicht lehren können, programmieren.«

Irgendwie werden wir das Gefühl nicht los, dass dies der richtige Spruch an der falschen Stelle war ...

Machen wir lieber weiter mit dem nächsten Thema: *Prioritäten*. Jeder Prozess hat eine Priorität von -20 bis 19. Die höchste Priorität ist dabei -20 und die niedrigste ist 19. Abhängig von der Priorität bekommt der Prozess mehr oder weniger Zeit von der CPU zugeteilt.

Multitasking bedeutet nicht, dass die CPU so leistungsstark ist, dass sie mehrere Aufgaben gleichzeitig bearbeiten kann. Im Gegenteil, zu einem gegebenen Zeitpunkt wird immer genau ein CPU-Befehl ausgeführt. Der Eindruck, dass der Rechner mehrere Aufgaben gleichzeitig bearbeitet, resultiert daraus, dass das Betriebssystem die verfügbare Rechenzeit zunächst einmal gleichmäßig unter allen Prozessen des Systems aufteilt und dann in kurzen Zeitabständen zwischen den einzelnen Prozessen umschaltet. Dies geschieht so schnell, dass für den Menschen der Eindruck entsteht, es würden alle Prozesse gleichzeitig abgearbeitet. Tatsächlich laufen nicht wirklich alle Prozesse gleichzeitig, einige warten auf Signale von anderen Prozessen und laufen so lange nicht weiter.

Die Rechenzeit, die ein Prozess erhält, bevor das Betriebssystem zum nächsten Prozess umschaltet, ist zum einen abhängig von der Anzahl der vorhandenen Prozesse, von der Leistung der CPU und von der Priorität. Je kleiner die Priorität, desto länger die Rechenzeit, die der Prozess erhält. Da die verfügbare Rechenzeit auf einem gegebenen Rechner jedoch eine Konstante ist (die CPU wird nicht schneller oder langsamer durch die gesetzten Prioritäten!), verlieren die Prozesse geringerer Priorität anteilig Rechenzeit.

8.5.1 Seid nett zueinander: nice

```
nice -n <nice> <befehl>
```

Je *netter* sich ein Prozess im System verhält, desto geringer wird er seine Priorität setzen und dadurch das System weniger belasten. Befehle, welche durch die Shell gestartet werden, übernehmen die Priorität der Shell. Allerdings können sie ihre Priorität verändern und sich so netter verhalten. Der Befehl dazu lautet `nice`. `nice` (meint tatsächlich nett) verändert die Priorität des `<befehl>s` , indem der `<befehl>` die aktuelle Priorität des Skripts (oder der Shell, falls aus der Shell gestartet) übernimmt und darauf den `<nice>`-Wert addiert. Normale Benutzer dürfen nur positive `<nice>`-Werte angeben. Nur der Superuser `root` darf auch negative Werte angeben und somit die Priorität erhöhen.

8.5.2 Lynchjustiz verhindern: nohup


```
nohup <befehl>
```

Wie bereits erwähnt, werden Hintergrundprozesse an einen Vaterprozess gebunden. Wird dieser beendet, werden gleichzeitig auch alle Kindprozesse beendet. Dies ist aber nicht immer erfreulich, schließlich ist es unsinnig, eine Anmeldung (=Vordergrundprozess) aktiv zu halten, nur damit ein Hintergrundprozess nicht beendet wird.

Mit dem Befehl `nohup` wird der `<befehl>` so gestartet, dass

- er eine ausreichende Priorität erhält, um im Hintergrund zu laufen.
- er `SIGHUP`-Signale ignoriert und sich nicht beendet.
- alle Ausgaben von Standardfehler und Standardausgabe in eine Datei namens `nohup.out` umgelenkt werden, falls diese sonst auf die Standardausgabe gegangen wären.

Die Datei `nohup.out` wird im aktuellen Arbeitsverzeichnis angelegt oder, falls dies nicht geht, unter `$HOME/nohup.out`. Lassen sich beide Dateien nicht anlegen, so startet `<befehl>` nicht.

`nohup` schickt den übergebenen Befehl nicht automatisch in den Hintergrund, das müssen Sie mit `&` selbst übernehmen.

```
buch@koala:/home/buch > nohup grep -i Christa `find / -name "*.txt" -print` &
buch@koala:/home/buch > exit
```

<Eine Anmeldung später ...>

```
buch@koala:/home/buch > cat nohup.out
./kapitel2.txt:<CHRISTA>Hi Christa, dieser Kommentar ist nur für Dich.
./kapitel2.txt:<CHRISTA>Hier vielleicht auch ein Schema?</CHRISTA>
./kapitel8.txt:</CHRISTA>./kapitel4.txt:<CHRISTA>In diesem Codestück
./kapitel6.txt:export a=Christa
./kapitel6.txt:in 33 #2: (Christa)
./kapitel6.txt:</CHRISTA>./kapitel7.txt:<CHRISTA>
./kapitel5.txt:<CHRISTA>
./kapitel13.txt:Christa: "BETTINA!!") okay, die ich
./kapitel13.txt:einfach gemacht"), von wem das Skript stammt (Christa
./kapitel9.txt:/home/christa zu groß: 567 Kb
./kapitel11.txt:buch@koala:/home/buch > echo ~christa
./kapitel12.txt: "/home/buch/skript" "/home/buch/Christa und Bettina"
...
```

8.6 Subshells

```
( <befehl1> ; <befehl2> ; ... )
```

Befehle lassen sich mit {} zu Gruppen zusammenfassen, wodurch Funktionen erst einen Sinn bekommen. Diese Befehlsgruppen laufen in der Umgebung der Shell ab, welche das Skript ausführt.

Sie können aber auch Befehle durch runde Klammern gruppieren. Dadurch wird eine Subshell gestartet, welche die geklammerten Befehle ausführt. Einfach ausgedrückt, bedeutet dies, dass ihr Skript eine neue Shell startet, die das aktuelle Environment übernimmt, die angegebenen Befehle ausführt und sich anschließend beendet. Der Rückgabewert einer Subshell ist der Exitstatus des letzten in der Subshell ausgeführten Befehls.

Der Vorteil von Subshells liegt darin, dass sämtliche Änderungen an Variablen und Environment nach Beendigung der Subshell verloren gehen. Auch die Shell selbst nutzt diese Technik aus, z.B. bei den Here-Documents und u.U. auch bei `while`-Schleifen.

Bitte beachten Sie, dass Leerzeichen zwischen den Klammern {} und den Befehlen stehen müssen, damit die Shell die Befehle als Gruppe und nicht als Brace Extension interpretiert und damit wahrscheinlich auf einen Fehler fällt.

Hier ein kleines Beispiel, das die Unterschiede zwischen *normalen* Skriptbefehlen, gruppierten Befehlen und Subshells demonstriert:


```
# Skript showsub.sh
# Zeigt den Unterschied zwischen () und {}
#
# 1. "Normale" Befehle, die Änderungen bleiben erhalten
#     betroffen: lvAnz und Verzeichnis
cd ..
echo "Verzeichnis ist `pwd`"
typeset -i lvAnz=`ls |wc -l`
echo "ergebnis=$lvAnz"
echo "Nun {}"
# 2. Änderungen in { } wirken sich ebenfalls auf die
#    Shell aus.
{ cd $HOME
  echo "Verzeichnis ist `pwd`"
  lvAnz=lvAnz+`ls | wc -l`
  echo "Ergebnis in {} = $lvAnz"
}
# 3. Hier der Beweis, dass pwd und lvAnz geändert bleiben
#
```

```
echo "Verzeichnis nach {} =`pwd`"
echo "Vor Subshell Ergebnis=$lvAnz"
# 4. Und nun in der Subshell
( cd ..
  echo "Verzeichnis ist `pwd`"
  lvAnz=lvAnz+`ls | wc -l`
  echo "Ergebnis in() = $lvAnz"
)
# 5. Der Beweis, dass die Änderungen der Subshell
#    hier nicht mehr Bestand haben
echo "Verzeichnis ist `pwd`"
echo "Ergebnis am Ende=$lvAnz"
exit 0
```

Dieses Skript gibt Folgendes aus:

```
buch@koala:/home/buch > ./showsub.sh
Verzeichnis ist /home
ergebnis=4
Nun {}
Verzeichnis ist /home/buch
Ergebnis in {} = 79
Verzeichnis nach {} =/home/buch
Vor Subshell Ergebnis=79
Verzeichnis ist /home
Ergebnis in() = 83
Verzeichnis ist /home/buch
Ergebnis am Ende=79
buch@koala:/home/buch >
```

8.7 Skript im Skript einlesen: . oder source

Mittlerweile haben Sie schon viele schöne Tricks und Techniken kennen gelernt, und viele Befehlskonstrukte, die Ihnen am Anfang des Buches Angstschweiß auf die Stirn getrieben hätten, sind nun bestenfalls eine Beleidigung Ihrer Intelligenz. Aber die Veränderung der Umgebung Ihrer Loginshell entzieht sich bisher hartnäckig allen Ihren Versuchen. Dass dies gehen muss, wissen Sie bereits, werden doch die Einträge für Variablen wie `PATH` und `TERM` nicht von der Shell gesetzt, sondern haben lediglich eine besondere Bedeutung.

Beim Start der Loginshell starten zwei Skripten, unter `sh`, `ksh` und `bash` sind dies `/etc/profile` und danach `.profile` aus dem Heimatverzeichnis des angemeldeten Benutzers. Diese Skripten tragen zuerst die systemweiten Vorga-

ben in die aktuelle Shellumgebung ein, bevor `.profile` die persönlichen Anpassungen für den Benutzer vornimmt. Erwartungsgemäß darf nur der Superuser `root` die systemweiten Vorgaben in `/etc/profile` verändern, während der Benutzer seine Einstellung durch `.profile` verändern kann.

Dies ist die Reihenfolge bei `sh` und `ksh`. Die Bash allerdings bietet mehr Möglichkeiten, die hier kurz aufgeführt werden sollen. Weitere Informationen finden Sie dazu in *bash(1)*, Stichwort *Invocation*.

Eine Loginshell ist eine Shell, deren Name mit »-« beginnt (siehe Beispiel zu `ps` weiter oben: `-bash`) oder die mit der Option `--login` aufgerufen wurde. Eine interaktive Shell kann mit der Option `-i` gestartet werden und verwendet Tastatur und Bildschirm als Ein-/Ausgabekanäle. Ist die Shell interaktiv, so wird `PS1` gesetzt, und `$-` enthält ein `i`.

Eine Loginshell liest zunächst `/etc/profile` und führt deren Befehle aus. Danach werden (falls vorhanden und in dieser Reihenfolge) `.bash_profile`, `.bash_login` und `.profile` gelesen und deren Befehle ausgeführt. Die Dateien werden im Heimatverzeichnis des aktuellen Benutzers gesucht. Die Option `--noprofile` unterdrückt dieses Verhalten.

Eine interaktive Shell führt beim Start Befehle aus, die in der Datei `.bashrc` aufgeführt sind, falls diese Datei im Heimatverzeichnis vorhanden ist. Die Option `--norc` unterdrückt dieses Verhalten.

Wird eine Loginshell beendet, und im Heimatverzeichnis des Benutzers existiert die Datei `.bash_logout`, so wird diese ausgeführt.

Sämtliche Dateien müssen ausführbar sein, sonst gibt die Bash eine Fehlermeldung aus. Dieses Verhalten ist das Defaultverhalten der Bash. Wird die Bash als `sh` oder mit der Option `--posix` gestartet, so ändert sich das Verhalten.

Die Änderungen an der Umgebung kommen nicht zum Tragen, da der Aufruf eines Skripts dazu führt, dass erst eine neue Shell (als Kindprozess) gestartet wird, welche dann die Steuerung und Ausführung des angegebenen Skripts übernimmt. Diese Umgebung geht allerdings mit der Beendigung der Shell verloren. Ziel muss es also sein, dass das Skript nicht in einer neuen Shell ausgeführt wird, sondern direkt in der aktuellen Loginshell.

Dies können Sie auf zwei verschiedene Arten erreichen: Entweder Sie rufen das Skript als Parameter des Befehls `source` auf, oder Sie setzen einfach einen Punkt und ein Leerzeichen vor dem Skriptnamen: »`. `«.

```
buch@koala:/home/buch > source /etc/profile
buch@koala:/home/buch > . .profile
```

Obiges Beispiel führt nochmals alle Befehle für die Initialisierung der Shellumgebung aus, die auch bei der Anmeldung durchlaufen werden. Nicht unbedingt eine gute Idee, aber es würde klappen.

8.8 Jobverwaltung


```
fg %<job>
bg %<job>
suspend
jobs <opt> [<job>]
```

Autoren, Lektoren und Verlag haben alle einen schweren Job hinter sich gebracht, wenn Sie dieses Buch in den Händen halten. Jedoch hat jede Partei eine andere Meinung, wer den schwersten Job von allen hatte, und dabei haben wir Sie noch gar nicht berücksichtigt, wo Sie doch den schweren Job haben, das Buch durchzuarbeiten.

Aber auch die Shell hat Jobs, und damit wollen wir uns in dem letzten Abschnitt dieses langen Kapitels beschäftigen.

In der Bash ist Jobcontrol in der Regel nur für Loginshells (auch interaktive Shells genannt) aktiviert. Sie können jedoch mit einem einfachen `set -m` die Jobverwaltung auch für Ihr Skript aktivieren. Da im Parameter `$-` alle für das aktuelle Skript/Bash gesetzten Optionen aufgeführt sind, führt die Aktivierung der Jobs dazu, dass mindestens ein `m` vom Parameter ausgegeben wird.

Mithilfe der Jobverwaltung können Sie von der Shell aus gestartete Prozesse vom Vordergrund in den Hintergrund legen, Prozesse anhalten (sprich ihnen `SIGSTOP` schicken), Prozesse beenden oder Hintergrundprozesse wieder in den Vordergrund holen. Sie haben also die vollkommene Kontrolle über Ihre Prozesse.

Wenn Sie einen Prozess mit `&` in den Hintergrund legen, so gibt die Shell folgendes aus:

```
buch@koala:/home/buch > ls -l *.txt&
[1] 753
-rw-r--r--   1 buch     users         995 Mar  5 17:53 anhanga.txt
-rw-r--r--   1 buch     root         4178 Feb  6 17:42 einleitung.txt
-rw-r--r--   1 buch     users       22869 Mar  3 21:49 kapitel1.txt
-rw-r--r--   1 buch     users       33436 Mar  8 21:18 kapitel2.txt
-rw-r--r--   1 buch     users       41562 Mar  8 21:18 kapitel3.txt
-rw-r--r--   1 buch     users       30048 Feb 23 20:55 kapitel4.txt
```

```
-rw-r--r--   1 buch     users       31716 Mar  5 22:53 kapitel5.txt
-rw-r--r--   1 buch     users       41530 Mar  6 12:36 kapitel6.txt
-rw-r--r--   1 buch     users       30038 Mar  8 23:12 kapitel7.txt
-rw-rw-rw-   1 buch     users       42164 Mar 14 02:06 kapitel8.txt
-rw-r--r--   1 buch     users        3157 Mar 14 02:19 toc.txt
[1]+  Done                    ls --color=tty -l *.txt
buch@koala:/home/buch >
```

Dabei bedeutet `[1] 753`, dass die Shell den Befehl als Job 1 mit der Prozess-ID 753 in den Hintergrund gelegt hat. Sie können aber auch bereits im Vordergrund laufende Prozesse in den Hintergrund legen – eine sehr nützliche Eigenschaft der Shell, wenn das Skript länger läuft, als ursprünglich erwartet und Sie den Bildschirm dringend benötigen. Eine Möglichkeit wäre, von einem zweiten Bildschirm ein `SIGSTOP` an die Shell zu schicken, welche das Skript ausführt. Nach Murphys Gesetz ist aber genau dann kein zweiter Bildschirm verfügbar, wenn er am nötigsten gebraucht wird. Drücken Sie also Strg+Z, und das Skript wird angehalten:

```
buch@koala:/home/buch > du -a / 2>/dev/null | wc -l
[1]+  Stopped                 du -a / 2>/dev/null | wc -l
buch@koala:/home/buch >
```

Die Tastenkombination Strg+Z ist der Standard. Allerdings können Sie mittels *stty(1)* die Funktion `susp` auch anders belegen.

Gestoppte Prozesse tauchen in der Prozessliste von `ps` als `T` (steht für *Traced*) auf.

Der aufgerufene Befehl ist jetzt angehalten und wartet darauf, dass ihm signalisiert wird, weiterarbeiten zu dürfen. Sie haben jetzt drei Möglichkeiten:

- Den Befehl im Vordergrund weiterlaufen lassen.

  Die Shell bietet dazu einen Befehl namens `fg` an. Es überrascht Sie wohl kaum, wenn ich darauf hinweise, dass dies eine Abkürzung für das englische *foreground* ist, welches nichts anderes als Vordergrund bedeutet. Als Parameter geben Sie die Jobnummer an. Damit die Shell erkennt, dass es sich um eine Jobnummer handelt, müssen Sie ein Prozentzeichen vor die Nummer setzen:

  ```
  buch@koala:/home/buch > fg %1
  ```

  Dies lässt den Prozess im Vordergrund weiterlaufen.

- Den Befehl im Hintergrund laufen lassen.

  Möchten Sie den Prozess im Hintergrund laufen lassen, sodass Sie in der Shell weiterarbeiten können, so schicken Sie den Job mittels `bg` (*Background* = Hintergrund) in den Hintergrund:

```
buch@koala:/home/buch > bg %1
[1]+ du -a / 2>/dev/null | wc -l &
buch@koala:/home/buch >
```

  Bitte beachten Sie aber, dass der Job automatisch gestoppt wird, wenn er eine Eingabe von der Tastatur benötigt und dass die Ausgaben (solange nicht umgelenkt) immer noch auf den Bildschirm gehen und sich mit den Ausgaben Ihrer aktuellen Arbeit vermischen.

- Den Prozess beenden.

  Den Prozess können Sie natürlich auch beenden, und zwar mit dem Befehl `kill` gefolgt von `%` und Jobnummer.

Alle Befehle geben so lange eine `0` als Exitstatus zurück, wie `<job>` eine gültige Jobnummer ist.

Worin liegt nun der Unterschied zwischen einem Prozess und einem Job? Solange Sie nur einen einzigen Befehl aufrufen, gibt es keine Unterschiede. Ein Job ist dann ein Prozess, der von der Shell gestartet wurde und zusätzlich zur Prozess-ID noch eine Jobnummer zugeteilt bekommen hat, mit der er sich für die Befehle `fg`, `bg`, `kill` usw. eindeutig identifizieren lässt.

Die Job-IDs sind übrigens shellspezifisch, d.h., die betroffene Shell vergibt diese ID nach eigenen Regeln. Daher ist die Job-ID zum einen nicht identisch mit der Prozess-ID und zum anderen nicht systemweit eindeutig. Es kann also durchaus sein, dass zwei Shells die gleiche Job-ID vergeben.

Etwas anders sieht die ganze Sache aus, wenn Sie eine Pipe in den Hintergrund legen. Jetzt ist die Jobnummer immer noch eindeutig, allerdings für alle Befehle innerhalb der Pipe gültig. Das heißt, wenn Sie ein Signal an die Shell schicken, so wird in Wirklichkeit dieses Signal an alle Prozesse innerhalb des Jobs geschickt. Einen Job kann man sich als eine Art Klammer vorstellen, die eine bestimmte Anzahl Befehle/Prozesse zu einer Gruppe zusammenfasst.

So weit, so gut. Laufen Ihre Prozesse noch, die Sie testweise schon mal gestartet hatten? Das wissen Sie nicht? Nun, dann wird es Zeit für den letzten Befehl in diesem Kapitel: `jobs -l`. Dieser gibt alle Jobs aus, die entweder noch laufen oder auf ein `SIGCONT` warten. Dabei unterteilt `jobs` die Jobs nach Prozessnummern:

```
buch@koala:/home/buch > du -a / 2>/dev/null | wc -l
[1]+  Stopped                 du -a / 2>/dev/null | wc -l
buch@koala:/home/buch > jobs -l
[1]+   585 Stopped                 du -a / 2>/dev/null
       586                       | wc -l
buch@koala:/home/buch > du -a / 2>/dev/null | wc -l
[2]+  Stopped                 du -a / 2>/dev/null | wc -l
buch@koala:/home/buch > jobs -l
[1]-   585 Stopped                 du -a / 2>/dev/null
       586                       | wc -l
[2]+   592 Stopped                 du -a / 2>/dev/null
       593                       | wc -l
buch@koala:/home/buch >
```

Sie können die Ausgabe von `jobs -l` auch weiter einschränken. Wenn Sie die Option `-lr` angeben, so werden nur alle Jobs ausgegeben, die gerade laufen. Konträr dazu gibt die Option `-ls` nur alle gestoppten Prozesse aus.

Und `jobs -n` gibt alle Jobs aus, deren Status sich geändert hat, seitdem der Benutzer das letzte Mal über ihren Status informiert wurde.

Falls Sie Ihr Skript anhalten möchten, wenn eine Bedingung innerhalb des Skripts erfüllt ist, so können Sie `suspend` aufrufen. Dieser Befehl sendet praktisch ein `SIGSTOP` an Ihr Skript. Der Effekt ist klar: Ihr Skript wird angehalten.

Das obige Beispiel macht nicht allzuviel Sinn, soll aber gleich noch auf den letzten Punkt aufmerksam machen, der dieses Kapitel abschließen soll. Wie Sie sehen, werden hinter den Jobnummern ein »+«- oder ein »-«-Zeichen ausgegeben. Ein Pluszeichen bedeutet, dass dies der aktuelle Job ist, während ein Minuszeichen bedeutet, dass dies der vorherige Job ist. In der Ausgabe können Sie sehen, dass Job 1 zunächst der aktuelle Job ist. Nachdem der zweite Job gestartet und angehalten wurde, wird Job 2 zum aktuellen Job und Job 1 zum vorherigen Job.

Sie können in den Befehlen, die eine Jobnummer akzeptieren, auch ein `%+` oder `%%` für den aktuellen Job und ein `%-` für den vorherigen Job einsetzen. Bitte beachten Sie aber, dass Job 1 nicht wieder zum aktuellen Job wird, wenn Job 2 beendet wurde. Er bleibt der vorherige Job. Erst der nächste gestartete Job wird wieder zum aktuellen Job und bekommt wieder das »+«-Zeichen!

Diese Informationen finden Sie in ähnlicher Form auch in den Manpages, aber was können Sie damit anfangen? Angenommen, Sie haben nur einen Bildschirm zur Verfügung, weil Sie z.B. per `telnet` auf Ihrem Unixrechner von Ihrem Windowsclientrechner aus arbeiten und nicht mehrere Telnetsitzungen gleichzeitig öffnen können.

Jetzt wollen Sie einen Text im Editor schreiben, brauchen dazu aber Informationen aus einer Manpage. Was tun? Sie könnten den Editor verlassen und die Manpage aufrufen, aber wenn Sie häufiger wechseln (müssen), müssen Sie jedes Mal die Stelle im Editor oder in der Manpage suchen. Mit Strg+Z geht das aber auch:

```
buch@koala:/home/buch > man bash
```

<Ausgabe man>

```
<CTRL>+<Z>
[1]+  Stopped                 man bash
buch@koala:/home/buch > vi meinskript.sh
```

<Editorausgaben>

```
<CTRL>+<Z>
[2]+  Stopped                 vi meinskript.sh
buch@koala:/home/buch > jobs
[1]-  Stopped                 man bash
[2]+  Stopped                 vi meinskript.sh
buch@koala:/home/buch > fg %-
```

<Ausgabe man, so wie oben abgebrochen>

```
<CTRL>+<Z>
[1]+  Stopped                 man bash
buch@koala:/home/buch > fg %-
```

<Editorausgaben>

```
[2]+  Stopped                 vi meinskript.sh
buch@koala:/home/buch >
```

Wurde der Job beendet, so gibt die Shell bei nächster Gelegenheit einen passenden Hinweis aus. Dieser Hinweis enthält die Jobnummer, den Status und den Shellbefehl, der durch den Job abgearbeitet wurde.

```
buch@koala:/home/buch > du -a $HOME 2>/dev/null | wc -l &
[1] 671
    115
[1]+  Done                    du -a $HOME 2>/dev/null | wc -l
buch@koala:/home/buch >
```

Neben dem Strg+Z gibt es auch noch ein Strg+Y. Wenn die Shell oder ein Befehl über die Standardeingabe von der Tastatur liest, dann stoppt Strg+Y ebenfalls den aktuellen Prozess.

8.9 Aufgaben

1. Worin liegt der Unterschied zwischen:

   ```
   trap 'echo $RANDOM' SIGINT
   ```

   und

   ```
   trap "echo $RANDOM" SIGINT
   ```

2. In Kapitel 4 hatten wir ein Skript entwickelt, welches mit dem `tar`-Befehl ein Archiv anlegt und gleichzeitig einen Fortschrittsbalken von 0% bis 100% ausgibt. Stellen Sie das Skript so um, dass `tar` im Hintergrund läuft.

3. Können die bisherigen Skripten alle ohne Probleme mittels `source` aufgerufen werden, oder sind Probleme zu erwarten?

4. `ps` kann auch einige Wahrheiten (?) über andere Dinge als nur Prozesse ausgeben. Es ist Ihnen ja sicherlich bekannt, dass ein PS: unter einem Brief für Postskriptum (Nachtrag) steht. Was macht also (GNU-) `ps` bei einem Brief?

   ```
   buch@koala:/home/buch > ps Brief
   unrecognized option or trailing garbage
   usage:  ps acehjlnrsSuvwx{t<tty>|#|O[-]u[-]U..} \
           --sort:[-]key1,[-]key2,...
           --help gives you this message
           --version prints version information
   buch@koala:/home/buch >
   ```

   Unerkannte Option oder nachfolgender Müll. Nachfolgender Müll? So kann man ein PS in einem Brief natürlich auch bezeichnen.

8.10 Lösungen

1. Der Unterschied wird sichtbar bei mehrmaligem Aufrufen der beiden Befehle. Im Falle der Anführungszeichen ersetzt die Shell `$RANDOM` durch eine Zufallszahl, sodass `echo` immer mit dieser Zahl aufgerufen wird. Die Apostrophe unterbinden die Interpretation durch die Shell. Dadurch wird immer `echo $RANDOM` ausgeführt.

2. Um den `tar`-Befehl in den Hintergrund zu legen, kann einfach `&` genutzt werden. Die eigentliche Schwierigkeit liegt darin, den Fortschrittsbalken zu berechnen. `tar` füllt im Hintergrund ein temporäre Datei, und der Vordergrundprozess liest diese Datei aus, um den Fortgang der Archivierung zu bestimmen.


```
# Skript 25: tput und printf
#    Diese Version speichert alle Dateien im
#    Verzeichnis $1 in das Archiv /tmp/bettina.tar
#    Neu ist die flexible Ausgabe der Box und
#    dass der Balken komplett gefüllt wird, wenn
#    wenige Dateien zu sichern sind und keine
#    "Löcher" hat
#
# Box ausgeben
tput clear
#
# Bildschirmzeilen und -spalten merken und so
# verschieben, dass zentriert ausgegeben wird
# (Aufgabe 6)
#
zeilen=`tput lines`
spalten=`tput cols`
x=`expr \( $spalten - 54 \) / 2`
y=`expr \( $zeilen - 4 \) / 2`
#
# Alle fixierten tputs entsprechend abändern:
#
tput cup $y $x
echo "+--------------------------------------------------+"
tput cup `expr $y + 1` $x
printf "! %50s!" " "
tput cup `expr $y + 2` $x
printf "! %50s!" " "
tput cup `expr $y + 3` $x
echo "+--------------------------------------------------+"
# Anzahl Dateien im Verzeichnis ermitteln
verz=$1
if [ ! -d "$1" ] ; then
   echo "Aufruf: $0 Verzeichnis"
   echo "  Verzeichnis  --> Das Verzeichnis, welches gesichert werden soll"
   exit 0
fi

anz=`find $verz -type f -print | tee /tmp/tar.cnt | wc -l`
# tar soll beim 1. Aufruf Archiv anlegen
taropt="cvf"
akt=0
while [ $anz -gt $akt ] ; do
  # Aktuelle Zeile ermitteln, entspricht Dateinamen
  akt=`expr $akt + 1`
  dnam=`head -$akt /tmp/tar.cnt | tail -1`
  # Ins Archiv damit
  tar $taropt /tmp/bettina.tar $dnam >/dev/null 2>&1
```

```
  # Beim nächsten Aufruf Option Datei anhängen
  taropt="rvf"
  #
  # Spalte, sodass erg in der Mitte des
  # Fensters im Titel erscheint
  #
  tput cup $y `expr $x + 26`
  erg=`expr  $akt \* 100 / $anz`
  echo "$erg%"
  #
  # Anzeige der Dateinamen in der
  # linken Ecke des Fensterinneren
  #
  tput cup `expr $y + 1` `expr $x + 2`
  printf "%-40.40s" $dnam
  #
  # Ausgabe des Fortgangs in der mittleren Zeile
  # des Statusfensters
  #
  pos=`expr $erg / 2 + 14`
  tput cup `expr $y + 2` $pos
  echo -n ":"
  sleep 1
done
rm /tmp/tar.cnt
tput cup 24 0
exit 0
```

3. Ein Aufruf kann erfolgen, aber beim Beenden der Skripten wird die aktuelle Shell mit beendet. Sollte dies Ihre Loginshell sein, stehen Sie wieder in der Anmeldung.

Kapitel 9

jetzt lerne ich

Befehlslisten und sonstiger Kleinkram

»Wir dürfen nicht alles so hochsterilisieren« –

Bruno Labadia

Das tun wir auch nicht, aber um dieses Kapitel kommen wir trotzdem nicht herum.

In diesem Kapitel wollen wir uns noch mit einigen Eigenheiten der Shell beschäftigen, die wir bisher noch nicht besprochen hatten. Hinzu kommen noch einige Themen, die wir bereits angerissen oder unter anderen Vorzeichen kennen gelernt hatten.

9.1 Befehlslisten

```
<befehl1> ; <befehl2> ...
```

In den vorherigen Kapiteln haben wir Befehlslisten schon häufig zum Einsatz gebracht, jedoch nicht ausführlich theoretisch betrachtet. Dies soll in diesem Abschnitt nachgeholt werden.

Eine Liste ist eine Folge von Befehlen, die durch einen der Operatoren **;**, **&**, **&&** bzw. **||** voneinander getrennt werden. Eine Befehlsliste kann optional durch ein **;**, **&** oder Zeilenumbruch beendet werden.

Dabei sind die gleichwertigen Operatoren || und && mit einer höheren Priorität versehen als die ebenfalls gleichwertigen Operatoren ; und &. Durch die Priorität wird sichergestellt, dass ein & einem Befehl zugeordnet werden kann und die Befehle in einer UND- bzw. ODER-Liste korrekt erkannt werden können.

Wird ein Befehl mit einem »&« abgeschlossen, so wird dieser Befehl in einer Subshell ausgeführt, und die Shell führt sofort den nächsten Befehl aus. Werden Befehle nur mit Semikola getrennt, so wird <befehl1> ausgeführt und beendet, bevor <befehl2> gestartet werden kann.

Wird die Liste in runden Klammern eingeschlossen (), so wird eine Subshell gestartet, welche eine Kopie der aktuellen Shellumgebung erhält.

Im Gegensatz dazu stehen Gruppenbefehle, die in {} eingeschlossen sind. Diese werden in der aktuellen (Shell-)Umgebung ausgeführt und können diese natürlich beeinflussen. Die Funktionen aus Kapitel 7 sind dafür das beste Beispiel.

Der Rückgabewert einer Liste ist identisch mit dem Rückgabewert des letzten Befehls in der Liste.

9.2 UND-Listen


```
<befehl1> && <befehl2>
```

Unser aktueller Wissensstand führt zu sehr umständlichen Konstrukten, wenn es um Befehle geht, die abhängig vom Erfolg eines weiteren Befehls gestartet werden sollen. Die einzige sinnvolle Methode sähe bisher ähnlich wie folgendes Skriptstückchen aus:

```
...
rm -f /tmp/test.dat 2>/dev/null >/dev/null
if [ $? -eq 0 ] ; then
   # Falls löschen ok, dann
   echo "Datei wurde gelöscht"
fi
...
```

Dieses Skript beschreibt aber nur eine Abhängigkeit, die auf eine Ebene beschränkt ist. Je mehr Abhängigkeiten aber hinzukommen, desto unübersichtlicher wird die ganze Sache.

Sicher wünschen Sie sich mittlerweile eine einfachere und kürzere Methode, solche Überprüfungen durchzuführen. Diese Alternative lernen Sie jetzt kennen: die UND-Listen.

Eine UND-Liste sind zwei (oder mehr) Befehle, die durch ein && getrennt werden. Dabei wir zunächst <befehl1> ausgeführt und beendet. Ist der Rückgabewert von <befehl1> gleich 0, so wird <befehl2> ausgeführt. In jedem anderen Fall wird der nächste Befehl nach der UND-Liste ausgeführt.

Unser Skript von oben sieht als UND-Liste wie folgt aus:

```
...
rm -f /tmp/test.dat 2>/dev/null >/dev/null && echo "Datei wurde gelöscht"
...
```

Sie können auch mehr als zwei Befehle miteinander durch && verknüpfen. In einer UND-Liste von drei Befehlen müssen somit die Befehle eins und zwei beide mit dem Rückgabewert 0 beendet werden, damit der dritte Befehl gestartet werden kann. Allgemein formuliert, gilt: Damit Befehl n in einer UND-Liste ausgeführt werden kann, müssen die Befehle 1 bis n-1 in der UND-Liste alle den Rückgabewert 0 haben.

9.3 ODER-Listen

```
<befehl1> || <befehl2>
```

Das hilft uns jetzt schon ganz erheblich weiter, aber es gibt ja auch Umstände, die es erfordern, dass ein Befehl ausgeführt werden soll, wenn etwas schief gegangen ist, also ein Fehler auftrat. Auch hier ist es nötig, den Rückgabewert auf 0 zu testen, um im negativen Fall eine Fehlermeldung oder Fehlerbehandlung durchzuführen.

Der erwachende Skriptprogrammierer in Ihnen sollte aber Folgendes signalisieren: umständlich! Da Sie sicherlich nicht mehr bereit sind, umständliche Skripten zu formulieren, ist es an der Zeit, zu den ODER-Listen zu kommen.

ODER-Listen sind zwei (oder mehr) Befehle, die durch || getrennt werden. Dabei wird zunächst <befehl1> ausgeführt und der Rückgabewert betrachtet. Konträr zu den UND-Listen wird hier aber <befehl2> nur dann ausgeführt, wenn dieser ungleich 0 ist. Das ist gleichbedeutend mit der Tatsache, dass <befehl1> auf einen Fehler lief.

Durch dieses Verhalten eignen sich ODER-Listen hervorragend, um Fehlerbehandlungen aufzurufen.

Nutzen wir unser neues Wissen, um den CW-Commander wieder ein wenig zu erweitern. Zum einen sollten wir eine neue Funktion hinzufügen: das Löschen der markierten Dateien mittels Eingabe von d. Kann eine Datei nicht gelöscht werden, so sollte in der Statuszeile ein Fehlerhinweis ausgegeben

werden. Gleiches gilt auch, wenn es nicht möglich ist, in ein ausgewähltes Verzeichnis durch Drücken von ↵ zu wechseln.

Schreiben wir eine Funktion, welche eine Meldung in der Statuszeile ausgibt und nach ca. vier Sekunden wieder löscht. Außerdem setzen wir eine ODER-Liste für das Wechseln der Verzeichnisse ein und schreiben eine zweite Funktion, die das Löschen der markierten Dateien übernimmt.

An dieser Stelle zeige ich Ihnen aus Platzgründen nur die Änderungen zur letzten Version des CW-Commanders. Am Ende des Kapitels findet sich aber die aktuelle Version mit allen Änderungen, die wir an diesem Skript bis zum Kapitelende vorgenommen haben.


```
# Skript 45:
#
# CW-Commander: Änderungen zu Skript 44
# Als da wären: Fehlerausgabe in Zeile 0
#               Löschen markierter Dateien
#
function CW_Error()
{
 tput cup 0 0
 tput el     # Lösche den Rest der Zeile
 CW_Print 0 0 "$1"
}
function CW_Delete ()
{
 local lvaifs="$IFS"
 IFS="$IFS:"
 for lvDatei in `echo $1 |cut -c2-` ; do
    rm -f $1 2>/dev/null >/dev/null || \
       CW_Error "Fehler: \"$lvDatei\" kann nicht gelöscht werden"
 done
 IFS="$lvaifs"
}
function CW_Eingabe()
{
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
                 offset=offset+17
                 if [ $offset -gt $anz ] ; then
                    offset=offset-17
                  else
                    zakt=zakt+17
```

```
                  if [ $zakt -gt $anz ] ; then
                     zakt=anz
                  fi
               fi
            ;;
     "z" | "Z")
               if [ $offset -gt 2 ] ; then
                  offset=offset-17 ;
                  zakt=zakt-17
               fi
            ;;
     "n" | "N") if [ $zakt -lt $anz ] ; then
                  zakt=zakt+1
                  if [ $zakt -gt $((offset+17)) ] ; then
                     offset=offset+17
                  fi
               fi ;;
     "l" | "L") if [ $zakt -gt 2 ] ; then
                  zakt=zakt-1
                  if [ $zakt -lt $offset ] ; then
                     offset=offset-17
                  fi
               fi ;;
     "m" | "M") CW_Mark $tmpfile
              ;;
     "a" | "A") CW_Archiv "$Mark"
               ;;
     "d" | "D") CW_Delete "$Mark"
               ;;
     "")       local line=`head -$zakt $tmpfile|tail -1`
               set -- $line
               local ch=`echo  $1 | cut -c1`
               if [ "$ch" = "d" ] ; then
                  ein="CR"
                  Mark=":"
                  CW_ReadDir $9;
                else
                  CW_NormalFile $9
               fi ;;
      *) CW_Print 0 0 ">>$Mark<<"
         ein="" ;;
    esac
  done
}
```

9.4 Rückgabewert negieren


```
! <befehl>
```

Wie bereits in Kapitel 2 angerissen, können Sie den Rückgabewert eines Befehls negieren. Die Shell interpretiert einen Rückgabewert von `0` ja als Erfolg (boolescher Wert: wahr) und jeden anderen Wert als Fehler (oder falsch).

Wenn Sie vor dem Befehl ein Ausrufungszeichen `!` setzen, so dreht die Shell die Logik um, und aus wahr wird falsch und umgekehrt.

Einige Beispiele:


```
# Skript 46: Negierung von Rückgabewerten
#            und ODER-Listen
#
cd /FalschesVerzeichnis   2>/dev/null   # Existiert nicht
echo "Rückgabewert war=$?"
! cd /FalschesVerzeichnis 2>/dev/null   # Existiert immer noch nicht
echo "Rückgabewert jetzt=$?"
! cd /tmp  || echo "Aktuelles Verzeichnis=`pwd`"
exit 0
```

9.5 Arithmetische Auswertung mittels let


```
let <ausdruck>
```

In Kapitel 6 hatten wir im Zusammenhang mit dem Variablentyp Ganzzahl (Integer), die Möglichkeit kennen gelernt, arithmetische Ausdrücke für die Wertzuweisung zu nutzen, um dadurch den Gebrauch von `expr` überflüssig zu machen.

