KAPITEL 6

Quoting

Die Shell beansprucht leider eine sehr große Zahl von Sonderzeichen für sich. Dies kann zu Schwierigkeiten führen, sobald eines dieser Sonderzeichen im Klartext verwendet werden soll:

```
$ echo Der Umrechnungskurs für 1 Euro: $2,30
Der Umrechnungskurs für 1 Euro: ,30
```

Diese scheinbare Wirtschaftskrise in Europa entstand nur, weil die Shell das Dollarzeichen interpretierte und deshalb den Wert der Positionsvariable *$2* einfügte. Die Lösung dieses Problems haben Sie bereits zu Anfang dieses Buchs kennengelernt:

```
$ echo 'Der Umrechnungskurs für 1 Euro: $2,30'
Der Umrechnungskurs für 1 Euro: $2,30
```

Man nennt diese Art, die Bedeutung von Sonderzeichen auszuschalten, »Quoting«, in deutschsprachigen Büchern auch manchmal »Abdecken« genannt (ich verwende nur sehr ungern den deutschen Ausdruck, da dieser im zoologischen Bereich negative Assoziationen hervorruft). Die Shell stellt hierzu folgende Methoden zur Verfügung:

- Quoting eines einzelnen Zeichens durch Voranstellen von \
- Quoting einer Zeichenkette durch Klammerung mit "..."
- Quoting einer Zeichenkette durch Klammerung mit '...'

6.1 Voranstellen von \

Ein einzelnes Zeichen kann mit Hilfe des Rückstriches (»\«, engl. Backslash) vor der Interpretation der Shell geschützt werden:

```
$ echo \$3,20
$3,20
```

Beispiele:

```
$ ls
hund
katze
maus
$ ls * > result_of_\*
$ cat result_of_\*
hund
katze
maus
$ ls \? > result_of_\?
ls: ?: No such file or directory.
$ cat result_of_\?
$ ls result_of_?
result_of_*   result_of_?
$ rm result_of_\?
$ ls result_of_*
result_of_*
$ rm result_of_?
$ ls result_of_?
ls: ?: No such file or directory.
```

Neben den Metazeichen verbirgt der Rückstrich auch das »Newline-Zeichen«. Normalerweise endet eine Befehlszeile mit dem Drücken der Return-Taste, also mit dem »Newline-Zeichen«, ist jedoch das letzte Zeichen in der Zeile ein Backslash, erwartet die Shell weitere Eingaben aus der nächsten Zeile. Dies wirkt sich folgendermaßen aus: An der Eingabeaufforderung gibt die Shell in diesem Fall das zweite Prompt *$PS2* aus und erwartet den Rest der Zeile, in Shellskripten wird einfach die nächste Zeile als Fortsetzung dieser Zeile angesehen. Sehr lange Konstrukte können mit »\« über mehrere Zeilen übersichtlich verteilt werden.

Beispiele:

```
$ echo Der satanarchäolügenial\
> kohöllische Wunschpunsch
Der satanarchäolügenialkohöllische Wunschpunsch
$ ec\
> ho Hallo Welt!
Hallo Welt!
$ cdrecord dev=0,0,0 speed=4 -v niewiederkunst.wav\
> woistderkaiser.wav kanndennschwachsinn\
> sündesein.wav burli.wav küssdiehandherrkerker\
> meister.wav # ist was für Linux-Freaks!
```

6.2 Klammerung mit ...

Bereits seit Anfang dieses Buchs machten wir von den doppelten Anführungszeichen regen Gebrauch, sobald mehrere Zeichen zusammenzufassen zu waren, jedoch bin ich Ihnen eine genaue Erklärung über die Hintergründe noch schuldig geblieben. Das wollen wir hier nachholen.

Innerhalb der doppelten Anführungszeichen verlieren so gut wie alle Metazeichen ihre Bedeutung. Ausnahmen stellen Kommandoersetzungen, Variablensubstitutionen und erweiterte Variablenausdrücke dar, sie werden von der Shell interpretiert. Sollen die durch diese Einschränkung betroffenen Metazeichen ebenfalls geschützt werden, muß dies über den Rückstrich (»\«) geschehen. Der Rückstrich besitzt allerdings nur bei den Sonderzeichen seine Bedeutung, die für die angegebenen Konstrukte notwendig sind. Das sind also »`« und *$* sowie »\« und »"«.

