# Shell-Einführung, Tipps und Tricks

Version 1.10 — 12.12.2016





OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
E-Mail: tb@ostc.de
Web: www.ostc.de

# Inhaltsverzeichnis

# 1 Eigenschaften der Shell

Bei vielen Betriebssystemen ist der **Kommando-Interpreter** fest eingebaut, er ist integraler Bestandteil des Betriebssystems. In UNIX ist er dagegen ein Programm wie jedes andere auch. Traditionell werden Kommando-Interpreter in UNIX als **Shell** bezeichnet, vielleicht weil damit die Benutzer vor dem zentralen Kernel — oder der Kernel vor den Benutzern — geschützt wird!

Bei der Anmeldung eines Benutzers am UNIX-System wird in aller Regel eine **Login-Shell in einem Terminal** gestartet (in einer GUI öffnet man dazu ein Terminal). Diese Shell zeigt einen **Eingabe-Prompt** (Standard: `$` oder `#`) an und wartet dann auf Eingaben des Benutzers. Nachdem dieser ein Kommando eingetippt und „Return“ gedrückt hat, liest sie die Eingabe, interpretiert diese, führt das Kommando aus und das Ergebnis wird auf dem Terminal ausgegeben (REPL = Read-Eval-Print Loop).

## 1.1 Shell-Mechanismen

Folgende Mechanismen (und die damit verbundenen Sonderzeichen) der diversen Shells sollte man kennen und verstehen, um sie richtig benutzen zu können:

- Eingabe in Worte (Token) zerlegen (anhand „Whitespaces“) und interpretieren
- Befehlsarten unterscheiden (Builtin, Alias, Funktion, externes Kmdo, Programm/Skript)
- Kommandos-Suche durchführen (per `PATH`-Variable)
- Kommando ausführen (1. Wort in der Befehlszeile)
- Exit-Status (`0`=Kein Fehler, `1-255`=Fehler)
- Dateimuster-Expansion (`* ? [...] [^...] [!...] ~ ~USER {...,...}`)
- Ein/Ausgabe-Umlenkung (`> >> 2> 2>> < 1>&2 2>&1`)
- Here-Dokument (`<<`)
- Pipes aufbauen (`|`)
- Hintergrund-Prozesse (`&`)
- Variablen-Substitution (`$VAR`)
- Quotierung (`"..." '...' \.`)
- Kommando-Substitution (`` `...` `` und `$(...)`)
- Steuerzeichen (`Strg-C Strg-D Strg-I=TAB Strg-L Strg-V Strg-Z`)
- Kommandozeile editieren
- Kommando/Datei/Variablen/Benutzernamen-Vervollständigung (Completion, `TAB`)

- History-Mechanismus (`history !NR !NAME !! Strg-R, ...`)
- Aliase (`alias ll="..."` und `unalias ll`)
- Funktionen (`FNAME() { ...; return ...; }`)

Diese Mechnismen werden in Kapitel 3 ab Seite 14 erklärt.

## 1.2 Shell-Programmierung

Die zweite wichtige Aufgabe einer Shell ist das **Ausführen von Shell-Skripten** („Batches“). Diese werden dabei nicht in ausführbaren Code übersetzt, sondern zeilenweise interpretiert. Daraus ergeben sich folgende Eigenschaften:

- Kein Compilerlauf notwendig
- Auswertung von Ausdrücken zur Laufzeit möglich
- Makros möglich
- Relativ langsame Ausführung
- Erweiterbar

Folgende Shell-Eigenschaften sollte man kennen, um sie zur Programmierung von Shell-Skripten einsetzen zu können:

- Konfigurations-Dateien (`/etc/profile ~/.profile ~/.bashrc ...`)
- Variable
  - ▷ Shell/Environment-Variable (`set env export unset`)
  - ▷ Standard-Variable (`HOME PS1 PS2 IFS SHELL TERM PATH PWD USER ...`)
  - ▷ Parameter-Übergabe (`$0 $# $* $@ $1 $2 ...`)
  - ▷ Spezial-Variable (`$? $! $$ $-`)
- Kontrollstrukturen
  - ▷ Das `test`-Kommando (Bedingung prüfen)
  - ▷ Sequenz (`;`)
  - ▷ Verzweigung (`if ... then ... elif ... else ... fi`)
  - ▷ Mehrfachverzweigung (`case ... in ... esac`)
  - ▷ Schleife (`for ... in .../while .../until ... do ... done`)
  - ▷ Funktionen (`func() { ...; return ...; }`)
  - ▷ (Vorzeitiger) Abbruch (Schleife: `break continue`, Skript: `exit`)
  - ▷ Signale abfangen (`trap`)

    - ▷ Boolesche Werte (`true false`)
- Verschiedene Aufruf-Arten
    - ▷ Sub-Shells
    - ▷ Quellcode einlesen (`. source`)

## 1.3 Kommandosyntax und -aufruf in der Shell

Die allgemeine Syntax von Kommando-Aufrufen in der Shell lautet:

```
CMD [OPTION...] [--] [ARGUMENT...]
```

Ein eingetipptes Kommando wird erst durch das Drücken der „Return"-Taste der Shell zur Interpretation und Ausführung übergeben, bis dahin kann es beliebig geändert werden. Die Shell zerlegt den auf der Kommandozeile eingegebenen Text in Worte (anhand der **Whitespaces** Leerzeichen, Tabulator und Zeilenvorschub).

Das 1. Wort auf der Kommandzeile ist das Kommando `CMD`. Alle direkt darauf folgende Worte mit `-` (Minus) als 1. Zeichen sind **Optionen** (Schalter), sie beeinflussen das Verhalten des Kommandos. Optionen sind entweder **einbuchstabig** oder (in GNU-Programmen) **mehrbuchstabig**. Optionsbuchstaben stehen für ein englisches Wort, das ihre Bedeutung beschreibt (Merkhilfe!).

- **Einbuchstabige Optionen** `-o` können einzeln (jede mit `-` davor) oder ohne Leerzeichen kombiniert (nur ein `-` am Anfang notwendig) angegeben werden.
- **Mehrbuchstabige Optionen** `--option` sind durch `--` einzuleiten und können nicht kombiniert werden.

Bei vielen Optionen genügt ihre reine Angabe, da sie nur etwas ein- oder ausschalten. Zu einigen Option kann ein **Parameter** nötig oder optional sein, dieser ist direkt dahinter anzugeben (und beginnt nicht mit „–"). Mögliche Formate:

- Bei einbuchstabige Optionen `-o` direkt anschließend dahinter (`-owert`) bzw. mit einem Leerzeichen getrennt (`-o wert`).
- Bei mehrbuchstabige Optionen `--option` direkt anschließend dahinter mit =-Zeichen (`--option=wert`) oder mit einem Leerzeichen (`--option wert`) getrennt.

Das 1. Wort, das nicht mit einem „–" beginnt, leitet die Liste der sonstigen **Kommandoargumente** ein. Typischerweise werden hier die vom Kommando zu bearbeitenden Dateien/Verzeichnisse angegeben.

**Explizit beendet** wird die Liste der Optionen auch durch Angabe von `--`. Alle danach aufgeführten Argumente (auch wenn sie als 1. Zeichen `-` enthalten) werden *nicht* als Optionen interpretiert.

Das Argument „–“ alleine steht für die **Standard-Eingabe** oder **-Ausgabe**. Es kann bei Kommandos notwendig sein, die standardmäßig nicht von/auf Standard-Ein/Ausgabe lesen/schreiben (z.B. `find tar gzip`).

Beispiele:

| | |
|---|---|
| `ls *.c *.h` | Dateimuster `*.c *.h` |
| `ls -l *.c *.h` | Option `-l`, Dateimuster `*.c *.h` |
| `ls -l -R -t` | 3 einbuchstabige Optionen `-l -R -t` |
| `ls -lRt` | Analog (kombiniert) |
| `ls --long --recursive --time` | 3 Optionsworte |
| `ls -- -*.c` | Dateimuster `-*.c` (wg. `--` kein Option) |
| `tar czf - /etc \| gzip - > etc.tgz` | Verz. `/etc` archivieren + komprimieren |

Gelegentlich kommt das Zeichen „`;`“ (Semikolon) in Kommandozeilen vor, es trennt (wie der Zeilenvorschub) die Kommandos: Man gibt ein Kommando ein und anschließend — anstatt „Return“ zu drücken — ein Semikolon und ein weiteres Kommando.

```
cd /tmp; ls -l; cd -
```

Diese **Verkettung** mehrerer Kommandos in einer Zeile ist insbesondere in Sub-Shells, Aliasen, Funktionen und Kommando-Listen hilfreich.

### 1.4 Exit-Status

Jedes Kommando gibt einen **Exit-Status** (0–255) zurück, der den Wert 0 hat, wenn das Kommando **korrekt** ablief und einen Wert ungleich 0, wenn bei seiner Ausführung irgendwelche **Fehler** auftraten. Die Bedeutung der einzelnen Exit-Fehlercodes ist von Kommando zu Kommando verschieden und kann in den jeweiligen `man`-Pages nachgelesen werden.

Der Exit-Status von Kommandos wird in Shell-Kontrollstrukturen und zur Steuerung des Programmflusses benutzt. Er steht auch in der Variablen `$?` zur Verfügung und kann darüber ausgegeben werden (genau ein Mal!). Beispiel:

| | |
|---|---|
| `grep` *TEXT FILE* | 0 falls *TEXT* in *FILE* gefunden |
| `grep` *TEXT FILE* | 1 falls *TEXT* in *FILE* nicht gefunden |
| `grep` *TEXT FILE* | 2 falls *TEXT* fehlerhaft oder *FILE* nicht vorhanden |
| `echo $?` | Exit-Status ausgeben (nur 1× möglich) |
| `echo $?` | Exit-Status von vorherigem `echo` ausgeben (da ebenfalls ein Kmdo) |

## 2 Die verschiedenen Shells

Aufgrund der historischen Entwicklung und der im Laufe der Zeit gestiegenen Anforderungen sind verschieden leistungsfähige Shells verfügbar, die sich in Syntax, Funktionalität und Konfigurationsmöglichkeiten unterscheiden.

`sh` **Bourne-Shell** (nach ihrem Ersteller *Steve Bourne* benannt). Sie ist die älteste UNIX-Shell und ist auf allen UNIX-Systemen verfügbar. Sie ist relativ einfach, es fehlen ihr folgende Möglichkeiten moderner Shells: *Job-Kontrolle* (das ist die Fähigkeit, Jobs aus dem Vordergrund in den Hintergrund zu stellen), *Command-Line-Edit* (die Fähigkeit, Kommandos beliebig zu editieren), *History* (die Fähigkeit, schon einmal eingegebene Kommandos zu wiederholen) und *Filename-Completion* (die Fähigkeit, teilweise eingegebene Dateien/Kommandos zu vervollständigen) sowie *Aliase* (Einführung eigener Kommandos). Sie ist sehr gut geeignet zur (portablen) *Shell-Programmierung* oder zur Erstellung von Kommando-Dateien, da sie die „**Lingua Franca**“ der meisten Shells darstellt. Sie ist weniger gut geeignet zur interaktiven Benutzung, jede der im folgenden beschriebenen Shells ist dafür besser geeignet.

`csh` **C-Shell** wurde in Berkeley als Teil der dortigen UNIX-Implementation von *Bill Joy* (Vi!) entwickelt, ihre Syntax ist an C angelehnt (daher der Name!). Sie ist eine populäre Shell für die interaktive Benutzung und besitzt eine Menge nützlicher Eigenschaften, die in der Bourne-Shell nicht verfügbar sind wie z.B. *Job-Kontrolle* und *History*. Während sie jedoch bei normalem Gebrauch problemlos ist, sind ihre Grenzen bei der Shell-Programmierung schnell erreicht, da sie eine ganze Reihe von versteckten Bugs enthält und ziemlich langsam ist.

`ksh` **Korn-Shell** (nach ihrem Erfinder *David Korn* benannt) ist kompatibel mit der Bourne-Shell, kennt aber die meisten Möglichkeiten der C-Shell und bietet zusätzlich weitere neue Eigenschaften wie z.B. *Command-Line-Edit* (die Fähigkeit, alte Kommandos wieder in die Kommandozeile zu holen und sie dort vor der Ausführung zu editieren) und *Filename-Completion* (die Fähigkeit, teilweise eingegebene Kommandos/Dateinamen vom System vervollständigen zu lassen). Sie ist außerdem zuverlässiger als die C-Shell. Die Korn-Shell ist die **Standard-Shell in UNIX System V Release 4**.

`bash` **Bo(u)rn(e)-again Shell**, entwickelt von der *Free Software Foundation (FSF)*, ist der Korn-Shell sehr ähnlich. Sie hat viele Eigenschaften der C-Shell plus *Command-Line-Editierung* und ein eingebautes Hilfe-Kommando. Die Bo(u)rn(e)-Again-Shell ist die **Standard-Shell unter Linux**.

`tcsh` **Tenex C-Shell**, eine erweiterte Versionen der C-Shell, arbeitet wie die originale C-Shell — hat aber wesentlich mehr Eigenschaften und weniger Fehler. Bietet darüber hinaus *Filename-Completion* (Vervollständigung von teilweise eingegebenen Dateinamen und Kommandos) und *Command-Line-Editierung* an.

`ash` **A-Shell** („A“ ist der Anfang des Alphabets), sehr einfache Shell abgeleitet von der `sh`. Wird aufgrund ihres geringen Codeumfangs gerne in kleinen (embedded) Linux-Systemen eingesetzt.

`rsh` **Restricted Shell** mit einigen Beschränkungen aufgrund von Sicherheitskriterien, abgeleitet von der `sh`.

`zsh` **Z-Shell** („Z“ ist das Ende des Alphabets), abgeleitet von der `sh`. Ultimative Shell, vereint alle Verbesserungen von `bash`, `ksh` und `tcsh` und viele zusätzliche Erweiterungen.

