

Shell-Einführung, Tipps und Tricks

Version 1.3 — 11.8.2007

© 2003–2007 T. Birnthaler, OSTC GmbH

Die Informationen in diesem Skript wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Der Autor übernimmt keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene fehlerhafte Angaben und deren Folgen.

Alle Rechte vorbehalten einschließlich Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Für Dokumente und Programme unter dem Copyright der OSTC GmbH gilt:

- Dürfen heruntergeladen und im privaten Bereich frei verwendet werden.
- Kommerzielle Nutzung bedarf der vorherigen Zustimmung durch die OSTC GmbH.
- Titelseite und Copyright-Hinweise darin dürfen nicht verändert werden.

Hinweise auf inhaltliche Fehler, Schreibfehler und unklare Formulierungen sowie Ergänzungen, Kommentare, Wünsche und Fragen können Sie gerne an den Autor richten:

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

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Shell-Versionen | 4 |
| 2 Eigenschaften der Shell | 4 |
| 2.1 Shell-Konfigurationsdateien | 6 |
| 2.2 Kommandosyntax und -aufruf in der Shell | 6 |
| 2.2.1 Kommando-Suche | 7 |
| 2.3 Dateinamen-Expansion | 8 |
| 2.4 Ein/Ausgabe-Umlenkung | 9 |
| 2.4.1 Here-Dokument | 11 |
| 2.5 Hintergrundprozesse | 11 |
| 2.6 Shell-Variable | 11 |
| 2.6.1 Umgebungs-Variable | 12 |
| 2.6.2 Hinweise | 12 |
| 2.6.3 Beispiele | 12 |
| 2.7 Quotierung | 13 |
| 2.7.1 Kommando-Substitution | 14 |
| 2.8 Exit-Code | 14 |
| 2.9 Aliase | 15 |
| 2.10 Funktionen | 15 |
| 2.11 History-Mechanismus | 15 |
| 2.12 Filename-Completion | 15 |
| 2.13 Shell-Skripte | 16 |
| 2.13.1 Parameterübergabe | 16 |
| 2.13.2 Spezial-Variable | 17 |
| 2.13.3 Das test-Kommando | 17 |
| 2.13.4 Kontrollstrukturen | 18 |
| 2.13.4.1 Sequenz | 18 |
| 2.13.4.2 Verzweigung | 18 |
| 2.13.4.3 Mehrfachverzweigung | 19 |
| 2.13.4.4 Schleifen | 19 |
| 2.13.4.5 Unterprogramme | 19 |
| 2.13.4.6 Vorzeitiger Abbruch | 19 |
| 2.13.4.7 Signale abfangen | 19 |
| 2.13.4.8 Sonstige Kontrollstrukturen | 19 |
| 2.13.5 Aufrufarten | 20 |
| 2.14 Eingabeprompt anpassen | 20 |
| 2.15 Die verschiedenen Shells | 20 |
| 2.16 Literatur | 22 |
| 3 Shell Setup-Dateien (Profile) | 22 |
| 3.1 Login-Shell versus Subshell | 22 |
| 3.2 Bourne-Shell | 23 |
| 3.3 C-Shell | 23 |
| 3.4 Korn-Shell | 23 |
| 3.5 Bash | 24 |

| | | |
|----------|---|-----------|
| 3.6 | Tcsh | 24 |
| 3.7 | Inhalt von Shell Setup-Dateien | 24 |
| 4 | Quotierung | 25 |
| 4.1 | Spezielle Zeichen | 25 |
| 4.2 | Wie arbeitet die Quotierung? | 26 |
| 4.2.1 | Backslash | 26 |
| 4.2.2 | Einfache Quotierungszeichen | 26 |
| 4.2.3 | Doppelte Quotierungszeichen | 27 |
| 4.2.4 | Einfache Quotierungszeichen verschachteln | 28 |
| 4.2.5 | Quotierung mehrerer Zeilen | 28 |
| 4.3 | Backslash am Zeilenende | 29 |
| 4.4 | Here-Dokumente | 30 |
| 5 | Kommandozeilenauswertung | 32 |
| 5.1 | Vorrang | 32 |
| 5.2 | Ein/Ausgabe-Umlenkung | 34 |
| 5.3 | Kommando-Ersetzung | 34 |

1 Shell-Versionen

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

| Typ | i | b | Src | Name, Besonderheiten |
|--------------------|----|----|-------------------|---|
| <code>sh</code> | -- | + | | Bourne-Shell („Ur“-Shell, Skript-orientiert) |
| <code>cs</code> h | + | - | | C-Shell (angelehnt an C, Interaktions-orientiert) |
| <code>tc</code> sh | ++ | + | <code>cs</code> h | Tenex-C-Shell |
| <code>k</code> sh | ++ | ++ | <code>sh</code> | Korn-Shell (UNIX System V-Standard) |
| <code>ba</code> sh | ++ | ++ | <code>sh</code> | Bo(u)rne-Again-Shell (LINUX-Standard) |
| <code>a</code> sh | +– | +– | <code>sh</code> | Simple Shell (umfaßt nur das Nötigste, klein) |
| <code>z</code> sh | ++ | ++ | <code>sh</code> | Ultimative Shell (umfaßt alle, groß, verwirrend) |

Folgende Shell-Typen gibt es:

- **Login-Shell:** Wird beim Einloggen standardmäßig gestartet, liest die Konfigurationsdateien
- **Sub-Shell:** Wird für jedes Unterkommando gestartet
- **Interaktive Shell:** Nimmt Kommandos entgegen (???)