Beispiele:

```
$ echo Hallo Welt > x
$ echo > xx
$ echo "Mit * und ? können Sie mehrere Dateien erreichen."
Mit * und ? können Sie mehrere Dateien erreichen.
$ echo Mit * und ? können Sie mehrere Dateien erreichen.
Mit x xx und x können Sie mehrere Dateien erreichen.
$ PATH="$PATH:/data/mp3 dateien"
$ echo Information > "weitere Informationen"
$ ls -C
x   xx   weitere Informationen
$ echo "Der Inhalt von Datei \"x\": `cat x`"
Der Inhalt von Datei "x": Hallo Welt
$ echo "Bash-Version: ${BASH:-"Dies ist keine Bash"}"
Bash-Version: Dies ist keine Bash
$ echo "1\+1=2"
1\+1=2
```

Eine Besonderheit der doppelten und einfachen Anführungszeichen gilt es noch zu klären: Ein durch Hochkommata geklammerter Ausdruck kann sich auch über mehrere Zeilen erstrecken. Die dabei auftretenden Zeilenumbrüche bleiben als solche in der Zeichenkette erhalten:

```
$ Z="MENÜ: a) StarOffice
      b) GOffice
      c) KOffice"
$ echo $Z
MENÜ: a) StarOffice
      b) GOffice
      c) KOffice
```

Befindet sich vor dem Zeilenumbruch ein Rückstrich, ignoriert die Shell wie gehabt das Newline-Zeichen:

```
$ Z="MENÜ: a) StarOffice\
      b) GOffice\
      c) KOffice"
$ echo $Z
MENÜ: a) StarOffice       b) GOffice       c) KOffice
```

6.3 Klammerung mit '...'

Die einfachen Anführungszeichen schützen die Zeichen innerhalb der Hochkommata komplett vor der Interpretation der Shell. Auch eventuell auftretende Zeilenumbrüche können nicht durch »\« verdeckt werden:

```
$ echo '${ARRAY[0]}'
${ARRAY[0]}
$ X='$x'
$ echo '$X='$X
$X=$x
$ echo 'Alle eMail-Adressen aus allen Dateien können Sie mit
folgendem Ausdruck finden:
egrep " ?.*@.* " `find -name "*" -print`'
Alle eMail-Adressen aus allen Dateien können Sie mit
folgendem Ausdruck finden:
egrep " ?.*@.* " `find -name "*" -print`
$ echo 'Die einfachen Anführungszeichen '\''...\'' sind rigoros.'
Die einfachen Anführungszeichen '...' sind rigoros.
```

In der letzten Zeile versuchten wir innerhalb von »'...'« die einfachen Hochkommata zu verwenden und behalfen uns damit, das Quoting vor dem auszugebenden »'« zu beenden und danach wieder zu beginnen. In den Originalversionen der Korn-Shell gibt eine etwas einfachere Lösung dieses Problems: Hier können die einfachen Hochkommata durch Voranstellen von »\« gequotet werden:

```
$ echo 'Die einfachen Anführungszeichen \'...\' sind rigoros.'
Die einfachen Anführungszeichen '...' sind rigoros.
```

Da jedoch die meisten anderen Shells diese Ausnahme nicht kennen, sollten Sie sich die erste Variante merken. Die Z Shell löst dieses Problem auf eine andere Weise, die aus der Programmiersprache C bekannt ist: Ist die Shelloption *RC_QUOTES* gesetzt, wird »''« innerhalb von »'...'« durch ein einfaches Hochkomma ersetzt:

```
$ setopt RC_QUOTES
$ echo 'Die einfachen Anführungszeichen ''...'' sind rigoros.'
Die einfachen Anführungszeichen '...' sind rigoros.
$ unsetopt RC_QUOTES
$ echo 'Die einfachen Anführungszeichen ''...'' sind rigoros.'
Die einfachen Anführungszeichen ... sind rigoros.
```