Da `bash`, `ksh` und `zsh` Skripten interpretieren können, die für die originale Bourne-Shell `sh` geschrieben wurden, wird üblicherweise der von der Bourne-Shell unterstützte Befehlsumfang zur Shell-Programmierung verwendet, wenn man *portable Shell-Skripten* schreiben möchte. Sollen Skripte nur unter Linux ausgeführt werden, werden meist die vollen Syntax-Möglichkeiten der `bash` genutzt.

## 2.1 Gemeinsamkeiten und Unterschiede

Die verschiedenen Shells sind historisch zu unterschiedlichen Zeitpunkten entstanden und unterscheiden sich (leider) in ihrer Syntax, ihren Fähigkeiten und ihren Konfigurationsmöglichkeiten. Von den beiden (inkompatiblen) **Hauptlinien** `sh` und `csh` leiten sich alle weiteren Shells ab (Spalte **Orig**). Shells sind sowohl zur **interaktiven** Benutzung auf der Kommandozeile (Spalte **Int**) als auch zur Ausführung von **Shell-Skripten** (Batch-Dateien) gedacht (Spalte **Bat**). Nachfolgend eine Liste der wichtigsten Shells und eine Bewertung ihrer Eignung für den interaktiven bzw. den Batch-Gebrauch:

| Kmdo | Name | Int | Bat | Orig | Beschreibung |
|---|---|---|---|---|---|
| `sh` | Bourne-Shell | −− | + | | „Ur“-Shell, Skript-orientiert |
| `csh` | C-Shell | + | − | | Angelehnt an C, Interaktions-orientiert |
| `tcsh` | Tenex-C-Shell | ++ | + | `csh` | BSD/SUN-Solaris |
| `ksh` | Korn-Shell | ++ | ++ | `sh` | HP/UX und UNIX System V |
| `bash` | Bo(u)rn(e)-Again-Shell | ++ | ++ | `sh` | Linux |
| `rsh` | Restricted Shell | ++ | +− | `sh` | Um Benutzer einzuschränken |
| `ash` | A-Shell | +− | +− | `sh` | Umfasst nur das Nötigste, klein |
| `zsh` | Z-Shell | ++ | ++ | `sh` | Umfasst alles, groß, verwirrend |

## 2.2 Login-, Sub- interaktive und Batch-Shell

Folgende verschiedene Aufruf-Arten für Shells gibt es:

- **Login-Shell:** Beim Einloggen gestartet, liest die Shell-Konfigurations-Dateien ein.
- **Sub-Shell:** Für jedes Unterkommando gestartet, liest keine oder nur sehr wenige Shell-Konfigurations-Dateien ein.
- **Interaktive Shell:** Zeigt Prompt an, nimmt interaktiv Kommandos entgegen und führt sie aus.
- **Batch-Shell:** Zeigt keinen Prompt an, liest Kommandos aus Skript-Datei ein und führt sie aus.

**Hinweis**: Eine Login-Shell ist meist interaktiv, eine Sub-Shell ist meist eine Batch-Shell.

## 2.3 Login-Shell versus Sub-Shell

Eine Shell kann in zwei verschiedenen Modi laufen: als **Login-Shell** und als **Sub-Shell** (Nicht-Login-Shell).

- Loggt man sich in ein UNIX-System ein, so startet das Programm `login` normalerweise eine Shell. Es setzt dabei ein besonderes Flag, um der Shell mitzuteilen, dass sie eine **Login-Shell** ist.
- Eine **Sub-Shell** ist niemals eine Login-Shell. In allen Shells führen die **Klammer-Operatoren** `()` dazu, dass eine weitere Instanz der aktuellen Shell gestartet wird (Klammern werden **Sub-Shell-Operatoren** genannt). Eine Sub-Shell wird auch durch den Aufruf eines **Shell-Skriptes** (das ist eine ausführbare (Batch-)Datei, die Shell-Befehle enthält), durch Aufruf des Shell-Kommandos (z.B. `sh`) auf der Kommandozeile oder durch einen **Shell-Escape** (das ist der Aufruf einer Shell aus einem Anwendungsprogramm wie z.B. dem Vi heraus) **gestartet**. Eine Sub-Shell gibt keinen Eingabe-Prompt aus und hat normalerweise nur eine kurze Lebenszeit.

Beim ersten Einloggen in ein System benötigt man eine **Login-Shell**, die Dinge wie z.B. den Terminal-Typ, Suchpfad, eigene Kommandos, u.ä. einrichtet. Andere Shells im gleichen Terminal sind Nicht-Login-Shells — um das erneute Ausführen dieser einmalig notwendigen Setup-Kommandos zu verhindern.

Die einzelnen Shells haben verschiedene Mechanismen, den ersten Shell-Aufruf von den folgenden Shell-Aufrufen zu unterscheiden, der restliche Abschnitt behandelt diese Unterschiede.

## 2.4 Inhalt von Shell Konfigurations-Dateien

Shell Konfigurations-Dateien wie `~/.login` oder `~/.profile` führen typischerweise mindestens folgende Tätigkeiten aus:

- Setzen/Erweitern des **Suchpfades** `PATH`.
- Setzen des **Terminaltyps** `TERM` und verschiedener Terminalparameter.
- Setzen von **Shell- und Umgebungs-Variablen**, die von den unterschiedlichen Programmen oder Skripten benötigt werden.
- Ausführen eines oder mehrerer **Kommandos** für Tätigkeiten, die bei jedem Login durchgeführt werden sollen. Wenn zum Beispiel das System-Login-Kommando die „Nachricht des Tages“ nicht anzeigt, kann das die Konfigurations-Datei erledigen. Viele Leute mögen es auch, einen lustigen oder lehrreichen „Fortune“ beim Login zu erhalten. Eventuell können auch von `who` oder `uptime` bereitgestellte Systeminformationen ausgegeben werden.

In der C-Shell wird die Datei `~/.cshrc` verwendet, um Einstellungen durchzuführen, die in jeder C-Shell-Instanz notwendig sind, nicht nur in einer Login-Shell. Zum Beispiel sollen meist Aliase in jeder ausgeführten interaktiven Shell zur Verfügung stehen.

Selbst Anfänger können einfache `~/.profile`, `~/.login` oder `~/.cshrc`-Dateien schreiben bzw. die vorhandenen ergänzen. **Achtung:** Bevor eine Konfigurations-Datei geändert wird, sollte immer eine **Sicherheitskopie** von ihr erstellt werden, damit (1) bei Problemen wieder auf das Original zurückgeschalten werden kann und (2) die Unterschiede zwischen der Originaldatei und der geänderten Version untersucht werden können, um den/die Fehler herausfinden zu können (*bereits ein vergessenes* `"` *kann die gesamte Datei unbrauchbar machen*). Die eigentliche Kunst besteht darin, diese Konfigurations-Dateien richtig einzusetzen. Hier einige Dinge, die man ausprobieren kann:

- Einen angepassten Eingabe-Prompt erstellen.
- Gemeinsame Konfigurations-Dateien für unterschiedliche Maschinen aufbauen.
- Je nach verwendetem Terminal unterschiedliche Terminaleinstellungen durchführen.
- Die Nachricht des Tages nur anzeigen, wenn sie sich geändert hat.
- Alle obigen Tätigkeiten durchführen, ohne dass der Login-Vorgang ewig dauert.

## 2.5 Shell-Konfigurations-Dateien

Konfigurations-Dateien der Shell sind einfach **Shell-Skripte**, in denen beliebige Shell-Befehle und normale UNIX-Kommandos stehen können. Sie werden je nach Shell beim:

- Login
- Start einer interaktiven Shell bzw. Sub-Shell
- Logout

gelesen und ausgeführt. Alle darin getroffene Einstellungen gelten dann in der soeben gestarteten Shell.

Unterschieden werden auch:

- **Zentrale** Shell-Konfigurations-Dateien für alle Benutzer, sie stehen im Verzeichnis `/etc`.
- **Persönliche** Shell-Konfigurations-Dateien für jeden einzelnen Benutzer `USER`, sie stehen in seinem Heimat-Verzeichnis `/home/USER` bzw. `~USER` bzw. `~`.

Die persönlichen Konfigurations-Dateien werden *nach* den zentralen Konfigurations-Dateien gelesen und können deren Einstellungen *überschreiben*. Ebenso kann jede später gelesene Konfigurations-Datei die Einstellungen einer vorher gelesenen überschreiben.

Je nach Abstammung versuchen moderne Shells die Konfigurations-Dateien der *Vorläufer-Shell* zu lesen bzw. sie nur dann zu lesen, wenn keine eigene Konfigurations-Datei speziell für sie vorhanden ist.

**Hinweis**: `rc` steht für *Resource/Run Control*.

### 2.5.1 Bourne-Shell `sh`

Die Bourne-Shell liest beim Einloggen zwei Dateien ein: sie heißen `/etc/profile` und `$HOME/.profile`. Die erste ist für alle Benutzer gleich, die zweite ist pro Benutzer definierbar, da sie in seinem Heimat-Verzeichnis steht.

| Konfigurations-Datei | Wann | Typ |
|---|---|---|
| `/etc/profile` | Login | Zentral |
| `~/.profile` | Login | Benutzerspez. |
| — | Interaktiv | — |
| — | Logout | — |

Die Bourne-Shell liest beide Dateien *nicht* ein, wenn eine Sub-Shell gestartet wird. Die Konfigurations-Informationen für die Sub-Shell stammen aus den **Umgebungs-Variablen**, die beim ersten Login oder in der Zwischenzeit durch Eingabe von Kommandos gesetzt wurden (Shell-Funktionen werden nicht vererbt).

**Hinweis**: Die Bourne-Shell kennt keine Konfigurations-Datei für die Interaktions/Sub-Shell und den Logout-Vorgang.

### 2.5.2 Korn-Shell `ksh`

Die Korn-Shell verhält sich fast wie die Bourne-Shell. Eine Login-Korn-Shell liest zunächst die Datei `/etc/profile` und dann die Datei `~/.profile`. Sie kann die Umgebungs-Variable `ENV` (*environment*) auf den Pfadnamen einer Datei (typischerweise `$HOME/.kshrc`) setzen. Jede interaktive Korn-Shell wird dann während dieser Login-Sitzung (auch jede Sub-Shell) bei ihrem Start die durch `ENV` bezeichnete Datei einlesen, bevor sie andere Kommandos ausführt.

| Konfigurations-Datei | Wann | Typ | Hinweis |
|---|---|---|---|
| `/etc/profile` | Login | Benutzerspez. | `sh` |
| `~/.profile` | Login | Zentral | `sh` |
| `~/.kshrc` | Interaktiv | Benutzerspez. | `ENV` |
| — | Logout | — | |

**Hinweis**: Die Korn-Shell kennt keine Konfigurations-Datei für den Logout-Vorgang.

### 2.5.3 Bo(u)rn(e)-Again-Shell `bash`

Die Bo(u)rn(e)-Again-Shell `bash` ist eine Kreuzung zwischen Bourne- und C-Shell. Beim Login liest sie erst die Datei `/etc/profile` und dann eine der Dateien `~/.bash_profile`,

`~/.bash_login` oder `~/.profile` ein (die Suche erfolgt in dieser Reihenfolge).

| Konfigurations-Datei | Wann | Typ | Hinweis |
|---|---|---|---|
| `/etc/profile` | Login | Zentral | `sh` |
| `~/.bash_profile` | Login | Benutzerspez. | |
| `~/.bash_login` | Login | Benutzerspez. | |
| `~/.profile` | Login | Benutzerspez. | `sh` |
| `/etc/bash.bashrc` | Interaktiv | Zentral | |
| `~/.bashrc` | Interaktiv | Benutzerspez. | |
| `~/.bash_aliases` | Interaktiv | Benutzerspez. | |
| `~/.bash_logout` | Logout | Benutzerspez. | |

Eine `bash`-Sub-Shell — nicht aber eine Login-Shell — liest die Datei `~/.bashrc` im Heimat-Verzeichnis ein.

Wird eine `bash`-Login-Shell beendet, dann liest sie die Datei `~/.bash_logout`.

### 2.5.4 C-Shell `csh`

Die C-Shell kennt vier Konfigurations-Dateien:

| Konfigurations-Datei | Wann | Typ |
|---|---|---|
| `/etc/cshrc` | Login | Zentral . |
| `~/.login` | Login | Benutzerspez. |
| `~/.cshrc` | Interaktiv | Benutzerspez. |
| `~/.logout` | Logout | Benutzerspez. |

- Zentrale Datei `/etc/csh.cshrc` (*C-Shell resource*) wird bei jedem Login als 1. Datei gelesen.
- Datei `~/.cshrc` (*C-Shell resource*) wird bei jedem Start einer C-Shell gelesen — dies schließt Shell-Escapes und Shell-Skripten ein. Hier sind Kommandos abzulegen, die bei jedem Start einer Shell ausgeführt werden sollen. Zum Beispiel sollten hier Shell-Variablen wie `cdpath` und `prompt` gesetzt werden, ebenso Aliase. Diese Dinge werden nicht über Umgebungs-Variablen an Sub-Shells weitergeleitet, daher gehören sie in der Datei `~/.cshrc` gesetzt.
- Datei `~/.login` wird bei jedem *Start einer Login-Shell* gelesen. Folgendes sollte darin gesetzt werden:
  - ▷ Umgebungs-Variablen (die UNIX automatisch an Sub-Shells weitergibt).
  - ▷ Initialisierungs-Kommandos wie `tset`, `tput` und `stty`.
  - ▷ Kommandos, die bei jedem Login ausgeführt werden sollen — ob Mails oder News angekommen sind, `fortune` ausführen, den Tageskalender ausgeben, usw.
- Datei `~/.logout` wird gelesen, wenn eine *Login-Shell beendet* wird.

