UNIX Systemverwaltung

Kristian Köhntopp

29. Juli 1996

Kapitel 1

Die UNIX Shell /bin/sh

1.1 Von Kommandos und Optionen

Wenn man in eine UNIX–Shell eine Kommandozeile eingibt, wird diese von der Shell erst einmal auf verschiedene Arten und Weisen bearbeitet, bevor die Shell beginnt, das Kommando tatsächlich auszuführen. Die Shell unterteilt die Kommmandozeile in Worte und beginnt damit, diese Worte nach bestimmten Buchstaben zu durchsuchen. Einige Worte werden von der Shell direkt bearbeitet und aus der Kommandozeile ersatzlos gestrichen, andere Worte durch ein oder mehrere andere Worte ersetzt. Das Resultat ist eine Zeile, die aus null oder mehr Worten besteht. Das erste Wort dieser Zeile wird als der Name eines Kommandos aufgefaßt und im Suchpfad der Shell gesucht. Existiert es, erzeugt die Shell einen Unterprozeß, in dem dieses Kommando ausgeführt wird. Möglicherweise vorhandene weitere Worte werden diesem Kommando als Parameter mitgegeben.

Während sich die folgenden fünf Kapitel vornehmlich mit den verschiedenen Ersetzungen beschäftigen, die die Shell an einer Kommandozeile vornimmt, soll sich dieses Kapitel mit der Struktur von UNIX–Kommandos beschäftigen.

Grundsätzlich ist jeder UNIX–Befehl in der Interpretation seiner Argumente frei. Das bedeutet, daß ein UNIX–Befehl Parameter und Optionen in jeder beliebigen verkorksten Syntax erhalten kann, die sich ein verdrehtes Programmiererhirn auszudenken in der Lage ist. Die UNIX–Programmierumgebung stellt jedoch die Bibliotheksfunktion `getopt(3)` zum Analysieren der Kommandozeile zur Verfügung und wer Programme schreiben möchte oder muß, die sich in die UNIX–Umgebung einfügen, der wird diese Routine benutzen. Nach den Regeln von `getopt(3)` gilt für UNIX–Kommandos:

1. Kommandonamen sind zwischen zwei und neun Zeichen lang.

2. Kommandonamen bestehen ausschließlich aus Kleinbuchstaben und Ziffern.

   Beispiele für gültige Kommandonamen im Sinne dieser beiden Regeln sind **ls**, **fstat** oder **hostname**. Beispiele für ungültige Kommandonamen im Sinne dieser Regel sind **[** (ein alternativer Name des Kommandos **test**), **su.nowheel** oder **cacheAFMData**.

3. Optionsbuchstaben bestehen nur aus einem einzelnen Zeichen.

4. Alle Optionen beginnen mit einem – (Minuszeichen).

   Beispiele für Kommandos mit Optionen im Sinne dieser beiden Regeln sind **rm –rf**, **cpio –ocv** oder **ls –laRF**. Beispiele für Kommandos mit Optionen, die gegen diese beiden Regeln verstoßen sind **tar cv**, **dd if=infile of=outfile** oder **ar tv**.

5. Optionen ohne Argumente dürfen hinter einem einzigen Minuszeichen zusammengefaßt werden. Das bedeutet, das Kommando **ls –laRF** ist identisch mit **ls –l –a –R –F**.

6. Das erste Argument nach einer Option muß von dieser durch Leerraum (Space, Tab oder Newline) getrennt sein. Das bedeutet, korrekt muß es **cc –o prog prog.c** statt **cc –oprog prog.c** heißen.

7. Argumente von Optionen können *nicht* optional sein. Stattdessen wäre in diesem Fall die ganze Option wegzulassen. Das bedeutet, wenn der Befehl **mt –f /dev/rmt0** ein gültiges Kommando wäre, kann **mt –f** kein gültiges Kommando sein.

8. Wenn eine Option mehrere Argumente hat, müssen diese durch Komma oder Leerräume getrennt sein. Im letzteren Fall müssen die Argumente alle zusammen durch Anführungszeichen zusammengruppiert sein. Ein Kommando sollte beide Schreibweisen (mit Komma oder mit Leerräumen) akzeptieren.

9. Alle Optionen müssen allen anderen Angaben einer Kommandozeile voraus gehen. Während in **ls –l datei** –l eine Option und datei eine andere Angabe ist, sind in **ls datei –l** beide Angaben andere Angaben und –l wird nicht als Option erkannt.

10. Die Zeichenfolge – – kann benutzt werden, um das Ende der Optionsliste anzuzeigen. Während in **rm –r datei** –r eine Option und datei eine andere Angabe ist, sind in **rm – – –r datei** beide Angaben andere Angaben.

11. Die Reihenfolge der Optionen untereinander sollte keine Reihenfolge spielen. Die Angabe **ls -lR** soll also äquivalent zu **ls -Rl** sein.

12. Die Reihenfolge sonstiger Parameter darf eine Rolle spielen und es steht dem jeweiligen Kommando frei, wie es solche Fälle handhabt.

13. Das Zeichen –, alleinstehend und auf beiden Seiten begrenzt durch Leerräume sollte nur benutzt werden, um die Standardeingabe zu bezeichnen.

Nicht alle UNIX–Befehle benutzen jedoch `getopt` und so gibt es einige Kommandos, die ihre Optionen nicht konform zu allen diesen Regeln handhaben. Einige Befehle, wie etwa das **dd**–Kommando haben gar eine Optionsschreibweise, die von diesen Regeln komplett abweicht.

1.2 Ein– und Ausgabeumlenkungen

Praktisch alle UNIX–Kommandos arbeiten mit drei Standarddatenströmen. Sie lesen ihre Eingabe aus dem sogenannten *Standardeingabekanal*, bearbeiten diese Eingabe in irgendeiner Form und generieren eine Ausgabe auf dem *Standardausgabekanal*. Kommt es bei dieser Verarbeitung zu Problemen, so werden diese Fehler auf dem *Standardfehlerkanal* gemeldet.

Diese Standarddatenkanäle werden in UNIX in der Regel von der Programmiersprache und ihrer Laufzeitumgebung zur Verfügung gestellt. Die unter UNIX am häufigsten verwendete Programmiersprache ist C, und in ihr heißen diese Kanäle *stdin*, *stdout* und *stderr*. Diese Bezeichnungen werden dann teilweise auch im täglichen Sprachgebrauch für die entsprechenden Kanäle verwendet.

Standardmäßig ist die Standardeingabe der Tastatur des Terminals zugeordnet, und Standardausgabe und Standardfehlerkanal zeigen auf den Bildschirm des Terminals. Es gibt in UNIX eine Gerätedatei **/dev/tty**, die immer genau für das aktuelle Terminal eines Prozesses steht.

Wenn man in einer UNIX–Shell ein Kommando eingibt, so zerlegt die Shell dieses Kommando in Worte. Normalerweise ist ein *Wort* dabei als Block von Zeichen ohne Leerzeichen definiert; Leerzeichen und Tabulatoren sowie Zeilenenden dienen als Worttrenner. Das erste Wort eines Kommandos ist der *Kommandoname*, die folgenden Worte sind die *Parameter* dieses Kommandos.

Einige Worte (nämlich zum Beispiel diejenigen, die Ein– und Ausgabeumleitungen darstellen) werden von der Shell zurückgehalten und nicht an das Kommando weitergegeben. Sie bestimmen nicht das Verhalten des Programmes, sondern die Art und Weise, wie die Shell dieses Kommando aufruft.

Beispiel:

```
Eingabe:
$ bla fasel >file <data sabbel

gestartetes Kommando:
bla

Parameter:
fasel
sabbel

von der Shell ausgefiltert:
>file
<data
```

Abbildung 1.1: Zerlegung eines UNIX–Kommandos in seine Komponenten

1.2.1 Umlenkung der Ausgabe (output redirection)

Normalerweise geben Kommandos ihre Ausgabe auf dem Bildschirm des Terminals aus. Mit dem Zeichen > („größer als“) kann man die Ausgabe des Kommandos aber auch in eine Datei umleiten. Der Name der Datei wird dabei an das Zeichen angehängt [1].

Existiert die Datei noch nicht, so wird sie angelegt. Die neue Datei hat dann den Eigentümer des erzeugenden Prozesses als Eigentümer und die Standardzugriffsrechte. Damit das Anlegen der Datei funktioniert, müssen die entsprechenden Rechte am Verzeichnis, in dem die Datei angelegt werden soll, gegeben sein.

Existiert die Datei schon, so wird sie zunächst geleert (auf 0 Byte gekürzt) und dann beschrieben. Der alte Eigentümer und die alten Zugriffsrechte bleiben erhalten. Damit das Überschreiben funktioniert, muß Schreibrecht (w–Recht) an der Datei gegeben sein.

Beispiel:

```
$ pwd >mitschnitt
$ cat mitschnitt
/u/home/kris/Texte/TeX/Afbb
```

Abbildung 1.2: Eine einfache Ausgabeumlenkung

[1] UNIX erlaubt hier auch, eine Leerstelle zwischen dem Umleitungszeichen und dem Dateinamen zu lassen.

Wenn man verhindern möchte, daß der alte Inhalt der Datei überschrieben wird, so kann man den neuen Text an das Ende der alten Datei anhängen lassen. Dazu gibt man statt eines einfachen **>datei** die Folge **>>datei** an. Existiert die Datei noch nicht, so wird sie neu angelegt.

Beispiel:

```
$ echo Hallo >> mitschnitt2
$ echo Du da >> mitschnitt2
$ cat mitschnitt2
Hallo
Du da
```

Abbildung 1.3: Anhängen an eine Datei

1.2.2 Umlenkung der Eingabe (input redirection)

Analog zur Umlenkung der Ausgabe läßt sich auf die Eingabe eines Kommandos umlenken. Durch die Angabe **<datei** kann man der Shell mitteilen, daß ein Kommando seine Daten nicht vom Terminal einlesen soll, sondern statt dessen die bezeichnete Datei als Eingabe vorgesetzt bekommt.

Beispiel:

```
$ cat mitschnitt2
Hallo
Du da
$ wc < mitschnitt2
       2       3      12
```

Abbildung 1.4: Umleiten der Eingabe

Die Datei mitschnitt2 besteht also aus 2 Zeilen, 3 Worten oder 12 Zeichen. Für viele Programme ist die Standardeingabe oder Standardausgabe die Voreinstellung, wenn keine anderslautenden Parameter angegeben sind. Andere Programme kann man durch die Angabe eines speziellen Dateinamens dazu zwingen, ihre Ausgabe auf die Standardausgabe zu schreiben bzw. von der Standardeingabe zu lesen. Oftmals wird der besondere Dateiname **–** („minus“) für diesen Zweck benutzt.

Beispiel:

```
$ tar -cf /dev/tape a b c
$ tar -cf - a b c >archiv.t
5120
```

Abbildung 1.5: Umleiten der Eingabe

Der erste Befehl archiviert die Dateien mit den Namen a, b und c und schreibt das Archiv in die Gerätedatei /dev/tape. Der zweite Befehl archiviert dieselben Dateien und zwingt das tar–Kommando durch die Angabe des Dateinamens –, das Archiv auf der Standardausgabe auszugeben. Diese ist in die Datei archiv.t umgeleitet.

1.2.3 Hier–Dokumente (here documents)

Manchmal möchte man Kommandos oder Texte für einen UNIX–Befehl in eine laufende Befehlsfolge einbetten, ohne dafür eine eigenständige Datei anzulegen. UNIX sieht dafür den Mechanismus der sogenannten *Hier–Dokumente* vor. Man erklärt diesen am besten an einem Beispiel:

Beispiel:

```
$ cat > datei <<TEXTENDE
> Dies ist
> ein langer Text
> der die Eingabe fuer cat darstellt.
> TEXTENDE
$ cat datei
Dies ist
ein langer Text
der die Eingabe fuer cat darstellt.
$
```

Abbildung 1.6: Ein Hier–Dokument

Im Beispiel wurde der Text zwischen dem Kommando und der Zeile mit dem Wort „TEXTENDE“ gewissermaßen ausgeschnitten und dem cat–Befehl als Eingabe bereit gestellt.

Ein Hier–Dokument wird mit **<<Endemarke** eingeleitet. Alle folgenden Zeilen bis zu einer Zeile, die genau nur das als Endemarke bezeichnete Wort enthält, werden zu einer Textdatei ohne Namen zusammengefaßt, die dem Kommando als Eingabe bereitgestellt wird.

Auf diese Weise ist es möglich, konstante Daten für Kommandos in eine laufende Kommandofolge einzubetten.

1.2.4 Umlenkung von Fehlermeldungen (error redirection)

Intern ordnet UNIX jedem Datenkanal eine Kanalnummer, den sogenannten *Filedescriptor*, zu. Diese Nummer ist eine kleine, positive Zahl, die für jede Datei, die ein Programm geöffnet hält, eindeutig ist. Anhand dieser Zahl identifiziert UNIX eine Datei. Per Konvention ist für jedes Programm der Datenkanal mit der Nummer 0 als Standardeingabe geöffnet, der Datenkanal 1 dient als Standardausgabe und der Datenkanal 2 als Standardfehlerausgabe.

Man kann jedem der Umleitungskommandos <, > und >> eine Nummer voranstellen, die die Kanalnummer des umzuleitenden Kanals bezeichnet. Dementsprechend läßt sich die Fehlerausgabe eines Kommandos mit der Sequenz **2>datei** in die Datei mit dem angegebenen Namen umleiten.

Beispiel:

```
$ cat diesedateigibtesnicht
diesedateigibtesnicht: No such file or directory
$ cat diesedateigibtesnicht 2>/dev/null
$
```

Abbildung 1.7: Umleitung der Fehlerausgabe

Im Beispiel wird die Fehlermeldung des cat–Kommandos an das Gerät /dev/null (den Datenmülleimer) weiter geleitet.

1.2.5 Pipelines (pipelines)

Häufig möchte man, daß die Ausgabe eines Kommandos zur Eingabe des nächstfolgenden Kommandos wird. Man will also, daß sich verschiedene Kommandos Hand in Hand arbeiten, daß man sie zusammenstecken kann wie Legosteine. Einer der Mechanismen, den UNIX dafür vorsieht, ist die Pipeline (pipeline, nameless pipe).

Die Konstruktion **kommando1 | kommando2** sorgt dafür, daß die Standardausgabe von kommando1 zur Standardeingabe von kommando2 wird.

Beispiel:

```
$ tar -cf - a b c | compress > archiv.t.Z
```

Abbildung 1.8: Eine einfache Pipeline

Im Beispiel wird der tar–Befehl durch den Dateinamen – gezwungen, sein Archiv nicht in eine Datei zu schreiben, sondern auf der Standardausgabe auszugeben. Diese Ausgabe wird durch das Pipelining zur Eingabe des compress–Kommandos. Der compress–Befehl bearbeitet die betreffenden Dateien und schreibt seine Standardausgabe (die komprimierte Datei) in die Datei mit dem Namen archiv.t.Z.

Im Gegensatz zu MS–DOS wird dabei **keine Zwischendatei** angelegt, sondern beide Kommandos laufen zeitgleich und arbeiten Hand in Hand. Unter UNIX belegen Pipelines dementsprechend auch keinen Plattenplatz für Zwischendateien.

MS–DOS ist nicht in der Lage, zwei Programme zeitgleich auszuführen. Deswegen können diese Programme auch nicht untereinander kommunizieren, sondern benötigen eine Zwischendatei. Unter MS–DOS wird eine Pipeline also nachgebildet, indem im aktuellen Verzeichnis vom ersten Kommando eine Datei mit der Endung $$$ angelegt wird, die dann die Eingabe des zweiten Kommando darstellt. Dazu muß man das aktuelle Verzeichnis beschreiben können, und auf dem entsprechenden Laufwerk muß genügend Platz für eine solche Zwischendatei vorhanden sein.

Wenn man in einem Netzwerk arbeitet, kann es vorkommen, daß man im aktuellen Verzeichnis keine Dateien anlegen darf. In diesem Falle würden DOS–Pipelines nicht funktionieren. In der Tat waren alte Versionen von MS–DOS in dieser Situation in großen Schwierigkeiten. Neue Versionen von MS–DOS kennen die Umgebungsvariable TMP, die ein Verzeichnis mit Schreibrecht bezeichnet. Fehlt das Schreibrecht im aktuellen Verzeichnis, so legt MS–DOS die Zwischendatei im TMP–Verzeichnis an.

Da unter UNIX beide an einer Pipeline beteiligten Programme zeitgleich ablaufen können und direkt miteinander kommunizieren arbeiten, braucht keine Zwischendatei angelegt zu werden, und das Problem tritt hier gar nicht erst auf.

1.3 Dateinamenexpansion

Wie im vorhergehenden Abschnitt bereits angedeutet, führt die Shell Kommandos, die der Benutzer eingibt, nicht unbedingt wörtlich aus, sondern nimmt bestimmte Modifikationen an der Kommandozeile vor. Worte, die Umlenkungen von Eingabe oder Ausgabe spezifizieren, werden zum Beispiel ausgefiltert und von der Shell vor dem Start des Kommandos bearbeitet.

Ebenso erweitert die Shell Worte, die Jokerzeichen (engl. *Wildcards*) für Dateinamen enthalten, zu einer Folge von Worten, auf die das Jokerzeichen zutrifft. Die Shell kennt die folgenden Jokerzeichen:

Zeichen	Bedeutung
*	eine beliebig lange Folge von beliebigen Zeichen
?	ein einzelnes beliebiges Zeichen
[...]	eines der Zeichen, die in der Klammer vorkommen
[!...]	ein beliebiges Zeichen, ausgenommen die in der Klammer

Abbildung 1.9: Liste der Jokerzeichen in der Shell

1.3.1 Der Joker *

Das Zeichen ***** steht fuer eine beliebig lange Folge von Zeichen aus beliebigen Zeichen. Die beliebige Länge schließt dabei auch die Länge Null mit ein, wie man an dem Beispiel im vorherigen Abschnitt erkennen kann.

Die Shell geht das aktuelle Verzeichnis durch und ersetzt das Wort mit dem Jokerzeichen durch alle Dateinamen, die auf das Wort mit dem Jokerzeichen passen. Das bedeutet, das Wort mit dem Jokerzeichen wird aus dem Kommando entfernt, und an seiner Stelle wird eine Liste von Worten (nämlich die passenden Dateinamen) eingefügt. Finden sich keine Dateinamen, die auf das Wort mit dem Jokerzeichen passen, so wird das Wort mit dem Jokerzeichen selber wieder eingesetzt.

So wird das Wort y* durch alle Dateinamen im aktuellen Verzeichnis ersetzt, die mit einem y beginnen. Enthält das aktuelle Verzeichnis keine Dateien, deren Name mit y beginnt, so wird das y* wieder eingesetzt. Im Beispiel des vorherigen Abschnittes wird das y* durch die Dateinamen ytest und yytest ersetzt.

Das Beispiel 1.10 sollte dieses Verhalten klarer machen.

Beispiel:

```
Vorhandene Dateien:
testa testaa testb testbb testab testac
testy ytest yytest murfc blarfc

Eingegebenes Kommando:  rm y*
Gestrichen:             y*
Eingesetzt:             ytest yytest
Ausgefuehrtes Kommando: rm ytest yytest

Eingebenes Kommando:    rm test*b
Gestrichen:             test*b
Eingesetzt:             testb testbb testab
Ausgefuehrtes Kommando: rm testb testbb testab
```

Abbildung 1.10: Dateinamenerweiterung durch Jokerzeichen

Der Stern darf dabei auch mitten im Wort oder gar mehrfach in einem Wort vorkommen. So wird das Wort test*b durch alle Dateinamen im aktuellen Verzeichnis ersetzt, die mit dem Teilwort test beginnen und dem Teilwort b enden. Dazwischen dürfen beliebig viele (auch null) beliebige Zeichen stehen. Im Beispiel wird das Wort test*b durch die Dateinamen testb und testbb sowie testab ersetzt.