Die arithmetische Auswertung mittels `let` funktioniert sehr ähnlich. Auch hier sind alle Operatoren wie in Kapitel 6 aufgeführt erlaubt. Während wir in Kapitel 6 diese Methoden zur Wertzuweisung für Variablen genutzt haben, prüft `let` das Ergebnis des arithmetischen Ausdrucks und gibt davon abhängig entweder eine `0` (Ausdruck wahr) oder eine `1` (Ausdruck falsch) zurück.

Schreiben wir doch ein kleines Skript, welches als Parameter ein Verzeichnis und eine Größe entgegennimmt. Dabei gibt das Skript alle Dateien im Verzeichnis aus, welche größer als die angegebene Größe sind. Wir nutzen dabei `let`, um die Größe abzufragen, und verzichten somit auf das `if`.


```
# Skript 47: Demo für let
#  Gibt alle Dateien im Verzeichnis $1 aus, die
#  größer als $2 Bytes (Vorgabe: 1000) sind.
#
# 1. Parameter und Konstanten
pgVerz=${1:-`pwd`}
pgSize=${2:-1000}   # Als Default: Alle Dateien > 1000 Bytes ausgeben
gcTmp=/tmp/erg$$
# 2. Initialisierung Eingabedatei
ls -l $pgVerz 2>/dev/null >$gcTmp
exec 4<$gcTmp
# 3. Erste Zeile lesen
read gvZeile <&4   # Überlese total 1234
read gvZeile <&4   # Erste echte Zeile
while [ "$gvZeile" != "" ] ; do
  # 4. Parameter neu setzen
  set -- $gvZeile
  # 5. let gibt 0 zurück, wenn Bedingung ok
  let "$5 > $pgSize" && echo "$9 zu groß: $5"
  read gvZeile <&4
done                # while gvZeile ungleich ""
rm $gcTmp
exit 0
```

Die Anführungszeichen beim `let` sind notwendig, um die Ausgabeumlenkung zu unterbinden.

Zum Abschluss noch einmal ein Beispiel im CW-Commander. Dieser ist beim Blättern doch recht langsam, baut er doch jedes Mal die Seite komplett neu auf, selbst wenn nur eine Zeile vor- oder zurückgeblättert wird. Sinnvoller wäre es, nur die geänderten Zeilen auszugeben.

Wann also müssen jetzt Zeilen ausgegeben werden?

- Wenn der Offset der Seite sich seit der letzten Ausgabe verändert hat, müssen alle Zeilen auf der Seite ausgegeben werden.
- Ansonsten werden alle Zeilen auf der Seite von `offset` bis `offset+17` durchlaufen. Diese nennen wir mal Anzeigezeile. Ist die Anzeigezeile gleich der alten aktuellen Zeile, so muss sie erneut ausgegeben werden. Dadurch wird die Zeile wieder in normaler Helligkeit ausgegeben. Außerdem muss die Zeile ausgegeben werden, wenn die aktuelle Zeile gleich der Anzeigezeile ist.

Diese Überlegungen führen dazu, dass wir zwei neue Variablen benötigen, in denen wir die alten Werte für den Seitenoffset (`gvAltOffs`) und die alte aktuelle Zeile (`gvAltZeil`) abspeichern. Diese Variablen müssen am Programmstart mit Werten belegt werden, die weder Punkt 1 noch Punkt 2 genügen und somit die Ausgabe erzwingen.

Ansonsten muss jede Änderung über die Cursortasten sich in diesen Variablen widerspiegeln.


```
# Skript 45:
#
# CW-Commander mit let
function CW_Error()
{
 tput cup 0 0
 tput el
 CW_Print 0 0 "$1"
}
function CW_Delete ()
{
 local lvaifs="$IFS"
 IFS="$IFS:"
 for lvDatei in `echo $1 |cut -c2-` ; do
    rm -f $1 2>/dev/null >/dev/null || \
       CW_Error "Fehler: \"$lvDatei\" kann nicht gelöscht werden"
 done
 IFS="$lvaifs"
}
function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     # Tarsicherung abbrechen
     kill -SIGTERM $prid 2>/dev/null
     rm -f $tmptar
     rm -f $tmptarcnt
     prid=""
     CW_Print 0 0 "Sicherung abgebrochen!"
    else
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     echo
     exit 0
  fi
}
function CW_Archiv ()
{
  fnamen=$1
  sleep 1
  if [ -n "$fnamen" ] ; then
     fnamen=`echo "$fnamen" | tr "/" " "`
     typeset -i pro=`echo "$fnamen" |wc -w`
     typeset -i anz=100*pro
     tar cvfz $tmptar $fnamen >$tmptarcnt 2>/dev/null &
     typeset -i akt=0
```

```
     typeset -i erg=1
     prid=$!
     while [ $anz -gt $akt ] ; do
       akt=`cat $tmptarcnt |wc -l`
       akt=akt*100
       tput cup 10 38
       erg=akt/pro
       CW_Print 0 0 "$erg%"
     done
     rm "$tmptarcnt"
     prid=""
  fi
}
function CW_Mark ()
{
 # 1. Schon Eingetragen?
 local line=`head -$zakt $1 |tail -1`
 set -- $line
 local name="$9"
if [ -z "`echo $Mark | grep \"/$name/\"`" ] ; then
     Mark="$Mark$name/"
else
    aifs=$IFS
    IFS="$IFS/"
    local neu="/"
    for wort in $Mark ; do
      if [ "$wort" != "$name" ] ; then
        neu="$neu$wort/"
      fi
    done
    IFS="$aifs"
    Mark=$neu
 fi
}
function CW_Eingabe()
{
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 22 0
     read -p "Eingabe:" ein
     case $ein in
      "q" | "Q") ein="q";;
      "w" | "W")
              gvAltZeil=zakt
              gvAltOffs=offset
                 offset=offset+17
                 if [ $offset -gt $anz ] ; then
                    offset=offset-17
```

```
                else
                 zakt=zakt+17
                 if [ $zakt -gt $anz ] ; then
                    zakt=anz
                 fi
              fi
            ;;
"z" | "Z")
              if [ $offset -gt 2 ] ; then
              gvAltZeil=zakt
              gvAltOffs=offset
                 offset=offset-17 ;
                 zakt=zakt-17
              fi
              ;;
"n" | "N") if [ $zakt -lt $anz ] ; then
              gvAltZeil=zakt
              gvAltOffs=offset
                 zakt=zakt+1
                 if [ $zakt -gt $((offset+17)) ] ; then
                    offset=offset+17
                 fi
              fi ;;
"l" | "L") if [ $zakt -gt 2 ] ; then
              gvAltZeil=zakt
              gvAltOffs=offset
                 zakt=zakt-1
                 if [ $zakt -lt $offset ] ; then
                    offset=offset-17
                 fi
              fi ;;
"m" | "M") CW_Mark $tmpfile
           ;;
"a" | "A") CW_Archiv "$Mark"
              ;;
"d" | "D") CW_Delete "$Mark"
              ;;
"")       local line=`head -$zakt $tmpfile|tail -1`
              set -- $line
              local ch=`echo  $1 | cut -c1`
              if [ "$ch" = "d" ] ; then
                 ein="CR"
                 Mark=":"
                 CW_ReadDir $9;
               else
               CW_NormalFile $9
              fi ;;
 *) CW_Print 0 0 ">>$Mark<<"
     ein="" ;;
```

```
    esac
  done
}
function CW_ReadDir ()
{
 offset=2
 zakt=2
 cd "$1" 2>/dev/null && verz="`pwd`"
 CW_Print 0 0
 anz=`ls -Ald .*|tee $tmpfile | wc -l | cut -c-7`
 CW_Box 0 1 40 20 "$verz"
}
function CW_PrintDir()
{
  # Ausgabe der Dateien
  typeset -i xp=$1
  typeset -i hoehe=$2
  ausdatei=$3
  typeset -i i=0
  typeset -i akt=0
  while [ $i -lt $hoehe -a $akt -lt $anz ] ; do
   akt=i+offset
   if [ $gvAltOffs -eq $offset ] ; then
      let "$akt == $gvAltZeil || $akt == $zakt"
      if [ $? -eq 0 ] ; then
         local lvOut="ja"
       else
         local lvOut="nein"
      fi
     else
      local lvOut="ja"
   fi
   if [ "$lvOut" = "ja" ] ; then
      zeile=`head -$akt $ausdatei | tail -1`
      set -- $zeile
      revers=`echo "$Mark" | grep ":$9:" `
      if [ -n "$revers" ] ; then
         CW_SetRev
      fi
      if [ $zakt -eq $akt ] ; then
         CW_SetBold
      fi
      local datei=`echo $9 | cut -c-15`
      local text="`printf "$1|%9i | %-15s" $5 $datei`"
      CW_Print $xp $((i+2)) "$text"
      CW_SetNormal
   fi
   i=i+1
  done
```

```
  while [ $i -lt $hoehe ] ; do
    CW_Print $xp $((i + 2)) "`printf \"%10s|%10s|%16s\"`"
    i=i+1
  done
}
function CW_NormalFile () {
# Schaut nach, welchen Dateityp die Datei hat und reagiert auf Tar / Zip
# Archive
typ=`file $1 | cut -f2 -d":"`
if [ "$typ" = " GNU tar archive" -o "$typ" = " POSIX tar archive" ] ; then
   echo "Dummy" >$tmptarcnt
   tar tvf $1 | awk '{ print $1 " 1 r r " $3 " " $4 " " $5 " " $6 " " $8 }'
>>$tmptarcnt
   boffs=$offset
   offset=2
   banz=$anz
   anz=`cat $tmptarcnt |wc -l`
   CW_PrintDir 41 18 $tmptarcnt $1
   offset=$boffs
   anz=$banz
fi
}

Mark="/"
tput clear
verz=${1:-"`pwd`"}
trap "CW_TrapSIGINT" SIGINT
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp w = PgDn  l = CuUp  n = CuDn a = Archiv   d = Löschen
"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=2
tmpfile="/tmp/cwc$$.tmp"
tmptar="/tmp/backup"
tmptarcnt="/tmp/tar.cnt$$"
cd "$verz"
gvpp="/usr/bin/"
typeset -i anz=`ls -ld .* * | tee $tmpfile | wc -l | cut -c-7`
typeset -i zakt=2               # Zähler akt. Zeile
typeset -i gvAltOffs=-1         # Alter Offset
typeset -i gvAltZeil=-1         # Alte aktuelle Zeile
until [ "$ein" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
rm -f $tmpfile
rm -f $tmptar $tmptarcnt
exit 0
```

9.6 $() anstelle von `


```
`<befehl>`
$(<befehl>)
```

Und wieder einmal muss eine unserer Unterlassungssünden ausgebügelt werden. Immer, wenn die Ausgaben von Befehlen ersetzt werden mussten, hatten wir die Backticks ` eingesetzt. Dies ist zwar kurz, aber auf den ersten Blick sehr leicht mit dem Apostroph ' zu verwechseln.

Wenn Ihnen das ein Dorn im Auge war, so können Sie in der Bash oder der Kornshell auch die Kombination `$(<befehl>)` nutzen. Unterschiede in der Funktion gibt es nicht, allerdings müssen Sie ein Zeichen mehr eintippen als bei den `-Zeichen.

Was Sie in Zukunft einsetzen werden, bleibt Ihnen überlassen, in folgendem Beispiel soll allerdings mit den Klammern gearbeitet werden. Dazu nehmen wir das Skript 19 aus Kapitel 3 und bauen es einfach um:


```
# Skript 19: Unsere erste while-Schleife
#
TMPFILE=/tmp/count
#
# Das cut ist hier nur zur Sicherheit eingefügt
#
anz=$(find $1 -name "$2" -type f -print | tee $TMPFILE | wc -l | cut -c-7)
nr=0
summe=0
while [ $nr -lt $anz ] ; do
  nr=$(expr $nr + 1)
  datei=$(head -$nr $TMPFILE | tail -1)
  echo $datei
  #
  # Dieses cut muss angegeben werden
  #
  erg=$(wc -c $datei | cut -c-7)
  summe=$(expr $summe + $erg)
done
if [ $anz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $anz Dateien belegen $summe Bytes"
fi
rm $TMPFILE
exit 0
```

9.7 Aufgaben

1. Das folgende Skript ist eine Abwandlung von Skript 47. Nur stellt es die Verzeichnisgrößen (sprich die Summe aller Dateigrößen) aller übergebenen Verzeichnisse fest und meckert, wenn die Größe überschritten wird. Der erste Parameter ist Anzahl an Bytes (!) die nicht überschritten werden dürfen, alle restlichen Parameter sind Verzeichnisse. Auf eine Prüfung verzichten wir. Folgende Ausgaben werden erwartet:

   ```
   buch@koala:/home/buch > ./skript48.sh 1000 /home/buch /home/christa
   /home/buch zu groß: 900 Kb
   /home/christa zu groß: 567 Kb
   Insgesamt 2 Verzeichnisse mit insgesamt 1467 Kb
   buch@koala:/home/buch >
   ```

   Tatsächliches Ergebnis ist:

   ```
   buch@koala:/home/buch > ./skript48.sh 1000 /home/buch /home/christa
   Insgesamt 0 Verzeichnisse mit insgesamt 0 Kb
   buch@koala:/home/buch >
   ```

   Wo liegen die Fehler?


```
# Skript 48: Fehlerhaftes Skript
#
pgSize=${1:-1000}
typeset -i gvSumme=0
typeset -i gvAnz=0
# Ersten Parameter ignorieren und dann alle restliche Parameter
shift 1
for gvVerz ; do
   gvZeile=$(du -k $gvVerz 2>/dev/null | tail -1)
   set -- $gvZeile
   let "$1 * 1024 > $pgSize" && ( echo "$gvVerz zu groß: $1 Kb" ; \
                          gvSumme=gvSumme+$1 ; gvAnz=gvAnz+1 )
done
echo "Insgesamt $gvAnz Verzeichnisse mit insgesamt $gvSumme Kb"
exit 0
```

2. Welche Möglichkeit gibt es noch, die folgende Befehlsliste zu kodieren?

   ```
   rm -f $1 2>/dev/null >/dev/null || echo "$1 konnte nicht gelöscht werden!"^
   ```

3. Vermissen Sie eine Lösung zu einer der hier gestellten Aufgaben? Fragen Sie doch die Shell, vielleicht kann sie Ihnen helfen:

```
buch@koala:/home/buch > [ Wo ist die Lösung zu Aufgabe 23 ?
[: missing `]'
buch@koala:/home/buch >
```

Hm, scheinbar ist sie nicht da :)

9.8 Lösungen

1. Änderungen an Variablen in einer Subshell wirken sich nicht in der aufrufenden Shell aus.


```
# Skript 48: Fehlerhaftes Skript in der korrekten Version:
#
pgSize=${1:-1000}
typeset -i gvSumme=0
typeset -i gvAnz=0
# Ersten Parameter ignorieren und dann alle restlichen Parameter
shift 1
for gvVerz ; do
   gvZeile=$(du -k $gvVerz 2>/dev/null | tail -1)
   set -- $gvZeile
   let "$1 * 1024 > $pgSize" &&  echo "$gvVerz zu groß: $1 Kb" ; \
                              gvSumme=gvSumme+$1 ; gvAnz=gvAnz+1
done
echo "Insgesamt $gvAnz Verzeichnisse mit insgesamt $gvSumme Kb"
exit 0
```

2. Die Logik einmal andersrum:

```
! rm -f $1 2>/dev/null >/dev/null && echo "$1 konnte nicht gelöscht werden!"
```

Kapitel 10

sed

»Wir hatten einfach kein Glück –
und dann kam auch noch Pech dazu« –
J. Wegmann (Fußballer)

... Sie allerdings dürfen sich glücklich schätzen: Was die reine Skriptprogrammierung betrifft, haben Sie fast alles gelernt. Einige Reste, die sich in kein anderes Kapitel einpassen ließen, finden Sie im nächsten Kapitel. Stellt sich die Frage, was lernen Sie in diesem Kapitel?

Dieses Kapitel gibt eine kurze Einweisung in den Stream-Editor `sed`. `sed` ist ein Programm, das die Manipulation von großen Textdaten erlaubt. Dabei nimmt `sed` die Daten zeilenweise von der Standardeingabe entgegen, führt die definierten Manipulationen durch und gibt die geänderten Daten auf der Standardausgabe wieder aus. `sed` bearbeitet immer nur ein paar Zeilen auf einmal und nutzt keine temporären Dateien, sodass die einzige Beschränkung bei der Dateigröße darin besteht, dass Quell- und Zieldatei gleichzeitig auf die Festplatte passen müssen.

Wenngleich `sed` auch keine Zeilen automatisch in Puffer sichert, so können Sie mit Befehlen auf einen Puffer, den so genannten *Holdspace*, zugreifen. Da Sie mit Ihrem `sed`-Skript selbst bestimmen, was in den Puffer geschrieben wird, lassen sich durchaus Fälle konstruieren, in denen sehr viel Speicher benötigt wird.

Da `sed` nie mehr als ein paar Zeilen zu einem gegebenen Zeitpunkt bearbeitet, bedeutet das gleichzeitig, dass `sed` bei großen Dateien diese in kleinere »Häppchen« unterteilt und diese dann nacheinander manipuliert werden. Und

noch eine wichtige Anmerkung: sed kann keine relative Adressierung durchführen (d.h. die Fähigkeit, eine Zeilenadresse zu definieren, die abhängig von der aktuellen Zeile ist). Dies liegt daran, dass sed selbst in den »Häppchen« nur jeweils eine Zeile nach der anderen abarbeitet.

Kurz und knapp formuliert agiert sed als Filter. Die Eingabe nimmt sed entweder von der Standardeingabe oder von der angegebenen Datei. Falls mehr als eine Datei für die Eingabe angegeben wird, so werden diese wie eine große Datei behandelt. Die Ausgaben werden auf der Standardausgabe ausgegeben. Aus diesem Grunde wird sed meistens innerhalb von Pipes eingesetzt, obwohl sein Einsatz nicht darauf beschränkt ist.

10.1 sed – Stream-Editor


```
sed [-n] -f <datei>    [> Ausgabe] [<Eingabe]
sed [-n] -e <befehle>  [> Ausgabe] [<Eingabe]
sed [-n] -f <datei>    [> Ausgabe] [<datei> ...]
sed [-n] -e <befehle>  [> Ausgabe] [<datei> ...]
```

Sie können die Befehle für sed auf zwei verschiedenen Wegen definieren. Zum einen können Sie über die Option -e den oder die <Befehle> direkt angeben. Zum anderen ist es vor allem bei längeren sed-Anweisungen einfacher, diese in einer <Datei> abzulegen und diese dann als Parameter zur Option -f anzugeben.

sed gibt nach Abschluss der Manipulationen das Ergebnis auf der Standardausgabe aus. Falls Sie aber selbst bestimmen möchten, wann ein Ergebnis ausgegeben wird, so wird dieses Verhalten mittels Option -n unterdrückt. In diesem Fall müssen Sie mit dem sed-Befehl p selbst dafür sorgen, dass ein Ergebnis ausgegeben wird.

Die Ein- und Ausgabe kann mit einer normalen Umlenkung direkt mit Dateien arbeiten, sollte das gewünscht sein. Falls Sie mindestens <Datei> angeben, so werden die Daten aus dieser Datei gelesen. Bei mehr als einer Datei werden diese nacheinander gelesen und bearbeitet. sed verhält sich aber so, als ob alle Dateien zu einer großen Datei zusammengefasst wurden:

```
buch@koala:/home/buch > sed -n -e '$=' /etc/fstab /etc/profile
262
buch@koala:/home/buch > sed -n -e '$=' /etc/fstab
10
buch@koala:/home/buch > sed -n -e '$=' /etc/profile
252
buch@koala:/home/buch >
```

Der sed-Befehl $= besteht aus der Adresse $, welche die letzte Zeile in der Eingabe darstellt und der Funktion =, welche die aktuelle Zeilennummer ausgibt. Obiges Beispiel demonstriert:

- wie wc -l durch sed ersetzt werden kann.
- wie sed mehrere Dateien als eine große Datei interpretiert.

Schauen wir uns nun genau an, wie sed-Befehle aussehen und welche Funktionen sed versteht.

10.1.1 sed-Befehle

Das allgemeine Format eines von sed anerkannten Befehls sieht so aus:

```
[<adresse1>[,<adresse2>]] <funktion> [<argumente>]
```

Ein sed-Befehl muss also einen Funktionsteil enthalten, während Funktionsargumente und Adreßangaben, auf die der sed-Befehl wirken soll, optional angegeben werden können.

Die Operation, die <funktion> durchführt, bezieht sich auf einen Zeilenbereich. Dieser Bereich kann eine Zeile umfassen, die durch <adresse1> bestimmt wird oder von Zeile <adresse1> bis Zeile <adresse2> reicht. Folgt auf die Adresse ein Ausrufezeichen »!«, so wird <funktion> nur für die Zeilen ausgeführt, die nicht von <adresse1> bzw. <adresse1> bis <adresse2> abgedeckt werden.

Adressen können entweder *Zeilennummern* oder *Muster* sein. Muster bestehen aus *regulären Ausdrücken*, die in »/« eingefasst sind. Reguläre Ausdrücke sind grob ausgedrückt Suchbedingungen. Sind diese Bedingungen wahr, so gilt die <funktion> ab der Zeile, wo die Bedingung erfüllt wurde. Reguläre Ausdrücke sind ein wichtiges Konzept, weshalb wir ihnen in einem eigenen Abschnitt noch unseren Tribut zollen werden. Die spezielle Adresse $ bezieht sich auf die letzte Zeile in der Eingabe.

Leerzeichen und Tabulatoren am Anfang von Befehlszeilen werden ignoriert. Die Anzahl an Leerzeichen, die Adresse und Funktion voneinander trennen, ist beliebig.

Schauen wir uns in Tabelle 10.1 einige gültige Adressen an.

Tabelle 10.1: Adressen bei sed

`4,7`	Alle Zeilen von 4 bis 7 einschließlich: 4,5,6,7.
`1,$`	Alle Zeilen der Datei.
`2,4!`	Alle Zeilen aus den Zeilen 2,3 und 4.
`/Platypus/`	Alle Zeilen, in denen `Platypus` steht.

Damit `sed` überhaupt Zeilen(-bereiche) bearbeiten kann, wird ein Zeilenzähler benötigt. Liest `sed` eine Zeile von der Eingabe, so wird der Zeilenzähler erhöht, und es wird geprüft, welche Befehle diese Zeilennummer in ihrer Adresse definiert haben. Diese Befehle werden dann ausgeführt.

Werfen wir deshalb noch einen Blick darauf, wie `sed` seine Befehle ausführt: `sed` führt die Befehle in der Reihenfolge aus, in der sie angegeben wurden (von oben nach unten), es sei denn, es wurden z.B. Sprungbefehle angegeben. Wird `sed` gestartet, so liest er die erste Zeile von der Eingabe und legt diese in einem internen Puffer ab. Dieser Puffer wird *Musterpuffer* (pattern space) genannt. (Wo habe ich diesen Begriff bloß schon einmal gehört, Scotty?)

Sämtliche Befehle, die `sed` nun ausführt, manipulieren diesen Puffer, wenn die angegebene Adresse mit der Zeile im Musterpuffer übereinstimmt (also die Zeilennummern übereinstimmen oder das Muster in der Zeile gefunden wurde).

Das hört sich kompliziert an, ist es aber nicht, wie Ihnen die folgenden Beispiele zeigen werden. Alle Beispiele nutzen die Datei `downunder.txt`, die folgende Zeilen beinhaltet:

```
buch@koala:/home/buch/skript > cat downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Nehmen wir die `sed`-Funktion `p`. Diese druckt den Musterpuffer auf die Standardausgabe aus. Ohne Angabe einer Adresse druckt `p` alle Zeilen aus:

```
buch@koala:/home/buch/skript > sed -e 'p' downunder.txt
1 Koala
1 Koala
2 Emu
2 Emu
3 Wallaby
3 Wallaby
4 Uluru
4 Uluru
5 Monkey Mia
5 Monkey Mia
6 Leichhardt
6 Leichhardt
7 Sydney
7 Sydney
8 Melbourne
8 Melbourne
9 Canberra
9 Canberra
10 Blue Mountains
10 Blue Mountains
11 Mount Buffalo
11 Mount Buffalo
12 Ovens Highway
12 Ovens Highway
13 Bright
13 Bright
14 Princess Highway
14 Princess Highway
buch@koala:/home/buch/skript >
```

Die Zeilen werden alle doppelt ausgegeben, weil `sed` die Zeilen ohne die Option `-n` selbstständig ausgibt.

```
buch@koala:/home/buch/skript > sed -n -e 'p' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Wie Sie eine bestimmte Zeile aus der Datei/Eingabe ausschneiden können, haben wir bereits in einem Vorgriff auf dieses Kapitel erwähnt:

```
buch@koala:/home/buch/skript > sed -n -e '8 p' downunder.txt
8 Melbourne
buch@koala:/home/buch/skript >
```

Jetzt sollte auch klar sein, was dieses sed-Miniskript bewirkt. Es druckt alle Zeilen aus (p), deren Adresse gleich Zeilennummer 8 ist. Eine äußerst komplizierte Formulierung für: Dieses Skript druckt Zeile 8 der Eingabe(-datei) aus.

Die Funktion n druckt den Inhalt des Musterpuffers auf die Standardausgabe aus und lädt die nächste Zeile in den Musterpuffer ein. Wurde die Option -n angegeben, so lädt n nur die nächste Zeile ein.

Damit können wir jetzt jede zweite Zeile ausgeben:

```
buch@koala:/home/buch/skript > sed -n -e 'p;n' downunder.txt
1 Koala
3 Wallaby
5 Monkey Mia
7 Sydney
9 Canberra
11 Mount Buffalo
13 Bright
buch@koala:/home/buch/skript >
```

Die erste Zeile wird eingeladen und mit p ausgedruckt. Danach wird die zweite Zeile in den Musterpuffer eingeladen, aber nicht ausgegeben (n druckt nichts aus, weil die Option -n aktiv ist). Dann geht es wieder von vorne los: Jetzt wird die dritte Zeile geladen und mit p gedruckt usw. bis zum Ende der Eingabe.

Schauen wir uns noch eine Adresse mit Muster an. Wir wollen nur Zeilen ausdrucken, in denen ein M enthalten ist:

```
buch@koala:/home/buch/skript > sed -n -e '/M/p' downunder.txt
5 Monkey Mia
8 Melbourne
10 Blue Mountains
11 Mount Buffalo
buch@koala:/home/buch/skript >
```

Dieses waren einige einfache Beispiele, die uns gezeigt haben:

- wie sed-Befehle aufgebaut sind,
- wie sich Adressen auswirken,
- wie sich -n auswirkt.

Wichtig ist noch folgende Feststellung: `sed` manipuliert immer den aktuellen Inhalt des Musterpuffers. Wird also eine Zeile in den Puffer geladen und manipuliert, so führt die zweite Manipulation dazu, dass das Ergebnis der ersten Änderung bearbeitet wird und *nicht* der ursprüngliche Inhalt des Musterpuffers.

Auch ist es möglich, mehr als eine Zeile im Muster- oder Sicherungspuffer zu halten. Wie das genau geht, werden wir etwas später lernen.

10.1.2 Reguläre Ausdrücke

Kommen wir nun zu den Adressen, die keine Zeilennummer, sondern ein Muster angeben. Wie weiter oben schon erwähnt, bestehen diese aus regulären Ausdrücken, die in »/« eingefasst sind. Ein regulärer Ausdruck ist dabei nichts anderes als ein Suchbegriff, der sich auf den Musterpuffer von `sed` bezieht.

Was ist nun ein *regulärer Ausdruck* (RA)?

- Ein normales Zeichen ist ein RA, das auf sich selbst passt. So passt der RA `/C/` auf jede Zeile der Eingabe, die das Zeichen `C` enthält.
- Ein Caret ^ am Anfang eines RA bedeutet, dass der RA am Anfang einer Zeile stehen muss. So findet der RA `/^Christa/` nur Zeilen, die mit `Christa` anfangen.
- Ein Caret ^ am Anfang eines RA findet den Nullstring am Anfang einer Zeile im Musterpuffer. Einfacher formuliert: Der reguläre Ausdruck wird nur gefunden, wenn er am Zeilenanfang steht. `/^S/` Zeile muss mit `S` anfangen.
- Ein `\<` am Anfang eines RA findet nur Zeilen, in denen die folgende Zeichenkette am Anfang eines Wortes steht. So findet `/\<euro/` die Zeilen `europa hat eine eigene Währung` und `Tasmanien ist der australische Staat, der am europäischten wirkt`, aber nicht `Neurologen sind Ärzte`.
- `\>` am Ende eines RA findet alle Zeilen, in denen der RA am Ende eines Wortes steht. `/inter\>/` findet z.B. `Winter ist in Australien von Mai bis August`. Aber nicht `Australien ist international geprägt`.
- Ein Dollarzeichen $ am Ende eines RA findet alle Zeilen, in denen die Zeichenkette vor dem $ am Ende der Zeile steht. `/way$/` findet alle Zeilen, die mit `way` enden.
- Ein `\n` findet einen Zeilenumbruch innerhalb des Musterpuffers. Ein Zeilenumbruch am Ende des Musterpuffers wird aber nicht gefunden. Dies ist nützlich, wenn Ihr `sed`-Skript mehr als eine Zeile in den Musterpuffer eingelesen hat und Sie nun die Zeilenumbrüche suchen.

- Ein Punkt ».« steht für genau ein Zeichen innerhalb des Musterpuffers. Dabei steht der Punkt für alle Zeichen außer dem abschließenden Zeilenumbruch.
- /Litchfi.ld/ findet Zeilen, in denen z.B. Litchfield, LitchfiEld oder auch Litchfi-ld steht.
- Ein RA, auf den ein Stern * folgt, findet alle Zeilen, in denen der RA, der auf den Stern folgt, keinmal oder mehrfach auftritt. Hier ein komplettes Minibeispiel. Es druckt alle Zeilen aus, in denen wenigstens ein al gefolgt von 0 oder mehr l steht (al, all, alll, ...).

```
buch@koala:/home/buch/skript > sed -n -e '/all*/p' downunder.txt
1 Koala
3 Wallaby
11 Mount Buffalo
buch@koala:/home/buch/skript >
```

- Ein RA kann außerdem Zeichenbereiche (ranges) enthalten, die durch [] eingefasst werden. Ein Zeichenbereich deckt genau ein Zeichen ab, welches in den Klammern angegeben ist. Ist das erste Zeichen ein Caret ^, so deckt der Zeichenbereich genau ein Zeichen ab, welches nicht in den Klammern angegeben ist und nicht der abschließende Zeilenumbruch im Musterpuffer ist.

  Möchten Sie z.B. alle Zeichen von a bis f im Zeichenbereich angeben, so können Sie alle sechs Buchstaben angeben (abcdef) oder auch einen Bereich definieren: [a-f].

  /[abc]/ oder /[a-c]/ findet alle Zeilen, in denen ein a, b oder c vorkommt.
- RA können miteinander verknüpft werden und decken Zeichenketten ab, die allen RA in der angegebenen Reihenfolge (von links nach rechts) genügen.
- RA können mittels \(und \) zu einer Gruppe zusammengefasst werden. Diese Gruppen können dann mittels \x referenziert werden. Dabei ist x eine Ziffer von 1 bis 9. Wird eine Gruppe mittels \x referenziert, so zählt sed die Anzahl der \(im aktuellen RA und ersetzt \x durch die x-te Gruppe.

  Dabei wird allerdings nicht das Muster der Gruppe, sondern der gemachte Text für \x ersetzt. Ein kurzes Beispiel:

```
buch@koala:/home/buch > cat tst.txt
aa
ab
ba
bb
ca
```

```
buch@koala:/home/buch > sed -n '/\([ab]\)\([ab]\)/p' tst
aa
ab
ba
bb
buch@koala:/home/buch > sed -n '/\([ab]\)\1/p' tst
aa
bb
buch@koala:/home/buch >
```

Die einzelnen Elemente einer Gruppe werden dabei durch \| getrennt. Wenn uns alle Zeilen aus `downunder.txt` interessieren, die Berge (»Mount«) oder Highways angeben und nur einen Einzeiler schreiben wollen, dann hilft folgende Zeile:

```
buch@koala:/home/buch/skript > sed -n -e '/\(Mount\|Highway\)/p' down-
under.txt
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
14 Princess Highway
buch@koala:/home/buch/skript >
```

- Ein leerer RA ist identisch mit dem letzten von `sed` ausgewerteten RA.

Natürlich gibt es unter Unix und der Shell mehr als eine Möglichkeit, ein Problem zu lösen. Wir hatten ja bereits in Kapitel 5 und 6 auf den Befehl `grep` hingewiesen. Bei vielen grep-Versionen existiert eine Option `-E`, welche die Angabe von RA erlaubt. Falls Ihre Version diese Option nicht kennt, dann versuchen Sie es doch mal mit `egrep`:

```
buch@koala:/home/buch/skript > egrep '(Mount|Highway)' test.txt
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
14 Princess Highway
buch@koala:/home/buch/skript >
```

Nutzen wir die RA nun für einige kleine Skripten, die den Inhalt von `downunder.txt` manipulieren. Geben wir zunächst die Zeilennummern aller Zeilen aus, die durch den RA `/all*/` abgedeckt werden:

```
buch@koala:/home/buch/skript > sed -n -e '/all*/=' downunder.txt
1
3
11
buch@koala:/home/buch/skript >
```

Geben wir jetzt alle Zeilen aus, in denen kein Wort mit Mo anfängt. Hier die erste Möglichkeit:

```
buch@koala:/home/buch/skript > sed -n -e '/\<Mo/!p' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Das »!« führt dazu, dass alle Zeilen berücksichtigt werden, auf die das Muster der Adresse nicht zutrifft. Es gibt aber auch noch einen Befehl d, der den Inhalt des Musterpuffers löscht und den Bearbeitungsdurchlauf von sed startet (also nächste Zeile laden und sed-Skript von vorne beginnen).

```
buch@koala:/home/buch/skript > sed -e '/\<Mo/d' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Damit möchten wir das Thema reguläre Ausdrücke abschließen. Es wird Zeit, sich mit einigen Funktionen von sed auseinander zu setzen.

10.1.3 Funktionen

Jetzt wissen wir zwar, wie wir bestimmte Zeilen zur Bearbeitung auswählen, aber wie wir eine Zeile bearbeiten können, wurde nur in einigen Beispielen erwähnt. Das soll sich nun ändern. Zunächst werden wir uns noch einige weitere Funktionen anschauen und diese mit einigen kurzen Bespielen erläutern. In Abschnitt 10.2 werden wir dann diese Befehle nutzen, um komplexere sed-Skripten zu schreiben. Wenn im Folgenden *adressierte Zeilen* erwähnt werden, so sind die Zeilen gemeint, die durch die Angabe von <adresse1> und evtl. <adresse2> zur Bearbeitung ausgewählt wurden (siehe auch 10.1.1).

Die Liste der Funktionen in diesem Kapitel ist keineswegs komplett. Sie soll Ihnen nur ein Gefühl für die Funktionsweise von sed vermitteln und aufzeigen, welche Probleme Sie mit diesem Programm angehen können. Für weitere Informationen zu sed möchten wir auf den Anhang D *Ressourcen im Netz* verweisen. Die Funktionen, die bereits weiter oben erläutert wurden, haben wir hier nicht nochmals aufgeführt.

Tabelle 10.2: Weitere Funktionen von sed

`: <Marke>`	Label	keine Adresse	Definiert eine Sprungmarke, zu der durch `b` oder `t` verzweigt werden kann.
`a\ <Text>`	Append	max. 1 Adresse	Gibt den `<Text>` am Ende jeder Zeile aus, wenn diese auf die Standardausgabe gedruckt wird. Der Zeilenumbruch hinter dem \ ist wichtig und darf nicht vergessen werden!
`b <Marke>`	Branch	max. 2 Adressen	Verzweigt zur angegebenen `<Marke>`. Ist diese nicht definiert worden (`: <Marke>`), so springt `b` an das Ende des `sed`-Skripts.
`c\ <Text>`	Change	max. 2 Adressen	Löscht die durch die Adresse(n) ausgewählten Zeilen und ersetzt diese durch den `<Text>`. Wird mehr als eine Zeile gelöscht (durch die Angabe von zwei Adressen), so wird `<Text>` dennoch nur einmal ausgegeben und nicht für jede gelöschte Zeile einmal. Der Zeilenumbruch hinter dem \ ist wichtig!
`d`	Delete	max. 2 Adressen	Löscht die angegebene(n) Zeilen(n) und lädt den Puffer mit einer neuen Zeile.
`h`	Hold	max. 2 Adressen	Kopiert den Musterpuffer in den Sicherungspuffer. Der vorherige Inhalt des Sicherungspuffers geht verloren.
`n`	Next	max. 2 Adressen	Gibt den aktuellen Inhalt des Musterpuffers auf die Standardausgabe aus (es sei denn, die Option `-n` wurde angegeben) und liest die nächste Zeile von der Eingabe in den Musterpuffer ein.
`N`	Next	max. 2 Adressen	Liest eine neue Zeile von der Eingabe ein und hängt diese, durch einen Zeilenumbruch getrennt, an den aktuellen Inhalt des Musterpuffers an.
`q`	Quit	max. 1 Adresse	Falls nicht durch die Option `-n` unterdrückt, wird der aktuelle Inhalt des Musterpuffers ausgegeben, falls nötig noch Text angehängt (durch Funktion `a\`). Beendet das aktuelle `sed`-Skript endgültig.
`r <Datei>`	Read File	max. 1 Adresse	Nachdem die angegebene Adresse erreicht wurde, wird `<Datei>` eingelesen und ausgegeben.

t <Marke>	Test	max. 2 Adressen	Wurde seit dem Aufruf von t oder seit dem letzten Einlesen einer Zeile von der Eingabe eine Ersetzung vorgenommen, so springt t zur angegebenen Marke. Ansonsten wird die nächste Funktion nach t ausgeführt.
w <datei>	Write	max. 2 Adressen	Schreibt die adressierten Zeilen in die <datei>. Es *muss* genau ein Leerzeichen zwischen w und <datei> stehen. Existiert <datei> nicht, wenn sed aufgerufen wird, so wird diese <datei> angelegt, ansonsten wird sie überschrieben. Wiederholte Aufrufe von Write innerhalb des sed-Skripts hängen die auszugebenden Zeilen an das Ende der Datei an. Innerhalb eines sed-Skripts dürfen nicht mehr als zehn verschiedene w-Aufrufe stehen. Dabei zählen w-Flags für die Ersetzungsfunktion s mit!
y/<alt>/<neu>/	Transform	max. 2 Adressen	Tauscht im Musterpuffer das erste in <alt> angegebene Zeichen durch das erste in <neu> angegebene Zeichen aus. Das Gleiche wird für alle Zeichen in <alt> durchgeführt. y/Ab/aB/ wandelt alle großen A in kleine a um und kleine b in große B.
x	eXchange	max. 2 Adressen	Tauscht den Inhalt von Musterpuffer und Sicherungspuffer miteinander aus.