**Achtung**: Bitte beachten, dass beim Login `~/.cshrc` vor `~/.login` gelesen wird!

### 2.5.5 Tenex-C-Shell `tcsh`

Die Tenex-C-Shell verhält sich bis auf eine Ausnahme wie die C-Shell: wenn eine Datei `~/.tcshrc` im Heimat-Verzeichnis steht, wird sie anstelle von `~/.cshrc` gelesen.

| Konfigurations-Datei | Wann | Typ | Hinweis |
|---|---|---|---|
| `/etc/csh.cshrc` | Login | Zentral | |
| `/etc/csh.login` | Login | Zentral | |
| `~/.tcshrc` | Login | Benutzerspez. | |
| `~/.cshrc` | Login | Benutzerspez. | `csh` |
| `~/.history` | Login | Benutzerspez. | |
| `~/.login` | Login | Benutzerspez. | `csh` |
| `~/.cshdirs` | Login | Benutzerspez. | |
| `/etc/csh.cshrc` | Interaktiv | Zentral | |
| `~/.tcshrc` | Interaktiv | Benutzerspez. | |
| `/etc/csh.logout` | Logout | Zentral | |
| `~/.logout` | Logout | Benutzerspez. | `csh` |

### 2.5.6 Z-Shell `zsh`

Die Z-Shell verzichtet vollständig auf das Einlesen der Konfigurations-Dateien der Vorgänger-Shell und verwendet statt dessen insgesamt 10(!) eigene Konfigurations-Dateien, die gemäß ihrem Einsatzzweck sauber strukturiert und benannt sind.

| Konfigurations-Datei | Wann | Typ |
|---|---|---|
| `/etc/zshenv` | Login | Zentral |
| `~/.zshenv` | Login | Benutzerspez. |
| `/etc/zprofile` | Login | Zentral |
| `~/.zprofile` | Login | Benutzerspez. |
| `/etc/zlogin` | Login | Zentral |
| `~/.zlogin` | Login | Benutzerspez. |
| `/etc/zshrc` | Interaktiv | Zentral |
| `~/.zshrc` | Interaktiv | Benutzerspez. |
| `/etc/zlogout` | Logout | Zentral |
| `~/.zlogout` | Logout | Benutzerspez. |

# 3 Shell-Mechanismen

## 3.1 Kommando-Suche

Die Suche nach einem eingetippten Kommando erfolgt mehrstufig in folgender Reihenfolge. Wird auf einer Stufe ein passendes Kommando (exakt gleicher Name) gefunden, so wird es ausgeführt und die Suche abgebrochen.

1. **Builtin** `CMD` (z.B. `echo, exit, ...`) vorhanden? → ausführen!
2. **Alias** `CMD` (z.B. `alias ll="..."`) vorhanden? → ausführen!

3. **Funktion** `CMD` (z.B. `funcname() { ...; }`) vorhanden? → ausführen!
4. Von links nach rechts alle im **Suchpfad** `PATH` (z.B. `/bin:/usr/bin:.`) angegebenen Verzeichnissen (per `:` getrennt) nach `CMD` absuchen (*muss ausführbar und bei Shell-Skripten auch lesbar sein*).
   (a) **Kommando** `/bin/CMD` vorhanden? → ausführen
   (b) **Kommando** `/usr/bin/CMD` vorhanden? → ausführen
   (c) **Kommando** `./CMD` vorhanden? → ausführen
5. Nicht gefunden → Meldung: `error: command CMD not found`

**Hinweis**: Alle direkt in die Shell eingebauten **Builtins** `CMD` listet der Befehl `help` auf, per `help CMD` erhält man eine Beschreibung dazu.

**Achtung**: Standardmäßig wird *nicht* im aktuellen Verzeichnis (**Arbeitsverzeichnis**) nach einem Kommando gesucht. Dazu wäre im Suchpfad das **aktuelle Verzeichnis** `.` (Punkt) mit aufzunehmen. Ein einzelner `:` am Anfang oder am Ende oder zwei direkt aufeinanderfolgende `::` im Suchpfad stehen ebenfalls für das aktuelle Verzeichnis. Kein Benutzer (insbesondere auch der Administrator `root`) sollte das aktuelle Verzeichnis (aus Sicherheitsgründen) im Suchpfad haben.

**Hinweis**: Die obige Kommandosuche wird ganz oder teilweise umgangen, falls das Kommando mit **relativen** oder **absoluten Pfad** davor angegeben wird. Mit einem „\“ vor dem Kommando wird ein gleichnamiger Alias ignoriert.

| | |
|---|---|
| `./CMD` | Echtes `CMD` im akt. Verz. aufrufen (keine Suche) |
| `/usr/bin/CMD` | Echtes `CMD` aus Verz. `/usr/bin` aufrufen (keine Suche) |
| `\CMD` | Alias `CMD` ignorieren (restliche Kmdo.suche schon) |
| `builtin CMD` | In Shell eingebautes `CMD` aufrufen (keine Suche) |

## 3.2 Dateinamen-Expansion

Um einem Kommando Dateinamen zur Verarbeitung zu übergeben, kann entweder die Liste der Dateinamen vollständig angegeben werden, oder per **Suchmuster** (Pattern) werden dazu passende Dateinamen von der Shell gesucht. Die Shell expandiert erst das Muster zu einer Liste von passenden Dateinamen (**filename globbing**), setzt diese dann anstelle des Suchmusters ein und führt schließlich das Kommando aus (d.h. das Kommando sieht die Suchmuster nicht). Dateinamen mit einem führenden Punkt (**versteckte Dateien**) werden nur gefunden, wenn der Punkt im Suchmuster explizit angegeben wird. Folgende **Metazeichen** (stehen nicht für sich selbst, sondern für andere Zeichen) können zur Angabe von Suchmustern verwendet werden:

| Symbol | Beschreibung |
|---|---|
| `?` | Ein beliebiges Zeichen |
| `*` | 0 oder mehr beliebige Zeichen (außer Verz.-Trenner „/“) |
| `\`$x$ | Das Zeichen $x$ *quoten* (`\\` steht für `\` selbst!) |
| `[abc]` `[a-z]` | Menge von Zeichen (**Zeichenklasse**, `[a-z]` = Zeichen von `a` bis `z`) |
| `[!abc]` `[!a-z]` | Negierte Menge von Zeichen (*sh, ksh, bash*) |
| `[^abc]` `[^a-z]` | Negierte Menge von Zeichen (*csh*) |
| `{abc,def,...}` | Liste von Zeichenketten (*csh, ksh, bash*) |
| `~` | Heimat-Verzeichnis des aktuellen Users (*csh, ksh, bash*) |
| `~`*USER* | Heimat-Verzeichnis des Users *USER* (*csh, ksh, bash*) |
| **Erweiterungen der Bash/Zsh** | |
| `**` | 0 oder mehr beliebige Zeichen (inkl. Verz.-Trenner „/“) |
| *OR-LIST* | Veroderung mehrerer Muster per „`|`“ (z.B. `xyz|abc|def`) |
| `?(`*OR-LIST*`)` | 0 oder 1 Treffer |
| `*(`*OR-LIST*`)` | 0 oder mehr Treffer |
| `+(`*OR-LIST*`)` | 1 oder mehr Treffer |
| `@(`*OR-LIST*`)` | Eines der angegebenen Muster liefert Treffer |
| `!(`*OR-LIST*`)` | Alle bis auf angegebene Muster (Inversion) |

Die Metazeichen können beliebig zu Suchmustern zusammengesetzt werden, alle passenden Datei- und/oder Verzeichnisnamen müssen das Suchmuster *vollständig* erfüllen. Eine Längenbeschränkung oder eine Beschränkung in der Anzahl der verwendeten Metazeichen gibt es nicht (außer dass die gesamte Kommandozeile max. 1-2 Mio Zeichen lang werden darf). Es können auch Muster für ganze Dateipfade (Verzeichnis + Dateiname) angegeben werden. Beispiele (`ls -d` oder `echo` voransetzen):

| | |
|---|---|
| `a*` | Dateinamen mit `a` am Anfang (auch `a`) |
| `*a` | Dateinamen mit `a` am Ende (auch `a`) |
| `*a*` | Dateinamen mit mind. einem `a` darin (auch `a`) |
| `*a*a*` | Dateinamen mit mind. zwei `a` darin (auch `aa`) |
| `[a-z][a-z]` | Kleingeschriebene zweibuchstabige Dateinamen |
| `*[0-9]*[0-9]*` | Dateinamen mit mind. zwei Ziffern darin (nicht `0-9`) |
| `*a*e*i*o*u*` | Dateinamen mit mind. 5 Vokalen in angegebener Reihenfolge |
| `?` | Einbuchstabige Dateinamen |
| `???` | Dreibuchstabige Dateinamen |
| `?a?` | Dateinamen mit 3 Buchstaben und `a` als zweitem Buchstaben |
| `*.c` | Dateinamen mit Endung `.c` |
| `/*/*.c` | Dateinamen mit Endung `.c` in Unterverz. des Root-Verz. |
| `/*/*/*.c` | Dateinamen mit Endung `.c` in Unter-Unterverz. des Root-Verz. |
| `*.[ch]` | Dateinamen die auf `.c` oder `.h` enden |
| `*[!.][!ch]` | Dateinamen die als vorletztes Zeichen nicht `.` und als letztes nicht `c` oder `h` haben (z.B. `a.`, `.b`, `aa`) |
| `*.{c,h,sh}` | Dateinamen, die auf `.c`, `.h` oder `.sh` enden |
| `*.[ch] *.sh` | Analog (zwei getrennte Muster) |
| `.[!.]*` | Versteckte Dateinamen, aber nicht `.` und `..` |

**Tipp**: Das Verzeichnis `/usr/bin` eignet sich aufgrund der vielen darin enthaltenen Dateien sehr gut zum Ausprobieren der Suchmuster. Die Angabe der Option `-d` (directory) verhindert bei `ls` das standardmäßige Auflisten des *Inhalts* von Verzeichnissen, es wird nur der Verzeichnisname selbst ausgegeben (Beispiel: `X11` in `/bin`).

## 3.3 Ein/Ausgabe-Umlenkung

Jedes UNIX-Kommando (genauer: jeder UNIX-Prozess) kennt **drei Standardkanäle** für die Datei-Eingabe und die Datei-Ausgabe:

| Kanal | Kürzel | Nummer | Default |
|---|---|---|---|
| Standard-Eingabe | `stdin` | 0 | Tastatur |
| Standard-Ausgabe | `stdout` | 1 | Bildschirm |
| Standard-Fehlerausgabe | `stderr` | 2 | Bildschirm |

Diese drei Standard-Ein/Ausgabekanäle sind standardmäßig wie angegeben mit der Tastatur oder dem Bildschirm verbunden. Sie lassen sich aber mit Hilfe der Shell über **Umlenk- und Pipe-Symbole** beliebig mit anderen Dateien oder Kommandos verbinden, ohne dass ein Kommando dies bemerkt. Die Kommandos lesen dann nicht mehr von der Tastatur bzw. schreiben nicht mehr auf den Bildschirm, sondern lesen/schreiben von/auf Datei bzw. von/an andere Kommandos. Diese Verbindungen werden von der Shell geschaffen, bevor das Kommando überhaupt gestartet wird (die Kommandos bekommen diese Umlenkung also nicht mit).

Folgende Syntax zur Ein/Ausgabe-Umlenkung gibt es (eine Pipe `|` muss *zwischen* zwei Kommandos stehen, *nicht nach* einem Kommando; nach einem Pipe-Symbol darf allerdings ein Zeilenumbruch stehen):

| Umlenkung | sh | csh |
|---|---|---|
| *stdin* aus *FILE* holen | *CMD* `<` *FILE* | *CMD* `<` *FILE* |
| *stdout* in *FILE* speichern | *CMD* `>` *FILE* | *CMD* `>` *FILE* |
| *stderr* in *FILE* speichern | *CMD* `2>` *FILE* | — |
| *stdout* + *stderr* getrennt in *FILE* + *ERR* sp. | *CMD* `>` *FILE* `2>` *ERR* | — |
| *stdout* zu *stderr* hinzufügen (kombinieren) | *CMD* `1>&2` | — |
| *stderr* zu *stdout* hinzufügen (kombinieren) | *CMD* `2>&1` | — |
| *stdout* + *stderr* zusammen in *FILE* speichern | *CMD* `>` *FILE* `2>&1` | *CMD* `>&` *FILE* |
| *stderr* + *stdout* zusammen in *FILE* speichern | *CMD* `2>` *FILE* `1>&2` | *CMD* `>&` *FILE* |
| *stdout* an *FILE* anhängen | *CMD* `>>` *FILE* | *CMD* `>>` *FILE* |
| *stderr* an *FILE* anhängen | *CMD* `2>>` *FILE* | — |
| *stdout* + *stderr* zus. an *FILE* anhängen | *CMD* `>>` *FILE* `2>&1` | *CMD* `>>&` *FILE* |
| *stderr* + *stdout* zus. an *FILE* anhängen | *CMD* `2>>` *FILE* `1>&2` | *CMD* `>>&` *FILE* |
| *stdout* + *stderr* getr. an *FILE* + *ERR* anhängen | *CMD* `>>` *FILE* `2>>` *ERR* | — |
| *stdin* bis *EOF* aus Eingabe holen (Here-Dok.) | *CMD* `<<` *EOF* | *CMD* `<<` *EOF* |
| *stdout* von *CMD1* in *CMD2* pipen | *CMD1* `\|` *CMD2* | *CMD1* `\|` *CMD2* |
| *stdout* + *stderr* von *CMD1* in *CMD2* pipen | *CMD1* `2>&1 \|` *CMD2* | *CMD1* `\|&` *CMD2* |

**Hinweis**: Einige Umlenkungen haben je nach Shell unterschiedliche Syntax und einige gibt es bei manchen Shells nicht.

Programme, die von Standard-Eingabe lesen und auf Standard-Ausgabe schreiben, werden auch **Filter** genannt, weil sie wie ein Filter Daten lesen, manipulieren und wieder ausgeben. Mehrere (einfache) Filter hintereinandergesetzt ergeben einen kombinierten (komplexen) Filter.