In MS–DOS verhält sich der Stern wesentlich anders als in UNIX. In MS–DOS bewirkt ein Stern, daß das Suchmuster bis zum Ende des Dateinamens oder Extenders mit Fragezeichen aufgefüllt wird. Dann vergleicht MS–DOS das resultierende Suchmuster (das keinen Stern mehr enthält) mit dem Dateinamen). Der Ausdruck a*b.exe steht deswegen nicht wie man erwarten sollte für eine Datei, deren Name mit a beginnt und mit b.exe endet, sondern fuer das Suchmuster a???????.exe.

Der Stern ist in UNIX als Suchmuster wesentlich flexibler und mächtiger als in MS–DOS.

Achtung: Der Stern als Suchmuster paßt in UNIX niemals auf Dateien, deren Name mit einem Punkt beginnt, es sei denn, dies ist ausdrücklich angegeben. Dadurch wird verhindert, daß man aus Versehen Punktdateien löscht.

Beispiel:

```
Loeschen von allen Dateien ausgenommen Punktdateien:
$ rm *

Loeschen der meisten Punktdateien:
$ rm .??*
```

Abbildung 1.11: Punktdateien werden gesondert behandelt

1.3.2 Der Joker ?

Das Fragezeichen in einem Wort steht fuer ein einzelnes, beliebiges Zeichen. Mit unseren Beispieldateinamen würde das Wort test? also durch die Worte testa, testb und testc ersetzt werden. Auch beim Fragezeichen gilt wie bei allen Jokerzeichen: Wenn kein Dateiname auf das Wort mit dem Jokerzeichen paßt, wird das Wort mit dem Jokerzeichen wieder eingesetzt.

Man kann die Ersetzung von Worten durch Dateinamen sichtbar machen, indem man etwa ein Kommando wie echo *c eingibt.

Beispiel:

```
$ echo *c
testac murfc blarfc
$
```

Abbildung 1.12: Demonstration von Textersatz

Die Shell schneidet das Wort *c aus der Kommandozeile aus und setzt statt dessen die Namen aller Dateien ein, die auf c enden. Es wird in Wirklichkeit also das Kommando **echo testac murfc blarfc** ausgeführt, wie man an der Ausgabe des Kommandos sehen kann.

Während unter UNIX die Shell, also der Kommandointerpreter, für die Erweiterung von Jokerzeichen zu Dateinamen zuständig ist und das Kommando Dateinamenjoker normalerweise niemals zu sehen bekommt, verhält sich MS–DOS genau anders herum.

Unter DOS kümmert sich der Kommandointerpreter niemals um die Erweiterung von Dateinamen. Stattdessen ist jedes Kommando selbst dafür verantwortlich, Jokerzeichen zu akzeptieren und zu passenden Dateinamen zu expandieren. Bei einigen Kommandos (etwa beim TYPE–Kommando) fehlt diese Dateinamenerweiterung. Diese Kommandos nehmen dann keine Joker an, obwohl dies sehr nützlich und benutzerfreundlich wäre.

Theoretisch ist es für einen MS–DOS Befehl auch möglich, seine eigene Dateinamenerweiterung zu benutzen und etwa andere Zeichen als die normalen DOS–Joker * und ? zu verwenden.

1.3.3 Die Joker [...] und [!...]

Mit der Konstruktion [...] ist es möglich, eine Menge von Zeichen (character class) aufzuzählen. Die gesamte Klammer steht dann für ein einzelnes Zeichen. Anders als beim

Fragezeichen ist aber nicht jedes beliebige Zeichen zugelassen, sondern nur eines der Zeichen, die in der Klammer aufgeführt sind. So kann man zum Beispiel mit **cat test[abc]** alle Dateien ausgeben, deren Name mit test beginnt und dann entweder mit a, b oder mit c endet. Die Datei testy wird von diesem Suchmuster nicht erfaßt.

Möchte man eine große Zahl von Zeichen aufführen, so kann es ziemlich mühsam sein, diese alle einzeln aufzulisten. UNIX erlaubt es statt dessen, einen Bereich von Zeichen in der Form [0–9] anzugeben. Letzterer Ausdruck entspricht der Angabe [0123456789]. Um also alle Dateien anzeigen zu lassen, deren Name mit einem Punkt und einer Ziffer endet, kann man das Kommando **ls –l *.[0–9]** geben.

Manchmal ist es leichter, nicht die Zeichen anzugeben, die zugelassen sein sollen, sondern statt dessen diejenigen Zeichen aufzuführen, die nicht vorkommen dürfen. Dafür bietet die UNIX–Shell die Konstruktion [!...] an. Sie steht für ein einzelnes, beliebiges Zeichen, ausgenommen die in der Klammer aufgeführten Zeichen. Auch hier können wieder Bereiche angegeben werden. Das Kommando **ls –l *.[!0–9]** listet also alle Dateien auf, deren Name mit einem Punkt und einem Zeichen endet, das keiner der Ziffern von 0 bis 9 entspricht.

1.4 Variablenexpansion

Neben der Ersetzung von Worten, die Jokerzeichen enthalten, durch Dateinamen gibt es noch einen anderen Wortersetzungmechanismus in der Shell. Dieser reagiert auf Worte, die mit einem **$** (Dollarzeichen) beginnen, und ersetzt Variablennamen durch ihren Wert.

1.4.1 Setzen und Löschen von Variablenwerten

Die UNIX–Shell erlaubt es, Variablen mit Werten zu belegen. Diese Werte können entweder in der Shell selber benutzt werden oder von UNIX–Kommandos aus abgefragt werden. Obwohl der Variablenmechanismus der Shell dem Variablenkonzept von anderen Programmiersprachen recht ähnlich ist, weist er doch einige Besonderheiten auf. So sind zum Beispiel alle Variablen in der Shell Zeichenketten (strings). Nur in Ausnahmefällen werden diese Zeichenketten als Zahlen oder logische Werte interpretiert.

Einer Shellvariablen wird mit einer einfachen Zuweisung ein Wert zugewiesen.

Beispiel:

```
$ name=wert
$ diesisteinlangername=diesisteinlangerwert
```

Abbildung 1.13: Zuweisung von Werten an Variablen

Dabei ist es wichtig, die Zuweisung selbst *ohne Leerzeichen* vor und hinter dem Gleichzeichen zu schreiben. Dies ist eine echte Ausnahme: Während in der Shell sonst überall Leerzeichen als Worttrenner notwendig sind, muß eine Zuweisung von der Shell als ein Wort gelesen werden.

Mit dem Shellkommando **set** ohne weitere Parameter kann man abfragen, welche Variablen zur Zeit in der Shell gesetzt sind und welche Werte diese haben. Bei einer großen Anzahl von Variablen kann es sinnvoll sein, die Ausgabe von set durch einen Pager (etwa more) oder durch einen Suchbefehl (etwa grep) zu leiten.

Wie fast überall in UNIX sollte man auf die Groß– und Kleinschreibung der Namen achten. Auch bei Variablennamen unterscheidet UNIX die Schreibweise.

Beispiel:

```
$ set | more
[ Ausgabe geloescht ]
$ set | grep HOME
HOME=/Benutzer/kris
```

Abbildung 1.14: Abfrage von Variablenwerten

1.4.2 Gebrauch von Variablen

In der Shell können Variablen an jeder Stelle verwendet werden. Immer wenn die Shell bei der Analyse einer Kommandozeile ein Wort findet, das mit dem Zeichen **$** (Dollar) beginnt, so schneidet sie das betreffende Wort aus und ersetzt es durch den Wert der entsprechenden Variablen. Wenn die benannte Variable nicht definiert ist, wird nichts eingesetzt.

Beispiel:

```
Kommando:    echo a $HOME b
Gestrichen:  $HOME
Eingesetzt:  /Benutzer/kris
Ausgefuehrt: echo a /Benutzer/kris b

Kommando:    echo a $existiertnicht b
Gestrichen:  $existiertnicht
Eingesetzt:  <nichts>
Ausgefuehrt: echo a b
```

Abbildung 1.15: Wortersetzung bei Variablen

In Variablen kann man so Kurzschreibweisen für häufig benutzte Verzeichnisse oder Kommandos speichern oder Konfigurationsinformationen für Befehle hinterlegen. Viele UNIX–Befehle fragen die Werte von bestimmten Variablen ab und verhalten sich je nach Wert, der in der Variablen hinterlegt ist, unterschiedlich. Man kann so das Verhalten der eigenen Arbeitsumgebung in gewissen Grenzen konfigurieren.

Eine Variable wird mit dem Kommando **unset** gelöscht. Weist man der Variablen dagegen nichts zu (also den leeren String), so ist der Wert der Variablen zwar leer, aber die Variable selbst ist noch definiert. Wir werden später sehen, daß dies einen Unterschied machen kann.

Beispiel:

```
$ VAR=WERT
$ set | grep VAR
VAR=WERT
$ VAR=
$ set | grep VAR
VAR=
$ unset VAR
$ set | grep VAR
$
```

Abbildung 1.16: Unterschied zwischen leeren und gelöschten Variablen

1.4.3 Exportieren und Schützen von Variablen

Damit ein UNIX–Kommando eine Variable abfragen kann, muß diese Variable zunächst einmal exportiert werden. Per Voreinstellung sind nämlich die Werte aller Variablen lokal zu der aktuellen Kommandoshell. Nur diejenigen Variablen, die für den Export markiert sind, werden an Untershells und von der aktuellen Shell gestartete Kommandos weitervererbt.

Man markiert Variablen mit dem Befehl **export <varname>** für den Export. Derartig markierte Variablen werden an alle Unterprozesse weitervererbt und können dort abgefragt werden. Mit dem Kommando **env** oder (je nach UNIX–Version) **printenv** kann man eine Liste der für den Export markierten Variablen und ihrer Werte bekommen.

Beispiel:

```
$ BLUE=daddeldu
$ export BLUE
```

Abbildung 1.17: Definition und Export einer Variablen

Eine exportierte Variable wird mit dem Kommando **unset** nicht nur gelöscht, sondern ist bei erneuter Zuweisung automatisch wieder eine lokale Variable (also nicht exportiert).

Mit dem Schlüsselwort **readonly** kann man Variablen in Konstanten verwandeln. Der Wert einer solchen Variablen kann dann nicht mehr geändert werden. **readonly** ohne Parameter zeigt eine Liste der Konstanten an.

1.4.4 Besondere Variablen

Einige Variablen haben in UNIX per Konvention eine besondere Bedeutung. Manche dieser Variablen sind in die Shell eingebaut, d.h. sie verändern ihren Wert automatisch, andere haben nur deswegen eine besondere Bedeutung, weil sie von gängigen UNIX–Kommandos abgefragt werden.

Vordefinierte Variablen mit fester Bedeutung sind zum Beispiel die folgenden Variablen:

$0 Die Variable $0 enthält den Namen des aufgerufenen Befehls, also bei einem Shellscript den Namen der Scriptdatei.

$1 bis $9 Die Variablen $1 bis $9 enthalten die Parameter 1 bis 9 des Aufrufes. Wurde das Kommando mit mehr als 9 Parametern aufgerufen, so sind die weiteren Parameter nicht direkt ansprechbar. Sie können aber mit dem Shell–Kommando **shift** in den Zugriff gebracht werden.

Die Variablen $1 bis $9 in der UNIX–Shell finden sich als die Variablen %1 bis %9 direkt und mit derselben Bedeutung in MS–DOS wieder. Auch in MS–DOS ist es möglich, mit einem **shift** auf weitere Parameter zuzugreifen.

$* Manchmal möchte man jedoch auf alle Parameter der Kommandozeile *als ein zusammenhängendes Wort* zugreifen können. Dies ist mit der Variablen $* möglich. Sie entspricht also der Konstruktion ”$1 $2 $3 ... ”. $* entspricht auch dann allen Parametern der Kommandozeile, wenn dies mehr als 9 Stück sind.

$@ Seltener möchte man alle Parameter der Kommandozeile *als eine Folge von einzelnen Worten* im Zugriff haben. Die Konstruktion $@ entspricht ”$1”, ”$2”, ”$3”, ... Beachten Sie den Unterschied zu $*!

$# Die Shell vermerkt in der Variablen $# die Anzahl der Parameter eines Kommandos.

$– In der speziellen Variablen $– speichert die Shell die derzeit eingeschalteten Optionsbuchstaben.

$? Jedes UNIX–Kommando hinterläßt seinem Aufrufer bei seiner Beendigung eine Zahl, den sogenannten Exit–Status. Sie kann dem Aufrufer Auskunft darüber geben, ob das Kommando normal beendet worden ist oder ob bei seiner Bearbeitung ein Problem auftrat. Per Konvention bedeutet ein Exit–Status von 0, daß das Kommando fehlerfrei abgearbeitet worden ist; jeder andere Exit–Status bedeutet eine in irgendeiner Form fehlerhafte Abarbeitung des Befehls. Welche Bedeutung ein Fehlercode genau hat, hängt natürlich vom Befehl ab.

In der Shell–Variablen $? kann man den Endestatus des zuletzt ausgeführten Befehls abfragen.

$$ In einer UNIX–Shell wird jedes Kommando als ein eigener Prozeß ausgeführt. Jeder Prozeß ist dabei durch seine Prozeßnummer von allen anderen Prozessen zu unterscheiden; Prozeßnummern sind also zu einem gegebenen Zeitpunkt eindeutig.

Die Prozeßnummer der eigenen Shell steht in der Variablen $$ zur Verfügung. Diese Variable wird oft verwendet, wenn man eindeutige Dateinamen für Zwischendateien erzeugen möchte.

Beispiel:

```
#! /bin/sh --
WRONG_TMPNAME=/tmp/datei
RIGHT_TMPNAME=/tmp/$0.$$

echo Kollision > $WRONG_TMPNAME
echo Keine Kollision > $RIGHT_TMPNAME

exit
```

Abbildung 1.18: Geschickte und ungeschickte Verfahren zur Erzeugung von Zwischendateinamen

Wenn das Script im Beispiel 1.18 zum selben Zeitpunkt von mehr als einem Benutzer aufgerufen wird, verwenden alle Instanzen des Scriptes denselben Dateinamen /tmp/datei. Würde man diesen festen Dateinamen zur Erzeugung von Zwischendateien verwenden, so würden alle Instanzen des Scriptes dieselbe Zwischendatei verwenden und sie so gegenseitig überschreiben. Definiert man den Namen der Zwischendatei dagegen als $0.$$, so wird für $0 der Name des Scriptes eingesetzt und für $$ die aktuelle Nummer dieses Prozesses. Da jede Instanz eines Scriptes eine eigene Prozeßnummer hat, würde jede dieser Instanzen also auch einen anderen Zwischendateinamen verwenden (etwa probe.1704 und probe.1706).

$! Unter diesem Namen ist die Prozeßnummer des letzten erzeugten Hintergrundprozesses verfügbar.

Andere Variablen haben deswegen eine spezielle Bedeutung, weil viele Programme sie abfragen oder sie schon beim Login eines Benutzers automatisch mit Werten belegt werden.

$EDITOR Viele UNIX–Programme verwenden einen Editor, wenn der Benutzer größere Mengen Text eingeben muß. Ist die Variable $EDITOR nicht gesetzt, wird der Standardeditor **vi** verwendet. Da die meisten Menschen den Editor vi nicht verwenden mögen, geben sie in der Variablen $EDITOR ihren Editor mit vollem Pfadnamen an. Die meisten Programme verwenden dann diesen Editor anstelle des vi.

$HOME In der Variablen $HOME ist der Name des Home–Verzeichnisses des aktuellen Benutzers gespeichert. Wenn man das Kommando **cd** ohne Parameter verwendet, wechselt es in dieses Verzeichnis. Praktisch alle UNIX–Kommandos erwarten Konfigurationsdateien als Punkt–Dateien in diesem Verzeichnis, wenn sie überhaupt Konfigurationsdateien verwenden.

$IFS Wie ganz am Anfang bemerkt, teilt die Shell eine Kommandozeile an Leerzeichen, Tabulatorzeichen und Returnzeichen in Worte ein, die nachher Kommandos, Parameter, Variablen und so weiter bilden. Welche Zeichen jetzt Trennzeichen zwischen Worten darstellen, ist nicht fest eingestellt, sondern wird anhand der Variablen $IFS entschieden. Die Variable enthält eine Liste von Zeichen, die Trennzeichen zwischen Worten darstellen. Voreingestellt enthält diese Liste genau die Zeichen *Leerzeichen*, *Tabulator* und *Zeilenende*.

$MAIL,

$MAILCHECK,

$MAILMSG UNIX überprüft alle $MAILCHECK Sekunden, ob neue Nachrichten in der durch $MAIL bezeichneten Datei angekommen sind. Falls dies der Fall ist, wird der Text $MAILMSG ausgegeben. $MAIL sollte normalerweise den vollen Pfadnamen der eigenen Maildatei enthalten, also üblicherweise /usr/spool/mail/<username> oder /usr/mail/<username>, je nachdem, wie das lokal geregelt ist. $MAILCHECK steht standardmäßig auf einem Wert von 600 Sekunden (10 Minuten), was normalerweise auch ausreichend ist. Setzt man diesen Wert auf 0, so wird vor jeder Kommandoausführung in der Shell (also vor jedem Prompt) eine Überprüfung vorgenommen. Die voreingestellte Nachricht ist „You have mail.“.

$PAGER Falls die Umgebungsvariable $PAGER gesetzt ist und den vollen Pfadnamen eines Textanzeigeprogrammes wie **pg**, **more** oder **less** enthält, verwenden einige UNIX–Kommandos (zum Beispiel der **man**–Befehl) diesen Anzeiger, um größere Mengen an Text anzuzeigen. Ist die Variable nicht gesetzt, so wird **cat** verwendet, und der Text rauscht ohne Pause durch.

$PATH Wenn man in der UNIX–Shell ein Kommando eingibt, so wird es nacheinander in den in dieser Variablen aufgeführten Verzeichnissen gesucht. Üblicherweise enthält diese Variable eine Liste von Verzeichnissen. Die Namen der Verzeichnisse sind dabei durch jeweils einen Doppelpunkt voneinander getrennt und werden genau in der aufgeführten Reihenfolge durchsucht.

Diese Variable scheint auf den ersten Blick genau dem Pfad von MS–DOS zu entsprechend. Jedoch gibt es einen wichtigen Unterschied: Bei MS–DOS ist das aktuelle Verzeichnis immer automatisch das erste Verzeichnis im Suchpfad. Unter UNIX dagegen **muß** das aktuelle Verzeichnis ausdrücklich mit dem Namen ”.” im Suchpfad aufgeführt sein, wenn es auch für Kommandos durchsucht werden soll. Es wird dann auch genau an der angegebenen Position im Suchpfad durchsucht und nicht wie bei MS–DOS immer als erstes Verzeichnis.

Für den UNIX–Systemverwalter ist es aus Sicherheitsgründen eine gute Idee, das aktuelle Verzeichnis überhaupt nicht im Suchpfad zu haben, oder, wenn doch, dann wenigstens als allerletztes Verzeichnis im Pfad.

