Unix-Shell am Beispiel *bash*

Michael Jäger

3. September 2018

(Version 1.1.0 vom 31.8.2018)

Inhaltsverzeichnis

1 Die *bash* - „Bourne again shell“

POSIX ist ein IEEE-Standard für UNIX-konforme Betriebssystemschnittstellen. Die *bash*-Shell ist ein POSIX-konformer Kommandointerpretierer, der Kommandos einliest und ausführt. Durch programmiersprachliche Konzepte (Variablen, bedingte Ausführung und Wiederholungsanweisungen usw.) ist eine Shell das richtige Werkzeug zur Automatisierung wiederholt benötigter Bedienprozesse.

Gegenüber einer älteren „Bourne-Shell“ (*sh*) bietet BASH mehr Komfort bei der interaktiven Nutzung, z.B. eine Kommandohistorie, aus der man schon einmal eingegebene Kommandos wieder abrufen kann.

Die Unterschiede zur verbreiteten Korn-Shell (*ksh*) und zu anderen POSIX-Shells sind gering, während C-Shells (*csh*, *tcsh*) eine völlig andere Syntax benutzen.

Die Beschreibung geht nicht auf die UNIX-Kommandos ein, die als selbständige Programme implementiert sind, sondern beschreibt „nur“ den Kommando-Interpretierer, mit dem man eben diese Programme aufrufen kann.

Die Beschreibung geht ebenfalls nicht auf einige *bash*-spezifischen Sprachkonstrukte ein, die nicht dem POSIX-Standard entsprechen. Dazu gehören z.B. Funktion-lokale Variablen, Array-Variablen, Assoziativ-Listen, *bash*-Ausdrücke der Form `[[ ... ]]` usw.

2 Verwendung mit graphischen Benutzeroberflächen

Als textorientiertes Programm benötigt die Shell eine Terminalemulation zur Ausführung unter X-Windows. Das Emulationsprogramm stellt ein Fenster für Textausgaben zur Verfügung und verwaltet die Tastatureingaben und Textausgaben für die Shell. Beispiele für solche Terminalemulationen sind *xterm* oder das KDE-Programm *konsole*.

3 Die Kommandozeile

Die Shell dient zum Aufruf beliebiger Programme. Der Benutzer gibt im einfachsten Fall den Programmnamen an, d.h. den Dateinamen der Programmdatei, und bei Bedarf, durch Leerzeichen abgetrennt, eine Reihe von Argumenten für das Programm.

Beispiel:

```
/usr/bin/ls -al /usr /usr/bin
```

Das Programm „/usr/bin/ls“ wird aufgerufen, dem Programm werden beim Aufruf der Programmname selbst und die hinter diesem Dateipfad stehenden drei Zeichenketten als Argumente übergeben. Man muss nicht immer den vollständigen Pfad der Programmdatei angeben. Im Beispiel genügt „ls“ statt „/usr/bin/ls“, wenn die Umgebungsvariable PATH, der „Kommando-Suchpfad“, einen geigneten Wert hat. Siehe dazu **??**, S. **??**.

Ein Programm-Aufruf wird in der Regel als fork-exec-Aufruffolge durchgeführt:

- *fork* erzeugt einen Sub-Prozess als „Clone“ der Shell,
- *exec* führt im Kontext des aktuellen Prozesses ein Programm aus.

In der Kommandozeile gibt man den Programmnamen und - durch Leerzeichen getrennt - die an das Programm zu übergebenden Parameter an, z.B.

```
ls -al /etc
```

Bei der Eingabe langer Dateinamen, Kommandos, Variablennamen usw. ist die TAB-Taste recht nützlich, denn sie vervollständigt die Eingabe, soweit sie eindeutig ist. Nicht-Eindeutigkeit wird durch ein akustisches Signal

angezeigt, falls TAB dann noch einmal betätigt wird, zeigt *bash* alle möglichen Vervollständigungen an. (*bash* hat genau betrachtet 5 verschiedene Vervollständigungs-Funktionen vgl. Manual)

Falls das letzte Zeichen ein `\` ist, wird die Folgezeile als Fortsetzung interpretiert. Klammerstrukturen (Apostrophe, runde, eckige oder geschweifte Klammern, if-fi, for-do-done, while-do-done usw.) können über beliebig viele Zeilen gehen.

4 Hintergrundausführung und Exec-Kommando

Für einen normalen Programmaufruf wird ein neuer Prozess erzeugt. Der Shellprozess wartet, bis das Programm terminiert und gibt erst danach wieder den Prompt für die Eingabeaufforderung aus.

Stattdessen kann man ein Programm aber auch im Hintergrund ausführen. Dies bedeutet zunächst, dass der Shellprozess nicht auf dessen Terminierung wartet, sondern sofort den nächsten Befehl einliest. Man benutzt dies oft für langwierige nicht interaktive Aktivitäten (Datensicherung, längere Compilerläufe usw.).

Syntax:

```
Kommando &
```

Eine andere Möglichkeit des Programmaufrufs verzichtet auf die Prozesserzeugung. Der aktuelle Shellprozess wechselt in ein anderes Programm, die Shell ist damit terminiert!

Syntax:

```
exec Kommando
```

5 Umlenkung der Ein- und Ausgabe

Die Standardeingabe eines jeden Programms kann über die Shell zum Zeitpunkt des Aufrufs abweichend von der Tastatur festgelegt werden.

Syntax: *Programm-Aufruf* < *Eingabedatei-Name*

Das gleiche gilt für die Standard-Ausgabe, z.B.:

```
ls /home/guest >liste
```

Mit dem „>“-Zeichen wird die Ausgabedatei „liste“ neu angelegt bzw. überschrieben, mit „>>“ wird an eine bereits existierende Datei hinten angehängt.

Man kann nicht nur Standard-Ein- und Ausgabe neu binden, sondern beliebige Dateien. Jede Datei wird über einen Deskriptor angesprochen (0 - Standard-Eingabe, 1 - Standard-Ausgabe, 2 - Standard-Fehlerausgabe, weitere Deskriptoren werden gemäß der Reihenfolge des Öffnens vergeben).