Das **Baukastenprinzip** von UNIX beruht stark auf dem Konzept der „Standardkanäle“ und der Pipes. Typische Filterprogramme sind z.B.: `grep`, `sort`, `uniq`, `tail`, `head`, `more`, `cut`, `paste`, `split`, `wc`, `ed`, `sed`, `awk`, `perl`, usw.

**Pipes** haben eine relativ kleine Größe (4/8 KByte), werden im Speicher aufgebaut und synchronisieren über den Datenfluss die beiden darüber verbundenen Prozesse. Sie belegen daher keinen Plattenplatz und sind sehr schnell. Werden mehrere Programme über Pipes verbunden, spricht man auch von einer **Pipeline** (Verarbeitungskette, Filterkette).

Beispiele:

| | |
|---|---|
| `cat` | `stdout` auf `stdin` schreiben (Tastatur → Bildschirm) |
| `cat >` *FILE* | `stdout` auf *FILE* schreiben (wird vorher geleert!) |
| `cat <` *FILE* | `stdin` von *FILE* lesen |
| `cat <` *FILE1* `>` *FILE2* | `stdin` von *FILE1* lesen, `stdout` auf *FILE2* schreiben |
| `cat >` *FILE2* `<` *FILE1* | Analog (Reihenfolge egal!) |
| `cat <` *FILE* `>` *FILE* | **Fehler**: (Datei *FILE* ist anschließend leer!) |
| `cat >>` *FILE* | `stdout` an *FILE* anhängen |
| `cat <` *FILE* `>>` *FILE* | **Fehler**: (Datei *FILE* wird beliebig lang!) |
| `cat 2>` *FILE* | `stderr` auf *FILE* schreiben |
| `cat < xyz` | Fehlermeldung `Datei xyz unknown` am Bildschirm |
| `cat 2>` *ERR* `< xyz` | Fehlermeldung `Datei xyz unknown` in Datei *ERR* |
| `more` *FILE* | Datei *FILE* seitenweise anzeigen |
| `cat <` *FILE* `\| more` | Analog (2 Prozesse) |
| `cat` *FILE* `\| more` | Analog (2 Prozesse) |

| | |
|---|---|
| `ls -l /bin > /tmp/liste` | Dateinamen in `/tmp/liste` ablegen |
| `ls -l /bin \| sort` | Dateien nach Typ und Rechten sortieren |
| `ls -l /bin \| sort \| more` | ... + seitenweise anzeigen |
| `ls -l /bin \| sort \| head` | ... + die ersten 10 aufsteigend |
| `ls -l /bin \| head \| sort` | ... + 10 beliebige aufsteigend |
| `ls -l /bin \| sort \| tail` | ... + die letzten 10 aufsteigend |
| `ls -l /bin \| sort -r \| head` | ... + die letzten 10 absteigend |
| `ls -l /bin \| sort -r \| head \| sort` | ... + die letzten 10 aufsteigend |
| `ls -l /bin \| sort +4nr \| head \| sort` | ... + die letzten 10 aufsteigend |

Mit Hilfe der Option `noclobber` kann verhindert werden, dass Dateien per Ein/Ausgabe-Umlenkung überschrieben werden können, falls sie *schon existieren*. Das **Setzen dieser Option** erfolgt per:

```
set -o noclobber
```

Das **Zurücksetzen dieser Option** erfolgt per:

```
set +o noclobber
```

### 3.3.1 Here-Dokument

Um Kommando-Aufrufe und Daten in einem Skript gemeinsam pflegen zu können, ist die Angabe von Daten zu einem Kommando als sogenanntes **Here-Dokument** möglich. Das folgende Beispiel (`EOF` steht für „end of file“)

```
sort << EOF
xx
gg
dd
aa
EOF
```

übergibt dem `sort`-Kommando die Textzeilen zwischen den beiden `EOF` zum Sortieren. Der Text `EOF` ist frei wählbar und muss am Ende auf einer Zeile für sich alleine stehen, damit er erkannt wird. Probiert man dieses Kommando auf der Kommandozeile aus, so gibt die Shell zur Kennzeichnung nach der 1. Zeile bis zur Eingabe des Textes `EOF` auf einer Zeile für sich den sogenannten **Fortsetzungs-Prompt** `>␣` aus (`PS2`).

## 3.4 Hintergrund-Prozesse (Jobs)

Hintergrund-Prozesse (Jobs) werden durch Anhängen von `&` an ein Kommando gestartet:

```
find / -name "*.conf" -print > /tmp/conf-files.txt &
```

Beim Start wird zu jedem **Job** seine **Job-Nummer** `%NR` ausgegeben, diese Job-Nummer beginnt pro Terminal bei 1. Die Job-Nummer kann statt der Prozess-Nummer für Steuerungsaufgaben verwendet werden (z.B. im Kommando `kill %NR`).

Das Kommando wird dann im **Hintergrund** ausgeführt und auf der Shell-Ebene kann parallel weitergearbeitet werden. Wird keine Umlenkung oder Pipe im Kommando verwendet, so erfolgen die Ein- und Ausgaben des Hintergrund-Prozesses von der Tastatur und auf dem Bildschirm (d.h. sie stören die interaktive Nutzung der Shell). Daher sollten Hintergrund-Prozesse nur zusammen mit Umlenkungen und Pipes eingesetzt werden.

Typischerweise werden Langläufer als Hintergrund-Prozess gestartet und ihre Ein/Ausgaben von Datei gelesen bzw. auf Datei geschrieben.

Weitere Kommandos im Zusammenhang mit Hintergrund-Prozessen sind `Strg-Z`, `bg` (background), `fg` (foreground) und `jobs` (alle Hintergrund-Prozesse einer Shell auflisten).

## 3.5 Shell-Variable

Shell-Variablen sind Paare der Form `NAME=Wert`, die unter dem Namen `NAME` den Wert (Text) `Wert` speichern. Jede Shell verwaltet in ihrem Datenspeicher eine (beliebig lange) Liste davon. Diese Variablen steuern das Verhalten der Shell oder von daraus aufgerufenen Programmen, sie können beliebig gesetzt, verändert und gelöscht werden. Neben einer Reihe von vordefinierten Standard-Variablen kann der Benutzer beliebige weitere Shell-Variablen anlegen (und auch wieder löschen). Die wichtigsten **Standard-Variablen** sind:

| Name | Bedeutung |
|---|---|
| CDPATH | Suchpfad für cd-Kommando |
| COLUMNS | Terminal-Spaltenbreite |
| DISPLAY | X-Terminalname |
| EDITOR | Default-Editorname (auch VISUAL) |
| HOME | Standardverzeichnis für cd (Heimat-Verzeichnis) |
| HOSTNAME | Rechnername |
| IFS | Worttrenner (Internal Field Separator) |
| LANG | Spracheinstellung |
| LANGUAGE | Spracheinstellung |
| LC_ALL | Spracheinstellung (locale) |
| LC_MESSAGE | Spracheinstellung (locale) |
| LC_CTYPE | Spracheinstellungen (?) |
| LC_NUMERIC | Spracheinstellungen (Zahl) |
| LC_TIME | Spracheinstellungen (Uhrzeit) |
| LC_COLLATE | Spracheinstellungen (Sortierung) |
| LC_MONETARY | Spracheinstellungen (Geld) |
| LC_MESSAGES | Spracheinstellungen (Nachrichten) |
| LC_PAPER | Spracheinstellungen (Papierformate) |
| LC_NAME | Spracheinstellungen (Namen) |
| LC_ADDRESS | Spracheinstellungen (Adresse) |
| LC_TELEPHONE | Spracheinstellungen (Telefonnummer) |
| LC_MEASUREMENT | Spracheinstellungen (Maßeinheit) |
| LC_IDENTIFICATION | Spracheinstellungen (?) |
| NLSPATH | Spracheinstellungen (?) |
| LESS | less-Standard-Optionen |
| LINES | Terminal-Zeilenanzahl |
| LOGIN | Loginname |
| LOGNAME | Loginname |
| LPDEST | Default-Druckername (Line Printer Destination, auch PRINTER) |
| LS_COLORS | Farbdefinition ls-Kommando |
| LS_OPTIONS | Standard-Optionen ls-Kommando |
| MANPATH | Suchpfad für man-Kommando |
| OLDPWD | Vorheriges Arbeitsverzeichnis (PWD) |
| PAGER | Default-Anzeigeprogramm (z.B. more, less) |
| PATH | Suchpfad für Kommando-Aufruf |
| PRINTER | Default-Druckername (auch LPDEST) |
| PS1 | Eingabe-Prompt (Std: $␣) |
| PS2 | Fortsetzungs-Prompt (Std: >␣) |
| PS3 | Select-Prompt (bei select, Std: #?) |
| PS4 | Debugging-Prompt (nach set -x, Std: +␣) |
| PWD | Aktuelles Arbeitsverzeichnis (Print Working Directory) |
| RANDOM | Zufallszahl (0–32767) |
| SHELL | Name der aktuellen Shell |
| SHLVL | Shell-Verschachtelungstiefe |
| TERM | Terminalname |
| TZ | Zeitzone (Time Zone) |
| USER | Benutzername |
| USERNAME | Benutzername |
| VISUAL | Default-Editorname (auch EDITOR) |

Die Kommandos zum Setzen, Anzeigen, Verwenden und Löschen von Shell-Variablen in der

`sh` lauten:

| | |
|---|---|
| *VAR*=*TEXT* | Erzeugt eine Shell-Variable (*kein* Leerzeichen um = verwenden!) |
| `$`*VAR* | Zugriff auf den Wert (Inhalt) einer Shell-Variablen |
| `${`*VAR*`}xxx` | Alternativer Zugriff, falls direkt dahinter Text `xxx` steht |
| *VAR*= | Löschen einer Shell-Variablen (anschließend ist sie *leer*) |
| `unset` *VAR* | Löschen einer Shell-Variablen (anschließend ist sie *undefiniert* != leer) |
| `set` | Alle Shell-Variablen auflisten |

In der `csh`/`tcsh` lauten diese Kommandos:

| | |
|---|---|
| `set` *VAR* = *TEXT* | Erzeugt eine Shell-Variable (Leerzeichen um = notwendig!) |
| `@` *VAR* = *TEXT* | Analog |
| `$`*VAR* | Zugriff auf den Wert (Inhalt) einer Shell-Variablen |
| `${`*VAR*`}xxx` | Alternativer Zugriff, falls direkt dahinter Text `xxx` steht |
| `set` *VAR* = | Löschen einer Shell-Variablen (anschließend ist sie *leer*) |
| `unset` *VAR* | Löschen einer Shell-Variablen (anschließend ist sie *undefiniert* != leer) |
| `set` | Alle Shell-Variablen auflisten |
| `@` | Analog |

**Hinweis**: Shell/Umgebungs-Variablen werden per Konvention GROSS geschrieben, obwohl die GROSS/kleinschreibung zählt. In der `csh`/`tcsh` haben einige Variablen zwei Namen — einen GROSS und einen klein geschriebenen — die immer gleichen Wert enthalten (z.B. `PATH` und `path`).

### 3.5.1 Umgebungs-Variable

Ein spezieller Typ von Shell-Variablen sind die sogenannten **Umgebungs-Variablen**, sie sind eine **Teilmenge** der Shell-Variablen (in der `sh` und allen davon abgeleiteten Shells). In der `csh`/`tcsh` sind die Shell- und die Umgebungs-Variablen getrennt.

Jeder Prozess besitzt Umgebungs-Variablen (nicht nur die Shell), sie werden vom Elternprozess an seine Kindprozesse **vererbt** (Shell-Variablen nicht). Die Kindprozesse können die vererbte Liste beliebig ändern und erweitern und an ihre eigenen Kindprozesse weitervererben. „Zurückvererben“ können Kindprozesse ihren Umgebungsbereich an Elternprozesse nicht.

Die Kommandos zum Erzeugen, Setzen und Auflisten in der `sh` (und allen davon abgeleiteten Shells) lauten (alle anderen Kommandos sind analog zu denen bei Shell-Variablen):

| | |
|---|---|
| `export` *VAR* | Erzeugt eine Umgebungs-Variable |
| `export` *VAR*=*TEXT* | Erzeugt eine Umgebungs-Variable (und belegt sie) |
| `env` | Alle Umgebungs-Variablen auflisten |
| `printenv` | Alle Umgebungs-Variablen auflisten |

In der `csh`/`tcsh` lauten diese Kommandos:

| | |
|---|---|
| `setenv` *VAR* | Erzeugt eine Umgebungs-Variable |
| `setenv` *VAR TEXT* | Erzeugt eine Umgebungs-Variable (und belegt sie) |
| `unsetenv` *VAR TEXT* | Löschen einer Umgebungs-Variablen |
| `env` | Alle Umgebungs-Variablen auflisten |

| | |
|---|---|
| `printenv` | Alle Umgebungs-Variablen auflisten |

### 3.5.2 Hinweise

- **Shell-Variablen** werden auch als **lokale Variablen** bezeichnet, sie sind auf die Shell beschränkt und werden nicht an andere Kommandos oder Sub-Shells weitergegeben.
- **Jede Shell** hat ihre eigene Liste von **Shell-Variablen**, die zusammen mit ihr erzeugt werden und zusammen mit ihr auch wieder sterben.
- **Umgebungs-Variablen** werden auch als **globale Variablen** bezeichnet, sie sind nicht auf die Shell beschränkt, sondern werden an andere Kommandos oder Sub-Shells weitergegeben.
- **Jeder Prozess** hat seine eigene Liste von **Umgebungs-Variablen**, die zusammen mit ihm erzeugt werden und zusammen mit ihm auch wieder sterben.
- Für die Ausführung von **Shell-Skripten** (Kommando/Batch-Prozeduren) wird immer eine **Sub-Shell** (d.h. ein Kindprozess) gestartet.
- **Aliase und Funktionen** werden *nicht* an Sub-Shells *vererbt*.