2 Eigenschaften der Shell

Die Shell ist der Befehlsprozessor des UNIX-Systems. Sie zeigt einen Prompt (Standard: `$` oder `#`) an und wartet dann auf die Eingabe des Anwenders. Nachdem dieser Text eingegeben und „Return“ gedrückt hat, liest sie die Eingabe, interpretiert die darin angegebenen Sonderzeichen und führt dann das entsprechende Kommando aus. Folgende Mechanismen kennen die diversen Shells:

Shell-Skripten werden 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

Häufig findet man das Zeichen `;` (Semikolon) in Kommandozeilen vor, es trennt (wie der Zeilenvorschub) Kommandos: Man gibt ein Kommando ein und anschließend — anstatt „Return“ zu drücken — ein Semikolon und ein weiteres Kommando. Diese Verkettung von Kommandos ist insbesondere in Subshells, Aliasen, Funktionen und Kommando-Listen sinnvoll.

Folgende Eigenschaften von Shells sollte man kennen, um sie richtig benutzen zu können:

- Konfigurationsdateien (`/etc/profile`, `~/.profile`, `~/.bashrc` ...)
- Eingabe in Worte zerlegen (anhand „Whitespaces“) und interpretieren
- Befehlsarten unterscheiden (Builtin, Alias, Funktion, Programm/Skript)
- Kommandos-Suche durchführen (über `PATH`-Variable)
- Kommando ausführen (1. Wort in der Befehlszeile, nach `|` und nach `;`)
- Dateimuster-Expansion (`*` `?` `[...]` `[^/!...]` `~` `{...}`)
- Ein/Ausgabe-Umlenkung (`>` `>>` `2>` `2>>` `<` `1>&2` `2>&1`)
- Here-Dokument (`<<`)
- Pipes aufbauen (`|`)
- Hintergrundprozesse (`&`)
- Variablensubstitution (`$VAR`)
 - ▷ Shell-/Environment-Variable (`set`, `env`, `export`, `unset`)
 - ▷ Standard-Variable (`HOME`, `PS1`, `PS2`, `IFS`, `SHELL`, `TERM`, `PATH`, ...)
- Quotierung (`"..."` `'...'` `\`)
- Kommandosubstitution (``...``)
- Exit-Code (0=Kein Fehler, 1-255=Fehler)
- Aliase (`alias ll="..."`, `unalias`)
- Funktionen (`func() { ...; }`)
- History-Mechanismus (`history`, `r NR`, `r NAME`, ...)
- ▷ Command-Line-Edit
- Command/File/Var/User-Completion
- Shell-Skripte
 - ▷ Parameterübergabe (`$0`, `$1` `$2...`, `$#`, `$*`, `$@`)
 - ▷ Spezial-Variable (`$?`, `$!`, `$$`, `$-`)
 - ▷ Das `test`-Kommando

- ▷ Kontrollstrukturen
 - * Sequenz
 - * Verzweigung
 - * Mehrfachverzweigung
 - * Schleife (Anzahl Durchläufe, abweisend, nicht abweisend)
 - * Unterprogramm (Funktion)
 - * (Vorzeitiger) Abbruch (von Schleifen, Funktionen, Skripten)
 - * Signale abfangen
 - * Sonstige Kontrollstrukturen
- ▷ Verschiedene Aufrufarten
- Subshells

2.1 Shell-Konfigurationsdateien

Allgemeine Konfigurationsdateien:

- `/etc/profile`: *Zentrale* Konfigurationsdatei für alle Anwender, wird beim Start einer Login-Shell zuerst gelesen
- `~/profile`: *Persönliche* Konfigurationsdatei für den einzelnen Benutzer, steht in seinem Home-Verzeichnis und wird beim Start einer Login-Shell nach der zentralen Konfigurationsdatei gelesen

Spezielle Shell-Konfigurationsdateien (`rc` = Resource):

| Datei | Shell |
|-----------------------------|---------------|
| <code>~/bashrc</code> | bash |
| <code>~/bash_profile</code> | bash (Login) |
| <code>~/bash_login</code> | bash (Login) |
| <code>~/bash_logout</code> | bash (Logout) |
| <code>~/kshrc</code> | ksh |
| <code>~/cshrc</code> | csh |
| <code>~/login</code> | csh (Login) |
| <code>~/logout</code> | csh (Logout) |
| <code>~/tcshrc</code> | tcsh |

2.2 Kommandosyntax und -aufruf in der Shell

Die allgemeine Syntax von Kommandoaufrufen lautet:

```
CMD [OPTION...] [--] [ARG...]
```

Ein Kommando wird erst durch 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 der Kommandozeile 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 allen GNU-Programmen) **mehrbuchstabig**, jeder Optionsbuchstabe steht für ein englisches Wort, das seine Bedeutung beschreibt (Merkhilfe!). Zu einer Option können weitere **Parameter** nötig sein, die direkt dahinter anzugeben sind (und nicht mit `-` beginnen).

- Einbuchstabile Optionen ohne Argumente können einzeln (jede mit `-` davor) oder ohne Leerzeichen aneinandergehängt (nur ein `-` am Anfang notwendig) angegeben werden.
- Mehrbuchstabile Optionen sind durch `--` einzuleiten und können nicht aneinandergehängt werden.

Das 1. Wort, das nicht mit einem `-` beginnt, leitet die Liste der sonstigen **Kommandoargumente** ein. Alternativ wird die Liste der Optionen auch durch `--` beendet, alle danach aufgeführten Argumente (auch wenn sie als 1. Zeichen `-` enthalten) werden *nicht* als Optionen interpretiert. Typischerweise werden als Argumente die vom Kommando zu bearbeitenden Dateien angegeben. Das Argument `-` steht für die **Standard-Eingabe**. Beispiele:

```
ls *.c *.h
ls -l *.c *.h
ls -l -R -t
ls -lRt
ls --long --recursive --time
ls -- *.c
```

2.2.1 Kommando-Suche

Die Suche nach einem eingetippten Kommando erfolgt mehrstufig in folgender Reihenfolge. Ist auf einer Stufe ein passendes Kommando gefunden worden, so wird es ausgeführt und die Suche abgebrochen.