Ein Beispiel zu c:

```
buch@koala:/home/buch/skript > sed -e '/Mount/c\
> Hier stand ein Berg
> ' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
Hier stand ein Berg
Hier stand ein Berg
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Das folgende Beispiel zu q gibt alle Zeilen aus, bis eine Zeile mit 9 beginnt:

```
buch@koala:/home/buch/skript > sed -e '/^9/q' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
buch@koala:/home/buch/skript >
```

Zum krönenden Abschluss ein Beispiel zu r:

Angenommen wir haben die Datei hinweis.txt mit dem Inhalt:

```
Hinweis:
Häufig wird Sydney oder Melbourne als Hauptstadt Australiens angenommen.
Richtig jedoch ist Canberra, was laut den Gründern der Stadt der Aborigine-
Begriff für
"großer Versammlungsplatz" ist. Dass er jedoch auch "Busen einer Frau"
bedeutet, wird
häufig verschwiegen.
```

und wollen diesen Text ausgeben, wenn wir "Canberra" in downunder.txt finden.

```
buch@koala:/home/buch/skript > sed -e '/Canberra/r hinweis.txt' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
Hinweis:
Häufig wird Sydney oder Melbourne als Hauptstadt Australiens angenommen.
Richtig jedoch ist Canberra, was laut den Gründern der Stadt der Aborigine-
Begriff für
"großer Versammlungsplatz" ist. Dass er jedoch auch "Busen einer Frau"
bedeutet, wird
häufig verschwiegen.
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

10.1.4 Die Substitute-Funktion


```
s/<muster>/<ersatz>/<flags>
```

Die Angabe eines Flags ist optional.

Ein großer Teil der Arbeit wird von sed-Skripten sicherlich mithilfe der Substitute-Funktion erledigt. Aus diesem Grunde wollen wir diese Funktion einmal genauer unter die Lupe nehmen.

Grundsätzlich tauscht Substitute das <muster> durch ein <ersatz>-Muster aus. Werden keine Adressen angegeben, so gilt das für alle Zeilen, die das zu ersetzende <muster> enthalten. Dabei können Sie auf <muster> alle Regeln für RA anwenden, die weiter oben für Kontextadressen aufgeführt wurden. Während Kontextadressen aber nur von / eingefasst werden dürfen, ist diese Beschränkung bei s nicht gegeben. Das heißt, ein s/a/A/ funktioniert genauso wie ein s!a!A!.

Normalerweise ersetzt s nur das *erste* Vorkommen von <muster> in einer Zeile durch das <ersatz>-Muster. Dies lässt sich durch die Angabe von g auf alle Vorkommen innerhalb einer Zeile ausdehnen.

Durch Flags (Tabelle 10.3) lässt sich die Funktion von Substitute manipulieren.

Tabelle 10.3: Flags für den Substitute-Befehl

Flag	Bedeutung
g	Global. Ersetzt alle Vorkommen von <muster> in einer Zeile, nicht nur das erste.
<x>	Ersetzt nur das x-te Vorkommen von <muster> in der Zeile durch <ersatz>-Muster. Dabei ist <x> eine Zahl. So ersetzt s/z/Z/2 das zweite kleine z durch ein großes Z.
p	Print. Wurde eine Ersetzung durchgeführt, so wird das Ergebnis auf die Standardausgabe ausgedruckt.
w <datei>	Write <Datei>. Gibt die Zeile in die Datei aus (siehe auch w Write), wenn eine Ersetzung vorgenommen wurde. Es muss *genau ein* Leerzeichen zwischen w und dem Dateinamen stehen.

Und noch ein letzter Tipp für s. Wenn Sie im <ersatz>-Muster ein kaufmännisches Und & (Ampersand) angeben, dann wird das & durch den Text ersetzt, auf den <muster> in der Zeile passte. Wollen wir z.B. alle Zeilen der Datei downunder.txt ausgeben, in der mindestens ein y steht und diese Stellen zur besseren Übersicht in () setzen, so hilft folgendes Miniskript:

```
buch@koala:/home/buch/skript > sed -n -e "s/y/(&)/gp" downunder.txt
3 Wallab(y)
5 Monke(y) Mia
7 S(y)dne(y)
12 Ovens Highwa(y)
14 Princess Highwa(y)
buch@koala:/home/buch/skript >
```

10.2 Einige Beispiele

So viele Seiten ohne ein längeres Beispiel. Höchste Zeit, dies zu ändern. Die folgenden Beispiele sollen Ihnen zeigen, welche Möglichkeiten in `sed` stecken und was Sie mit `sed` erreichen können. Wir hoffen, dass sich die Beispiele auch für Ihre Aufgaben als nützlich erweisen. Falls Sie die Skripten als Basis für eigene Versuche nutzen wollen: Nur zu, denn Übung macht den Meister!

10.2.1 Text mit Rand versehen

Irgendwann kommt einmal die Zeit, wo Sie Ihre Texte an den Drucker schicken wollen. Unglücklicherweise ist der Text so weit am Papierrand gedruckt, dass ein Lochen nicht möglich ist, ohne den ausgedruckten Text mit Löchern zu versehen. Was tun?

Die Antwort wird Sie wohl kaum überraschen: `sed` nutzen!

Wir möchten hier nicht darauf eingehen, wie Sie Daten auf den Drucker schicken, auf den meisten Systemen sollte ein Pipe nach `lp` oder `lpr` die Daten an den Druckerspooler übergeben. Da wir hier aber sehen wollen, wie die Ausgabe aussieht, lassen wir das Spoolen (Übergabe der Druckdaten an die Druckerverwaltung) hier weg.

```
buch@koala:/home/buch/skript > sed -e 's/^/...../' downunder.txt
.....1 Koala
.....2 Emu
.....3 Wallaby
.....4 Uluru
.....5 Monkey Mia
.....6 Leichhardt
.....7 Sydney
.....8 Melbourne
.....9 Canberra
.....10 Blue Mountains
.....11 Mount Buffalo
.....12 Ovens Highway
.....13 Bright
.....14 Princess Highway
buch@koala:/home/buch/skript >
```

Die Funktion s hat keine Adresse angegeben und bezieht sich daher auf alle Zeilen in der Datei `downunder.txt`. Sie ersetzt den Zeilenanfang, dargestellt durch den RA ^, durch Punkte (oder beispielsweise auch Leerzeichen) »`.....`«. Da sich der Zeilenanfang aber nicht ersetzen lässt, werden die Punkte am Anfang der Zeile eingefügt. Die Punkte haben wir hier übrigens nur verwendet, damit Sie sehen können, wie viele Zeichen wo eingefügt werden. Gerade in einem Buch lassen sich Leerzeichen nur sehr schwer darstellen.

10.2.2 Textbereich aus Datei ausgeben

Was aber tun, wenn Sie nicht die ganze Datei ausgedruckt haben wollen, sondern nur die Zeilen 3 bis 5? Auch hier bietet `sed` eine Lösung an:

```
buch@koala:/home/buch/skript > sed -n -e '3,5p' downunder.txt
3 Wallaby
4 Uluru
5 Monkey Mia
buch@koala:/home/buch/skript >
```

Wenn Sie ab Zeile 3 alle Zeilen bis zum Ende ausgeben wollen, so könnten Sie `3,5p` durch `3,14p` ersetzen. Sinnvoll ist das aber nicht, weil Sie dann davon ausgehen, dass die Datei immer maximal 14 Zeilen haben wird. Es ist kein Problem, eine Zeilenadresse anzugeben, die größer ist als die Zeilenanzahl in der Datei. `sed` hört mit der letzten Zeile auf, ohne Fehler auszugeben. Wenn die Datei aber mehr als 14 Zeilen enthält, könnten Sie zum einen anfangen zu zählen, oder Sie nutzen die Adresse $, die für die letzte Zeile der Datei steht.

```
buch@koala:/home/buch/skript > sed -n -e '3,$p' downunder.txt
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

10.2.3 Suchen ohne Beachtung der Groß-/Kleinschreibung

Wenn Sie in Kontextadressen Texte angeben, werden diese nur dann gefunden, wenn die Schreibweise 1:1 übereinstimmt. Manchmal ist es aber sinnvoller, ohne Beachtung der Groß-/Kleinschreibung zu suchen. `sed` bietet keine Option an, die dies direkt ermöglicht. Daher hier ein Skript, das eine solche Suche über Sicherungspuffer und Transformfunktion ermöglicht:

```
buch@koala:/home/buch > cat sed.sed
h;                  # Kopiere akt. Zeile in Sicherungspuffer
y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
                    # Im Musterpuffer Großbuchstaben in Kleinbuchstaben
                    # wandeln
/kapitel/!b ende ;  # Wenn Kapitel nicht gefunden, dann ans Skriptende
x       ;           # Sicherung zurück in den Hauptpuffer
p ;                 # und ausdrucken.
: ende              # Fertig mit akt. Zeile
```

```
buch@koala:/home/buch > sed -n -f sed.sed kapitel1.txt
dieses Kapitel die wichtigsten Grundlagen der Shellprogrammierung erklären
Ursache des Fehlers wird uns in Kapitel 6 klar werden. Vorerst fügen Sie
Ausführliche Informationen über Variablen verbreitet das Kapitel 6. Dort
uns Kapitel 4 (For-Schleife) und in Kapitel 5 (Parameter) noch mal genauer
So das soll es für dieses Kapitel gewesen sein. War doch gar nicht sooo
bringen, aber leider fällt dieses Kapitel aus diesem Rahmen. Die in diesem
Kapitel vermittelten Grundlagen müssen Sie unbedingt verstehen, sonst sind
Sie in den folgenden Kapiteln verloren. Also Zähne zusammenbeißen:
buch@koala:/home/buch/skript >
```

Zuerst kopieren wir die aktuelle Zeile im Musterpuffer in den Sicherungspuffer. Im Musterpuffer tauschen wir mittels Transformfunktion alle Großbuchstaben gegen Kleinbuchstaben aus. In Zeilen, in denen kein `kapitel` steht (Ausrufezeichen beachten!), springt `b` zum Label `ende` am Ende des Skripts.

Dann wird der Inhalt von Sicherungspuffer und Musterpuffer ausgetauscht. Im Musterpuffer steht dadurch wieder die Originalzeile, die dann mit `p` ausgedruckt wird.

Die Semikola sind notwendig, damit die Kommentare nicht als Argument für die Funktionen interpretiert werden. Die `L` sind Kommentarzeichen, genau wie in der Shell. Alle Zeichen danach werden von `sed` bis zum Zeilenende ignoriert. Steht ein Kommentar hinter einem `sed`-Befehl, so muss dieser mit einem »`;`« abgeschlossen werden. Da `sed` Leerzeichen zwischen Befehlsende und »`;`« dem Befehl zuschlägt, kann es sein, dass diese nicht damit klar kommt. Deshalb ist es am besten, direkt hinter dem Befehl das Semikolon zu setzen.

Genau genommen ist die Angabe des Labels `ende` unnötig. Wird für »b« kein Label angegeben, so springt `b` auch ans Ende des Skripts.

Was aber machen wir nun, wenn Sie nicht einen festen Begriff suchen, sondern nach einem Begriff, der Ihrem Shellskript übergeben wurde? Ihr Shellskript benötigt an einer Stelle `sed`, aber eben nicht mit festem Suchbegriff. Eine Lösung wäre zum Beispiel, dass Sie das `sed`-Skript von Ihrem Skript in eine Datei schreiben. Dazu können Sie die Umlenkung nutzen und Parameter eintragen:

```
...
echo "y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/" >$tmpsed
echo "/$1/s/\</./" >>$tmpsed
...
```

Zwar können Sie Funktionen auch mit »`;`« trennen und dann alles in ein `-e` unter Verwendung von Gänsefüßchen `""` packen, allerdings finden wir die gezeigte Methode bei längeren `sed`-Skripten wesentlich übersichtlicher.

Die zweite Möglichkeit wäre, den Befehl in der Shell mit `""` zu quoten:

```
sed -n -e datei "/${1}/s/\</./
```

10.2.4 Wörter in Anführungszeichen setzen

Angenommen, Sie wollen in allen Zeilen das dritte Wort in Anführungszeichen setzen und alle anderen Zeilen unverändert ausgeben. In diesem Fall würde Ihnen dieses Skript weiterhelfen:

```
buch@koala:/home/buch/skript > sed -e 's/\</"/3; s/\>/"/3' downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey "Mia"
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
10 Blue "Mountains"
11 Mount "Buffalo"
12 Ovens "Highway"
13 Bright
14 Princess "Highway"
buch@koala:/home/buch/skript >
```

Zunächst wird in jeder Zeile der dritte Wortanfang gesucht und durch ein Anführungszeichen ersetzt (`s/\</"/3`)

Wortanfang und Wortende sind Zeichenketten der Länge 0 (also ""), die vor bzw. hinter jedem Wort folgen. Werden diese Zeichenketten ersetzt, so entsteht ein neues Wort. Und laut Definition stehen vor und nach jedem Wort diese Zeichenketten.

```
10 Blue Mountains
```

hat somit für sed folgende Wortanfänge und -enden:

```
\<10\> \<Blue\> \<Mountains\>
```

Fügen wir nun beim Wort Mountains für das \< ein Anführungszeichen ein, so ergibt sich "Mountains\>. Da dies aber wieder ein Wort ist, ergibt sich \<"Mountains\>. Wortanfänge und -enden verschwinden nur, wenn das Wort selbst verschwindet :)

10.2.5 Funktionen zusammenfassen

Wir wollen ein sed-Skript schreiben, welches alle Zeilen aus downunder.txt ausgibt, in denen kein Mount vorkommt. Alle Zeilen, die Mount enthalten, sollen am Ende ausgegeben werden.

Folgendes wäre eine mögliche Lösung:

```
/Mount/H ;      # Alle Zeilen, die ein Mount enthalten, bearbeiten
                # und am Ende des Sicherungspuffers hinzufügen
${      ;       # Am Ende der Eingabe (letzte Zeile)
g       ;       # Sicherungspuffer in Musterpuffer kopieren
p       ;       # und Musterpuffer ausgeben.
}
/Mount/!p;      # Alle Zeilen, die kein Mount enthalten, ausgeben
buch@koala:/home/buch/skript > sed -n -f s1.sed downunder.txt
1 Koala
2 Emu
3 Wallaby
4 Uluru
5 Monkey Mia
6 Leichhardt
7 Sydney
8 Melbourne
9 Canberra
12 Ovens Highway
13 Bright
14 Princess Highway
10 Blue Mountains
11 Mount Buffalo
buch@koala:/home/buch/skript >
```

Dieses Beispiel zeigt Ihnen, wie Sie mehr als einen Befehl für eine Adresse ausführen können. Wenn Sie Befehle in geschweifte Klammern fassen, so werden alle geklammerten Befehle für die entsprechende Adresse ausgeführt.

In diesem Beispiel wird am Ende der Datei zunächst der Inhalt des Sicherungspuffers in den Musterpuffer kopiert und danach der Musterpuffer ausgedruckt.

10.2.6 Ersetzungen

Wir wollen in diesem Beispiel alle Zeilen durch < am Zeilenanfang und > am Zeilenende markieren, die mit einer einstelligen Zahl beginnen.

```
buch@koala:/home/buch/skript > sed -e '/^[0-9] /s/^.*/<&>/' downunder.txt
<1 Koala>
<2 Emu>
<3 Wallaby>
<4 Uluru>
<5 Monkey Mia>
<6 Leichhardt>
<7 Sydney>
<8 Melbourne>
<9 Canberra>
10 Blue Mountains
11 Mount Buffalo
12 Ovens Highway
13 Bright
14 Princess Highway
buch@koala:/home/buch/skript >
```

Der erste Teil des Skripts `/^[0-9] /` trifft auf alle Zeilen zu, die mit den Zahlen 1 bis 9 und einem Leerzeichen beginnen. Für diese Zeilen werden alle Zeichen der Zeile vom Zeilenanfang (^ -> Zeilenanfang, `.*` -> ein Zeichen 0 bis beliebig oft) durch `<&>` ersetzt. Dabei steht `&` für den Text, auf den das Muster von s passte. Und da `^.*` für die ganze Zeile steht, wird diese für `&` eingesetzt.

Angenommen, wir wollen nur die Wörter der Zeile ausgeben, aber nicht die Zeilennummer. In diesem Fall könnte eine Lösung so aussehen:

```
buch@koala:/home/buch/skript > sed -e 's/\([0-9]*\) \(.*\)/\2/' downunder.txt
Koala
Emu
Wallaby
Uluru
Monkey Mia
Leichhardt
Sydney
Melbourne
Canberra
Blue Mountains
Mount Buffalo
Ovens Highway
Bright
Princess Highway
buch@koala:/home/buch/skript >
```

Der Schlüssel zur Lösung ist an dieser Stelle die Aufteilung der Zeile in zwei Gruppen. Jede Gruppe wird durch ein \(eingeleitet und durch \) abgeschlossen. Die erste Gruppe enthält alle Ziffern in beliebiger Anzahl hintereinander, also Zahlen. Die zweite Gruppe folgt nach einem Leerzeichen und enthält den Rest der Zeile (alle Zeichen in beliebiger Anzahl).

Ersetzt werden soll dieser RA durch den Text, auf den das Muster der zweiten Gruppe passte (also den Rest der Zeile nach Zahl und Leerzeichen).

10.2.7 Daten in eine Datei schreiben

Und wieder einmal malträtieren wir `sed` mit unserer Datei `downunder.txt`. Diesmal soll er alle Zeilen in die Datei `erg.txt` schreiben, die ein `a` oder `h` beinhalten. Diese Zeichen sollen dabei groß geschrieben werden. `sed` soll jede in `erg.txt` geschriebene Zeile in der Originalversion mit dem Kommentar »`Schreibe Zeile :...`« ausgeben.

Bitte beachten Sie, dass `sed` mit `w` beschriebene Dateien nicht beim Start löscht oder deren Inhalt löscht, sondern immer ans Dateiende anhängt. Daher müssen Sie solche Dateien vorher löschen!

```
buch@koala:/home/buch/skript > cat s2.sed
/[ah]/{         ;             # Alle Zeilen, in denen a oder h vorkommt,
h               ;             # in den Sicherungspuffer
s/^.*/Schreibe :<&>/;         # Musterpuffer für Ausgabe anpassen
p               ;             # und ausdrucken: Schreibe...
g               ;             # Sicherung in den Musterpuffer kopieren
y/ah/AH/;                     # a und h gegen A / H austauschen
w erg.txt
}
buch@koala:/home/buch/skript > sed -n -f s2.sed downunder.txt
Schreibe :<1 Koala>
Schreibe :<3 Wallaby>
Schreibe :<5 Monkey Mia>
Schreibe :<6 Leichhardt>
Schreibe :<9 Canberra>
Schreibe :<10 Blue Mountains>
Schreibe :<11 Mount Buffalo>
Schreibe :<12 Ovens Highway>
Schreibe :<13 Bright>
Schreibe :<14 Princess Highway>
buch@koala:/home/buch/skript > cat erg.txt
1 KoAlA
3 WAllAby
5 Monkey MiA
6 LeicHHArdt
9 CAnberrA
10 Blue MountAins
11 Mount BuffAlo
```

```
12 Ovens HigHwAy
13 BrigHt
14 Princess HigHwAy
buch@koala:/home/buch/skript >
```

Und als letztes Beispiel ein Miniskript, welches Dateien nur dann in `erg.txt` schreibt, wenn `Mount` durch `Berg` ersetzt werden konnte.

```
buch@koala:/home/buch/skript > sed -n -e 's/Mount/Berg/w erg.txt'
downunder.txt
buch@koala:/home/buch/skript > cat erg.txt
10 Blue Bergains
11 Berg Buffalo
buch@koala:/home/buch/skript >
```

10.3 Aufgaben

1. Schreiben Sie ein `sed`-Skript, das jede gerade Zeile der Datei ausgibt.

   Tipp: Es könnte ähnlich wie das Skript funktionieren, welches die ungeraden Zeilen ausgibt.

2. Fügen wir noch eine Zeile an das Ende von `downunder.txt` an:

   ```
   15   Stuart   Highway
   ```

   Wenn Sie das zweite Skript aus 10.2.6 laufen lassen, ist die Ausgabe nicht sehr schön:

   ```
   ...
   Ovens Highway
   Bright
   Princess Highway
    Stuart    Highway
   buch@koala:/home/buch/skript >
   ```

   Wie können Sie den RA so abändern, dass alle drei (!) Wörter mit einer beliebigen Anzahl von Leerzeichen als Trenner erkannt werden? Kleiner Tipp: Die Wörter zwei und drei bestehen nur aus Buchstaben!

3. Nehmen Sie die letzte Lösung als Basis, und drehen Sie die Wörterreihenfolge um. Hier einige Beispielzeilen:

   ```
   Koala 1
   Emu 2
   Higway Stuart 15
   ```

   Es sollen keine Leerzeichen am Zeilenanfang stehen!

4. Erinnern Sie sich noch an unseren Sprücheklopfer aus Kapitel 6? Dort haben wir noch mit `head` und `tail` gearbeitet. Erstellen Sie eine Version, die `sed` nutzt und auf `expr` verzichtet.

5. Wie müsste ein einzeiliges Skript aussehen, das ab der Zeile `2 Emu` bis zur letzten Zeile alles ausgibt?

10.4 Lösungen

1. Lösung zur ersten Aufgabe:

```
buch@koala:/home/buch/skript > sed -n -e 'n;p' downunder.txt
2 Emu
4 Uluru
6 Leichhardt
8 Melbourne
10 Blue Mountains
12 Ovens Highway
14 Princess Highway
buch@koala:/home/buch/skript >
```

2. Der RA könnte folgendermaßen aussehen:

```
s/\([0-9]*\) *\([A-Za-z]*\) *\([A-Za-z]*\)/\1-\2-\3/
```

   Die Eingabezeile wird in dieser Lösung in drei Gruppen unterteilt. Die erste Gruppe enthält die Ziffern von `0-9`. Die zweite und dritte Gruppe enthält alle Groß- und Kleinbuchstaben in beliebiger Anzahl. Als Trenner wird eine beliebige Anzahl von Leerzeichen erkannt.

3. Um die Wörter umzudrehen, verwenden Sie den Befehl:

```
sed -e 's/\([0-9]*\) *\([A-Za-z]*\) *\([A-Za-z]*\)/\3 \2 \1/;
s/^ //' downunder.txt
```

   Das zweite Substitute `s/^ //` entfernt alle Leerzeichen, die daraus resultieren, dass das dritte Wort fehlt.

4. Und hier der Sprücheklopfer in einer kurzen Version:


```
#!/bin/bash
# Fortune: Eine eigene Version. Zum zweiten
#
declare -a zeilen
declare -i anz nr von bis
zeilen=(`grep -n '^#' txt.txt | cut -d: -f1`)
anz=${#zeilen[@]}-2
nr=${RANDOM}%anz
von=${zeilen[nr]}+1
bis=${zeilen[nr+1]}-1
sed -n "${von},${bis}p" txt.txt
exit 0
```

5. Der Einzeiler könnte `sed -n -e "/2 Emu/,$ p" downunder.txt` lauten.

Kapitel 11

Reste und Sonderangebote

»Die Breite an der Spitze ist dichter geworden« –
Berti Vogts

... und Sie gehören jetzt schon fast dazu. Um die letzten Weihen zu erhalten, fehlt nur noch dieses Kapitel. Die Kapitel 12 und 13 beschäftigen sich mit Problemen, die aus unterschiedlichen Shells heraus resultieren sowie mit Methoden zur Fehlervermeidung und Fehlerbehebung. Mit anderen Worten: Zum letzten Mal lernen Sie hier neue Programmiertechniken. Die letzten beiden Kapitel stellen sicher, dass Ihre gelernten Techniken auch höheren Ansprüchen genügen.

Genug des Vorgeplänkels, was findet sich nun wirklich in diesem Kapitel? Im Prinzip alles, was ich aus den verschiedensten Gründen nicht in einem der bisherigen Kapitel eingebaut habe, aber zu wichtig ist, um unter den Tisch zu fallen.

11.1 Zeitgesteuertes Starten von Skripten

Alle in diesem Buch erstellten Skripten wurden immer aus der Loginshell aus aufgerufen. Nun ist das zum Testen sicherlich eine feine Sache, und Sie konnten und können jederzeit eingreifen und das Skript abbrechen, sollte etwas nicht wie erwartet laufen. Ziel jedoch sollte es sein, dass Ihre Skripten bei Bedarf auch im Hintergrund laufen können. Sie können nun mit einigem Recht

einwenden, mithilfe von & bzw. nohup hätten wir schon einige Skripten im Hintergrund gestartet.

Dies ist zwar korrekt, aber dennoch mussten Sie das Skript von Hand aus starten, und es lief sofort los. Was aber, wenn Sie Skripten zu einem Zeitpunkt laufen lassen wollen, an dem Sie nicht angemeldet sind. Oder was, wenn Sie ein Skript zur Datensicherung jede Nacht um 01:00 Uhr laufen lassen wollen? Unwahrscheinlich, dass Sie sich jedes Mal anmelden, um das Sicherungsskript zu starten!

Zur Lösung dieser Probleme wurden unter Unix die cron- und atd-Daemonen eingeführt. Sie werden durch die Programme crontab und at mit Aufträgen versorgt. Damit Sie in einer eventuellen Diskussion mit anderen Unixexperten auch verstanden werden, hier der Hinweis, dass der englische Begriff für Auftrag *job* lautet.

Es kann durchaus sein, dass auf Ihrem System der cron-Daemon beide Aufgabenbereiche verwaltet. Wichtig ist für Sie aber nur, wie Sie sicherstellen, dass Ihre Aufträge wie von Ihnen gewünscht gestartet werden. Welcher Prozess sich nun wirklich darum kümmert, ist eher von geringerem Interesse.

crontab dient dazu, Skripten wiederholt aufrufen zu können. Dabei haben Sie eine weitreichende Kontrolle, zu welchen Zeitpunkten ein Skript aufgerufen werden soll. Die Datei, in der Sie angeben, welche Skripten wann laufen, wird irritierenderweise auch crontab genannt.

Die Datei crontab enthält die Anweisungen, wann welche Skripten laufen. Dabei haben Sie eine weitreichende Kontrolle, zu welchen Zeitpunkten ein Skript aufgerufen werden soll. Diese Datei wird mithilfe des Befehls crontab editiert und an den cron-Daemon übergeben. Diese Namensgleichheit kann schon einmal zu Missverständnissen führen.

Im Gegensatz dazu können Sie mit at einen Job an den atd-Daemon übergeben, der genau einmal zu einem von Ihnen definierten Zeitpunkt ausgeführt und dann von der Liste der auszuführenden Aufträge (Jobs) gestrichen wird.

Wie Sie nun Jobs mittels at oder crontab eintragen, werden wir in den nächsten beiden Abschnitten genauer beleuchten. Vorher will ich Ihre Sinne noch für einige Fallstricke schärfen, die daraus resultieren, dass Ihr Skript im Hintergrund läuft.

1. Kein `read` oder `echo` ohne Umleitung

   Bedenken Sie, dass Ihr Skript losgekoppelt von einem Bildschirm läuft. Ein Lesen von der Tastatur kann nicht funktionieren! Wenn Sie Ausgaben machen, die nicht in eine Datei umgelenkt werden, so verhält sich der `cron`-Daemon unterschiedlich, abhängig davon, wie er konfiguriert wurde. Weitere Informationen zu dieser Konfiguration etwas später.

2. Anderes Environment

   Das Environment zum Zeitpunkt der Ausführung ist unterschiedlich zu dem Environment, wenn Sie das Skript interaktiv ausführen. Bei `at` wird die Umgebung so übernommen, wie sie aussah, als der Job an `atd` übergeben wurde. Ausnahmen sind die Variablen `DISPLAY` und `TERM` (steht z.B. auf `dumb`). Bei `crontab` wiederum wird das Environment des `cron`-Daemons zugrunde gelegt. `LOGNAME` (bzw. `USER` auf einigen Systemen) wird auf den Eigentümer der `crontab` gesetzt und `HOME` auf dessen Heimatverzeichnis, so wie dies in `/etc/passwd` eingetragen ist. `SHELL` wird auf `/bin/sh` gesetzt. Während `SHELL` und `HOME` innerhalb der `crontab` neu gesetzt werden dürfen, ist dies für `LOGNAME` nicht möglich.

Gehen Sie also nicht davon aus, dass nur, weil das Skript in der Loginshell lief, es auch im Hintergrund laufen wird. Stellen Sie sicher, dass die Werte bzw. Einstellungen im Environment, die Sie benötigen, eingetragen sind. So ist auch nicht sichergestellt, dass der `PATH` genauso aussieht, wie in der interaktiven Shell! Jetzt müssen Sie nur noch wissen, wie Sie feststellen können, ob sie im Hintergrund laufen oder nicht. Dazu gibt es einige Tipps:

- `echo $-`

  Gibt die aktuellen Optionen der Shell aus. Ist ein Eintrag `i` enthalten, so ist die Shell interaktiv.

- `tty`

  Gibt einen Fehler aus, wenn kein Bildschirm für das Skript zur Verfügung steht (Exitstatus ist 1). Dies kann nur sein, wenn das Skript im Hintergrund läuft. Ansonsten bekommen Sie Ausgaben wie `/dev/tty1`. Wollen Sie auf diese Art von Ausgaben verzichten und nur den Rückgabewert erhalten, so rufen Sie bitte `tty` mit dem Parameter `-s` auf.

- `PS1`

  wird meistens nur für interaktive Shells gesetzt.

Die Anzahl der Skripten, die Sie dem `cron`-Daemon zur Verwaltung übergeben, ist dabei weder für `crontab` noch für `at` limitiert.

11.1.1 at


```
at [-q <queue>] <zeit>
at [-q <queue>] -f <datei> <zeit>
at -l [-q <queue>]
at -d <job>
batch [-q <queue>] <zeit>
batch [-q <queue>] -f <datei> <zeit>
batch -l [-q <queue>]
batch -d <job>
```

`at` dient dazu, einen Job anzulegen, der genau einmal zu einer definierten Zeit läuft. Übernimmt `at` den Job und gibt ihn zur Bearbeitung an `atd` weiter, so wird die für `at` sichtbare Shellumgebung kopiert und dem geplanten Job zu seiner Ausführungszeit zur Verfügung gestellt. Ausgenommen davon sind die Variablen `TERM` und `DISPLAY`. Als Shell wird `/bin/sh` genutzt.

Auch, wenn es der Name suggeriert: `/bin/sh` ist nicht unbedingt die Bourneshell. Auf Linuxsystemen handelt es sich dabei in der Regel um einen symbolischen Link auf die Bash.

`<zeit>` definiert den Zeitpunkt, an dem das Skript ausgeführt werden soll. Dieser Ausdruck kann recht komfortabel definiert werden, sogar umgangssprachliche (englische) Ausdrücke sind erlaubt. Die Zeit wird grundsätzlich auf zwei Arten definiert: zu einem festen Zeitpunkt oder zu einem Zeitpunkt relativ zur aktuellen Zeit.

Tabelle 11.1: Mögliche Zeitangaben für at

Format	Beispiel	Bedeutung
`HH:MM`	`22:55`	Läuft zur definierten Uhrzeit. Liegt diese Zeit vor der aktuellen Zeit, so wird die Aufgabe am nächsten Tag ausgeführt.
`MMDDYY`	`031299`	Führt die Aufgabe am definierten Tag aus. Kann nur in Kombination mit einer Zeit verwendet werden und muss nach der Zeitangabe stehen.
`DD.MM.YY`	`12.03.99`	"
`MM/DD/YY`	`03/12/99`	"
`midnight`	`midnight`	Zeitangabe: Mitternacht
`noon`	`noon`	Zeitangabe: Mittags (12:00)

Format	Beispiel	Bedeutung
teatime	teatime	Offensichtlich wurde die erste Version von at von einem Briten geschrieben, die wichtige Teezeit (16:00 Uhr) durfte auf keinen Fall fehlen. Oh Mortimer, some things never change :)
today	22:03 today	Datumsangabe: heute/aktuelles Datum. Muss in Verbindung mit einer Zeitangabe angegeben werden.
tomorrow	12:00 tomorrow	Datumsangabe: morgen/aktuelles Datum. Muss in Verbindung mit einer Zeitangabe angegeben werden.
now + <offs>	now + 1 days	Sie können <offs> in Gestalt von minute(s), now + 1 minute, hour(s), day(s) bzw. week(s) definieren. Die Zeitangabe bezieht sich dann auf x Minuten, Stunden, Tage bzw. Wochen vom Zeitpunkt, an dem at aufgerufen wurde.

Wenn Sie keine <datei> angeben, so gibt at einen Prompt at> aus und wartet auf Ihre Eingaben. Diese Eingaben stellen normale Shellbefehle dar, die zur angegebenen <Zeit> so ausgeführt werden. Die Eingabe wird mit Strg+D (<EOT>) beendet und die Aufgabe an atd übergeben. Wird <datei> aber angegeben, so liest at die Datei zeilenweise aus und übergibt die so gelesenen Zeilen an atd. Wird das Dateiende erreicht, so wird die Aufgabe wiederum an atd übergeben.

Mit at -l können Sie sich eine Liste der Aufträge ausgeben lassen, die noch nicht ausgeführt wurden. Wenn Sie aus dieser Liste einen Auftrag löschen wollen, so rufen Sie at -d gefolgt von der Jobnummer auf.