$PS1 Die Abkürzung PS1 steht für **P**rompt **S**tring **1**. Der *Prompt* ist das Bereitschaftszeichen der Shell. Standardmäßig ist es $ für normale Benutzer und # für den Systemverwalter. Viele Systemverwalter setzen sich den aktuellen Pfad, ihren Usernamen und bei Netzwerken auch noch den Namen der Maschine in den Prompt, damit sie auf den ersten Blick erkennen, unter welcher Benutzer–ID und auf welcher Maschine sie gerade arbeiten.

$PS2 Wenn eine Fortsetzungszeile angefordert wird, verwendet die Shell den in der Variablen PS2 definierten Promptstring. Standardmäßig steht dieser auf >.

$TERM Bildschirmorientierte Programme müssen wissen, wie sie ein Terminal dazu bringen können, den Bildschirm zu löschen oder den Cursor zu bewegen. Leider müssen unterschiedliche Terminaltypen auf unterschiedliche Weise angesprochen werden. In der Variablen $TERM ist der Name des aktuellen Terminals gespeichert. Programme können diesen Namen abfragen und in der *terminal capabilities database*, kurz *termcap*, nachschlagen, wie dieser Terminaltyp nun genau angesteuert wird.

$TZ Ein UNIX–Rechner hat nur eine interne Uhrzeit und diese wird normalerweise in UT (*universal time*) gespeichert, um eine für alle Benutzer einheitliche Zeitbasis zu haben. Der Benutzer möchte Zeitangaben aber natürlich in der für ihn gültigen und gewohnten Zeitzone haben. Die Angabe in der $TZ–Variablen kann ziemlich kompliziert werden, falls eine Sommerzeit im Spiel ist und man die Umschaltung zwischen Sommerzeit und Winterzeit automatisieren möchte. Im einfachsten Fall enthält die Variable nur den Namen der lokalen Zeitzone und um wieviele Stunden diese Zeitzone von UT abweicht.

In Mitteleuropa gibt man als Zeitzone **CET–1** an. Beachten Sie, daß **c**entral **e**uropean **t**ime um **plus** eine Stunde von der UT abweicht, aber **minus** eine Stunde angegeben werden muß. Das liegt daran, daß UNIX trotz aller Internationalität ein amerikanisches Produkt ist und die eigenen (amerikanischen) Zeitzonen selbstverständlich niemals negativ sein können.

$LOGNAME,

$USER Je nach lokaler Konvention befindet sich der Benutzername, unter dem man sich in das System eingeloggt hat, in der Variablen $LOGNAME oder $USER.

1.4.5 Tricks mit geschweiften Klammern

Manchmal hat man beim Einsatz von Variablen Probleme, zu unterscheiden, wo der Variablenname aufhört und der normale Text des Kommandos wieder beginnt.

Beispiel:

```
$ mit=miteiner
$ echo einstring$mitvariablen
einstring
$ echo einstring${mit}variablen
einstringmiteinervariablen
```

Abbildung 1.19: Wo hört der Variablenname auf?

In diesem Fall *muß* man den Namen der Variablen mit geschweiften Klammern einschließen, damit die Shell den Anfang und das Ende des Variablennamens erkennen kann. Man *darf* diese Einschließung immer machen, aber aus Bequemlichkeit läßt man sie meistens weg.

Wenn eine Variable undefiniert ist, wird sie normalerweise durch nichts ersetzt. Manchmal möchte man das nicht; statt dessen möchte man, daß an Stelle von nichts ein Standardwert eingesetzt wird. Das kann man erreichen, indem man diesen Standardwert in der Form **$**{**varname–standardwert**} mit dem Variablennamen zusammen angibt. Der Variablenname ist danach aber immer noch undefiniert.

Beispiel:

```
$ echo ${nichtdefiniert--machtnix}
machtnix
$ set | grep nichtdefiniert
$ nichtdefiniert=dochdefiniert
$ echo ${nichtdefiniert--machtnix}
dochdefiniert
```

Abbildung 1.20: Vorgabe von Standardwerten bei der Verwendung von Variablen

Verwendet man an Stelle des Minuszeichens ein Gleichheitszeichen, so wird der Standardwert nicht nur ausgegeben, sondern der undefinierten Variablen wird dieser Standardwert in einem Arbeitsgang auch gleich zugewiesen:

Beispiel:

```
$ echo ${nichtdefiniert=jetztaberdoch}
jetztaberdoch
$ set | grep nichtdefiniert
nichtdefiniert=jetztaberdoch
$ echo $nichtdefiniert
jetztaberdoch
```

Abbildung 1.21: Zuweisung von Standardwerten

Wenn kein Standardwert Sinn macht, kann man immerhin noch abfragen, ob eine Variable definiert ist. Ist sie es nicht, wird das laufende Script abgebrochen und mit einer entsprechenden Nachricht beendet.

Beispiel:

```
$ echo ${mussdefiniertsein?ist_es_aber_nicht}
mussdefiniertsein: ist_es_aber_nicht
$ echo ${mussdefiniertsein?}
mussdefiniertsein: parameter null or not set
```

Abbildung 1.22: Abfrage, ob eine Variable überhaupt definiert ist

Und schließlich kann es noch sein, daß man nur testen möchte, ob ein Parameter gesetzt ist.

Im täglichen Leben wird man allenfalls die Möglichkeiten mit dem Minuszeichen und dem Gleichzeichen einmal verwenden. Die beiden anderen Fälle lassen sich mit einem schnellen **if** wesentlich besser lesbar und leichter verständlich erschlagen.

1.5 Quoting

Wir haben jetzt gelernt, daß die Shell eine ganze Reihe von Ersetzungen an unserer Kommandozeile vornimmt, bevor sie überhaupt daran denkt, unser Kommando auszuführen. Meistens ist man für diese Ersetzungen dankbar, weil sie einem eine ganze Menge Tipparbeit ersparen können. Aber es gibt Fälle, wo man nicht möchte, daß die Shell das Kommando oder einen bestimmten Parameter davon anfaßt. In diesem Fall muß man der Shell

Beispiel:

```
$ istgesetzt=${mussgesetztsein+ja}
$ echo $istgesetzt
ja
```

Abbildung 1.23: Abfrage, ob eine Variable überhaupt definiert ist

mitteilen, daß dieser Parameter geschützt ist. Dies geschieht durch das Quoting (*quotes* sind im englischen Anführungszeichen jeder Art).

Zeichen	Name	Bedeutung
”	double quote	Dateinamenersetzung abschalten.
’	tick mark	Jede Ersetzung abschalten.
‘	backtick	Kommando durch seine Ausgabe ersetzen.
\	backslash	Das folgende Zeichen schützen.

Abbildung 1.24: Verzeichnis der Quotezeichen

1.5.1 Doppelte Anführungszeichen (double quotes)

Doppelte Anführungszeichen können verwendet werden, um einen String, der Leerzeichen enthält, als ein Wort zu definieren. Die Shell behandelt den Text zwischen den Anführungszeichen als ein Wort, also ein Kommando oder einen Kommandoparameter, egal ob dieser Leerzeichen, Tabulatoren oder gar Zeilenenden enthält. Dieses Verhalten kann benutzen, um beispielsweise mit dem **echo**–befehl mehr als eine Zeile auszugeben.

Beispiel:

```
$ echo "Dies ist ein langer Text
> der sich ueber mehr als eine Zeile erstreckt."
Dies ist ein langer Text
der sich ueber mehr als eine Zeile erstreckt.
$
```

Abbildung 1.25: Gruppierung durch Anführungszeichen

Beachten Sie, wie die Shell die folgenden Zeilen als Fortsetzungszeilen einliest. Daß dieser String dann auch nur als ein einziger Parameter zählt, kann man leicht zeigen, in dem man sich das folgende kleine Shellscript ansieht.

Innerhalb von doppelten Anführungszeichen ist die Erweiterung von Worten zu Dateinamen deaktiviert. Die Erweiterung von Variablen zu ihren Werten und die Kommandoersetzung (siehe unten) funktionieren jedoch noch wie gewohnt.

1.5.2 Einfache Anführungszeichen (tick marks)

Einfache Anführungszeichen verhalten sich bezüglich der Gruppierung von Worten genau wie die doppelten Anführungszeichen. Allerdings sind innerhalb der einfachen Anführungszeichen alle Textersetzungsmechanismen der Shell ausgeschaltet; bis auf das einfache Anführungszeichen selbst haben alle Zeichen ganz normale Bedeutung.

Beispiel:

```
$ cat paramcount
# /bin/sh --
echo "Mit $# Parametern aufgerufen."
exit
$ ./paramcount a b c
Mit 3 Parametern aufgerufen.
$ ./paramcount "a b c"
Mit 1 Parametern aufgerufen.
$ ./paramcount "a
> b
> c"
Mit 1 Parametern aufgerufen.
```

Abbildung 1.26: Anzahl der Parameter

Beispiel:

```
$ echo test*b $HOME
testb testbb testab /Benutzer/kris
$ echo "test*b" "$HOME"
test*b /Benutzer/kris
```

Abbildung 1.27: Ersetzungen in Anführungszeichen

Beispiel:

```
$ echo '*' '$HOME' ''$HOME''
* $HOME `$HOME`
```

Abbildung 1.28: Keine Ersetzungen in einfachen Anführungszeichen

1.5.3 Rückwärtsstriche (backticks)

Rückwärtsstriche verhindern keine Ersetzungen, sondern stellen selbst auch wieder einen Ersetzungsmechanismus der Shell dar. Es handelt sich um die sogenannte Kommandoersetzung.

Bei dieser wird der Text zwischen den Rückwärtsstrichen als UNIX–Kommando interpretiert und durch die Shell ausgeführt. Die Standardausgabe dieses Kommandos wird dann statt des Textes zwischen den Rückwärtsstrichen in die Kommandozeile eingesetzt.

```
Kommando:          basename /usr/spool/news/log
Ausgabe:           log

Kommando:          cat ‘basename /usr/spool/news/log‘
Gestrichen:        ‘basename /usr/spool/news/log‘
Eingesetzt:        log
Ausgefuehrt:       cat log
```

Abbildung 1.29: Mechanismus der Kommandoersetzung

Auch die Backticks haben, wie die anderen Anführungsstriche, gruppierende Wirkung. Dadurch kann ein Kommando in den Backticks sich über mehr als eine Zeile erstrecken, wenn es notwendig ist.

Beispiel:

```
$ tar -cf /dev/tape ‘find /home
-type f
-mtime -7
-print‘
```

Abbildung 1.30: Kommandoersetzung im Einsatz

Im Beispiel generiert der **find**–Befehl eine Liste aller Dateien unterhalb des Verzeichnisses /home, die innerhalb der letzten 7 Tage verändert worden sind. Normalerweise druckt der Befehl diese Liste auf seiner Standardausgabe aus. Wir möchten jetzt genau diese Dateien, die in der letzten Woche geändert worden sind, auf Band sichern. Normalerweise geschieht dies, indem die Namen dieser Dateien hinter einem **tar –cf /dev/tape**–Kommando aufgelistet werden. Natürlich möchten wir diese Liste nicht von Hand erzeugen, denn sie kann sehr lang sein. Mit den Backticks sorgen wir dafür, daß die Ausgabe des find–Kommandos an genau der richtigen Stelle hinter dem tar–Befehl eingesetzt wird. Der aus der Kommandoersetzung resultierende Befehl, das extrem lang sein kann, sieht dann zum Beispiel so aus:

Beispiel:

```
$ tar -cf /dev/tape /home/kris/.profile /home/kris/.login
/home/kris/logdata /home/kris/Texte /home/kris/Texte/Afbb
/home/kris/Texte/Afbb/redirection.tex
[ ... lange, lange Liste geloescht ... ]
```

Abbildung 1.31: Resultat einer Kommandoersetzung

Viele UNIX–Kommandos machen nur dann Sinn, wenn man sie im Kontext der Kommandoauswertung betrachtet. Meistens sind dies Befehle, die relativ einfache Aufgaben wahrnehmen, indem sie ihre Argumente in irgendeiner Form auswerten und dann etwas drucken. Im Einzelnen:

basename Der Befehl **basename** bekommt als Argument einen Pfadnamen und optional ein Suffix (eine Dateinamenendung):

```
basename pfadname [suffix]
```

Er druckt die letzte Komponente des Pfadnamens bzw. falls ein Suffix angegeben ist, druckt er die letzte Komponente des Pfadnames ohne das angegebene Suffix.

Beispiel:

```
$ basename /home/kris/quelltext.c .c
quelltext
$ datei=‘/home/kris/quelltext.c .c‘
$ echo $datei
quelltext
```

Abbildung 1.32: Beispiel für basename

Der Befehl ist für den interaktiven Gebrauch offensichtlich sinnlos. Mit der Kommandoersetzung durch Backticks und innerhalb eines größeren Shellscriptes macht er aber Sinn.

dirname Das Gegenstück zu basename heißt dirname und nimmt nur einen Pfad als Argument. Im Beispiel von oben würde dirname den Text /home/kris drucken.

date Das date–Kommando druckt normalerweise die aktuelle Zeit in der lokalen Zeitzone. Mit einer Reihe von Formatoptionen kann das Aussehen der Ausgabe beeinflußt werden.

Beispiel:

```
$ date
Wed Oct  6 17:18:48 MET 1993
$ zeit=‘date ’+%H%M’‘
$ if [ "$zeit" -gt "1600" ]
> then
>    echo Feierabend.
> fi
Feierabend.
```

Abbildung 1.33: Beispiel für date

expr Das expr-Kommando nimmt einen arithmetischen Ausdruck, wertet ihn aus und druckt das Resultat. Es kann auch begrenzt benutzt werden, um Teile von Zeichenketten auszuschneiden.

getopt Das Kommando getopt kann eingesetzt werden, um in einem Shellscript die Kommandzeilenparameter auszuwerten. An späterer Stelle wird noch einmal ausführlich auf dieses Kommando eingegangen.

Beispiel:

```
$ expr 1 + 4
5
$ zaehler=17
$ zaehler=‘expr $zaehler + 1‘
$ echo $zaehler
18
```

Abbildung 1.34: Beispiel für expr

Beispiel:

```
$ line
Hallo
Hallo
$ variable=‘line‘
Hallo
$ echo $variable
Hallo
```

Abbildung 1.35: Beispiel für line

line Das Kommando line ist extrem simpel. Es liest eine Zeile von der Standardeingabe und druckt diese auf der Standardausgabe.

Beachten Sie, daß man mit dem in die Shell eingebauten Befehl read dasselbe erreichen kann, ohne ein externes Kommando starten zu müssen.

1.5.4 Schrägstrich rückwärts (backslash)

Wie Ihnen vielleicht schon aufgefallen ist, fallen bei jeder Art von Quoting die Quotes selber fort. Was macht man also, wenn man ein einzelnes Zeichen mit Spezialbedeutung in den laufenden Text einfügen möchte, ohne daß die Ersetzungsmechanismen der Shell greifen? Für diesen Zweck sieht die Shell das Zeichen \ vor, das dem nachfolgenden Zeichen seine Spezialbedeutung nimmt und dafür sorgt, daß es statt dessen wörtlich in den laufen Text eingefügt wird.

Gequotet werden müssen in der Shell die Zeichen ; (Semikolon), & (Kaufmanns–Und, Ampersand), ((linke, runde Klammer, left parenthesis, bra),) (rechte runde Klammer, right parenthesis, ket), | (pipeline, pipe), < (linke spitze Klammer), > (rechte spitze Klammer), Zeilenende, Leerzeichen und Tabulator.

1.6 Reihenfolge der Auswertung

Wir kennen nun eine ganze Reihe von Mechanismen, die auf der Kommandozeile einer UNIX–Shell herumarbeiten und sie verändern. Normalerweise kommen sich diese einzelnen Textersetzungen nicht ins Gehege, weil sie ja auf verschiedene Zeichen reagieren. Manchmal aber muß man aufpassen, und dann ist es wichtig, in welcher Reihenfolge die Shell die einzelnen Ersetzungen ausführt.

- Der Benutzer gibt eine Zeile ein und schließt sie mit Return ab. Die Shell kopiert diese Zeile in einen internen Puffer und beginnt dort, sie zu bearbeiten.

- Als erstes wird die Kommandoersetzung durchgeführt. Dazu werden von links nach rechts alle Textteile gesucht, die zwischen Backticks stehen. Diese Textteile werden jeweils wieder in einen separaten Puffer kopiert und dort wie eine volle, eigenständige Kommandozeile behandelt. Die Ausgabe dieser Kommandos wird aufgefangen und an Stelle des Kommandos in den Puffer zurückkopiert.
- Als nächstes werden alle Variablen gesucht und entsprechend den Ersetzungsregeln für Variablen durch ihre Werte ersetzt.
- Erst jetzt wird die Zeile anhand der als Worttrenner bekannten Zeichen (Inhalt der $IFS–Variablen) in Worte zerlegt.
- Worte, die Jokerzeichen enthalten (und nicht durch Anführungszeichen geschützt sind), werden nun durch passende Dateinamen ersetzt.
- Die resultierende Zeile wird schließlich ausgeführt.

1.7 Hintergrundprozesse und Job–Control

Bis auf ganz wenige eingebaute Kommandos werden alle Shell–Kommandos in einem eigenen Prozeß ausgeführt. Normalerweise wartet die Shell dabei auf das Ende des von ihr gestarteten Unterprozesses, bevor sie mit einem neuen Promptzeichen zurück kommt. Durch das Anhängen eines Undzeichens (Kaufmanns–Und, Ampersand) an eine Kommandozeile kann man der Shell mitteilen, daß sie *nicht* auf das Ende des Kommandos warten soll, sondern statt dessen sofort mit einem neuen Prompt zurückkommen soll. Auf diese Weise kann der Benutzer schon weiterarbeiten, ohne auf das Ende eines langwierigen Kommandos warten zu müssen.

Beispiel:

```
$ sort -o sortiertedatei langedatei &
$ jobs
[1]  + Running      sort -o sortiertedatei langedatei
$ ls -l
-rw-r--r--  1 kris         0 Oct  4 13:28 sortiertedatei
-rw-r--r--  1 kris   1743759 Oct  4 13:28 langedatei
$
[1]    Done         sort -o sortiertedatei langedatei
```

Abbildung 1.36: Verarbeitung im Hintergrund

Man sagt, daß Kommandos, auf die die Shell wartet, *im Vordergrund* abgearbeitet werden. Kommandos, auf die die Shell nicht wartet, werden *im Hintergrund laufend* genannt. Kommandos, denen ein & nachgestellt wird, laufen im Hintergrund ab. Einige spezielle Dienstprogramme für den Systemverwalter sind in der Lage, sich selbst in den Hintergrund zu schicken, ohne daß man ausdrücklich ein & anhängen müßte.

Mit dem in die Shell eingebauten Befehl **jobs** kann man sich ansehen, welche Hintergrundkommandos man zu einem gegebenen Zeitpunkt laufen hat. Der Befehl gibt eine Kommandozeile in der Form

```
[1]  + Running      sort -o sortiertedatei langedatei
```

aus. Die Ausgaben bedeuten im Einzelnen:

- [1]

  In den eckigen Klammer steht eine kleine ganze Zahl, die Jobnummer. Man kann diese Zahl verwenden, um den Job zu stoppen, ihn in den Vordergrund zu holen oder anders zu beeinflussen.

- +

  Das Pluszeichen steht vor dem neuesten Hintergrundprozeß. Vor dem zweitneuesten Prozeß steht ein Minuszeichen. Man kann diese Kürzel statt der Jobnummer für den neuesten und zweitneuesten Prozeß verwenden.