Beispiel: Unterdrückung der Fehlermeldungen

```
xterm -display server:0.1  2> /dev/null
```

Beispiel: Kombination mehrerer Umlenkungen

```
ftp < ftp-kommandos > ftp-ausgabe 2> ftp-fehlerausgabe
```

Um Standardausgabe und Fehlermeldungen in die selbe Datei zu leiten, verwende man

```
programm  > ausgabedatei 2>&1
```

oder

```
programm  &> ausgabedatei
```

6 Operatoren für komplexe Kommandos

In einer Kommandozeile können beliebig viele Kommandos stehen, die durch verschiedene Operatoren verknüpft sind, z.B:

- sequentiell Ausführen:

```
gcc -c test1.c; gcc -o test1 test1.o; rm -f test1.o
```

- Ausführung nur bei Erfolg des vorangegangenen Kommandos:

```
gcc -c test1.c && gcc -o test1 test1.o && rm -f test.o
```

  Im Gegensatz zur sequentiellen Ausführung wird hier die Verarbeitungskette bei einem Fehler abgebrochen.

- Ausführung nur bei Misserfolg des vorangegangenen Kommandos:
  `test -d verzeichnis || mkdir verzeichnis`
  (Falls ein Unterverzeichnis namens *verzeichnis* nicht existiert, liefert der Aufruf des test-Kommandos -1 = "false" = Misserfolg. Nur in diesem Fall wird das zweite Kommando ausgeführt, das ein solches Unterverzeichnis erzeugt.)

6.1 Pipelines

Mit Pipelines wird Fließbandarbeit modelliert. An jeder „Station" des Fließbands werden Daten stationsspezifisch verarbeitet und an die nächste Station weitergereicht.

Die Verarbeitung erfolgt zeitlich verschränkt. Konkret wird die Ausgabe einer Station über einen Kommunikationskanal („Pipe") an die nächste Station weitergeleitet und dort als Eingabe gelesen.

```
ls -l | grep ^d | awk '{print $6}'
```

Im Beispiel sind 3 Prozesse an der Pipeline beteiligt, zwei Pipes werden genutzt.

Das Programm *ls* liefert eine Dateiliste. *grep* filtert aus der Ausgabe von *ls* alle Zeilen heraus, die ein d am Zeilenanfang haben, das sind die Einträge der Unterverzeichnisse. *awk* filtert diese Unterverzeichnis-Einträge noch einmal, und zwar spaltenweise: Von jedem Eintrag wird nur die 6. Spalte, der Dateiname, ausgegeben.

Ein weiteres Beispiel zeigt eine Pipeline zwischen Prozessen auf verschiedenen Rechnern zum Kopieren eines Verzeichnisses als tar-Archiv über das Netz:

```
tar cf - "$HOME" | gpg -c | bzip2 | ssh hg52@saturn "bzcat > archiv.tar.gpg"
```

Auf dem lokalen Rechner wird das Heimatverzeichnis rekursiv mit dem Archivprogramm *tar* in ein Dateiarchiv kopiert, das von gpg verschlüsselt, von bzip2 komprimiert und von ssh auf den entfernten Rechner *saturn* übertragen wird. Dort wird es wieder dekomprimiert (*bzcat*) und als Datei gespeichert.

Dieses Pipelining ist die Grundlage für die elegante Kombination mehrerer spezialisierter Utilities zu komplexeren Werkzeugen und damit ein in der Praxis äußerst wichtiges und häufig genutztes Konzept.

Durch den Einsatz spezialisierter Tools wie Archivierer + Verschlüsseler + Komprimierer erreicht man nicht nur eine in der Regel überlegene Funktionalität gegenüber einem „aufgeblasenen" Archivierer, der auch komprimieren und verschlüsseln kann.

Das ganze geht auch oft viel schneller, da die an der Pipeline beteiligten Prozesse gleichzeitig agieren und ohne Zwischendateien Daten austauschen. Man denke hier z.B. an die Verarbeitung großer Multimediaströme, wo in einer Pipeline Demultiplexer, Multiplexer, Video- und Audiostrom-Dekodierer und -Kodierer effizient kombiniert werden können, was sich in drastisch reduzierten Verarbeitungszeiten bemerkbar machen kann.

7 Kontrollstrukturen

Strukturierte Anweisungen können innerhalb von Kommandoprozeduren genauso wie im interaktiven Dialog benutzt werden. Die Anweisungen können mehrzeilig sein. Im Dialog benutzt die Shell inmitten einer komplexen Anweisung den Prompt $PS2 statt $PS1.

Die in der Syntaxspezifikation verlangten Semikolons können wegfallen, falls das nachfolgende Schlüsselwort in einer neuen Zeile steht!

Hier soll nur eine kurze Übersicht mit einigen Beispielen gegeben werden.

7.1 if-Anweisung

Syntax:

```
if list; then list; [ elif list; then list; ] ... [ else list; ] fi
```

Beispiel:

```
if test ! -d bs1; then
   mkdir bs1
   echo "Verzeichnis bs1 erzeugt"
else
   echo "Verzeichnis bs1 existiert schon"
fi
```

Man beachte, dass beliebige *Programme* hier als Bedingungen verwendet werden können, im Beispiel das Programm „test“. Wenn das Programm mit dem Exit-Wert 0 terminiert, wird der Programmaufruf mit dem booleschen Wert „WAHR“ gleichgesetzt, sonst mit „FALSCH“.

Das gleiche gilt auch für die Schleifenbedingung einer While-Schleife.