6.4 ANSI-C-Quoting

C-Programmierern sind sie schon seit langem ein Begriff, die Shellprogrammierer können sie jedoch erst seit deren Einführung in der Korn-Shell86 nutzen: die ANSI-C-Escape-Sequenzen. Mit ihnen ist es möglich, nicht über die Tastatur erreichbare Zeichen durch einfach zu schreibende Sequenzen zu umschreiben. Escape-Sequenzen sind je nach Shell an den unterschiedlichsten Stellen gültig, besonders jedoch beim Befehl *print* oder innerhalb des ANSI-C-Quotings (gültig in der Korn-Shell93, der Bash2 und der Z Shell):

```
$'...'
```

Folgende Tabelle enthält eine Aufstellung aller verfügbaren Escape-Sequenzen und deren Bedeutung:

Escape-Sequenz	Steht für
\a	Kurzer Piepser.
\b	Backspace (Cursor ein Zeichen zurücksetzen).
\e	Escape-Zeichen, notwendig für ANSI-Formatierungen (siehe Kapitel 13.4).
\f	Formfeed (neue Seite), an einen Drucker gesandt, bewirkt dieses Zeichen, daß der Drucker eine neue Seite beginnt.
\n	Neue Zeile (Newline-Zeichen, »Neue-Zeile-Zeichen«).
\r	Carriage return, Wagenrücklauf ohne in die nächste Zeile zu wechseln.
\t	Horizontaler Tabulator, horizontale Einrückung.
\v	Vertikaler Tabulator, wechselt in die nächste Zeile, macht aber keinen Wagenrücklauf, bleibt also in der gleichen Spalte.
\\	Backslash, Rückstrich.
\nnn	Zeichen, das den ASCII-Code *nnn* (oktal) besitzt.
\xnnn	Zeichen, das den ASCII-Code *nnn* (hexadezimal) besitzt.

Folgendes Beispiel verdeutlicht die Verwendung der Sequenz *r*:

```
echo -n Kopiere ...  &&
cp backup.tgz /mnt/bckup/data &&
echo $'\r'Kopieren erfolgreich.
```

Hier wird zuerst die temporäre Meldung »Kopiere ...« ausgegeben, wobei allerdings mit der Option *-n* des Befehls *echo* verhindert wird, daß nach der Ausgabe

der Zeichenkette in die nächste Zeile gewechselt wird. Nach dem eigentlichen Kopieren der Datei wird mit *$'\r'* der Cursor an den Beginn der aktuellen Zeile gesetzt, so daß die Zeichenkette »Kopieren erfolgreich.« die alte Zeile überschreibt. Da die Escape-Sequenzen gerade beim Befehl *echo* sehr nützlich sind, können sie direkt mit *echo* angewendet werden. Das obige Beispiel sieht dadurch etwas lesbarer aus:

```
echo -n Kopiere ...  &&
cp backup.tgz /mnt/bckup/data &&
echo \rKopieren erfolgreich.
```

Eine weitere, oft in Here-Dokumenten verwendete Escape-Sequenz ist *\f*:

```
$ cat << ENDE | lpr
Die Menschheit sollte Papier sparen, deshalb dieses
Dokument bitte nicht ausdrucken,$'\f'
da auf dem Drucker sonst nach jeder zweiten
Zeile eine neue Seite ausgespuckt wird.$'\f'
ENDE
```

6.5 Ein Bug geht um die Welt

»ksh can safely replace /bin/sh on most systems today unlike zsh which would cause many script to fail in mysterious ways.«

David Korn

In Verbindung mit den Quoting-Mechanismen gibt es bereits seit der Bourne-Shell eine Besonderheit. Die Shell verhält sich in (mindestens?) einem Punkt nicht erwartungsgemäß, man könnte auch sagen: Für die einen ist es der größte Shell-Bug der Welt, für die anderen ein undokumentiertes Feature. Doch urteilen Sie selbst.