### 3.5.3 Beispiele

Die Suchpfad-Variable `PATH` anzeigen/erweitern/verkürzen:

```
echo $PATH                      Inhalt der Variable PATH anzeigen
OLDPATH="$PATH"                 Variable PATH nach OLDPATH sichern
PATH=                           Variable PATH löschen
PATH="/bin:/usr/bin"            Variable PATH gleich /bin:/usr/bin setzen
PATH="$PATH:."                  Variable PATH hinten um :. erweitern
echo $PATH                        → /bin:/usr/bin:.
PATH="/sbin:$PATH"              Variable PATH vorn um /sbin: erweitern
echo $PATH                        → /sbin:/bin:/usr/bin:.
PATH=`echo $PATH |              Verzeichnis /usr/bin aus Variable PATH entfernen
  sed "s#^/usr/bin:##" |          am Anfang
  sed "s#:/usr/bin:#:#" |         in der Mitte
  sed "s#:/usr/bin$##"`           am Ende
echo $PATH                        → /sbin:/bin:.
PATH="$OLDPATH"                 Variable PATH aus OLDPATH wieder herstellen
```

Das folgende Skript `var.sh` (mit führenden Zeilennummern) dient zur Verdeutlichung des unterschiedlichen Verhaltens von Shell-Variablen und Umgebungs-Variablen:

```
# Übergebene Werte ausgeben
echo "SHVAR=$SHVAR"
echo "ENVVAR=$ENVVAR"

# Werte neu belegen
```

```
SHVAR="wert1"
ENVVAR="wert2"

# Neu belegte Werte ausgeben
echo "SHVAR=$SHVAR"
echo "ENVVAR=$ENVVAR"
```

Die Ausführung des Skriptes `var.sh` ergibt folgende Ausgaben:

```
$ SHVAR=sss              Variable SHVAR in Login-Shell belegen
$ ENVVAR=eee             Variable ENVVAR in Login-Shell belegen
$ export ENVVAR          ENVVAR ist Umgebungs-Variable (wird „vererbt“)
$ echo $SHVAR $ENVVAR    Variableninhalt in Login-Shell ausgeben
sss eee                    → Ergebnis
$ sh var.sh              Skript var.sh aufrufen (→ Sub-Shell)
SHVAR=                     → Shell-Variable ist leer, da nicht „vererbt“
ENVVAR=eee                 → Umgebungs-Variable ist „vererbt“ worden
SHVAR=wert1                → Variableninhalt in Sub-Shell
ENVVAR=wert2               → Variableninhalt in Sub-Shell
$ echo $SHVAR $ENVVAR    Variableninhalt in Login-Shell ausgeben
sss eee                    → Ergebnis
```

## 3.6 Quotierung

Die Shell kennt eine Reihe von **Meta/Sonderzeichen**, die nicht für sich selber stehen, sondern die von ihr interpretiert werden und bestimmte Funktionen auslösen (26 Stück!):

```
*   ?   [   ]   =   $   <   >   |   &   \
"   '   `   ;   (   )   {   }   ~   ^   !   #
<Space>   <Tabulator>   <Newline>
```

Sollen diese Zeichen nicht ihre Sonderbedeutung haben, sondern als ganz normaler Text interpretiert werden, so sind sie vor der Shell zu **schützen**, der Fachausdruck dafür ist **quotieren** (zitieren). Die Shell kennt drei verschiedene Möglichkeiten des Schützens von Zeichen, die für unterschiedliche Anwendungsfälle benötigt werden:

| | |
|---|---|
| `"..."` | Alle Sonderzeichen bis auf `$`, `` ` ``, `$(...)` und `\` in `...` abschalten |
| `'...'` | Alle Sonderzeichen in `...` abschalten (Quote, Tick) |
| $\backslash x$ | Genau ein (das folgende) Sonderzeichen $x$ abschalten (Backslash) |

Folgende Tabelle gibt eine Übersicht über die von den Quotierungszeichen jeweils geschützten ($\times$) bzw. nicht geschützten (–) Sonderzeichen (E = Ende, v = verbose):

| | `\` | `$VAR` | `*?[]` | `` `...` `` | `$(...)` | `"` | `'` | CR | `!` | Whitespace | Rest |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `"..."` | – | – | × | – | – | E | × | v | – | × | × |
| `'...'` | × | × | × | × | × | × | E | v | – | × | × |
| $\backslash x$ | × | × | × | × | × | × | × | × | × | × | × |

Beispiele:

| | |
|---|---|
| `echo * 'date' > x $TERM &` | Alle Sonderzeichen ausgewertet |
| `echo "* 'date' > x $TERM &"` | Ein Argument, nur `'date'` und `$TERM` ausgewertet |
| `echo '* 'date' > x $TERM &'` | Ein Argument, kein Sonderzeichen ausgewertet |
| `echo \* \'date\' \> x \$PATH \&` | Sechs Argumente kein Sonderzeichen ausgewertet |
| `echo \*\ \'date\'\ \>\ x\ \$PATH\ \&` | Ein Argument kein Sonderzeichen ausgewertet |
| `echo\ \*\ \'date\'\ \>\ x\ \$PATH\ \&` | Ein Kommando (1. Wort) → Fehler |

## 3.7 Kommando-Substitution

In vielen Situationen kann ein Kommando zum Erzeugen einer Liste von Werten verwendet und das Ergebnis dann als Argument in einem anderen Kommando eingesetzt werden. Dieser Vorgang wird als **Kommando-Substitution** bezeichnet. Hierzu wird das Zeichen `` ` `` (backquote, backtick) verwendet. **Achtung**: *Verwechslungsgefahr* mit dem Quotierungszeichen `'` (Hochkomma)!

Das innerhalb von `` `...` `` stehende Kommando wird ausgeführt, und seine *Standard-Ausgabe* wird an dieser Stelle eingesetzt (**Kommando-Ersetzung**). Erst danach wird das eigentliche Kommando ausgeführt. Es ist sogar möglich, Kommando-Substitutionen ineinander zu verschachteln.

In der `bash`, `ksh` und `zsh` ist als **alternative Schreibweise** `$(...)` möglich. Diese Schreibweise ist vorzuziehen, da sie besser lesbar ist, die Form dem Zugriff auf Variablenwerten `$VAR` gleicht und Verschachtelungen mit ihr leichter realisiert werden können.

Mit dem folgendem Kommando können z.B. alle C-Dateien, die mindestens 1× den Text `error` enthalten, herausgesucht und der Editor `vi` mit ihren Dateinamen als Argument aufgerufen werden (`-l`=list):

```
vi `grep -l "error" *.c`    # Alte Syntax
vi $(grep -l "error" *.c)   # Neue Syntax
```

Die Liste, die in einer Kommando-Ersetzung verwendet wird, muss keine Liste von Dateinamen sein. Hier z.B. wird an alle eingeloggten Benutzer die Datei `brief.txt` als Mail geschickt:

```
mail `who | cut -f1 | sort | uniq` < brief.txt    # Alte Syntax
mail $(who | cut -f1 | sort | uniq) < brief.txt   # Neue Syntax
```

Als weiteres Beispiel werden im folgenden Kommando alle `bash`-Prozesse herausgesucht und beendet (`-d`=delimiter):

```
kill -9 `ps -e | grep "bash" | cut -d" " -f2/3`    # Alte Syntax
kill -9 $(ps -e | grep "bash" | cut -d" " -f2/3)   # Neue Syntax
```

**Hinweis**: Die „**vertikale**“ Ausgabe von `grep` (durch Zeilenvorschub getrennte Zeilen)und die „**horizontale**“ Art (durch Leerzeichen getrennte Worte), in der man normalerweise Argumente auf der Kommandozeile eingibt, machen für die Shell keinen Unterschied. Die Shell verarbeitet beides problemlos, innerhalb von Backquotes zählen auch Zeilenvorschübe als Argumenttrenner.

## 3.8 Interaktive Arbeitsweise

### 3.8.1 Prompt anpassen

Die Umgebungs-Variable `PS1` legt das Aussehen des **Eingabe-Prompts** fest. Sie ist entweder systemweit in `/etc/profile` oder für einzelne Benutzer in `~/.profile` zu initialisieren. Folgende Sonderzeichen in `PS1` setzen spezielle (variablen) Komponenten im Eingabe-Prompt ein:

| Code | Bedeutung | Begriff |
|---|---|---|
| `\d` | Datum im Format `Sun Dec 24` | date |
| `\h` | Rechner-Name | host |
| `\n` | Zeilenvorschub | newline |
| `\w` | Arbeitsverzeichnis | working directory |
| `\W` | Arbeitsverzeichnis (letzter Teil) | working directory |
| `\s` | Shell-Name | shell |
| `\t` | Zeit im Format `hh:mm:ss` | time |
| `\u` | Benutzer-Name | user |
| `\$` | `$` für normalen Benutzer, `#` für `root` | prompt |
| `\#` | Nummer des aktuellen Kommandos | number |
| `\!` | History-Nummer des aktuellen Kommandos | history |
| `\[` | Start zu ignorierende Zeichen für Kmdo-Edit | left bracket |
| `\]` | Ende zu ignorierende Zeichen für Kmdo-Edit | right bracket |

Der Standardwert von `PS1` beträgt für den Benutzer `root` (`Rechner:Arbeitsverz.␣#␣`)

```
PS1="\h:\w # "
```

und für normale Benutzer (`Benutzer@Rechner:Arbeitsverz.␣>␣`)

```
PS1="\u@\h:\w > "
```

**Tipp**: Beachten Sie das Leerzeichen am Ende, damit die Kommandoeingabe etwas abgesetzt vom Eingabe-Prompt beginnt. Der Eingabe-Prompt sollte auch nicht zu lang werden, da er sonst unübersichtlich ist und wenig Platz für die Eingabe der Kommandozeile übrig lässt. Eine nützliche farbig hinterlegte Variante des Eingabe-Prompts wird so definiert (`Benutzer@Rechner:Arbeitsverz.␣BefehlsNr␣`):

```
PS1="\[$(tput bold)$(tput setaf 7)$(tput setab 0)\]\u@\h:\w \!\[$(tput sgr0)\] "
```

Weitere Prompt-Variablen:

- Die **Fortsetzungs-Prompt-Variable** `PS2` (Secondary Prompt) legt den Text fest, der ausgegeben wird, wenn die Shell ein Kommando als noch nicht abgeschlossen betrachtet (Defaultwert `>␣`).
- Die **Select-Prompt-Variable** `PS3` (Third Prompt) legt den Text fest, der ausgegeben wird, wenn die Shell die Eingabe für ein `select`-Kommando abfragt (Defaultwert `#?`).
- Die **Debug-Prompt-Variable** `PS4` (Fourth Prompt) legt den Text fest, der im **Debugging-Modus** (`set -x`) vor jedem Kommando ausgegeben wird, bevor es ausgeführt wird (Defaultwert `+␣`).

### 3.8.2 Filename-Completion

Teilweise eingegebene Kommandos, Pfadnamen, Variablennamen, Benutzernamen und Hostnamen können mit Hilfe der **Tabulator-Taste** `<TAB>` automatisch vervollständig werden (sofern die Eingabe bereits eindeutig ist). Ist die Eingabe noch mehrdeutig, dann listet **zweimaliges Drücken** der Tabulator-Taste die noch verfügbaren Möglichkeiten auf.

```
CMD... <TAB> <TAB>
... PATH <TAB> <TAB>
... $VAR <TAB> <TAB>
... ~USER <TAB> <TAB>
... @HOST <TAB> <TAB>
```

**Hinweis**: Hierdurch kann man schneller arbeiten, vermeidet Tippfehler und muss sich nicht so viel merken.

### 3.8.3 History-Mechanismus

Login-Shells merken sich alle eingegebenen Kommandos und erlauben ihre spätere Wiederholung. Wird die Shell beendet, dann speichert sie diese Liste von Kommandos in einer History-Datei (z.B. `~/.history` oder `~/.bash_history` und lädt diese beim erneuten Login wieder.

Mit den Cursor-Tasten Auf/Ab kann auf die letzten Kommandos zugegriffen werden. Ein „Return“ löst das gerade angezeigte Kommando erneut aus. Der Befehl `history` gibt alle alten Kommandos aus, der Befehl `history 20` die letzten 20 Kommandos. Mit `!NR` kann das Kommando mit der Nummer `NR` dieser Liste wiederholt werden. Mit `!!` wird das letzte eingegebene Kommand wiederholt. Mit `!TEXT` wird das letzte Kommando mit `TEXT` am Kommandoanfang ausgeführt. Mit `Strg-R TEXT` wird die Kommandliste von hinten nach vorne nach einem Kommando mit `TEXT` durchsucht (wiederholte Eingabe von `Strg-R` sucht dann weiter).

| | |
|---|---|
| `history` | Alle alten Kommandos auflisten |
| `history 20` | Die letzten 20 Kommandos auflisten |
| `!123` | Kommando Nr 123 wiederholen (sofort) |
| `!123:p` | Kommando Nr 123 zum Editieren holen (nicht aufrufen, print) |
| `!vi` | Editor erneut aufrufen |

| | |
|---|---|
| `!!` | Vorheriges Kommando wiederholen |
| `Strg-R ssh` | In alten Kommandos nach `ssh` suche (Strg-R wiederholt) |

Folgende Zeile listet alle alten Kommandos auf, die `ssh` enthalten:

```
history | grep "ssh"
```

### 3.8.4 Aliase

Aliase sind einfach eine **Ersetzung** des 1. Wortes auf der Kommandozeile, die durchgeführt wird, bevor das Kommando ausgeführt wird. Alle anderen Argumente bleiben auf der Kommandozeile so stehen, wie sie eingegeben wurden.

| | |
|---|---|
| `alias ls='/bin/ls -lF'` | Alias `ls` definieren (Rekursion vermeiden!) |
| `alias hg='history\|grep'` | Suche in alten Kommandos abkürzen |
| `ls *.c *.h` | → wird zu `/bin/ls -lF *.c *.h` expandiert |
| `alias` | Alle Aliase anzeigen |
| `alias ls` | Alias `ls` anzeigen |
| `unalias ls` | Alias `ls` löschen |

## 3.9 Reihenfolge/Vorrang der Shell-Mechanismen

Aufgrund der Vielzahl von Mechanismen zur Kommandozeilen-Manipulation in den diversen Shells ist es wichtig, ihren gegenseitigen Vorrang zu kennen. Hier die Reihenfolge der Schritte, in der die C-Shell eine Kommandozeile interpretiert:

1. History-Ersetzung (`!`) [`csh`]
2. Eingabezeile in Worte (**Token**) zerlegen (inkl. Sonderzeichen)
3. History-Liste updaten [`csh`]
4. Quotierung interpretieren (`"..." '...' \`$x$)
5. Alias-Ersetzung [`csh`]
6. Ein/Ausgabe-Umlenkung (`< > 2> >> << | ...`)
7. Variablen-Ersetzung (`$VAR`)
8. Kommando-Ersetzung (`` `cmd ...` `` `$(cmd ...)`)
9. Dateimuster expandieren (`? * [...] [!...] [^...] {...,...}`)
10. Pfadsuche (`PATH`)
11. Hintergrund-Prozess starten (`&`)

Die Bourne-Shell verhält sich analog, allerdings führt sie keine History- oder Alias-Ersetzung aus (die mit [csh] gekennzeichneten Schritte 1, 3 und 5).

