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 Skripte 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 Skripte sind nicht viel besser als Handarbeit, die Sie ja gerade durch Skripte 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 Teilausdrükken 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.
str1 = str2	`[ "A" = "a" ]`	Wahr, wenn beide Zeichenketten identisch sind. Das Ergebnis dieses Beispiels ist übrigens falsch, da zwischen Groß- und Kleinschrift unterschieden wird.

Ausdruck	Beispiel	Erklärung
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.

Tabelle 3.1: Gültige Teilausdrücke zur Verwendung in Bedingungen (Fortsetzung)

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`-Variable 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 <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     -> 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 Fehlerfalle 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 soviel:

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 Skripte, 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 entgegen nimmt 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 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!	

Abbildung 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 "HHallo 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, so daß 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 Skripte 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 Parameter 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« dieser Problematik nochmals annehmen.

Nun wird es Zeit für ein wenig Praxis, denn die letzten beiden Skripte 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 >
```

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 diese
Kapitel direkt nach Kapitel 8 geschrieben wurde, kann ich noch nicht
sagen, wieviel Skripte 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 ein. `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 werden. 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 wie unter 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
...
```