- Running

  Running ist einer der möglichen Zustände für einen Hintergrundprozeß. Ein solcher Prozeß arbeitet im Hintergrund fleißig vor sich hin. Andere mögliche Zustände sind *Done* als letzte Meldung eines Prozesses, der gerade fertig geworden ist, und *Stopped* für Prozesse, die angehalten sind und deswegen nicht arbeiten können.

- sort –o sortiertedatei langedatei

  Zu guter Letzt wird das Kommando, um das es geht, noch einmal ausgegeben.

1.7.1 Vordergrund – Hintergrund

Gelegentlich möchte man einen Prozeß, den man als Vordergrundprozeß gestartet hat, nachträglich in den Hintergrund senden. Dies geht, indem man das *suspend*–Zeichen (normalerweise Control–Z) eingibt. UNIX reagiert darauf mit einer Meldung und bringt wieder den Kommandoprompt. Der ehemalige Vordergrundprozeß liegt jetzt **angehalten** im Hintergrund. Mit dem Kommando **bg** kann man ihn im Hintergrund weiter laufen lassen.

Beispiel:

```
kris@black /Benutzer/kris/Texte/TeX/Afbb> sleep 3600
^Z
Stopped
kris@black /Benutzer/kris/Texte/TeX/Afbb> jobs
[1]  + Stopped              sleep 3600
kris@black /Benutzer/kris/Texte/TeX/Afbb> bg
[1]    sleep 3600 &
kris@black /Benutzer/kris/Texte/TeX/Afbb> jobs
[1]    Running              sleep 3600
kris@black /Benutzer/kris/Texte/TeX/Afbb>
```

Abbildung 1.37: Prozesse in den Hintergrund senden

Wenn man mehrere Hintergrundprozesse hat, kann man einzelne Hintergrundprozesse wieder zurück in den Vordergrund holen. Dazu dient das UNIX–Kommando **fg**. Damit UNIX weiß, welches Hintergrundkommando in den Vordergrund rücken soll, muß die Jobnummer dieses Kommandos mit einem Prozentzeichen angegeben werden.

1.7.2 Prozesse anhalten

Vordergrundprozesse können durch Eingabe des Unterbrechungszeichens (*interrupt character*, normalerweise Control–C) oder des Abbruchzeichens (*quit character*, normalerweise Control–\) angehalten werden. UNIX stoppt den Vordergrundprozeß dann, indem ihm intern ein Signal (in diesem Fall das Signal zum Abbruch) gesendet wird.

Beispiel:

```
kris@black /Benutzer/kris/Texte/TeX/Afbb> jobs
[1]    Running              sleep 3600
kris@black /Benutzer/kris/Texte/TeX/Afbb> fg %1
sleep 3600
^C
kris@black /Benutzer/kris/Texte/TeX/Afbb>
```

Abbildung 1.38: Prozesse in den Vordergrund holen

Hintergrundprozesse können ebenfalls durch Signale angehalten werden. Weil Hintergrundprozesse aber nicht mit der Tastatur verbunden sind, kann man ihnen das Signal nicht durch einen Tastendruck senden, sondern muß das UNIX–Kommando **kill** dafür bemühen. Der kill–Befehl nimmt als Option die Nummer oder den Namen eines Signals und als Parameter die Jobnummern von Prozessen. Er sendet dann das betreffende Signal an die bezeichneten Jobs.

Beispiel:

```
kris@black /Benutzer/kris/Texte/TeX/Afbb> jobs
[1]  + Running              sleep 3600
[2]  - Running              sleep 1800
kris@black /Benutzer/kris/Texte/TeX/Afbb> kill %1
kris@black /Benutzer/kris/Texte/TeX/Afbb>
[1]    Terminated           sleep 3600
kris@black /Benutzer/kris/Texte/TeX/Afbb> kill -QUIT %2
kris@black /Benutzer/kris/Texte/TeX/Afbb>
[2]    Quit                 sleep 1800
```

Abbildung 1.39: Abbrechen von Hintergrundprozessen

Prozesse können, wenn sie wollen, einige Signale abfangen (etwa um vor dem Abbruch noch Aufräumarbeiten zu erledigen) oder einfach ganz ignorieren. Das Signal mit der Nummer 9 (es hat den Namen KILL) kann jedoch weder abgefangen noch ignoriert werden. Es beendet einen Prozeß in jedem Fall sofort. Man sollte dieses Signal jedoch tatsächlich nur als Notbremse einsetzen, wenn alle anderen Anhaltemethoden versagt haben. Der betroffene Prozeß kann nämlich beim Empfang eines KILL–Signals keine Aufräumungsarbeiten mehr vornehmen und hinterläßt unter Umständen das Terminal in einer unerwünschten Betriebsart oder läßt Zwischendateien stehen.

1.8 Kontrollstrukturen

Die Kommandosprache der UNIX–Shell ist eine vollwertige Programmiersprache mit allen Kontrollstrukturen, die zu einer solchen Programmiersprache gehören (nur ein **goto** gibt es nicht). Shell als Programmiersprache ist eine interpretierte Sprache mit allen Vor– und Nachteilen, die sich daraus ergeben: Shellscript brauchen nicht compiliert zu werden und Shell–Variablen brauchen vor ihrer ersten Verwendung nicht deklariert zu werden, aber Shell–Scripte sind auch deutlich langsamer, als etwa C–Programme.

1.8.1 Bedingte Ausführung (if)

Die Syntax für die bedingte Ausführung von Anweisungen ist in Shell

```
if anweisungen1
then
        anweisungen2
else
        anweisungen3
fi
```

Der else–Zweig ist dabei, wie in den meisten anderen Programmiersprachen auch, optional. Die Anweisung funktioniert folgendermaßen:

- Die Kommandofolge *anweisungen1* wird ausgeführt.
- Die Shell betrachtet den Exit–Status des letzten Kommandos der Anweisungsfolge.
- Ist der Status 0, wird die Kommandofolge *anweisungen2* gestartet.
- Ist der Status ungleich 0, wird die Kommandofolge *anweisungen3* ausgeführt.

Häufig braucht man Konstruktionen von ineinander geschachtelten if–Anweisungen für Mehrfachverzweigungen. Bei Programmiersprachen, bei denen das if mit einer ausdrücklichen Anweisung angeschlossen werden muß (also auch in der Shell) sind solche Konstruktionen sehr unübersichtlich, weil man am Ende einer solchen Kaskade massenhaft fi–Anweisungen stehen hat. Um diese etwas unglückliche Konstruktion zu umgehen, kennt die Shell die elif–Anweisung zum Schachteln von if–Blöcken.

Beispiel:

```
#! /bin/sh --
if [ $# -eq 0 ]
then
    echo -n "Welcher Drucker soll benutzt werden:"
    read printer
    echo -n "Welche Datei soll gedruckt werden:  "
    read datei
elif [ $# -eq 1 ]
then
    echo -n "Welche Datei soll gedruckt werden:  "
    read datei
    printer=$1
else
    printer=$1
    datei=$2
fi
lpr -P$printer $datei
```

Abbildung 1.40: Mehrfachverzweigung mit if und elif

Das Programm im Beispiel verwendet den noch zu erläuternden Befehl [alias **test**, um die Anzahl der Parameter beim Aufruf des Scriptes festzustellen. Ist die Anzahl der Parameter gleich 0, so fragt das Script interaktiv nach einem Drucker und einer zu druckenden Datei.

Falls ein Parameter angegeben ist, so wird dieser als der Name des zu verwendenden Druckers angegeben und nur der Name der zu druckenden Datei wird erfragt. Sind beide Parameter angegeben, so druckt das Script kommentarlos. Zusätzliche Parameter werden verworfen.

Der Befehl [alias **test** wird extrem häufig für if–, while– und until–Konstruktionen verwendet. Um die Abarbeitung derartiger Kontrollstrukturen zu beschleunigen, wurde er in neueren Versionen von UNIX fest in die Shell eingebaut.

1.8.2 Bedingte Ausführung (|| und &&)

Gelegentlich wird in der Shell eine besondere, verkürzte Schreibweise für ein if–then ohne else oder ein if–else ohne then verwendet.

- Falls ein if–then ohne else vorliegt, kann man dies als

  ```
  anweisungen1 \&\& anweisungen2
  ```

  schreiben. Nur dann, wenn der exit–Status der letzten Anweisung von *anweisungen1* gleich 0 ist, wird die Anweisungsfolge *anweisungen2* gestartet.

- Falls ein if–else ohne then vorliegt, kann man dies als

  ```
  anweisungen1 $\|$ anweisungen3
  ```

  schreiben. Nur dann, wenn der exit–Status der letzten Anweisung von *anweisungen1* ungleich 0 ist, wird die Anweisungsfolge *anweisungen3* gestartet.

Beispiel:

```
[ -x /etc/lavd ] && /etc/lavd

[ -r /usr/local/lib/config ] ||
    chmod a+r /usr/local/lib/config
```

Abbildung 1.41: Beispiel für && und ||

1.8.3 Mehrfachentscheidungen (case)

Für Mehrfachentscheidungen hat die Shell die Konstruktion

```
case wort in
    muster1) anweisungen1 ;;
    muster2) anweisungen2 ;;
    .
    .
    *) anweisungenx ;;
esac
```

Das *wort* kann dabei eine Konstante oder eine Variable oder sonst ein Ausdruck sein, der einen String generiert. Es wird von oben nach unten mit den angegebenen Mustern verglichen. Paßt eines der Muster auf das Wort, so wird die dahinter angegebene Anweisungsfolge ausgeführt. Danach endet das case, sodaß bei mehr als einem passenden Muster nur die erste Übereinstimmung aktiviert wird.

Das *muster* darf dabei ein Suchmuster sein, wie es auch für Dateinamen erlaubt ist. Es können mehrere alternative Muster durch ein | (pipeline–Zeichen) getrennt angegeben werden.

Das Suchmuster * paßt auf jedes Wort. Es kann damit verwendet werden, um einen Default–Fall zu formulieren. Man sollte jedoch darauf achten, daß dieser Default als Letzter in der Liste der angegebenen Fälle aufgeführt ist, weil sonst die Fälle dahinter niemals erreicht werden können.

Beispiel:

```
echo $datei
read eingabe
case "$eingabe" in
    del) rm $datei ;;
    m)   mv $datei $HOME ;;
    p)   $PAGER $datei ;;
    *)   echo "Ungueltige Eingabe" ;;
esac
```

Abbildung 1.42: Beispiel für case ... esac

1.8.4 Listeniterator (for)

Mit der Befehlsfolge

```
for variable in wort1 wort2 ...
do
   anweisungen
done
```

kann man eine Anweisungsfolge in der Shell für unterschiedliche Belegungen einer Variablen ausführen lassen. Die *variable* im Kopf der Schleife wird nacheinander mit den Werten *wort1*, *wort2* und so weiter aus der Liste belegt. Für jede Belegung der Variablen wird der Rumpf *anweisungen* der Schleife einmal ausgeführt. Auf diese Weise kann man zum Beispiel eine Liste von Worten abarbeiten lassen (Siehe Beispiel 1.43).

Interessante Anwendungen ergeben sich, wenn die Liste der abzuarbeitenden Wörter nicht ausdrücklich angegeben wird, sondern durch eine Kommandoersetzung oder durch Dateinamenexpansion erzeugt wird (Siehe Beispiel 1.44).

Das Beispiel 1.44a erzeugt die Liste der abzuarbeitenden Worte durch Dateinamenexpansion. Die Shell ersetzt das Wort "*.c" durch eine Folge von Dateinamen, die auf dieses Muster passen. Angenommen, die Ersetzung würde die Worte **prog.c modul.c daten.c** erzeugen, dann würden nacheinander **cc**–Aufrufe für diese Dateien erzeugt werden.

Im Beispiel 1.44b wird ein hypothetischer Befehl **userliste** verwendet. Angenommen, diese Befehl gebe eine Liste von Benutzern auf einem UNIX–Rechner aus. Dann würde die Datei

Beispiel:

```
$ for i in 1 2 ja nein "langes wort"
> do
>   echo $i
> done
1
2
ja
nein
langes wort
$
```

Abbildung 1.43: Beispiel für for

Beispiel:

```
$ for i in *.c
> do
>   cc -o ‘basename $i .c‘ $i
> done
$ for i in ‘userliste‘
> do
>   cp datei /home/$i/wichtigeinformation
> done
```

Abbildung 1.44: Beispiele für for mit variabler Liste

mit dem Namen datei nacheinander unter dem Namen wichtigeinformation in die Home–Verzeichnisse aller diese Benutzer kopiert werden.

1.8.5 Abweisende, bedingte Schleife (while)

Während die **for**–Schleife eher zum Abzählen oder Durchzählen einer vorgegeben Menge von Worten geeignet ist, dienen **while**–Schleifen dazu, eine Anweisung zu wiederholen, solange eine bestimmte Bedingung gilt. Eine while–Schleife ist ein abweisendes Konstrukt. Das bedeutet, daß der Schleifenrumpf niemals betreten wird, wenn die Schleifenbedingung gleich am Anfang falsch ist. Die Shell kennt eine solche Schleife mit folgender Sytax:

```
while anweisungen1
do
        anweisungen2
done
```

Die Shell führt die Anweisungsfolge *anweisungen1* aus und wertet dann den Exit–Status des letzten Kommandos dieser Kette aus. Ist dieser Status 0, so werden die *anweisungen2* gestartet. Danach wiederholt sich dieser Zyklus, solange bis *anweisungen1* einen von 0 verschiedenen Exit–Status ergibt. Wenn der Exit–Status der Anweisungen in *anweisungen1* also immer garantiert 0 ist, hat man eine Endlosschleife, die nur mit einem **break** (Siehe Abschnitt 1.8.7) verlassen werden kann.

Man kann die while–Konstruktion verwenden, um in der Shell einen primitiven Zähler aufzubauen. An der Ablaufgeschwindigkeit der Schleife kann man bei diesem Beispiel

aber auch schon deutlich die Grenzen der Leistungsfähigkeit der Shell erkennen: Immerhin müß für jede arithmetische Operation der externe Befehl **expr** geladen und gestartet werden.

Beispiel:

```
$ zaehler=0
$ while [ $zaehler -le 10 ]
> do
>         zaehler=‘expr $zaehler + 1‘
>         echo -n "$zaehler "
> done
1 2 3 4 5 6 7 8 9 10 11 $
```

Abbildung 1.45: Beispiel für eine zäehlende while–Schleife

1.8.6 bedingte Schleife, invertierte Bedingung (until)

Die **until**–Schleife stellt eine **while**-Schleife mit invertierter Schleifenbedingung dar. Im Gegensatz zur gleichnamigen Konstruktion in anderen Programmiersprachen handelt es sich *nicht* um eine nichtabweisende Schleife, sondern wie ein kleiner Test schnell zeigt, ist auch die until–Konstruktion abweisend, d.h. die Schleifenbedingung wird vor dem Schleifendurchlauf getestet. Dies wird auch schon durch die syntaktische Anordnung der Bedingung klar.

```
until anweisungen1
do
        anweisungen2
done
```

Hier wird der Schleifenrumpf *anweisungen2* ausgeführt, wenn und solange der Exit–Status von *anweisungen1* ein Ergebnis ungleich 0 liefert.

Beispiel:

```
$ until true
> do
>         echo aha
> done
$
```

Abbildung 1.46: Auch die until–Schleife ist eine abweisende Schleife

Eine nichtabweisende Schleifenkonstruktion fehlt der Shell, aber mit der im Folgenden erläuterten **break**–Anweisung kann man diese (und jede andere Schleifenkonstruktion) simulieren.

1.8.7 Schleifenkurzschlüsse (break, continue)

Gelegentlich möchte man eine Schleife kurzschließen, d.h. man möchte sie nicht auf dem normalen Wege bis zum Ende durchlaufen, sondern sofort einen neuen Schleifendurchlauf beginnen oder die Schleifenbearbeitung abbrechen und die Abarbeitung von Anweisungen

nach dem Schleifenrumpf fortsetzen. Für beide Fälle stellt die Shell spezielle Anweisungen zur Verfügung.

Die Anweisung **continue** bewirkt einen Abbruch des aktuellen Schleifendurchlaufes. Es wird sofort der nächste Schleifendurchlauf eingeleitet. Auf diese Weise könnte man etwa bestimmte Sonderfälle in einer Schleifenbearbeitung auslassen.

Beispiel:

```
$ for i in 1 2 3 4 5 6 7 8 9
> do
>         if [ "$i" = 5 ]
>         then
>                 continue
>         fi
>         echo -n "$i "
> done
1 2 3 4 6 7 8 9 $
```

Abbildung 1.47: Zählschleife mit Auslassung eines Sonderfalles

Die Anweisung **break** bewirkt einen sofortigen, vorzeitigen Abbruch der Schleife. Das Programm wird hinter dem **done** wieder aufgenommen. Im Zusammenhang mit der Schleifenbedingung **true** hat man so eine Konstruktion mit einem Schleifenausgang in der Mitte.

Beispiel:

```
$ zaehler=0
$ while true
> do
>         read eingabe
>         if [ "$eingabe" = "ende" ]
>         then
>                 break
>         fi
>         zaehler=‘expr $zaehler + $eingabe‘
>         echo "Summe = $zaehler"
> done
10
Summe = 10
2
Summe = 12
14
Summe = 26
ende
$
```

Abbildung 1.48: Beispiel für eine Schleife mit mittigem Ausgang

1.8.8 Definition von Funktionen

Die Shell erlaubt die Zusammenfassung von mehrfach benutzten Anweisungsfolgen unter einem Namen zu einer Funktion. Die Syntax ist

```
name()
{
        anweisungen
}
```

Durch diese Konstruktion wird dem angegebenen Bezeichner *name* die Anweisungsfolge *anweisungen* zugeordnet. Durch den Aufruf von *name* kann diese Anweisungsfolge jetzt beliebig oft abgerufen werden. Der Aufruf der Funktion kann mit Parametern erfolgen, die in der Funktion entsprechend ihrer Position als **$1**, **$2**, ... zur Verfügung stehen. Der Parameter $0 wird nicht belegt, sondern enthält unverändert den Namen des Shellscriptes.

Beispiel:

```
$ pause()
> {
>         echo -n "Bitte RETURN druecken:"
>         read q
> }
$ pause
Bitte RETURN druecken:
$
```

Abbildung 1.49: Der pause–Befehl von MS–DOS als UNIX–Shellfunktion

Funktionsdefinitionen werden wie Variablen mit **unset** gelöscht, können aber nicht an Untershells exportiert werden. Die Eingabe von **set** listet unglücklicherweise auch die definierten Funktionen komplett mit den Funktionsrümpfen. Bei einem größerem Script, das viele und große Funktionen verwendet, wird die Ausgabe von set so bis zur Unbrauchbarkeit unübersichtlich. Trotzdem ist der Funktionsmechanismus sehr nützlich und insbesondere bei der Strukturierung größerer Shellscripte sehr nützlich.

Mit der Anweisung **return** kann eine Funktion einen Exit–Status für ihren Aufrufer hinterlassen. Dieser kann den Exit–Status wie üblich auswerten und entsprechend reagieren.

Das Beispiel 1.50 zeigt die Funktion yesno. Die Funktion nimmt als Parameter einen Prompttext an, der dem Benutzer praesentiert wird. Die Option –n beim **echo**–Kommando bewirkt dabei, daß nach der Ausgabe des Textes kein Zeilenvorschub erzeugt wird. Danach wird eine Eingabe vom Benutzer in die Variable $ein eingelesen und ausgewertet. Falls die Benutzereingabe mit einem der Buchstaben aus der Menge [yYjJ] beginnt, wird die Eingabe als Bestätigung gewertet und die Funktion beendet sich mit einem Exit–Status von 0 (wahr). Beginnt die Eingabe mit einem der Buchstaben aus der Menge [nN], wird die Eingabe als Verneinung gewertet und die Funktion beendet sich mit einem Exit–Status von 1. In jedem anderen Fall wird ein Hinweistext ausgegeben und der ganze Ablauf wiederholt sich in einer endlosen Schleife.