1. Built-in Kommando `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 rechts nach links in allen im **Suchpfad** `PATH` (z.B. `/bin:/usr/bin:.`) angegebenen Verzeichnissen (werden durch `:` getrennt) nach `CMD` suchen (*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`

Standardmäßig wird *nicht* im aktuellen Verzeichnis (Arbeitsverzeichnis) nach einem Kommando gesucht. Dazu ist in den Suchpfad das **aktuelle Verzeichnis** `.` (Punkt) aufzunehmen. Ein einzelner `:` am Anfang oder am Ende oder zwei direkt aufeinanderfolgende `::` in der Mitte des Suchpfades stehen ebenfalls für das aktuelle Verzeichnis. Die Root hat das aktuelle Verzeichnis üblicherweise (aus Sicherheitsgründen) nicht in ihrem Suchpfad. Die obige Kommandosuche kann vollständig umgangen werden, indem das Kommando mit einem **relativen** oder **absoluten Pfad** davor angegeben wird:

```
./CMD          CMD im aktuellem Verzeichnis aufrufen
/usr/bin/ls    ls aus Verzeichnis /usr/bin aufrufen
```

2.3 Dateinamen-Expansion

Um einem Kommando Dateinamen zu übergeben, kann entweder die Liste der Dateinamen vollständig angegeben werden, oder über ein **Suchmuster** (Pattern) werden dazu passende Dateinamen 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 |
|------------------------------|--|
| <code>?</code> | Ein beliebiges Zeichen |
| <code>*</code> | 0 oder mehr beliebige Zeichen (<i>nicht</i> Wiederholung eines Zeichens) |
| <code>\x</code> | Das Zeichen <i>x</i> <i>quoten</i> (<code>\\</code> steht für <code>\</code> selbst!) |
| <code>[abc], [a-z]</code> | Menge von Zeichen (Zeichenklasse , <code>[a-z]</code> = Zeichen von a bis z) |
| <code>[!abc], [!a-z]</code> | Negierte Menge von Zeichen (<i>sh</i> , <i>ksh</i> , <i>bash</i>) |
| <code>[^abc], [^a-z]</code> | Negierte Menge von Zeichen (<i>csh</i>) |
| <code>{abc, def, ...}</code> | Liste von Zeichenketten (<i>csh</i> , <i>ksh</i> , <i>bash</i>) |
| <code>~</code> | Home-Verzeichnis des aktuellen Users (<i>csh</i> , <i>ksh</i> , <i>bash</i>) |
| <code>~USER</code> | Home-Verzeichnis des Users <i>USER</i> (<i>csh</i> , <i>ksh</i> , <i>bash</i>) |

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. Es können auch Muster für ganze Dateipfade (Verzeichnis + Dateiname) angegeben werden. Beispiele (immer `ls -d` oder `echo` voransetzen):

```
a*      Dateinamen mit a am Anfang (auch a)
*a      Dateinamen mit a am Ende (auch a)
```

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

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 ausgegeben (Beispiel: `x11` in `/bin`).

2.4 Ein/Ausgabe-Umlenkung

Jedes UNIX-Kommando (genauer: jeder UNIX-Prozess) kennt 3 Standardkanäle für die Dateiein- und die Dateiausgabe:

| Kanal | Kürzel | Nummer | Default |
|------------------------|---------------------|--------|------------|
| Standard-Eingabe | <code>stdin</code> | 0 | Tastatur |
| Standard-Ausgabe | <code>stdout</code> | 1 | Bildschirm |
| Standard-Fehlerausgabe | <code>stderr</code> | 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- oder Pipe-Symbole beliebig mit anderen Dateien oder Kommandos verbinden, ohne dass das 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 nicht mit.

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

| Bedeutung | Syntax |
|---|--|
| <i>stdin</i> aus <i>file</i> holen | <code>cmd < file</code> |
| <i>stdout</i> in <i>file</i> speichern | <code>cmd > file</code> |
| <i>stderr</i> in <i>file</i> speichern | <code>cmd 2> file</code> |
| <i>stdout</i> ans Ende von <i>file</i> anhängen | <code>cmd >> file</code> |
| <i>stderr</i> ans Ende von <i>file</i> anhängen | <code>cmd 2>> file</code> |
| <i>stdout</i> von <i>prog1</i> in <i>cmd2</i> pipen | <code>cmd cmd2</code> |
| <i>stdout</i> und <i>stderr</i> in <i>file</i> speichern | <code>cmd > file 2>&1</code> |
| <i>stdout</i> und <i>stderr</i> ans Ende von <i>file</i> anhängen | <code>cmd >> file 2>&1</code> |
| <i>stdin</i> bis <i>text</i> von Tastatur holen | <code>cmd << text</code> |
| <i>stdout</i> und <i>stderr</i> von <i>prog1</i> in <i>cmd2</i> pipen | <code>cmd 2>&1 cmd2</code> |

Programme, die von Standardeingabe lesen und auf Standardeingabe schreiben, werden auch **Filter** genannt, weil sie wie ein Filter Daten lesen, manipulieren und wieder ausgeben. Mehrere (einfache) Filter hintereinandergesetzt ergeben ein kombiniertes (komplexes) 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 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 2> ERR                  Fehlermeldung Datei xyz unknown am Bildschirm
cat 2> ERR < xyz                 Fehlermeldung Datei xyz unknown in ERR
more FILE                         Datei FILE seitenweise anzeigen
cat < FILE | more                 Analog (2 Prozesse)
cat FILE | more                   Analog (2 Prozesse)

ls -l /bin > /tmp/liste           Dateiliste 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 einfach per Dateiumlenkung überschrieben werden können, falls sie schon existieren. Das Setzen dieser Option erfolgt mittels:

```
set -o noclobber
```

Das Zurücksetzen der Option erfolgt mittels:

```
set +o noclobber
```

2.4.1 Here-Dokument

Um Kommandoaufrufe und Daten in einer Datei 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 Texten `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 **Fortsetzungsprompt** `>` aus.