**Hinweis**: Die History-Ersetzung wird zuerst durchgeführt. Daher können Quotes ein ! nicht vor der Shell schützen; die Shell sieht das Ausrufungszeichen und fügt ein Kommando aus der History-Liste ein, bevor sie überhaupt über Quotes nachdenkt. Um die Interpretation eines ! als History-Ersetzung zu verhindern, ein Backslash \! davor angeben.

Die obigen Schritte werden nun anhand eines einfachen Kommandos durchgegangen, um ein Gefühl für die Bedeutung von *„Die Shell führt Variablen-Ersetzung nach Alias-Ersetzung durch“* zu bekommen. Hier die Kommando-Zeile, sie enthält sowohl Leerzeichen als auch Tabulatoren:

```
ls -l    $HOME/* `grep -l error *.c` |     grep "Mar  7" | !!
```

Der Ablauf:

1. Der History-Operator (!) ist einmal vorhanden, die Bedeutung von !! ist „**Wiederholung des vorherigen Befehls**“. Angenommen der vorherige Befehl lautete more (etwas unsinnig, okay), dann wird !! durch more ersetzt (die Bourne-Shell würde diesen Schritt überhaupt nicht ausführen) und die Befehlszeile sieht folgendermaßen aus:

   ```
   ls -l    $HOME/* `grep -l error *.c` |     grep "Mar  7" | more
   ```

2. Die Kommandozeile wird gemäß dem Leerraum (nicht innerhalb Backquotes!) in die **Token** ls, -l, $HOME/*, `grep -l error *.c`, |, grep, Mar␣␣7, |, und more zerlegt. Die Shell ignoriert die Anzahl der Leerraum-Zeichen (Leerzeichen und Tabulatoren) zwischen den Token einer Kommandozeile. Jeder nicht quotierte Leerraum leitet ein neues Token ein. Die Shell behandelt Optionen (wie -l) nicht speziell, sie werden wie jedes andere Token an das auszuführende Kommando übergeben[1] und das Kommando entscheidet, wie sie zu interpretieren sind. Die Anführungszeichen verhindern auch, dass die Shell Mar␣␣7 in zwei Token zerlegt oder die beiden Leerzeichen darin ignoriert[2] — obwohl die eigentliche Interpretation der Anführungszeichen erst später folgt. An diesem Punkt hat die Kommandozeile folgende Form:

   ```
   ls -l $HOME/* `grep -l error *.c` | grep "Mar  7" | more
   ```

3. Die Shell legt die Kommandozeile in der History-Liste ab (die Bourne-Shell würde diesen Schritt ebenso nicht ausführen).

4. Die Shell erkennt die doppelten Anführungszeichen um Mar␣␣7 und merkt sich, dass darin keine Dateinamen-Expansion (die noch folgt) durchzuführen ist.

[1] Die Konvention, Optionen mit einem Bindestrich (-) beginnen zu lassen, ist einfach eine Konvention: Obwohl die Behandlung von Optionen standardisiert ist, kann jedes Kommando seine Optionen nach eigenem Belieben interpretieren

[2] In einer von ls -l erzeugten Dateiliste stehen zwei Leerzeichen vor Tagen kleiner 10 (sie werden in einem 3 Zeichen breiten Feld ausgegeben).

5. Die Shell erkennt die beiden Pipe-Symbole | und führt alles Notwendige für den Pipeline-Mechanismus durch.

6. Die Shell erkennt die Umgebungs-Variable HOME und ersetzt sie durch ihren Wert (/home/mike). An diesem Punkt hat die Kommandozeile folgende Form:

```
ls -l /home/mike/* `grep -l error *.c` | grep "Mar  7" | more
```

7. Die Shell sucht nach Backquotes, führt jedes Kommando darin aus und fügt das Ergebnis in die Kommandozeile ein. In diesem Falle wird also das Kommando grep -l error *.c ausgewertet (mit dem gleichen Ablauf wie gerade beschrieben, d.h. rekursiv!) und das Ergebnis, nämlich die Namen aller C-Dateien, die das Wort error mindestens 1x enthalten, in die Kommandozeile eingefügt. (Falls innerhalb der Backquotes Wildcards oder Variablen vorkommen, werden sie erst interpretiert, wenn die Shell das Kommando in den Backquotes ausführt):

```
ls -l /home/mike/* aaa.c...zzz.c | grep "Mar  7" | more
```

8. Die Shell sucht nach Wildcards, sieht den * und expandiert entsprechend die Dateinamen, das Ergebnis hat etwa folgende Form:

```
ls -l /home/mike/ax.../home/mike/zip aaa.c...zzz.c | grep "Mar  7" | more
```

9. Die Shell führt die Kommandos ls, grep und more aus, mit den vorher erwähnten Pipes, die die Ausgabe von ls mit der Eingabe von grep und die Ausgabe von grep mit der Eingabe von more verbinden.

# 4 Shell-Skripte

## 4.1 Aufruf-Arten in der Shell

Je nach Aufruf eines Kommandos oder Shell-Skripts wird die **Pfadsuche** durchgeführt oder nicht, muss das x**-Recht** gesetzt sein oder nicht, wird eine **Sub-Shell** gestartet oder nicht oder erfolgt eine **Rückkehr** zur ursprünglichen Shell oder nicht (der Befehl . bzw. source liest eine Datei ein und verwendet dabei die Pfadsuche):

| **Aufruf** | PATH **notwendig** | x**-Recht notwendig** | **Sub-Shell gestartet** | **Rückkehr zur Ausgangs-Shell** |
|---|---|---|---|---|
| cmd.sh | × | × | × | × |
| sh cmd.sh | – | – | × | × |
| ./cmd.sh | – | × | × | × |
| . cmd.sh | × | – | – | × |
| . ./cmd.sh | – | – | – | × |
| exec cmd.sh | – | – | – | – |

## 4.2 Parameter-Übergabe

Neben den normalen Shell-Variablen kennt die Shell eine Reihe von **speziellen Variablen**, die vor allem für die Übergabe von Parametern an Shell-Skripte und die Steuerung von Shell-Skripten vorhanden sind:

| Variable | Beschreibung |
|---|---|
| `$0` | Skript-Name |
| `$1..$9` | Kommandozeilen-Argumente Nummer 1-9 (Zugriff auf `${10}...` mit `shift`) |
| `$#` | Anzahl Kommandozeilen-Argumente |
| `$* $@` | Alle Kommandozeilen-Argumente (einzeln quotiert bei `$@`) |

Das folgende Skript dient zur Illustration der Verwendung obiger Variablen:

```
echo "Anzahl Argumente \$# = <$#>"
echo "Skript-Name      \$0 = <$0>"
echo "1. Argument      \$1 = <$1>"
echo "2. Argument      \$2 = <$2>"
echo "3. Argument      \$3 = <$3>"
echo "4. Argument      \$4 = <$4>"
echo "5. Argument      \$5 = <$5>"
echo "6. Argument      \$6 = <$6>"
echo "7. Argument      \$7 = <$7>"
echo "8. Argument      \$8 = <$8>"
echo "9. Argument      \$9 = <$9>"
echo "Alle Argumente   \$* = <$*>"
echo "Alle Argumente   \$@ = <$@>"
```

Der folgende Aufruf dieses Skriptes mit den angegebenen Argumenten

```
args.sh aaa "bbb ccc" '   ddd   ' "" eee
```

ergibt folgende Ausgabe:

```
Anzahl Argumente $# = <5>
Skript-Name      $0 = <args.sh>
1. Argument      $1 = <aaa>
2. Argument      $2 = <bbb ccc>
3. Argument      $3 = <   ddd   >
4. Argument      $4 = <>
5. Argument      $5 = <eee>
6. Argument      $6 = <>
7. Argument      $7 = <>
8. Argument      $8 = <>
9. Argument      $9 = <>
Alle Argumente   $* = <aaa bbb ccc    ddd    eee>
Alle Argumente   $@ = <aaa bbb ccc    ddd    eee>
```

**Hinweis**: Um Kommandozeilen-Parameter 1:1 an ein anderes Skript oder eine Funktion weiterzureichen, folgende Syntax verwenden.

```
${1+"$@"}    # Kein Parameter --> 1. Parameter, sonst alle einzeln quotiert
```

## 4.3 Spezial-Variablen

Weiterhin kennt die Shell noch folgende speziellen Variablen:

| Variable | Beschreibung |
|---|---|
| `$?` | Exit-Status des letzten ausgeführten Kommandos |
| `$$` | Prozess-ID der das Skript ausführenden Shell |
| `$!` | Prozess-ID des letzten im Hintergrund gestarteten Kommandos |
| `$-` | Shell-Optionen |

Das folgende Skript dient zur Illustration der Verwendung obiger Variablen:

```
grep TEXT Diese_Datei_existiert_nicht 2> /dev/null
echo "Exit-Status                     = <$?>"
echo "Aktuelle PID                    = <$$>"
sleep 100 &   # Hintergrund-Prozess starten
echo "PID letzter Hintergrund-Prozess = <$!>"
echo "Shell-Optionen                  = <$->"
```

Der Aufruf dieses Skriptes ergibt in etwa folgende Ausgabe:

```
Exit-Status                     = <2>
Aktuelle PID                    = <2371>
PID letzter Hintergrund-Prozess = <2372>
Shell-Optionen                  = <Bh>
```

## 4.4 Das Kommando `test`

Das Kommando `test` wird in Shell-Anweisungen häufig verwendet, um eine **Bedingung** zu überprüfen und entsprechend zu verzweigen. Die Aufrufsyntax lautet (2 Möglichkeiten, die zweite ist vorzuziehen):

`if test` *EXPR* `...` Kommando-Schreibweise
`if [` *EXPR* `] ...` Programm-Schreibweise

Als Bedingung *EXPR* sind folgende Angaben möglich:

| Ausdruck | Beschreibung | Name |
|---|---|---|
| `-n` *TEXT* | *TEXT* ist nicht leer | nonzero |
| `-z` *TEXT* | *TEXT* ist leer | zero |
| *TEXT1* `=` *TEXT2* | *TEXT1* und *TEXT2* sind gleich (alte Syntax) | |
| *TEXT1* `==` *TEXT2* | *TEXT1* und *TEXT2* sind gleich (neue Syntax) | |
| *TEXT1* `!=` *TEXT2* | *TEXT1* und *TEXT2* sind verschieden | |
| *TEXT1* `<` *TEXT2* | *TEXT1* kleiner *TEXT2* (*nur bash!*) | |
| *TEXT1* `>` *TEXT2* | *TEXT1* größer *TEXT2* (*nur bash!*) | |
| *NUM1* `-eq` *NUM2* | Zahl *NUM1* ist gleich *NUM2* | equal |
| *NUM1* `-ne` *NUM2* | Zahl *NUM1* ist nicht gleich *NUM2* | not equal |
| *NUM1* `-le` *NUM2* | Zahl *NUM1* ist kleiner gleich *NUM2* | less equal |
| *NUM1* `-lt` *NUM2* | Zahl *NUM1* ist kleiner *NUM2* | less than |
| *NUM1* `-ge` *NUM2* | Zahl *NUM1* ist größer gleich *NUM2* | greater equal |
| *NUM1* `-gt` *NUM2* | Zahl *NUM1* ist größer *NUM2* | greater than |
| `-a` *FILE* | Datei *FILE* existiert (*nur bash!*) | available |
| `-e` *FILE* | Datei *FILE* existiert (*nur bash!*) | exists |
| `-s` *FILE* | Datei *FILE* ist *nicht* leer (Größe in Byte) | size |
| `-d` *FILE* | Datei *FILE* ist Verzeichnis | directory |
| `-f` *FILE* | Datei *FILE* ist normale Datei | file |
| `-L` *FILE* | Datei *FILE* ist symbolischer Link (*groß!*) | link |
| `-h` *FILE* | Datei *FILE* ist symbolischer Link (*nur bash!*) | |
| `-b` *FILE* | Datei *FILE* ist blockorientiert | block device |
| `-c` *FILE* | Datei *FILE* ist zeichenorientiert | character device |
| `-p` *FILE* | Datei *FILE* ist Named Pipe | pipe |
| `-S` *FILE* | Datei *FILE* ist Socket (*groß!*) | socket |
| `-t` *FILE* | Datei *FILE* ist Terminal | terminal |
| `-r` *FILE* | Datei *FILE* ist lesbar | readable |
| `-w` *FILE* | Datei *FILE* ist schreibbar | writable |
| `-x` *FILE* | Datei *FILE* ist ausführbar | executable |
| `-g` *FILE* | Datei *FILE* hat Group-ID Bit gesetzt | group id |
| `-k` *FILE* | Datei *FILE* hat Sticky-Bit gesetzt | sticky bit |
| `-u` *FILE* | Datei *FILE* hat User-ID Bit gesetzt | user id |
| `-O` *FILE* | Datei *FILE* gehört effektivem User (*groß!*) | Owner |
| `-G` *FILE* | Datei *FILE* gehört effektivem Group (*groß!*) | Group |
| `-N` *FILE* | Datei *FILE* geändert seit letztem Lesen (*nur bash!*) | New |
| *FILE1* `-nt` *FILE2* | Datei *FILE1* ist neuer als *FILE2* | newer than |
| *FILE1* `-ot` *FILE2* | Datei *FILE1* ist älter als *FILE2* | older than |
| *FILE1* `-ef` *FILE2* | Datei *FILE1* und *FILE2* haben ident. Inode | equal file |
| `!` *EXPR* | Negation von *EXPR* | not |
| *EXPR1* `-a` *EXPR2* | *EXPR1* und *EXPR2* | and |
| *EXPR1* `-o` *EXPR2* | *EXPR1* oder *EXPR2* | or |
| `\( ... \)` | Klammerung (*Quotierung notwendig!*) | |
| `-v` *VAR* | Shell-Variable *VAR* definiert (*nur bash!*) | variable |
| `-o` *OPT* | Shell-Option *OPT* gesetzt (*nur bash!*) | option |