1.8.9 Klammerung von Anweisungen

Manche Shell–Befehle erstrecken sich über mehr als eine Zeile. In diesem Fall fordert die Shell weitere Zeilen durch Ausgabe des Prompts für Fortsetzungszeilen (ein „>“–Zeichen) an. Umgekehrt ist es auch möglich, mehr als einen Shell–Befehl in eine Kommandozeile zu schreiben. Die einzelnen Befehle sind dann durch ein „;“ voneinander zu trennen [2].

[2] Als Trennzeichen zwischen den Befehlen ist bei einigen älteren Shells auch noch das „^“(Dach) erlaubt. Dies ist jedoch veraltet und soll nicht mehr verwendet werden. Neuere Shells erkennen das Dach auch nicht mehr als Trennzeichen zwischen den Befehlen an.

Beispiel:

```
$ yesno()
> {
>     while true
>     do
>         echo -n "$1 (j/n):"
>         read ein
>         case "$ein" in
>             [yYjJ]*) return 0 ;;
>             [nN]*)   return 1 ;;
>             *)       echo "Bitte j oder n eingeben." ;;
>         esac
>     done
> }
$ while yesno "Fortsetzen?"
> do
>         machwas
> done
```

Abbildung 1.50: Die Funktion yesno mit Parameter und Exit–Status

Mit den geschweiften und den runden Klammern lassen sich Shell–Befehle zu einem Block zusammenklammern. Die Syntax lautet

```
{ anweisungen ; }
```

und

```
( anweisungen )
```

Man beachte dabei, daß in der ersten Variante hinter der letzten Anweisung und vor der Klammer noch ein Semikolon stehen muß. Bei den runden Klammern ist das nicht notwendig.

Die Klammern bewirken, daß die Anweisungen in der Klammer zu einem Anweisungsblock zusammengefaßt werden, dessen Ein– und Ausgabe gemeinsam umgelenkt werden kann. Der Unterschied zwischen geklammerten und ungeklammerten Anweisungen wird in einem Beispiel schnell deutlich.

Beispiel:

```
$ echo "Titelzeile"; cat /etc/passwd | wc -l
Titelzeile
22
$ { echo "Titelzeile"; cat /etc/passwd; } | wc -l
23
```

Abbildung 1.51: Klammerung von Befehlen

Im ersten Fall wird die Titelzeile direkt ausgegeben und der nachfolgende **wc –l**–Befehl wird nur auf das **cat**–Kommando. In der geklammerten Folge von Anweisungen wird die Titelzeile zusammen mit der Ausgabe des **cat**–Kommandos zur Eingabe des hinteren Teils

der Pipeline. Intern in der Shell werden die geschweiften Klammern als die Definition einer namenlosen Funktion angesehen, die auch noch an Ort und Stelle ausgeführt wird.

Während Kommandos in geschweiften Klammern innerhalb der Shell ausgeführt werden, wird zur Ausführung von Befehlen in runden Klammern ein eigenständiger Subshell–Prozeß gestartet. Definitionen von Variablen, Veränderungen am aktuellen Verzeichnis und sonstige Veränderungen an der Umgebung eines Prozesses finden innerhalb der Subshell statt und haben keine Wirkungen auf die Umgebung der eigentlichen Kommandoshell.

Der Exit–Status einer Kommandoklammer entspricht dem Exit–Status des letzten Kommandos in der Klammer.

1.9 Eingebaute Befehle

1.9.1 Verzeichnis der eingebauten Befehle

Eine ganze Reihe von Befehlen ist in die Shell eingebaut und wird direkt ausgeführt, ohne daß die Erzeugung eines Unterprozesses notwendig wird. Es sind dies:

*kommentar* Der Teil der Kommandozeile vom Doppelkreuz bis zum Ende der Zeile wird ignoriert.

: *parameter* Der Doppelpunkt wirkt als no–operation Befehl. Er kann beliebige *Parameter* erhalten, die ebenfalls nichts bewirken. Der Befehl erzeugt den Exit–Status 0.

. *dateiname* Das Punkt–Kommando läßt die aktuelle Shell eine Folge von Anweisungen aus der angegebenen *datei* lesen. Im Gegensatz zu einem regulären Shellscript werden die Kommandos *nicht* in einer Subshell ausgeführt.

break *level* Beendet die aktuelle **for**–, **while**– oder **until**–Schleife. Mit der Angabe *level* kann man auch verschachtelte Schleifen abbrechen; die Angabe bestimmt, wie viele Verschachtelungsebenen auf einmal verlassen werden sollen.

cd *verzeichnis* Wechselt das aktuelle Verzeichnis bzw. in das durch die Variable **$HOME** angegebene Verzeichnis, wenn die Angabe *verzeichnis* fehlt.

continue *level* Beendet die aktuelle Schleife und startet sofort den nächsten Schleifendurchlauf. Die Angabe *level* wirkt wie beim **break**–Kommando.

echo *text* Das Kommando gibt seine Argumente aus. Bei älteren UNIX–Systemen ist **echo** *kein* internes Kommando.

eval *kommando* Wertet den Parameter *kommando* als Kommando aus (mit allen Textersetzungen) und führt ihn dann aus.

exec *kommando* Startet das als *kommandp* angegebene Programm an Stelle der Shell, d.h. der Shell–Prozeß ersetzt sich selbst durch das angegebene Kommando.

exit *status* Beendet die aktuelle Shell mit dem angegebenen *status*.

export *variable* **...** Die angegebene *variable* wird zum Export markiert, d.h. sie wird Bestandteil der Ausführungsumgebung aller nachfolgenden Kommandos.

hash (-r) *name* Bei langen Suchpfaden kann das Durchsuchen des aktuellen Pfades sehr lange dauern. Die Shell merkt sich daher den vollen Pfadnamen eines Kommandos, das sie schon einmal ausgeführt hat. Das Kommando **hash -r** löscht diese Liste und erzwingt ihren Neuaufbau. Es ist notwendig, wenn während der Laufzeit der Shell

neue Kommandos im Suchpfad dazugekommen sind. Ohne Angabe von Parametern bewirkt das Kommando die Ausgabe der kompletten Liste, mit der Angabe von *name* wird nur der gemerkte Pfad zum angegebenen Kommando ausgegeben.

pwd Der Name des aktuellen Verzeichnisses wird ausgegeben. Das Kommando existiert zum Teil auch noch als externer Befehl.

read *variable* **...** Der Befehl liest eine Zeile der Standardeingabe und unterteilt sie in Worte. Siehe Abschnitt 1.9.2

readonly *variable* **...** Die angegebenen Variablen werden schreibgeschützt, d.h. sie sind fortan nur noch als Konstanten zu verwenden. Ohne Angabe von Variablennamen werden die Namen aller schreibgeschützten Variablen ausgegeben.

return *status* Beendet die aktuelle Shellfunktion mit dem angegebenen *status*.

set *option parameter* Siehe Abschnitt 1.9.3.

shift *stellen* Verschiebt die Liste der Variablen **$1**, **$2**, ... um eine Stelle nach links. Siehe Abschnitt 1.9.3.

test Prüft die angegebene Bedingung und erzeugt je nachdem einen Exit–Status von 0 oder nicht 0. Der Befehle ist auch unter dem Namen **[** verfügbar. In vielen UNIX–Versionen existiert er außerdem noch als externes Kommando. Siehe Abschnitt 1.11.8

times Gibt die bisher verbrauchte CPU–Zeit der aktuellen Shell aus.

trap *kommando signal ...* Wenn die Shell das angebene *signal* empfängt, wird das angegebene *kommando* gestartet, danach wird die Ausführung normal fortgesetzt. Siehe Abschnitt 1.9.4.

type *name* Die Shell gibt aus, welches Programm gestartet wird, wenn *name* als Kommando eingegeben wird.

umask *wert* Mit diesem Kommando kann festgelegt werden, welche Zugriffsrechte eine Datei beim Anlegen standardmäßig bekommt. Beim Anlegen einer Datei gibt ein Prozeß die von ihm gewünschten Zugriffsrechte vor (z.B. 0666). Davon wird die umask (z.B. 0022) ohne Übertrag abgezogen. Die resultierende Zahl gibt die tatsächlich installieren Zugriffsrechte (hier: 0644) an.

unset *name* **...** Löscht die Variablen oder Funktionen mit den angegebenen *namen*.

wait *pid* Der Befehl läßt die Shell auf das Ende des Hintergrundprozesses mit der Prozeßnummer *pid* warten. Fehlt die Angabe, so wird auf das Ende aller Hintergrundprozesse gewartet.

1.9.2 read – Einlesen von Daten

Das Kommando `read` bekommt als Parameter eine Liste von Variablennamen. Es liest eine Zeile von der Standardeingabe und unterteilt diese in Worte. Jedes gelesene Wort wird von links nach rechts einer der angegebenen Variablen zugewiesen. Sind nicht genügend Worte vorhanden, werden die verbleibenden Variablen mit dem leeren Wort belegt. Sind mehr Worte als Variablennamen vorhanden, so werden der letzten angegebenen Variablen die übriggebliebenen Worte zugewiesen.

`read` liest von der Standardeingabe, diese kann wie gewohnt umgelenkt werden. Der Exitcode von `read` ist Null, falls das Einlesen der Werte gelungen ist. Trifft der Befehl beim Lesen auf ein Dateiende (*end of file*, wird dies mit einem Exitcode ungleich Null signalisiert.

Beispiel:

```
$ read a b c   # 3 Variablen, 3 Worte
eins zwei drei
$ echo x $a x $b x $c x
x eins x zwei x drei x
$ read a b c   # 3 Variablen, 1 Wort
eins
$ echo x $a x $b x $c x
x eins x x x
$ read a b c   # 3 Variablen, 5 Worte
eins zwei drei vier fuenf
$ echo x $a x $b x $c x
x eins x zwei x drei vier fuenf x
```

Abbildung 1.52: Verhalten von `read`

Beispiel:

```
$ while read zeile
> do
>   echo $zeile # Zeile verarbeiten
> done < /tmp/eingabedatei
zeile1
zeile2
zeile3
```

Abbildung 1.53: `read` und `while`

Dies erlaubt es, den Befehl als Steuerkommando einer `while`–Schleife einzusetzen und so z.B. eine Datei zeilenweise zu bearbeiten.

Im Beispiel wird die Standardeingabe der gesamten `while`–Schleife umgelenkt, sodaß aus einer Datei gelesen wird. Jeweils eine Zeile der Eingabedatei wird in die Variable *zeile* eingelesen und dann in der Schleife verarbeitet. Die Schleife bricht ab, wenn der Exitcode von `read` ungleich Null wird. Das ist der Fall, wenn das Ende der Eingabedatei erreicht wird.

1.9.3 set und shift – Kommandozeilenparameter

Die Anweisung **set** hat in der Shell mehrfache Funktionen. Einmal dient sie dazu, Kommandozeilenoptionen bei einer laufenden Shell nachträglich zu setzen (mit **set –options**) und zu löschen (mit **set +options**).

Andererseits kann man den **set**–Befehl auch verwenden, um die Variablen **$1**, **$2**, ... neu zu belegen. Die Syntax lautet in diesem Fall

```
$ set -- wort1 wort2 ...
```

Die Angabe von ”– –” ist in diesem Fall optional. Die beiden Minuszeichen stehen, wie im Kapitel 1.1 erwähnt, um das Ende der Optionsliste zu kennzeichnen. Dadurch werden *wort1* und alle folgenden Parameter als Worte und nicht als Optionen für den set–Befehl erkannt,sodaß das Kommando auch dann richtig arbeitet, wenn *wort1* mit einem Minus– oder Pluszeichen beginnt.

Wenn der Befehl in der beschriebenen Weise gebraucht wird, ordnet er *wort1* der Variablen **$1**, *wort2* der Variablen **$2**, ... und *wort9* der Variablen **$9** zu. Auf diese Weise kann man eine Folge von Worten bequem zerlegen und auf einzelne Worte der Folge ohne große Schwierigkeiten zugreifen.

Beispiel:

```
$ read ein
$ set -- $ein ""
$ echo "Wort1: $1  Wort2: $2  Wort3: $3"
```

Abbildung 1.54: Zerlegung einer Benutzereingabe in Einzelteile

Die Shell erlaubt den direkten Zugriff nur auf die ersten neun Kommandozeilenparameter. Auf eventuell vorhandene, weitere Parameter kann man nur mit der Anweisung **shift** zugreifen. Dieses Kommando schiebt die Kommandozeilenparameter um ein Wort nach links, sodaß **$2** nach dem shift unter dem Namen **$1**, **$3** unter dem Namen von **$2** und so weiter verfügbar ist.

Vor dem Shift	Nummer	Nach dem Shift
Dies	**$1**	sind
sind	**$2**	die
die	**$3**	Parameter
Parameter	**$4**	einer
einer	**$5**	Kommandozeile.
Kommandozeile.	**$6**	<leer>

Abbildung 1.55: Funktionsweise von shift

Wenn man ein Kommando eine beliebige Anzahl von Parametern verarbeiten lassen möchte, geht man in der Regel so vor, daß man immer nur einen Parameter (etwa immer **$1**) betrachtet und dort nacheinander mit **shift** die Werte hineinschiebt. Sobald der nachgeschobene Wert leer ist, ist man fertig.

Beispiel:

```
#! /bin/sh --
while [ "$1" != "" ]
do
        echo "Bearbeite Parameter $1"
        shift
done
```

Abbildung 1.56: Beispiel für die Abarbeitung beliebig vieler Parameter

1.9.4 trap – Behandlung von Signalen durch die Shell

Das Kommando `trap` dient der Handhabung von Signalen in Shellscripten. Es ermöglicht unter anderem, ein Shellscript gegen den Abbruch durch den Benutzer zu schützen. Signale können von einem UNIX–Prozeß auf drei Arten behandelt werden:

- Der Prozeß kann durch das Signal abgebrochen werden. Dies ist für die meisten Signale die Voreinstellung.
- Der Prozeß kann das Signal ignorieren. Er verhält sich so, als ob das Signal gar nicht gesendet worden wäre.
- Der Prozeß installiert eine Signalbehandlungsroutine. Tritt ein Signal auf, so wird diese Routine abgearbeitet und das Programm danach normal fortgesetzt.

Der Befehl `trap` kann dementsprechend auf drei Arten benutzt werden, um jeden dieser Effekte zu erzielen.

trap *signalliste* Die Angabe einer Signalliste mit `trap` bewirkt, daß die angegebenen Signale defaultmäßig behandelt werden. Für die meisten Signale bedeutet dies, daß der Prozeß beim Eintreffen des Signals abgebrochen wird.

trap ”” *signalliste* Die Angabe eines leeren Kommandos vor der Signalliste bewirkt, daß die betreffenden Signale vom Shellscript ignoriert werden. Einige Signale, namentlich das Signal KILL und das Signal STOP, können nicht ignoriert werden.

trap ”*kommando***”** *signalliste* Die Angabe eines Kommandos und einer Liste von Signalen bewirkt, daß das angegebene Kommando ausgeführt wird, wenn eines der aufgeführten Signale eintrifft. Nach der Abarbeitung des Kommandos wird die Ausführung des Shellscriptes an der Stelle fortgesetzt, an der es unterbrochen wurde.

Das *kommando* wird von der Shell zweimal gelesen, d.h. es läuft zweimal durch die verschiedenen Ersetzungsmechanismen der Shell: Einmal wenn der Trap installiert wird und einmal, wenn er ausgeführt wird.

Die Signalnummer 0 hat in der Shell eine besondere Bedeutung. Ein Trap, der für das Signal 0 installiert wird, wird ausgeführt, wenn das Shellscript beendet wird.

Beispiel:

```
$ cat traptest
#! /bin/sh --

trap "echo Tschuess" 0
trap "" 2
trap "echo SIGQUIT gesehen" 3

zaehler=0
while [ $zaehler -le 10 ]
do
        echo -n "$zaehler "
        zaehler=‘expr $zaehler + 1‘
        sleep 1
done
$ traptest
0 ^C1 ^\traptest: 16050 Quit
SIGQUIT gesehen
2 3 4 5 6 7 8 9 10 Tschuess
```

Abbildung 1.57: Ignorieren von SIGINT (Control-C)

Im Beispiel 1.57 werden drei Signalbehandlungsroutinen installiert: Das Signal 0 wird mit einem `echo`–Kommando belegt. Vor dem Ende des Scriptes wird dieses Kommando ausgeführt. Das Signal 2 (SIGINT, Drücken von Control-C) wird ignoriert. Das Signal 3 (SIGQUIT, Drücken von Control-\) wird mit einem `echo`–Kommando belegt. Das eigentliche Script zählt im Sekundentakt von 0 bis 10. Nach dem Starten des Scriptes versucht ein Benutzer, das Script durch Drücken von Control-C und Control-\ zu unterbrechen.

1.10 Reguläre Ausdrücke

Sehr häufig muß man aus einer Datei alle Worte oder Zeilen heraussuchen, die auf ein bestimmtes Suchmuster passen oder genau nicht passen. Einen, wenn auch sehr eingeschränkten Mechanismus, der das leistet, stellen die Jokerzeichen der Shell dar (Siehe Abschnitt 1.3). UNIX kennt noch eine zweite Klasse von *regulären Ausdrücken*, die leistungsfähiger, aber auch komplizierter sind und in vielen Programmen Anwendung finden[3]. Weil das Programm `ed` das erste Programm war, das das hier beschriebene Format von regulären Ausdrücken verwendet hat, beziehen sich viele Manual–Pages in UNIX auf *ed–style regular expressions*. Gemeint ist, daß das Programm dieselbe Form von regulären Ausdrücken versteht, wie `ed`.

Ausdrücke jeder Art definiert man häufig induktiv. Das bedeutet, man definiert zunächst einmal, was ein atomarer Ausdruck ist und was er bedeutet. Danach definiert man, wie man atomare Ausdrücke zu komplizierteren Formeln zusammensetzen kann und was diese Formeln bedeuten sollen. Im Falle von regulären Ausdrücken heißt das, daß man zunächst die Ausdrücke definiert, die auf ein einzelnes Zeichen passen und danach Regeln angibt, wie man kompliziertere Ausdrücke konstruieren kann, die auf mehr als ein Zeichen passen.

[3] Die Programme `ed`, `awk`, `sed`, `egrep`, `vi` und noch einige andere verwenden reguläre Ausdrücke in der hier beschriebenen Form. Zwischen den verschiedenen Programmen existieren noch weitere, kleine Unterschiede im Umfang der regulären Ausdrücke, die erkannt werden, aber im Großen und Ganzen kennen alle Programme denselben Umfang an Sonderzeichen und -bedeutungen

In den folgenden Definitionen bedeutet SZeichenëin beliebiges Zeichen ausgenommen den Zeilenvorschub.

Normale Zeichen Ein Zeichen, das keine Spezialbedeutung hat, paßt auf sich selber. Zeichen mit Spezialbedeutung sind das Begrenzungszeichen des regulären Ausdrucks, die Zeichen \ (Backslash), [(eckige Klammer auf) und . (Punkt) sowie an bestimmten Stellen auch die Zeichen^(Dach), * (Stern) und $ (Dollar).