Wenn Sie denken, das Szenario, in dem sich der vermeintliche Bug zeigt, sei konstruiert, dann irren Sie. Wir beginnen mit einer Wertzuweisung mit Hilfe von Anführungszeichen:

```
$ variable="Ist es wichtig"
```

Das Quoting benötigen wir, weil Leerzeichen in der Zeichenkette vorkommen. Nun übergeben wir den Wert der Variable an ein Skript, das die Anzahl der übergebenen Parameter liefert:

```
$ cat zaehleargs
echo $#
```

Welches Resultat erwarten Sie bei folgendem Aufruf?

```
$ zaehleargs $variable
3
```

Zu erwarten wäre eigentlich, daß *zaehleargs* nur eine Positionsvariable mit dem Wert »Ist es wichtig« erhält. Dies ist aber offenbar nicht der Fall, da die Shell die Variablen *vor* dem Aufsplitten der Befehlszeile auflöst. Die Schritte im einzelnen:

1. Variablen auflösen:

```
zaehleargs Ist es wichtig
```

2. Befehlszeile in Programm und Argumente aufteilen:

Positionsvariable	Wert
$0	zaehleargs
$1	Ist
$2	es
$3	wichtig

Obwohl dieses Verhalten vom Standpunkt des Hochsprachenprogrammierers sehr ungewöhnlich ist, wurde es trotzdem aus Gründen der Abwärtskompatibilität beibehalten. Um die komplette Zeichenkette »Ist es wichtig« als ein einziges Argument zu übergeben, müssen leider noch einmal die Anführungszeichen dafür sorgen:

```
$ zaehleargs "$variable"
1
```

Als aufmerksamer Leser werden Sie jetzt anmerken, daß der Autor dieses Buchs bereits viele Male diesen »Bug« ausnutzte (vgl. Kapitel 5.4.4 auf Seite 115). Aber wie ich bereits sagte: it's not a bug, it's a feature. Außerdem möchte ich nicht verschweigen, daß in vielen Fällen die automatische Wortaufspaltung enorm praktisch ist und daher häufig verwendet wird. Andererseits entstehen die meisten Fehler in Shellskripten dadurch, daß Anführungszeichen um einen Variablenausdruck vergessen wurden. Für den Vater der Z Shell, Paul Falstad, war es deshalb eines der wichtigsten Bestreben, dieses Verhalten aus der Welt zu schaffen. Deshalb funktioniert der in diesem Buch vielfach verwendete *set*-Trick in der Z Shell nicht, außer Sie sorgen mit Hilfe der Shelloption *SH_WORD_SPLIT* für die Abwärtskompatibilität.

Die automatische Wortaufsplittung ist also ein bequemer Service, der jedoch oft nicht erwünscht ist, beispielsweise, wenn eine Variable einen Dateinamen oder Pfad enthalten soll. Enthält der Dateiname nämlich Leerzeichen, macht die Shell durch die Aufspaltung mehrere Dateinamen daraus. Vergleichen Sie selbst:

```
$ [ -n "$ZSH_VERSION" ] && setopt SH_WORD_SPLIT  # nur für Z Shell
$ filename="Ringsgwandl - Gache Wurzn.mp3"
$ ls -C $filename
```

```
$ ls -C "$filename"
Ringsgwandl - Gache Wurzn.mp3
$ set $filename
$ echo $1
Ringsgwandl
$ set "$filename"
Ringsgwandl - Gache Wurzn.mp3
```

Die wichtige Erkenntnis dieses Beispiel ist also: Falls Sie es nicht mit der Z Shell zu tun haben, müssen Sie jede Variable, die einen Dateinamen enthält, quoten!

Um die Verwirrung komplett zu machen, gibt es eine weitere Besonderheit der Z Shell: Obwohl die Wortaufspaltung durch *IFS* bei Variablen strikt abgelehnt wird, geschieht sie dennoch bei Kommandosubstitutionen standardmäßig. Dies ist ein notwendiges Zugeständnis an die Kompatibilität und Praktikabilität, da besonders Newline als Worttrenner oft ausgenützt wird. Denken Sie nur an einen *find*-Aufruf wie

```
ls -l $(find -name "*.java" -print)
```

find gibt die gefundenen Dateien zeilenweise aus, das heißt also getrennt von Newline-Zeichen. Würde die Z Shell keine Wortaufspaltung betreiben, müßten Sie durch irgendeinen anderen Zusatz dafür sorgen, daß dies geschieht. Da hier aber die vermeintliche Ausnahme zur Regel wird, macht die Z Shell die Wortaufspaltung bei Kommandosubstitutionen zum Standard. Merken Sie sich also:

Bei Variablen wird standardmäßig keine Wortaufspaltung betrieben, bei der Kommandosubstitution jedoch schon!