## 4.5 Kontrollstrukturen

### 4.5.1 Sequenz

Hintereinander geschriebene Kommandos werden **sequentiell**, d.h. eines nach dem anderen durchgeführt:

| | |
|---|---|
| `CMD1; CMD2; ...` | Sequenz (CMD1 zuerst, dann CMD2, dann ... ausführen) |
| `CMD <Return> CMD2` | Sequenz (CMD1 zuerst, dann CMD2, dann ... ausführen) |
| `CMD1 & CMD2 & ...` | Parallel (CMD1 + CMD2 + ... gleichzeitig ausführen) |
| `CMD1 && CMD2` | UND (CMD2 nur ausführen falls CMD1 klappt) |
| `CMD1 \|\| CMD2` | ODER (CMD2 nur ausführen falls CMD1 schief geht) |
| `( CMD1; CMD2; ...)` | Kommandos zusammenfassen (mit Subshell) |
| `{ CMD1; CMD2; ...; }` | Kommandos zusammenfassen (ohne Subshell) |

### 4.5.2 Verzweigung

Die Shell verwendet den **Exit-Status von Kommandos** (meist `test`), um einen Wahrheitswert als Entscheidungsbasis für eine Verzweigung zu erhalten. Hat Kommando `CMD1`, usw. den Exit-Status 0, dann wird der `then`-Teil ausgeführt. Ansonsten wird der `elif`-Teil ausgeführt, d.h. das nächste Kommando ausgeführt und sein Exit-Status als Entscheidungskriterium verwendet.

Falls kein einziges Kommando nach `if`/`elif` den Exit-Status 0 liefert, werden die Kommandos im `else`-Teil ausgeführt (sofern dieser vorhanden ist).

```
if CMD1
then
    ...
{elif CMD2
 then
     ...}
[else
    ...]
fi
```

### 4.5.3 Mehrfachverzweigung

Soll ein Wert mit vielen Fällen verglichen werden, dann bietet sich die **Mehrfachverzweigung** `case...esac` an. Hier wird kein Exit-Status verglichen, sondern der Wert `TEXT` der Reihe nach mit allen Mustern `MUSTERx` von oben nach unten verglichen. Die Anweisungen `CMDx` nach dem ersten zutreffenden Muster werden ausgeführt. Als Muster sind **Globbing-Ausdrücke** mit folgenden Metazeichen erlaubt: `* ? [...] [^...] ...|...` (keine Regulären Ausdrücke!).

```
case TEXT in
   MUSTER1) CMD1 ... ;;
   MUSTER2) CMD2 ... ;;
   ...
   MUSTERn) CMDn ... ;;
   *)       CMD ... ;;
esac
```

### 4.5.4 Schleifen

Die `for`-Schleife arbeitet eine **Liste von Werten/Worten** ab. Wird keine Liste angegeben (ohne `in ...`), dann werden die beim Skriptaufruf übergebenen Parameter abgearbeitet.

`while` und `until` verwenden den **Exit-Status eines Kommandos**, um den Durchlauf der Schleife zu steuern. `while` läuft bei Exit-Status 0 erneut durch, `until` bei Exit-Status $\neq 0$. Von beiden Schleifen gibt es eine **kopfgesteuerte** und eine **fußgesteuerte** Variante. Bei der ersten Variante wird das Schleifeninnere evtl. gar nicht ausgeführt, bei der zweiten mindestens $1\times$.

```
for VAR                     for VAR in WORT1 WORT2 ... WORTn
do                          do
    ...                         ...
done                        done

while CMD                   do
do                              ...
    ...                     while CMD
done
                            do
until CMD                       ...
do                          until CMD
    ...
done
```

**Hinweis**: Eine Zählschleife in dem Sinne gibt es nicht (die Shell ist auch schlecht bei numerischen Berechnungen). In der `bash`/`ksh` gibt es allerdings doch eine Zählschleife mit folgender Syntax:

```
for (( N=0; N <= 100; ++N))
do
    ...
done
```

### 4.5.5 Funktionen

Funktionen sind eine Liste von Kommandos, die unter einem (Funktions)Namen zusammengefasst werden. Einer Funktion können **Argumente** übergeben werden (analog einem Skript) und eine Funktion kann einen **Exit-Status** zurückgeben. Mit ihrer Hilfe lassen sich z.B. Skripte strukturieren (zusammengehörige Code-Teile am Skriptanfang in Funktionen verpacken und am Skriptende nur noch die Funktionen aufrufen).

```
FUNCNAME()              # Definition (Variante 1)
{                       #
    ...                 #
    return              #
}                       #
                        #
function FUNCNAME()     # Definition (Variante 1)
```

```
{                       #
    ...                 #
    return              #
}                       #
                        #
FUNCNAME ARG1 ...       # Aufruf
```

Beispiele:

| | |
|---|---|
| `ls() {`<br>`  /bin/ls -lF`<br>`}` | Funktion `ls` definieren (Rekursion vermeiden!) |
| `ls() { /bin/ls -lF; }` | Analog in einer Zeile (`;` + Leerzeichen nötig!) |
| `ls *.c *.h` | → Inhalt der Funktion = `/bin/ls -lF *.c *.h` ausführen |
| `typeset -f` | Alle Funktionen anzeigen |
| `typeset -f ls` | Funktion `ls` anzeigen |
| `unset -f ls` | Funktion `ls` löschen |

### 4.5.6 Vorzeitiger Abbruch

| | |
|---|---|
| `break` [*N]* | Schleife `for`, `while`, `until` abbrechen (Tiefe *N*) |
| `continue` [*N]* | Schleife `for`, `while`, `until` wiederholen (Tiefe *N*) |
| `exit` [*EXITCODE]* | Skript abbrechen (*EXITCODE* 0-255 möglich, Std: 0) |
| `return` [*EXITCODE]* | Rücksprung aus Funktion (*EXITCODE* 0-255 möglich, Std: 0) |

### 4.5.7 Signale abfangen

```
trap "CMD; ..." SIGNAL...
```

### 4.5.8 Sonstige Kontrollstrukturen

| | |
|---|---|
| `. DATEI` | DATEI mit Skriptbefehlen einlesen (alte Syntax) |
| `source DATEI` | DATEI mit Skriptbefehlen einlesen (neue Syntax) |
| `: ...` | Leerer Befehl (... wird ignoriert) |
| `true` | Liefert immer Exit-Status 0 zurück |
| `false` | Liefert immer Exit-Status 1 zurück |

# 5 Quotierung

Auch wenn sie zunächst kompliziert erscheint, die Quotierung in der Shell ist eigentlich ganz einfach. Die generelle Idee dahinter ist: *Quotierung schaltet die spezielle Bedeutung von „Metazeichen“ für die Shell ab*. Man sagt auch, die Zeichen werden vor der Shell „geschützt“ oder „zitiert“. Es gibt drei Quotierungszeichen:

- **Doppeltes Anführungszeichen** `"`
- **Einfaches Anführungszeichen** `'`
- **Backslash** `\`

Das Zeichen `` ` `` (Backquote) ist *kein* Quotierungszeichen — es dient zur **Kommando-Ersetzung** und wird in Abschnitt 3.7 auf Seite 24 behandelt.

## 5.1 Spezielle Zeichen

Im folgenden sind die **Metazeichen** aufgelistet (22 Stück!), die für die *Bourne-Shell* Sonderbedeutung haben. Das Quotieren dieser Zeichen schaltet ihre Sonderbedeutung ab (die drei Quotierungs-Zeichen sind ebenfalls aufgelistet, man kann Quotierungszeichen also auch quotieren, mehr darüber später):

```
*   ?   [   ]   =   $   <   >   |   \
&   "   '   `   ;   (   )   ^   #
<Space>   <Tabulator>   <Newline>
```

In den anderen Shells haben zusätzlich folgende 4 Zeichen eine Sonderbedeutung:

```
~   {   }   !
```

Ein **Slash** „/“ hat für UNIX die Sonderbedeutung „Verzeichnistrenner“, zählt aber sonst als normales Zeichen — Quotierung ändert daher die Bedeutung von Slashes nicht.

Die **Whitespaces** — das sind standardmäßig die Zeichen Leerzeichen, Tabulator und Zeilenvorschub — haben auch eine Sonderbedeutung: *sie zerlegen die Eingabe auf der Kommandozeile in* **Worte** *(auch* **Token** *genannt), das sind die Kommandos, Optionen und Argumente*. Sie werden durch die Variable `IFS` (*internal field separator*) festgelegt, die nur in Ausnahmefällen verändert wird.

Stehen mehrere Whitespaces hintereinander, so zählen sie wie ein Whitespace.

## 5.2 Wie arbeitet die Quotierung?

Folgende Tabelle fasst die Quotierungs-Regeln zusammen:

| Quotierung | Erklärung |
|---|---|
| `"..."` | Alle Sonderzeichen bis auf `$`, `` `...` ``, `$(...)` und `\` in `...` abschalten. |
| `'...'` | Alle Sonderzeichen in `...` abschalten. |
| $\backslash x$ | Die Sonderbedeutung von $x$ abschalten. Am Zeilenende entfernt ein `\` den Zeilenvorschub (setzt die Zeile fort). |

Die Bourne-Shell liest die Eingabe — oder die Zeilen eines Shellskripts — Zeichen für Zeichen von links nach rechts ein (in Wirklichkeit verhält es sich etwas komplizierter, aber nicht im Rahmen der Quotierung). Liest die Shell eines der drei Quotierungszeichen, dann:

- Entfernt sie das Quotierungszeichen.
- Schaltet die Sonderbedeutung eines oder aller weiteren Zeichen bis zum Ende des quotierten Abschnitts gemäß den Regeln in obiger Tabelle ab.

### 5.2.1 Backslash

Ein **Backslash** (`\`) schaltet die Sonderbedeutung (falls überhaupt eine besteht) des nächsten Zeichens ab. Zum Beispiel ist `\*` das Zeichen Stern, aber kein Dateinamen-Wildcard mehr. Daher erhält im ersten Beispiel das Kommando `expr` die drei Argumente `79 * 45` und multipliziert die beiden Zahlen miteinander:

```
$ expr 79 \* 45
3555
```

Im zweite Beispiel ohne den Backslash expandiert die Shell den `*` zu einer Liste von Dateinamen — was `expr` verwirrt. Um das nachzuvollziehen, kann man die beiden Kommandos mit `echo` statt `expr` wiederholen.

```
$ expr 79 * 45
expr: syntax error
```

**Hinweis**: Gelegentlich werden normale Zeichen durch einen Backslash davor zu einem Metazeichen (z.B. „`\n`“).

### 5.2.2 Einfache Quotierungszeichen

Ein **einfaches Quotierungszeichen** (`'`) schaltet die Sonderbedeutung aller Zeichen bis zum nächsten einfachen Anführungszeichen ab. In der folgenden Kommandozeile ist daher der Text `s␣next?␣␣Mike` zwischen den beiden einfachen Anführungszeichen quotiert. Die Quotierungszeichen selbst werden von der Shell entfernt. Obwohl es sich um ein unsinniges Kommando handelt, stellt es ein gutes Beispiel für die Arbeitsweise der Quotierung dar:

```
$ echo Hey.        What's next?  Mike's #1 friend has $$.
Hey. Whats next?  Mikes
```

Die Leerzeichen außerhalb der Quotierungszeichen werden als **Argumenttrenner** betrachtet, mehrfache Leerzeichen betrachtet die Shell als ein Leerzeichen („Leerraum“). Leerzeichen innerhalb von Quotierungszeichen werden vollständig an `echo` übergeben, `echo` selbst gibt zwischen jedem seiner Argumente ein Leerzeichen aus. Das Fragezeichen (`?`) ist quotiert, es wird daher wörtlich an `echo` übergeben und nicht als Wildcard interpretiert.

Daher gibt `echo` sein erstes Argument `Hey.` und ein einzelnes Leerzeichen aus. Das zweite Argument von `echo` ist `Whats␣next?␣␣Mikes`. Es stellt ein einzelnes Argument dar, weil die einfachen Quotierungszeichen die Leerzeichen umgeben (bitte beachten, dass `echo` nach dem Fragezeichen zwei Leerzeichen ausgibt: `?␣␣`). Das nächste Argument `#1` fängt mit dem Kommentarzeichen `#` an. Dies hat zur Folge, dass die Shell den Rest der Zeichenkette ignoriert, er wird nicht an `echo` weitergegeben.