2.5 Hintergrundprozesse

2.6 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 Standardvariablen kann der Benutzer beliebige weitere Shell-Variablen anlegen (und auch wieder löschen). Die wichtigsten **Standardvariablen** sind:

| | |
|---------------------|--|
| <code>CDPATH</code> | Suchpfad für <code>cd</code> -Kommando |
| <code>HOME</code> | Standardverzeichnis für <code>cd</code> (Home-Verzeichnis) |
| <code>IFS</code> | Worttrenner („internal field separator“) |
| <code>PATH</code> | Suchpfad für Kommandoaufruf |
| <code>PS1</code> | Shell-Prompt (<code>\$_</code>) |
| <code>PS2</code> | Fortsetzungsprompt (<code>>_</code>) |
| <code>SHELL</code> | Name der aktuellen Shell |
| <code>TZ</code> | Zeitzone („time zone“) |
| <code>TERM</code> | Terminalname |

Die Kommandos zum Setzen, Anzeigen, Verwenden und Löschen von Shell-Variablen lauten:

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

2.6.1 Umgebungs-Variable

Ein spezieller Typ von Shell-Variablen sind die sogenannten **Umgebungs-Variablen**, sie sind eine Teilmenge der Shell-Variablen. Jeder Prozeß besitzt Umgebungsvariablen (nicht nur die Shell), sie werden vom Elternprozeß an Kindprozesse **vererbt** (Shell-Variablen nicht). Die Kindprozesse können die vererbte Liste beliebig ändern und erweitern und an ihre Kindprozesse weitervererben. „Zurückvererben“ können Kindprozesse ihren Umgebungsbereich an Elternprozesse nicht. Die Kommandos zum Erzeugen, Setzen und Auflisten 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 |

2.6.2 Hinweise

- Shell-Variablen werden auch als **lokale** Variablen bezeichnet.
- Umgebungs-Variablen werden auch als **globale** Variablen bezeichnet.
- 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**.

2.6.3 Beispiele

Die Variable PATH erweitern/verkürzen:

| | |
|----------------------|---|
| echo \$PATH | Inhalt der Variable PATH anzeigen |
| 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 $PATH |
sed "s#^/usr/bin:##" |
sed "s#:/usr/bin:##" |
sed "s#:/usr/bin$##" `
echo $PATH

```

Verzeichnis /usr/bin aus Variable PATH entfernen
am Anfang
in der Mitte
am Ende
→ /sbin:/bin:.

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

```

1 # Übergebene Werte ausgeben
2 echo "SHVAR=$SHVAR"
3 echo "ENVVAR=$ENVVAR"
4
5 # Werte neu belegen
6 SHVAR="wert1"
7 ENVVAR="wert2"
8
9 # Neu belegte Werte ausgeben
10 echo "SHVAR=$SHVAR"
11 echo "ENVVAR=$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

```

2.7 Quotierung

Die Shell kennt eine Reihe von Sonderzeichen, die nicht für sich selber stehen, sondern die von ihr interpretiert werden und bestimmte Funktionen auslösen:

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

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

Folgende Tabelle gibt eine Übersicht über die von den Quotierungszeichen jeweils geschützten (*) bzw. nicht geschützten (-) Sonderzeichen (+ = Ende, v = verbose):

| | \ | \$VAR | *?[] | \ . . . \ | " | ' | CR | ! | Whitespace | Rest |
|-----------|---|-------|-------|-----------|---|---|----|---|------------|------|
| " . . . " | - | - | * | - | + | * | V | - | * | * |
| ' . . . ' | * | * | * | * | * | + | V | - | * | * |
| _ | * | * | * | * | * | * | * | - | * | * |

Beispiele:

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

2.7.1 Kommando-Substitution

Neben dem Quotierungszeichen ' (quote) wird auch das Zeichen ` (backquote, backtick) verwendet, allerdings in einer ganz anderen Bedeutung (*Verwechslungsgefahr!*). Das innerhalb von ` . . . ` stehende Kommando wird ausgeführt, und sein *Ergebnis* wird an dieser Stelle eingesetzt. Erst danach wird das eigentliche Kommando ausgeführt. Mit dem folgenden Kommando können zum Beispiel alle C-Dateien, die den Text `error` enthalten, herausgesucht und der Editor für sie aufgerufen werden:

```
vi `grep error *.c`
```

An alle eingeloggten Benutzer die Datei `brief.txt` als Mail schicken:

```
mail `who | cut -f1 | sort | uniq` < brief.txt
```

Alle `bash`-Prozesse beenden:

```
kill -9 `ps -e | grep bash | cut -d" " -f2/3` (besser sed ???)
```

2.8 Exit-Code

Jedes Kommando gibt einen **Exit-Code** (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 `man`-Pages nachgelesen werden. Der Exit-Code von

Kommandos kann in Shell-Kontrollstrukturen abgefragt und für die Verzweigungen des Programmflusses benutzt werden. Er steht auch in der Variablen `$?` zur Verfügung und kann darüber ausgegeben werden. Beispiel:

```
grep TEXT FILE      0 falls TEXT in FILE gefunden
                   1 falls TEXT in FILE nicht gefunden
                   2 falls FILE nicht vorhanden oder TEXT fehlerhaft
echo $?            Exit-Code ausgeben (nur 1x möglich, da echo auch ein Kommando ist)
```

2.9 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!)
                        → wird zu /bin/ls -lF *.c *.h expandiert
ls *.c *.h
alias                  Alle Aliase anzeigen
alias ls              Alias ls anzeigen
unalias ls            Alias ls löschen
```

2.10 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-Code** 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)

```
ls() {                Funktion ls definieren (Rekursion vermeiden!)
    /bin/ls -lF
}
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
```

2.11 History-Mechanismus

2.12 Filename-Completion

bash: TAB + TAB TAB nach CMD FILE \$VAR ~USER

ksh: ESC *, ESC \

2.13 Shell-Skripte

2.13.1 Parameterübergabe

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>
```

2.13.2 Spezial-Variable

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 |
|----------|--|
| \$? | Exit-Code 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-Code      \ $? = <$?>"
echo "Aktuelle PID   \$\$ = <$$>"
sleep 100
echo "PID Hintergrund \$! = <$!>"
echo "Shell-Optionen \$- = <$->"
```

Der folgende Aufruf dieses Skriptes mit den angegebenen Argumenten

```
args.sh
```

ergibt folgende Ausgabe:

```
Exit-Code      $? = <2>
Aktuelle PID   $$ = <2371> # kann andere Prozeßnummer sein!
PID Hintergrund $! = <2372> # kann andere Prozeßnummer sein!
Shell-Optionen $- = <Bh>
```

2.13.3 Das test-Kommando

Das Kommando `test` wird in Shell-Anweisungen häufig verwendet, um eine Bedingung zu überprüfen und entsprechend zu verzweigen. Seine Aufrufsyntax lautet:

```
if test TYPE ... oder
if [ TYPE ] ...
```

Die verschiedenen Test-Typen `TYPE` lauten:

| | |
|--------------------|---|
| -n "TEXT" | TEXT ist nicht leer [nonzero] |
| -z "TEXT" | TEXT ist leer [zero] |
| "TEXT1" = "TEXT2" | TEXT1 und TEXT2 sind gleich |
| "TEXT1" != "TEXT2" | TEXT1 und TEXT2 sind verschieden |
| -d FILE | Datei FILE ist Verzeichnis [directory] |
| -e FILE | Datei FILE existiert [exists] |
| -f FILE | Datei FILE ist normale Datei [file] ??? |
| -r FILE | Datei FILE ist lesbar [readable] |
| -s FILE | Datei FILE nicht leer [size] |
| -w FILE | Datei FILE schreibbar [writable] |
| -x FILE | Datei FILE ausführbar [executable] |
| NUM1 -eq NUM2 | Zahl NUM1 gleich NUM2 [equal] |
| NUM1 -ne NUM2 | Zahl NUM1 nicht gleich NUM2 [not equal] |
| NUM1 -le NUM2 | Zahl NUM1 kleiner gleich NUM2 [less equal] |
| NUM1 -lt NUM2 | Zahl NUM1 kleiner NUM2 [less than] |
| NUM1 -ge NUM2 | Zahl NUM1 größer gleich NUM2 [greater equal] |
| NUM1 -gt NUM2 | Zahl NUM1 größer NUM2 [greater than] |
| -b FILE | Datei FILE ist blockorientiert [block device] |
| -c FILE | Datei FILE ist zeichenorientiert [character device] |
| -L FILE | Datei FILE ist ein symbolischer Link [link] |
| -p FILE | Datei FILE ist eine Named Pipe [pipe] |
| -S FILE | Datei FILE ist ein Socket [socket] |
| -t FILE | Datei FILE ist ein Terminal [terminal] |
| -g FILE | Datei FILE hat group id bit gesetzt |
| -k FILE | Datei FILE hat sticky bit gesetzt |
| -u FILE | Datei FILE hat user id bit gesetzt |
| -O FILE | Datei FILE gehört effektiver user id [Owner] |
| -G FILE | Datei FILE gehört effektiver group id [Group] |
| FILE1 -nt FILE2 | Datei FILE1 neuer als FILE2 [newer than] |
| FILE1 -ot FILE2 | Datei FILE1 älter als FILE2 [older than] |
| FILE1 -ef FILE2 | Datei FILE1 und FILE2 haben identische device/inode Nummer [equal file] |
| ! EXPR | Negation von EXPR [not] |
| EXPR1 -a EXPR2 | EXPR1 und EXPR2 [and] |
| EXPR1 -o EXPR2 | EXPR1 oder EXPR2 [or] |
| \(... \) | Klammerung (<i>Quotierung notwendig!</i>) |

2.13.4 Kontrollstrukturen

2.13.4.1 Sequenz

```

;
&&          [ -e FILE ] && rm FILE
||          [ -n "$VAR" ] || VAR = Default
(...)
{...}

```

2.13.4.2 Verzweigung

```

if CMD1
then
...

```

```

{elif CMD2
  then
    ...}
[else
  ...]
fi

```

2.13.4.3 Mehrfachverzweigung

```

case TEXT in
  MUSTER1) ... ;;
  MUSTER2) ... ;;
  ...
  MUSTERn) ... ;;
esac

```

2.13.4.4 Schleifen

| | |
|--------------------------------------|---|
| <pre> for VAR do ... done </pre> | <pre> for VAR in WORT1 WORT2 ... WORTn do ... done </pre> |
| <pre> while CMD do ... done </pre> | <pre> until CMD do ... done </pre> |

2.13.4.5 Unterprogramme

2.13.4.6 Vorzeitiger Abbruch

| | |
|---|--|
| <pre> break [N] continue [N] exit [EXITCODE] return [EXITCODE] </pre> | <pre> Schleife for, while, until abbrechen Schleife for, while, until wiederholen EXITCODE aus 0-255 möglich, Std: 0 In Funktionsdefinition </pre> |
|---|--|

2.13.4.7 Signale abfangen

```

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

```

2.13.4.8 Sonstige Kontrollstrukturen

```

.
:
true/false

```

2.13.5 Aufrufarten

| | PATH notwendig | x-Bit notwendig | Sub-Shell gestartet | Rückkehr zur Login-Shell |
|-------------|-------------------|--------------------|------------------------|-----------------------------|
| cmd.sh | ja | ja | ja | ja |
| sh cmd.sh | nein | nein | ja | ja |
| ./cmd.sh | nein | ja | ja | ja |
| . cmd.sh | ja | nein | nein | ja |
| ./cmd.sh | nein | nein | nein | ja |
| exec cmd.sh | nein | nein | nein | nein |

2.14 Eingabeprompt anpassen

Die Umgebungsvariable `PS1` legt das Aussehen des Eingabeprompts 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 Prompt ein:

| | |
|------------------|--|
| <code>\d</code> | Datum im Format <code>Sun Dec 24 [date]</code> |
| <code>\h</code> | Rechner-Name <code>[host]</code> |
| <code>\n</code> | Zeilenvorschub <code>[newline]</code> |
| <code>\w</code> | Arbeitsverzeichnis <code>[working directory]</code> |
| <code>\W</code> | Arbeitsverzeichnis (letzter Teil) <code>[working directory]</code> |
| <code>\s</code> | Shell-Name <code>[shell]</code> |
| <code>\t</code> | Zeit im Format <code>hh:mm:ss [time]</code> |
| <code>\u</code> | Benutzer-Name <code>[user]</code> |
| <code>\\$</code> | <code>\$</code> -Zeichen für normalen Benutzer, <code>#</code> -Zeichen für die Root |
| <code>\#</code> | Nummer des aktuellen Kommandos |
| <code>\!</code> | History-Nummer des aktuellen Kommandos |

Der Standardwert von `PS1` beträgt für die Root

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

und für normale Benutzer

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

Beachten Sie bitte das Leerzeichen am Ende, damit die Kommandoeingabe etwas abgesetzt vom Prompt beginnt. Der Prompt sollte auch nicht zu lang werden, da er sonst unübersichtlich ist und zu wenig Platz für die Eingabe der Kommandozeile übrig läßt.

Die Prompt-Variablen `PS2` legt das Bereitschaftszeichen fest, das immer dann ausgegeben wird, wenn die Shell ein Kommando als auf einer Zeile noch nicht abgeschlossen betrachtet. Sie hat den Defaultwert `>_`. Häufig ist das Kommando mit `Ctrl-D` oder `.` abzuschließen.

2.15 Die verschiedenen Shells

Bei den meisten 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!

Es sind eine ganze Reihe verschiedener Shells verfügbar:

- sh Die **Bourne-Shell** (nach ihrem Ersteller *Steve Bourne* benannt). Sie ist die älteste UNIX-Shell und ist auf den meisten 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 *Aliases* (Einführung eigener Kommandos).
Die Bourne-Shell ist aber sehr gut geeignet für die (portable) *Shell-Programmierung* oder die Erstellung von Kommando-Dateien, da sie die „**Lingua Franca**“ der meisten Shells darstellt. Sie ist weniger gut geeignet für die interaktive Benutzung, jede der im folgenden beschriebenen Shells ist dafür besser geeignet.
- csch Die **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 Die **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 Dateinamen/Kommandos 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 Die **Bourne-again Shell**, entwickelt von der *Free Software Foundation*, ist der Korn-Shell sehr ähnlich. Sie hat viele Eigenschaften der C-Shell plus *Command-Line-Editierung* und ein eingebautes Hilfe-Kommando. Die Bourne-Again-Shell ist die Standard-Shell unter LINUX.
- tcsh 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.

Da `bash` und `ksh` Skripten lesen können, die für die originale Bourne-Shell geschrieben wurden, wird üblicherweise ausschließlich der von der Bourne-Shell unterstützte Befehlsumfang zur Shell-Programmierung verwendet.

2.16 Literatur

- 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 & Associates.
- Abrahams, Larson, *UNIX for the Impatient, 2. Edition*, Addison-Wesley.
- Peek, O'Reilly, Loukides, *UNIX Power Tools, 2. Edition*, O'Reilly & Associates.

3 Shell Setup-Dateien (Profile)

3.1 Login-Shell versus Subshell

Eine Shell kann in zwei verschiedenen Modi laufen: als **Login-Shell** und als **Subshell** (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 **Subshell** ist niemals eine Login-Shell. In allen Shells führen die **Klammer-Operatoren** dazu, dass eine weitere Instanz der aktuellen Shell gestartet wird (Klammern werden **Subshell-Operatoren** genannt). Eine Subshell 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 Subshell gibt keinen 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-Kommandos u.ä. einrichtet. Andere Shells auf dem gleichen Terminal sollen Nicht-Login-Shells sein — 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.

3.2 Bourne-Shell

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 Home-Verzeichnis steht.

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

3.3 C-Shell

Anwender der C-Shell haben drei Setup-Dateien zur Verfügung:

- Die 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 Umgebungsvariablen an Subshells weitergeleitet, daher gehören sie in der Datei `.cshrc` gesetzt.
- Die Datei `.login` wird bei jedem Start einer Login-Shell gelesen. Folgendes sollte darin gesetzt werden:
 - ▷ Umgebungsvariablen (die UNIX automatisch an Subshells weitergibt).
 - ▷ Initialisierungs-Kommandos wie `tset` und `stty`.
 - ▷ Kommandos, die bei jedem Login ausgeführt werden sollen — ob Mails oder News angekommen sind, `fortune` ausführen, den Tageskalender ausgeben, usw.
- Die C-Shell liest `.logout`, wenn eine Login-Shell beendet wird.

Bitte beachten, dass beim Login `.cshrc` vor `.login` gelesen wird!

3.4 Korn-Shell

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

Login
auch
???

3.5 Bash

`bash` ist eine Kreuzung zwischen Bourne- und C-Shell. Eine Login-`bash` liest erst die Datei `/etc/profile` und dann eine der Dateien `.bash_profile`, `.bash_login` oder `.profile` ein (die Suche erfolgt in dieser Reihenfolge). Eine `bash`-Subshell — nicht aber eine Login-Shell — liest die Datei `.bashrc` im Home-Verzeichnis ein. Sie liest `.bash_logout`, wenn eine `bash`-Login-Shell beendet wird.

3.6 Tcsh

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

3.7 Inhalt von Shell Setup-Dateien

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

- Setzen/Erweitern des **Suchpfades**.
- Setzen des **Terminaltyps** und verschiedener Terminalparameter.
- Setzen von **Umgebungsvariablen**, 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 Setup-Datei erledigen. Viele Leute mögen es auch, einen lustigen oder lehrreichen „Fortune“ auszugeben. 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 sicher 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 Setup-Datei geändert wird, sollte immer eine **Sicherheitskopie** von ihr erstellt werden, damit (1) bei Problemen wieder auf das Original zurückgeschaltet 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 Setup-Dateien richtig einzusetzen. Hier einige Dinge, die man ausprobieren kann:

- Einen angepaßten Prompt erstellen.

- Gemeinsame Setup-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 die Login-Datei ewig und drei Tage dauert.

4 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 Zeichen für die Shell ab*. Man sagt auch, die Zeichen werden vor der Shell „geschützt“ oder „zitiert“. Es gibt drei Quotierungszeichen:

- Das **einfache Anführungszeichen** `'`
- Das **doppelte Anführungszeichen** `"`
- Den **Backslash** `\`

Das Zeichen `\` ist *kein* Quotierungszeichen — es dient zur **Kommando-Ersetzung** und wird in Abschnitt 5.3 behandelt.

4.1 Spezielle Zeichen

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

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

In der C-Shell haben zusätzlich folgende Zeichen eine Sonderbedeutung:

```
~ { } !
```

Ein Slash (`/`) hat für UNIX selbst eine Sonderbedeutung, nicht aber für die Shell — 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.

4.2 Wie arbeitet die Quotierung?

Folgende Tabelle faßt die Quotierungs-Regeln zusammen:

| Quotierungszeichen | Erklärung |
|--------------------|---|
| '...' | Alle Sonderzeichen in ... abschalten. |
| "..." | Alle Sonderzeichen bis auf \$, ` und \ in ... abschalten. |
| \. | Die Sonderbedeutung von . 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.

Die nächsten Abschnitte erläutern diese Regeln mittels Beispielen, sie können durch Eintippen ausprobiert werden. Bitte nicht die C-Shell verwenden (sie verhält sich anders), sondern zuerst `sh` eintippen, um eine Bourne-Shell zu starten. Zum Beenden dieser Subshell `Ctrl-D` oder `exit` eintippen.

4.2.1 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

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

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.

4.2.2 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.

4.2.3 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 durchgeführt, bzw. kann auch an den Stellen durch Backslash ausgeschaltet 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's next? _ _ 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.*

4.2.4 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.

4.2.5 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 **Secondary Prompts** (>) aus, 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.

4.3 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 durchnummeriert (1\$, 2\$, usw.):

```
1$ echo "a long
> line or two"
a long
line or two

2$ echo a long\
> line
a longline

3$ echo a long \
> line
a long line

4$ echo "a long\
```

```
> line"
a longline

5$ echo 'a long\
> line'
a long\
line
```

- 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.

4.4 Here-Dokumente

Bisher wurden drei verschiedene Arten von Quotierung erläutert: Backslashes (`\`), einfache Anführungszeichen (`'`) und doppelte Anführungszeichen (`"`). Die Shell unterstützt noch eine weitere Art von Quotierung, die sogenannten **Here-Dokumente**. Ein Here-Dokument ist dann sinnvoll, wenn etwas von der Standardeingabe 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 immer `EOF` oder `END_OF_FILE` verwendet).

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

Das Ergebnis bei der Ausführung des obigen Kommandos ist:

```

abacus
anton
babel
moses
mulut
zygote

```

Diese Form ist sehr nützlich, da in Here-Dokumenten Variablen- und Kommando-Ersetzung ausgewertet 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 im Here-Dokument Variablen- und Kommando-Ersetzungen durchgeführt. Ist dies nicht erwünscht, ist ein Backslash vor das Schlüsselwort zu schreiben:

```

cat > FILE <<\END_OF_FILE
Text
...
END_OF_FILE

```

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 TABS an die Standardeingabe 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

```

5 Kommandozeilenauswertung

5.1 Vorrang

Aufgrund der Vielzahl von Ersetzungsmechanismen 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. Zerlegen in **Token** (einschließlich Sonderzeichen)
3. History-Liste updaten [csh]
4. Einfache und doppelte Anführungszeichen interpretieren (' ")
5. Alias-Ersetzung [csh]
6. Ein/Ausgabe-Umlenkung (<, >, 2>, >>, <<, |, ...)
7. Variablen-Ersetzung (\$VAR)
8. Kommando-Ersetzung (`cmd ... `)
9. Dateinamen expandieren (? , * , [], [!])

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

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, muss ein Backslash von dem ! stehen.

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 letzten Befehls**“. Angenommen der letzte 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 **Tokens** `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 Tokens 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 übergeben¹ und das Kommando entscheidet, wie sie zu interpretieren sind. Die Anführungszeichen verhindern auch, dass die Shell `Mar_7` in zwei Tokens zerlegt oder die beiden Leerzeichen ignoriert² — 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.
5. Die Shell erkennt die beiden Pipe-Symbole `|` und führt alles Notwendige für den Pipeline-Mechanismus durch.
6. Die Shell erkennt die Umgebungsvariable `$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` verbindet.

¹Die 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

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

5.2 Ein/Ausgabe-Umlenkung

Die `sh` bzw. `csch` kennen folgende Syntax für Datei-Umlenkungen:

| Umlenkung | sh | csch |
|---|--|-------------------------------------|
| <i>stdout</i> in <i>file</i> speichern | <code>cmd > file</code> | <code>cmd > file</code> |
| <i>stderr</i> in <i>file</i> speichern | <code>cmd 2> file</code> | — |
| <i>stdout</i> und <i>stderr</i> in <i>file</i> speichern | <code>cmd > file 2>&1</code> | <code>cmd >& file</code> |
| <i>stdin</i> aus <i>file</i> holen | <code>cmd < file</code> | <code>cmd < file</code> |
| <i>stdout</i> ans Ende von <i>file</i> anhängen | <code>cmd >> file</code> | <code>cmd >> file</code> |
| <i>stderr</i> ans Ende von <i>file</i> anhängen | <code>cmd 2>> file</code> | — |
| <i>stdout</i> und <i>stderr</i> ans Ende von <i>file</i> anhängen | <code>cmd >> file 2>&1</code> | <code>cmd >>& file</code> |
| <i>stdin</i> bis <i>text</i> von Tastatur holen | <code>cmd << text</code> | <code>cmd << text</code> |
| <i>stdout</i> von <i>prog1</i> in <i>cmd2</i> pipen | <code>cmd cmd2</code> | <code>cmd cmd2</code> |
| <i>stdout</i> und <i>stderr</i> von <i>prog1</i> in <i>cmd2</i> pipen | <code>cmd 2>&1 cmd2</code> | <code>cmd & cmd2</code> |

5.3 Kommando-Ersetzung

Ein Paar Backquotes (`` . . . ``) führt eine **Kommando-Ersetzung** durch. Dies ist wirklich sehr nützlich, denn damit läßt sich die Standard-Ausgabe eines Kommandos als Argument für ein anderes Kommando einsetzen.

Hier ein Beispiel: Angenommen man will alle C-Dateien im aktuellen Verzeichnis editieren, die das Wort `error` enthalten. Dies läßt sich mit folgender Eingabe erreichen:

```
$ vi `grep -l error *.c`
3 files to edit
"bar.c" 254 lines, 28338 characters
...
$
```

Aber warum funktioniert das? Wie ist der obige Aufruf zusammengestellt worden? Was hätte man ohne obige Technik gemacht: Man hätte `grep` verwendet, um herauszufinden, welche C-Dateien das Wort `error` enthalten. Dann hätte man den `vi` verwendet, um diese Liste zu editieren:

```
$ grep error *.c
bar.c: error("input to long");
bar.c: error("input to long");
baz.c: error("data formatted incorrectly");
foo.c: error("can't divide by zero");
foo.c: error("insufficient memory");

$ vi bar.c baz.c foo.c
```

Dies kann durch Kommando-Ersetzung in einem einzigen Kommando zusammengefaßt werden. Zunächst muss das Kommando `grep` so modifiziert werden, dass es anstelle der Dateinamen und der Texte nur die Liste der Dateinamen ausgibt (`l=list`):

```
$ grep -l error *.c
bar.c
baz.c
foo.c
```

Die Option `-l` listet jeden Dateinamen mit mindestens einer passenden Zeile *nur einmal* auf, auch wenn viele Zeilen in der Datei passen. Jetzt sollen diese Dateien editiert werden, daher wird das Kommando `grep` in Backquotes gesetzt und als Argument für den `vi` verwendet:

```
$ vi `grep -l error *.c`
3 files to edit
"bar.c" 254 lines, 28338 characters
...
$
```

Die „**vertikale**“ Ausgabe von `grep` und die „**horizontale**“ Art, in der man normalerweise Argumente auf der Kommandozeile eingibt, machen für die Shell keinen Unterschied. Die Shell verarbeitet beides problemlos, innerhalb von Backquotes sind sowohl Zeilenvorschübe als auch Leerzeichen Argumenttrenner.

Die Liste, die in einer Kommando-Ersetzung verwendet wird, muss keine Liste von Dateinamen sein. Wie kann man z.B. eine Mailnachricht an alle derzeit auf dem System eingeloggten Benutzer schicken? Dazu ist eine Kommandozeile wie folgt abzusetzen:

```
% mail joe lisa franka mondo bozo harpo ...
```

Um dies zu erreichen, muss man sich überlegen, welche UNIX-Kommandos für diese gewünschte Ausgabe gebraucht werden. Um eine Liste der eingeloggten Benutzer zu erhalten, kann das Kommando `who` verwendet werden. Es gibt auch die Login-Zeit und andere Informationen aus — aber diese Daten können mit einem Kommando wie `cut` entfernt werden:

```
% who | cut -c1-8
joe
lisa
franka
lisa
joe
mondo
joe
...
```

Einige Benutzer sind mehr als einmal eingeloggt. Um eine eindeutige Liste zu erhalten, wird `sort | uniq` (*unique*) verwendet. Das ganze Kommando zum Erzeugen der Namensliste wird in Backquotes gesetzt und als Argument an `mail` übergeben:

```
% mail `who | cut -c1-8 | sort | uniq`
```

Wenn man sich nicht ganz sicher ist, ob das funktioniert, kann anstelle des gewünschten Kommandos erst einmal `echo` eingesetzt werden:

```
% echo `who | cut -c1-8 | sort | uniq`
bozo franka harpo joe lisa mondo
```

Verwendet man UNIX eine Zeitlang, so wird man feststellen, dass dies eine seiner nützlichsten Eigenschaften ist. Man wird viele Situationen antreffen, in denen ein Kommando zum Erzeugen einer Liste von Werten verwendet und diese Liste dann als Argument für ein anderes Kommando eingesetzt werden kann. Es ist sogar möglich, Backquotes zu verschachteln.