In einem regulären Ausdruck paßt das Zeichen `a` also auf ein `a` in einem Text.

. (*Punkt*) Der Punkt steht für ein einzelnes, beliebiges Zeichen. Er entspricht also dem Jokerzeichen ? in der Shell.

Der reguläre Ausdruck `.` paßt also auf die Texte `a`, `b`, `0`, `$` und so weiter.

\ **(*backslash*)** Der Backslash gefolgt von einem beliebigen Zeichen (ausgenommen eine Ziffer und die runden Klammern) steht für dieses Zeichen. Auf diese Weise kann man also einzelne Zeichen mit Spezialbedeutung maskieren.

Der reguläre Ausdruck `\\` paßt also auf den Text \, der Ausdruck `\[` auf den Text `[` und so weiter.

^ **(Dach)** steht für einen Zeilenanfang oder einen Zeichenkettenanfang. Mit dem Zeichen ^am Anfang eines regulären Ausdrucks läßt sich dieser an einem Zeilenanfang verankern.

Dementsprechend paßt der Ausdruck `a` auf ein `a` an beliebiger Stelle in einem Text, der Ausdruck`^a` hingegen nur auf ein `a` am Anfang einer Zeile oder eines Strings.

$ (Dollar) steht für ein Zeilenende oder ein Zeichenkettenende. Mit dem Zeichen $ am Ende eines regulären Ausdrucks läßt sich dieser an einem Zeilenende verankern.

Analog zum Dach paßt der Ausdruck `a$` nur auf ein `a` am Ende einer Zeile oder Zeichenkette.

[...](*Zeichenmengen*) Ein nichtleeres Paar von eckigen Klammern steht für eines der Zeichen, die in den Klammern aufgeführt worden sind. Dieser Mechanismus ist analog zum selben Mechanismus in der Shell. In einer Zeichenmenge hat der Backslash keine Spezialbedeutung und die schließende, eckige Klammer (]) darf nur als erstes Zeichen aufgeführt werden. Der Ausdruck `a-b` bezeichnet den Bereich der ASCII–Zeichen von `a` bis `b`, jeweils einschließlich.

Der Ausdruck `[0-3]` paßt also auf den Text `0`, `1`, `2` oder `3`.

[^...](*negierte Zeichenmengen*) Definiert entsprechend eine Zeichenmenge, die für ein einzelnes Zeichen steht, das **nicht** aufgeführt worden ist.

Diese einzelnen Zeichen lasssen sich jetzt zu komplizierteren Ausdrücken zusammensetzen.

ab (Verkettung) Wenn zwei reguläre Ausdrücke *a* und *b* hinterander geschrieben werden, bilden sie einen neuen regulären Ausdruck, dessen vorderer Teil auf *a* und dessen Rest auf *b* passen muss.

a|b (Alternation) Wenn zwei reguläre Ausdrücke *a* und *b* durch einen senkrechten Strich zusammengesetzt werden, bilden sie einen regulären Ausdruck, der entweder auf *a* oder *b* paßt.

Der Ausdruck `ta|bx` paßt also entweder auf den Text `tax` oder den Text `tbx`.

a* (Wiederholung ab 0) Wenn *a* ein regulärer Ausdruck ist, dann paßt der reguläre Ausdruck *a** auf null oder mehr Wiederholungen des Textes, auf den *a* paßt.

Der Ausdruck `xy*z` paßt auf die Texte `xz`, `xyz`, `xyyz` und so weiter.

a+ (Wiederholung ab 1) Wenn *a* ein regulärer Ausdruck ist, dann paßt der reguläre Ausdruck *a** auf mindestens eine oder mehr Wiederholungen des Textes, auf den *a* paßt.

Der Ausdruck `xy+z` paßt auf die Texte `xyz`, `xyyz` und so weiter, nicht jedoch auf `xz`.

a? (optionaler Teilausdruck) Wenn *a* ein regulärer Ausdruck ist, dann paßt der reguläre Ausdruck *a?* auf null oder eine Wiederholung des Textes, auf den *a* paßt.

Der Ausdruck `xy?z` paßt also genau auf die Texte `xz` und `xyz`.

(a) (Gruppierung) Wenn *a* ein regulärer Ausdruck ist, dann ist auch *(a)* ein regulärer Ausdruck, der auf denselben Text paßt wie *a*.

Der Ausdruck `(text)|(test)` paßt also entweder auf den Text `text` oder auf den Text `test`.

1.11 Einige nicht eingebaute Befehle

Einige UNIX–Kommandos kommen sind für die Programmierung von Shellscripten unentbehrlich, sie kommen in praktisch jedem Script vor. In den nachfolgenden Abschnitten sollen die wichtigsten dieser Kommandos vorgestellt werden.

1.11.1 cut und paste – Löschen und Einfügen von Spalten

Mit Befehlen wie `cat`, `head`, `split` und `tail` kann man Dateien zeilenweise in kleinere Stücke zerlegen oder wieder zu einer großen Datei zusammensetzen. Mit den Programmen `cut` und `paste` kann man solche Operationen auch spalten– oder zeichenweise vornehmen.

Der Befehl `cut` dient dazu, aus jeder Zeile einer Datei einen bestimmten Bereich von Feldern oder Zeichen auszuschneiden und auszudrucken. In der Form `cut -c<liste> datei` schneidet der Befehl die in der *liste* benannten Zeichen aus und druckt diese. In der Form `cut -f<liste> datei` werden statt einzelner Zeichen ganze Felder ausgeschnitten. Dabei ist ein Feld ein Wort, das durch ein Feldtrennzeichen begrenzt wird. Standardmäßig ist dieses Feldtrennzeichen ein Tabulatorzeichen, aber mit der Option `-d<zeichen>` kann ein beliebiges anderes Zeichen eingestellt werden.

Die *liste* die in beiden Formen des Befehls angegeben werden muß, spezifiziert den Bereich von Feldern, der auszuschneiden ist. In jedem Fall muß die Angabe ein Wort sein, also ohne Leerzeichen geschrieben werden. In der Bereichsangabe können mit Komma getrennt die Nummern der auszudruckenden Felder oder Zeichen angegeben werden. Dabei können zusammenhängende Bereiche in der Form `von-bis` angegeben werden. Offene Bereiche („Alles vom Anfang bis hier“ oder „Alles von hier bis zum Ende“) können in der Form `-bis` oder `von-` geschrieben werden.

Im Beispiel werden die Felder 1,3 und 4 der Datei /etc/passwd gedruckt. Feldtrenner ist hier der Doppelpunkt. Im zweiten Beispiel werden alle Zeichen vom Anfang der Zeile bis zum fünften Zeichen der Zeile ausgegeben. Wie man sieht, schließen Bereichsangaben in jedem Fall die Bereichsgrenzen mit ein.

Die gegenteilige Operation zu `cut` ist `paste`, das spaltenweise Zusammenfügen mehrerer Dateien zu einer großen Tabelle. Das Kommando `paste datei1 datei2` gibt die jeweils eine Zeile von *datei1*, einen Tabulator, eine Zeile von *datei2* und schließlich ein Newline aus. Selbstverständlich funktioniert die Operation auch mit mehr als zwei Dateien.

Mit der Option `-d<liste>` kann eine Liste von Feldtrennern angegeben werden. Das erste Zeichen der Liste wird dabei zwischen *datei1* und *datei2* eingefügt, das zweite Zeichen

Beispiel:

```
$ cut -d: -f1,3-4 /etc/passwd
root:0:1
nobody:-2:-2
agent:1:1
daemon:1:1
[ ... geloescht ... ]
$ cut -c-5 /etc/passwd
root:
nobod
agent
daemo
[ ... geloescht ... ]
```

Abbildung 1.58: Beispiele für cut

zwischen *datei2* und *datei3* und so weiter. Wenn alle Zeichen der Liste verbraucht sind, wird wieder von vorne begonnen.

Beispiel:

```
$ cut -d: -f1 /etc/passwd > /tmp/eins
$ cut -d: -f5- /etc/passwd > /tmp/zwei
$ paste /tmp/eins /tmp/zwei | tail -4
sybase  Sybase Administrator:/usr/sybase:/bin/csh
me      My Account:/nosuchdir:/bin/noshell
kris    Kristian Koehntopp:/Benutzer/kris:/bin/csh
marit   Marit Hansen:/Benutzer/marit:/bin/csh
```

Abbildung 1.59: Beispiele für paste

Im Beispiel wird das erste Feld der Paßwort–Datei ausgeschnitten und in die Datei /tmp/eins geschrieben. Das fünfte und alle folgenden Felder der Paßwort–Datei landen in der Datei /tmp/zwei. Beide Dateien werden nun ohne die Angabe eines besonderen Trennzeichens mit `paste` zusammengefügt. Also verwendet `paste` ein Tabulatorzeichen zur Trennung der Spalten.

1.11.2 expr – Berechnung von Ausdrücken

Der Befehl **expr** bekommt in seinen Parametern einen Ausdruck mitgegeben und rechnet diesen aus. Das Ergebnis wird auf der Standardausgabe ausgedruckt, sodaß es z.B. mittels Kommandoersetzung weiterverwendet werden kann. Der Ausdruck muß so geschrieben sein, daß jeder Teilausdruck von der Shell als ein einzelnes Wort erkannt wird. expr kann arithmetische Ausdrücke ausrechnen und wird demzufolge oft für einfache Berechnungen in der Shell benutzt. Außerdem hat der Befehl einige einfache Stringfunktionen, die es erlauben, Teilstrings zu erkennen und auszuschneiden sowie die üblichen logischen Operationen. Um auch im Kontext einer **if**– oder **while**–Konstruktion einsetzbar zu sein, setzt expr einen Exitstatus von 0, falls das Ergebnis der Operation weder leer noch 0 war. Falls das Ergebnis leer oder 0 war, wird dagegen ein Exitstatus von 1 angegeben. Ein Exitstatus von 2 zeigt einen Syntaxfehler im Ausdruck an.

Arithmetische Funktionen

Die häufigste Verwendung von expr ist, um einfache arithmetische Ausdrücke auszurechnen. Dazu stehen die fünf arithmetischen Operatoren der Sprache C zur Verfügung:

- Addition und Subtraktion werden als $+$ und $-$ notiert.
- Multiplikation und Division werden als $*$ und $/$ notiert. Da der Stern in der Shell zugleich ein Jokerzeichen ist, muß er bei der Verwendung in arithmetischen Ausdrücken durch Quoting geschützt werden. Das Divisionsergebnis ist dabei vom Typ *integer*, ein möglicherweise auftretender Rest wird unterschlagen.
- Der Modulo-Operator wird wie in C als % notiert. Er liefert den ganzzahligen Rest einer Division.

Ein häufiger Fehler bei der Verwendung von expr ist, den Ausdruck ohne Leerzeichen hinzuschreiben. expr erfordert unbedingt, daß die einzelnen Komponenten eines Ausdrucks einzelne Worte einer Kommandozeile sind.

Beispiel:

```
$ expr 3+4        # Ausdruck als ein Wort
3+4
$ expr 3 + 4      # Ausdruck korrekt in Worte zerlegbar
7
$ expr 3 + 4 \* 5 # Punktrechnung vor Strichrechnung
23
$ expr 10 / 3     # ganzzahliges Divisionsergebnis
3
```

Abbildung 1.60: Rechnen mit expr

Typischerweise wird expr verwendet, um Schleifenzähler zu inkrementieren oder Einheiten in der Ausgabe von Shellbefehlen anzupassen.

Beispiel:

```
$ zaehler=0
$ while [ $zaehler -le 10 ]
> do
>   read dateiname
>   mv $dateiname nummer.$zaehler
>   zaehler=‘expr $zaehler + 1‘
> done
```

Abbildung 1.61: Schleife mit expr

Im Beispiel 1.61 wird ein Zähler auf den Wert Null gesetzt. Die **while**–Schleife wird so lange durchlaufen, wie der Wert der Zählvariablen numerisch kleiner oder gleich 10 ist. Der Vergleich wird dabei vom Befehl **test** durchgeführt (Vergleiche auch Beispiel 1.64). In der Schleife wird ein Dateiname mit **read** vom Benutzer angefordert und in der Variablen *dateiname* gespeichert. Die vom Benutzer angegebene Datei wird in dann in „nummer.x“ umbenannt, wobei x der momentane Zählerwert ist. Danach wird der neue Zählerwert mit **expr** berechnet. Die Schleife wird also insgesamt elfmal für Werte von 0 bis 10 durchlaufen.

Beispiel:

```
$ du -s -k ~
231480  /Benutzer/kris
$ set -- ‘du -s -k ~‘
$ megabytes=‘expr $1 / 1024‘
$ echo "Sie belegen $megabytes Megabytes Plattenplatz."
Sie belegen 226 Megabytes Plattenplatz.
```

Abbildung 1.62: Umrechnung mit expr

expr kann auch verwendet werden, um die Einheiten der Ausgabe von Shellbefehlen umzurechnen.

Die Option –s veranlaßt den **du**–Befehl, nur eine Gesamtsumme der belegten Kilobytes in diesem Verzeichnis auszugeben. Die Option -k bewirkt die Ausgabe in Kilobytes, nicht in Plattenblöcken. Die Ausgabe von du besteht aus zwei Worten: Das erste Wort ist die Menge des belegten Platzes, das zweite Wort gibt den Namen des Verzeichnisses an, das ausgemessen worden ist.

Mit dem **set**–Kommando (siehe Abschnitt 1.9.3) wird die Ausgabe dieses Befehls in Worte zerlegt und den Argumentvariablen $1 und $2 zugewiesen. Demnach steht in $1 jetzt das erste Wort der Ausgabe von du, nämlich die Anzahl der belegten Kilobytes des $HOME-Verzeichnisses.

In der folgenden Zuweisung wird per Kommandoersetzung und **expr** der belegte Platz in Megabytes umgerechnet, um schließlich als Klartext mit **echo** ausgegeben zu werden.

Logische Operationen und Vergleiche

Mit expr sind die üblichen sechs Vergleichsoperationen möglich:

Operator	Operation
<	Echt kleiner als
<=	Kleiner oder gleich
=	Gleich
!=	Ungleich
>=	Größer oder gleich
>	Echt größer als

Abbildung 1.63: Vergleichsoperatoren von expr

Die Vergleiche sind numerisch, wenn beide Operanden numerisch sind, andernfalls findet ein lexikographischer Vergleich statt. Bei der Verwendung der Vergleichsoperatoren muß man wieder mittels Quoting darauf achten, daß die Shell keine Eingabe– oder Ausgabeumlenkungszeichen in den Operatoren erkennt. Trifft der Vergleich zu, druckt expr eine **1**, sonst wird eine **0** ausgegeben.

Zusammen mit den Operatoren | und & lassen sich auch kompliziertere Bedingungen formulieren. Beide Zeichen haben in der Shell eine Spezialbedeutung (Pipezeichen und Hintergrundprozeß) und müssen daher grundsätzlich gequoted werden. Der Ausdruck `expr a | b` liefert a, wenn a nicht leer oder 0 ist. Andernfalls wird b zurückgeliefert. Der Ausdruck `expr a & b` ergibt nur dann a, wenn weder a noch b leer oder 0 sind. Andernfalls wird 0 ausgedruckt.

Beispiel:

```
$ zaehler=0
$ while expr $zaehler '<=' 10 >/dev/null
> do
>   read dateiname
>   mv $dateiname nummer.$zaehler
>   zaehler=`expr $zaehler + 1`
> done
```

Abbildung 1.64: Schleife mit expr, Version II

Das Beispiel 1.64 ist identisch mit dem Beispiel 1.61 bis auf den Test der Schleifenbedingung. Diese wird im Beispiel 1.64 mit **expr** getestet. Um ein Verhalten zu erreichen, das mit dem anderen Beispiel übereinstimmt, muß die Ausgabe des expr–Kommandos nach `/dev/null` umgeleitet werden. Der Vergleichsoperator mußte mit Anführungszeichen vor der Interpretation durch die Shell geschützt werden.

Beispiel:

```
$ expr delta \< alpha; echo $?
0
1
$ expr alpha \< delta; echo $?
1
0
```

Abbildung 1.65: Stringvergleiche mit expr

Falls einer der beiden Vergleichsoperatoren in einem Ausdruck nicht numerisch ist, findet ein lexikographischer Vergleich statt. Im Beispiel sind zur Verdeutlichung der von expr gedruckte Text (erste Ausgabezeile) und der Exitcode (zweite Ausgabezeile) ausgedruckt.

Stringfunktionen

Für Operationen auf Zeichenketten hat expr den Operator **:** (Doppelpunkt). Das Kommando `expr string : regexp` vergleicht den ersten Ausdruck (einen String) mit dem zweiten Ausdruck. Der zweite Ausdruck wird dabei als ein regulärer Ausdruck interpretiert, der von derselben ist Form wie sie das Kommando **ed** akzeptiert . Wenn der reguläre Ausdruck auf den String paßt (match), wird die Anzahl der Zeichen ausgegeben, auf die der reguläre Ausdruck im String paßt.

Beispiel:

```
$ expr testtext : test    # Einfacher Match von 4 Zeichen
4
```

Abbildung 1.66: Der Matchoperator : von expr

Optional läßt sich ein Abschnitt des regulären Ausdrucks mit geschützten runden Klammern (in der Form \(...\) geschrieben) markieren. In diesem Fall druckt expr nicht die Anzahl der erfaßten Zeichen, sondern den Teil des Ausdrucks, der zwischen den Klammern ge-

matched wird. Auf diese Weise lassen sich Teile eines Strings mit Hilfe von Suchmustern ausschneiden.

Beispiel:

```
$ # Ausschneiden eines Teilstrings mit \( ... \)
$ expr /home/unix03/probe : '/\([a-z]*\)'
home
```

Abbildung 1.67: Ausschneiden von Teilstrings mit expr

Einige Varianten von expr haben noch weitere Stringfunktionen: index, substr und length. Dabei bestimmte der Indexoperator das erste Vorkommen eines Strings in einem anderen, der Substr–Operator schneidet ein bestimmtes Teilstueck aus einem String aus und length bestimmt die Laenge eines Strings.

1.11.3 find – Suchen von Dateien

Häufig muß man eine Liste von Dateinamen nach bestimmten Kriterien erzeugen. Gesucht sind etwa alle Dateien eines bestimmten Namens, über einer bestimmten Größe oder alle Dateien, die seit einem gewissen Zeitraum nicht mehr geändert worden sind. In UNIX steht für diesen Zweck das Kommando `find` zur Verfügung. Der Befehl hat die Syntax `find <pfadliste> <optionen>`. Er durchsucht die in der *pfadliste* angegebenen Verzeichnisse und deren Unterverzeichnisse nach Dateien, die den in den Optionen angegebenen Bedingungen genügen. Falls kein Kriterium angegeben wird, gelten alle Dateinamen in den genannten Dateibäumen als Treffer.

Mit den folgenden Kriterien kann man die Suche auf Dateien mit bestimmten Eigenschaften eingrenzen. Die Optionen sind nach der Reihenfolge der Anzeige beim `ls`–Kommando sortiert. Der Parameter *name* steht für den Namen einer Datei oder für ein Suchmuster nach den Regeln der Shell. Falls es sich um ein Suchmuster handelt, ist darauf zu achten, daß es vom `find`–Befehl und nicht von der Shell expandiert wird! Der Parameter *n* steht für eine Zahl. Dabei steht *n* für die genaue Anzahl von Vorkommen, *–n* für weniger als n Vorkommen und *+n* für mehr als n Vorkommen.

-name *datei* Dateien, deren Name auf das angegebene Suchmuster *datei* passen, zählen als Treffer.