### 5.2.3 Doppelte Quotierungszeichen

**Doppelte Quotierungszeichen** (`"`) arbeiten fast so wie einfache. Der Unterschied liegt darin, dass die Zeichen `$` (Dollarzeichen), `` `...` `` (Backquote) und `\` (Backslash) ihre Sonderbedeutung beibehalten. Daher wird *Variablen- und Kommando-Ersetzung* innerhalb dieser Quotierungszeichen noch durchgeführt, bzw. kann auch an den Stellen durch Backslash ausgeschalten werden, wo das benötigt wird.

Hier nochmal das Beispiel von oben, dieses Mal sind allerdings doppelte Quotierungszeichen um die beiden einfachen Quotierungszeichen (beziehungsweise um die ganze Zeichenkette) gesetzt:

```
$ echo "Hey.        What's next?  Mike's #1 friend has $$."
Hey.        What's next?  Mike's #1 friend has 18437."
```

Das öffnende Quotierungszeichen wird erst am Ende der Zeichenkette beendet. Daher verlieren alle Leerzeichen zwischen den Quotierungszeichen ihre Sonderbedeutung — die Shell gibt die gesamte Zeichenkette als ein einzelnes Argument an `echo` weiter. Ebenso verlieren die einfachen Quotierungszeichen ihre Sonderbedeutung — da doppelte Anführungszeichen ihre Sonderbedeutung abschalten! Infolgedessen werden die einfachen Quotierungszeichen nicht wie im vorhergehenden Beispiel entfernt, sondern `echo` gibt sie aus.

Das Doppelkreuz `#` hat ebenfalls seine Sonderbedeutung als Kommentar-Beginn verloren, also wird auch der Rest der Zeichenkette an `echo` weitergegeben. Das Dollarzeichen (`$`) hat seine Sonderbedeutung als Variablen-Zugriff dagegen nicht verloren, d.h. `$$` wird zur *Prozessnummer* der Shell expandiert (`18437` in dieser Shell).

Was wäre passiert, wenn das Zeichen `$` innerhalb der einfachen Quotierungszeichen gestanden wäre? (Bitte daran denken, einfache Quotierungszeichen schalten die Bedeutung von `$` ab.) Würde die Shell dann immer noch `$$` zu seinem Wert expandieren? Klar: die einfachen Quotierungszeichen haben ihre Sonderbedeutung verloren, daher beeinflussen sie die Zeichen dazwischen nicht:

```
$ echo "What's next?  How many $$ did Mike's friend bring?"
What's next?  How many 18437 did Mike's friend bring?"
```

Wie kann man sowohl `$$` als auch die einfachen Quotierungszeichen wörtlich ausgeben? Am einfachsten mit Hilfe eines Backslash, der innerhalb von doppelten Quotierungszeichen noch funktioniert:

```
$ echo "What's next?  How many \$\$ did Mike's friend bring?"
What's next?  How many $$ did Mike's friend bring?"
```

Hier ein anderer Weg, das Problem zu lösen. Ein gründlicher Blick darauf zeigt eine Menge über die Quotierungsmechanismen der Shell:

```
$ echo "What's next?  How many "'$$'" did Mike's friend bring?"
What's next?  How many $$ did Mike's friend bring?"
```

Um dieses Beispiel zu verstehen, bitte daran denken, dass doppelte Anführungszeichen alle Zeichen bis zum nächsten doppelten Anführungszeichen quotieren. Das gleiche gilt für einfache Quotierungszeichen. Daher steht die Zeichenkette `What'snext?␣␣How␣many␣` (einschließlich des Leerzeichens am Ende) innerhalb doppelter Anführungszeichen. Das `$$` steht innerhalb einfacher Anführungszeichen, der Rest der Zeile steht wieder in doppelten Anführungszeichen. Beide Zeichenketten in doppelten Anführungszeichen enthalten ein einfaches Anführungszeichen, sie schalten die Sonderbedeutung dafür ab, daher wird es wörtlich ausgegeben. *Da zwischen den 3 quotierten Token kein Leerraum steht, werden sie zu einem Token zusammengefasst.*

### 5.2.4 Einfache Quotierungszeichen verschachteln

Einfache Quotierungszeichen können nicht innerhalb einfacher Quotierungszeichen verwendet werden. Ein einfaches Quotierungszeichen schaltet die Sonderbedeutung *aller* Zeichen bis zum nächsten einfachen Quotierungszeichen ab, d.h. in einem solchen Fall sind doppelte Anführungszeichen und Backslashes zu verwenden.

### 5.2.5 Quotierung mehrerer Zeilen

Sobald ein einfaches oder doppeltes Quotierungszeichen auftritt, wird alles folgende quotiert. Die Quotierung kann sich über viele Zeilen hinweg erstrecken (die C-Shell erlaubt das nicht!).

Zum Beispiel könnte man denken, dass im folgenden kurzen Skript das `$1` innerhalb von Anführungszeichen steht . . . , dem ist aber nicht so:

```
awk '/foo/ { print '$1' }'
```

Tatsächlich werden alle Argumente von Awk *außer* dem `$1` quotiert. Daher wird `$1` von der Shell expandiert und nicht von Awk.

Hier ein weiteres Beispiel: In einer Shell-Variablen wird eine mehrzeilige Zeichenkette abgelegt, ein typischer Fall für ein Shell-Skript. Eine Shell-Variable kann nur über ein einzelnes Argument gefüllt werden, alle Argumenttrenner (Leerzeichen, usw.) müssen daher quotiert werden. Innerhalb von doppelten Quotierungszeichen werden `$...`, `` `...` ``, `$(...)` und `!...` interpretiert (übrigens *bevor* die Variable mit dem Wert belegt wird). Das öffnende doppelte Anführungszeichen wird am Ende der ersten Zeile nicht abgeschlossen, die Bourne-Shell gibt daher **Fortsetzungs-Prompt** (`>␣`) aus (`PS2`), bis alle Quotierungszeichen abgeschlossen sind:

```
$ greeting="Hi, $USER.
> The date and time now
> are:  `date`."

$ echo "$greeting"
Hi, jerry.
The date and time now
```

```
are:  Tue Sep  1 13:48:12 EDT 1992.

$ echo $greeting
Hi, jerry. The date and time now are: Tue Sep 1 13:48:12 EDT 1992.
```

Das erste `echo` verwendet doppelte Anführungszeichen. Daher wird die Shell-Variable expandiert, aber die Shell betrachtet die Leerzeichen und Zeilenvorschübe in der Variablen nicht als Argumenttrenner (bitte die beiden Leerzeichen am Ende von `are:␣␣` beachten). Das zweite `echo` verwendet keine doppelten Anführungszeichen. Die Leerzeichen und Zeilenvorschübe werden als **Argumenttrenner** betrachtet, die Shell übergibt daher 14 Argumente an `echo`, das sie mit einzelnen Leerzeichen dazwischen ausgibt.

## 5.3 Backslash am Zeilenende

Werden Backslashes außerhalb von Anführungszeichen am Zeilenende (genau vor dem Zeilenvorschub) verwendet, dann *schützen* sie den Zeilenvorschub. Innerhalb von einfachen Anführungszeichen wird ein Backslash einfach kopiert. Hier ein Beispiel, die Prompts sind durchnumeriert (`1$`, `2$`, usw.):

```
# echo "a long     # Beispiel 1
> line or two"     # Beispiel 1
a long             # Ausgabe 1
line or two        # Ausgabe 1
                   #
# echo a long\     # Beispiel 2
> line             # Beispiel 2
a longline         # Ausgabe 2
                   #
# echo a long \    # Beispiel 3
> line             # Beispiel 3
a long line        # Ausgabe 3
                   #
# echo "a long\    # Beispiel 4
> line"            # Beispiel 4
a longline         # Ausgabe 4
                   #
# echo 'a long\    # Beispiel 5
> line'            # Beispiel 5
a long\            # Ausgabe 5
line               # Ausgabe 5
```

- Ein Beispiel analog **Beispiel 1** wurde bereits beschrieben: Der Zeilenvorschub steht in Anführungszeichen, daher ist er kein Argumenttrenner und `echo` gibt ihn zusammen mit dem (einzelnen zweizeiligen) Argument aus.
- In **Beispiel 2** teilt der Backslash vor dem Zeilenvorschub der Shell mit, den Zeilenvorschub zu entfernen, die Wörter `long` und `line` werden als ein Argument an `echo` weitergegeben.
- **Beispiel 3** entspricht normalerweise dem Gewünschten, wenn man lange Listen von Kommandozeilen-Argumenten eingibt: Es ist ein Leerzeichen (als Argumenttrenner) *vor* dem Backslash und dem Zeilenvorschub einzutippen.

- In **Beispiel 4** führt der Backslash vor dem Zeilenvorschub innerhalb der doppelten Anführungszeichen dazu, dass der Zeilenvorschub ignoriert wird (siehe Beispiel 1).
- Innerhalb einfacher Anführungszeichen, wie in **Beispiel 5**, hat ein Backslash keine Sonderbedeutung, er wird an `echo` übergeben.

## 5.4 Here-Dokument

Bisher wurden drei verschiedene Arten von Quotierung erläutert: Backslashes (`\`), einfache Anführungszeichen (`'`) und doppelte Anführungszeichen (`"`). Die Shell unterstützt noch eine weitere Art von Quotierung, das sogenannte **Here-Dokument**.

Ein Here-Dokument ist dann sinnvoll, wenn etwas von der Standard-Eingabe gelesen werden soll, ohne für den Eingabetext ein eigenes Dokument zu erzeugen. Stattdessen soll dieser Eingabetext direkt im Shellskript abgelegt (oder direkt auf der Kommandozeile eingegeben) werden.

Hierzu ist der `<<`-Operator gefolgt von einem (beliebigen) **Schlüsselwort** zu verwenden, das im Text nicht auf einer Zeile für sich vorkommen darf (üblicherweise wird aber `EOF` oder `END_OF_FILE` verwendet).

```
sort > file <<EOF
zygote
babel
moses
abacus
mulut
anton
EOF
```

Das Ergebnis bei der Ausführung des obigen Kommandos ist:

```
abacus
anton
babel
moses
mulut
zygote
```

Diese Form ist sehr nützlich, da in einem Here-Dokument Variablen- und Kommando-Ersetzung durchgeführt werden. Hier eine Möglichkeit, eine Datei über anonymes `ftp` aus einem Shellskript heraus zu übertragen:

```
#!/bin/sh
# Usage:
#     ftpfile machine file
# set -x
SOURCE="$1"
FILE="$2"
GETHOST="uname -n"
ftp -n "$SOURCE" <<-EOF
```

```
    ascii
    user anonymous $USER@`$GETHOST`
    get "$FILE" "/tmp/$FILE"
    quit
EOF
```

Wie zu sehen ist, werden in einem Here-Dokument Variablen- und Kommando-Ersetzungen durchgeführt. Ist dies nicht erwünscht, ist ein Backslash vor das Schlüsselwort zu schreiben:

```
cat > FILE <<\EOF
Text
...
EOF
```

Viele Shells kennen auch den Operator `<<-`. Der Bindestrich `-` am Ende weist die Shell an, alle `TAB`-Zeichen am Anfang jeder Textzeile zu ignorieren. Auf diese Weise kann der Text geeignet eingerückt werden, ohne dass die `TAB`s an die Standard-Eingabe des Kommandos weitergereicht werden. Beispiel (Tabulatoren, keine Leerzeichen am Zeilenanfang!):

```
cat > input.sql <<-EOF
    SELECT COUNT(*)
    FROM   test_table
        WHERE  flag = 1
        AND    country = 'CH'
        AND    date >= '1.1.1998'
    GO
EOF
```

Die Datei `input.sql` enthält anschließend keine Tabulatoren mehr am Zeilenanfang:

```
SELECT COUNT(*)
FROM   test_table
WHERE  flag = 1
AND    country = 'CH'
AND    date >= '1.1.1998'
GO
```

**Hinweis**: Soll keine Variablen- und Kommando-Substitution im Here-Dokument durchgeführt werden, dann einen `\` vor den Begrenzungstext schreiben oder ihn in `'...'` setzen. Mit `"..."` um den Begrenzungstext können auch Sonderzeichen (z.B. Leerzeichen) darin vorkommen.

```
cat > input.txt <<\EOF
$VAR bleibt stehen
$(date) bleibt stehen
EOF

cat > input.txt <<'EOF'
$VAR bleibt stehen
$(date) bleibt stehen
EOF

cat > input.txt <<"E O F"
$VAR bleibt stehen
$(date) bleibt stehen
E O F
```

# 6 Literatur


- Dietze, *Praxiskurs Unix-Shells*, O'Reilly.
- Wolf, Kania, *Shell-Programmierung — Das umfassende Handbuch*, Rheinwerk.
- Graiger, *Bash Programmierung — Einstieg und professioneller Einsatz*, Entwickler Press.
- Meißner, *Bash — Arbeiten und programmieren mit der Shell*, Open Source Press.
- Guckes, Plenz, *Zsh — die magische Shell*, Open Source Press.
- Robbins, Beebe, *Klassische Shell-Programmierung*, O'Reilly.
- Kochan, Wood, *UNIX Shell-Programmierung*, te-wi.
- Krienke, *UNIX Shell-Programmierung*, Hanser.
- Arthur & Burns, *UNIX Shell Programming, 3. Edition*, Wiley.
- Gulbins, Obermayer, *UNIX, 4. Auflage*, Springer.
- Gilly, *UNIX in a Nutshell, 2. Edition*, O'Reilly.
- Abrahams, Larson, *UNIX for the Impatient, 2. Edition*, Addison-Wesley.
- Peek, O'Reilly, Loukides, *UNIX Power Tools, 2. Edition*, O'Reilly.
- Bock, *Shellprogrammierung*, bhv.
- Ditchen, *Shell-Skript Programmierung*, mitp.
- Taylor, *Raffinierte Shell Scripts*, mitp.