```
buch@koala:/home/buch > at now + 2 minutes
at> ls
at> <EOT>
warning: commands will be executed using /bin/sh
job 20 at 1999-04-01 20:01
buch@koala:/home/buch > at -l
20      1999-04-01 20:01 a
buch@koala:/home/buch > at -d 20
buch@koala:/home/buch > at -l
buch@koala:/home/buch >
```

An Stelle von at -l können Sie atq und anstatt von at -d auch atrm nutzen.

batch bewirkt das Gleiche wie ein at. Allerdings werden die Aufträge bei batch nur dann ausgeführt, wenn es die Systemauslastung erlaubt. Auf mei-

nem Linuxsystem ist das eine Auslastung von weniger als 0,8 bzw. einem Wert, den der Superuser mithilfe von `atrun -l <auslastung>` definieren kann. Ist die Auslastung höher, so wird der Auftrag nicht ausgeführt und aus der Liste der auszuführenden Jobs entfernt.

Nutzen wir das neu erworbene Wissen dazu, den CW-Commander erneut zu verbessern. Verwenden wir `at`, um einen Prozess in den Hintergrund zu legen, der nach fünf Minuten prüft, ob sich der Inhalt des aktuellen Verzeichnisses geändert hat. Falls ja, so sendet er ein `SIGUSR1` an den Commander, worauf dieser das aktuelle Verzeichnis neu lädt und anzeigt.

Bevor wir loslegen, noch einige Überlegungen:

- Damit der Hintergrundprozess weiß, an wen er eventuell ein Signal schicken muss, braucht er zumindest die Prozess-ID des Commanders (`$$`).
- Außerdem benötigt der Hintergrundprozess Informationen über das aktuelle Verzeichnis. Um die Sache einfach (aber nicht narrensicher!) zu machen, wollen wir an dieser Stelle die Ausgaben von `wc `ls .`` miteinander vergleichen.
- Die Existenz der Variable `PS1` ist ein Hinweis darauf, dass Ihre Shell interaktiv ist (siehe Kapitel 8.7).

An dieser Stelle könnten wir jetzt wieder viel Platz schinden, indem wir das komplette Skript ausdrucken. Wir sparen uns das aber lieber für nützliche Tips auf und führen nur die Änderungen zur letzten Version auf. Demnach könnte eine mögliche Lösung wohl so aussehen:


```
# Skript 50:
#
# CW-Commander mit Änderungsüberprüfung
function CW_TrapSIGUSR1 ()
{ # Fängt Signal Usr1 ab und liest das akt. Verzeichnis nochmals
  CW_ReadDir "."
  CW_PrintDir 1 18 $tmpfile
  tput cup 22 8
}
function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     # Tarsicherung abbrechen
     kill -SIGTERM $prid 2>/dev/null
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $gvAtName
     at -d $gvLastAt
     prid=""
     CW_Print 0 0 "Sicherung abgebrochen!"
```

```
    else
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     rm -f $gvAtName
     at -d $gvLastAt
     echo
     exit 0
  fi
}
function CW_At()
{
 if [ "$gvLastAt" != "" ] ; then
    at -d $gvLastAt
    gvLastAt=""
 fi
 echo "# At-Skript zur Überwachung des akt. Verzeichnisses">$gvAtName
 echo "gvPID=$$" >>$gvAtName
 local lvInh=`wc \`ls .\` 2>/dev/null |tail -1`
 echo "gvInh='$lvInh'" >>$gvAtName
 echo "set -- \$gvInh" >>$gvAtName
 echo "gvZei=\$1 ; gvWort=\$2 ; gvByte=\$3" >>$gvAtName
 echo 'gvInh2=`wc \`ls .\` 2>/dev/null | tail -1`' >>$gvAtName
 echo "set -- \$gvInh2" >>$gvAtName
 echo 'if [ $1 -ne $gvZei -o $2 -ne $gvWort -o $3 -ne $gvByte ] ; then'
>>$gvAtName
 echo "   kill -SIGUSR1 \$gvPID" >>$gvAtName
 echo "fi" >>$gvAtName
 echo "exit 0" >>$gvAtName
 at -f $gvAtName now + 4 minutes 2>/dev/null
 gvLastAt=`at -l | tail -1 | cut -d" " -f1`
}
function CW_ReadDir ()
{
 offset=2
 zakt=2
 CW_At
 gvAltZeil=-1
 gvAltOffs=-1
 cd "$1" 2>/dev/null && verz=`pwd`
 CW_Print 0 0
 anz=`ls -ld .* *|tee $tmpfile | wc -l | cut -c-7`
 CW_Box 0 1 40 20 "$verz"
 rm -f $gvAtName
}
Mark=":"
gvpp="/usr/bin/"
tput clear
verz=${1:-"`pwd`"}
trap "CW_TrapSIGINT" SIGINT
```

```
trap "CW_TrapSIGUSR1" SIGUSR1
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn a = Archiv   d = Löschen
"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=2
tmpfile="/tmp/cwc$$.tmp"
tmptar="/tmp/backup"
tmptarcnt="/tmp/tar.cnt$$"
gvAtName="/tmp/at$$"
gvLastAt=""
cd "$verz"
typeset -i anz=`ls -lAd * | tee $tmpfile | wc -l | cut -c-7`
typeset -i zakt=2            # Zähler akt. Zeile
typeset -i gvAltOffs=-1      # Alter Offset
typeset -i gvAltZeil=-1      # Alte aktuelle Zeile
CW_At                        # Erste Prüfung aktivieren
until [ "$ein" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
rm -f $tmpfile $gvAtName
rm -f $tmptar $tmptarcnt
exit 0
```

Das Konzept der Prozessprioritäten aus Kapitel 8 können Sie auch bei `at` zum Einsatz bringen. Es gibt insgesamt 53 Warteschlangen, die von `a` bis `z` und von `A` bis `Z` durchnummeriert werden. Eine Warteschlange ist eine Liste von Aufträgen (Jobs), die noch ausgeführt werden müssen. Dabei ist `a` die Standardqueue für `at` und `b` die Queue für `batch`. Je später der Buchstabe der Queue im Alphabet auftaucht, desto höher ist die Priorität, mit der der Auftrag ausgeführt wird.

Wird für `<queue>` ein Großbuchstabe angegeben, so verhält sich `at` so, als ob `batch` aufgerufen worden wäre. Die 53. Queue ist die Queue =, in der Jobs aufgeführt werden, die gerade ausgeführt werden.

Bitte beachten Sie, dass `at` keine Prüfung durchführt. Ob die übergebenen Befehle korrekt sind, dafür müssen Sie sorgen. Falls Fehler auftreten, so gelten diese als Ausgabe des Skripts und werden so behandelt, wie Sie es vorgegeben haben. Das heißt, die Ausgaben werden per Mail an den Benutzer geschickt, der den Job mittels `at` erstellt hatte. Wie das genau geschieht, sehen wir uns etwas später in diesem Kapitel an.

11.1.2 cron


```
crontab <datei>
crontab -e
crontab -l
crontab -r
```

Im Gegensatz zu `atd` führt `cron` die ihm übergebenen Aufträge wiederholt aus. Dazu führt der Daemon für jeden Benutzer, der `cron` nutzt, eine `crontab` genannte Tabelle. Diese Tabellen werden unter dem Benutzernamen im `cron`-Verzeichnis (oft `/var/spool/cron/crontabs`) abgelegt. Das Format für alle darin abgelegten `crontabs` ist dabei identisch, während die globalen `crontabs` im Verzeichnis `/etc/` ein anderes Format haben.

Und noch ein wichtiger Hinweis: Leider hat fast jedes Unixsystem seine eigene Verwaltung von `cron`-Jobs. Vor allem das Format der Einträge in der Datei `crontab` kann sich erheblich unterscheiden. In diesem Abschnitt beschreiben wir die Linux-Version von Paul Vixie. Uns bekannte Unterschiede führen wir noch einmal gesondert in einem Hinweis am Ende dieses Abschnitts auf.

Kommentare werden durch das Doppelkreuz `#` eingeleitet. Wenn Sie die Shellumgebung der `cron`-Aufträge beeinflussen wollen, so können Sie Variablenzuweisungen in der `crontab` durchführen, die dann für alle Aufträge in der entsprechenden Tabelle gelten. Die Zuweisung funktioniert genau wie eine normale Zuweisung in einem Skript.

Jeder Auftrag wird in einer Zeile der `crontab` definiert. Dabei teilt sich eine Zeile in sechs Spalten auf, die durch Leerzeichen oder Tabulatoren voneinander getrennt werden. Dabei definiert die erste Spalte die Minuten (0-59), an denen das Skript ausgeführt wird. Die zweite Spalte definiert die Stunden (0-23), die dritte den Tag des Monats (1-31), die vierte den Monat (1-12), die fünfte den Wochentag (0-7, wobei 0 und 7 für Sonntag stehen). Die sechste Spalte enthält den Befehl, den `cron` ausführen soll.

Wenn Sie in der sechsten Spalte ein `%` angeben, so wird es in einen Zeilenumbruch umgeformt, falls nicht mit einem \ versehen. Für die so entstandenen Zeilen gilt: Die erste Zeile ist der Befehl, und alle folgenden Zeilen werden zur Standardeingabe des Befehls weitergeleitet.

Bei den ersten fünf Spalten können Sie mehrere Werte angeben, die Sie durch Kommata trennen können. Wenn Sie einen Bereich von x bis y angeben wollen, so trennen Sie die Zahlen durch ein Minuszeichen `-`, z.B.: `1-5`. Soll die Spalte alle gültigen Werte beinhalten, so können Sie ein * eintragen.

Möchten Sie einen Auftrag nur in Schritten von zwei Stunden ausführen, so können Sie durch ein / solche Schritte definieren: `*/3` in der ersten Spalte würde das Skript alle drei Minuten ausführen.

Die Variable `MAILTO` hat eine besondere Bedeutung. Die Ausgaben des ausgeführten Skripts werden an den Eigentümer der `crontab` per E-Mail geschickt. Wird `MAILTO` auf `""` (leer) gesetzt, so wird diese Mail nicht verschickt. Enthält sie eine gültige Mailadresse, so geht die Ausgabe per Mail an diese Adresse und nicht an den Besitzer der `crontab`.

Als Shell wird in der Regel die Bourne-Shell (`sh`) genutzt. Wie wir noch sehen werden (Kapitel 12), ist diese Shell nicht in der Lage, alle Funktionen der Bash (Bourne Again Shell) zur Verfügung zu stellen. Wollen Sie, dass Ihre Skripten von `cron` immer in der Bash gestartet werden, so setzen Sie die Variable `SHELL` auf `/bin/bash` bzw. den Pfad, unter dem sich bei Ihnen die Bash findet.

Bei Linux ist das im Allgemeinen nicht nötig, da `/bin/sh` dort in der Regel ein symbolischer Link auf `/bin/bash` ist.

Mit `crontab <datei>` können Sie die `<datei>` als `crontab` für den aktuellen Benutzer (oder den per Option `-u` angegebenen `<benutzer>`) anlegen. Möchten Sie eine existierende `crontab` editieren, so rufen Sie `crontab -e` auf. Der Befehl `crontab` ruft dazu den Editor auf, der in der Variable `EDITOR` oder `VISUAL` angegeben wurde. Wurde darin kein Editor gesetzt, so nutzt `crontab` den `vi`.

Die Aufträge der Tabelle können Sie mit `crontab -l` ausgeben lassen, und `crontab -r` löscht die Tabelle für den aktuellen Benutzer.

Schauen wir uns einmal ein Beispiel an:

```
# Wir nutzen immer die Bash
SHELL=/bin/bash
# Die Mails gehen immer an "koala", egal, wem diese crontab gehört
MAILTO=koala
#
# Dieses Skript alle 5 Minuten von Mitternacht bis
# (ausschließlich 1 Uhr) starten
*/5 0 * * *       $HOME/bin/tag.sh >> $HOME/tmp/out 2>&1
# Dieses Skript jeden 2. des Monats um 17:15 Uhr starten
15 17 2 * *     $HOME/bin/monat.sh
# Jeden Samstag/Sonntag bei Christa anfragen, wie weit sie mit
# dem Buch ist (um 14:00)
0 14 * * 6,7   mail -s "Wg. Buch" christa%Christa,%%Wie weit bist Du?%.%
```

Mit `mail` kann eine Mail aus der Shell heraus an den angegebenen Benutzer geschickt werden. Das Thema (engl. *Subject*) wird mit `-s` definiert, und alle

folgenden Eingaben bis zum ».« in der ersten Spalte enthalten den Nachrichtentext. Obige Zeile sähe in der Shell so aus:

```
buch@koala:/home/buch > mail -s "Wg. Buch" christa
Christa,
wie weit bis Du?
.
EOT
buch@koala:/home/buch >
```

Es gibt mindestens drei verschiedene Versionen von `cron`: Versionen, deren Quelltexte von einer BSD-Variante abstammen, System-V-Versionen und mindestens eine Linuxvariante. Wie mittlerweile schon fast üblich, bietet die Linuxversion einer Software die meisten Möglichkeiten:

- Die Ausgaben an den Benutzer der `crontab` schicken (können BSD-Versionen nicht).
- Die Ausgaben werden einer anderen (definierten) Person geschickt (geht nicht mit den System-V-Versionen).
- Die Ausgaben sollen verloren gehen, und es soll keine Mail verschickt werden (System-V-Versionen sind damit überfordert).

Ähnliches gilt für die Angabe der Zeitpunkte in der `crontab`. So ist es z.B. nicht in allen Versionen möglich, mehr als einen Bereich pro Spalte anzugeben. Ein `1-5,9-12` ist daher nicht überall erlaubt. Auch die »/«-Notation findet sich nur in den wenigsten `Cron`-Versionen.

- Die Zuweisungen von Variablen in den Crontabs werden nicht generell unterstützt.
- Spalten werden auf den verschiedenen Systemen anders getrennt. So kann es durchaus passieren, dass einige Versionen auf Leerzeichen bestehen und mit Tabs nicht klarkommen bzw. umgekehrt.
- `MAILTO` ist Linux-spezifisch.
- Unter SunOS gilt: Als Editor wird der in `VISUAL` eingetragene Editor genutzt oder `ed`. (Stellen Sie also unbedingt diese Variable ein, sonst wird es archaisch `:)`)

Deshalb ist es grundsätzlich eine gute Idee, die Manpages zu Rate zu ziehen. Falls Ihre Skripten `at`- oder `crontab`-Einträge automatisch generieren, nehmen Sie unbedingt das kleinste gemeinsame Vielfache aller Versionen.

Und noch ein Hinweis: Diese Beschreibungen von `at` und `crontab` beziehen sich auf einige Möglichkeiten für einen normalen Anwender. Der Superuser `root` hat dazu noch einige weitere Optionen, die aber an dieser Stelle nicht weiter vertieft werden sollen.

11.2 Tildeextension

Neben den Ersatzmustern, der Brace Extension aus Kapitel 2 und den Parametersetzungen aus Kapitel 5 gibt es noch eine Ersetzung, die die Shell ausführt: die Tildeerweiterung.

Fängt ein Wort mit einer Tilde ~ an, so werden alle Zeichen bis zum nächsten Schrägstrich / (Slash) oder (falls kein / folgt) alle restlichen Zeichen des Wortes als mögliche Anmeldung (sprich Benutzername) interpretiert. Ist dieser Benutzername gleich "", so wird der aktuelle Benutzer angenommen. Die Tilde wird dann durch das Heimatverzeichnis (`$HOME`) des Benutzers ersetzt. Ist `HOME` nicht im aktuellen Environment der Shell gesetzt, wird das Heimatverzeichnis des aktuellen Benutzers für die Tilde eingesetzt:

```
buch@koala:/home/buch > echo ~
/home/buch
buch@koala:/home/buch > echo ~christa
/home/christa
buch@koala:/home/buch >
```

Mittels ⇥ (bzw. Esc-↓) komplettiert die Bash den Benutzernamen, falls möglich. Eine Liste aller möglichen Benutzernamen unter Berücksichtigung der aktuellen Eingabe erhalten Sie durch Strg-x ↓.

Folgt ein Pluszeichen + der Tilde, so wird der Wert von `PWD` für ~+ eingesetzt. Folgt der Tilde ein Minuszeichen -, so wird der Wert von `OLDPWD` für ~- ersetzt.

```
buch@koala:/home/buch > cd org
buch@koala:/home/buch/org > echo ~+
/home/buch/org
buch@koala:/home/buch/org > echo ~-
/home/buch
uch@koala:/home/buch/org > cd -
buch@koala:/home/buch >
```

`cd -` setzt das aktuelle Arbeitsverzeichnis wieder auf das letzte Arbeitsverzeichnis.

11.3 eval


```
eval [<argument1> ...]
```

Die übergebenen Argumente werden eingelesen, ausgewertet und zu einem Befehl verknüpft, der dann ausgeführt wird. Der Rückgabewert des ausgeführten Befehls ist der Rückgabewert von `eval`. Wurden keine Argumente an

eval übergeben, so ist der Rückgabewert von eval gleich 0. eval wird genutzt, wenn die Shellumgebung durch den Befehl geändert werden soll.

Tolle Erklärung, nur wird nicht ganz klar, wie mächtig dieser Befehl sein kann.

In Kapitel 6 hatten Sie gelernt, wie Sie die Werte von Variablen abfragen, deren Name in einer anderen Variablen gespeichert war. Mittels

```
...
gvVar=Test
gvName=gvVar
echo ${!gvName}
...
```

konnten Sie den Inhalt von gvVar ausgeben. Allerdings war eine Zuweisung nicht möglich. Falls Sie es einmal probiert haben sollten, kam ein Fehler zustande:


```
buch@koala:/home/buch > gvVar=Test
buch@koala:/home/buch > gvName=gvVar
buch@koala:/home/buch > ${!gvName}=Dummy
bash: gvVar=Dummy: command not found
buch@koala:/home/buch >
```

Die Shell versucht offensichtlich, die Zeile als Befehl gvVar=Dummy auszuführen, und erkennt nicht, dass es sich um eine Zuweisung handelt. Mit eval erkennt die Shell aber diese Intention, und das gewünschte Ergebnis kommt zustande:

```
buch@koala:/home/buch > gvVar=Test
buch@koala:/home/buch > gvName=gvVar
buch@koala:/home/buch > eval $gvName=Dummy
buch@koala:/home/buch > echo $gvVar
Dummy
buch@koala:/home/buch >
```

Schreiben wir ein kleines Skript, das ausgibt, wie viele Dateiendungen die Dateinamen im aktuellen Verzeichnis haben. Als Parameter eins wird dem Skript ein Ersatzmuster übergeben, welches die zu untersuchenden Dateinamen definiert. Folgendes Beispiel:

```
buch@koala:/home/buch > ls t*
toc.txt   toc.txt~  trp.sh    trp.sh~   tst.ksh   tst.ksh~  tst.txt
tst.txt~
buch@koala:/home/buch > ./skript52.sh t\*
Ergebnis für toc=2
Ergebnis für trp=2
Ergebnis für tst=4
buch@koala:/home/buch >
```

Die Idee ist es, alle Dateinamen zu ermitteln und von diesen die Suffixe (Erweiterungen) zu entfernen. Die so erhaltenen Namen nutzen wir als Variablennamen, die mit 1 initialisiert und bei jedem weiteren Vorkommen um 1 erhöht werden. Die folgende Lösung geht dabei davon aus, dass nur Dateinamen auftauchen, die aus Buchstaben, Unterstrich und Ziffern bestehen, weil Variablennamen nur aus diesen Zeichen bestehen dürfen.


```
# Skript 52: Alle Suffixe für beliebige
#            Dateien zählen.
#
gvErg="/"
for gvDatei in $1 ; do
  gvVarname=${gvDatei%.*}
  if [ -z "`echo $gvErg | grep /$gvVarname/`" ] ; then
     gvErg="$gvErg$gvVarname/"
     eval typeset -i $gvVarname=1
    else
     eval $gvVarname=$gvVarname+1
  fi
done
IFS="$IFS/"
for i in ${gvErg:1} ; do
  echo -n "Ergebnis für $i="
  eval echo \${$i}
done
exit 0
```

11.4 dirname/basename


```
basename <Dateiname> [suffix]
dirname  <Dateiname>
```

Falls Sie von einem Dateinamen den Pfadanteil benötigen, so können Sie diesen mittels `dirname` bestimmen. Den reinen Dateinamen ohne Pfadanteil können Sie mit `basename` ermitteln.

Falls Sie noch `<suffix>` angeben, so wird außerdem noch ein eventuell existierender Suffix vom Dateinamen entfernt:

```
buch@koala:/home/buch > basename /tmp/ba.txt .txt
ba
buch@koala:/home/buch >
```

Wofür ist dies gut? Ein einfaches Beispiel: Sie erinnern sich bestimmt noch an das Skript 15 aus Kapitel 3. Wenn es mit kompletter Pfadangabe, aber falschen Parametern aufgerufen wurde, kam folgende Fehlermeldung:

```
buch@koala:/home/buch > /home/buch/skript15.sh
/home/buch/skript15.sh datei ab bis
   Gibt die Zeilen ab Zeile <ab> - <bis> der Datei <datei> aus
    datei  -> Datei aus der die Zeilen angezeigt werden sollen
    ab     -> Die Zeile der Datei, ab der angezeigt werden soll
    bis    -> Bis zur wievielten Zeile soll ausgegeben werden
buch@koala:/home/buch >
```

Dies sieht einfach nicht gut aus, `skript15.sh datei ab bis` wäre wünschenswert. Mit `basename` können Sie auch diesen Punkt auf der Liste der unerledigten Probleme abhaken, und glauben Sie uns, diese Liste schrumpft zusehends. Ob Sie die zusätzlichen Informationen des Pfads angeben oder nicht, bleibt aber Ihnen überlassen. Es kann unter Umständen durchaus hilfreich sein, zu wissen, von wo das Programm aufgerufen wurde.


```
# Skript 53: definierten Zeilenbereich ausgeben
# abgeänderte Version von Skript15.sh (Version 1)
#
if [ $# -ne 3 ] ; then
   echo "`basename $0` datei ab bis"
   echo "   Gibt die Zeilen ab Zeile <ab> - <bis> der Datei <datei> aus"
   echo "    datei  -> Datei, aus der die Zeilen angezeigt werden sollen"
   echo "    ab     -> Die Zeile der datei, ab der angezeigt werden soll"
   echo "    bis    -> Bis zur wievielten Zeile soll ausgegeben werden?"
   exit 1
fi
# Variablen zuweisen
datei=$1
bis=$3
ab=$2
#
# Berechne die letzte Zeile, die mit head auszugeben ist:
# anz = bis - von + 1
#
anz=`expr $bis - $ab + 1`
#
# und anzeigen
#
head -$bis $datei | tail -$anz
exit 0
```

Hier steht noch die Kombination mit `head` und `tail`. Dies ist natürlich mittlerweile keine akzeptable Lösung mehr für Sie. Besser wäre der Einsatz von `sed`, aber das heben wir uns für die Aufgaben auf :)

11.5 umask/ulimit


```
umask <Maske>
ulimit -n [<Grenze>]
ulimit -c [<Grenze>]
```

Wenn Sie eine Datei anlegen, dann werden dieser Datei Zugriffsberechtigungen zugeteilt. Diese Berechtigungen wurden bereits in Kapitel 1 angesprochen. Es werden drei Berechtigungen für Lesen, Schreiben und Ausführen verteilt. Diese werden wiederum für den Besitzer, die Gruppe und für den Rest definiert:

```
buch@koala:/home/buch > ls -l kapitel11.txt
-rw-r--r--   1 buch     users       28918 Apr  3 01:11 kapitel11.txt
buch@koala:/home/buch >
```

Der Benutzer darf in diesem Beispiel schreiben und lesen, die Gruppe `users` darf lesen, und der Rest der Welt darf ebenfalls lesen. Im Prinzip wird jede Berechtigung durch ein Bit repräsentiert. Ist das Bit 0, so ist das Recht nicht verfügbar, ansonsten ist es (Überraschung!) 1. Unter diesen Umständen könnte man die oben aufgeführten Rechte wie in Tabelle 11.2 darstellen.

Tabelle 11.2: Darstellung von Rechten als Oktalzahl

Benutzer	Gruppe	Rest
r w -	r - -	r - -
1 1 0	1 0 0	1 0 0
6	4	4

Interpretiert man die Zahlengruppen in der vorletzten Zeile als Binärzahlen, so ergibt sich 6 4 4. Da genau drei Stellen pro Benutzer/Gruppe/Rest vergeben werden, können die Werte für jede Zahl nur von 0 bis 7 reichen. Ein Zahlensystem, dem, wie hier erläutert, die Basis 8 zugrunde liegt, nennt man Oktalsystem.

Wenn Sie drei so berechnete Zahlen an `chmod` übergeben, können Sie ebenfalls Berechtigungen vergeben:

```
buch@koala:/home/buch > >wangaratta
buch@koala:/home/buch > ls -l wangaratta
-rw-r--r--   1 buch     users           0 Apr  3 01:36 wangaratta
buch@koala:/home/buch > chmod 755 wangaratta
buch@koala:/home/buch > ls -l wangaratta
-rwxr-xr-x   1 buch     users           0 Apr  3 01:36 wangaratta
buch@koala:/home/buch >
```

Jede von der Shell neu angelegte Datei bekommt zunächst einmal die Berechtigung `rw-rw-rw-` oder oktal `666`. Dies ist nur scheinbar ein Widerspruch mit der Ausgabe aus dem letzten Beispiel. An dieser Stelle kommt nämlich die `umask` (*User file-creation mask*) ins Spiel. Sie wird zunächst binär invertiert (NOT) und dann mit den ursprünglichen Rechten (`MODE`) mit `AND` verknüpft: `erg=MODE and (not UMASK)`.

```
buch@koala:/home/buch > umask
022
buch@koala:/home/buch >
```

```
NOT UMASK     ---> 111 101 101
AND MODE      ---> 110 110 110
```

Ergebnis `644  ---> 110 100 100`

Und das ist genau die Berechtigung, die der Datei `wangaratta` schließlich vergeben wurde. Wollen Sie für die Gruppe ebenfalls Schreibberechtigung vergeben, so setzen Sie die `umask` auf `002`:

```
buch@koala:/home/buch > umask 002
buch@koala:/home/buch > >wangaratta
buch@koala:/home/buch > ls -l wangaratta
-rw-rw-r--   1 buch     users           0 Apr  3 01:45 wangaratta
buch@koala:/home/buch >
```

Falls Sie sich ein wenig genauer mit der Erstellung von Dateien und der Vergabe der Berechtigungen beschäftigen wollen, werfen Sie doch einen Blick auf *creat(2)*. Dort werden Sie erkennen, dass die Dateien nicht von Anfang an die Rechte 666 bekommen, sondern dass dies in den Händen des Programmierers liegt. Die Bash vergibt jedenfalls bei uns zunächst einmal die Rechte 666.

Haben Sie Ihren lokalen Unixexperten schon mal fluchen hören? Vielleicht sagte er so was wie »core dumped und zugenäht«, und Sie haben sich überlegt, ob Sie sich wohl verhört hatten?

Wohl kaum. Wenn ein Prozess unsachgemäß beendet wird, z.B. durch bestimmte Signale (Speicherverletzung: `SIGSEGV` oder Division durch Null: `SIGFPE`), so legt Unix einen Speicherabzug des betroffenen Prozesses im Arbeitsverzeichnis des Prozesses ab. Da diese zum einen recht groß werden können und zum anderen nur den Programmierern etwas sagen, kann eingestellt werden, wie groß diese `core`-Dateien wirklich werden können. Für nicht programmierende (gemeint sind Compilersprachen wie C, C++ oder Pascal) Benutzer wird diese `<Grenze>` daher schon einmal auf `0` gesetzt.

Mit ulimit können Sie viele Einstellungen ermitteln und modifizieren, wir möchten in diesem Abschnitt nur auf die maximale Anzahl an offenen Dateien (Option -n) und die core-Dateien eingehen.

Eine <Grenze> kann zum einen hart (Option -H) oder weich (Option -S) sein. Eine harte <Grenze> kann nicht mehr verändert werden, während eine weiche Grenze zwischen 0 und der harten Grenze beliebig verschoben werden kann. Wird weder -S noch -H angegeben, so werden beide Grenzen gesetzt (in der Bash-Version 1.x nur das Softlimit). Wird die <Grenze> nicht angegeben, so wird die aktuelle Einstellung ausgegeben.

<Grenze> kann entweder unlimited oder eine Zahl größer oder gleich 0 sein.

```
buch@koala:/home/buch > ulimit -n
256
buch@koala:/home/buch > ulimit -c
1000
buch@koala:/home/buch > ulimit -c unlimited
buch@koala:/home/buch > ulimit -c
unlimited
buch@koala:/home/buch > ulimit -Hn 200
bash: ulimit: cannot modify limit: Operation not permitted
buch@koala:/home/buch >
```

Taucht die Fehlermeldung auf, so wurde die harte Grenze bereits einmal definiert. Eine erneute Veränderung einer harten Grenze ist für die Dauer der Anmeldung nicht erlaubt.

Die Beschränkung der Core-Größe auf 0 ist für einen normalen Anwender dann sinnvoll, wenn er nicht möchte, dass sein Plattenplatz von den core-Dateien belegt wird. Speicherabzüge sind für gewöhnliche Benutzer (und darunter fallen diesmal auch Skriptprogrammierer) kaum von Nutzen und belegen nur Platz, den man an anderer Stelle besser brauchen kann.

11.6 Prompts

In Kapitel 6 hatten wir bereits die Ausgaben der Prompts PS1 bis PS4 besprochen und untersucht, wie sie geändert werden können. In diesem Abschnitt möchten wir noch einige nette Tricks anbringen, die dort ungesagt blieben.

So soll es an dieser Stelle um die Manipulation des Prompts durch Escapesequenzen gehen. Dies sind Zeichenfolgen, die bestimmte Aktionen hervorrufen, wie z.B. die Farbe oder den Cursor setzen. Wir haben diese Funktionalität in Kapitel 4 mit dem Befehl tput kennen gelernt. Die Escapesequenzen bieten Ähnliches auf eine andere Art. Gut, wenn tput nicht verfügbar ist.

Diese Art der Escapesequenzen funktioniert nur bei der Bash! Bei anderen Shells kann dies zu seltsamen Effekten führen. Außerdem müssen die ANSI-Sequenzen auch vom Terminaltyp unterstützt werden. So klappt die Cursorpositionierung z.B. häufig unter `xterm` nicht.

Escapesequenzen müssen in der Bash in `\[` und `\]` eingeschlossen werden. Dadurch ignoriert die Shell diese Zeichen bei der Längenberechnung des Prompts. Wird dies vergessen, so steht der Eingabecursor viel weiter rechts vom Prompt, als erwartet.

Die eigentlichen Escapesequenzen werden durch ein `\033[` eingeleitet. Schauen Sie sich die Liste aller verfügbaren Escapesequenzen in Tabelle 11.3 an.

Tabelle 11.3: Escapesequenzen der Bash

Vordergrund			
Schwarz	`0;30`	Dunkelgrau	`1;30`
Blau	`0;34`	Hellblau	`1;34`
Grün	`0;32`	Hellgrün	`1;32`
Cyan	`0;36`	Helles Cyan	`1;36`
Rot	`0;31`	Hellrot	`1;31`
Lila	`0;35`	Helles Lila	`1;35`
Braun	`0;33`	Gelb	`1;33`
Hellgrau	`0;37`	Weiß	`1;37`
Hintergrund			
Schwarz	`40`		
Blau	`44`		
Grün	`42`		
Cyan	`46`		
Rot	`41`		
Lila	`45`		
Braun	`43`		
Hellgrau	`47`		
Reguläre Attribute	`0m`		

Reguläre Attribute setzt die Farben etc. auf die Werte vor der Ausgabe von `PS1` zurück. Für den Hintergrund gibt es keine hellen Farben.

Die Attribute der Farbe sind durch den Wert vor dem Semikolon bestimmt. Deren Bedeutungen sind in Tabelle 11.4 aufgeführt.

Tabelle 11.4: Farbattribute für Escapesequenzen

Attribut	Bedeutung
0	Normal
1	Bold. Wird hell ausgegeben.
4	Unterstrichen. Nur, wenn der Zeichensatz dies unterstützt. PC-Textmodi beherrschen dies meistens nicht.
5	Blinken. Wenn es wirklich nerven soll, dann lassen wir es blinken.
7	Revers.
8	Versteckt. Gibt die Zeichen nicht aus. Gut z.B. für Kennworteingaben.

Als Letztes gibt es noch einige Sequenzen für die Positionierung des Cursors (Tabelle 11.5).

Tabelle 11.5: Sequenzen für die Positionierung des Cursors

\033[<Zeile>;<Spalte>H	Cursor positionieren: Zeile geht von 1 bis $LINES, Spalte von 1 bis $COLUMNS.
\033[<Anzahl>A	Bewegt den Cursor <Anzahl> Zeilen nach oben.
\033[<Anzahl>B	Bewegt den Cursor <Anzahl> Zeilen nach unten.
\033[<Anzahl>C	Bewegt den Cursor <Anzahl> Spalten nach rechts.
\033[<Anzahl>D	Bewegt den Cursor <Anzahl> Spalten nach links.
\033[s	Speichert die aktuelle Cursorposition. Wird nicht von allen Terminalemulatoren unter X unterstützt.
\033[u	Cursor auf die gespeicherte Position setzen. Wird nicht von allen Terminalemulatoren unter X unterstützt.

Nutzen wir dieses Wissen für einen total überflüssigen, aber hübschen Prompt: den Uhrenprompt. Dieser gibt oben rechts immer die aktuelle Zeit aus. Auf den Terminalemulatoren geht dieser Prompt leider schief, da diese die aktuelle Cursorposition nicht abspeichern.


```
# Skript 54: Ein etwas anderer Prompt
#
# Läuft nur auf Textkonsolen (nicht auf Xterms)
function uhr ()
{
  local       BLAU="\[\033[1;34m\]"
  local        ROT="\[\033[0;31m\]"
  local    HELLROT="\[\033[1;31m\]"
  local      WEISS="\[\033[1;37m\]"
```

```
  local      RESET="\[\033[0m\]"
  local       SAVE="\[\033[s\]"
  export PS1="\[$SAVE\033[1;\$(echo -n \${prompt_x})H\]\
$BLAU[$HELLROT\$(date +%k:%M)$BLAU]\[\033[u\033[1A\]
$BLAU[$HELLROT\u@\h:\w$BLAU]\
$WEISS >$RESET "
}
# Verhindern, dass der alte Prompt wieder eingesetzt wird
prompt_x=$((COLUMNS-8))
export PROMPT_COMMAND=""
# Und aktivieren
uhr
```

Was macht der Prompt nun genau?

1. Escapesequenz einleiten: `\[`
2. Cursorposition sichern: `$SAVE`
3. Positionierung oben rechts (1 Zeile, 8 Zeichen links vom rechten Rand)
4. Ende Escapesequenz 1
5. Farbe auf Blau umsetzen und [ausgeben: `$BLAU[`
6. Farbe auf Hellrot umsetzen und Zeit ausgeben
7. Farbe auf Blau setzen und] ausgeben: `$BLAU]`
8. Alte Cursorposition laden und eine Zeile hoch (verhindert einen Zeilenumbruch): `\[\033[u\033[1A\]`
9. Und nun Farbe auf Blau und [ausgeben: `$BLAU[`
10. Farbe auf Hellrot setzen und Benutzernamen, Rechnernamen und Verzeichnis ausgeben: `$HELLROT\u@\h:\w`
11. Blau setzen und Klammer zu ausgeben: `$BLAU]`
12. Abschließend ein weißes > und die alte Farbe wiederherstellen für die Benutzereingabe: `$WEISS >$RESET`

Das `exit` haben wir bewusst herausgelassen, damit das Skript mit `. ./skript54.sh` aufgerufen werden kann.

Viel Spaß beim Probieren.

11.7 alias/unalias


```
alias <name>=<befehl>
unalias <name>
```

Ein Alias ist ein anderer Name für einen bekannten Befehl. Dieser Alias wird von der Shell verwaltet, es wird kein neuer Befehl im Dateisystem angelegt, sondern die Shell verwaltet eine Tabelle, welcher Alias für welchen realen Befehl steht. Eine Liste aller angelegten Aliase wird durch die einfache Eingabe von `alias` ausgegeben:

```
buch@koala:/home/buch > alias
alias dir='ls -l'
alias l='ls -alF'
alias la='ls -la'
alias ll='ls -l'
alias ls='ls --color=tty'
alias ls-l='ls -l'
alias md='mkdir -p'
alias rd='rmdir'
alias unzip='unzip -L'
buch@koala:/home/buch > alias dir
alias dir='ls -l'
buch@koala:/home/buch >
```

Ein neuer Alias wird wie eine Zuweisung erstellt. Wichtig ist nur der Befehl `alias` davor:

```
buch@koala:/home/buch > alias adm="ps ax|wc -l"
buch@koala:/home/buch >
```

Ein Alias wird durch den Befehl `unalias` aufgehoben:

```
buch@koala:/home/buch > type dir
dir is aliased to `ls -l'
buch@koala:/home/buch > unalias dir
buch@koala:/home/buch > type dir
dir is /usr/bin/dir
buch@koala:/home/buch >
```

Im Unterschied zu den Shellfunktionen ist ein Alias nur ein anderer Name für eine mögliche Shelleingabe. So ersetzt die Shell intern den Alias durch die entsprechende Zeichenkette und wertet diese dann aus. So ist es z.B. nicht möglich, einem Alias Parameter zu übergeben. So wäre ein solcher Alias nicht möglich:


```
buch@koala:/home/buch > alias cw='ps ua | grep "^$1" | wc -l'
buch@koala:/home/buch > cw html
wc: html: No such file or directory
buch@koala:/home/buch >
```

Der Fehler resultiert daraus, dass $1 dem Alias gar nichts sagt und durch "" ersetzt wird. Somit führt die Bash folgende Zeile aus:

```
buch@koala:/home/buch > ps ua | grep "^" | wc -l html
```

Und da diese Datei (html) nicht existiert, tritt der Fehler auf.

Da die Bash etwas verwirrende Regeln hat, wann ein Alias aktiv wird, hier zwei wichtige Hinweise:

- Steht eine Alias-Definition mit mehreren Befehlen in einer Zeile, so ist der Alias für diese Befehle noch unbekannt. Der Alias wird erst aktiv, wenn die nächste Zeile des Skripts eingelesen wird!
- Wird ein Alias in einer Funktion definiert, so kann der Alias erst nach dem Verlassen der Funktion genutzt werden.

Deshalb:

- Alias immer einzeln in einer Zeile definieren.
- Alias möglichst nicht in einer Shellfunktion anlegen.

11.8 Startvorgang

Noch einige kurze Worte zum Startvorgang der Bash. Eine Loginshell ist eine Shell, die gestartet wird, nachdem der Benutzer sich mit Namen und Kennwort angemeldet hat. Die Bash wird in der Regel durch Angabe der Option --login zur Loginshell.

Eingabe und Ausgabe einer interaktiven Shell sind mit Tastatur bzw. Bildschirm verbunden. Interaktive Shells können Sie explizit mit der Option -i starten. In beiden Fällen enthält $- ein i, welches anzeigt, dass die Bash interaktiv ist.

Wird die Bash interaktiv gestartet, so führt sie die Befehle aus /etc/profile aus, sollte diese Datei existieren. Danach werden im Heimatverzeichnis des aktuellen Benutzers die Dateien .bash_profile, .bash_login und .profile in dieser Reihenfolge herangezogen. Die erste gefundene Datei wird genau wie /etc/profile ausgeführt. Soll keine dieser drei Dateien im Heimatverzeichnis ausgeführt werden, so kann die Option --noprofile angegeben werden.

Wird eine (Bash-)Loginshell beendet, so führt dies die Datei `$HOME/.bash_logout` aus, falls diese Datei existiert.

Wird eine interaktive Shell gestartet, die *keine* Loginshell ist, so führt die Bash Befehle aus der Datei `$HOME/.bashrc` aus (falls vorhanden). Mit `--rcfile <datei>` können Sie eine andere `<datei>` anstelle von `.bashrc` definieren.

Wird die Bash nicht interaktiv gestartet (z.B. von `at` oder `cron`), so wertet sie den Inhalt der Variable `BASH_ENV` aus und interpretiert ihn als Dateinamen, der auszuführen ist. Sie verhält sich so wie folgende Skriptzeilen:

```
if [ -n "$BASH_ENV" ] ; then
   . "$BASH_ENV"
fi
```

Wird Bash unter dem Namen `sh` aufgerufen, so versucht die Shell sich so ähnlich wie nur möglich wie die Original-`sh` zu verhalten, ohne den POSIX-Standard zu verletzen. Eine Loginshell führt erst `/etc/profile` aus und danach `$HOME/.profile`, falls diese Datei existiert.

Eine `sh`, die nicht interaktiv ist, führt keinerlei Dateien aus!

11.9 xargs

Kratzen wir an dieser Stelle noch mal an den etwas höheren Shellweihen und kommen noch mal auf ein Problem zurück, welches uns durch das gesamte Buch ein wenig verfolgt hat: Dateinamen mit Leerzeichen und deren Weiterverarbeitung. Sicherlich erinnern Sie sich nur zu gut an die Probleme und ihre Lösungen (Stichwort Quoting).

Im Zusammenhang mit dem Befehl `find` möchten wir Ihnen an dieser Stelle noch zwei Alternativen vorstellen. Zum einen bietet `find` die Option `-exec`. Diese Option erwartet einen gültigen (Shell-)Befehl und einen Platzhalter `{}`, den `find` durch die gefundenen Dateinamen ersetzt.

Stehen wir nun im Unterverzeichnis `/home/buch/projects/Buch/` und wollen wissen, welche Textdateien bereits für die neue Version dieses Buches überarbeitet wurden, so hilft uns folgende Zeile:

```
buch@koala:~/projects/Buch > find . -name "*.txt" -exec ls -l {} \;
-rw-r--r--    1 prog   users     19934 Jun 27 20:12 ./pdkshnotes.txt
-rwxrwxrwx    1 prog   users      3345 Jun 27 21:29 ./Texte/anhangd.txt
-rw-r--r--    1 prog   users     47626 Jul  5 09:05 ./Texte/kapitel11.txt
-rw-r--r--    1 prog   users         2 Jun 28 19:14 ./Texte/Aenderungen.txt
-rwxrwxrwx    1 prog   users     41070 Jun 28 21:12 ./Texte/kapitel10.txt
-rw-r--r--    1 prog   users       174 Jun 28 19:57 ./test.txt
-rw-r--r--    1 prog   users       275 Jun 28 21:09 ./hinweis.txt
-rw-r--r--    1 prog   users         0 Jul  5 09:16 ./Ge mein heit.txt
p
buch@koala:~/projects/Buch >
```

Neben einigen Arbeitsdateien im höher liegenden Verzeichnis haben wir zu diesem Zeitpunkt offensichtlich noch nicht allzuviel korrigiert/erweitert. Aber für einen Dateinamen mit Leerzeichen hat dies offensichtlich schon gereicht...

`-exec` interpretiert alle Zeichen bis zum folgenden Semikolon als auszuführenden Befehl. Da die Shell ein Semikolon aber als Ende eines Befehls erachtet, muss dieses Semikolon durch ein Fluchtzeichen vor der »Vernichtung« durch die Shell bewahrt werden. Damit die von `find` gefundenen Dateinamen auch an der richtigen Stelle im Befehl eingesetzt werden können, wird ein Platzhalter benötigt. Dies sind die geschweiften Klammern `{}`.

Diese Methode hat allerdings einen Nachteil: Sie führt für jeden gefundenen Dateinamen den durch `-exec` angegebenen Befehl aus. Dies kann zu einer großen Anzahl von Prozessen führen, was das System stark belasten kann.

Da die meisten UNIX-Befehle mehr als einen Parameter entgegennehmen, wäre es wünschenswert, möglichst nur einen Prozess zu starten. Die Lösung ist uns bereits bekannt, es sind die Ausführungszeichen `` ` ``. Allerdings handeln wir uns nun den Nachteil ein, dass die Shell nur eine begrenzte Anzahl an Zeichen in der Kommandozeile akzeptiert und den Rest ignoriert. Gerade bei sehr großen (verschachtelten) Verzeichnissen kann diese Grenze aber schnell erreicht sein. Außerdem führen Dateinamen mit Leerzeichen zu Problemen, was also tun?

Für diese Fälle gibt es den Befehl `xargs`. Er nimmt ähnlich wie die `-exec`-Option von `find` einen Befehl als Parameter und übergibt diesem dann die per Standardeingabe übergebenen Dateinamen als Parameter. Dabei beachtet `xargs` die maximale Länge der Kommandozeile und, falls die droht überzulaufen, startet es den Befehl erneut mit den restlichen Parametern. Unser Beispiel von oben:

```
buch@koala:~/projects/Buch > find . -name"*.txt" -print | xargs ls -l
ls: ./Ge: Datei oder Verzeichnis nicht gefunden
ls: mein: Datei oder Verzeichnis nicht gefunden
ls: heit.txt: Datei oder Verzeichnis nicht gefunden
-rw-r--r--    1 prog   users        41 Jul  5 09:16 ./Texte/Aenderungen.txt
-rwxrwxrwx    1 prog   users      3345 Jun 27 21:29 ./Texte/anhangd.txt
-rwxrwxrwx    1 prog   users     41070 Jun 28 21:12 ./Texte/kapitel10.txt
-rw-r--r--    1 prog   users     49448 Jul  5 09:23 ./Texte/kapitel11.txt
-rw-r--r--    1 prog   users       275 Jun 28 21:09 ./hinweis.txt
-rw-r--r--    1 prog   users     19934 Jun 27 20:12 ./pdkshnotes.txt
-rw-r--r--    1 prog   users       174 Jun 28 19:57 ./test.txt
buch@koala:~/projects/Buch >
```

Und wieder einmal gibt es Probleme mit den Leerzeichen im Dateinamen. Aber auch dies lässt sich relativ einfach umgehen. Wie Sie bereits wissen, unterteilt die Shell die Wörter/Befehle ja mithilfe der Variable `IFS`. Eine Lösung

wäre hier die einzelnen Zeilen mit sed mit Anführungszeichen zu versehen, aber das heben wir uns für eine der Aufgaben auf …

Zumindestens die GNU-Versionen bieten hier zwei verschiedene Abhilfemöglichkeiten:

```
find . -name "*.txt" -printf \"%p\"\\n | xargs ls -l
```

Die Option -printf gibt ähnlich wie der printf-Befehl aus Kapitel 4.2 die gefundenen Dateinamen in einem vom Benutzer definierten Format aus. An dieser Stelle wollen wir den kompletten Pfadnamen (%p) in Anführungszeichen setzen und mit einem Zeilenumbruch abschließen. Die Backslashes verhindern hierbei eine ungewünschte Interpretation durch die Shell. Weitere Informationen über die formatierte Ausgabe finden sind unter *find(1)*.

Die zweite Möglichkeit wäre die Nutzung von Feldtrennern, die garantiert nie in einem Dateinamen auftreten können. Auf Unixsystemen gibt es tatsächlich so ein Zeichen, das Zeichen mit dem Wert 0 (auch Nullbyte genannt). Mittels Option -print0 gibt find die Dateinamen mit diesem Trenner aus. Damit xargs dies auch richtig erkennt, benötigt dieses die Option -0:

```
buch@koala:~/projects/Buch > find . -name "*.txt" -print0 |xargs -0 ls -l
-rw-r--r--    1 prog   users         0 Jul  5 09:16 ./Ge mein heit.txt
-rw-r--r--    1 prog   users        41 Jul  5 09:16 ./Texte/Aenderungen.txt
-rwxrwxrwx    1 prog   users      3345 Jun 27 21:29 ./Texte/anhangd.txt
-rwxrwxrwx    1 prog   users     41070 Jun 28 21:12 ./Texte/kapitel10.txt
-rw-r--r--    1 prog   users     52896 Jul  5 10:05 ./Texte/kapitel11.txt
-rw-r--r--    1 prog   users       275 Jun 28 21:09 ./hinweis.txt
-rw-r--r--    1 prog   users     19934 Jun 27 20:12 ./pdkshnotes.txt
-rw-r--r--    1 prog   users       174 Jun 28 19:57 ./test.txt
buch@koala:~/projects/Buch >
```

Verwechseln Sie das Nullbyte nicht mit dem Zeichen 0. Unter Unix und den meisten anderen Systemen werden alle Zeichen mit Werten von 0 bis 255 versehen. Die Tabelle, welcher Wert welches Zeichen darstellt, nennt man Zeichensatz und in diesem Falle nutzen wir den ASCII Zeichensatz. Darin ist festgelegt, dass das Zeichen 0 den Wert 48 hat, das Leerzeichen den Wert 32 usw. Neuere Systeme nutzen mittlerweile Werte bis 65535, um Zeichen zu kodieren. Dies wird dann Unicode genannt.

11.10 Aufgaben

1. Angenommen, Sie rufen `at -f MeineBefehle.sh now + 1 minute` auf. Wenn die Datei `MeineBefehle.sh` existiert und korrekte Shellbefehle enthält, muss sie dann ausführbar sein, damit `atd` den so erstellten Job ausführen kann?
2. Der CW-Commander ist eher unvollständig, was die Überwachung betrifft. Zum einen gibt es sicher bessere Methoden, zu prüfen, ob sich der Verzeichnisinhalt geändert hat (Nebenfrage: Welche fallen Ihnen denn so ein?), zum anderen treten Probleme auf, wenn der Benutzer das aktuelle Verzeichnis im Commander ändert. Angenommen, wir stehen in `/home/buch/koala` und setzen den Auftrag per `at` auf. Was passiert, wenn wir das Verzeichnis auf `/home/buch` ändern und sich vor dem Ablauf des ersten Auftrags der Inhalt von `/home/buch/koala` ändert? Wir bekommen ein Signal `SIGUSR1`, und das Verzeichnis `/home/buch` wird neu geladen, obwohl es sich nicht geändert haben muss! Löschen Sie den letzten Überwachungsauftrag, bevor Sie einen neuen aufsetzen!
3. Es gibt mindestens noch einen logischen Fehler im letzten Skript, zumindestens so, wie es kodiert wurde. Was passiert, wenn Sie mehrere Jobs mithilfe von `at` und `batch` aufgesetzt haben? Tipp: Was gibt `at -l` aus?
4. Wie können Sie ohne die GNU-Versionen von `find` und `xargs` die Probleme mit den Leerzeichen im Programmnamen umgehen?

11.11 Lösungen

1. Die Datei muss nicht ausführbar sein, weil `at` nur die Befehle aus der Datei liest, die Datei selbst jedoch nicht ausführt.
2. Die Lösungen entnehmen Sie bitte dem nachfolgenden Skript (`CW_ReadDir` und `CW_At`). Wir haben es uns an dieser Stelle etwas vereinfacht, denn wir gehen davon aus, dass unser Job der letzte in der Liste von `at -l` ist.


```
# Skript 50:
#
# CW-Commander mit Änderungsüberprüfung

function CW_TrapSIGUSR1 ()
{ # Fängt Usr1 ab und liest das akt. Verzeichnis nochmal
 CW_ReadDir "."
 CW_PrintDir 1 18 $tmpfile
 tput cup 22 8
}
```

```
function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     # Tarsicherung abbrechen
     kill -SIGTERM $prid 2>/dev/null
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $gvAtName
     at -d $gvLastAt
     prid=""
     CW_Print 0 0 "Sicherung abgebrochen!"
 else
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     rm -f $gvAtName
     at -d $gvLastAt
     echo
     exit 0
 fi
}

function CW_At()
{
 # Existiert ein alter Job, dann löschen
 if [ "$gvLastAt" != "" ] ; then
    at -d $gvLastAt
    gvLastAt=""
 fi
 echo "# At-Skript zur Überwachung des akt. Verzeichnisses">$gvAtName
 echo "gvPID=$$" >>$gvAtName
local lvInh=`wc \`ls .\` 2>/dev/null |tail -1`
 echo "gvInh='$lvInh'" >>$gvAtName
 echo "set -- \$gvInh" >>$gvAtName
 echo "gvZei=\$1 ; gvWort=\$2 ; gvByte=\$3" >>$gvAtName
 echo 'gvInh2=`wc \`ls .\` 2>/dev/null | tail -1`' >>$gvAtName
 echo "set -- \$gvInh2" >>$gvAtName
 echo 'if [ $1 -ne $gvZei -o $2 -ne $gvWort -o $3 -ne $gvByte ] \
           ; then' >>$gvAtName
 echo "   kill -SIGUSR1 \$gvPID" >>$gvAtName
 echo "fi" >>$gvAtName
 echo "exit 0" >>$gvAtName
 at -f $gvAtName now + 4 minutes 2>/dev/null
 #
 # Hier die queue angeben, um 3) zu lösen!!!
 gvLastAt=`at -l -q a| tail -1 | cut -d" " -f1`
}
```

```
function CW_ReadDir ()
{
 offset=2
 zakt=2
 CW_At   # Löscht den letzten At-Auftrag und setzt ihn neu
 gvAltZeil=-1   # Fehlte in letzter Version! Fehler
 gvAltOffs=-1   # Dito
 cd $1 2>/dev/null && verz=`pwd`
 CW_Print 0 0
 anz=`ls -ld .* *|tee $tmpfile | wc -l | cut -c-7`
 CW_Box 0 1 40 20 "$verz"
 # rm -f $gvAtName
}

function CW_PrintDir()
{
  # Ausgabe der Dateien
  typeset -i xp=$1
  typeset -i hoehe=$2
  ausdatei=$3
  typeset -i i=0
  typeset -i akt=0
  while [ $i -lt $hoehe -a $akt -lt $anz ] ; do
  akt=i+offset
  if [ $gvAltOffs -eq $offset ] ; then
      let "$akt == $gvAltZeil || $akt == $zakt"
      if [ $? -eq 0 ] ; then
         local lvOut="ja"
      else
         local lvOut="nein"
      fi
  else
      local lvOut="ja"
  fi
  if [ "$lvOut" = "ja" ] ; then
     zeile=`head -$akt $ausdatei | tail -1`
      set -- $zeile
      revers=`echo "$Mark" | grep ":$9:" `
      if [ -n "$revers" ] ; then
         CW_SetRev
      fi
      if [ $zakt -eq $akt ] ; then
         CW_SetBold
      fi
      local datei=`echo $9 | cut -c-15`
      local text="`printf "$1|%9i | %-15s" $5 $datei`"
      CW_Print $xp $((i+2)) "$text"
      CW_SetNormal
   fi
   i=i+1
 done
```

```
 while [ $i -lt $hoehe ] ; do
    CW_Print $xp $((i + 2)) "`printf \"%10s|%10s|%16s\"`"
    i=i+1
 done
}

function CW_NormalFile () {
# Schaut nach, welchen Dateityp die Datei hat und reagiert auf Tar / Zip
# Archive
typ=`file $1 | cut -f2 -d":"`
if [ "$typ" = " GNU tar archive" -o "$typ" = " POSIX tar archive" ] ; then
   echo "Dummy" >$tmptarcnt
   tar tvf $1 | awk \
    '{ print $1 " 1 r r " $3 " " $4 " " $5 " " $6 " " $8 }' >>$tmptarcnt
   boffs=$offset
   offset=2
   banz=$anz
   anz=`cat $tmptarcnt |wc -l`
   CW_PrintDir 41 18 $tmptarcnt $1
   offset=$boffs
   anz=$banz
fi
}

Mark=":"
tput clear
verz=${1:-"`pwd`"}
trap "CW_TrapSIGINT" SIGINT
trap "CW_TrapSIGUSR1" SIGUSR1
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn a = Archiv   d = Löschen
"
#
# Zeilen-Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=2
tmpfile="/tmp/cwc$$.tmp"
tmptar="/tmp/backup"
tmptarcnt="/tmp/tar.cnt$$"
gvAtName="/tmp/at$$"
gvLastAt=""
cd $verz
typeset -i anz=`ls -ld .* * | tee $tmpfile | wc -l | cut -c-7`
typeset -i zakt=2         # Zähler akt. Zeile
typeset -i gvAltOffs=-1   # Alter Offset
typeset -i gvAltZeil=-1   # Alte aktuelle Zeile
CW_At
```

```
until [ "$ein" = "q" ] ; do
  CW_PrintDir 1 18 $tmpfile
  CW_Eingabe
done
rm -f $tmpfile
rm -f $tmptar $tmptarcnt
exit 0
```

3. Unser Skript geht an dieser Stelle davon aus, dass `at` unseren Job immer als letzten Job in der Liste ausgibt. Dies ist kein Problem, wenn Sie ansonsten keine `at`-Jobs aktiviert haben, aber wenn diese Bedingung nicht mehr erfüllt ist, kann durchaus der falsche `at`-Job ermittelt und gelöscht werden. Besser wäre an dieser Stelle, die `at`-Queue oder das Datum unseres Jobs zu suchen.

4. Eine mögliche Lösung sieht wie folgt aus:

```
buch@koala:~/projects/Buch > find . -name "*.txt" -print| sed -n -e
's/^.*/"&"/;p'|xargs ls -l
-rw-r--r--    1 prog   users         0 Jul  5 09:16 ./Ge mein heit.txt
-rw-r--r--    1 prog   users        41 Jul  5 09:16 ./Texte/Aenderungen.txt
-rwxrwxrwx    1 prog   users      3345 Jun 27 21:29 ./Texte/anhangd.txt
-rwxrwxrwx    1 prog   users     41070 Jun 28 21:12 ./Texte/kapitel10.txt
-rw-r--r--    1 prog   users     52896 Jul  5 10:05 ./Texte/kapitel11.txt
-rw-r--r--    1 prog   users       275 Jun 28 21:09 ./hinweis.txt
-rw-r--r--    1 prog   users     19934 Jun 27 20:12 ./pdkshnotes.txt
-rw-r--r--    1 prog   users       174 Jun 28 19:57 ./test.txt
buch@koala:~/projects/Buch >
```

Kapitel 12

Die Kornshell und Portabilität

»Die Anzahl der Neider bestätigt unsere Fähigkeiten« –
Oscar Wilde

Wenn Sie an dieser Stelle angelangt sind, können Sie sicher sein, dass Sie einige Neider haben, die Sie um Ihre Fähigkeiten im Rahmen der Skriptprogrammierung beneiden. In diesem Kapitel wollen wir uns noch mit zwei Themengebieten auseinander setzen: der Kornshell und der Portabilität.

12.1 Kornshell

Am Anfang der Unixgeschichte gab es die Bourneshell `/bin/sh`. Diese hatte eine mächtige Skriptsprache, war aber für interaktive Benutzung eher ungeeignet. Um die Interaktion zu verbessern, wurde an der Universität von Kalifornien in Berkeley eine neue Shell namens C-Shell entwickelt, deren Skriptsprache der Sprache C ähnelte und einige neue Konzepte wie Jobverwaltung und Alias beinhaltete.

Während diese Shell Stärken in der Interaktion brachte, stellte die Skriptsprache einen herben Rückschlag dar. Ergebnis war, dass die C-Shell als interaktive Shell und die Bourne-Shell als Skriptingshell genutzt wurde. Um das Lernen von zwei Shells zu vermeiden, entwickelte David Korn bei AT&T die nach ihm benannte Kornshell `ksh`. Diese nutzte eine verbesserte Version der Bourneshell-Skriptsprache und übernahm gleichzeitig die besten Features für die Interaktion von der C-Shell. Leider hatte diese Shell einen Nachteil: Sie war nicht frei erhältlich, sondern musste von AT&T erworben werden.

Etwa zu dieser Zeit entwickelte sich das GNU-Projekt, welches ebenfalls eine neue Shell brauchte. Auch hier entschied man sich für eine Shellsprache auf Basis der Bourneshell und verbesserte Fähigkeiten im Bereich der Interaktion, ähnlich denen der C-Shell und Kornshell. Der Name dieser Shell: Bourne Again Shell (ein englisches Wortspiel: Born again Shell, wiedergeborene Shell): `Bash`.

Dies ist also der Grund, warum `ksh`, `sh` und `bash` sich stark ähneln. Leider heißt *ähneln* nicht *identisch sein*, weshalb die Unterschiede fein sind. Während in den bisherigen Kapiteln meistens die Bash im Mittelpunkt stand, soll dieses Kapitel sich ein wenig mehr um die Aspekte der Kornshell kümmern. Eine Übersicht der wichtigsten Fähigkeiten dieser drei Shells findet sich als Gegenüberstellung in Anhang A.

Leider stand uns zum Test nur ein Linuxsystem mit der `pdksh` (Public Domain Kornshell) zur Verfügung. Da diese jedoch nicht alle Funktionen der originalen Kornshell unterstützt, laufen die Skripten so möglicherweise nicht mit der `ksh`. An Punkten, wo uns Unterschiede bekannt sind, finden Sie entsprechende Hinweise. Schauen Sie deshalb auch noch zusätzlich in die Manpage der Kornshell.

12.1.1 Parameter jenseits $9

Wie Sie bereits erfahren haben, können Sie in der Bash auch auf mehr als neun Positionsparameter zugreifen, die Sie einem Skript übergeben möchten (`${10}`, `${11}`...). Die Kornshell erlaubt diesen Zugriff ebenfalls, während die `sh` diese Möglichkeit nicht zur Verfügung stellt.

12.1.2 Weitere Ersatzmuster in der ksh

Auch Ersatzmuster sollten Ihnen nun kein Fremdwort mehr sein. In diesem Abschnitt werden Sie noch weitere Ersatzmuster kennen lernen, die Ihnen die Kornshell bietet.

Alle Beispiele beziehen sich dabei auf folgendes Verzeichnis:

```
Kornshell:/home/buch/ksh > ls
a        amr       cwcwcwmrmrmrcwmr   mrcw
abc      amrcw     cwmrcwmr           mrcwmr
acw      cw        mr                 mrmrmrcwcw
Kornshell:/home/buch/ksh >
```

In der `ksh` können Sie so genannte *Musterlisten* definieren. Eine Musterliste ist eine Zeichenkette, die durch `|` separiert ist. So steht `a|b|c` für die Zeichen `a`, `b` und `c`. Groß-/Kleinschreibung wird hierbei berücksichtigt. Anhand des `ls`-Befehls möchten wir Ihnen die Arbeitsweise dieser Ersatzmuster aufzeigen.

Sie sind aber nicht auf den `ls`-Befehl beschränkt, wie Sie schon bei den allgemeinen Ersatzmustern in Kapitel 2 gesehen haben, wo wir `echo` genutzt haben. Dies steht Ihnen auch hier offen.

Geben Sie vor der Musterliste ein `*(<Musterliste>)` an, werden Ihnen alle Dateinamen zurückgegeben, in denen eine Kombination der Zeichenketten aus der Musterliste einmal oder mehrfach an der entsprechenden Stelle auftritt oder in denen die Zeichenketten der Musterliste nicht auftreten, weil die Länge des Dateinamens an der Stelle der Musterliste bereits endet.

```
Kornshell:/home/buch/ksh > ls a*(mr|cw)
a      acw    amr    amrcw
Kornshell:/home/buch/ksh >
```

Das heißt, `(mr|cw)` steht in diesem Fall für beliebige Kombinationen und Wiederholungen von `mr` und `cw` am Ende des Dateinamens oder für eine leere Zeichenkette.

```
Kornshell:/home/buch/ksh > ls *(mr|cw)
cw                cwmrcwmr          mrcw              mrmrmrcwcw
cwcwcwmrmrmrcwmr  mr                mrcwmr
Kornshell:/home/buch/ksh >
```

Damit kommen Sie möglicherweise nicht in allen Fällen aus. Wenn es unerwünscht ist, dass beliebige Kombinationen oder Wiederholungen der Zeichenketten Ihrer Musterliste das Muster erfüllen, dann geben Sie statt des `*` ein `?` an.

Ein `?` vor der Musterliste, d.h. `?(<Musterliste>)`, liefert alle Dateinamen zurück, in denen genau eine der Zeichenketten der Liste enthalten ist, oder Dateinamen, in denen keine Zeichenkette der Musterliste enthalten ist, weil der Dateiname kürzer als das angegebene Ersatzmuster ist.

```
Kornshell:/home/buch/ksh > ls ?(mr|cw)
cw  mr
Kornshell:/home/buch/ksh > ls a?(mr|cw)
a    acw  amr
Kornshell:/home/buch/ksh >
```

Geben Sie vor der Musterliste ein `+(<Musterliste>)` an, werden Ihnen alle Dateinamen zurückgegeben, in denen wenigstens eine Zeichenkette der Musterliste einmal oder mehrfach auftritt.

```
Kornshell:/home/buch/ksh > ls a+(mr|cw)
acw    amr    amrcw
Kornshell:/home/buch/ksh >
```

In diesem Beispiel deckt das Ersatzmuster also nicht mehr die Datei `a` ab, wie dies der Fall war beim Befehl `ls a*(mr|cw)`.

Wollen Sie das Auftreten einer Zeichenkette der Musterliste oder eine beliebige Kombination der Zeichenketten innerhalb des Dateinamens abdecken, dann wird Ihnen das folgende Beispiel nützlich sein:

```
Kornshell:/home/buch/ksh > ls a*+(b|c)*
abc   acw   amrcw
Kornshell:/home/buch/ksh >
```

Ein `@(<Musterliste>)` deckt *eine* Zeichenkette der Musterliste exakt an der angegebenen Stelle ab. Der Befehl `ls @(a|b|c)*` ermittelt Ihnen beispielsweise alle Dateinamen, die mit `a`, `b` oder `c` beginnen.

```
Kornshell:/home/buch/ksh > ls @(mr|mrmr|cw)
cw  mr
Kornshell:/home/buch/ksh > ls a@(mr|bc|cw)
abc  acw  amr
Kornshell:/home/buch/ksh >
```

Der Dateiname `amrcw` wird nun nicht mehr zurückgegeben, da beim `@`-Befehl Kombinationen der Zeichenketten aus der Musterliste nicht mehr abgedeckt werden, wie bei `+(...)`.

Durch `!(<Musterliste>)` decken Sie alle Zeichenketten ab, die dem angegebenen Muster nicht entsprechen, d.h., das Muster enthält keine der Zeichenketten der Musterliste an der entsprechenden Stelle, wobei Kombinationen der Zeichenketten enthalten sein dürfen.

```
Kornshell:/home/buch/ksh > ls a!(mr|bc|cw)
a      amrcw
Kornshell:/home/buch/ksh >
```

Die in diesem Absatz beschriebenen Ersatzmuster sind nicht auf den `ls`-Befehl beschränkt. Dazu noch ein kleines Beispiel. Da es Kornshell-spezifisch ist, sagen wir in der ersten, mit `#!` beginnenden Zeile explizit, dass wir zur Ausführung `/usr/bin/ksh` aufrufen wollen, und nicht etwa `/bin/sh`.


```
#!/usr/bin/ksh
# Eingabe prüfen auf Ganzzahl
typeset -i zahl
case $1 in
   +(0|1|2|3|4|5|6|7|8|9) ) zahl=$1;;
   *) echo "Falsche Eingabe (keine Ganzzahl)" >&2
      exit 1;;
esac
...
```

12.1.3 [[– Bedingte Ausdrücke/Conditional Expressions

[[wird von der Shell fast identisch wie [und damit wie `test` ausgewertet. Folgende Unterschiede sind dabei zu beachten:

- UND `-a` und ODER `-o` sind durch `&&` bzw. `||` ersetzt worden und entsprechen damit der Notation in C.
- Es gibt neue Operatoren, wie z.B. `-a <datei>` (gibt 0 zurück, wenn die `<datei>` existiert) oder `-o <option>` (gibt wahr zurück, wenn die `<Option>` aktiviert ist. Dies verwirrt stark, da `[] -a` und `-o` für UND- bzw. ODER-Verknüpfungen verwendet.
- Keine Dateinamenerweiterung: `[[ -n kap*.txt ]]` fragt ab, ob die Zeichenkette `kap*.txt` ungleich `""` ist. Was der Fall ist. Selbst wenn keine Datei diesem Muster entspricht, ist das Ergebnis immer noch wahr.
- Das zweite Argument bei Vergleichen mit `!=`, `=`, und `==` darf Ersatzmuster enthalten. In diesem Fall wird ein reiner Zeichenvergleich durchgeführt (wie bei der Brace Extension). So ist `[[ "koala" == [kK]oala ]]` wahr, während `[[ [kK]oala == "koala" ]]` falsch ist. (Das erste Argument beachtet keine Muster.)
- `[ str ]` steht für `[ -n str ]`. Eine solche verkürzte Schreibweise ist nicht erlaubt für `[[`. Sie müssen also immer `[[ -n str ]]` ausschreiben.
- Boolesche Ausdrücke werden nicht komplett ausgewertet, wenn bereits durch einen Teilausdruck das Ergebnis des Gesamtausdrucks feststeht (*Short-Circuit-Evaluation* heißt dieses Vorgehen im Compilerbau).

  ```
  [[ -x dolphin && $(dolphin) == "delphin" ]]
  ```

  Ist die Datei `dolphin` nicht ausführbar, wird `'$(dolphin) == "delphin"'` gar nicht mehr ausgewertet, da FALSE und FALSE / TRUE immer FALSE ergibt.

Wenden wir uns abschließend dem folgenden Skript zu. Das Skript `chk.ksh` testet, ob im aktuellen Verzeichnis Dateien mit der Endung `sh` stehen, die nicht mit der Zeile `#!/bin/bash` beginnen. Falls eine solche Datei gefunden wird, listet das Skript diese Dateien zur Überprüfung auf. Dabei gehen wir davon aus, dass die Bash unter `/bin/bash` zu finden ist, was meist nur auf Linuxsystemen der Fall ist. Eine korrekte erste Zeile mit `#!/bin/bash` wird hier auch beanstandet.


```
#!/usr/bin/ksh
# Shell für die Kornshell chk.ksh
# Testet, ob evtl. ".sh" als ausführbar
# gekennzeichnet sind und in der ersten Zeile
# kein "#!/bin/bash" steht
#
if ls *.sh 2>/dev/null ; then
   for gvDatei in *.sh ; do
      echo "Teste $gvDatei..."
      [[ -x $gvDatei && "$(head -1 $gvDatei )" != "#!/bin/bash" ]] && \
      echo "$gvDatei bitte überprüfen, ob die Zeile \"#!/bin/bash\" fehlt."
      [[ ! -x $gvDatei && "$( head -1 $gvDatei )" = "#!/bin/bash" ]] && \
      echo "$gvDatei bitte Berechtigungen überprüfen auf executable (x)"
   done
fi
exit 0
```

Die einzigen Dateien, die bereits mit der Zeile `#!/bin/bash` ausgestattet sind, sind die Dateien `skript1.sh` und `skript2.sh`. Hier die Rechte an den Textdateien:

```
Kornshell:/home/buch/Skripten > ls -l *.sh
-rw-r--r--   1 buch     users       22869 Mar  3 21:49 skript1.sh
-rw-rw-rw-   1 buch     users       23961 Apr 14  1998 skript10.sh
-rw-r--r--   1 buch     users       46227 Apr 15 22:35 skript11.sh
-rw-rw-rw-   1 buch     users        5477 Apr 18 14:56 skript12.sh
-rw-r--r--   1 buch     users       34227 Apr 16 23:07 skript13.sh
-rwxr-xr-x   1 buch     users       33436 Mar  8 21:18 skript2.sh
-rwxr-xr-x   1 buch     users       41562 Mar  8 21:18 skript3.sh
-rw-r--r--   1 buch     users       30048 Feb 23 20:55 skript4.sh
-rw-r--r--   1 buch     users       31712 Mar 31 22:51 skript5.sh
-rw-r--r--   1 buch     users       41530 Mar  6 12:36 skript6.sh
...
Kornshell:/home/buch/Skripten > ./chk.ksh
Teste skript1.sh...
skript1.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript10.sh...
skript10.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript11.sh...
skript11.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript12.sh...
skript12.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript13.sh...
skript13.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript2.sh...
Teste skript3.sh...
skript3.sh bitte überprüfen, ob die Zeile "#!/bin/bash" fehlt.
Teste skript4.sh...
skript4.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript5.sh...
```

```
skript5.sh bitte Berechtigungen überprüfen auf executable (x)
Teste skript6.sh...
...
Kornshell:/home/buch/Skripten >
```

12.1.4 $(< ...)-Umlenkung

Eine Umleitung durch `$(<abc)` bewirkt grundsätzlich das Gleiche wie der Befehl `$(cat abc)`: Die angegebene Datei wird auf der Standardausgabe ausgegeben. Für `$(cat abc)` muss die `ksh` allerdings einen zusätzlichen Prozess starten, was immer Zeit kostet, und damit nicht so effektiv ist wie eine Umlenkung durch `$(<abc)`.

Schreiben wir ein kleines Skript, welches als Parameter einen Verzeichnisnamen (dabei sind Wildcards erlaubt) und ein Suchmuster erwartet. Dieses suchen wir in allen darin enthaltenen Dateien und geben am Ende aus, wie viele Zeilen das Muster enthalten:


```
#!/usr/bin/ksh
# Skript show.ksh
#
# Ermittelt alle Dateien im Verzeichnis $1, die
# die Zeichenkette $2 enthalten
#
typeset -r gcTmpfile="/tmp/show$$.tmp"
rm -f $gcTmpfile
gvSDatei=""
# Dateinamen ermitteln in Liste
for gvDatei in $1 ; do
  [ -r "$gvDatei" ] && gvSDatei="$gvDatei $gvSDatei"
done
# Die Dateinamen ermitteln, in denen der Suchbegriff existiert,
# und das Ergebnis ausgeben
[[ -n "$gvSDatei" ]] && \
    grep -l "$2" $gvSDatei >$gcTmpfile && \
    wc -l $(<$gcTmpfile)
rm -f $gcTmpfile
exit 0
```

Ein Beispiel, das den Dateinamen und die ersten 79 Zeichen (genau 80 - 3 - Länge des Dateinamens) der Datei ausgibt, verwendet die Variante der Umlenkung:

```
#!/usr/bin/ksh
# Skript show.ksh
# Gibt die ersten 80 Zeichen aus Dateien aus
#
```

```
for gvDatei in $1 ; do
    gvZeile=`echo $gvDatei : $(<$gvDatei) | tr -d "\n" | cut -c-79`
    echo $gvZeile
done
exit 0
```

12.1.5 Co-Prozesse: |&

Wird ein Befehl oder eine Pipeline mit `|&` abgeschlossen, dann wird dieser Befehl oder die Pipeline in den Hintergrund gelegt. Mit `read -p` bzw. `print -p` kann dann nach einem solchen Vorgang *asynchron* auf die Pipeline zugegriffen werden:


```
#!/usr/bin/ksh
# Asynchrone Pipeline: Lesen
#
ls |&
while read -p gvDatei ; do
  echo "Datei : $gvDatei"
done
exit 0
```

Zu einem Zeitpunkt kann immer nur ein Co-Prozess auf einem Kanal aktiv sein. Daher wird auch kein Mechanismus benötigt, um genau einen Co-Prozess z.B. über eine Nummer zu identifizieren. Um einen weiteren Co-Prozess starten zu können, muss deshalb zunächst der alte beendet werden oder seine Ein-/Ausgabe (mindestens aber seine Eingabe) einem anderen Kanal zugeordnet werden. Soll ein Co-Prozess auf diese Weise umgeleitet werden, so stehen dafür die Befehle `<&p` und `>&p` zur Verfügung. Diese leiten die Standardeingabe bzw. Standardausgabe des Co-Prozesses um.

Wie kann die Umlenkung eines Co-Prozesses permanent gültig gemacht werden?

Der Befehl `exec` wird normalerweise verwendet, um die Shell zu überlagern. Ein durch `exec <Kommando>` angegebenes Kommando wird anstelle der Shell ausgeführt, ohne dass dazu ein neuer Prozess generiert wird. Damit die Umlenkung eines Co-Prozesses permanent gültig wird (und nicht nur für einen Befehl), muss der Befehl `exec` genutzt werden. `exec` führt in diesem Zusammenhang dazu, dass die Umlenkung bis zur nächsten Abmeldung oder bis der Kanal geschlossen wird gültig bleibt.

Folgendes Skript nummeriert die Ausgaben von `ls` über einen Co-Prozess:


```
#!/usr/bin/ksh
cat -n - |&
for gvName in `ls` ; do
    print -p "$gvName"
done
exec 3>&p    # Eingabe des Co-Prozesses permanent auf Kanal 3
exec 3>&-    # Eingabe des Co-Prozess beendet
# Die Pipe kann aber noch ausgelesen werden:
#
while read -p gvEin ; do
   echo $gvEin
done
exit 0
```

Die pdksh verhält sich etwas anders als die ksh, was die Behandlung von Co-Prozessen betrifft. Die Kornshell schließt die Pipe, in die der Co-Prozess schreibt, wenn der zuletzt gestartete Co-Prozess beendet wurde, während die pdksh die Pipe erst schließt, wenn alle Co-Prozesse, die in diese Pipe schreiben, beendet wurden. Dies kann zu Problemen führen.

Über den Befehl `cat -n - |&` wird `cat` als Co-Prozess gestartet und kann asynchron angesprochen werden. Die Eingabe des Co-Prozesses, d.h. die Eingabe von `cat`, ist in diesem Fall die Ausgabe von `print -p`. Über den Befehl `exec 3>&p` wird die Ausgabe von `print -p` permanent auf Kanal 3 umgelenkt. `print -p` schickt die Dateinamen über `exec` zeilenweise an `cat`. Am Ende der For-Schleife wird `EOF` über `exec` an `cat` geschickt, und `cat` beendet sich, während die Pipe lesbar bleibt. Durch `exec 3>&-` wird die Eingabe des Co-Prozesses geschlossen. Durch `read -p` kann die Ausgabe des Co-Prozesses weiterhin ausgelesen werden.

Natürlich kann die Pipe nicht alle Daten aufnehmen, die zwischen Co-Prozess und Shell ausgetauscht werden. Irgendwo muss eine Grenze sein. Diese Grenze liegt bei Linux bei ca. 8 Kbyte, während Solaris die Grenze bei ca. 20 Kbyte zieht. Stellen Sie also sicher, dass die Daten nicht zu groß werden, die die Pipe zwischenspeichern soll.

12.1.6 Eingabe-Prompt:

In der Bash kann über die Option `-p` beim `read`-Befehl ein Prompt angegeben werden:

```
read -p "Eingabe bitte:" lvVar
```

Da die Option -p in der ksh bereits reserviert ist für Co-Prozesse, kann ein Prompt entweder mit

```
echo -n "Eingabe bitte:" ; read lvVar
```

realisiert werden oder, was nur in der ksh so erlaubt ist, durch:

```
read lvVar?"Eingabe bitte:"
```

12.1.7 Variablen

Benötigen Sie einmal die Version der verwendeten Kornshell, so steht Ihnen die Shellvariable KSH_VERSION zur Verfügung (siehe auch Anhang A).

Ein kleines Beispiel, wie die Variable KSH_VERSION genutzt werden kann:

```
if [ -n "$KSH_VERSION" ] ; then
   read lvVar?"Eingabe bitte:"
  else
   # sh oder Bash
   echo -n "Eingabe bitte:"
   read lvVar
fi
```

12.2 Portabilität

Skripten haben es an sich, dass sie Probleme lösen, die sich mit einfachen Bordmitteln so erst einmal nicht lösen lassen. Außerdem stellt sich Ihnen das Problem häufiger, denn sonst würde eine Programmierung kaum Sinn machen, schließlich wären Sie per Hand wahrscheinlich schneller, wenn es darum geht das Problem *einmal* zu lösen. Wenn sich dieses Problem nicht nur Ihnen, sondern auch anderen stellt, dann hat es gute Chancen sich zu verbreiten und somit nicht mehr nur auf Ihrer Systemkonfiguration zu laufen. Spätestens dann zahlt es sich aus, wenn Ihr Skript portabel gestaltet wurde.

Portabilität bedeutet in unserem Fall entweder die Möglichkeit, unser Skript für alle in diesem Buch besprochenen Shells lauffähig zu gestalten oder unser Skript auf mehr als einem Betriebssystem zum laufen zu bringen, denn wie bereits ganz am Anfang des Buches erwähnt: Bash, ksh und sh laufen nicht nur unter Linux oder Unix, sondern auch z.B. unter BEOS, QNX oder Windows.

Und noch ein Punkt ist zu beachten: Shells sind auch Programme und Programme werden den Wünschen ihrer Benutzer angepasst, entwickeln sich also weiter. Das führt zwangsläufig dazu, dass der Leistungsumfang der Shell von Version zu Version wächst. Somit kann es durchaus sein, dass ein Leistungsmerkmal erst in der Bash 2.02 auftaucht und in allen vorherigen Versionen nicht verfügbar ist. Probleme dieser Art, die uns bekannt sind, haben wir in Anhang A aufgeführt.

12.2.1 Portabilität über Shells hinweg

Portabilität kann recht wichtig sein, insbesondere dann, wenn Sie nicht genau wissen, auf welchem Unix-System Ihr Skript ablaufen wird. In diesem Fall sollten Sie

- sich auf den kleinsten gemeinsamen Nenner einigen. `sh` z.B. nur nutzen, wenn Sie keine Ersetzungen wie `${#var}` oder `${!var}` benötigen. Gleiches gilt für Ersatzmuster oder Shellvariablen/Shellbuiltins oder Konstrukte wie die Co-Prozesse, die nur unter der `ksh` oder `bash` verfügbar sind.
- eine Shell festlegen, falls dies nötig sein sollte: `#!/bin/bash`, `#!/usr/bin/ksh`

Während man die `ksh` fast auf allen Systemen (unter `/usr/bin/ksh`) findet, gilt das für die Bash nicht. So wird man sie als `/bin/bash` wohl nur unter Linux antreffen, ansonsten lohnt es sich, in `/usr/local/bin/bash`, `/usr/gnu/bin/bash`, `/opt/gnu/bin/bash` zu suchen. Damit ist die Portabilität allerdings verloren gegangen, da das Skript (Zeile 1) angepasst werden muss. Zusätzlich ist unter Linux `/bin/sh` nur ein Verweis auf `/bin/bash`, wodurch sich die Bash überwiegend so wie die original Bourneshell zu verhalten versucht.

- Variablen nutzen: So enthält die Variable `PAGER` den Pager (z.B. `less`, `more` oder `pg`), den der Benutzer einzusetzen wünscht. Gleiches gilt für die Variablen `VISUAL` und/oder `EDITOR`. Sie können diese Variablen ja mittels Parameterersetzung (siehe Kapitel 5.4.1) mit Standardwerten belegen, falls diese Variablen wider Erwarten nicht gesetzt sein sollten.
- eigene Funktionen schreiben, falls ein benötigter Befehl nicht verfügbar ist:

```
function PO_FindDir ()
{ # Aufruf: PO_FindDir "Muster"
  # find wurde nicht gefunden. Daher versuchen wir es mit
  # locate.
  echo "find Befehl nicht verfügbar. Nutze locate!" >&2
  echo "Stellen Sie sicher, dass locatedb auf dem akt. Stand ist!" >&2
  if [ $# -eq 1 ] ; then
     local lvErg=""
     for lvDatei in $(locate $1) ; do
         [ -d "$lvDatei" ] && lvErg="$lvErg $lvDatei"
     done
  fi
  echo $lvErg
}
for gvVar in $(PO_FindDir "/home/buch") ; do
    echo "Verzeich: $gvVar"
done
exit 0
```

Dieses Skript geht davon aus, dass wir genau eine Funktionalität von `find` ersetzen wollen, und zwar den Aufruf `find $1 -type d -print`. Um den ganzen `find` zu ersetzen, bedarf es wesentlich mehr Aufwand.

- shellspezifische Funktionen nur aufrufen, wenn Sie überprüfen, welche Shellversion eingesetzt wird:

```
[ "$SHELL" == "/bin/bash" ] && \
[ ${PIPESTATUS[0]} -ne 0 ] && echo "Fehler in der Pipe!" >&2
```

- shellabhängige Lösungen durch allgemeinere Versionen ersetzen.

  Beispiel: statt `echo ${!var}` besser `eval echo \$$var`

- Falls Sie nicht sicher sind, dass Ihr Skript in einer bestimmten Shell läuft, und testen wollen, ob ein bestimmter Befehl verfügbar ist, so müssen Sie zunächst prüfen, in welcher Shell Ihr Skript läuft, und dann eine passende Prüfung einleiten, ob der benötigte Befehl existiert.

Angenommen, Sie sind sich nicht sicher, ob `wc` verfügbar ist, womit Sie die Anzahl an Zeilen in einer Datei zählen wollen, so könnten Sie Folgendes machen:

```
...
gvwc=""
# In der KSH
[ -n "$KSH_VERSION" ] && gvDummy=`whence -p wc` || gvwc="sed -n -e '$='"
# In den anderen Shells
[ -z "$KSH_VERSION" ] && gvDummy=`type wc` || gvwc="sed -n -e '$='"
# ist gvwc hier noch leer, so trat kein Fehler auf und
# wc existiert im Pfad
[ -z "$gvwc" ] && gvwc="wc -l"
# Und ein Demoaufruf
ls -l | $gvwc
...
```

Wenn allerdings kein `wc` verfügbar ist, stimmt eh etwas nicht. Dieser Befehl sollte eigentlich überall existieren. (Er sollte hier nur als ein einfaches Beispiel dienen.)

12.2.2 Portabilität über Betriebssystemgrenzen

Zugegeben, dieses Buch ist im Zweifelsfalle eher in Richtung Linux ausgelegt als in Richtung eines anderen Betriebssystems. Zwar haben wir überall, wo uns Unterschiede bekannt waren, dies deutlich im Text benannt, aber grundsätzlich ist Shellprogrammierung nicht auf Linux oder Unix festgelegt. Somit kann es durchaus passieren, dass ihr Skript unter Linux entwickelt und getestet wurde, jedoch auch unter *BSD zum Einsatz kommt. Auch in diesem Falle sind einige Dinge zu beachten, um Portabilität zumindestens wahrscheinlicher zu machen:

- Beschränkung auf weit verbreitete Kommandozeilenbefehle. Nutzen Sie nur Befehle, die bekanntermaßen auf (fast) allen gängigen Unixplattformen verfügbar sind (z.B. `cp`, `mv`, `mkdir`, `tar`, `ls`, `cat`, `true`, `test`)
- Nutzen Sie nur Optionen, die nicht abhängig sind von einer bestimmten Implementation. So hat `tar` z.B. in der GNU-Version die Option "`z`", welche das Archiv automatisch während des Anlegens komprimiert. Leider kann dann dieses Archiv z.B. unter SUN Solaris nicht mehr gelesen werden (es sei denn unter SUN wären die GNU-Tools installiert, in welchem Falle allerdings ein `gtar` existieren würde plus das Original `tar` von SUN).
- Verzichten Sie auf die ausführlichen Optionen, wie sie die meisten GNU-Tools anbieten und meist durch ein doppeltes Minuszeichen eingeleitet werden. Wenn nicht, stellen Sie sicher, dass entweder die Tools installiert sind und reagieren Sie entsprechend falls nicht.
- Verzichten Sie möglichst auf systemabhängigen Code. Nutzen Sie keine Befehle, die sich unter den verschiedenen Betriebssystemen unterschiedlich verhalten und am Ende noch total unterschiedliche Optionen unterstützen. Bestes Beispiel hierfür ist `ps`.
- Falls es sich gar nicht verhindern lässt, dass Sie betriebssystemabhängigen Code schreiben, dann fassen Sie diesen in Funktionen zusammen und rufen diese Funktionen abhängig von `uname` auf:

```
#!/usr/bin/ksh
case $( uname ) in
   AIX)
      # Aufruf der AIX spezifischen Teile hier oder
      # Aufruf der passenden eigenen Funktion an dieser Stelle:
      OS_Aix ;;
   HP-UX)
      # Dito für HP-UX
      OS_HP ;;
   Linux)
      # und Linux
      OS_Linux ;;
   *)
      echo "Nicht unterstütztes OS"
      exit 1  ;;
esac
```

Versuchen Sie aber in Ihrem eigenen Interesse, solche Abfragen möglichst selten zu machen, da sie mit jedem weiteren Betriebssystem, welches unterstützt werden soll, wachsen. Schließlich ist Ihr Code nur noch eine Skriptwüste, was wir ja vermeiden wollen. Der beste Rat an dieser Stelle, wenn Sie portable skripten wollen, ist der: Vermeiden Sie systemabhängige Abfragen. Ehrlich, wir sind uns sicher ;)

Hier noch eine Tabelle mit den Ausgaben von `uname` auf einigen uns bekannten Systemen:

Unixvariante	Ausgabe
AIX	AIX
IRIX	IRIX*
Linux	Linux
SOLARIS	SunOS
SCO OpenServer	SCO:_SV

Ein * in der zweiten Spalte bedeutet, dass 0-n weitere Zeichen nach den angegebenen folgen können.

12.2.3 Probleme mit Befehlen

Obwohl die meisten in diesem Buch vorgestellten Befehle auf allen Plattformen verfügbar sind, von den erwähnten Ausnahmen abgesehen, kann es selbst bei den gängigsten Befehlen zu leichten Unterschieden kommen. In diesem Abschnitt wollen wir Ihnen einige solcher Fälle kurz vorstellen, damit Sie sich ein Bild machen können, warum Ihr Shellskript auf einer anderen Plattform möglicherweise Probleme hat:

- `test` hat unter BSD Unix 4.3 keine `-x`-Option
- Nutzen Sie bei mehreren Bedingungen in einer `test`-Anweisung `||` und `&&` anstelle von `-a` und `-o`. Unter System-V- und BSD-Systemen sind die letzteren Optionen mit unterschiedlichen Prioritäten in der Reihenfolge der Auswertung gesegnet, was zu unterschiedlichen Ergebnissen führen kann.
- In der Bash vermeiden Sie folgendes Konstrukt: `var=${var:-value}` und nutzen Sie `${var=value}` als Ersatz. Einige ältere Versionen der Bash unter BSD-Systemen kommen mit der ersten Syntax nicht klar.
- In der Kornshell vermeiden Sie es besser, Variablen mit neuen Werten als letzten Befehl in einer Pipe (oder Befehlsliste) zu belegen. Die `pdksh` führt den letzten Befehl einer Pipe in einer Subshell aus, weshalb die Änderung verloren geht:

```
#!/bin/ksh
gvdir=0
cd /tmp && gvdir=1
echo "Dir = $gvdir"
```

  Dies klappt in der Kornshell (Versionen von 1988 und 1993), aber nicht in der `pdksh`, diese gibt immer `0` aus.

- Vermeiden Sie in der Kornshell doppelte =-Zeichen in `[[ $var == string ]]` und nutzen Sie stattdessen `[[ $var = string ]]`. Einige Versionen der Kornshell unter AIX 3 und AIX 4 kommen mit der ersteren Version nicht klar.
- Einige Versionen der Kornshell unter Ultrix sind nicht posix-konform. Sie erkennt z.B. die gültige Syntax `[[ expression1 || expression2 ]].` nicht, sondern bevorzugt `[[ expression1 ]] || .[[ expression2 ]]`.

Dies soll als Beispiel für einige unerwartete Tücken reichen, seien Sie an solchen Stellen wachsam.

12.3 Aufgaben

1. Welches Problem wird bei `PO_FindDir` komplett ignoriert? Erstellen Sie dazu einmal ein Verzeichnis namens `Neues Verzeichnis` in Ihrem Homeverzeichnis, und rufen Sie die Funktion `PO_FindDir` für dieses Verzeichnis auf. (Vergessen Sie nicht, vorab `updatedb` aufzurufen, da sonst die Informationen über Ihr System für `locate` noch nicht sichtbar sind.)
2. Kodieren Sie die Funktion `PO_FindDir` neu, sodass das Problem aus Aufgabe 1 nicht mehr auftaucht.
3. Schreiben Sie `chk.ksh` so, dass alle Dateien im aktuellen Verzeichnis geprüft werden, ob in der ersten Zeile ein `/bash` enthalten ist und sie ausführbar sind. Somit werden auch Zeilen wie `#! /usr/gnu/bin/bash` als korrekt erkannt.

12.4 Lösung

1. Ein Verzeichnis mit einem Leerzeichen im Namen bringt `PO_FindDir` ins Stolpern.
2. Im nachfolgenden Skript sollte das Problem mit den Leerzeichen im Namen überwunden sein:


```
# findneu.sh
# Ein Versuch, eine Funktionalität von
# find durch locate zu ersetzen :
# find / -name "$1" -print
function PO_FindDir
  {
    local lcTmpFile="/tmp/loc$$"
    if [ $# -eq 1 ] ; then
       local lvErg=""
       typeset -i lvAnz=`locate $1 |tee $lcTmpFile|wc -l`
```

```
      typeset -i i=1
      while [ $i -le $lvAnz ] ; do
          gvErg[$((i-1))]=`sed -n -e "${i}p" $lcTmpFile`
          i=i+1
      done
      rm -f lcTmpFile
      return $lvAnz
   fi
   return 0
}

typeset -i ind=0
PO_FindDir "$1"
gvAnz=$?
echo "Anzahl=$gvAnz"
[ $gvAnz -eq 0 ] && exit 0
while [ $ind -lt $gvAnz ] ; do
  echo "$ind. <${gvErg[$ind]}>"
  ind=ind+1
done
exit 0
```

3. Machen Sie es folgendermaßen:


```
#!/usr/bin/ksh
# chk.ksh Version 2
#
for i in * ; do
   [[ -f $i && -x $i ]] && {
    if head -1 $i | grep -q /bash ; then echo $i ; fi
  }
done
exit 0
```

`grep -q` macht keine Ausgaben, sondern gibt nur `0` (was gefunden) oder `1` (nichts gefunden) zurück.

Kapitel 13

Debugging/Fehlersuche

»Alles was schief gehen kann, geht schief« –
Murphys Gesetz

Dies ist das letzte Kapitel dieses Buches, und irgendwie finden wir Thema und Kapitelnummer passen sehr gut zusammen. Wie auch immer, Fehler treten trotz sorgfältigster Planung immer auf. Dies ist ärgerlich, und Ihr Ziel sollte es sein, so wenig Fehler in die Skripten einzubauen wie möglich.

Natürlich bringt allein der Vorsatz, keine Fehler zu machen, kein fehlerfreies Programm hervor. Aus diesem Grunde brauchen Sie Methoden und Hilfsprogramme, die Fehler reduzieren helfen. Fehler treten in mannigfaltiger Form auf, lassen sich aber auf vier Kategorien zusammenfassen:

- Logische Fehler

  Dies sind Fehler, wo der Algorithmus nicht korrekt ist. Ohne eine Korrektur der Logik wird dieses Skript nie richtig zum Laufen kommen.

- Syntaktische Fehler

  Ein Tippfehler reicht schon, und das Format der Befehle stimmt nicht mehr mit den Anforderungen der Shell überein, sodass sie mit einem Fehler abbricht. Einfach zu beheben, aber nicht unbedingt leicht zu finden.

- Programmfehler

  Dies sind Fehler, die unter bestimmten Umständen auftreten. Dies können syntaktische oder auch logische Fehler sein. Der Unterschied zu Punkt 1 liegt darin, dass der ausgedachte Algorithmus zwar korrekt ist, aber die ko-

dierten Anweisungen nicht dieser Logik entsprechen. Gute Beispiele dafür sind Schleifen, die einmal zu wenig oder zu oft durchlaufen werden.

- Fehler in den eingesetzten Programmen

  Es kann durchaus sein, dass Sie auf einen Fehler gestoßen sind, der nicht Ihnen, sondern dem Programmierer anzulasten ist, der die Shell oder einen der von Ihnen eingesetzten Befehle programmiert hat. In diesem Fall können Sie nur den Autor des Programmes kontaktieren und/oder wenn möglich den Fehler durch den Einsatz anderer Befehle umgehen.

  Hierzu zähle ich auch Fehler, die aufgrund von Limitierungen der eingesetzten Programme entstehen. So werden Sie mit einem `for dat in * ; do` bei Ihren Tests keine Probleme bekommen, aber irgendwann läuft Ihr Skript in einem Verzeichnis, das mehr Dateien enthält, als die Shell verkraften kann (oder auf einem System, in dem die Shell weniger Parameter verkraften kann als auf Ihrem Testsystem).

Ich will mich gar nicht als Expertin in Sachen Fehler aufspielen, da diese in meinen Programmen nie auftreten. Aber ich erinnere mich noch an Zeiten, wo ich weniger Erfahrung hatte und selten einmal Fehler auftraten, die ich geschickt und schnell fand ... (Christa: »Bettina, so geht das nicht. Unser Buch sollte glaubhaft sein!«; Bettina: »Aber Christa, ich wollte doch ...«; Christa: »BETTINA!!«) Okay, die ich recht schnell fand (Christa: »Bettina, es staubt!!«). Seufz, okay. Wie gesagt, Fehler treten immer auf, selbst die Besten werden nicht davon verschont. Dieses Kapitel soll Ihnen das Werkzeug an die Hand geben, zumindestens Ihre Fehler zu finden und auszumerzen.

Und scheuen Sie sich nicht, Ihr Skript abzubrechen, wenn Sie in eine Situation geraten, die sich nicht mehr lösen lässt. Besser ein Hinweis auf das Problem und ein `exit` als ein *Amok laufendes* Programm.

13.1 Planung

Bei kleinen Skripten ist eine langwierige Planung sicherlich etwas, was mit einer Kanone und mehreren Spatzen zu tun hat. Aber wenn Sie ein Skript schreiben, welches umfangreicher ist, dann ist vor dem Start etwas Planung angesagt. Überlegen Sie sich genau, was Ihr Skript machen soll.

Erstellen Sie sich zunächst eine Übersicht, ähnlich wie wir das bei den etwas umfangreicheren Skripten gemacht haben. Diese Übersicht sollte alle logischen Schritte, die das Skript ausführen sollte, von Punkt eins bis zum letzten Punkt aufführen. Schleifen und Abfragen springen genau auf einen der aufgeführten Punkte und nicht mitten in die Beschreibung eines Punktes. Ist dies notwendig, so ist der Punkt aufzuteilen.

Beispiel:

1. Initialisiere Variablen für Summe, aktuelle Zeile und Konstante für Temporärdatei. Dann ermittle Verzeichnisdaten (`ls -l >$tmpfile`) und Anzahl Einträge (`gvAnz`).
2. Schleife: Solange aktuelle Zeilennummer kleiner als `gvAnz`, sonst 7.
3. Ermittle Zeile aus Temporärdatei.
4. Ermittle Dateiname und Größe der Datei.
5. Addiere Größe auf die Summe.
6. Gehe nach 2.
7. Gib Ergebnis aus.
8. Ressourcen wieder freigeben.

Außerdem wird Ihr Skript sicherlich einige Bereiche haben, bei denen Sie sich nicht völlig sicher sind, ob es so funktioniert, wie Sie sich das vorstellen. Schieben Sie solche Teile nicht bis zum letzten Moment hinaus. Wäre doch ärgerlich, wenn Ihr Skript fertig ist, nur der letzte problematische Abschnitt kommt einfach nicht zum Laufen. Besser ein kleines Skript schreiben, mit dem Sie diesen Teil ganz am Anfang probieren. Und geben Sie nicht sofort auf, wenn es einfach nicht klappen will. Sprechen Sie doch ihren lokalen Skriptguru an (oder nutzen Sie eine der Ressourcen aus dem Anhang D), vielleicht kann er Ihnen ja einige Tipps geben. Sind alle Klippen umschifft, dann ran an das große Skript ;)

13.2 Variablen und Konstanten benennen

Wenn Sie Funktionen, Variablen und Konstanten benennen, so sollten Sie sich immer an einem Schema orientieren, nach dem Sie die Namen vergeben. Welche Methode Sie verwenden, bleibt Ihnen überlassen, aber es sollte sich aus den Namen erkennen lassen, was jetzt zugewiesen oder aufgerufen wird. An dieser Stelle möchten wir Ihnen eine Vorgehensweise vorschlagen, welche sich in leicht abgewandelter Form schon in großen Pascal- und C-Projekten bewährt hat.

Der Name einer Funktion sollte sich aus einem zweistelligen Kürzel, einem Unterstrich und dem eigentlichen Namen zusammensetzen. Dabei sollte das Kürzel den Skriptnamen widerspiegeln. So beginnen alle unsere Funktionen im CW-Commander mit `CW_`. Der dem Präfix folgende Name sollte nicht weniger als vier Zeichen umfassen und den Zweck der Funktion widerspiegeln.

Setzt sich der Name aus mehreren Worten zusammen, so sollte jedes Wort mit einem Großbuchstaben beginnen (`CW_PrintDir`, `CW_SetBold`). So können Sie ganz schnell erkennen, ob ein aufgerufener Befehl ein Skript (u.a. deshalb auch immer die Endung `.sh`), ein Shellbefehl oder eine eigene Funktion ist.

Bei Variablen und Konstanten gilt Ähnliches. Das sollte aus dem Präfix allerdings hervorgehen, ob sie lokal oder global ist. Wenn Sie einen Positionsparameter einer Variablen zuweisen, die nur einen lesbaren Namen anstatt `$1`, `$2` usw. vergeben soll, so sollte dies ebenfalls aus dem Namen hervorgehen. Einen Vorschlag finden Sie in Tabelle 13.1.

Tabelle 13.1: Beispiele für Variablen-, Konstanten- und Parameterbezeichnungen

Kürzel	Bedeutung	Beispiel
`lv`	lokale Variable	`lvAnzahl`
`lc`	lokale Konstante	`lcDMzuEuro`
`gv`	globale Variable	`gvSumme`
`gc`	globale Konstante	`gcTmpfile`
`pg`	Positionsparameter, der dem Skript übergeben wurde	`pgDir`
`pl`	Positionsparameter einer Funktion	`plFak`

Sie könnten sich noch überlegen, ob Sie den Typ (Array, Integer, Zeichenkette) auch noch in das Präfix kodieren oder noch andere Informationen, die Ihnen wichtig sind, nur: Halten Sie sich daran. Die Namensgebung sollte innerhalb eines Skripts einem klar zu erkennenden Muster folgen.

Regeln gibt es wie Sand am Meer, und welche Sie nutzen, ist auch Geschmackssache. In der Welt der Skriptprogrammierung finden sich auch häufig folgende Regeln:

- Konstanten in Großbuchstaben: `MAXLINE`
- Trennung durch `_` und nicht durch Groß-/Kleinschreibung: `TAR_FILE` oder `tar_file` statt `TarFile`

13.3 Kodieren

Logik und Benennung stehen, jetzt kommt das eigentlich Interessante: die Programmierung. Zu diesem Thema ist in diesem Buch schon viel gesagt worden, sodass wir wohl kaum hier wieder Schneisen in die deutschen Wälder schlagen müssen `:)`. Wir listen einfach mal die Version zum Aufaddieren der Dateigrößen in einem Verzeichnis entsprechend Tabelle 13.1 auf.


```
# Skript 55: Basiert auf einer Version von Skript 19
# M&T Verlag presents:
# A C.Wieskotten production
# A M.Rathmann Script:
# Dokumentation einfach gemacht.
#
# Zeigt eine mögliche Dokumentation von Skripten.
# mittels Benennungsschema und Kommentierung
# 1. Initialisierung
#
typeset -r gcTmpfile=/tmp/count$$
typeset -i gvAnz=`find . -name "$1" -type f -print | tee $gcTmpfile | wc -l
| cut -c-7`
typeset -i gvNr=0
typeset -i gvSumme=0
# 2. Schleife über alle Einträge in $gcTmpfile
while [ $gvNr -lt $gvAnz ] ; do
  gvNr=gvNr+1
  # 3. Zeile ermitteln und Dateinamen setzen
  gvDatei=`sed -n -e "${gvNr}p" $gcTmpfile`
  echo $gvDatei
  # 4. Dateigröße setzen
  typeset -i gvErg=`wc -c <$gvDatei`
  # 5. Addieren
  gvSumme=gvSumme+gvErg
done
# 7. Ausgaben
if [ $gvAnz -eq 0 ] ; then
   echo "Keine Dateien gefunden"
else
   echo
   echo "Insgesamt $gvAnz Dateien belegen $gvSumme Bytes"
fi
rm $gcTmpfile
exit 0
```

Wie Sie erkennen können, ist dies eine leicht angepasste Version von Skript 19 aus Kapitel 3. Zugegeben, dies sind nur kosmetische Korrekturen. Aber ich denke, Sie werden sich nach kurzer Eingewöhnungsphase blendend darin zurechtfinden. Vernünftige Namensgebung ist auch eine Form der Dokumentation (es sollte nur nicht die einzige sein).

Kommen wir zu den Kommentaren. Diese sind eine Gratwanderung zwischen zu viel und zu wenig. Es ist wenig sinnvoll, so viele Kommentare einzufügen, dass Sie nach den Skriptbefehlen suchen müssen. Genauso schlecht sind aber Skripten, die keine Kommentare enthalten. Am Anfang des Skripts sollten Sie Informationen aufführen, die angeben, was das Skript macht (»Dokumentation einfach gemacht«), von wem das Skript stammt (Christa Wieskotten und Bettina Rathmann) und welche Version (Skript 55) es ist.

Die wichtigsten Punkte des Skripts versehe ich mit Nummern und kurzen Kommentaren. Die Nummern decken sich dabei geschickterweise mit den Nummern, die für die einzelnen Schritte in der Planung vergeben wurden.

Bei Funktionen sollten Sie zumindestens im Kommentar angeben, welche Parameter erwartet werden, was die Funktion macht und ob sie Ergebnisse zurückgibt. Falls ja, sollten Sie auch noch angeben, in welcher Art Ergebnisse zurückgegeben werden.

Wichtig auch, dass Sie zumindestens eine rudimentäre Prüfung der Parameter vornehmen. Wenn Ihr Skript zwei Parameter erwartet, so prüfen Sie ab, dass auch zwei Parameter übergeben wurden. Setzen Sie Vorgaben, wenn die Parameter nicht übergeben wurden, oder brechen Sie mit einer erklärenden Fehlermeldung ab. Für Funktionen gilt das Gleiche für deren Parameter.

Testen Sie z.B. bei `case` auch immer auf den Fall `*)`, selbst wenn Sie absolut sicher sind, dass dieser Fall nicht auftritt. Spätestens, wenn Sie einige Änderungen vornehmen, wird der Zeitpunkt kommen, an dem dieser Fall eintritt. Selbst wenn Sie nur eine Fehlermeldung ausgeben und abbrechen, ist das noch besser, als wenn Ihr Skript weiterläuft, so als wäre alles in Ordnung.

Womit wir zum letzten Punkt kämen. Wenn Sie ratlos sind, wie das Skript auf ein Problem (falsche Daten, keine Rechte usw.) reagieren soll, dann brechen Sie es ab. Geben Sie vorher noch eine Fehlermeldung aus, und geben Sie (falls möglich) ein paar Tipps, was der Benutzer tun kann, um das Problem zu lösen.

13.3.1 Ordnung ins Skript

Jeder hat einen anderen Ordnungssinn, aber es lohnt sich auf jeden Fall, die Befehle nicht einfach in eine Zeile zu packen, sondern einzurücken. Dadurch können Sie Schleifen bzw. `if`-Abfragen viel leichter erkennen. Werden die Schleifen länger als eine Bildschirmseite, so kann es sinnvoll sein, am Ende der Schleife einen Kommentar zu setzen, der genau angibt, zu welcher Schleife das `done` gehört bzw. welche `if`-Abfrage durch das `fi` beendet wird.

Zum Beweis, welche Version ist leichter verständlich? Diese hier:

```
PS1=`if test "$UID" = 0 ; then \
       echo "\h:\`pwd -P\ # " ; \
     else \
       echo "\u@\h:\`pwd -P\ > " ;  \
    fi `     # if test $UID
```

Oder doch eher diese:

```
PS1=`if test "$UID" = 0 ; then \ echo "\h:\`pwd -P\ # " ; \ else \ echo
"\u@\h:\`pwd -P\ > " ; \ fi `
```

Es gibt auch noch einige weitere Regeln, die es erleichtern, das Skript zu verstehen:

- Jedes Skript und jede Funktion im Skript hat nur einen Einsprungspunkt und einen Endpunkt. Ausnahmen sind Abbrüche mit `exit` wegen Fehlern.
- Die Länge einer Funktion oder der Steuerung sollte nicht länger als ca. eine Bildschirmseite sein. Dies erleichtert die Übersicht ganz erheblich. Schleifen und `if`-Abfragen über mehr als eine Bildschirmseite verwirren mehr, als dass sie helfen. Machen Sie aus dem Schleifeninhalt (bzw. Abfrageinhalt) eine Funktion und rufen Sie diese auf.
- Wenn Sie viele Schleifen bzw. Abfragen schachteln, so hilft ein Kommentar hinter dem `fi` bzw. `done`, der besagt, welche Schleife bzw. Abfrage hier abgeschlossen wird.

```
if [ $gvZahl -ne 0 ] ; then
   if [ if $gvZahl -lt 10 ] ; then
      while [ $gvZahl -gt 0 ] ; do
        ...
        cat $lvDatei | while read lvZeile ; do
                        ...
                       done   #    of read lvZeile
      done    # of gvZahl -gt 0
   fi         # of gvZahl -lt 10
fi
```

  Diese Kommentierung erfordert Disziplin, wenn Sie häufiger die Schleifen/Abfragen ändern (sowohl auf Schachtelung als auch auf Bedingungen bezogen), aber je mehr Abfragen/Schleifen Sie schachteln, desto hilfreicher können sie sein.

- Rücken Sie ein! Dieses Buch hat diese Technik von Anfang an demonstriert. Das Einrücken erlaubt eine einfache Zuordnung, wo Schleifen und Abfragen beginnen und enden. Da die Shell ein Einrücken nicht vorschreibt, ist es Ihnen überlassen, wie Sie einrücken. Wichtig ist, dass Sie eine bessere Übersicht bekommen, der Stil ist egal.

Hier ist z.B. der (Korn)Shell-Mode des Emacs ziemlich hilfreich, auch wenn er durch zu wilde Konstrukte schon mal verwirrt.

13.4 Syntaxfehler entfernen

Erinnern Sie sich noch an die letzte Version von Skript 12 aus Kapitel 4.7.2? Nun, das sah wie folgt aus:


```
# Skript 12: (Vorerst) Letzte Version
#
tmpfile="/tmp/erg"
ls $1  > $tmpfile
ls $2 >> $tmpfile
echo "Die Verzeichnisse $1 und $2 enthalten `wc -l <$tmpfile` Dateien"
# Jetzt mit Here-Dokument
anz=`wc -l <<ENDE
\`find $1 -type d -print\`
\'find $2 -type d -print\`
ENDE`
echo "Insgesamt existieren genau" $anz "Unterverzeichnisse"
exit 0
```

Genauer gesagt, so sah die Version aus, bevor es durch die Hände des zweiten Autorenteils ging. Leider ist darin ein kleiner Syntaxfehler verborgen, der uns beim Testen zur Weißglut trieb, kehrte das Skript doch immer mit der falschen Anzahl an Dateien und Unterverzeichnissen zurück. Die Befehle waren scheinbar in Ordnung, aber das Ergebnis war es nicht. Erst nach langem Hin und Her fiel uns auf, dass in der Zeile

```
"\'find $2 -type d -print\`"
```

ein effektiver Tippfehler enthalten war. Anstelle eines Ausführungszeichens ` steht da ein Apostroph ', sodass die Shell den zweiten `find`-Befehl gar nicht ausführt, sondern als normale Eingabezeile des Heredokuments betrachtet. Die Meldungen der Shell sind in der Regel recht nützlich, dass hier keine Meldung kam, liegt in der Natur der Sache.

Wenn Sie Skripten schreiben und Schleifen/Abfragen einbauen, so fügen Sie erst das passende `esac`, `fi` oder `done` ein und danach erst die Anweisungen, die innerhalb der Schleife/Abfrage ausgeführt werden sollen:

```
while true ; do
...
done  # of while true
```

Und dann erst die Befehle:

```
while true ; do
   gvAnz=`ls | wc -l`
   echo "Es existieren $gvAnz Dateien"
   sleep 10
done  # of while true
```

So stellen Sie sicher (mit Einrückung und Kommentar), dass jede Abfrage/Schleife auch abgeschlossen wird. Fehlt z.B. ein `done`, so ereilt Sie am Ende des Skripts folgender Schicksalsschlag:

```
syntax error: unexpected end of file
```

Falls dies noch nicht reicht, so gibt es noch zwei Optionen, die Sie mittels `set` im Skript setzen können. Mit Hilfe von `set -v` gibt die Bash die Zeilen so aus, wie sie eingelesen wurden. Keinerlei Ersetzungen finden vor dieser Ausgabe statt. Möchten Sie das Skript einmal auf Syntax prüfen, ohne dass es wirklich ausgeführt wird, so setzen Sie bitte die Option `-n`. Durch `set -n` führt die Bash die erkannten Befehle nicht aus. Da die Shell aber Fehler wie fehlende Klammern o.ä. ausgibt, können Sie so schnell auf Tippfehler testen.

13.5 Logische Fehler

Ihre Planung ist korrekt, die Namen sprechen Bände, und es existieren auch keinerlei Tippfehler mehr, aber das Skript verhält sich immer noch nicht korrekt? Dann haben Sie einen logischen Fehler eingebaut. Damit soll nicht behauptet werden, dass Ihre Logik einer generellen Überarbeitung bedarf, sondern dass Sie eine Unterlassungssünde in einer Anweisung begangen haben.

Beste Beispiele für diese Art von kleinen Fehlern sind Schleifen, die einmal zu wenig (oder zu oft) durchlaufen werden, oder Anweisungen, deren Ergebnisse nicht vollständig beachtet werden und somit zu Folgefehlern führen:


```
...
typeset -i gvAnzZeile=0
gvAnzZeile=gvAnzZeile + `wc -l Datei`
...
```

Haben Sie den Fehler erkannt? Er hat uns bereits in Kapitel 2 unter anderen Vorzeichen heimgesucht. `wc` gibt auch den Dateinamen aus, deshalb versucht obige Anweisung Folgendes auszuführen:

```
...
typeset -i gvAnzZeile=0
gvAnzZeile=gvAnzZeile +    1234 Datei
...
```

Es gibt natürlich Fehler, die auch diesen Methoden, sie auszumerzen, widerstehen werden. In diesem Falle müssen Sie härtere Geschütze auffahren.

13.5.1 Tracen

Der nächste Schritt bei der Fehlerbekämpfung wäre die Überprüfung, welche Befehle ausgeführt werden, und wie das Skript abläuft. Dazu wäre es sehr nützlich, die jeweils aktuelle Zeile auszugeben mit einer eventuellen Übersetzung der Parameter.

Auch hierzu bietet Ihnen die Bash eine Möglichkeit an: das Tracen (Nachverfolgen) von Skripten. Setzen Sie dazu in Ihrem Skript die Option `set -x`. Danach gibt die Bash aus, welche Zeile ausgeführt wird. Vor der Zeile wird noch ein Zeichen wiederholt ausgegeben, welches anzeigt, wie tief die Verschachtelung ist, in der die Zeile ausgeführt wird. Für jede Verschachtelungsebene wird dabei `PS4` einmal ausgegeben.

Die Variable `PROMPT_COMMAND` haben wir bereits in Kapitel 6 besprochen. Auf unserem Linuxsystem enthält sie die unten aufgeführte Abfrage. Die Zeilenumbrüche wurden in der Variablen durch ein »\« unterdrückt, aber zur besseren Übersichtlichkeit fächern wir die Ausgabe etwas auf. Dadurch würde die Shell es nicht besser und wir wesentlich eher verstehen:

```
buch@koala:/home/buch > echo $PROMPT_COMMAND
PS1=`if test "$UID" = 0 ; then
        echo "\h:\`pwd -P\` # " ;
      else
        echo "\u@\h:\`pwd -P\` > " ;
     fi `
```

Diese Abfrage setzt `PS1` abhängig davon, ob es sich um den Benutzer `root` oder einen normalen Benutzer handelt. Stellen wir jetzt einmal den Tracemodus an:

```
buch@koala:/home/buch > set -x
+++ test 503 = 0
++++ pwd -P
+++ echo '\u@\h:/home/buch > '
++ PS1=\u@\h:/home/buch >
buch@koala:/home/buch > set +x
buch@koala:/home/buch >
```

Mit `set +x` wird der Tracemodus wieder deaktiviert. Die Schachtelung bezieht sich erkennbar auf die Ausführungsebene und nicht auf die Rekursionsebene. Die erste Ebene ist dabei die Loginshell selbst. Diese weist `PS1` das Ergebnis der Ausführung zu. Um die Ausführung zwischen den `` ` ` `` auswerten zu können, wird eine Subshell gestartet, in der dieses Miniskript ausgeführt wird. Diese ruft wiederum `test` auf, deshalb die Ebene 3 (`+++ test 503 = 0`). Im positiven Fall wird `pwd` ausgeführt und damit Ebene 4 erreicht und auch beendet. Mit dem `echo` wird die dritte Ebene beendet, und mit der Zuweisung `PS1=...` wird die letzte Ebene abgeschlossen.

Was passiert nun in einem Skript? Nehmen wir dazu einfach mal Skript 38, setzen den Tracemodus und führen es aus:

```
buch@koala:/home/buch > set -x
+++ test 503 = 0
++++ pwd -P
+++ echo '\u@\h:/home/buch > '
++ PS1=\u@\h:/home/buch >
buch@koala:/home/buch > fak 2
+ fak 2
Fak(2)=2
+++ test 503 = 0
++++ pwd -P
+++ echo '\u@\h:/home/buch > '
++ PS1=\u@\h:/home/buch >
buch@koala:/home/buch >
```

Schon haben wir ein weiteres Problem, wird doch das Skript nicht getraced, obwohl die Option klar aktiviert wurde. Ein Bug? Nein, ein Feature. Die Traceoption ist nur in der aktuellen Shell und ihren Subshells aktiv. Sie wird nicht in die Umgebung der gestarteten Shell übernommen, die für die Steuerung des Skripts zuständig ist. Nun ist aber der Aufruf eines Skripts identisch mit dem Start einer neuen Shell, in der dann das Skript ausgeführt wird. Aus diesem Grunde wird das Skript nicht getraced.

Ok, fügen wir also ins Skript am Anfang des Hauptteils ein `set -x` ein und starten das Skript noch einmal:

```
buch@koala:/home/buch > fak 2
++ typeset -i zahl=2
++ typeset -i fak=1
++ '[' 2 -lt 1 ']'
++ FAK 2
++ local zahl=2
++ '[' 2 -lt 1 ']'
++ FAK 1
++ local zahl=1
++ '[' 1 -lt 1 ']'
++ FAK 0
++ local zahl=0
++ '[' 0 -lt 1 ']'
++ typeset -i fak=1
++ fak=1*fak
++ fak=2*fak
++ echo 'Fak(2)=2'
Fak(2)=2
++ exit 0
buch@koala:/home/buch >
```

Man kann auch eine Skriptänderung vermeiden, indem das Skript wie folgt aufgerufen wird:

```
buch@koala:/home/buch > bash -x ./fak 2''
```

Der Ablauf des Skripts wird jetzt deutlich. Zu erkennen ist die Tatsache, dass das gesamte Skript auf einer Ebene abläuft und keinerlei Subshells öffnet. Der `if`-Befehl wird auf den `test`-Befehl reduziert. Sehr gut sind die Werte der Variablen und Zuweisungen zu erkennen. Wie schon in den Aufgaben in Kapitel 7 besprochen, lässt sich hier deutlich erkennen, dass eine Rekursionsebene zu tief verzweigt wird, der Aufruf von `FAK 0` ist überflüssig.

Natürlich sind Sie nicht gezwungen, die Nachverfolgung global für das gesamte Skript zu setzen. Sie können gezielte Passagen in den Tracemodus setzen, falls Sie den Fehler auf einen bestimmten Bereich einkreisen können.

13.5.2 DEBUG- und ERR-Signale

Die Bash bietet neben den richtigen Signalen aus Kapitel 8 auch noch ein Signal, welches nur zum Debuggen von Programmen dient: `DEBUG`.

Wie bereits aus Kapitel 8 bekannt, können Sie beliebige Befehle ausführen, wenn ein Signal an Ihr Skript geschickt wird. Mit der gleichen Syntax können Sie aber auch das Pseudosignal `DEBUG` setzen, welches nur in der Bash funktioniert. Falls aktiviert, führt es nach jedem einfachen Shellbefehl den im `trap` angegebenen Befehl aus. Diesen können Sie dazu nutzen, um Informationen zur Laufzeit in eine Datei oder auf den Bildschirm auszugeben.

Die Kornshell (`ksh`) bietet im Gegensatz dazu das Pseudosignal `ERR` an. Dieser wird genauso wie ein beliebiges Signal angefangen (`trap <befehl> ERR`) und wird dann aufgerufen, wenn ein Fehler im Skript aufgetreten ist. Dabei bezieht sich Fehler nur auf den Rückgabewert eines aufgerufenen Befehls. Das gilt allerdings nur für Befehle, deren Exitstatus nicht mittels `if`, `until` oder `while` abgefragt wird. Dadurch werden Schleifenbedingungen nicht vom Signal beeinträchtigt.

Ein Beispiel zum `DEBUG`-Signal finden Sie in Kapitel 13.8.3, hier ein kleines Skript für die Kornshell:


```
#!/usr/bin/ksh
#
# Skript: err.ksh
# Demonstriert, wie sich das ERR-Signal
# in der Kornshell verhält
```

```
function ER_Handler
{
  # Die pdksh kennt momentan keine der beiden Variablen
  echo "Fehler $ERRNO in Zeile $LINENO"
  exit 1
}
# 1. Die Fehlerbehandlung installieren
echo $LINENO
trap 'ER_Handler' ERR
# 2. Einen Fehler generieren, den ERR nicht abfängt
#    weil der Fehler von einem Befehl abgefragt wird
cd /FalschesVerzeichnis || echo "Dieses Verzeichnis existiert nicht!"
# 3. Dieser Fehler ruft ER_Handler auf den Plan
cd /FalschesVerzeichnis
# 4. Hier kommen wir schon nicht mehr hin
echo "Der Inhalt lautet"
ls -l
exit 0
```

13.6 Sonstige Methoden

Geben Sie sich nicht der Hoffnung hin, dass alle aufgeführten Methoden sämtliche Fehler ausmerzen können. Sie sollten Ihre Kreativität im Bereich Fehler nicht unterschätzen :)

Aus diesem Grunde hier noch einige Tipps, wie Sie Fehler ermitteln können.

13.6.1 Abbruch forcieren

```
set -e
set +e
```

Die Bash erlaubt es, ein Skript sofort abzubrechen, wenn ein Fehler aufgetreten ist. Ein Fehler in diesem Zusammenhang ist ein Rückgabewert eines Befehls, der ungleich 0 ist. Rückgabewerte, die in einem `test`-Konstrukt abgefragt oder durch ein »`!`« negiert werden, bleiben in diesem Zusammenhang unberücksichtigt. Das gilt ebenfalls für die ODER-Listen (`||`) bzw. UND-Listen (`&&`).

`set -e` aktiviert dieses Verhalten, und `set +e` deaktiviert es wieder.

Ein Abbruch ist dann sinnvoll, wenn Sie einen Fehler anhand von Ausgaben Ihres Skripts nachvollziehen können. Gibt Ihr Skript jede Menge Daten aus, die an einer Stelle nicht mehr stimmen, und Sie erkennen in der Menge der Ausgaben gerade noch eine Fehlermeldung, so ist es sinnvoll, das Skript an

der Fehlerstelle abzubrechen und die Ausgaben bis zu diesem Punkt zu analysieren.

13.6.2 EXIT-Signal nutzen


```
trap <befehl> EXIT
```

Wird dieses Pseudosignal, welches sowohl von der Kornshell als auch von der Bash angeboten wird, abgefangen, so wird der `<Befehl>` ausgeführt, bevor das Skript beendet wird.

Dabei kann das Ende des Skripts durch ein `exit` oder auch durch einen Fehler hervorgerufen werden. Wie allerdings in Kapitel 13.6.1 bereits erläutert, bricht die Shell nur dann im Fehlerfall ab, wenn die Option `set -e` gesetzt wurde.

Wird `EXIT` in einer Funktion gesetzt, so führt die Kornshell (nicht die `pdksh` und auch nicht die Bash 2.x) den `<befehl>` aus, wenn die Funktion beendet wird.

Für Hintergrundskripten, die z.B. durch `cron` aufgerufen werden, ist diese Funktion sehr hilfreich. Sie kann dazu genutzt werden, Informationen über Fehler oder korrektes Ablaufen von Skripten an den Benutzer zu mailen.

Die Idee dahinter ist, den EXIT-Handler zu installieren und in einer Variablen den letzten erfolgreichen (Logik-)Schritt des Skripts abzulegen. Tritt ein Fehler auf, so wird der Inhalt dieser Variablen an den Benutzer geschickt.