-inum *n* Dateien mit der angegebenen I–Node–Nummer zählen als Treffer.

-type *typ* Dateien, die den angegebenen Dateityp *typ* haben, zählen als Treffer. Zugelassene Typen sind:

- **b** Die Datei ist vom Typ *block special*.
- **c** Die Datei ist vom Typ *character special*.
- **d** Die Datei ist vom Typ *directory*.
- **f** Die Datei ist vom Typ *plain file*.
- **l** Die Datei ist vom Typ *symbolic link*
- **p** Die Datei ist vom Typ *pipeline*.
- **s** Die Datei ist vom Typ *socket*.

-perm *p* Dateien mit genau den oktal angegebenen Zugriffsrechten *p* zählen als Treffer. Falls die Zugriffsrechte mit einem Minuszeichen beginnen, gelten mehr erweiterte

Zugriffsrechte (017777) und der Vergleich wird mit der Bedingung (mode&p)==p durchgeführt. Auf diese Weise kann man Dateien mit gewissen Mindestrechten finden, etwa alle SUID/SGID-Programme und andere.

-links *n* Dateien mit der angegebenen Anzahl von Links zählen als Treffer.

-user *name* Dateien, die dem angegebenen Benutzer (kann als Name oder als Benutzernummer gegeben werden) zählen als Treffer.

-nouser Dateien, die zu einer UID gehören, die nicht oder nicht mehr in der Passwort–Datei eingetragen ist, zählen als Treffer.

-group *name* Dateien, die zu der angegebenen Gruppe gehören (kann als Name oder als Gruppennummer gegeben werden) zählen als Treffer.

-nogroup Dateien, die zu einer GID gehören, die nicht oder nicht mehr in der Gruppen–Datei eingetragen ist, zählen als Treffer.

-size *n* Dateien, die die angegebene Anzahl von Plattenblöcken groß sind, zählen als Treffer.

-atime *n* Dateien, deren Datum des letzten Zugriffs die angegebene Anzahl von Tagen zurück liegt, zählen als Treffer. Das Datum des letzten Zugriffs auf Verzeichnisse wird von `find` selbst beeinflußt!

-ctime *n* Dateien, deren *change* die angegebene Anzahl von Tagen zurück liegt, zählen als Treffer. Ein *change* ist eine Änderung am Dateiinhalt oder an den Attributen (Eigentümer, Gruppe, Anzahl der Links usw.) der Datei.

-mtime *n* Dateien, deren letzter Schreibzugriff die angegebene Anzahl von Tagen zurürck liegt, zählen als Treffer.

-newer *datei* Dateien, deren Veränderungsdatum jünger ist als das Veränderungsdatum der angegebenen Datei, zählen als Treffer.

Beispiel:

```
$ find /usr/include -name ctype.h -print
/usr/include/ctype.h
$ find . -mtime -1 -print | head -4 # Ausgabe begrenzen
./shell.log
./shell.aux
./shell.dvi
./shell.toc
$ ls -l ‘find . -size +128 -print‘
-rw-r--r-- 1 kris other 155748 Nov 2 22:03 ./shell.dvi
-rw-r--r-- 1 kris other 105340 Nov 2 22:13 ./shell.tex
```

Abbildung 1.68: Beispiele für ein einfaches `find`

Im Beispiel wird zunächst im Verzeichnis `/usr/include` nach einer Datei mit dem Namen `ctype.h` gesucht. Gefundene Namen sollen ausgedruckt werden. Das zweite Beispiel sucht nach Dateien im aktuellen Verzeichnis, deren Veränderungsdatum maximal einen Tag zurück liegt. Das letzte Beispiel zeigt `find` im Kontext einer Kommandoersetzung: Gesucht werden Dateien mit einem Größe von mehr als 128 Blöcken (hier: KB). Die Namen dieser Dateien werden dann als Argument in einem `ls`–Befehl benutzt[4].

[4] Das Kommando funktioniert so nicht, wenn `find` keine Dateien findet. Was passiert dann? Was wäre, wenn statt des `ls -l` ein `rm -r` stünde und keine Dateien gefunden werden?

Die Suche nach Dateien kann durch die Angabe von mehr als einem Kriterium noch weiter eingeschränkt werden. Die einzelnen Optionen werden standardmäßig miteinander durch ein logisches Und miteinander verknüpft. Die Angabe von **-o** erlaubt eine Exklusiv–Oder–Verknüpfung (ein Entweder–Oder), die Angabe von `!` eine Negation. Mit den runden Klammern (Achtung, Quoting!) können Bedingungen gruppiert werden. `find` arbeitet schneller, wenn möglichst wenige Bedingungen ausgewertet werden müssen. Da das Kommando seine Bedingungen strikt von links nach rechts auswertet, kann es nützlich sein, eine stark einschränkende Bedingung möglichst früh, d.h. möglichst weit links zu positionieren.

Beispiel:

```
$ cores=‘find $HOME -name core -mtime +7 -print‘
$ rm -i $cores
remove /Benutzer/kris/Source/funcpoi/core? y
$ find $HOME \( -name ’*~’ -o -name ’*.bak’  \) \
>               -atime +7 \
>               -exec rm {} \; \
$ chown 755 ‘find /Benutzer/ftp/pub/ -type d \
>                                    -group public \
>                                    -print‘
```

Abbildung 1.69: Beispiele für ein zusammengesetztes `find`

Im ersten Beispiel wird eine Variable mit einer Liste aller Coredump–Dateien belegt, die seit mehr als 7 Tagen nicht mehr modifiziert worden sind. Diese Dateien werden dann mit einem `rm`–Kommando interaktiv gelöscht. Die Bedingung für das `find` lautet also: „Finde alle Dateien, die den Namen `core` tragen **und** die seit mehr als 7 Tagen nicht mehr verändert worden sind.

Im folgenden Beispiel wird nach Dateien gesucht, deren Name entweder auf eine Tilde oder `.bak` endet. Falls diese Dateien seit mehr als sieben Tagen nicht mehr gelesen worden sind, wird das Kommando `rm` für diese Dateien ausgeführt. Die Bedingung lautet also: Es werden Dateien gesucht, die **entweder** den einen **oder** den anderen Namen haben **und** die längere Zeit nicht mehr im Zugriff waren **und** für die `rm` erfolgreich ausgeführt werden kann. Die Oder–Bedingung muß geklammert werden, die Klammern und die Jokerzeichen müssen vor der Interpretation geschützt werden, damit sie vom `find` unverändert gelesen werden können. Die `-exec`–Option wird mit einem Semikolon begrenzt, das ebenfalls vor der Shell geschützt werden muß. Durch die Begrenzung auf einen bestimmten Namen werden die meisten Dateien schon sehr früh im `find` ausgeschlossen.

Das dritte Beispiel sucht nach Verzeichnissen, die der Gruppe `public` gehören. Die Zugriffsrechte dieser Dateien werden angepaßt.

Gerade in Installationen mit vielen Platten oder angemeldeten Netzwerklaufwerken möchte man die Suche evtl. auf bestimmte Dateisystemtypen einschränken oder zwecks Vermeidung von Netzwerküberlastung auf lokale Platten begrenzen. `find` stellt dazu einige Optionen zur Verfügung:

-fstype *typ* Gilt als wahr, wenn die aktuelle Datei auf einem Dateisystem des angegebenen Typs liegt. (BSD UNIX)

-xdev Bewirkt, daß die Suche im aktuellen Dateibaum Dateisystemgrenzen nicht überschreitet. (BSD UNIX)

-mount Bewirkt, daß die Suche im aktuellen Dateibaum Dateisystemgrenzen nicht überschreitet. (System V UNIX)

-prune Bewirkt, daß die Suche **nicht** in Unterverzeichnisse absteigt.

-depth Bewirkt, daß alle Einträge eines Verzeichnisses **vor** dem Verzeichnis selbst bearbeitet werden. (Tiefensuche)

Die bisher aufgeführten Optionen bewirken zwar eine Einschränkung von Suchkriterien auf Dateien mit bestimmten Eigenschaften, aber der `find`–Befehl hat keinen sichtbaren Effekt. Um eine Aktion auszulösen hat `find` eine Anzahl von weiteren Optionen:

-print Liefert immer den Wert *wahr*. Bewirkt die Ausgabe des gerade bearbeiteten Pfadnamens.

-exec *kommando* Führt das angegebene Kommando aus. Liefert das Kommando den Exitstatus 0, ergibt die Option den Wert *wahr*. Innerhalb des Kommandos werden die geschweiften Klammern durch den aktuellen Dateinamen ersetzt. Das Kommando muß mit einem Semikolon beendet werden (Achtung, Quoting!).

-ok *kommando* Funktioniert wie **-exec**, jedoch wird das Kommando zunächst auf der Standardausgabe gedruckt und eine Eingabe eingelesen. Ist die Eingabe `y`, wird das Kommando ausgeführt.

-ls Gilt immer als *wahr*. Bewirkt eine `ls`–ähnliche Ausgabe der bekannten Informationen über diese Datei. Die Ausgabe wird jedoch von `find` intern abgewickelt, ohne einen externen Prozeß zu starten. (BSD UNIX)

-cpio *device* Gilt immer als *wahr*. Bewirkt die Ausgabe der aktuellen Datei als `cpio`–Archiv auf der Standardausgabe (5120 Byte Records).

-ncpio *device* Gilt immer als *wahr*. Bewirkt die Ausgabe der aktuellen Datei als `cpio-c`–Archiv auf der Standardausgabe (5120 Byte Records).

Beispiel:

```
$ find /Benutzer -user kris -depth -print |
> cpio -ocv |
> dd of=/dev/rst0
$ find / -name bla -print -o -fstype nfs -prune
```

Abbildung 1.70: Noch mehr Beispiele für `find`

Das erste Beispiel sucht alle Dateien, die dem Benutzer `kris` gehören und druckt deren Namen aus. Dabei werden Verzeichnisse *nach* den darin enthaltenen Dateien genannt. Diese Dateiliste ist die Eingabe für das Datensicherungskommando `cpio`, das daraus ein Backup macht. Das Backup wiederum wird mittels `dd` auf dem Bandlaufwerk `/dev/rst0` gesichert. Durch das Einschalten der Tiefensuche wird das Zurückspielen von schreibgeschützten Verzeichnissen erst möglich: Die Zugriffsrechte an einem Verzeichnis werden beim Wiedereinlesen des Bandes erst dann gesetzt, wenn alle Dateien im Verzeichnis installiert sind.

Das zweite Beispiel sucht nach Dateien, die **entweder** den angegebenen Namen haben **oder** auf einem NFS–Dateisystem liegen. Falls das Letztere der Fall ist, wird die Suche im NFS–Dateibaum nicht weiter fortgesetzt. Auf diese Weise schränkt man eine Suche auf die lokalen Platten ein und vermeidet so exzessive Netzbelastung.

1.11.4 getopt, getopts – Auswerten der Kommandozeile

In Abschnitt 1.1 wurden einige Regeln vorgestellt, denen ein UNIX–Kommando bei der Auswertung seiner Optionen folgenden sollte. Auf diese Weise soll eine Einheitlichkeit in der Auswertung der Kommandozeile erreicht werden, die den Umgang mit Kommandos für den Benutzer vereinfacht. UNIX ist ein sehr altes Betriebssystem und nicht alle Kommandos genügen den Anforderungen. Um wenigstens die Erstellung neuer Kommandos für den Programmierer zu vereinfachen, stellt UNIX die Bibliotheks `getopts(3C)` für den C–Programmierer zur Verfügung.

Für die Autoren von Shellscripten war bisher der inzwischen veraltete externe Befehl `getopt(1)` das Werkzeug der Wahl. `getopt` wird normalerweise auf eine ganz bestimmte Weise am Anfang von Shellscripten verwendet, um die Optionen des Scriptes einzulesen und zur späteren Auswertung zur Verfügung zu stellen. Dazu werden dem Kommando eine Liste der Optionsbuchstaben, die das Script kennen soll und die kompletten Parameter zur Verfügung gestellt. `getopt` liest die Kommandozeile des Shellscriptes durch und stellt sie reformatiert zur Verfügung.

Zur Illustration sei ein Kommando `listuser` angenommen, das durch ein Shellscript realisiert werden soll. Das Script soll die Optionen `-s` (für *short list*) und `-l` (für *long list*) verstehen, die sich gegenseitig ausschließen. Es bezieht seine Informationen standardmäßig aus der Datei `/etc/passwd`, wenn nicht nach der Option `-f` (für *file*) eine andere Datei angeben ist. Uns soll hier weniger interessieren, wie die Benutzerliste erzeugt wird als wie das Script seine Optionen einliest.

Beispiel:

```
$ getopt slf: -sl -f /etc/passwd.saved
-s -l -f /etc/passwd.saved --
$ getopt slf: -lf bla blafasel
-l -f bla -- blafasel
$ getopt slf: -l -x -f urgs
getopt: illegal option -- x
-l -f urgs --
$ echo $?
1
$ getopt slf: -f
getopt: option requires an argument -- f
```

Abbildung 1.71: `getopt` normalisiert die Liste der Optionen

Der `getopt`–Befehl erhält als ersten Parameter eine Liste von Optionsbuchstaben, die das Script verstehen soll. Optionsbuchstaben, die einen Parameter nach sich ziehen, werden durch einen Doppelpunkt markiert. In unserem Beispiel ist die Liste der Optionen also `slf:`, wobei die Reihenfolge jedoch keine Rolle spielt: `lf:s` und `f:sl` wären genauso gültig. Die folgenden Parameter des Kommandos sind die Originalparameter des Shellscriptes, angegeben durch $*. `getopt` liest diese Parameter ein und sortiert sie in einzelne Optionen und weitere Angaben auseinander. Optionen werden in der Ausgabe durch einen Doppelstrich `--` von den weiteren Angaben getrennt.

`getopt` erkennt illegale Optionen und meldet diese ebenso wie fehlende Parameter von Optionen mit Parametern. Eine fehlerhafte Kommandozeile wird durch einen entsprechenden Exitstatus vermerkt. Auf der anderen Seite wird kein Versuch unternommen, Optionen, die sich gegenseitig ausschließen zu erkennen und entsprechend zu trennen.

Normalerweise wird man die Ausgabe von `getopt` mit `set` wiederum in einzelne Worte zerlegen, die man dann schrittweise abarbeitet. Die Manualpage zum Kommando empfiehlt zu diesem Zweck eine bestimmte Konstruktion (siehe Beispiel 1.72).

Beispiel:

```
# Optionen lesen und in Worte zerlegen
set -- ‘getopt slf: $*‘
# Falls ein Fehler aufgetreten ist, Hilfe anzeigen und Ende
if [ $? != 0 ]
then
  echo "usage: $0 -s -l -f passwdfile"
  exit 2
fi

# Standardwerte annehmen
LISTFORM=standard
FILE=/etc/passwd
# Optionen auswerten und bei -- abbrechen.
for i in $*
do
    case $i in
      -s) LISTFORM=short; shift ;;
      -l) LISTFORM=long;  shift ;;
      -f) FILE=$2;        shift 2;;
      --)                 shift; break;;
    esac
done
```

Abbildung 1.72: Empfohlene Konstruktion zum Gebrauch von `getopt`

Im Beispiel werden die Optionen des Scriptes von `getopt` gelesen und normalisiert. Durch das `set` werden die normalisierten Optionen auf die Kommandozeilenparameter verteilt. Sollte ein Fehler aufgetreten sein, wird eine Hilfe ausgegeben und das Script mit `exit` beendet. Nachdem alle entscheidenden Variablen mit Standardwerten belegt sind, werden in der `for`–Schleife die Kommandozeilenparameter durchgesehen.

Im Fall von einfachen Optionen wie `-s` und `-l` wird einfach ein Schalter auf den richtigen Wert gesetzt. Werden beide Optionen angegeben, wird kommentarlos der letzte der beiden Werte benutzt. Alternativ könnte man diesen Fall auch mit einem `if` erkennen und einen Fehler melden. In jedem Fall wird der Kommandozeilenparameter nach dem Lesen mit `shift` verworfen. Im Fall von Optionen mit Parametern steht der Parameter der Option durch das ständige shiften an zweiter Stelle und kann dort einfach in eine benannte Variable übernommen werden. Danach werden sowohl die Option als auch der Parameter durch ein `shift 2` verworfen. Die Liste der Optionen in auf alle Fälle durch ein `--` beendet. Hier brauchen gar keine Variablen besetzt werden: Die Endemarke kann durch ein einfaches `shift` verworfen werden und die `for`–Schleife wird durch ein `break` zwangsweise beendet. Die eventuell vorhandenen sonstigen Argumente des Scriptes stehen jetzt als $1 bis $x zur Verfügung.

Im Probelauf im Beispiel 1.73 erkennt man, daß das Script so ganz gut funktioniert, solange man nicht beim Parameter von `-f` Dateinamen angibt, die Leerzeichen enthalten. In diesem Fall gehen gruppierenden Anführungszeichen um den Parameter leider beim getopt verloren und der Dateiname wird in zwei Worte aufgebrochen. Anstatt den `getopt`–Befehl so zu korrigieren, daß er die Anführungszeichen um Parameter mit Leerzeichen reproduziert hat

Beispiel:

```
$ listuser einfach
Der Inhalt von $LISTFORM ist "standard"
Der Inhalt von $FILE ist "/etc/passwd"
Die weiteren Argumente sind:
einfach
$ listuser -l -f bla zwei weitere
Der Inhalt von $LISTFORM ist "long"
Der Inhalt von $FILE ist "bla"
Die weiteren Argumente sind:
zwei
weitere
$ listuser -x hoppla
getopt: illegal option -- x
usage: ./listuser -s -l -f passwdfile
$ listuser -f "problem file" "panne zwei"
Der Inhalt von $LISTFORM ist "standard"
Der Inhalt von $FILE ist "problem"
Die weiteren Argumente sind:
file
panne
zwei
$ getopt slf: -f "problem file" "panne zwei"
-f problem file -- panne zwei
```

Abbildung 1.73: Probelauf des Scriptes *listuser*

man die sicherere Lösung gewählt und das Kommando durch den in die Shell eingebauten Befehl `getopts` ersetzt. Da `getopts` ein shellinternes Kommando ist, kann es Shellvariablen direkt belegen und so gleich noch ein anderes, unschönes Merkmal von `getopt` vermeiden, nämlich die Zerstörung der Originalkommandozeile.

Dem `getopts`–Kommando muß genau wie seinem externen Vorläufer eine Liste der zugelassenen Optionen mitgegeben werden. Als internem Befehl steht ihm jedoch die Kommandozeile sowieso zur VerfÜgung, sodaß diese nicht mehr angegeben werden muß. Der Befehl wird anders als `getopt` mehrfach aufgerufen und legt mit jedem Aufruf jeweils eine Option in einer Shell–Variablen ab, deren Name als zweiter Parameter anzugeben ist. In der Variablen **OPTIND** wird mitgezählt, das wievielte Wort der Kommandozeile bearbeitet wird und in der Variablen **OPTARG** wird ein Optionsparameter hinterlegt, falls die betreffende Optionen einen solchen erforderlich macht. Wenn eine illegale Option vorgefunden wird, stellt der Befehl ein Fragezeichen in die Optionsvariable. Falls nach der Analyse der Optionen diese verworden werden sollen, kann dies durch ein berechnetes `shift` geschehen, wie im Beispiel gezeigt.

Das interne `getopts` hat gegenüber dem externen `getopt` den Vorteil der Geschwindigkeit, der korrekten Behandlung von Parametern mit Leerzeichen im Namen und es ist obendrin der empfohlene Weg, Optionen in Shellscripten einzulesen und verarbeiten. Andererseits ist `getopts` allerdings recht neu und nur auf UNIX System V Release 3 und neuer vorhanden. Wer auf einem älteren System arbeitet oder darauf angewiesen ist, portable Scripte zu schreiben, muß mit dem anfälligeren `getopt` vorlieb nehmen.

Beispiel:

```
#! /bin/sh --
# Syntax des getopts-Befehles:
#
# getopts optionsliste variablenname
#
FILE=/etc/passwd
LISTFORM=standard
while getopts slf: opt
do
    case $opt in
      s) LISTFORM=short ;;
      l) LISTFORM=long ;;
      f) FILE=$OPTARG ;;
      \?) echo "usage: $0 -s -l -f passwdfile"
          exit 2
         ;;
    esac
done
shift ‘expr $OPTIND - 1‘
```

Abbildung 1.74: Syntax von `getopts` und Beispiel

1.11.5 grep, egrep, fgrep und bm – Durchsuchen von Dateien

Zum Suchen von Texten in Dateien steht in UNIX eine ganze Familie von Befehlen zur Verfügung. Volltextsuche kann eine ziemlich komplizierte Aufgabe sein, insbesondere dann, wenn große Textmengen zu durchsuchen sind oder der Suchbegriff ein komplexer regulärer Ausdruck ist. Ein einzelner Befehl reicht dann nicht aus, um das ganze Spektrum der vorhandenen Algorithmen abzudecken.

Der Befehl `bm` durchsucht eine Textdatei nach einem oder mehreren konstanten Strings. Die String selber dürfen dabei keine regulären Ausdrücke enthalten und es kann nicht einmal die Unterscheidung von Groß- und Kleinschreibung abgeschaltet werden. Der Befehl verwendet für die Suche jedoch den extrem schnellen Algorithmus von Boyer und Moore. Der Befehl ist deswegen um so schneller, je länger der gesuchte Text ist. Er schlägt die Familie der `grep`–Befehle um Längen.

Die Befehle der `grep`–Gruppe sind dagegen sehr viel flexibler als `bm`. Diese Flexibilität muß jedoch mit einem Geschwindigkeitsnachteil erkauft werden. Der Befehl `grep` kann als Suchbegriff dieselben regulären Ausdrücke verarbeiten, die auch der Editor `ex` versteht. Der Befehl braucht vergleichsweise wenig Speicher. `egrep` dagegen verwendet einen anderen Algorithmus, der erweiterte reguläre Ausdrücke versteht, der in einigen Fällen jedoch extrem viel Speicherplatz benötigt. `fgrep` sucht nach konstanten Strings, es ist schnell und kompakt.

Man hat sich bemüht, die Optionen der einzelnen Befehle so weit wie möglich einheitlich zu gestalten. In der folgenden Liste ist jeweils markiert, welche Option von welchem Befehl verstanden wird. Dabei kennzeichnet ein *b* Optionen, die beim Kommando `bm` verfügbar sind, ein *e* markiert `egrep`–Optionen, ein *f* steht für `fgrep` und ein *g* zeigt Optionen an, die nur bei `grep` vorhanden sind.

-b *efg* Jeder Fundstelle wird eine Nummer vorangestellt, die angibt, im wievielten Plattenblock in einer Datei die Fundstelle sich befindet.

Beispiel:

```
cat konfigurationsdatei |
grep -v "^[ \t]*#" |
auswertung
```

Abbildung 1.75: Ausfiltern von Kommentarzeilen mit `grep`

-c *befg* Die Anzahl der Fundstellen wird gezählt und nur der Zählerstand wird ausgedruckt.

-e *suchbegriff befg* Entspricht der einfachen Angabe von *suchbegriff*. Auf diese Weise können auch Suchbegriffe angegeben werden, die mit einem Minuszeichen beginnen.

-f *datei bef* Der Suchbegriff bzw. eine Liste von Suchbegriffen wird aus der Datei *datei* gelesen.

-h *b* Es werden keine Dateinamen gedruckt, selbst wenn mehrere Dateien durchsucht werden.

-i *fg* Beim Vergleich werden Groß- und Kleinschreibung von Worten nicht beachtet. Bei einigen alten UNIX–Versionen war dies die Option **-y**.

-l *befg* Statt die Fundstellen auszugeben, werden nur die Namen der Dateien mit mindestens einer Fundstelle ausgegeben.

-n *befg* Jeder Fundstelle wird die Zeilennummer (gerechnet vom Beginn der jeweiligen Datei) vorangestellt.

-s *befg* Der Befehl druckt nichts, sondern erzeugt nur einen Exitstatus.

-v *efg* Negation. Alle Zeilen, die keine Fundstelle enthalten, werden gefunden.

-w *g* Wortweise Suche. Das Suchwort wird nur erkannt, wenn es ein einzelnstehendes Wort ist.

-x *bf* Nur Zeilen, auf die der Suchbegriff exakt und vollständig zutrifft, werden gefunden.

1.11.6 head und tail – Abschneiden von Dateienden

Oftmals benötigt man die Befehle `head` oder `tail` um den Anfang oder das Ende einer Datei zu finden. Beide Kommandos nehmen eine Option in der Form `-zahl` um die Anzahl der Zeilen zu bestimmen, die vom Anfang bzw. Ende der Datei gerechnet ausgegeben werden sollen.

Der Befehl `tail` kann außerdem mit der Option `+zahl` eine Anzahl vom Zeilen gerechnet vom Anfang der Datei ausgeben. Mit der Option `-f` kann man den Befehl das Ende eine Datei beobachten lassen. Wann immer die Datei wächst, werden die neu hinzugekommenen Zeilen ausgegeben. Auf diese Weise kann man z.B. eine Logdatei überwachen lassen.

Im Beispiel 1.76 wird das Kommando `head` benutzt, um die erste Zeile einer Konfigurationsdatei zu extrahieren. Der Rest der Datei ohne die erste Zeile wird in eine neue Datei kopiert, danach wird die ehemalige erste Zeile hinten an die Datei angefügt. Auf diese Weise entsteht ein ringförmig umlaufender Puffer von Datensätzen. Im Beispiel soll dieser Mechanismus dafür benutzt werden, jeden Tag einen anderen Satz von Dateien zu sichern.

Im Beispiel 1.77 wird das Programm `uucico` im Hintergrund gestartet. Während es seine Arbeit tut, hinterlegt es Statusinformationen in der Datei `Log`. Diese Datei wird permanent

Beispiel:

```
#! /bin/sh --

BACKUPCONFIG=/usr/root/Cronjobs/config.backup
LOGFILE=/usr/root/Cronjobs/system.activities

TODAY=‘head -1 $BACKUPCONFIG‘

tail +2 $BACKUPCONFIG > $BACKUPCONFIG.new
echo $TODAY >> $BACKUPCONFIG.new
mv $BACKUPCONFIG.new $BACKUPCONFIG

echo "‘date‘: Heutiges Backup: $TODAY" >> $LOGFILE
dobackup $TODAY
echo "‘date‘: Backup von $TODAY beendet." >> $LOGFILE
```

Abbildung 1.76: Umschichten einer Datei

Beispiel:

```
/usr/lib/uucp/uucico -r1 -Stpki &
tail -f /usr/spool/uucp/Log
```

Abbildung 1.77: Überwachen einer Logdatei

durch den `tail`–Befehl überwacht. Neu hinzukommende Zeilen werden sofort ausgegeben. Der Befehl endet nicht von allein, sondern muß mit Control–C abgebrochen werden.

Auf einen anderen netten Trick greifen einige UNIX–Programmierer zurück, um schnelle Schleifen programmieren zu können. Dieser Trick ist jedoch nicht unbedingt portabel, da nicht alle UNIX-Versionen einen `cat`–Befehl haben, der die Option `-n` versteht um Zeilennummern zu erzeugen. Im Beispiel 1.78 erzeugt der Befehl `yes` eine endlose Liste von leeren Zeilen. Der Befehl `cat -n` zählt diese Leerzeilen und erzeugt so Zeilennummern, denen jedoch kein Text folgt (da es sich ja um Leerzeilen handelt). Die so erzeugte Liste von Zahlen ist jedoch endlos. Möchte man, daß die Zählerei nach einem bestimmten Zählerstand abbricht, muß man etwa mit dem `head`–Kommando die gewünschte Zahl Zeilen abschneiden.

Warum aber endet die Pipeline, wenn der `head`–Befehl die gewünschte Anzahl Zeilen gesehen hat? Nun, wenn `head` endet, schreibt der Befehl `cat` in eine Pipeline ohne lesenden Prozeß. Er empfängt ein SIGPIPE, das den Prozeß beendet. Daraufhin schreibt jedoch auch `yes` in eine Pipeline ohne lesenden Prozeß und wird ebenfalls mit einem SIGPIPE abgebrochen. Durch das Ende des letzten Prozesses in einer Pipeline wird die Pipe also von hinten nach vorne abgebaut und es kommt nicht zu endlos laufenden Prozessen.

1.11.7 sort – Sortieren von Dateien

Mit dem UNIX-Kommando `sort` stellt das Betriebssystem einen Universalbefehl zum Sortieren von Dateien zur Verfügung. Das Kommando kann Dateien mit unterschiedlichen Feldtrennzeichen nach den verschiedensten Kriterien sortieren. Normalerweise sortiert das Kommando die Zeilen einer Datei in aufsteigender Reihenfolge. Der Sortierschlüssel wird dabei von der gesamten Zeile gebildet und sortiert wird nach der ASCII–Reihenfolge.

Beispiel:

```
$ yes "" | cat -n |head -5
     1
     2
     3
     4
     5
$ summe=0
$ for i in ‘yes "" | cat -n | head -5‘
> do
> summe=‘expr "$summe" + $i‘
> done
$ echo $summe
15
```

Abbildung 1.78: Schnelle Schleifen mit `head` und `for`

Die Sortierreihenfolge kann jedoch mit einer der nachstehenden Optionen beeinflußt werden:

-b Wenn diese Option gesetzt ist, werden führende Leerzeichen in einem Feld ignoriert.

-d Die Sortierreihenfolge wird auf eine Wörterbuchreihenfolge (*dictionary*) gesetzt: Nur Buchstaben, Ziffern und Leerzeichen sind für den Vergleich relevant.

-f Beim Vergleich wird Groß– und Kleinschreibung nicht mehr unterschieden (*fold case*).

-i Die Steuerzeichen außerhalb des Bereiches der druckbaren Zeichen von ASCII 32-127 werden ignoriert.

-n Der Vergleich erfolgt numerisch statt alphabetisch. Zahlen werden also korrekt nach ihrem Wert einsortiert.

-r Die Sortierreihenfolge wird umgedreht (*reverse*).

-t *x* Das Trennzeichen zwischen den Feldern einer Zeile wird auf *x* gesetzt. Zusammen mit den Optionen zur Einschränkung des Vergleichs auf Teile der Zeile kann so nach unterschiedlichen Schlüsseln sortiert werden.

Normalerweise sortiert der Befehl die Zeilen einer Datei, indem die Zeilen einfach ganze Zeilen miteinander verglichen werden. Das funktioniert jedoch nur, wenn die Sortierschlüssel in der richtigen Reihenfolge von links nach rechts vom Anfang der jeweiligen Zeile aufgeführt worden sind. Ist dies nicht der Fall, kann man mit der Option `+pos1` den Beginn eines Schlüsselfeldes festlegen und mit `-pos2` das Ende dieses Feldes angeben. Die Angabe eines Feldendes ist dabei optional. Wenn sie fehlt, geht das Schlüsselfeld bis zum Ende der Datei.

Im Beispiel 1.79 wird eine unsortierte Datei *nummern* zunächst alphabetisch sortiert. In diesem Fall stehen die zweistelligen Zahlen in numerisch falscher Reihenfolge am Anfang der Datei. Erst nach der Angabe der Option `-n` wird numerisch korrekt sortiert. Die Ergebnisse der verschiedenen Sortierläufe wurden aus Platzgründen mit dem Kommando `paste` nebeneinander plaziert.

Im zweiten Beispiel wird gezeigt, wie sich die Optionen `-b` und `-f` auf die Sortierreihenfolge auswirken: Schreibweise und führende Leerzeichen spielen keine Rolle mehr. Damit

Beispiel:

```
$ sort nummern > nummern.falsch
$ sort -n nummern > nummern.korrekt
$ paste nummern nummern.falsch nummern.korrekt
19      13      4
16      16      7
7       19      8
8       4       13
13      7       16
4       8       19
$ sort salat > salat.normal
$ sort +0 -bf salat > salat.spezial
$ paste salat salat.normal salat.spezial
testwort         TESTWORY       GNURZELwurm
TestWorx        GNURZELwurm     testwort
 TESTWORY       TestWorx        TestWorx
GNURZELwurm     testwort         TESTWORY
```

Abbildung 1.79: Einfaches Sortieren von Dateien

die Optionen wirksam werden können, muß `sort` mit der Option `+0` gezwungen werden, die Zeilen der Datei nicht als ganze Zeilen zu unterteilen, sondern in Felder zu unterteilen.

Standardmäßig ist das Feldtrennzeichen dabei eine Folge von Leerzeichen und Tabulatoren. Mit der Option `-t` kann jedoch ein anderes Feldtrennzeichen angegeben werden.

Durch mehrfache Angabe von Sortierschlüsselanfängen und –Enden kann nach mehreren Feldern sortiert werden. Im Beispiel 1.80 wird etwa die Paßwortdatei eines Systems nach der Gruppennummer und innerhalb von Einträgen mit gleicher Gruppennummer nach der Benutzernummer sortiert. Dazu wird der Doppelpunkt als Feldtrennzeichen angegeben und es wird verlangt, numerisch zu sortieren. Die Gruppennummer beginnt mit dem dritten Feld der Paßwortdatei und endet vor dem vierten Feld. Als zweiter Sortierschlüssel wird das zweite, durch Doppelpunkte markierte Feld angegeben. Der zweite Schlüssel endet vor dem dritten Feld.

Die vollständige Syntax für ein solches Schlüsselbegrenzerfeld lautet `-m.nx`. Dabei steht *m* für die Anzahl der Felder vom Beginn der Zeile gerechnet, die zu überspringen sind. *n* gibt an, wieviele Buchstaben vom Anfang des Feldes an gerechnet zu überspringen sind. Fehlt die Angabe, wird sie als 0 angenommen. Zusätzlich kann mit *x* ein besondere Sortierreihenfolge für dieses spezielle Feld angegeben werden. Dabei darf *x* einen der Werte *bdfinr* haben, die Wirkung entspricht den oben aufgeführten Optionen.

`sort` außerdem noch einige Optionen, die sich nicht mit der Sortierreihenfolge beschäftigen, sondern das Verhalten des Befehls an sich beeinflussen.

-c Mit der Option *check* überprüft `sort` ob eine Datei sortiert entsprechend den angegebenen Regeln ist. Wenn dies der Fall ist, wird keine Ausgabe gemacht.

-m Mit der Option *merge* kann das Kommando benutzt werden, mehrere bereits sortierte Dateien zu einer sortierten Datei zu verbinden.

-o *datei* Anstatt auf der Standardausgabe auszugeben, druckt das Kommando in die angegebene Datei *datei*. Die Datei darf dabei ausdrücklich eine der angegebenen Eingabedateien sein.

Beispiel:

```
$ sort -t: -n +3 -4 +2 -3 /etc/passwd
nobody:*:-2:-2::/tmp:
root:x:0:1:Operator:/:/bin/csh
agent:*:1:1::/tmp:
daemon:*:1:1::/tmp:
uucp:*:4:4::/usr/spool/uucppublic:/usr/lib/uucp/uucico
sybase:*:8:8:Sybase Administrator:/usr/sybase:/bin/csh
marit:x:100:20:Marit Hansen:/Benutzer/marit:/bin/csh
kris:x:102:20:Kristian Koehntopp:/Benutzer/kris:/bin/csh
schule1:*:500:20:UNIX Schulung 1:/nosuchdir:/bin/noshell
schule2:*:501:20:UNIX Schulung 2:/nosuchdir:/bin/noshell
schule3:*:502:20:UNIX Schulung 3:/nosuchdir:/bin/noshell
schule4:*:503:20:UNIX Schulung 4:/nosuchdir:/bin/noshell
schule5:*:504:20:UNIX Schulung 5:/nosuchdir:/bin/noshell
schule6:*:505:20:UNIX Schulung 6:/nosuchdir:/bin/noshell
news:x:101:100:USENET of black:/Benutzer/news:/bin/csh
ftp:*:995:101:FTP Archive Owner:/Benutzer/ftp:/bin/noshell
```

Abbildung 1.80: Ein komplizierterer Sortiervorgang

-T `verzeichnis sort` legt beim Sortieren einige Zwischendateien an. Normalerweise werden diese in einem der Verzeichnisse `/usr/tmp` oder `/tmp` angelegt. Mit der Option können diese Zwischendateien in ein anderes Verzeichnis verlagert werden.

-u Aufeinanderfolgendes gleiche Zeilen werden in der Ausgabe unterdrückt. Für den Vergleich auf Gleichheit werden nur die ggf. angegebenen Schlüsselfelder zu Rate gezogen.

1.11.8 test – Prüfen von Bedingungen

In Verzweigungen mit `if` und in Schleifen mit `while` ist das Kommando `test` praktisch unentbehrlich. Es ist ein universelles Vergleichskommando für Zahlen und Strings in der Shell und es kann außerdem noch viele Eigenschaften an Dateien überprüfen. In der Tat wird `test` so häufig benutzt, daß es in einigen UNIX–Kommandointerpretern mittlerweile aus Geschwindigkeitsgründen als internes Kommando zur Verfüng steht. Vielfach wird `test` unter dem Aliasnamen `[` (eckige Klammer auf) aufgerufen. In diesem Fall muß am Ende des Kommandos eine schliessende eckige Klammer stehen.

`test` stellt zwei verschiedene Operatoren für den Stringvergleich und die üblichen sechs Vergleiche für numerische Vergleiche zur Verfügung.

Weiterhin hat man mit `test` die Möglichkeit, Zugriffsrechte an Dateien und den Typ von Dateien zu überprüfen. Die entsprechenden Abfragen haben die Form von Optionen. So testet die Anweisung `test -r datei` etwa, ob eine Datei mit dem Namen `datei` existiert und Leserecht für diese Datei gegeben ist.

1.11.9 xargs – Kommandos synthetisieren

Beispiel:

Operator	Wirkung
=	Gleichheit von Strings
!=	Ungleichheit von Strings
-z	String ist der Leerstring
-n	String ist nicht der Leerstring
-eq	numerische Gleichheit
-ne	numerische Ungleichheit
-lt	< (echt kleiner)
-le	<= (kleiner gleich)
-ge	>= (größer gleich)
-gt	> (echt größer)

Abbildung 1.81: Übersicht: Vergleichsoperatoren von `test`

Beispiel:

Operator	Wirkung
-r	r–Recht vorhanden
-w	w–Recht vorhanden
-x	x–Recht vorhanden
-u	SUID Bit gesetzt
-g	SGID Bit gesetzt
-k	Sticky–Bit gesetzt
-f	Datei ist eine normale Datei
-d	Datei ist ein Verzeichnis
-b	Datei ist ein Blockdevice
-c	Datei ist ein Characterdevice
-h	Datei ist ein Symlink
-s	Datei ist länger als 0 Byte

Abbildung 1.82: Übersicht: Abfrage von Dateiattributen mit `test`

Inhaltsverzeichnis

Abbildungsverzeichnis