7.2 While-Schleife, Until-Schleife

Syntax:

```
      while list; do list; done
      until list; do list; done
```

Beispiel: Verarbeitung einer Datei

```
find / ( -name "*.mpg" -o -name "*.avi" ) -atime +365 |
 while read FILM; do
   mplayer "$FILM"
   echo -n "Löschen? [j/n]"
   read ANTWORT < /dev/tty
   [ "$ANTWORT" = j ] && rm "$FILM"
 done
```

Das find-Kommando sucht alle seit einem Jahr nicht mehr gelesenen Filmdateien und gibt die Liste über eine Pipeline an das While-Kommando, das jeden Film anspielt und nach Rückfrage ggf. löscht. Man beachte, dass die Standardeingabe des read-Kommandos explizit auf die Tastatur (/dev/tty) gesetzt werden muss, da die Standardeingabe des gesamten While-Kommandos die von *find* kommende Pipe-Ausgabe ist.

7.3 For-Schleife

Syntax:

```
for name [ in word ...; ]; do list ; done
```

Mit der For-Schleife wird eine Liste von Wörtern (Zeichenketten) auf die gleiche Weise verarbeitet. Oft ist es eine Liste von Dateipfaden oder die Liste der Parameter einer Funktion oder Kommandoprozedur.

Beispiel:

```
for DATEI in *.c *.cc *.h; do
   ci "$DATEI"  # Datei in der RCS-Versionsverwaltung einchecken
done
```

Falls die Liste der Wörter fehlt, werden die Parameter (Funktion oder Kommandoprozedur) $1, $2, . . . verarbeitet.

7.4 Case-Kommando

Syntax:

```
 case word in [ pattern [ | pattern ] ... ) list ;; ] esac
```

Case kommt zur Anwendung, wenn die Verarbeitung davon abhängt, ob eine Zeichenkette („word“) zu einem bestimmten Muster („pattern“) passt.

Beispiel:

```
function show {
   if [ $# = 0 ]; then echo "Argument fehlt"; return 1; fi
   for DATEI; do
      case "$DATEI" in
        *.jpg|*.png|*.gif) gimp "$DATEI" ;;
        *.mp3|*.mp4|*.avi) mplayer "$DATEI" ;;
        *) echo "show für Dateityp von \"$DATEI\" nicht definiert";;
      esac
   done
}
```

7.5 Listen

Syntax:

```
(list)
{ list; }
```

Man kann durch Einschließen mehrerer Kommandos in runde oder geschweifte Klammern Verbundkommandos konstruieren. Dies macht man oft wegen einer gemeinsamen Umlenkung der Ein- und/oder Ausgabe.

Bei runden Klammern erfolgt die Ausführung in einem Subprozess, bei geschweiften Klammern durch den aktuellen Shell-Prozess.

Im ersten Fall haben Änderungen von Variablen oder Verzeichniswechsel innerhalb der Kommandoliste keine Auswirkung auf die aktuelle Shell.

Beispiel:

```
$ pwd
/home/jaeger
$ ( cd /home; date; du -s * ) > platzbedarf-home
$ pwd
/home/jaeger
$ { cd /home; date; du -s *; } > platzbedarf-home
$ pwd
/home
```

In beiden Fällen dient die Gruppierung dazu, die Ausgabe des „date“- und des „du“-Kommandos („Disk usage“) in die gleiche Datei zu leiten. Im ersten Fall führt nur der Subprozess einen Verzeichniswechsel durch, im zweiten Fall die interaktive Shell.

8 Prädikate

Als Prädikate bezeichnen wir im folgenden solche Anweisungen, bei deren Ausführung man nur am exit-Code interessiert ist, und die in der Regel innerhalb von bedingten Anweisungen oder while-Anweisungen verwendet werden.

Die *bash* erlaubt zwei gleichwertige Schreibweisen: `test` *Bedingung* und `[␣`*Bedingung*`␣]`. Man achte auf die Leerzeichen bei den Klammern ! *Bedingung* ist dabei eine einfache Bedingung oder eine mittels logischer Operatoren (Konjunktion, Disjunktion, Negation) und Klammern gebildete komplexe Bedingung.

Nachfolgend die online-Beschreibung dazu:

```
test: test [expr]
    Exits with a status of 0 (trueness) or 1 (falseness) depending on
    the evaluation of EXPR.  Expressions may be unary or binary.  Unary
    expressions are often used to examine the status of a file.  There
    are string operators as well, and numeric comparison operators.

    File operators:

        -b FILE        True if file is block special.
        -c FILE        True if file is character special.
        -d FILE        True if file is a directory.
        -e FILE        True if file exists.
        -f FILE        True if file exists and is a regular file.
        -g FILE        True if file is set-group-id.
        -L FILE        True if file is a symbolic link.
        -k FILE        True if file has its "sticky" bit set.
        -p FILE        True if file is a named pipe.
        -r FILE        True if file is readable by you.
        -s FILE        True if file is not empty.
        -S FILE        True if file is a socket.
        -t [FD]        True if FD is opened on a terminal.  If FD
                       is omitted, it defaults to 1 (stdout).
        -u FILE        True if the file is set-user-id.
        -w FILE        True if the file is writable by you.
        -x FILE        True if the file is executable by you.
        -O FILE        True if the file is effectively owned by you.
        -G FILE        True if the file is effectively owned by your group.

      FILE1 -nt FILE2  True if file1 is newer than (according to
                       modification date) file2.
```

```
  FILE1 -ot FILE2  True if file1 is older than file2.

  FILE1 -ef FILE2  True if file1 is a hard link to file2.

String operators:

    -z STRING      True if string is empty.

    -n STRING
 or STRING         True if string is not empty.

    STRING1 = STRING2
                   True if the strings are equal.
    STRING1 != STRING2
                   True if the strings are not equal.

Other operators:

    ! EXPR         True if expr is false.
    EXPR1 -a EXPR2 True if both expr1 AND expr2 are true.
    EXPR1 -o EXPR2 True if either expr1 OR expr2 is true.

    arg1 OP arg2   Arithmetic tests.  OP is one of -eq, -ne,
                   -lt, -le, -gt, or ge.

Arithmetic binary operators return true if ARG1 is equal, not-equal,
less-than, less-than-or-equal, greater-than, or greater-than-or-equal
than ARG2.
```

Man beachte, dass es auch ein selbständiges Programm *test* gibt, das den gleichen Zweck erfüllt. Im Detail existieren jedoch einige Unterschiede bei den Ausdrücken.

9 Variablen und Umgebungsvariablen

Die Shell erlaubt die Verwendung von internen Variablen und Umgebungsvariablen. Die Verwendung ist nahezu identisch, die Werte sind in der Regel Zeichenketten. Variablen müssen vor ihrer Verwendung nicht deklariert werden.

Der Wert einer Variablen `X` ist `$X`. Innerhalb von Kommandos empfiehlt sich i.d.R. den Wert in Apostrophe einzuschließen.

Beispiel:

Wenn man in der weiter oben definierten Funktion *show* im Aufruf `gimp "$DATEI"` die Apostrophe weglässt, gibt es Probleme bei Dateinamen mit Leerzeichen. So werden für

```
DATEI="Dies ist ein schönes Bild.jpg"
```

beim Aufruf `gimp $DATEI` an Stelle des Dateinamens 5 Argumente an das Programm *gimp* übergeben: `"Dies" "ist" "ein" "schönes" "Bild.jpg"`. (Tatsächlich wird auch der Programmname „gimp“ noch zusätzlich als erstes Argument übergeben!)

Verwendungsbeispiele für Variablen:

```
$ BASISVERZEICHNIS=/home/jaeger/bs1   # Wert zuweisen
```

```
$ echo -n "Unterverzeichnis: "
$ read UVZ                          # Wert einlesen
$ cd "$BASISVERZEICHNIS/$UVZ"       # Verwendung

$ x=1
$ y=1
$ z="$x"+"$y"
$ echo "$z"'                        # Wert ausgeben
1+1
```

Die wichtigsten Operationen:

- Auswerten: Dem Variablennamen wird ein $ vorangestellt, z.B. `$HOME`. Es gibt eine ganze Reihe von weiteren Möglichlkeiten, etwa die Definition von Defaultwerten. Vergleiche hierzu das Manual.
- Wertzuweisung, z.B. `filename=test.c`
- Einlesen, z.B. `read filename`
- Ausgabe: auswerten und echo verwenden, z.B. `echo "$filename"`
- Konkatenation: einfach hintereinander schreiben. Variablennamen sind dabei in geschweifte Klammern einzuschließen, falls dies zur Abgrenzung des Namens erforderlich ist.
  Beispiel:

  ```
  echo -n "Dateiname ?"
  read filename
  if [ ! -f "$filename" ]
  then
     echo "Datei $filename existiert nicht"
   else
     mv "$filename" "${filename}.old"
     echo "Datei $filename wurde umbenannt in ${filename}.old"
   fi
  ```

- Stringlänge, z.B. `${#filename}`
- Anfangsstück (${*Bezeichner*#*Präfix*}) oder Endung (${*Bezeichner*%*Suffix*}) entfernen, z.B.

  ```
  for f in *.txt
  do
     mv "$f" "${f%.txt}.text"
  done
  ```

  (Endungen von `.txt` auf `.text` umstellen.)

Weitere String-Operationen sind im Manual beschrieben.

9.1 Umgebung

Unter „Umgebung“ (engl. „environment“) ist eine Reihe von Textvariablen zu verstehen, deren Definitionen bei Programmaufrufen und Prozesserzeugungen automatisch weitergegeben werden. (Im C-Programm wird mit *getenv* darauf zugegriffen).

Die Werte werden von unterschiedlichen Programmen zu verschiedenen Zwecken benutzt. Beispielsweise enthält der Wert der Umgebungsvariablen „HOME“ den Pfad zum Heimatverzeichnis des Benutzers. „PATH“ ist

die Liste der Verzeichnisse, in denen nach Programmen gesucht wird, falls nur deren Name (ohne Verzeichnis) angegeben wird. Ein anderes Beispiel ist die Variable „DISPLAY“, in der man einem graphischen Programm mitteilt, auf welchem Bildschirm es sein Fenster öffnen soll.

Eine Variable wird mit der *export*-Anweisung als Umgebungsvariable deklariert. Bei der Deklaration kann auch gleich ein Wert zugewiesen werden, z.B.

```
export PATH=/usr/bin:/bin:/usr/jaeger/bin:.
```

Mit dem *env*-Kommando kann man sich in der Shell die aktuelle Umgebung ansehen, mit dem *export*-Kommando ohne Argumente erhält man die Liste der Umgebungsvariablen.

Einige Umgebungsvariablen:

`$PATH`	Kommandosuchpfad
`$TERM`	Terminaltyp
`$LD_LIBRARY_PATH`	Pfad für die dynamisch benutzten Bibliotheken
`$DISPLAY`	Display-Identifikation für X-Clients
`$HOME`	Heimatverzeichnis
`$HOSTNAME`	Name des Rechners
`$MAIL`	Mail-Verzeichnis
`$MANPATH`	Pfade zu den On-Line-Manuals
`$SHELL`	Name der Shell
`$LANG`	Sprache für Internationalisierung

Einige interne Variablen der BASH:

`$?`	Status des letzten Kommandos
`$CDPATH`	Suchpfad für cd-Kommando
`$EDITOR`	Editor für bash-Historie
`$$`	Prozessnummer der Shell
`$!`	Prozessnummer des letzten Hintergrund-Kommandos
`$PWD`	aktuelles Verzeichnis
`$OLDPWD`	letztes aktuelles Verzeichnis
`$PS1`	Prompt-String
`$PROMPT_COMMAND`	Kommando, das vor jedem Prompt ausgeführt werden soll
`$MAILCHECK`	Zeitabstand zwischen zwei Tests auf neue E-Mail

9.2 Numerische Variablen (nicht POSIX-konform)

Will man Arithmetik mit Integer-Variablen betreiben, so sind die Variablen entweder explizit als integer zu deklarieren, z.B.

```
declare -i wert1
```

oder

```
typeset -i wert1
```

oder es ist bei Wertzuweisungen ein *let* voranzustellen.

Beispiel:

```
$ x=1
$ y=1
$ z=$x+$y
$ echo $z
1+1
```

```
$ let z=$x+$y
$ echo $z
2
```

Bei *let*-Anweisungen kann man das `$`-Zeichen auf der rechten Seite der Wertzuweisung auch weglassen, also statt `let z=$x+$y` schreibt man `let z=x+y` .

10 Anpassung durch Initialisierungsdateien

Die Login-Shell, das ist diejenige, die direkt beim Login eines Benutzers aufgerufen wird, liest zunächst die systemweite `/etc/profile`-Datei, in der der Administrator eine sinnvolle Standardumgebung definiert und danach die Datei `$HOME/.profile` .

Für benutzerspezifische Anpassungen sollten zwei Initialisierungsdateien im $HOME-Verzeichnis stehen:

`.profile` - wird von der Login-Shell gelesen

Hier gehören einmalig beim Login durchzuführende Aktionen hinein, außerdem benutzerspezifische Definitionen von Umgebungsvariablen, z.B. PATH-Erweiterungen.

Umgebungsvariablen werden bei Subprozesserzeugung und Programmaufruf vererbt, so dass die hier definierten Werte in der gesamten Sitzung verfügbar sind.

`.bashrc` - wird von anderen *bash*-Shells beim Aufruf gelesen (z.B. bei Start einer neuen Shell im *xterm*-Fenster).

Hier gehören benutzerdefinierte Kommandos (*alias* und *function*) und andere Shell-Initialisierungen (z.B. PROMPT-Einstellungen) hinein, die in allen Shells – nicht nur in der Login-Shell – verfügbar sein sollen.

Man beachte, dass die Dateien `.login` und `.cshrc` den gleichen Zweck erfüllen, aber speziell für die Verwendung mit der C-Shell *csh* gedacht sind.

Beispiel für eine `.profile`-Datei:

```
# Kommando-Suchpfad (in "/etc/profile" vorbelegt) erweitern
export PATH=$HOME/bin:$PATH:.

# englische Sun-Tastatur mit deutschen Umlauten auf ALT-Taste erweitern
if finger | grep "hg52.*dtlocal.*$DISPLAY" > /dev/null
then
    xmodmap $HOME/.Xmodmap.deutsche-umlaute-fuer-sunray
fi
```

Beispiel für eine `.bashrc`-Datei:

```
# Anzahl der während der Sitzung zu speichernden Kommandozeilen
HISTSIZE=200
# Anzahl der auf Platte zu speichernden Kommandozeilen
HISTFILESIZE=50
# alle 2 Minuten nach E-Mail schauen
MAILCHECK=120

# Prompt einstellen
PS1="$(hostname):\$(basename \$(pwd))> "

# einige Kommandos
alias ll="ls -la"
alias l="ls -a"
```

```
# Schrift fuer Terminal-Emulation groesser einstellen, Rollbalken (-sb):
alias xterm="xterm -sb -fn 9x15"

# Funktions-Beispiel: Anzeigen der Unterverzeichnisse
function dirs {
    ls -al "$@" | grep ^d | egrep -v "\.\.?"
    }
```

11 Eingebaute Kommandos

Meistens sind die Kommandos nichts anderes als Aufrufe selbständiger Programme, wobei man an diese Programme Parameter übergeben kann.

Die *bash* hat aber auch eine Reihe eingebauter Kommandos.

Das Kommando „help“ gibt eine Übersicht aus:

```
GNU bash, version 2.01.1(1)-release (i586-pc-linux-gnu)
These shell commands are defined internally.  Type ‘help’ to see this list.
Type ‘help name’ to find out more about the function ‘name’.
Use ‘info bash’ to find out more about the shell in general.

A star (*) next to a name means that the command is disabled.

 %[DIGITS | WORD] [&]               . filename
 :                                  [ arg... ]
 alias [-p] [name[=value] ... ]     bg [job_spec]
 bind [-lpvsPVS] [-m keymap] [-f fi break [n]
 builtin [shell-builtin [arg ...]]  case WORD in [PATTERN [| PATTERN].
 cd [-PL] [dir]                     command [-pVv] command [arg ...]
 continue [n]                       declare [-afFrxi] [-p] name[=value
 dirs [-clpv] [+N] [-N]             disown [-h] [jobspec ...]
 echo [-neE] [arg ...]              enable [-pnds] [-a] [-f filename]
 eval [arg ...]                     exec [-cl] [-a name] file [redirec
 exit [n]                           export [-nf] [name ...] or export
 false                              fc [-e ename] [-nlr] [first] [last
 fg [job_spec]                      for NAME [in WORDS ... ;] do COMMA
 function NAME { COMMANDS ; } or NA getopts optstring name [arg]
 hash [-r] [-p pathname] [name ...] help [pattern ...]
 history [-c] [n] or history -awrn  if COMMANDS; then COMMANDS; [ elif
 jobs [-lnprs] [jobspec ...] or job kill [-s sigspec | -n signum | -si
 let arg [arg ...]                  local name[=value] ...
 logout                             popd [+N | -N] [-n]
 pushd [dir | +N | -N] [-n]         pwd [-PL]
 read [-r] [-p prompt] [-a array] [ readonly [-anf] [name ...] or read
 return [n]                         select NAME [in WORDS ... ;] do CO
 set [--abefhkmnptuvxBCHP] [-o opti shift [n]
 shopt [-pqsu] [-o long-option] opt source filename
 suspend [-f]                       test [expr]
 time [-p] PIPELINE                 times
 trap [arg] [signal_spec ...] or tr true
 type [-apt] name [name ...]        typeset [-afFrxi] [-p] name[=value
 ulimit [-SHacdflmnpstuv] [limit]   umask [-S] [mode]
 unalias [-a] [name ...]            unset [-f] [-v] [name ...]
 until COMMANDS; do COMMANDS; done  variables - Some variable names an
 wait [n]                           while COMMANDS; do COMMANDS; done
 { COMMANDS }
```

Diese Kommandos können nicht alle hier erläutert werden. Beachten Sie die online-Hilfe, bzw. das Manual. Bei „vertrackten“ Problemen an Herrn Jäger wenden.

12 Kommandoprozeduren und Funktionen

Kommando-Prozeduren sind in einer Datei abgespeicherte Shell-Programme. Man nennt sie im UNIX-Umfeld *Shellscripts*. Sie eignen sich insbesondere zur Automatisierung komplexer Aktionen, einfache Befehlsfolgen ruft man effizienter als *bash-Funktionen* auf.

Man kann eine Kommandoprozedur als Programm ansehen, da zu den Shell-Kommandos programmiersprachliche Konstrukte wie bedingte Anweisungen oder Wiederholungsanweisungen, sowie ein Variablenkonzept gehören.

Der Aufruf erfolgt beim interaktiven Arbeiten mit der Shell in gleicher Weise wie bei einem (Maschinen-)Programm, es können auch Parameter übergeben werden. Während bei Maschinenprogrammen eine direkte Ausführung durch den Prozessor möglich ist, benötigt man zur Ausführung einer Kommandoprozedur eine Shell, die den Inhalt der Kommandoprozedurdatei Befehl für Befehl interpretiert und ausführt.

Aus Sicht eines Benutzers kann dieser Unterschied aber unsichtbar bleiben, wenn der Lademechanismus des Betriebssystems dies unterstützt. Dazu fügt man in die erste Zeile des Shellscripts eine besondere Kommentarzeile ein, z.B:

```
#!/bin/sh
```

Der Systemlader erkennt in diesem Fall das Shellscript als zu interpretierendes Programm und lädt statt des (nicht von der Maschine ausführbaren) Scripts den in dieser Zeile angegeben Interpretierer. Dabei werden die in der ersten Zeile angegebenen Argumente und zusätzlich der Pfad des Shellscripts an den Interpretierer übergeben. Auf diese Weise lassen sich Shellscripts und andere interpretierte Programme ohne expliziten Aufruf des Interpretierers aktivieren.

Beim Aufruf einer Kommandoprozedur wird in der Regel ein *neuer* Shellprozess erzeugt, der die Kommandofolge in der Datei bearbeitet.

Scripts kann man durch Sub-Shells ausführen lassen oder durch die aktuelle Shell:

- Bei der Ausführung durch eine Sub-Shell wird einfach der Pfad des Skripts angegeben, dabei können beliebig viele Parameter übergeben werden. Veränderungen Prozess-lokaler Variablen oder cd-Kommandos innerhalb des Skripts haben keinen Einfluss auf die aufrufende Shell.

  Im Skript kann auf die Parameter-Werte über $1, $2, ... zugegriffen werden. $# bezeichnet die Anzahl der Parameter, $$*$ und $@ stehen für die gesamte Parameterliste. Werden $$*$ bzw. $@ in doppelte Anführungszeichen eingeschlossen, expandiert die Shell bei $@ für jeden Parameter einen separaten String, für $$*$ dagegen nur einen String für alle Parameter.

- Bei Ausführung durch die aktuelle Shell ist dem Skript-Namen ein Punkt oder das Schlüsselwort `source` voranzustellen.

  Eine Parameterübergabe ist hier nicht möglich.

12.1 Funktionen

Funktionen erlauben einerseits die Strukturierung umfangreicher Shellscripts, können andererseits auch unabhängig von Shellscripts als ein weiterer Mechanismus zur Definition benutzerdefinierter komplexer Operationen verwendet werden. Gegenüber einem Skript erfordert der Aufruf einer Funktion keinen Dateizugriff, da die Shell die Funktionsdefinitionen (interaktiv oder in „.bashrc“) alle im Hauptspeicher hält.

Die Syntax:

```
function <NAME> { KOMMANDOS; }
```

oder

```
<NAME> () { KOMMANDOS; }
```

Der Zugriff auf Parameter erfolgt in gleicher Weise wie bei Shellscripts. Lokale Variablen werden mit „local“ definiert, und der Erfolg der Ausführung wird mit „return 0“ statt „exit 0“ bekanntgegeben.

Beispiele:

```
function abort {
  echo "** ABBRUCH: $*"
  exit 1
}

function filecheck {
  local F
  for F; do
    [ -f "$F" ] || abort "$F nicht lesbar"
  done
}

function datum {
  # setzt 3 globale Variablen
  JAHR="'$(date "+%Y")"
  MONAT="$(date "+%m")"
  TAG="$(date "+%d")"

  # falls $1=--show, Datum ausgeben
  [ "$1" = --show ] && echo "$TAG.$MONAT.$JAHR"
}

function zeit  {
  STUNDE="$(date "+%H")"
  MINUTE="$(date "+%M")"
  SEKUNDE="$(date "+%S")"

  # falls $1=--show, Datum ausgeben
  [ "$1" = --show ] && echo -n "$STUNDE:$MINUTE:$SEKUNDE"
}

# Datum und Uhrzeit via Kommandoersetzung merken:
zeitpunkt="$(datum --show) $(zeit --show)"
```

13 Kommando-Ersetzung

Ein von UNIX-Kennern ständig benutztes, von Einsteigern leider oft zunächst nicht genügend beachtetes Konzept ist die Umleitung der Ausgabe eines Programms in die aktuelle Kommandozeile, auch als „Kommando-Ersetzung“ bekannt:

Dafür gibt es zwei syntaktische Varianten:

```
$(kommando)
```

oder die ältere Form:

```
‘kommando‘
```

Die $-Syntax hat den Vorteil, dass sie verschachtelte Kommandoersetzungen ermöglicht.

Beides wird jeweils durch die Standardausgabe des Kommandos ersetzt: Die Shell führt das Kommando aus, lenkt dessen Ausgabe aber in eine Pipe um. Die Shell liest die Ausgabe des Kommandos aus der Pipe und setzt dann in der Kommandozeile die gelesenen Daten ein.

Dies macht man sich regelmäßig zunutze, um dasselbe Programm, das im „Normalfall“ als Prozedur mit dem Seiteneffekt *Ausgabe* benutzt wird, im Bedarfsfall als Funktion zu verwenden, die ein String-Resultat zurückliefert.

Beispiel:

Das Kommando *pwd* gibt das aktuelle Verzeichnis auf die Standard-Ausgabe aus. In einem beliebigen Kommando wird also `$(pwd)` (oder `‘pwd‘`) durch den Namen des aktuellen Verzeichnisses ersetzt.

Die Eleganz dieses Konzepts zeigt sich beim Vergleich mit VMS, wo zur Anzeige von System-Informationen SHOW-Kommandos genutzt werden müssen (z.B. SHOW DEFAULT-aktuelles Verzeichnis oder SHOW PROCESS-Prozess-Info) zur Verwendung in der Kommandozeile allerdings nur eine beschränkte Reihe sogenannter lexikalischer Funktionen (F$DIRECTORY()-aktuelles Verzeichnis, F$GETJPI()-Prozess-Info) zur Verfügung stehen.

Man betrachte die `.bashrc`-Datei weiter oben als Beispiel:

```
if [ "$(tty)" != /dev/console ]; then exit; fi
```

tty ist ein Programm, das als Resultat den aktuellen Terminal-Namen ausgibt. Diese Ausgabe wird hier via Kommando-Ersetzung als Funktionsresultat verwendet.

Ein anderes Beispiel, das weiter oben schon genutzt wurde:

```
PS1='$(hostname): '
```

Ein weiteres Beispiel: Alle Dateien, die neuer sind als die Datei *last-backup.time-stamp*, sollen mit tar auf Band gesichert werden:

```
find / -newer last-backup.time-stamp -print >liste
tar cv $(cat liste)
```

oder

```
tar cv $(find / -newer last-backup.time-stamp)
```

14 Dateinamen-Ersetzung

Die Verwendung von sogenannten „Wildcards“ innerhalb von Dateinamen, die für eine wohldefinierte Menge von Zeichenketten stehen, ist in vielen Kommandosprachen üblich. Die konkrete Semantik unterscheidet sich jedoch teilweise sehr stark zwischen den Betriebssystemen.

Bei UNIX-Shells, wie der bash, repräsentiert ein „*“ eine beliebige, ggf. auch leere, Zeichenkette und ein „?“ genau ein (beliebiges) Zeichen.

Will man sich etwa alle Dateien im Verzeichnis /usr/src auflisten lassen, deren Namen mit .c enden, so kann man dies mit dem Kommando

```
ls /usr/src/*.c
```

auf einfache Weise bewerkstelligen.

Der entscheidende Punkt im Hinblick auf die Semantik ist, dass die Bearbeitung dieser Sonderzeichen durch die Shell geschieht, nicht aber durch das Programm *ls*. Die Shell ersetzt also zunächst innerhalb der Kommandozeile die Dateinamen-Schablone `/usr/src/*.c` durch die Liste der Namen aller „passenden“ Dateien. Die resultierende, ggf. sehr lange Liste von Pfaden wird sodann an das ls-Programm übergeben, welches selbst keinerlei „Wildcard“-Bearbeitung durchführt.

Ganz anders behandelt dagegen MSDOS das Kommando

```
dir \usr\src\*.c
```

Hier wird die Schablone vom dir-Kommando in einer Kommando-spezifischen Weise bearbeitet, einen allgemeinen Ersetzungsmechanismus gibt es nicht.

Was ist zu tun, wenn man an ein Programm eine Zeichenkette übergeben will, die ein solches Sonderzeichen enthält ? Wenn man etwa dem Archiv-Programm *tar* auftragen will, aus dem Archiv `arc.tar` alle Dateien mit der Endung `.c` zu extrahieren.

```
tar xf arc.tar *.c
```

führt nicht zum gewünschten Erfolg, da *tar* statt der Schablone `*.c` das Ergebnis der Expansion durch die Shell übergeben bekommt.

Man kann die Expansion durch die Shell auf verschiedene Weise unterbinden:

- Einschließen des Parameters in Gänsefüßchen, z.B. "`*.c`"
- Einschließen des Parameters in Apostrophe, z.B. `'*.c'`
- Voranstellen des Fluchtzeichens \ vor das Sonderzeichen, z.B. `\*.c`

Die erste und zweite Möglichkeit unterscheiden sich im Hinblick auf die Ersetzung von Variablennamen: innerhalb von Gänsefüßchen werden Variablen ersetzt, innerhalb von Apostrophen nicht.

Eine UNIX-Besonderheit ist die Behandlung von Dateinamen, die mit einem Punkt beginnen. Solcherart benannte Dateien werden in mancher Hinsicht schamhaft vor den Augen des Benutzers versteckt. Dazu gehört zum einen, dass das Kommando *ls* solche Dateien nur auflistet, wenn spezielle Optionen im Aufruf angegeben sind. Zum anderen werden beim Expandieren von Schablonen, die mit einem * beginnen, diese Dateien ignoriert.

Man beachte, dass ein Punkt an anderer Stelle im Dateinamen dagegen keinerlei Sonderbehandlung erfährt.

Weitere Möglichkeiten zur Musterbildung entnehme man dem *bash*-Manual.

15 Kommando-Puffer

Die *bash* merkt sich die letzten $HISTSIZE Kommandozeilen in einem Puffer, der als history-Puffer bezeichnet wird.

Es gibt viele Möglichkeiten, auf diesen Puffer zuzugreifen, sowohl mit *emacs*-Editierkommandos als auch mit den „history“-Kommandos der C-Shell *csh.*

Die history wird am Ende der Sitzung automatisch abgespeichert ($HISTFILE) und bei der nächsten Sitzung wieder geladen.

Basis-Funktionen:

- Mit den Cursor-Tasten kann man innerhalb des Kommando-Puffers Kommandos selektieren, editieren und erneut ausführen. Zum Editieren werden, abhängig von der Umgebungsvariable `$EDITOR`, *emacs*- oder *vi*-Kommandos verwendet.

- *Ctrl-R* wird oft benutzt, um in der History rückwärts nach einem bestimmten Kommando zu suchen (nur emacs-Modus, entspricht incremental-search-backward).
- Als Default-Alias ist *r* für "`fc -s`" definiert (vgl. help fc), wenn nicht, sollte man es in `.bashrc` so definieren:

  ```
  alias r="fc -s"
  ```

  Dieses Kommando erlaubt die Wiederholung eines im Puffer gespeicherten Kommandos, wobei man in der Form *Pattern=Relacement* Änderungen spezifizieren kann. Im Gegensatz zu den oben beschriebenen Zugriffsmethoden wird das gefundene Kommando sofort ausgeführt, d.h. man kann es nicht mehr editieren und muss die RETURN-Taste nicht noch einmal betätigen.

  Beispiele:

`r`	letztes Kommando wiederholen
`r cc`	letzten C-Compiler-Aufruf wiederholen
`r .txt=.text cp`	letztes cp-Kommando wiederholen, dabei aber `.txt` durch `.text` ersetzen

16 Heimat-Verzeichnis-Ersetzung

Das Heimatverzeichnis $HOME wird jedem Benutzer in dessen /etc/passwd-Datei-Eintrag zugeordnet. Stellt man dem login-Namen eines Benutzers eine Tilde ~ voran, wird die Shell dafür das Heimatverzeichnis des Benutzers einsetzen.

Dieser Ersetzungsmechanismus ist in erster Linie für Administratoren nützlich, die öfters in den HOME-Verzeichnissen anderer Benutzer Dateien verarbeiten müssen.

Eine Tilde ohne Benutzername ist äquivalent zu $HOME, also zum Heimatverzeichnis des aktuellen Benutzers.

17 Eval-Kommando – Beispiel: dynamische Berechnung von Variablennamen

Vor der Kommandoausführung werden die üblichen Ersetzungen auf der Kommandozeile durchgeführt. Das eingebaute *eval*-Kommando bewirkt einen zusätzlichen Auswertungs- bzw. Ersetzungsdurchgang, der für einige Sonderfälle nützlich ist. Als Beispiel betrachten wir dynamische Variablennamen.

Bei manchen Anwendungen möchte man Variablenbezeichner mit Textverarbeitungsoperationen konstruieren. Ein typisches Beispiel ist die Simulation von Feld-Variablen (Felder werden derzeit von der bash nicht direkt unterstützt):

Man konkateniert den Feldnamen und den Wert des Index zu einem neuen Namen, der die Feldkomponente repräsentiert.

Die Bearbeitung solcher Kommandos erfordert mehrere Auswertungsphasen, die durch das eingebaute eval-Kommando kontrolliert werden können. eval wertet zunächst seine Argumente aus. Das Resultat wird dann als Kommandozeile interpretiert, wobei ein weiteres Mal Ersetzungen vorgenommen werden können.

Bsp: 20 Werte einlesen und wieder ausgeben

```
i=1
while [ $i -le 20 ]; do
   read wert$i
   let i=$i+1
done
i=1
```

```
while [ $i -le 20 ]; do
   eval echo \$wert$i
   let i=$i+1
done
```

Bei der Wertausgabe benötigt man 2 Auswertungsphasen, um den i-tem Wert zu erhalten. In der ersten Phase wird der Variablenname wert$i „berechnet“, in der zweiten die dynamisch bestimmte Variable ausgewertet. Die direkte Angabe von $wert$i hätte, wie man sofort erkennt, nicht den gewünschten Erfolg, da die *bash* versucht, den Wert der Variable beiden *wert* und *i* zu bestimmen.

18 Job-Kontrolle

Hat ein Benutzer mehrere aktive Prozesse, so kann er zu einem bestimmten Zeitpunkt doch nur mit einem dieser Prozesse interaktiv (im Vordergrund) arbeiten, die anderen sind nicht interaktive Hintergrund-Prozesse.

Job-Kontrolle bezeichnet den Shell-Mechanismus zur Steuerung von Subprozessen, insbesondere die Fähigkeit, einen Prozess vom Vordergrund in den Hintergrund zu verlagern und umgekehrt. (Diese Steuerungsfunktion wird unter X-Windows in der Regel durch den Window-Manager übernommen.)

Der Mechanismus ist einfach zu bedienen: Durch Anfügen von & am Ende eines Kommandos wird ein Hintergrundprozess gestartet. Der Vordergrundprozess kann jederzeit durch *CTRL-Z* unterbrochen werden. Mit dem Hintergrund-Kommando *bg* (background), kann der unterbrochene Vordergrundprozess im Hintergrund weitergeführt werden. Mit dem Vordergrund-Kommando *fg* (foreground) kann ein beliebiger Hintergrund-Prozess zum Vordergrund-Prozess gemacht werden.

Ein Hintergrund-Job kann mit dem *stop*-Kommando unterbrochen und mit dem *kill*-Kommando abgebrochen werden. Mit dem *jobs*-Kommando lässt sich eine Statustabelle aller vorhandenen Jobs ausgeben.

Falls ein Hintergrund-Job auf eine Ein- oder Ausgabe wartet, meldet dies die Shell.

Beispiel:

Abwechslendes Editieren eines Quelltextes mit dem *vi*-Editor und Compilieren ohne X-Windows.

`vi source.c`	(Editor-Aufruf)
Betätigen von *CTRL-Z*	(Editor unterbrechen)
`make source&`	(Compiler-Lauf im Hintergrund starten)
`fg %vi`	(Editieren fortsetzen)

19 Signal-Behandlung

Bei der Entwicklung komplexer Kommandoprozeduren ist in der Regel eine gezielte Behandlung von Unterbrechnungen notwendig, z.B. um bei einem Abbruch-Signal noch irgendwelche temporären Dateien löschen zu können.

Hierzu dient das *trap*-Kommando, das meistens wie folgt benutzt wird:

trap *Kommando Signale*

Das spezifizierte *Kommando* wird ausgeführt, sobal eines der aufgelisteten Signale eintrifft.

Bsp.:

```
trap " echo Abbruch; rm /tmp/mytempfile; exit -1 " INT QUIT HUP
```

(Weitere Möglichkeiten vgl. Online-Beschreibung.)