```
# Skript exit.sh
#
# Sollte unter ksh und bash laufen
# Demonstriert die Anwendung des
# "EXIT"-Signals der Bash / Ksh
# plus des "set -e" Befehls
# 1. Der Exit-Handler
EX_Error()
{
   mailx -s "Failure: $gvLastStep" buch@koala $gcLogFile
   rm -f $gcLogFile
   exit 1
}
gvLastStep="Exit-Handler aktiviert"
gcLogFile="/tmp/demo$$.tmp"
trap EX_Error EXIT
# 2. alle Ausgaben in die Logdatei
#    Bei einem Fehler sofort abbrechen
set -e
```

```
exec 2>&1
exec > $gcLogFile
# 3. Hier startet die eigentliche Arbeit
LAST='Schritt 1: Initialisieren'
gvDir=`pwd`
...
# 4. Fertig mit der Arbeit, jetzt das positive Ergebnis
#    mailen und aufräumen.
mailx -s "Jubel, Trubel, Heiterkeit" buch@koala <$gcLogFile
rm -f $gcLogFile
trap '' EXIT
exit 0
```

Das `trap '' EXIT` unter Punkt 4 ist wichtig, weil sonst mit dem Ende des Skripts noch einmal die Funktion `EX_Error` aufgerufen wird.

Wenn das Skript in der `sh` laufen soll, so muss die Angabe von `EXIT` beim `trap` durch eine `0` ersetzt werden, z.B.: `trap '' 0`

13.6.3 Debugausgaben einbauen

Wenn Ihr Skript sich fehlerhaft verhält, die Logik aber in Ordnung zu sein scheint, so ist es sinnvoll, an den fragwürdigen Stellen eine Ausgabe mit `echo` einzubauen. Dabei sollten die Ausgaben klar machen, an welcher Stelle das Skript gerade ist.

Ist das Verhalten Ihres Skripts von einer Variablen abhängig, die bestimmt, ob und wenn ja, welche Aktionen im Skript durchgeführt werden, so sollten Sie diese auch ausgeben.

Diese Technik ist im Prinzip eine Variante des Tracebefehls. Der Tracebefehl kann aber manchmal zu viele Informationen ausgeben. Gerade wenn Sie nur eine Übersicht benötigen, welcher Bereich des Skripts gerade ausgeführt wird, so ist die Ausgabe per `echo` einem Trace vorzuziehen.

13.6.4 Zugriffe auf Variablen prüfen

```
set -u
set +u
```

Falls Sie sicherstellen wollen, nicht versehentlich auf falsche Variablen zuzugreifen, so können Sie die Shell mittels `set -u` veranlassen, einen Fehler auszugeben, wenn Sie auf nicht gesetzte Variablen zugreifen. Dies können Sie auch mit den bereits erwähnten Methoden kombinieren (Abbruch, Exit, Debug etc).

Nehmen wir das folgende Skriptstückchen:


```
# Skript 56: Fehler bei uninitialisierten Variablen erzwingen
#
# Dieses Skript dient nur zur Demonstration von set -u
set -u
lcInputfile="australia.dat"
gvAnz=`wc -l $lcinputfile`
set +u
lcInputfile="australia.dat"
gvAnz=`wc -l $lcinputfile`
exit 0
```

Wir nehmen einmal an, es existiert eine Datei namens `australia.dat`, sodass `wc` auf kein Problem stößt. In diesem Fall bekommen wir einen Fehler von der Bash für den ersten Versuch, und im zweiten Anlauf hängt das Skript, da es eine Eingabe von der Standardeingabe erwartet.

```
buch@koala:/home/buch > ./skript56.sh
./skript101.sh: lcinputfile: unbound variable
_
```

13.6.5 Die Shell und nicht existente Befehle

Es kann natürlich sein, dass ein Befehl, den Sie nutzen, nicht auf dem Zielsystem vorliegt. Wie Sie diese Probleme umgehen können, haben wir in Kapitel 12 anhand von `wc`- und `find`-Befehlen untersucht.

Möglich ist aber auch, dass die Shell, auf der Sie getestet haben, eine Bash war, und das Skript in der Kornshell läuft. Zwar sind beide Shells ziemlich kompatibel, aber die `ksh` kann z.B. auch Arrays mit Zeichenketten indizieren. Ein `array ["Perth"]="West Australien"` ist dort erlaubt, die Bash kann dies nicht. Gleiches gilt auch andersherum.

Ziel muss es sein, dass Ihr Skript möglichst kompatibel zur `ksh`, `bash` und `sh` sein sollte. Die wichtigsten Unterschiede zwischen diesen Shells können Sie in Anhang A nachschlagen.

Wollen Sie erzwingen, dass Ihr Skript in einer bestimmten Shell läuft, so geben Sie in der ersten Zeile der Datei Folgendes ein:

```
#!/bin/bash
```

für die Bash oder

```
#!/usr/bin/ksh
```

für die Kornshell. In diesem Fall stellt das Betriebssystem sicher, dass erst das Programm aufgerufen wird, welches mit dem absoluten (!) Pfad hinter dem `#!` angegeben wurde, und dann darin das Skript ausführt.

Diese Methode hat scheinbar den krassen Nachteil, dass ein `File not found`-Fehler ausgegeben wird, falls ein Programm mit diesem Pfad und Namen nicht existiert. Diese Meldung ist mehr als irritierend, da das Skript vorliegt, der Pfad auch in `PATH` enthalten ist und dennoch ein Fehler auftritt.

Umgekehrt wird aber ein Schuh daraus:

Braucht Ihr Skript eine Bash, und die lässt sich nicht finden, so ist nicht mehr sichergestellt, dass Ihr Skript noch korrekt läuft oder die richtige Funktionalität bietet. In diesem Zusammenhang sei nur auf `read -p` in der Kornshell und der Bash verwiesen. In diesem Fall ist es besser, das Skript startet erst gar nicht, bevor größerer Schaden angerichtet wird.

Eine andere Möglichkeit besteht darin, den Skriptnamen ein Suffix zu verpassen, aus dem hervorgeht, für welche Shell sie vorgesehen sind. So könnte `.sh` auf Bash- oder sh-Skripten hinweisen, während die Endung `.ksh` auf Kornshellskripte hindeutet.

Damit sollte klar sein, warum wir unsere Skripten immer auf `.sh` enden ließen.

13.7 Sonstige Tipps

Sollte der Fehler jetzt immer noch Bestand haben, noch ein paar schnelle Tipps, wie Sie ihm auf die Spur kommen können.

Überprüfen Sie zunächst einmal die Anzahl an Klammern bzw. Anführungszeichen. Diese treten naturgemäß fast immer paarweise auf. Für jede offene Klammer `(` gibt es auch eine schließende Klammer `)`. Gleiches gilt für Anführungszeichen und geschweifte bzw. eckige Klammern `{}`, `[]`. Ausnahme von dieser Regel ist `case`, dessen Bedingungen werden durch `)` abgeschlossen, zu denen es keine `(` gibt.

Haben Sie eine geschachtelte Anweisung mit Backticks `` ` ``, die nicht funktioniert, so ist es sinnvoll, die einzelnen Teilausdrücke außerhalb des fehlerhaften Ausdrucks zu testen. Funktionieren die Teilausdrücke, so sollte der komplette Ausdruck Stück für Stück wieder zusammengesetzt und erneut getestet werden. Zu einem Zeitpunkt wird der Fehler wieder auftreten, und dann müssen Sie nur vergleichen, welche Änderungen Sie zuletzt vorgenommen haben.

Denken Sie ans Quoting, wenn Sie Ausführungszeichen `` ` `` in mehreren Ebenen schachteln wollen/müssen.

Komplexe Algorithmen unterteilen Sie am besten in viele einfache Logikschritte. Aus Gründen der Übersichtlichkeit erstellen Sie am besten gleich daraus Funktionen und prüfen deren Verhalten genauestens. Aus den einzelnen getesteten Funktionen können Sie den kompletten Algorithmus zusammensetzen. Mit dieser Methode haben Sie gleich zwei Vorteile:

1. Der Algorithmus bzw. dessen Logik wird übersichtlicher.
2. Die Teilschritte sind weniger komplex als der komplette Algorithmus. Allein dadurch sind sie wesentlich weniger fehleranfällig. Dazu kommt aber noch, dass sie einfacher auszutesten sind, schließlich sind Parameter und Rückgabewerte klar definiert, und die Logik ist recht kurz.

Ein `PS1='\u@\h $? > '` kann sehr praktisch sein, denn der Exitstatus des letzten Befehls lässt sich direkt im Prompt erkennen, was Tipparbeit ersparen kann.

Im Vordergrund läuft Ihr Skript, aber nicht mehr im Hintergrund, wenn es z.B. über `at` oder `cron` gestartet wird? In diesem Falle sollten Sie überprüfen, welche Unterschiede zwischen Vordergrund und Hintergrund existieren. So ist die Shellumgebung mit Sicherheit anders eingestellt, wird doch die `/etc/profile`- oder die `.profile`-Datei im Hintergrund nicht abgearbeitet. Oder nutzen Sie noch einen Pager für eventuelle Ausgaben? Das wird im Hintergrund sicherlich zu Problemen führen.

Sind Sie sich nicht sicher, dass Ihr Skript in den richtigen Funktionen landet, bzw. ob die Funktionen in der richtigen Reihenfolge aufgerufen werden? Dann könnte Ihnen die Variable `FUNCNAME` nützlich sein.

Natürlich schließen sich die in diesem Kapitel vorgestellten Methoden zum Debuggen von Skripten nicht gegenseitig aus. Zögern Sie nicht, die verschiedenen Methoden nach Ihren Vorlieben miteinander zu kombinieren. Zum Austilgen von Fehlern darf Ihnen jedes (hier vorgestellte) Mittel recht sein.

Und noch ein Tipp, der wahrscheinlich etwas eher hätte auftauchen dürfen, aber Sie eventuell in Zweifel gestürzt hätte, was die Effektivität der vorgestellten Methoden betrifft:

Bevor Sie Ihr möglicherweise fast fertiges Skript total auf den Kopf stellen und das Skript immer weiter von den Zielvorgaben abweicht, sichern Sie ihr Skript an eine sichere Stelle. Es ist auch uns schon passiert, dass wir ein Skript »kaputt repariert« haben, in dem verzweifelten Versuch, einen kleinen Fehler »mal eben« zu beheben.

Der Fehler ist immer noch da? Sie ändern das Skript ab, aber selbst ein simples `echo` wird nicht ausgeführt? Dann sollten Sie auf jeden Fall einmal den Namen Ihres Skripts überprüfen. So ist die Idee, ein Skript `test` zu nennen, eher unklug, gibt es doch den Befehl `test` für die Bedingungen in der `if`-Abfrage und den Schleifen.

Welchen Befehl die Shell ausführt, ergibt sich daraus, welcher Befehl zuerst gefunden wird. Zum einen hat die Shell interne Befehl (z.B. `read`), und zum anderen sucht sie alle Verzeichnisse in `PATH` ab, um den angegebenen Befehl zu suchen. Der erste Befehl, der in dieser Suchreichenfolge (intern dann `PATH`) gefunden wird, kommt zur Ausführung. In der Regel ist das aktuelle Verzeichnis »`.`«, als letzter Eintrag in `PATH` vorhanden, sodass Ihr Skript erst als letztes gefunden werden kann.

Wie aber bereits erwähnt, ist vor allem als Benutzer `root` ein »`.`« oder »`::`« im Pfad ein Sicherheitsloch. Auch aus diesem Grunde rufen wir in diesem Buch unsere Skripten immer mit »`./`« auf.

13.8 Beispiel

An dieser Stelle wollen wir noch ein letztes kleines Skript erstellen, welches wir jetzt nach den oben aufgeführten Methoden durcharbeiten. Ziel soll es sein, eine Funktion zu schreiben, welche den Rückgabewert und die Zeilennummer des zuletzt ausgeführten Befehls ausgibt. Dieser Status soll nach jedem Befehl ausgegeben werden, wenn die Variable `gvDebug` nicht leer ist. Dadurch kann die Debugfunktionalität gezielt ein- und ausgeschaltet werden.

13.8.1 Planung

Da das Skript aber auch einige Befehle ausführen soll, damit die Auswirkungen von `DEBUG` sichtbar werden, packen wir die oben erwähnte Funktion in ein Skript, das ein Muster als Parameter erwartet und überprüft, ob die durch das Muster ermittelten Dateien ein Backup mit der Endung ~ haben (wird z.B. vom Editor `joe` angelegt). Falls ja, fragen wir nach, ob die Unterschiede zwischen alter und neuer Version ausgegeben werden sollen.

1. Die Parameter prüfen und Variablen initialisieren.
2. Für jeden Dateinamen, der dem Muster entspricht.
3. Prüfen, ob die Datei ein lesbares Backup hat. Wenn nicht, weiter bei 2.
4. Abfragen, ob Unterschiede ausgegeben werden sollen.
5. Ist Eingabe [j], so wird `diff` ausgeführt.
6. Gehe nach 2.

13.8.2 Namensvergabe

Wir nutzen die Variablen `gvDebug`, um das Debugging gezielt ein- bzw. auszuschalten. Die Abfrage wird in Variable `gvDiff` eingelesen, und das war alles, was wir an eigenen Variablen brauchen. Die Zeilennummer steht in `LINENO` (`pdksh` hat diese Variable noch nicht implementiert), und der Exitstatus wird bekanntlich mit `$?` abgefragt.

13.8.3 Kodierung

Das sollte Ihnen mittlerweile kaum noch Schwierigkeiten bereiten, daher hier einfach mein Vorschlag für die Lösung der gestellten Aufgabe:


```
#!/bin/bash
# Skript debug.sh
#
# Zeigt die Anwendung von DEBUG.
# Überprüft, ob Backups mit "~" am Ende existieren
# und vergleicht Backup mit Originalversion
#
function DB_Debug()
{
 if [ -n "$gvDebug" ] ; then
    echo "Zeile: $1 Exit : $2"
 fi
}
# 1. Den Debughandler einsetzen
#    und Parameter prüfen
trap 'DB_Debug $LINENO $?' DEBUG
[ $# -eq 0 ] && echo "Suchmuster übergeben!" >&2 && exit 1
gvDebug="ja"
# 2. Für jeden Dateinamen
for gvDatei in $1 ; do
   # 3. Backup lesbar?
   if [ -r "${gvDatei}~" ] ; then
     # 4. Nachfragen
     echo "Datei $gvDatei hat backup"
     echo -n " Unterschiede ausgeben (J/n):"
     read gvDiff
     # 5. Unterschiede ausgeben
     echo $gvDiff | grep -qi "J" && diff -DAlt $gvDatei ${gvDatei}~ |pg
  fi   # -r
# 6. Und zum nächsten Namen, falls der existiert
done    # gvDatei
exit 0
```

Ein Blick auf das Skript enthüllt nichts Weltbewegendes, allerdings einen weiteren uns unbekannten Befehl: `diff`.


```
diff -D<Marke> <Datei1> [<Datei2>]
diff <Datei1> [<Datei2>]
diff -q <Datei1> [<Datei2>]
```

`diff` vergleicht den Inhalt von `<Datei1>` mit dem Inhalt von `<Datei2>` und gibt die Unterschiede aus. Wird `<Datei2>` nicht angegeben, so vergleicht `diff <Datei1>` mit den Daten, die von der Standardeingabe kommen. Die Option `-q` gibt nur aus, ob die Dateien unterschiedlich sind, aber nicht, was die Unterschiede genau sind. Mit `-D<Marke>` gibt `diff` das Ergebnis in einer Datei aus und markiert Unterschiede mit Direktiven für den C-Präprozessor. Hier ein Beispiel:

```
Orginaldatei        Backup               Ergebnis
Rosella             Dingo                #ifndef alt
Lyre-Bird           Lyre-Bird            Rosella
Katherine Gorge     Katherine Gorge      #else /* alt */
Kuranda             Atherton Tableland   Dingo
                    Kuranda              #endif /* alt */
                                         Lyre-Bird
                                         Katherine Gorge
                                         #ifdef alt
                                         Atherton Tableland
                                         #endif /* alt */
                                         Kuranda
```

Dies steht für »Wenn `<Marke>` definiert ist, dann beachte nur den `if`-Anteil, ansonsten den `else`-Teil der Anweisung«. Dabei kann natürlich der `else`-Teil auch wegfallen. Je nach Gegebenheiten kann auch ein `#ifndef <Marke>` auftauchen, was für »Wenn `<Marke>` nicht definiert ist, dann beachte den `if`-Anteil, ansonsten den Teil nach dem `else` (falls vorhanden)« steht.

Was aber gibt unser Skript aus, wenn wir die beiden Dateien, die wir gerade per Hand verglichen haben, über unser Skript vergleichen? Wir nehmen an, die Originaldatei heißt `org.txt` und das Backup `org.txt~`.

```
buch@koala:/home/buch/mysh > ./debug.sh org.txt
Zeile: 17 Exit : 1
Zeile: 19 Exit : 0
Datei org hat backup
Zeile: 20 Exit : 0
 Unterschiede ausgeben (J/n):Zeile: 21 Exit : 0
j
Zeile: 22 Exit : 0
Zeile: 23 Exit : 0
Zeile: 23 Exit : 0
Zeile: 23 Exit : 0
#ifndef Alt
Rosella
#else /* Alt */
```

```
Dingo
#endif /* Alt */
Lyre-Bird
Katherine Gorge
#ifdef Alt
Atherton Tableland
#endif /* Alt */
Kuranda
Zeile: 23 Exit : 0
buch@koala:/home/buch/ >
```

Gleich die erste Zeile gibt einen Exitwert von 1 aus, weil der Vergleich `[ $# -eq 0 ]` falsch (also 1) ergibt.

13.9 Aufgaben

Was ist für dieses Kapitel besser zum Üben geeignet als Skripten, die den hier besprochenen Regeln nicht entsprechen und dazu auch noch nicht korrekt laufen. Versuchen Sie bitte, solche Skripten zum Laufen zu bringen.

1. Das letzte Übungsskript:

   Das folgende ist ein simples Skript: Es liest den ersten Parameter ein, ist dieser leer, so wird er durch das aktuelle Verzeichnis ersetzt. Ist eine Datei angegeben, so wird die Anzahl Zeilen gezählt und ausgegeben. In einem Verzeichnis zählen wir alle Dateien auf und geben für jede Datei die Zeilenanzahl aus. Am Ende fragen wir eine neue Datei ab oder beenden das Skript durch die Eingabe von [q].

   PS: Wir wissen, dass man einige Probleme dieses Skripts mit einfachsten Mitteln umgehen kann, dennoch haben wir bewusst diese Lösung vorgesehen, damit einige Probleme nochmals angesprochen werden können. (Nebenfrage: Welche Probleme meinen wir, und wie kann man sie umgehen?)

   Das Skript ist von der Logik her okay, allerdings sind die Befehle nicht alle korrekt kodiert. Bringen Sie das Skript zum Laufen!


```
#!/bin/bash
#
# Skript zur Aufgabe in Kapitel 13.
# Die längere Version mit zwei Fehlern
# logikerr.sh
gvDat=${1:=`pwd`}
```

```
while [ $gvDat != "q" ] ; do
  if [ -d "$gvDat" ] ; then
     ls $gvDat | wc -l $gvDat
   else
     wc -l ${gvDat}
  fi
  read -p "Neue Eingabe (q=Ende) :" gvDat
done
echo "Ende"
exit 0
```

2. Fehler werden im Englischen als Bug bezeichnet. Computer wurden zwar in Deutschland von Konrad Zuse zu Zeiten des Zweiten Weltkriegs erfunden, jedoch wurde auch in Amerika der erste Rechner noch vor dem Ende des Kriegs in Betrieb genommen. Während in Deutschland das Potenzial dieser neuen Technik nicht erkannt wurde (Hitler konnte mit solchen Maschinen weder Länder erobern noch unschuldige Menschen töten), wurden in Amerika elektronische Rechner gebaut und auch genutzt.

   Dabei bedeutet *elektronisch* aber nicht klein, sondern nur, dass neben Relais auch elektronische Bauteile wie Röhren und Steckplatinen verwendet wurden. Da Röhren aber recht groß waren, umfassten Rechner, die weniger leisteten als heutige Taschenrechner, Räume, in denen leicht mehrere Wohnungen hätten eingerichtet werden können.

   Da es zu diesem Zeitpunkt auch noch keine CPUs gab, wurde ein Programm »hart verdrahtet«. Dies ist wörtlich zu nehmen, die Komponenten des Rechners wurden mittels Kupferdraht so verbunden, dass der Computer rechnen konnte. Eine Neuprogrammierung erforderte also großen Aufwand.

   Es geht die Geschichte um, dass eines Tages ein Programm absolut nicht so lief, wie die Programmierung es eigentlich vorsah. Somit musste der Programmierer in die Tiefen des Rechners klettern (Kein Scherz! Die Dinger waren wirklich so groß!) und den Fehler suchen. Dabei stellte sich heraus, dass sich ein Käfer in die Tiefen des Rechners verirrt hatte und einen Kurzschluss in der Verdrahtung hervorgerufen hatte, weshalb der Rechner zu falschen Ergebnissen kam. Aus diesem Grunde werden Fehler Bugs genannt.

13.10 Lösungen

1. Die zwei Fehler sind nun behoben:

```
#!/bin/bash
#
# Lösung logikok.sh zum fehlerhaften
# Skript logikerr.sh
# Demonstriert UND-Listen
#
gvDat=${1:-`pwd`}    # erster Fehler
# 2. Anführungszeichen fehlten
while [ "$gvDat" != "q" ] ; do
  [ -d "$gvDat" ] && ls "$gvDat" | wc -l
  [ ! -d "$gvDat" ] && wc -l "${gvDat}"
  read -p "Neue Eingabe (q=Ende) :" gvDat
done
echo "Ende"
exit 0
```

Die `if`-Schleife wurde durch UND-Listen ersetzt, um das Skript noch mehr zu verkürzen. An der Funktionalität hat sich dadurch nichts geändert.

Anhang A

sh, ksh und bash

»Fantasie ist wichtiger als Wissen, denn Wissen ist begrenzt« – Albert Einstein

An dieser Stelle sollen einige wichtige Unterschiede zwischen den Shells dargestellt werden. Damit sollten Sie in der Lage sein, Ihre Skripten portabel zu gestalten. Und noch einige Hinweise zu diesem Anhang:

- Die Kornshell haben wir in drei Versionen berücksichtigt. Die freie Version nennt sich `pdksh`. Die neueste Version der Kornshell ist ksh93, die Version davor ksh88.
- Die Bash wird hier in der Version 2.05 berücksichtigt.
- Die Bourneshell `sh` hält sich weitestgehend an die Solaris-Version, allerdings ist es nicht einfach, *die* `sh` zu finden.

Tabelle A.1: Unterschiede im Befehlsumfang von Bourne-, Korn- und Bourne-Again-Shell

Beschreibung	sh	bash	ksh
`!` Invertiert Exitstatus	nein	ja	ja (10)
`$()` Befehlsersetzung	nein	ja	ja
Parameter mittels `${10}`	nein	ja	ja
`alias`	nein	ja	ja
`source` / »`.`«	nur »`.`«	ja	nur »`.`«
`declare` / `typeset`	nein	ja (5)	nur `typeset`
`type`	ja	ja	Alias `whence -v`
`test` hat Vergleich `<` `>` `<=` `>=`	nein (15)	ja	nur `<` `>`
`[[]]`	nein	nein (13)	ja
`getopts`	ja	ja	ja
`readonly`	ja (1)	ja	ja (1)
`set --`	ja	ja	ja
`trap`	ja (2)	ja (3)	ja (4)
Jobverwaltung	nein (6)	ja	ja
`disown`	nein (6)	ja	nein (11)
`stop`	ja	nein, alias	nein, alias
Shellfunktionen	ja (7)	ja	ja
Lokale Variablen	nein	ja	ja
Überschreiben per Umleitung (Clobber) »`>|`«	nein	ja	ja
`let`/Arithmetische Ausdrücke	nein	ja	ja
Kommandozeile editierbar	nein	ja	ja (16)
History/Verlauf für Kommandozeile	nein	ja	ja (16)
Umleitung/Kanalduplizierung	ja	ja	ja
Parametererersetzung	ja	ja	ja
Prozessersetzung/Process Substitution	nein	ja (8)	ja (9)
`echo` interner Shellbefehl	nein	ja	nein
`test` interner Shellbefehl	ja	ja	ja
Assoziative Arrays	nein	nein	ja (12)
Brace Extension	nein	ja	ja
Tilde-Extension	nein	ja	ja
`$(<datei)` kurz für `$(cat datei)`	nein	nein (13)	ja
for ((ex1; ex2; ex3))	nein	nein (17)	nein (17)

1. Keine Optionen `-a -p –f`.
2. `EXIT` nur als Nummer `0` und keine `DEBUG`, `ERR`, kein `trap –l`.
3. Kein Signal `ERR`. `DEBUG` erst ab Version 2.01.
4. Signal `DEBUG` nicht in der pdksh.
5. `typeset` bietet nicht alle Funktionen der `ksh -afFirx`, `-a` kennt die `ksh` nicht.
6. Die Orginal-`sh` hatte keine Jobfunktionen. Es gibt mittlerweile aber Versionen, die dies beinhalten. Diese müssen dann aber als `jsh` aufgerufen werden. `disown` ist dennoch nicht verfügbar.
7. Die Orginal-`sh` kannte keine Funktionen, dies hat sich mittlerweile geändert. Fast alle Versionen beherrschen dies. Die Syntax sieht aber so aus:

   ```
   <Funktionsname> ()
   { <Befehl1> ; ...
   }
   ```

8. Nur auf Systemen, die *Named Pipes* oder `/dev/fd/x` unterstützen. (x) ist dabei ein positive Ganzzahl.
9. Nur auf SunOS oder Systemen, die `/dev/fd/x` unterstützt.
10. `pdksh` und ksh93 beherrschen dies, nicht aber ksh88.
11. Nur in der ksh93 als Builtin, in der `pdksh` und ksh88 als `alias`.
12. Nur in der ksh93.
13. Erst ab Version 2.02.
14. Nicht alle Versionen von `sh` kennen `type`.
15. Es ist sogar häufig so, dass `test` ein externer Befehl ist und nicht in der `sh` eingebaut ist (kein Builtin).
16. Die Kornshell hat zwei Modi: den Vi- und den Emacs-Modus. Alle `ksh`-Shells teilen sich eine History, was verwirrt. Denn nicht immer ist so der letzte Befehl in der History der letzte von Ihnen eingegebene Befehl!
17. Bash ab Version 2.04, Kornshell ab ksh93 aber nicht pdksh (siehe auch *Änderungen in der Bash seit 2.02* weiter unten.

Tabelle A.2 zeigt, welche Shellvariablen von welcher Shell unterstützt werden. Aufgeführt haben wir an dieser Stelle nur die Variablen, die wir innerhalb der Shell genutzt oder angesprochen haben.

Tabelle A.2: Shellvariablen und ihre Verfügbarkeit unter Bourne-, Korn- und Bourne-Again-Shell

Shellvariable	sh	bash	ksh
BASH_VERSION	nein	ja	nein
COLUMNS	nein	ja	ja
HOME	ja	ja	ja
IFS	ja	ja	ja
KSH_VERSION	nein	nein	ja
LINENO	nein	ja	ja (4)
LINES	nein	ja	ja
OPTARG	ja	ja	ja
OPTERR	nein	ja	nein (1)
OPTIND	ja	ja	ja
PATH	ja	ja	ja
PIPESTATUS	nein	ja (3)	nein
PS1	ja	ja	ja
PS2	ja	ja	ja
PS3	nein	ja	ja
PS4	nein	ja	ja
RANDOM	nein	ja (2)	ja (2)
SHELL	ja	ja	ja
SHLVL	nein	ja	nein

- pdksh und ksh88 haben OPTERR nicht.
- RANDOM gibt eine Ganzzahl im Bereich von 0 bis 32767 aus.
- Erst ab Version Bash-2.01
- In der pdksh erst ab Version 5.2.14.

Noch einige kurze Anmerkungen zum getopts unter der sh:

- OPTERR wird nicht unterstützt.
- Trifft getopts auf einen Fehler in den Parametern, so gibt es eine Meldung auf die Fehlerausgabe aus. Dieses Verhalten kann unterbunden werden, indem als erstes Zeichen im Optionsstring ein »:« angegeben wird.
- Ist eine illegale Option angegeben worden, so steht in der nach getopts angegebenen Variablen ein ?.

  Tabelle A.3 listet einige Parameter auf, die mittels $<parameter> angesprochen werden können.

Parameter	sh	bash	ksh
-	ja	ja	ja
!	ja	ja	ja
?	ja	ja	ja
$	ja	ja	ja
*	ja	ja	ja
@	ja	ja	ja
#	ja	ja	ja
0	ja	ja	ja

Tabelle A3: Shell-Parameter

Die Parameterersetzungen, die von den Shellversionen unterstützt werden, zeigt Ihnen Tabelle A.4.

Ersetzung	sh	bash	ksh
${parameter:-word}	ja	ja	ja
${parameter:=word}	ja	ja	ja
${parameter:?word}	ja	ja	ja
${parameter:+word}	ja	ja	ja
${parameter-word}	ja	ja	ja
${parameter=word}	ja	ja	ja
${parameter?word}	ja	ja	ja
${parameter+word}	ja	ja	ja
${#parameter}	nein	ja	ja
${parameter%word}	nein	ja	ja
${parameter%%word}	nein	ja	ja
${parameter#word}	nein	ja	ja
${parameter##word}	nein	ja	ja
${parameter:offset}	nein	ja	nein
${parameter:offset:length}	nein	ja	nein
${!parameter}	nein(2)	ja	ja (1)

Tabelle A.4: Mögliche Parameterersetzungen in Bourne-, Korn- und Bourne-Again-Shell

- in der `pdksh` und ksh93, nicht in der ksh88
- mittels `eval` zu realisieren

Mittlerweile liegt die Bash in der Version 2.06b und die pdksh in der Version 5.2.14 vor. Allerdings hat sich in Bezug auf Shellskripten nicht viel geändert, lediglich einige Optionen sind hinzugekommen und Fehler behoben worden. Da die verschiedenen Linuxdistributionen nicht immer auf dem aktuellsten Stand sind, hier die wichtigsten Änderungen, welche sich primär auf das Schreiben von Skripten auswirken (können). Eine komplette Übersicht über alle behobenen Fehler finden Sie in den Änderungsübersichten (meistens Changelog oder News genannt), die z.B. auf den Homepages der Shells zu finden sind. In Klammern wird übrigens die Version aufgeführt, in welcher diese Änderung das erste Mal auftrat.

Änderungen in der Bash seit 2.02

- `set` ohne Parameter gibt Funktion nun so aus, dass sie wieder als Eingabe für eine neue Funktionsdefinition dienen können. Da die POSIX-Norm für Shells dies nicht erlaubt, geschieht dies nur, wenn die Bash nicht im POSIX-Modus läuft (2.04).
- Neue arithmetische Operatoren:
  - `var++`

    Gib den Wert von `var` zurück und erhöhe danach `var` um 1
  - `++var`

    Erhöhe `var` um 1 und gib dann den Wert von `var` zurück (2.04)
  - `var--` und `--var`

    Verhalten sich genau wie `var++` bzw. `++var`, nur dass der Wert von `var` um 1 erniedrigt wird.
- Neue Variable `FUNCNAME`, welche den Namen der aktuell ausgeführten Variablen zurückgibt. Eine Zuweisung auf diese Variable wird ohne Fehlermeldung ignoriert (2.04).
- Akzeptiert auch die ksh93-Syntax:

  ```
  for ((expr1 ; expr2; expr3 )); do list; done
  ```

  Diese Schleife initialisiert beim ersten Durchlauf mit `expr1`. Bei jedem weiteren Durchlauf wird dann `expr2` ausgeführt. Ist das Ergebnis des Ausdrucks 0, dann wird erst `list` ausgeführt und danach wird `expr3` ausgewertet. Wird ein Ausdruck weggelassen, dann wird dieser als 1 (false) ausgewertet. Der Rückgabewert dieser Schleife ist Wert des letzten ausgeführten Befehls in der Schleife oder `false`, wenn einer der Ausdrücke ungültig war (2.04).

Änderungen in der pdksh seit 5.2.13:

- Syntax: Ein Semikolon vor dem `then` ist nun optional `if [[ 1 ]] then ...` ist jetzt erlaubt (5.2.13).
- Funktionen: `$0` in Funktionen mit Funktionen im `sh` Stil verhält sich genauso wie `$0` in der `sh` (5.2.13).
- Erweitertes globbing: Innerhalb von `@(...)` und `${foo#...}` sind jetzt auch `( <muster> | <muster> )` erlaubt (5.2.13).
- `LINENO` wird unterstützt (5.2.14).
- Variablen-/Parameterersetzung: `${#array[*]}` gibt jetzt die Anzahl an Einträgen zurück und nicht den Index des höchsten Eintrages (5.2.14).
- `unset` gibt keinen Fehlerstatus mehr zurück, wenn eine nicht gesetzte Variable oder Funktion übergeben wird (2.06)
- `source` bzw. `.` setzt die Übergabeparameter `$0` ... `$9` (und weiter, falls nötig) wieder auf die Ursprungswerte vor dem Aufruf von `source` bzw. `.` zurück. Dies gilt allerdings nicht, falls das so aufgerufende Skript die Parameter mittels `set` umgesetzt hat. (2.06)
- Das `echo` Kommando der Bash aktzeptiert jetzt auch Oktalzahlen, die im Format `\0xxx` angegeben werden. Diese Notation wird neben der existierenden `\xxx` aktzeptiert, für größere Kompatibilität. (2.06)
- `select` gibt einen Fehlerstatus zurück, falls die Eingabe des Benutzer nicht gelesen werden kann.(2.05)
- `select` verhält sich nun so wie die Ksh wenn `REPLY` auf `NULL` gesetzt ist, wird das Auswahlmenu mit jedem Schleifendurchlauf ausgegeben. (2.05)
- Arrays: Zuweisungen ala `let b[2]=5` und `b[2}++` werden jetzt korrekt interpretiert. (2.05)
- `read` hat jetzt die Option `-u fd` mit der es möglich ist, direkt aus einem angegebenen Dateideskriptor zu lesen.(2.05)
- `type -P` erzwingt eine Suche durch den `$PATH` und `type -f` ignoriert evtl. passende Funktionsdefinitionen. (2.05)
- `<<< word` (2.05)
  Here-String / Here-Zeichenketten (siehe Kapitel 4.7.2)
- `[:word:]` (2.05)
  Zeichenbereiche mit vordefinierten Zeichenklassen/-mengen (siehe 2.3.4)

Anhang B

Das letzte Skript

»Und ist das Werk auch gut gelungen, schon verträgt es Änderungen«

Das Buch neigt sich endgültig dem Ende zu, und es wird Zeit, dass wir uns dem endgültig letzten Skript zuwenden. Es ist der CW-Commander, das Skript, das uns über fast die gesamte Länge dieses Buches mit Aufgaben, Rätseln und Problemen versorgt hat.

An dieser Stelle wollen wir als Autoren Ihnen unsere letzte Version vorstellen, damit Sie einmal das gesamte Skript zu sehen bekommen. In allen bisherigen Kapiteln haben wir Ihnen nur von Zeit zu Zeit eine komplette Version vorgestellt und danach immer nur die Verbesserungen/Unterschiede zum letzten Versuch vorgestellt. Das hat zwar mehr Platz für Aufgaben, Theorie und praxisnahe Beispiele gelassen, der CW-Commander hat allerdings darunter doch ein wenig gelitten, ging die Übersicht doch ein wenig verloren.

Damit dieser Anhang aber nicht nur bereits behandelte Probleme und Skriptteile aufwärmt, haben wir an dieser Stelle noch einige Verbesserungen eingebaut:

1. Dieses Skript basiert als einziges Skript in diesem Buch auf dem Hilfsprogramm `GetKey`.
2. Mittels der ⇆-Taste können Sie zwischen den beiden Fenstern wechseln.
3. Das zweite Fenster kann ebenfalls durchblättert werden, wenn es aktiv ist.

4. Das Blättern baut die Seite nur falls nötig wieder auf.

5. Ein `@` refresht den Bildschirm.

Zu Punkt 4 möchten wir nur noch so viel anmerken:

Eine Seite muss nur dann neu aufgebaut werden, wenn sich der Offset des Seitenanfangs seit der letzten Ausgabe verändert hat. Aus diesem Grunde wird jetzt eine Variable geführt, die den alten Offset festhält.

Für den zweiten Rahmen gehen wir genauso vor und nutzen entsprechende Variablen. Wenn wir den Rahmen mit ⇆ wechseln, tauschen wir die Variablen gegeneinander aus, sodass wir keine zusätzlichen Fälle für das Blättern beachten müssen. Nur beim ↵ achten wir darauf, dass `CW_NormalFile` nicht im rechten Rahmen (und damit für den Inhalt eines Tar-Archivs) aufgerufen wird.

Wir hoffen, dass wir alle Probleme beseitigt haben, die sich aus Dateinamen ergeben, die mit Leerzeichen versehen sind. Aus Gründen der Übersichtlichkeit gibt der CW-Commander keine `.*`-Dateien aus. Außerdem hat der CW-Commander einige Verbesserungen erfahren, was die Kodierung betrifft (Fehlerabfragen, kein `cut` mehr, um Zeichen aus einer Variablen zu ermitteln). Alle Änderungen haben Sie in diesem Buch schon kennen gelernt, nur im CW-Commander haben wir die Techniken nicht alle angewandt.

Kommen wir nun endgültig zum Quelltext des Skripts:

```
#!/bin/bash
# Die letzte Version des CW-Commanders
# Von:
# Christa Wieskotten und Bettina Rathmann
# für den Markt und Technik-Verlag
#
# Es funktioniert:
#  - Errorausgaben, Löschung
#  - Signale
#  - Archivierung
#  - Markierung bei Dateinamen mit " ", ":"
#  - Bildschirmsteuerung in Funktionen CW_Clear etc
#  - Volle Funktionalität im rechten Fenster
#  - GetKey (C) eingebaut und Abfragen angepaßt

function CW_Error()
{
 tput cup 0 0
 tput el
 CW_Print 0 0 "$1"
}

function CW_Delete ()
{ # Löscht die an $1 übergebenen Dateien
  # Format /datei/datei/ ...
```

```
  local lvMark=${1:1}
  while [ -n "$lvMark" ] ; do
      local lvWort=${lvMark%%/*}
      lvMark=${lvMark#*/}
      rm -f "$lvWort" 2>/dev/null >/dev/null || \
         CW_Error "Fehler: \"$lvWort\" kann nicht gelöscht werden"
    done

}

function CW_TrapSIGUSR1 ()
{ # Fängt SIGUSR1 ab und liest das akt. Verzeichnis nochmal
  CW_ReadDir "."
  CW_PrintDir 1 18 $tmpfile
  tput cup 22 8

}

function CW_TrapSIGINT ()
{
  if [ -n "$prid" ] ; then
     # Tarsicherung abbrechen
     kill -SIGTERM $prid 2>/dev/null
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $gvAtName
     at -d $gvLastAt
     prid=""
     CW_Print 0 0 "Sicherung abgebrochen!"
    else
     rm -f $tmptar
     rm -f $tmptarcnt
     rm -f $tmpfile
     rm -f $gvAtName
     at -d $gvLastAt
     echo
     exit 0
  fi
}

function CW_Archiv ()
{
  fnamen=$1
  sleep 1
  if [ -n "$fnamen" ] ; then
     fnamen=`echo "$fnamen" | tr ":" " "`
     typeset -i pro=`echo "$fnamen" |wc -w`
     typeset -i anz=100*pro
     tar cvfz $tmptar $fnamen >$tmptarcnt 2>/dev/null &
     typeset -i akt=0
     typeset -i erg=1
     prid=$!
     while [ $anz -gt $akt ] ; do
```

```
      akt=`wc -l <$tmptarcnt`
      akt=akt*100
      tput cup 10 38
      erg=akt/pro
      CW_Print 0 0 "$erg%"
    done
    rm $tmptarcnt
    prid=""
  fi
}

function CW_Clear ()
{ # Bildschirm löschen
  tput clear
}

function CW_SetBold ()
{  # Setzt den Modus Bold
   tput bold
}

function CW_SetNormal ()
{  # Setzt den Modus zurück
   tput sgr0
}

function CW_SetRev ()
{  # Setzt Inversmodus
   tput rev
}

function CW_Print  ()
{
 # Setzt an $1, $2 den Text $3
 tput cup $2 $1
 echo -n "$3"
}

function CW_Box ()
{ # Setzt das Fenster an Pos $1, $2 mit $3 Zeichen Breite und $4 Zeilen Höhe
  # Falls $5 gesetzt ist, so wird dies als Titel der Box ausgegeben
typeset -i anz=2
typeset -i xp=$1
typeset -i yp=$2
titel=${5:-""}
bt="+"
while [ $anz -lt $3 ] ; do
  bt="$bt-"
  anz=anz+1
done
bt="$bt+"
anz=1
CW_Print $xp $yp $bt
yp=yp+1
```

```
leze=`${gvpp}printf "!%$(($3-2)).$(($3-2))s!"`
while [ $anz -lt $(($4-1)) ] ; do
   CW_Print $xp $yp "$leze"
   anz=anz+1
   yp=yp+1
done
CW_Print $xp $yp $bt
if [ -n "$titel" ] ; then
   typeset -i len=${#titel}
   if [ $len -gt $3 ] ; then
      typeset -i rest=len-$3+6
      titel="..."`echo "$titel" | cut -c${rest}-`
      len=len-rest+4
   fi
   CW_SetBold
   xp=$3-len
   xp=xp/2+$1
   CW_Print $xp $2 "$titel"
   CW_SetNormal
fi
}

function CW_Mark ()
{
 # 1. Schon Eingetragen?
 local line=`sed -n -e "${zakt}p" $1`
 set -- $line
 shift 8
 local name=$*
 if [ -z "`echo $Mark | grep \"/$name/\"`" ] ; then
    Mark="$Mark$name/"
   else
    neu="/"
    Mark=${Mark:1}
    while [ -n "$Mark" ] ; do
      wort=${Mark%%/*}
      Mark=${Mark#*/}
      if [ "$wort" != "$name" ] ; then
        neu="$neu$wort/"
      fi
    done
    Mark=$neu
 fi
}

function CW_Swap ()
{ # Nimmt den Namen zweier Variablen und tauscht deren
  # Inhalte aus
  local lvBak
  eval lvBak=\$$1
  eval $1=\$$2
  eval $2=$lvBak
}
```

```
function CW_SwapVars ()
{  # Diese Funktion tauscht alle Variablen für die
   # Berechnung der Anzeigen für die linke und rechte
   # Box gegeneinander aus
   CW_Swap offset    gvOffsetR
   CW_Swap zakt      gvZAktR
   CW_Swap gvAltZeil gvAltZeilR
   CW_Swap gvAltOffs gvAltOffsR
   CW_Swap anz       gvAnzR
}


function CW_Eingabe()
{
  ein=""
  until [ -n "$ein" ] ; do
     tput cup 23 0
     echo -n "Eingabe:"
     ein=`"${gvGKP}"/gk`
     case $ein in
      "q" | "Q") break ;;
      "338" | "w" | "W")
                 gvAltZeil=zakt
                 gvAltOffs=offset
                 offset=offset+17
                 if [ $offset -gt $anz ] ; then
                    offset=offset-17
                   else
                    zakt=zakt+17
                    if [ $zakt -gt $anz ] ; then
                       zakt=anz
                    fi
                 fi
                ;;
      "339" | "z" | "Z")
                 if [ $offset -gt 2 ] ; then
                    gvAltZeil=zakt
                    gvAltOffs=offset
                    offset=offset-17 ;
                    zakt=zakt-17
                 fi
                 ;;
      "258" | "n" | "N")
                 if [ $zakt -lt $anz ] ; then
                    gvAltZeil=zakt
                    gvAltOffs=offset
                    zakt=zakt+1
                    if [ $zakt -gt $((offset+17)) ] ; then
                       offset=offset+17
                    fi
                 fi ;;
```

```
"259" | "l" | "L")
          if [ $zakt -gt 1 ] ; then
             gvAltZeil=zakt
             gvAltOffs=offset
             zakt=zakt-1
             if [ $zakt -lt $offset ] ; then
                offset=offset-17
             fi
          fi ;;
"m" | "M")       # Dateien (de-)markieren
            CW_Mark "$tmpfile"
            ;;
"a" | "A")       # Markierte Dateien archivieren
            CW_Archiv "$Mark"
          ;;
"330" | "d" | "D")   # Markierte Dateien löschen
          CW_Delete "$Mark"
          CW_ReadDir "."
          ;;
"403" | "@")   # Refresh
          CW_Clear
          CW_Box 0 1 40 20 "$verz"
          CW_Box 40 1 40 20
          CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn
             a = Archiv   d = Löschen "
          CW_Print 0 22 "s = Pos1  m = Mark."
          gvAltZeil=-1
          gvAltOffs=-1
          gvAltOffsR=-1
          gvAltZeilR=-1
          [ "$gvAktiv" = "R" ] && CW_SwapVars
          CW_PrintDir 1 18 $tmpfile
          [ "$gvAktiv" = "R" ] && CW_SwapVars
          ;;
"009" | "R" | "r")# Rahmen wechseln
          if [ "$gvAktiv" = "L" ] ; then
             gvAktiv="R"
          else
             gvAktiv="L"
          fi
          # Alle Variablen für Ausgabe tauschen
          CW_SwapVars
          ;;
"S" |"s" | "262") # Start / Pos 1
          if [ $offset -gt 1 -o $zakt -gt 1 ] ; then
             gvAltZeil=zakt
             gvAltOffs=offset
             offset=1
             zakt=1
          fi
          ;;
"010" |"")          # Return -> Dir oder Datei
          [ "$gvAktiv" = "R" ] && continue
```

```
                local line=`sed -n -e "${zakt}p" $tmpfile`
                set -- $line
                local ch=${1:0:1}
                shift 8
                datei="$*"
                if [ "$ch" = "d" ] ; then
                   Mark="/"
                   shift 8
                   local datei="$*"
                   CW_ReadDir "$datei"
                   echo "done"
                   break
                else
                   CW_NormalFile "$datei"
                fi ;;
       *) CW_Print 0 0 " >$Mark< "
          continue ;;
     esac
  done
}

function CW_At()
{
 if [ "$gvLastAt" != "" ] ; then
    at -d $gvLastAt
    gvLastAt=""
 fi
 echo "# At-Skript zur Überwachung des akt. Verzeichnisses">$gvAtName
 echo "gvPID=$$" >>$gvAtName
 local lvInh=`wc \`ls .\` 2>/dev/null |tail -1`
 echo "gvInh='$lvInh'" >>$gvAtName
 echo "set -- \$gvInh" >>$gvAtName
 echo "gvZei=\$1 ; gvWort=\$2 ; gvByte=\$3" >>$gvAtName
 echo 'gvInh2=`wc \`ls .\` 2>/dev/null | tail -1`' >>$gvAtName
 echo "set -- \$gvInh2" >>$gvAtName
 echo 'if [ $1 -ne $gvZei -o $2 -ne $gvWort -o $3 -ne $gvByte ] ; then'
   >>$gvAtName
 echo "   kill -SIGUSR1 \$gvPID" >>$gvAtName
 echo "fi" >>$gvAtName
 echo "exit 0" >>$gvAtName
 at -f $gvAtName now + 4 minutes 2>/dev/null
 gvLastAt=`at -l | tail -1 | awk '{ print $1 }'`
}

function CW_ReadDir ()
{
 offset=1
 zakt=1
 CW_At
 gvAltZeil=-1 # Fehlte in letzter Version! Fehler
 gvAltOffs=-1 # Dito
 cd "$1"  2>/dev/null  && verz="`pwd`"
 CW_Print 0 0 " "
```

```
 anz=`ls -lAd .. * 2>/dev/null |tee $tmpfile | wc -l`
 CW_Box 0 1 40 20 "$verz"
}

function CW_PrintDir()
{
  # Ausgabe der Dateien
  typeset -i xp=$1
  typeset -i hoehe=$2
  ausdatei=$3
  typeset -i i=0
  typeset -i akt=0
  while [ $i -lt $hoehe -a $akt -lt $anz ] ; do
   akt=i+offset
   if [ $gvAltOffs -eq $offset ] ; then
      let "$akt == $gvAltZeil || $akt == $zakt"
      if [ $? -eq 0 ] ; then
         local lvOut="ja"
        else
         local lvOut="nein"
      fi
     else
      local lvOut="ja"
   fi
   if [ "$lvOut" = "ja" ] ; then
      zeile=`sed -n -e "${akt}p" $ausdatei`
      set -- $zeile
      local lvSize=$5
      local lvRechte=$1
      shift 8
      local datei=$*
      if [ ${lvRechte:0:1} = "l" ]  ; then
        datei=`echo "$datei" | sed -n -e 's/^\([^/]*\) -> \([^\000]*\)/\1/p'`
      fi
      if echo "$Mark" | grep -q "/$datei/" ; then
         CW_SetRev
      fi
      if [ $zakt -eq $akt ] ; then
         CW_SetBold
      fi
      local text="`${gvpp}printf "$lvRechte|%9i | %-15s" $lvSize
"${datei:0:14}"`"
      CW_Print $xp $((i+2)) "$text"
      CW_SetNormal
   fi
   i=i+1
  done
  while [ $i -lt $hoehe ] ; do
    CW_Print $xp $((i + 2)) "`${gvpp}printf \"%10s|%10s|%16s\"`"
    i=i+1
  done
}
```

```
function CW_NormalFile () {
# Schaut nach, welchen Dateityp die Datei hat und reagiert auf Tar / Zip
# Archive
typ=`file $1 | cut -f2 -d":"`
if [ "$typ" = " GNU tar archive" -o "$typ" = " POSIX tar archive" ] ; then
   rm -f $tmptarcnt
   tar tvf $1 | awk '{ print $1 " 1 r r " $3 " " $4 " " $5 " " $6 " " $8 }'
     >>$tmptarcnt
   # Anzeigevars tauschen
   CW_SwapVars
   anz=`wc -l < $tmptarcnt`
   offset=1
   zakt=1
   CW_PrintDir 41 18 $tmptarcnt "$1"
   # Und den alten Zustand wieder herstellen
   CW_SwapVars
fi
}

# Pfad zum GetKey ermitteln, Trap für Exit setzen
#
trap 'stty sane' EXIT
cd "`dirname $0`"
gvpp="/usr/bin/"
gvGKP=`pwd`
Mark="/"
CW_Clear
verz=${1:-"`pwd`"}
trap "CW_TrapSIGINT" SIGINT
trap "CW_TrapSIGUSR1" SIGUSR1
CW_Box 0 1 40 20 "$verz"
CW_Box 40 1 40 20
CW_Print 0 21 "z = PgUp  w = PgDn  l = CuUp  n = CuDn  a = Archiv   d = Löschen
"
CW_Print 0 22 "s = Pos1  m = Mark."
#
# Zeilen Offset setzen, Dateien ermitteln, Anzahl ermitteln
#
typeset -i offset=1
tmpfile="/tmp/cwc$$.tmp"
tmptar="/tmp/backup"
tmptarcnt="/tmp/tar.cnt$$"
gvAtName="/tmp/at$$"
gvLastAt=""
cd "$verz"
rm -f $tmpfile $tmptarcnt $tmptar
gvAktiv="L"                # Linke oder Rechte Box aktiv
typeset -i anz=`ls -lAd * .. 2>/dev/null | tee $tmpfile | wc -l`
typeset -i zakt=1          # Zähler akt. Zeile
typeset -i gvAltOffs=-1    # Alter Offset
typeset -i gvAltZeil=-1    # Alte aktuelle Zeile
```

```
# Alle Variablen für den rechten Rahmen
typeset -i lvOffsetR=1     # Offset
typeset -i lvAnzR=0
typeset -i lvZAktR=1       # Zähler akt. Zeile
typeset -i gvAltOffsR=-1   # Alter Offset
typeset -i gvAltZeilR=-1   # Alte aktuelle Zeile
#
# Refresh setzen,
#
CW_At
until [ "$ein" = "q" ] ; do
  [ "$gvAktiv" = "L" ] && CW_PrintDir 1 18 $tmpfile
  [ "$gvAktiv" = "R" ] && CW_PrintDir 41 18 $tmptarcnt
  CW_Eingabe
done
rm -f $tmpfile $gvAtName
rm -f $tmptar $tmptarcnt
exit 0
```

Das war endgültig das letzte Skript in diesem Buch. Auch dieses Skript ist weder fertig noch vollkommen, nutzen Sie es doch als Basis für Ihre eigenen Versuche!

Keine Ahnung, wo Sie anfangen können? Und es drängt sich keine Aufgabe auf, die gelöst werden müsste? In diesem Falle haben wir an dieser Stelle einige Vorschläge, was Sie mit diesem Skript noch anstellen können:

- Es gibt neben der hier implementierten Möglichkeit, Leerzeichen in Dateinamen zu behandeln, auch noch die Möglichkeit, mit `xargs` zu arbeiten.
- Bei der Kontrolle des Skripts für die neue Auflage ist uns aufgefallen, dass die Bash mittlerweile einen Fehler ausgibt, wenn `printf` z.B. folgende Zeile ausgeben soll:

  ```
  buch@koala:/home/buch > printf "-rwxr-x--x|%9i" 123
  bash: printf: illegal option: -r
  printf: usage: printf format [arguments]
  buch@koala:/home/buch >
  ```

  Das Problem klärt sich, wenn man `type -a printf` eintippt:

  ```
  buch@koala:/home/buch > type -a printf
  printf is a shell builtin
  printf is /usr/bin/printf
  buch@koala:/home/buch >
  ```

  Der Grund: Mittlerweile hat sich die Rechnerkonfiguration der Autoren geändert (Christa hat meinen alten und ich bin bei PIII 700 MHz, 256 MB Ram gelandet) und eine Neuinstallation von Linux war angesagt. Dabei hatte sich aber auch die Version und Konfiguration der Bash geändert, weshalb `printf` mittlerweile als Shell-Builtin genutzt wird, welches aber

mit führenden "-"-Zeichen nicht klarkommt. Das Shell könnte sicherstellen, das es immer mit der externen Version von `printf` arbeitet.

- Gefiel Ihnen eigentlich der `sed` aus Kapitel 10? Fanden alles ganz nett, aber ein praktischer Nutzen für Ihre Arbeit war nicht zu erkennen? Wie wäre es dann mit:

```
#!/bin/bash
# dtree: prints a directory tree from the current directory downwards
#        or specify a directory from which to print
# e.g.   dtree
# e.g.   dtree mydir
#
# Variable: Angegebenes Verzeichnis oder aktuelles
dir=${1:-.}
# Ins passende Verzeichnis wechseln
(cd $dir; pwd)
#
find $dir -type d -print | sort -f | sed -e "
        s:^$1::
        /^$/d
        /^\.$/d
        s:[^/]*/\([^/]*\)$:|-----\1:
        s:[^/]*/:|     :g"
```

Dieses Skript haben wir vom Seder's Grabbag geklaut (URL im Anhang D), eine Webseite, die für solche Skripten eine sehr gute Anlaufstelle ist ...

Was gibt es denn für `/etc` aus (leicht gekürzt um einige Verzeichnisse, um die Ausgabe nicht über drei Seiten auszudehnen)?

```
buch@koala:/home/buch > ./dtree.sh /etc
/etc
|-----cron.d
|-----httpd
|     |-----modules
|     |-----ssl.crl
|     |-----ssl.crt
|     |-----ssl.csr
|     |-----ssl.key
|     |-----ssl.prm
|-----init.d
|     |-----boot.d
|     |-----rc0.d
|     |-----rc1.d
|     |-----rc2.d
|     |-----rc3.d
|     |-----rc4.d
|     |-----rc5.d
|     |-----rc6.d
|     |-----rcS.d
|-----mail
```

```
|-----opt
|     |-----gnome
|     |     |-----CORBA
|     |     |     |-----servers
|     |     |-----sound
|     |     |     |-----events
|     |-----kde2
|     |     |-----share
|     |     |     |-----config
|-----pam.d
|-----permissions.d
|-----profile.d
|-----rc.config.d
|-----security
|-----skel
|     |-----.kde
|     |     |-----share
|     |     |     |-----config
|     |-----.xfm
|-----ssh
```

Nun, wenn das nicht eine nette Ausgabe für den CW-Commander (rechtes Fenster) wäre…

- Der letzte Punkt hat sie unterfordert? Nun dann könnten Sie ja eine Navigation mit den Cursortasten durch den Baum versuchen …

Das war es nun aber wirklich alles – und nein, die Lösungen fehlen in diesem Anhang B tatsächlich nicht. Wenn Sie so weit sind, wie wir glauben (und dieses Buch halbwegs etwas taugt, etwas woran wir Autoren auf keinen Fall zweifeln `:o)`), dann sollten Ihnen diese Anregungen keinerlei Probleme mehr bereiten.

Anhang C

Taste abfragen in C

»Bei der Eroberung des Weltraums sind zwei Probleme zu lösen: die Schwerkraft und der Papierkrieg. Mit der Schwerkraft wären wir fertig geworden.« –
Wernher von Braun

C.1 Einleitung

Wie bereits mehrfach im Buch angesprochen, möchten wir Ihnen in diesem Anhang noch ein kleines C-Programm mit auf den Weg geben, mit dem Sie Tasten einzeln abfragen können. Im Gegensatz zum Miniskript, das `dd` nutzte, hat dieses Programm den Vorteil, dass Sie auch Tasten abfragen können, die mehr als ein Zeichen zurückgeben. Nachteilig ist die Tatsache, dass dieses Programm nur für die Steuercodes der PC-Tastatur (unter Linux `TERM="linux"`) geschrieben wurde.

Es wäre zwar möglich, das Programm so umzustellen, dass es `TERM` auswerten und die Rückgabewerte der Tasten aus der `terminfo` oder `termcap` ermitteln würde. Leider würde dies das Programm verlängern und damit den Rahmen des Anhangs sprengen.

Deshalb haben wir uns dazu entschlossen, Ihnen ein kleines Programm an die Hand zu geben, das Sie bei Bedarf schnell an Ihre Bedürfnisse anpassen können.

Die Erklärungen sind notwendigerweise etwas knapp gehalten, bitte erwarten Sie nicht, dass wir Ihnen auch noch C-Programmierung beibringen :-) Wenn Sie allerdings etwas Ahnung von C haben, dürften Ihnen die hier gegebenen Hinweise den richtigen Weg weisen.

C.2 Die Rückgabewerte

Das Programm wertet alle Strg+Zeichen-Tasten aus, bis auf Strg+S und Strg+Q. In Tabelle C.1 finden Sie die Rückgabewerte des Programms aufgeführt.

Tabelle C.1: Rückgabewerte von GetKey

Strg+A	001	Strg+Z	026
Esc	027	Entf	330
F1	400	F2	401
F2	402	F4	403
Pos 1	262	Ende	360
↑	259	↓	258
←	260	→	261
Bild↓	338	Bild↑	339
Einfg	361	Pos 1	262
Alle anderen Tasten	000	Alphanumerische Zeichen 0-9, A-Z etc.	Gibt das Zeichen selbst aus, also A, B 0 usw.

Nach dem Auslösen der Esc-Taste dauert es ca. eine Sekunde, bis das Programm zurückkommt. Dies liegt daran, dass auch Funktionstasten und Cursortasten mit ESC anfangen. Um sicher zu sein, dass es keine dieser Tasten ist, wartet das Programm.

Alle Tasten, die sich nicht drucken lassen (Cursorblock, Funktionstasten, Strg+Zeichen) werden von diesem Programm als dreistellige Zahl mit führenden Nullen ausgegeben. So lässt sich Strg+A von 1 unterscheiden.

C.3 Das Programm

Das folgende Programm wurde unter Linux 2.0.35 überprüft und sollte (bis auf die Escapesequenzen der Tasten) so auf jedem POSIX-konformen System laufen.

```
#include <termios.h>
#include <unistd.h>
#include <signal.h>
static struct termios V_GK_SaveTerm;
static int V_GK_SaveFD = -1;
int GK_TTYCBreak(int p_fd)
{
  /* Diese Funktion setzt den Modus für den Eingabekanal
     so um, dass die Tasten einzeln abgefragt werden können.
  */

 struct termios l_buff;

 if (tcgetattr(p_fd,&V_GK_SaveTerm) <0)
    return(-1);
 l_buff             = V_GK_SaveTerm;
 l_buff.c_lflag     = l_buff.c_lflag & ~(ECHO | ICANON | ISIG);
 l_buff.c_cc[VMIN]  = 1;
 l_buff.c_cc[VTIME] = 0;
 if (tcsetattr(p_fd,TCSAFLUSH, &l_buff)<0)
    return(-1);
 V_GK_SaveFD = p_fd;
 return(0);
}
int GK_TTYReset(int p_fd)
{
 /* Setzt das Terminal wieder in den normalen Modus
    zurück
 */
 if (tcsetattr(p_fd,TCSAFLUSH,&V_GK_SaveTerm)<0)
    return(-1);
 return(0);
}
static void GK_CatchTimeout(int p_sig)
{
  /* Diese Routine wird aufgerufen, wenn SIGALARM
     an unser Programm geschickt wird. Dies kann in
     diesem Programm nur geschehen, wenn das zweite
     read  keine Zeichen mehr lesen kann, sprich: Es folgte
     keine Escape-Sequenz und damit keine Funktionstaste. Es
     wurde nur die Taste ESC gedrückt.
  */
  printf("027");
  GK_TTYReset(STDIN_FILENO);
  exit(0);
}

int main(int argc, char *argv[])
{int   i, l_rd;
 char  c, l_buff[10];
 /* 1. Den Modus des Terminals umsetzen
       und Signalhandler für SIGALRM aufsetzen
 */
```

```
if (GK_TTYCBreak(STDIN_FILENO)< 0)
   { printf("-1");
     exit(1);
   }
signal(SIGALRM, GK_CatchTimeout);

/* 2. Das erste Zeichen lesen */

if (read(STDIN_FILENO,&c,1) <1)
   { /* Modus wieder zurückstellen, Fehlermeldung
        und Programm beenden
     */
     GK_TTYReset(STDIN_FILENO);
     printf("-1");
     exit(1);
   }
/* 3. Wurde ein ESC gelesen?  */
if (c == 27)
    { /* 4. Ja, es könnte eine Escapesequenz sein.
            Timeout setzen und versuchen, den Rest zu lesen
      */
    alarm(1);
    l_rd=read(STDIN_FILENO, l_buff, 3);
    alarm(0);
    l_buff[l_rd]=0;
    /* 5. Ausgesuchte Tasten prüfen */
    i=0;
    if (strcmp(l_buff,"[1~")==0)
       i = 262;  // Pos 1
    if (strcmp(l_buff,"[2~")==0)
       i = 331;  // Insert
    if (strcmp(l_buff,"[3~")==0)
       i = 330;  // Entf
    if (strcmp(l_buff,"[B")==0)
       i = 258;   // CuDn
    if (strcmp(l_buff,"[A")==0)
       i = 259;   // CuUp
    if (strcmp(l_buff,"[D")==0)
       i = 260;   // CuLe
    if (strcmp(l_buff,"[C")==0)
       i = 261;   // CuRi
    if (strcmp(l_buff,"[4~")==0)
        i = 360;  // End
    if (strcmp(l_buff,"[5~")==0)
       i = 339;   // PgUp
    if (strcmp(l_buff,"[6~")==0)
       i = 338;   // PgDn
    if (strcmp(l_buff,"[[A")==0)
       i = 400;   // F1
    if (strcmp(l_buff,"[[B")==0)
       i = 401;   // F2
    if (strcmp(l_buff,"[[C")==0)
       i = 402;   // F3
```

```
     if (strcmp(l_buff,"[[D")==0)
        i = 403;   // F4
     printf("%03d",i);
    }
  else
    { /* 6. Nein, kein ESC, entweder ein Zeichen < " ", dann
            numerischen Wert ausgeben, sonst das Zeichen
            ausgeben
      */
      if ((c < 32) || (c == 127))
         printf("%03d",c);
        else
         printf("%c",c);
    }
 /* 7. Modus zurückstellen und beenden */
 GK_TTYReset(STDIN_FILENO);
 exit(0);
}
```

Wenn Sie das Programm als `gk2.c` gespeichert haben, so sollte ein `cc gk2.c -o gk2` genügen, um das Programm als `gk2` nutzen zu können.

```
buch@koala:/home/buch > cc gk2.c -o gk2
buch@koala:/home/buch > gk2      # Hier "1" gedrückt
1buch@koala:/home/buch > gk2     # Hier CTRL-A gedrückt
001buch@koala:/home/buch >
```

C.4 Anpassen an andere Terminals

Unter Linux führt das Drücken von F1 dazu, dass folgende Zeichen gesendet werden: `ESC [ [ A`

Unser Programm muss also vier Zeichen lesen, um bestimmen zu können, welche Funktionstaste (F1, F2, aber auch Cursortasten) gedrückt wurde. Aus diesem Grund liest es erst ein Zeichen (Punkt 2.) und prüft dann, ob es Escape war (Punkt 3.). Falls ja, wird noch weitergelesen, aber spätestens nach einer Sekunde abgebrochen (`alarm()`). Konnte kein weiteres Zeichen gelesen werden, so war es keine Funktions- oder Cursortaste.

Wie können nun neue Sequenzen bestimmt werden, wenn Ihr Terminal von unseren Einstellungen abweicht?

Möglichkeit 1:

Geben Sie einfach in der Shell `cat` ein, und drücken Sie dann die Taste Ihrer Wahl. Es wird die Escapesequenz ausgegeben, wobei Escape als `^[` erscheint. Hier ein Beispiel für F1:

```
buch@koala:/home/buch > cat
^[[[A
buch@koala:/home/buch >
```

Brechen Sie die Eingabe ab mit Strg+C. Wie Sie sehen, wird genau die Sequenz ausgegeben, die ich weiter oben angeführt hatte. Die entsprechende C-Abfrage lautet dann:

```
if (strcmp(l_buff,"[[A")==0)
        i = 400;   // F1
```

Bedenken Sie, dass das Escape ja schon gelesen wurde, wenn das Programm auf diese Abfrage stößt.

Möglichkeit 2:

Das Programm `infocmp` gibt genau die Informationen aus, die unser Programm benötigt. Dabei werden die Tasten benannt. Die Zeichenfolgen für die Tasten werden hinter dem Namen und einem Gleichheitszeichen ausgegeben. Dabei wird das Escape durch `\E` dargestellt. Schauen wir uns einmal an, was `infocmp` für `TERM=Linux` ausgibt (gekürzt):

```
buch@koala:/home/buch > infocmp linux
#       Reconstructed via infocmp from file: /usr/lib/terminfo/l/linux
linux|linux console,
        am, bce, eo, mir, msgr, xenl, xon,
        colors#8, it#8, pairs#64,
        kcub1=\E[D, kcud1=\E[B, kcuf1=\E[C, kcuu1=\E[A,
        kdch1=\E[3~, kend=\E[4~, kf1=\E[[A, kf2=\E[[B, kf3=\E[[C, kf4=\E[[D,
        kf5=\E[[E, kf6=\E[17~, kf7=\E[18~, kf8=\E[19~, kf9=\E[20~,
        khome=\E[1~, kich1=\E[2~, knp=\E[6~, kpp=\E[5~, kspd=^Z,
        nel=^M^J, op=\E[39;49m, rc=\E8, rev=\E[7m, ri=\EM,
        rmacs=\E[10m, rmir=\E[4l, rmpch=\E[10m, rmso=\E[27m,
        rmul=\E[24m, rs1=\Ec, sc=\E7, setab=\E[4%p1%dm,
        setaf=\E[3%p1%dm,
        sgr0=\E[m, smacs=\E[11m, smir=\E[4h, smpch=\E[11m,
        smso=\E[7m, smul=\E[4m, tbc=\E[3g, u6=\E[%i%d;%dR,
        u7=\E[6n, u8=\E[?6c, u9=\E[c, vpa=\E[%i%p1%dd,
  [...}
```

Ein Blick auf die Namen zeigt uns einiges, was wir vor allem in Kapitel 4 besprochen haben: `sgr0`, `setab` und `colors`, um nur einige zu nennen. Der Name für unsere Funktionstaste F1 lautet `kf1`, und die eingetragene Zeichenkette lautet:

```
kf1=\E[[A
```

Und da `\E` für Escape steht, haben wir hier wieder die gleiche Zeichenfolge ermittelt.

Schauen wir jetzt nach, was für `kf1` im `xterm` (Terminalprogramm unter X11) eingetragen ist (ebenfalls gekürzt):

```
buch@koala:/home/buch > infocmp xterm
#       Reconstructed via infocmp from file: /usr/lib/terminfo/x/xterm
xterm|vs100|xterms|xterm terminal emulator (X Window System),
        am, bce, km, mir, msgr, xenl,
        colors#8, cols#80, it#8, lines#24, pairs#64,
```

```
       acsc=++\,\,--..OO``aaffggiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
        clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
        csr=\E[%i%p1%d;%p2%dr, cub=\E[%p1%dD, cub1=^H,
        cud=\E[%p1%dB, cud1=^J, cuf=\E[%p1%dC, cuf1=\E[C,
        cup=\E[%i%p1%d;%p2%dH, cuu=\E[%p1%dA, cuu1=\E[A,
        cvvis=\E[?25h, dch=\E[%p1%dP, dch1=\E[P, dl=\E[%p1%dM,
        dl1=\E[M, ech=\E[%p1%dX, ed=\E[J, el=\E[K, el1=\E[1K,
        enacs=\E(B\E)0, flash=\E[?5h\E[?5l, home=\E[H,
        hpa=\E[%i%p1%dG, ht=^I, hts=\EH, ich=\E[%p1%d@, ich1=\E[@,
        il=\E[%p1%dL, il1=\E[L, ind=^J,
        is2=\E7\E[r\E[m\E[?7h\E[?1;3;4;6l\E[4l\E8\E>,
        kDC=\EOn, kIC=\EOp, kLFT=\EOt, kNXT=\EOr, kPRV=\EOx,
        kRIT=\EOv, ka1=\EOw, ka3=\EOy, kb2=\EOu, kbeg=\EOE, kbs=\177,
        kc1=\EOq, kc3=\EOs, kcub1=\EOD, kcud1=\EOB, kcuf1=\EOC,
        kcuu1=\EOA, kdch1=\E[3~, kend=\EOF, kent=\EOM, kf0=\E[21~,
        kf1=\E[11~, kf10=\E[21~, kf11=\E[23~, kf12=\E[24~,
        kf13=\E[25~, kf14=\E[26~, kf15=\E[28~, kf16=\E[29~,
        kf17=\E[31~, kf18=\E[32~, kf19=\E[33~, kf2=\E[12~,
        kf20=\E[34~, kf21=\E[35~, kf22=\E[36~, kf3=\E[13~,
        kf4=\E[14~, kf5=\E[15~, kf54=\EOo, kf55=\EOj, kf56=\EOm,
  [...]
```

Für `xterm` findet sich ein `kf1=\E[11~`, was dazu führt, dass eine Abfrage auf F1 in C wie folgt programmiert werden müsste:

```
if (strcmp(l_buff,"[11~")==0)
        i = 400;   // F1
```

Weitere Informationen zur Bedeutung der von `infocmp` ausgegebenen Einträge finden Sie unter *terminfo(5)*.

Wir möchten nochmals darauf hinweisen, dass dieses Programm nur als Demonstration, wie Tasten abgefragt werden können, zu sehen ist. Schon allein die Tatsache, dass die Cursor- und Funktionstasten fest auf einen Terminaltyp programmiert sind, schränkt seinen Nutzen ein.

Außerdem wurden einige Abfragen bewusst einfach gehalten, um somit auch Lesern mit wenig(er) C-Erfahrung die Chance zu bieten, das Programm zu verstehen.

Sie können das Programm aber gern als Basis für eigene Versuche nutzen, um diese Nachteile auszumerzen.

Mit dem Skript aus Kapitel 4 und diesem C-Programm haben Sie jetzt zwei Möglichkeiten, Tasten einzeln abzufragen.

Anhang D

Ressourcen im Netz

»Und wenn wir schon nicht gewinnen, dann treten wir ihnen wenigstens den Rasen kaputt« –
Rolf Rüßmann

... wenn Sie nicht gewinnen können und die Shell Ihnen wiederholt den Stinkefinger zeigt, dann sollten Sie auf Online-Ressourcen zurückgreifen. Hier eine kleine Aufstellung der wichtigsten URLs und Newsgroups, meistens in englischer Sprache. Die Verweise stehen in keiner besonderen Reihenfolge.

D.1 Newsgroups

- *comp.unix.shell*

  Fragen rund um die Shells, sei es `sh`, `ksh`, `bash`, `csh` und wie sie nicht alle heißen.

- *de.comp.os.unix.shell*

  Das Gleiche auf deutsch.

D.2 World Wide Web

- *http://web.cs.mun.ca/~michael/pdksh/*

  Die Homepage der pdksh

- *http://www.kornshell.com/*

  Informationen rund um die Kornshell. Das Original wird von AT&T vertrieben, und hier findet sich alles von Interesse.

- *http://www.gnu.org/manual/manual.html*

  Die Anlaufstelle für die aktuelle Dokumentation zu Bash, `find` und vielen andere Utilities, die vom GNU-Projekt zur Verfügung gestellt werden.

- *ftp://ftp.cwru.edu/pub/bash/FAQ*

  Die FAQ zur Bash. Enthält Informationen zu den Unterschieden zwischen den verschiedenen Shells und zu den verschiedenen Versionen der Bash, zur Programmierung und zur Bedienung.

- *http://www.ptug.org/sed/sedfaq.html*

  Das FAQ (Frequently Asked Questions = Häufig gestellte Fragen) zum `sed`.

- *http://linuxgazette.net/*

  Die Linux-Gazette, ein englischsprachiges Online-Magazin, thematisiert Probleme der Shellprogrammierung häufiger.

- *http://www.linux-magazin.de/*

  Das Online-Angebot des deutschen Linux Magazins. Auch einige Artikel über Shellprogrammierung finden sich hier.

- *http://www.infodrom.org/projects/manpages-de/*

  Anlaufstelle für die deutschen Manpages. Noch sind nicht alle übersetzt und für Linux ausgerichtet, aber das ist keine so große Einschränkung, oder?

- *http://www.google.com/*

  Okay, hat nicht wirklich etwas mit Shellprogrammierung zu tun. Aber es ist eine Suchmaschine auf Linux-Basis. Und eine sehr gute noch dazu. Die Ergebnisse können sich wirklich sehen lassen.

- *ftp://ftp.gnu.org/gnu/bash/*

  Der FTP-Server des GNU-Projekts. Hier mit dem Unterverzeichnis, in dem die verschiedenen Bashversionen im Quelltext zu finden sind.

- *http://www.gnu.org/order/ftp.html*

  Liste der Mirrors und weitere Informationen zu den Quellpaketen von GNU.

- *http://www.gnu.org/software/bash/*

  Die Homepage der Bash mit Hinweisen und Verweisen auf die Dokumentation.

- *http://www.debian.org/*

  Homepage der Debian-Distribution für Linux. Hier sind auch die Quellen für die `pdksh` zu finden. Einfach nach dem Paket `pdksh` mit der Paketsuchfunktion suchen.

- *http://www.dbnet.ece.ntua.gr/~george/sed/*

  Tipps und Tricks zu `sed` mit den verschiedensten Anwendungsgebieten.

- *http://spazioinwind.libero.it/seders/*

  The Seder's Grab-bag. Skripten, Ressourcen und Einführung in `sed`. Alles, was man so braucht oder auch nicht `:-)`

  Das Paradies auf Erden für Freunde des `sed`.

- *http://www.tldp.org/LDP/abs/html/*

  Ein How-To zur Programmierung der Bash.

- *http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html*

  BASH-Homepage

D.3 Die Skripten zu diesem Buch ...

… können Sie auf der Markt+Technik-Website unter *www.mut.de* herunterladen. Geben Sie unter SUCHE: einfach den Buchtitel ein, um auf die Katalogseite zu diesem Buch zu gelangen. Dort können Sie sich das Archiv über einen Link herunter laden.

Stichwortverzeichnis

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen