# Bourne-Shell als Kommando– und Programmiersprache

Matthias Clauß*
TU Chemnitz
Universitätsrechenzentrum

3. November 1997

*Telefon: (0371) 531 1468 E–Mail: m.clauss@hrz.tu-chemnitz.de

# Gliederung

# 1 Einführung

Shell: Kommunikationsschnittstelle Nutzer $\Leftrightarrow$ System

Merkmale:

- dialogorientierter Kommandointerpreter
- Sprachkonzept einer höheren Programmiersprache implementiert:
  - Datentypen: Zeichenketten, Files, Verzeichnisse
  - Operatoren: Kommandos (Werkzeuge)
  - Steuerstrukturen
- herausgelöst aus dem Systemkern; keine Privilegien, sondern „normales“ Anwenderprogramm (austauschbar)
- „Bourne“–Shell/„C“–Shell/„Korn“–Shell
- hier erläutert: „Bourne“-Shell

## 2 Literaturverzeichnis


- Kochan, S. , Wood, P. : UNIX Shell Programmierung
  te–wi Verlag, München, 1991

- Kerninghan, B.W. , Pike, R. : The UNIX Programming Environment
  Prentice Hall, Inc., 1984

- Arthur, L. J. : UNIX Shell Programming
  John Wiley & Sons, Inc., 1990

- Herold, H. : UNIX–Shells
  Addison–Wesley, 1992

- Morgan, R. , McGilton, H. : Introducing the UNIX System V.4
  McGraw-Hill, New York, 2nd ed. 1991

- Banahan, M. ,Rutter, A. : UNIX: Lernen, Verstehen, Anwenden
  Carl Hanser Verlag, München, 1984 (Übersetzung)

- Gulbins, J. : UNIX
  Springer-Verlag, 3. Auflage, 1988

- Bolsky, M. I. , Korn, D. G. : The KORNSHELL Command and Programming Language
  Prentice Hall, 1989

- Anderson, G. , Anderson, P. : The UNIX C Shell Field Guide
  Prentice Hall, 1986

# 3 Kommandos

- **Kommando**: Auftrag an das System zur Ausführung einer Funktion. Im Normalfall resultiert hieraus die Ausführung eines Programmes, nur wenige Funktionen sind fest in der Shell „verdrahtet“ (built-in-commands).

- **Kommandoaufruf**: Folge von ein oder mehreren Worten, die untereinander durch Leerzeichen oder Tabulatorzeichen getrennt sind

  *cmd_name* [$arg_1$ ... $arg_n$ ]

  wobei *cmd_name* den Namen des Kommandos (d.h. des aufzurufenden Programmes) angibt und die restlichen Worte (falls vorhanden) die Kommandoargumente spezifizieren, d.h. zusätzlich Informationen an das Programm übergeben

  – Beispiele:

```
$ ls              Listen der Filenamen im aktuellen Verzeichnis
kap01.tex
kap02.tex
kap03.tex
kap04.tex
kap05.tex
abbildungen
$
$ ls -l          "lange Liste"
total 106
-rw-rw----   1 wmc      urzbur      1802 Apr 23 13:41 kap01.tex
-rw-rw----   1 wmc      urzbur      4681 Apr 23 14:25 kap02.tex
-rw-rw----   1 wmc      urzbur      4165 Apr 23 13:29 kap03.tex
-rw-rw----   1 wmc      urzbur      2898 Apr 23 13:26 kap04.tex
-rw-rw----   1 wmc      urzbur      3101 Apr 23 13:26 kap05.tex
drwxrwx---   1 wmc      urzbur       512 Apr  9 16:20 abbildungen
$
$ ls abbildungen
bild01.fig
bild02.fig
bild03.fig
bild04.fig
bild05.fig
bild06.fig
bild07.fig
$ vi kap01.tex
```

- Mehrere Kommandos können durch Semikolon auf einer Zeile eingegeben werden, sie werden sequentiell nacheinander abgearbeitet:

```
$ cd sort; pg sort.c sort.h subr.c; cd
```

- Bemerkungen:
  - Arbeitsweise der Shell
  - Argumente werden an das Programm durchgereicht
  - „Prompt“-Zeichen

$ *cmd* &

- Hintergrundprozesse $ $cmd_1$ & $cmd_2$ $\Rightarrow$ $cmd_1$ *als Hintergrundprozeß*

  $cmd_2$ *als Vordergrundprozeß*

  - Beispiele:

```
$ cc prog.c &
632
$ ps
  PID TT STAT  TIME COMMAND
  630 p3 S     0:00 sh
  632 p3 S     0:00 cc prog.c
  634 p3 R     0:00 ps
$
$ ls & ps
652
kap01.tex
kap02.tex
kap03.tex
kap04.tex
kap05.tex
kap06.tex
kap07.tex
kap08.tex
kap09.tex
prog.c
  PID TT STAT  TIME COMMAND
  640 p4 S     0:00 /usr/openwin/bin/xview/shelltool
  641 p4 IW    0:00 -bin/csh (csh)
  643 p4 IW    0:00 -sh -c umask 006; cd ~bur;newgrp urzbur (csh)
  645 p4 S     0:00 -bin/csh (csh)
  651 p4 S     0:00 sh
  652 p4 R     0:00 ls
  653 p4 R     0:00 ps

$ ps & ls
654
kap01.tex
kap02.tex
```

```
kap03.tex
kap04.tex
kap05.tex
kap06.tex
kap07.tex
kap08.tex
kap09.tex
prog.c
  PID TT STAT  TIME COMMAND
  640 p4 S     0:02 /usr/openwin/bin/xview/shelltool
  641 p4 IW    0:00 -bin/csh (csh)
  643 p4 IW    0:00 -sh -c umask 006; cd ~bur;newgrp urzbur (csh)
  645 p4 IW    0:00 -bin/csh (csh)
  651 p4 S     0:00 sh
  654 p4 R     0:00 ps
$
```

– Bemerkungen:
  * Simultane Kommandoausführung
  * während der Verarbeitung eines Vordergrundkommandos „schläft“ die Shell
  * Prozeßnummer des Hintergrundprozesses ⇒ Abbruch möglich
  * (evtl.) „gemischte“ Ausgaben am Terminal
  * vermeiden: Terminaleingabe für Hintergrundprogramm
    Beispiel: (evtl.) folgendes Script **inputst** ausführen lassen:

```
$ cat inputtst

:
#
while true
do
    echo -n "Soll die Verarbeitung fortgesetzt werden? [j/n]:"
    read ans
    case $ans in
        j|J)    echo JA;continue;;
        n|N)    echo NEIN;break;;
          *)    echo $ans : FEHLERHAFTE EINGABE;continue;;
    esac
done

$ inputtst
soll die Verarbeitung fortgesetzt werden? [j/n]:j
JA
Soll die Verarbeitung fortgesetzt werden? [j/n]:n
NEIN
$
$ inputtst &
```

```
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
soll die Verarbeitung fortgesetzt werden? [j/n]:: FEHLERHAFTE EINGABE
 ...
```

# 4 Standard- E/A und Fileumlenkung

- Shell stellt jedem Programm, das sie aufruft, drei E/A- Verbindungen (Kanäle) bereit:
  - Standardeingabefile (standard input — stdin)
  - Standardausgabefile (standard output — stdout)
  - Diagnoseausgabefile (standard error — stderr)
- Programme, die ihre E/A über diese logischen Verbindungen erledigen, brauchen sich um das Öffnen bzw. Schließen nicht kümmern

  Beispiel: Funktionen der C-Standardbibliothek:

  | | | |
  |---|---|---|
  | **getchar(), scanf()** | ⇒ | Lesen von stdin |
  | **putchar(), printf()** | ⇒ | Schreiben nach stdout |
  | **fprintf()** | ⇒ | Schreiben nach stderr |

  Beispiele für Programme, die ihre Informationen auf stdout ausgeben:

  $ *ls*, *ps*, *who*, *date*, *echo*, *cat*

  Beispiele für Programme, die von stdin lesen und nach stdout ausgeben:

  $ *sh*, *sort*, *cat*, *wc*, *tr*
- Normalerweise sind diese 3 Standardfiles dem Terminal des betreffenden Nutzers zugeordnet, sie können jedoch auf einfache Weise auf beliebige Files „umgelenkt“ werden (i/o redirection):

  Beispiel: für Fileumlenkung

```
$ ls >tmpfile

$ wc -l <tmpfile
    11
$ cat tmpfile
kap01.tex
kap02.tex
kap03.tex
kap04.tex
kap05.tex
kap06.tex
kap07.tex
kap08.tex
kap09.tex
rahmen.tex
tmpfile
$
```

  allgemeine Notation:

  | | |
  |---|---|
  | <*ifile* | Lesen *ifile* |
  | >*ofile* | Schreiben *ofile*, falls *ofile* bereits existierte, so wird es überschrieben |
  | >>*ofile* | Fortschreiben von *ofile* |

Problem: cmd <file >file ?!

- Shell als Kommando:

  File *shellproc* enthalte Folge von Kommandos

  $ sh <*shellproc*

  ...

- Bemerkungen:
  - Ausnutzung von zwei Tatsachen
    1. Shell ist „gewöhnliches“ Programm
    2. Shell liest von der Standardeingabe
  - Ablauf: „aktuelle“ Shell ruft sich quasi selbst auf (Kopie) und wartet, bis diese neue Shell-Instanz das File *shellproc* gelesen hat (EOF) und die darin enthaltenen Kommandos ausgeführt worden sind
  - es existieren noch weitere Möglichkeiten zur Ausführung von Shell-Scripts (s. Abschnitt Shell-Scripts)
  - *shellproc* kann mit dem Editor vi erzeugt werden, hier jedoch Nutzung von cat plus Fileumlenkung („primitive Texterfassung“):

```
$ cat >shellproc
ls >tmpfile
wc -l <tmpfile
rm tmpfile
<CRTL-D>
$
```

  - Es existieren weitere (selten benutzte) Möglichkeiten der Fileumlenkung; den logischen E/A-Kanälen sind sogenannte Filedeskriptoren (*fd_nr*) zugeordnet, z.B.

    0 ⇒ stdin
    1 ⇒ stdout
    2 ⇒ stderr

    1. Umlenken Ausgabekanal *fd_nr* nach *ofile*:
       *fd_nr* >*ofile*

       z.B. **2**>*diagnose* ⇒ stderr in *diagnose* umgelenkt

    2. Umlenken Eingabekanal *fd_nr* auf *ifile*:

       *fd_nr* <*ifile*

    3. Umlenken Standardausgabe in Ausgabekanal:

       *fd_nr* >&*fd_nr*

4. Umlenken Standardeingabe auf Eingabekanal

   $fd\_nr <\&fd\_nr$

5. Verallgemeinerung:

   $fd\_nr_1 > \&fd\_nr_2$

   $fd\_nr_1 < \&fd\_nr_2$

   z.B. **2>&1** $\Rightarrow$ Standardfehlerausgabe auf Standardausgabe gemischte Ausgaben

6. stdin, stdout schließen

   >**&-** bzw. <**&-**

7. „here"-Dokumente

   <<*wort*   darauf wird später im Zusammenhang mit Shell-Scripts eingegangen.

# 5 Pipelineverarbeitung

- Eine Pipeline (pipe) entsteht, wenn die Standardausgabe eines Programmes mit der Standardeingabe eines anderen Programmes verknüpft wird:

  $\ldots cmd_i \;\; | \; cmd_{i+1} \;\; \ldots$

  allgemein:

  $cmd_1 \mid cmd_2 \mid \ldots \mid cmd_n$

- Pipelines kann man sich vorstellen als einen Datenstrom, der durch verschiedene Programme (Filter) hindurchläuft.
  - Beispiel:
    ```
    $ who | wc -l
    $ ls | wc -l
    $ ls -l | grep  '^[Dd]'
    $ ls | pr -3 | lpr
    ```
  - Bemerkungen:
    * wc, grep,pr sind Filter
    * ls ist kein Filter ⇒ nur als erstes Element einer Pipeline möglich
    * falls Ausgabe sowohl auf stdout als auch in ein File geschrieben werden soll: **tee** (T-Verbindung)
  - Vergleich zur Methode der Fileumlenkung:
    ```
    $ ls >tmpfile; wc -l <tmpfile; rm tmpfile
    ```
    liefert (fast) das gleiche Ergebnis wie
    ```
    $ ls | wc -l
    ```
    aber:
    * Pipeline-Methode ist einfacher und bequemer
    * **tmpfile** wird durch Shell vor Aufruf von **ls** erzeugt ⇒ ein File mehr im aktuellen Verzeichnis (evtl. Problem: Berechtigung, Files im akt. Verzeichnis zu erzeugen!)
    * **ls** und **wc** werden synchron von der Shell gestartet, die zu einer Pipeline gehörenden Programme werden parallel abgearbeitet, die Datenübergabe erfolgt über interne Puffer (evtl. warten)

- Pipelines sind eine Grundlage für das Tool-Konzept im UNIX:
  - kleine, überschaubare Programme für einen bestimmten Zweck; komplexere Aufgaben können leicht durch Kombination der Programme gelöst werden
  - Beispiel:

    * anstelle **ls** mit spezieller Option „Anzahl der Files“ und **who** mit der Option „Anzahl der Nutzer“ können beide Programme mit **wc** kombiniert werden
    * dto. bzgl. Mehrspaltendruck usw.

- Bemerkungen:
  - Programme merken nicht, daß sie Bestandteil einer Pipeline sind
  - Sie müssen (bis auf das erste) von der Standardeingabe lesen und auf die Standardausgabe (bis auf das letzte) ausgeben
  - Konvention: Lesen der Standardeingabe, falls bei Aufruf kein Filename-Argument angegeben wird (z.B. sort, grep, wc)
  - (fast) nie bedeutet die Angabe eines Filename-Argument das Ausgabefile des Programmes ! (Ausnahme: **tee**)
  - Forderung an Filter:
    * Trennung „normale“ Ausgabe von Fehlernachrichtenausgabe
    * Keine Überschriftenausgabe und dergleichen in das Programm „verdrahten“

# 6 Filenamengenerierung

- Verwendung von Metazeichen zur Bezeichnung einer Gruppe von Filenamen bei der Angabe von Filenamen-Argumenten

  (Begriffe: Filenamengenerierung, pattern matching, wild-cards (Joker))

- Metazeichen und ihre Bedeutung

  | | |
  |---|---|
  | * | repräsentiert jede Zeichenkette einschließlich der leeren Zeichenkette (Zeichenkette mit der Länge 0) |
  | ? | repräsentiert ein beliebiges Zeichen |
  | [ | Beginn einer Zeichenklassenangabe |
  | ] | Abschluß einer Zeichenklassenangabe |
  | - | Zeigt eine lexikographisch geordnete Zeichenklasse an |

- Beispiele:

  im aktuellen Verzeichnis befinden sich die Files

```
kap01.tex
kap01.dvi
kap02.tex
kap02.dvi
kap03.tex
kap03.dvi
kap04.tex
kap04.dvi
kap05.tex
kap05.dvi
prog.c
prog.o
prog1.c
sort

$ ls *               -> alle Files

$ ls kap*            -> kap01.tex, kap01.dvi, kap02.tex, kap02.dvi,
                        kap03.tex, kap03.dvi, kap04.tex, kap04.dvi,
                        kap05.tex, kap05.dvi

$ ls *.c             -> prog.c, prog1.c

$ ls kap*.tex        -> kap01.tex, kap02.tex, kap03.tex, kap04.tex, kap05.tex

$ ls kap04.tex*      -> kap04.tex

$ ls kap04.tex?      -> nichts

$ ls ?????????       -> kap01.tex, kap01.dvi, kap02.tex, kap02.dvi,
```

```
                      kap03.tex, kap03.dvi, kap04.tex, kap04.dvi,
                      kap05.tex, kap05.dvi

$ ls kap0[123]*    -> kap01.tex, kap01.dvi, kap02.tex, kap02.dvi,
                      kap03.tex, kap03.dvi

$ ls kap0[1-3]*    -> kap01.tex, kap01.dvi, kap02.tex, kap02.dvi,
                      kap03.tex, kap03.dvi

$ ls kap0[1-9].dvi -> kap01.dvi, kap02.dvi, kap03.dvi, kap04.dvi, kap05.dvi

$ ls kap0[1-12].tex -> kap01.tex, kap02.tex

$ ls *.[a-d]*      -> kap01.dvi, kap02.dvi, kap03.dvi, kap04.dvi,
                      kap05.dvi, prog.c, prog1.c
```

- Bemerkungen:
  - trifft natürlich für alle Kommandos zu
  - Beispiel: cc *.c
  - Problem: Anzahl der Kapitel übersteigt 9, z.B. 12
    **kap0[6-9].tex kap1[0-2].tex** $\Rightarrow$ kap06.tex bis kap12.tex
  - Problem: Metazeichen sind selbst im Filenamen enthalten $\Rightarrow$ „Flüchten“ der Sonderbedeutung mittels \
  - Beispiel: Filter, die auf ***** enden, können durch das Muster ** „adressiert“ werden
  - Testen der Effekte der Filenamengenerierung durch das Kommando **echo**:
  - Beispiel:

```
$ echo rm kap0[1-3].dvi
   rm kap01.dvi kap02.dvi kap03.dvi
```

# 7 Shell-Scripts

## 7.1 Shell-Prozedur/Kommandofile/Shell-Programm, Shell-Script:

- Folge von Kommandos, die in einem File abgespeichert ist
- Das File kann Anweisungen enthalten, die zur Steuerung der Kommandoausführung dienen
- File wird normalerweise mit einem Texteditor erstellt (z.B. *vi*)
- Beispiel: File *akt_info*

```
/bin/echo -n "aktuelles Datum:"
date
echo  "angemeldet sind:"
who
/bin/echo -n "das aktuelle Verzeichnis ist:"
pwd
```

## 7.2 3 Möglichkeiten der Ausführung von Shell-Scripts:

**sh <cmdfile**

- Aufruf einer neuen Shell-Instanz (Sub-Shell) mit Umlenkung der Standardeingabe auf das File **cmdfile**. Beim Erreichen von EOF beendet diese Shell ihre Arbeit und der ursprüngliche Shell-Prozeß wird wieder aktiv. Diese Möglichkeit ergibt sich aus dem normalen Mechanismus der Verarbeitung von Kommandos durch die Shell.

**sh** [*flags*] **cmdfile** [$arg_1$ ... $arg_n$ ]

- Wenn die Shell mit einem Filename als Argument aufgerufen wird, so werden die in diesem File enthaltenen Kommandos durch die neue Shell-Instanz abgearbeitet. Der Vorteil besteht darin, daß Argumente an dieses Kommandofile übergeben werden können.

**cmdfile** [$arg_1$ ... $arg_n$ ]

- Diese Aufrufform ist die gebräuchlichste, Kommandofiles werden genauso behandelt wie „normale“ Kommandos. Auch hier wird eine Sub-Shell gestartet. Dies setzt voraus, daß das Kommandofile als „ausführbar“ gekennzeichnet wird (und gelesen werden darf), dies geschieht mit dem Programm **chmod**:

  chmod +x cmdfile

  Die Shell unterscheidet selbständig, ob ein Shell-Script auf diese Art oder ein übersetztes Programm (Maschinencode) gestartet wurde.

- Bemerkung: Man beachte die Konsequenzen, die sich aus der Abarbeitung von Shell-Scripts in Sub-Shells ergeben. Mit dem built-in-Kommando . (Punkt-Kommando) kann die Ausführung in der aktuellen Shell-Instanz gefordert werden, z.B.

```
. .profile
```

### 7.3 Kommentare

Kommentare in Shell-Scripts können mittels # eingeleitet oder über das „leere" Kommando (:) jeweils bis zum nächsten Newline gebildet werden:

- Beispiel: (siehe oben: akt_info)

```
#!/bin/sh
#
# Das ist das Shell-Script akt_info, welches anzeigt:
#
# 1. aktuelles Datum
# 2. angemeldete Nutzer
# 3. aktuelles Verzeichnis
#
/bin/echo -n "aktuelles Datum:"
date
echo  "angemeldet sind:"
who
/bin/echo -n "das aktuelle Verzeichnis ist:"
pwd
```

Alternativ: „leeres" Kommando:

```
: Das ist das Shell-Script akt_info, welches anzeigt:
: 1. aktuelles Datum
: 2. angemeldete Nutzer
: 3. aktuelles Verzeichnis
:
   ...
```

- Achtung: in diesem Fall führt Shell Textsubstitutionen durch!

Unterscheidung Bourne-Shell-Script ⇔ C-Shell-Script:

#!/bin/sh bzw. : ⇔ #!/bin/csh

z.B.: SUNOS (csh aktiv)

```
% cat B1
#!/bin/sh
ps | grep $0

83 % B1
787 p4 S     0:00 /bin/sh B1
788 p4 S     0:00 grep B1

84 % cat B2
:
ps | grep $0

85 % B2
791 p4 S     0:00 /bin/sh B2
792 p4 S     0:00 grep B2

86 % cat C1
#!/bin/csh
ps | grep $0

87 % C1
SUN-CSHRC
795 p4 S     0:00 /bin/csh C1
798 p4 S     0:00 grep C1

88 % cat K1
#!/bin/ksh
ps | grep $0

89 % K1
K1: Command not found.
90 %
```

z.B.: SUNOS (sh aktiv)

```
$ B1
804 p4 S     0:00 /bin/sh ./B1
805 p4 S     0:00 grep ./B1
$ B2
808 p4 S     0:00 grep B2
$ C1
SUN-CSHRC
810 p4 R     0:00 /bin/csh ./C1
813 p4 S     0:00 grep ./C1
$
```

z.B.: AIX (csh aktiv)

```
79 % cat B1
#!/bin/sh
ps | grep $0

80 % B1
24780  pts/2  0:00 grep B1
40907  pts/2  0:00 sh B1

81 % cat B2
:
ps | grep $0

82 % B2
23247  pts/2  0:00 /bin/sh B2
40912  pts/2  0:00 grep B2

83 % cat C1
#!/bin/csh
ps | grep $0

84 % C1
24787  pts/2  0:00 csh C1

85 % cat K1
#!/bin/ksh
ps | grep $0

86 % K1
24794  pts/2  0:00 ksh K1
40924  pts/2  0:00 grep K1

87 %
```

z.B.: AIX (sh aktiv)

```
$ B1
24799  pts/2  0:00 sh ./B1
40928  pts/2  0:00 grep ./B1
$ B2
24803  pts/2  0:00 grep B2
$ C1
40933  pts/2  0:00 csh ./C1
$ K1
```

```
24813  pts/2  0:00 grep ./K1
40939  pts/2  0:00 ksh ./K1
$
```

## 7.4 Argumentübergabe beim Aufruf eines Shell-Scripts

**cmdfile** $arg_1$ $arg_2$ ... $arg_n$

Die Bezugnahme auf die Argumentwerte **cmdfile** (entspricht $arg_0$ ), $arg_1$ , $arg_2$ ... $arg_n$ erfolgt in **cmdfile** selbst über die speziellen Shell-Variablen $0, $1, $2 ... $n
(die Namen der Shell-Variablen sind 0, 1 usw.)

- Beispiel:
  - Das ausführbare File **lsdir** enthalte die Zeile

    ```
    ls -l $1 | grep '^[Dd]'
    ```

  - So ergeben sich beispielweise folgende Aufrufmöglichkeiten:

    ```
    lsdir /usr     # Listen aller Verzeichnisse,
                   # die im Verzeichnis /usr enthalten sind

    lsdir .        # Listen aller Verzeichnisse im aktuellen Verzeichnis

    lsdir          # dto. ("leere Zeichenkette")
    ```

  - allgemein:

    ```
    lsdir [verzeichnis]
    ```

  - Bemerkung: Behandlung mehrerer Verzeichnisse kommt später

- Mit **$0** kann auf den Namen des Kommandofiles zugegriffen werden. Die Angabe **$*** umfaßt alle Argumente $arg_1$ bis $arg_n$ ; sie wird angewendet, wenn die genaue Anzahl der Argumente nicht bekannt ist bzw. wenn **n** den Wert 9 übersteigt.
  - Bemerkung: Hinweis auf **shift**-Kommando (später)
  - Beispiel:
    Aufgabe: Shell-Script, die den mehrspaltigen Druck von Files über folgenden
    **2 [file1 file2**...] ⇒ file1, file2 ... zweispaltig drucken
    **3 [file1 file2**...] ⇒ file1, file2 ... dreispaltig drucken
    Aufruf erlaubt.
    ...
    **6 [file1 file2**...] ⇒ Druck in 6 Spalten
  - Lösung: File mit dem Namen **2 (3,4,5,6)** enthält das Kommando

    ```
    pr -$0 -t -l1 $*
    ```

- Konstruktion dieses „Werkzeuges“:

```
$ cd /usr/bin
$ echo 'pr -$0 -t -l1 $*' >2
$ chmod 0755 2
$ ln 2 3; ln 2 4; ln 2 5; ln 2 6
```

- Anwendung:

```
$ ls | 4
kap01.tex     kap02.tex     kap03.tex     kap04.tex
kap05.tex     kap06.tex     kap07.tex     kap08.tex
kap09.tex     kap10.tex     abbildungen
```

• Mit der speziellen Shell-Variablen **$#** kann auf die Anzahl der beim Aufruf übergebenen Argumente ($arg_1$ bis $arg_n$ ) Bezug genommen werden

- Beispiel: Shell-Script **argecho**

```
echo -n  Es wurden $# Argumente uebergeben
echo : $*
```

Aufruf:

```
$ argecho rot gelb blau
Es wurden 3 Argumente uebergeben: rot gelb blau
$ argecho
Es wurden 0 Argumente uebergeben
```

# 8 Shell-Variablen

- Shell-Variablen speichern Zeichenketten. Namen von Shell-Variablen müssen mit einem Buchstaben beginnen, dem Buchstaben, Ziffern und Unterstreichungszeichen folgen können. Die Zuweisung eines Zeichenkettenwertes geschieht folgendermaßen

  $ *varname=zeichenkette*

  Beispiel:

  ```
  $ farbe=blau
  $ vorname=Hans
  ```

  Bemerkung:

  Variable werden durch ihre Verwendung (Nennung des Namens) definiert, sie besitzen standardmäßig den Wert der leeren Zeichenkette.

- Wenn die zuzuweisende Zeichenkette Leerzeichen, Tabulatorzeichen oder Newlinezeichen enthält, so muß die Zeichenkette **markiert** werden, z.B.

  ```
  $ name="HANS MEYER"
  $ name=HANS\ MEYER
  $ name='HANS MEYER'
  ```

  Bemerkung: Markierungsmechanismen der Shell werden noch genauer erläutert

- Eine Variable kann mit dem Attribut „Nur lesen" versehen werden, so daß ihr Wert nicht mehr verändert werden kann, z.B.

  ```
  $ einrichtung="Fachbereich Informatik"
  $ readonly einrichtung
  ```

- Shell-Variable besitzen einen definierten Gültigkeitsbereich, sie sind standardmäßig lokal bezüglich des aktuellen Shell-Prozesses. Eine Erweiterung des Gültigkeitsbereiches auf untergeordnete Shell-Instanzen (Kindprozesse) ist über das **export** - Kommando möglich, z.B.

  ```
  $ export einrichtung
  ```

  Bemerkung: export ohne Argument liefert eine Liste der exportierten Variablen

- Der Zugriff auf den aktuellen Wert einer Variablen geschieht dadurch, daß vor den Variablenname das Zeichen $ gesetzt wird, z.B.:

```
$einrichtung
$farbe
```

somit liefert

```
$ echo $einrichtung
Fachbereich Informatik
```

während **echo** für eine Variable, der noch kein Wert zugewiesen wurde, nichts liefert:

```
$ echo $xyz
$
```

Der Wert einer exportierten Variablen kann in einer Sub-Shell nur lokal verändert werden.
Beispiel:

```
$ trainer=HANS\ MEYER
$ echo $trainer
HANS MEYER
$ export trainer
$ sh
$ echo $trainer
HANS MEYER
$ trainer=UWE\ REINDERS
$ echo $trainer
UWE REINDERS
$ exit
$ echo $trainer
HANS MEYER
$
```

- Mit dem Kommando *unset* kann eine Variablendefinition rückgängig gemacht werden, z.B.

  ```
  $ unset trainer
  ```

- Wenn der Wert einer Variablen mit einer nicht dazugehörigen Zeichenfolge verkettet werden soll, so muß der Variablenname in geschweiften Klammern eingeschlossen werden, z.B.

  ```
  $ jahreszeit=Winter
  $ echo ${jahreszeit}ferien
  Winterferien
  ```

```
$ file=prog
$ echo ${file}.c
prog.c
```

- Spezielle Ersetzungsmechanismen für Shell-Variable

| | |
|---|---|
| ${*name*:-*wort*} | ersetzt *name* durch seinen Wert, falls dieser ungleich Null ist, ansonsten wird *wort* eingesetzt |
| ${*name*:=*wort*} | ersetzt *name* durch seinen Wert, falls dieser ungleich Null ist, andernfalls wird *wort* eingesetzt und *name* zugewiesen. |
| ${*name*:?*wort*} | ersetzt *name* durch seinen Wert, falls dieser ungleich Null ist, andernfalls wird *wort* auf der Standardfehlerausgabe ausgegeben und das Script abgebrochen. |
| ${*name*:+*wort*} | Setzt den Wert *wort* ein, falls *name* ungleich Null ist, ansonsten wird nichts ersetzt. |

- Shell-Variablen werden oft verwendet, um den Dialog am Terminal zu erleichtern, d.h. den „Tippaufwand" zu reduzieren, indem z.B. für Pfadnamen Synonyme eingeführt werden:

```
$ dir=/home/urz/fs1/wmc
$ ls $dir
$ ls $dir/*.c
```

Falls z.B. /home/urz/fs1/wmc/cmd ein Shell-Script enthält, so ist der Aufruf auch einfach möglich

```
$ $dir/cmd ...
```

- Folgende spezielle Shell-Variablen besitzen einen von der Shell vordefinierten Wert:

| | |
|---|---|
| # | Anzahl der Argumente, die dem Shell-Script übergeben wurden (bzw. Anzahl der mittels *set* gesetzten Shell-Variablen i (Werte $1, $2, ...) |
| * | Bezugnahme auf alle Argumentwerte $1, $2, etc. (z.B. beim Aufruf des Shell-Scripts) |
| @ | analog wie **$***; die Angabe „$@" bezieht sich auf alle Variablen „$1", „$2" etc. |
| 0 | Name des aktuell ausgeführten Shell-Scripts |
| - | Optionen dieser Shell |
| ? | Exitstatus (Return value) des letzten Kommandos |
| $ | Prozessnummer der Shell |
| ! | Prozessnummer des letzten Kommandos mit & |
| IFS | Liste von Worttrennzeichen in Argument |
| MAIL | File, das bei Änderung „you have mail"- Meldung bewirkt |
| PS1 | Primäre Prompt-Zeichenkette; Voreinstellung '$' |
| PS2 | Sekundäre Prompt-Zeichenkette; Voreinstellung '>' |

- Die folgenden speziellen Shell-Variablen sind nicht automatisch gesetzt; ihre Werte werden gewöhnlich beim Sitzungsbeginn festgelegt (die Wertzuweisungen sind dabei im File $HOME/.profile enthalten). Sie können jedoch wie „normale“ Variablen modifiziert werden.

| | |
|---|---|
| HOME | Standardargument für das **cd**- Kommando (HOME-Verzeichnis, für jeden Nutzer im Passwordfile /etc/passwd festgelegt) |
| PATH | Suchpfade für Kommandos, d.h. Reihenfolge der Verzeichnisse, in denen nach den Programmen bzw. Shell-Scripts gesucht werden soll. Die einzelnen Verzeichnisse sind durch das Zeichen : voneinander getrennt (p1:p2:p3:...:pn) |

- Beispiele:

```
$ echo $PATH
/bin:/usr/bin/:            #  1. /bin
                           #  2. /usr/bin
                           #  3. aktuelles Verzeichnis

$ PATH=:/bin:/usr/bin:$HOME/bin
                           #  Suchreihenfolge:
                           # 1. aktuelles Verzeichnis
                           # 2. /bin
                           # 3. /usr/bin
                           # 4. Verzeichnis bin im Homeverzeichnis

$ PATH=/bin::/usr/bin
                           # Suchreihenfolge:
                           # 1. /bin
                           # 2. aktuelles Verzeichnis
                           # 3. /usr/bin
```

- Bemerkung:
  - Privatversionen von „Systemprogrammen“
  - Suchreihenfolge ⇔ Antwortzeitverhalten

| | |
|---|---|
| PS1 | Festlegung der Promptzeichenkette der Shell, standardmäßig \$<br>Beispiel: PS1="Sie wünschen:" |
| PS2 | Promptzeichenkette bei Folgeeingaben, standardmäßig ><br>Beispiel: **\$echo** 'Hallo >wie geht's?' |
| IFS | Separatorzeichen der Shell, standardmäßig Leerzeichen, Tabulator- und Newlinezeichen |
| MAIL | Name des Nachrichtenfiles des Nutzers; er wird benachrichtet, wenn Nachrichten in dem File eingetroffen sind |

- Kommandosubstitution

**‘cmd‘** ⇒ wird ersetzt durch die Standardausgabe von **cmd**

– Beispiel:

```
$ date        # Uhrzeit, Datum (AIX)
Tue Apr 07 16:58:26 MEZ 1992
$

$ date        # Uhrzeit, Datum (SUNOS)
Tue Apr  7 17:03:14 MET DST 1992

$ echo Aktuelles Datum und Uhrzeit: ‘date‘
Aktuelles Datum und Uhrzeit: Tue Apr  7 17:03:14 MET DST 1992
$
$ pwd
/home/urz/fs1/wmc
$ dir=‘pwd‘
$ echo $dir
/home/urz/fs1/wmc
```

- Textsubstitution und Markierungsmöglichkeiten (quoting)
  - Die Shell führt Substitutionen in folgender Reihenfolge durch:
    1. Variablensubstitution

       alle mit $ beginnenden Worte werden ersetzt durch ihren Zeichenkettenwert

    2. Kommandosubstitution

       **‘cmd‘** ⇒ wird ersetzt durch Standardausgabe von **cmd**

    3. Leerzeichen-, Tabulator-, Newline-Interpretation

       Das nach 1. und 2. entstehende Ergebnis wird nach Trennzeichen durchsucht und die entsprechende Separierung durchgeführt (Ausnahmen: Trennzeichen in markierten Worten)

    4. Filenamengenerierung

       Auswertung der Metazeichen *****, **?**, **[-]**
  - Markierungsmechanismen (quoting) dienen dazu, die Sonderbedeutung von Metazeichen aufzuheben
    1. **z** hebt die (eventuelle) Sonderbedeutung von **z** auf
    2. In Apostrophe eingeschlossene Texte *(’. . . ’ )* werden keinerlei Textsubstitution unterworfen. Das Zeichen ’ darf in diesem Text nicht enthalten sein.

3. Bei in Anführungszeichen *("...")* eingeschlossenen Texten ist die Sonderbedeutung der Metazeichen außer \, ‘, “ und **$** aufgehoben, d.h. die Shell führt Kommando- und Variablensubstituion durch.

– Beispiel:

```
$ echo "heute ist ‘date‘; HOME-Verzeichnis: $HOME"
heute ist Tue Apr 07 17:17:38 MEZ 1992; HOME-Verzeichnis: /home/urz/fs1/wmc
$ echo "1 \$ = 1.67 DM"
1 $ = 1.67 DM
$
```

# 9 Ablaufsteuerung

- Entscheidungskriterium: Exit-Status (return value, Rückkehrcode) eines Programmes

  Konvention:

  falls Programm erfolgreich beendet $\Rightarrow$ Exit-Status gleich 0

  falls nicht erfolgreich $\Rightarrow$ Wert ungleich 0

- einfacher Bedingungstest

  $cmd_1$ && $cmd_2$ # $cmd_2$ nur ausgeführt, wenn $cmd_1$ Exit-Status = 0 liefert

  - Beispiel:

    ```
    cmp -s filea fileb && echo "Die Files sind identisch"
    ```

  $cmd_1$ || $cmd_2$ # $cmd_2$ nur ausgeführt, wenn $cmd_1$ Exit-Status ungleich 0 liefert

## 9.1 if-Kommando

- Beispiel:

  ```
  if cc -c sort.c
  then ld -o sort sort.o subr.o
  else echo Fehler beim Uebersetzen
  fi
  ```

  Falls der Exit-Status des **cc**-Kommandos gleich 0 ist, so wird **ld** ausgeführt, ansonsten **echo**.

- allgemeine Formen:

  **if** *cmd_liste_1*
  **then** *cmd_liste_2*
  [**else** *cmd_liste_3*]
  **fi**

  **if** *cmd_liste_1*
  **then** *cmd_liste_2*
  **elif** *cmd_liste_3*
  **then ...**
  **...**
  **fi**

- Erläuterung:

  - **if, then, else, elif, fi** sind Schlüsselworte, die müssen jeweils als erstes Wort einer Zeile auftreten (Ausnahme: nach **;** bzw. **&**)

  - *cmd_liste_1* ist eine Folge von ein oder mehreren Pipelines der allgemeinen Form

    $cmd_1 \ [| \ cmd_2 \ [| \ldots | \ cmd_n \ ]]$

    die durch **;** oder **,** oder **&** oder **&&** oder || getrennt sind, und wahlweise durch **;** oder **&** abgeschlossen werden

    Der Exit-Status einer Kommandoliste wird durch den Exit-Status der letzten Pipeline (d.h. im einfachsten Fall durch den Exit-Status des letzten Kommandos) bestimmt.
  - Beispiele für die Form der Kommandolisten:

    1. $cmd_1$
       $cmd_2$
       ...
       $cmd_n$

    2. $cmd_1 \ ;cmd_2 \ ;\ldots;cmd_n$

    3. $cmd$

    4. $cmd_1 \ | \ cmd_2$

    5. $cmd_1 \ | \ cmd_2$
       $cmd_3$

## 9.2 test-Kommando

Mit dem Kommando **test** kann der Exit-Status in Abhängigkeit vom Wahrheitswert eines Ausdruckes gezielt gesetzt werden.

Beispiel:

```
$ test -f sort.c      # Exit-Status gleich 0, falls File sort.c existiert
```

Alternative Syntax:

```
$ [ -f sort.c ]       # Exit-Status gleich 0, falls File sort.c existiert
```

Das Kommando **test** erlaubt neben der Untersuchung von Files und Filetypen außerdem numerische Tests und Zeichenkettenvergleiche:

- numerische Vergleiche
  - allgemeine Form:

    [ *N operator M* ]

    Wobei *N* und *M* für numerische Werte stehen, möglicherweise durch Shell-Variablen repräsentiert und *operator* für eine der folgenden Möglichkeiten steht:

    | | |
    |---|---|
    | **-eq** | Gleichheit von *N* und *M* |
    | **-ne** | Ungleichheit von *N* und *M* |
    | **-gt** | *N* ist größer als *M* |
    | **-lt** | *N* ist kleiner als *M* |
    | **-ge** | *N* ist größer oder gleich *M* |
    | **-le** | *N* ist kleiner oder gleich *M* |

  - Beispiele:

```
#!/bin/sh
#
# Pruefen, ob mehr als <anzahl> Nutzer zum
# Zeitpunkt des Aufrufs angemeldet sind
#
# Aufruf:  $0  <anzahl>
#
nutzer=`who | wc -l`
if [ $# -eq 0 ]
then   echo "Aufruf: $0 anzahl"
elif [ $nutzer -gt $1 ]
then echo "Mehr als $1 Nutzer angemeldet"
fi
```

```
#
# Varianten der Wertezuweisung an Variable
#

nummer=1
number=0001
numero=' 1'
```

Bei einem numerischen Test auf Gleichheit zwischen diesen Variablenwerten wird immer der Exit-Status 0 (d.h. Gleichheit) geliefert.

- Zeichenkettenvergleiche

  - allgemeine Formen

    [ $zk_1 = zk_2$ ]  # Test auf Gleichheit

    [ $zk_1$ != $zk_2$ ]  # Test auf Ungleichheit

    Beispiele:

    ```
    [ $nummer = $number ]      # Exit-Status ungleich 0!

    [ $nummer = $numero ]      # Exit-Status gleich 0 (!!) da durch
                               # test fuehrende Leerzeichen verloren gehen
    aber:

    [ "$nummer" = "$numero" ]  # Exit-Status ungleich 0
    ```

- Existenz einer Zeichenkette

  - allgemeine Formen:

    [ **-z** *zeichenkette* ]  # Test auf Länge 0 (leere Zeichenkette)

    [ **-n** *zeichenkette* ]  # Test auf Länge ungleich 0, d.h. Existenz

  - Beispiel:

    ```
    #!/bin/sh
    #
    # Aufruf: $0 <filename}>
    #
    if [ -z "$1" ]
    then echo "Aufruf: $0 filename"
    else
        ...
    fi
    ```

  - Bemerkung:

    ```
    if [ -z $1 ]
    ```

    führt bei fehlendem Argument zu Fehlernachricht:
    cmd_name: argument expected

- Untersuchung von Files

  - allgemeine Form:

    [ *operator filename* ]

    wobei *operator* steht für:

    | | |
    |---|---|
    | **-s** | File existiert und ist nicht leer |
    | **-f** | File ist normales File, kein Verzeichnis |
    | **-d** | File ist ein Verzeichnis |
    | **-w** | für das File besteht Schreiberlaubnis |
    | **-r** | für das File besteht Leseerlaubnis |
    | **-x** | File ist ausführbar |

  - Beispiel:

```
#!/bin/sh
#
# Loeschen eines Files bzw.
# eines leeren Verzeichnisses
#
# Aufruf  remove <filename>
#

if [ -d $1 ]
then rmdir $1
elif [ -f $1 ]
then rm $1
fi
```

  - Bemerkung:
    Die erläuterten Ausdrücke können darüber hinaus folgendermaßen zu komplexen Ausdrücken kombiniert werden:

    [ **!***ausdruck* ]  # Negation von *ausdruck*

    [ $ausdruck_1$ **-a** $ausdruck_2$ ]  # **logisches „UND“**

    [ $ausdruck_1$ **-o** $ausdruck_2$ ]  # **logisches „ODER“**

### 9.3 for-Kommando

- Beispiel:

```
#!/bin/sh
#
# lsdir (Version 1):
# Aufruf: lsdir1 <verz_1> [<verz_2> ... ]
#
for i
do   ls -l $i | grep '^[Dd]'
done
```

Für jedes Aufrufargument wird die Pipeline ausgeführt.

- **Problem:** bei Aufruf ohne Argument erfolgt keine Anwendung auf aktuelles Verzeichnis, da **for**-Körper überhaupt nicht abgearbeitet wird
- Lösung:

```
#!/bin/sh
#
# lsdir-Version:
# Aufruf: lsdir2 <verz_1> [<verz_2> ... ]
#
if [ $# -eq 0 ]
then
lsdir2 .
else
    for i
    do   ls -l $i | grep '^[Dd]'
    done
fi
```

- allgemeine Form des **for**-Kommando:

**for** *var* [**in** $zk_1$ $zk_2$ $\ldots zk_n$ ]
**do**
    *cmd_liste*
**done**

Der Shell-Variablen **var** wird in jedem Zyklus eine **Zeichenkette** $zk_i$ zugewiesen und damit **cmd_liste** abgearbeitet. (Dort wird natürlich auf **var** Bezug genommen.)

- Beispiele:

```
#!/bin/sh
#
# Erzeugen von Files
#
```

```
# Aufruf:  create <file_1> [<file_2> ...]
#

if [ $# -eq 0 ]
then echo "Aufruf: $0 file_1 [file_2 ...]"
else
     for i
     do
         >$i
     done
fi
```

```
#!/bin/sh
#
# lsdir-Version fuer das aktuelle Verzeichnis
#
# Aufruf: lsdir3
#

for i in *
do
     if [ -d $i ]
     then echo $i
     fi
done
```

- Bemerkung:
  - Metazeichen (z.B. *) erlaubt, d.h. es wird Filenamengenerierung durchgeführt
  - nur Verzeichnisnamen aufgelistet
- Beispiel:

```
#!/bin/sh
# Ausgabe von Informationen zu einem Nutzerkennzeichen
# in der Form:
# NUTZER-KZ: <nkz>   HOME-DIR: ...  LOGIN-SHELL: ...
#
# Aufruf: $0 <nkz_1> ...

if [ $# -eq 0 ]
then
    echo "Aufruf: $0 nutzerkennzeichen ..."
    exit 1;
fi
for i in $*
do
    if grep "^$i:" /etc/passwd >/tmp/nkz$$
```

```
    then
        h=`cat /tmp/nkz$$ | cut -f6 -d':'`
        p=`cat /tmp/nkz$$ | cut -f7 -d':'`
        echo "NUTZER-KZ:$i  HOME-DIR:$h  LOGIN-SHELL:$p"
    else echo "NUTZER-KZ $i auf dieser Maschine nicht vorhanden"
    fi
done
rm /tmp/nkz$$
```

## 9.4 while-Kommando

- Beispiel:

```
#!/bin/sh
#
# Warten, bis <file> nicht mehr existiert
#
# Aufruf: $0 <file>
#

if [ $# != 1 ]
then
    echo "Aufruf: warte <filename>"
    exit 1
fi
while [ -r $1 ]
do sleep 5
done
```

- allgemeine Form der **while**-Konstruktion:

  **while** *cmd_liste_1*

  **do**

  *cmd_liste_2*

  **done**

  Solange der Exit-Status von *cmd_liste_1* gleich 0 ist, wird die *cmd_liste_2* abgearbeitet.

- Beispiel:

```
#!/bin/sh
#
#  Statistik der angemeldeten Nutzer im File <file>
#
# Aufruf: nustat <file>
```

```
#
if [ $# = 0 ]
then echo "Aufruf: nustat <filename>"
else
     while sleep 300
     do
           date >> $1
           who >> $1
     done
fi
```

- Bemerkungen:

  Ausführung im Hintergrund sinnvoll:

  $ *nustat statistik* &

- Programm bricht nicht von selbst ab, d.h. expliziter Abbruch (**kill**-Kommando bzw. CRTL-C im Vordergrund) notwendig

## 9.5 shift-Kommando

Oft wird **while** im Zusammenhang mit **shift** benutzt; **shift** verschiebt die Aufrufargumente, so daß **$1** den ursprünglichen Wert von **$2** erhält, **$2** den von **$3** usw. der Wert von **$#** wird demzufolge um 1 verringert.

- Beispiel:

```
#!/bin/sh
#
# alternative lsdir-Version (Version 4)
#
# Aufruf: lsdir4 [ verzeichnis ]
#

if [ $# -eq 0 ]
then lsdir4 .
else
    while [ $# -gt 0 ]
    do
        ls -l $1 | grep '^[dD]'
        shift
    done
fi
```

### 9.6 until-Kommando

- Beispiel:

```
#!/bin/sh
#
# Warten, bis Nutzer user_id angemeldet ist
#
# Aufruf: waitfor user_id
#
if [ $# != 1 ]
then
    echo "Aufruf: waitfor user_id"
    exit 1
else
    until who | grep $1
    do
        sleep 60
    done
fi
```

- Bemerkung: **grep** liefert Exit-Status gleich Null, falls Muster existiert

- Aufruf: **warte gsf&** ⇒ Hintergrundprozeß

- allgemeine Form:

**until** *cmd_liste_1*
**do**
    *cmd_liste_2*
**done**

Solange der Exit-Status von *cmd_liste_1* ungleich 0 ist, wird *cmd_liste_2* abgearbeitet.

### 9.7 case-Kommando

- Beispiel:

```
#!/bin/sh
#
# Shell-Script append, das ein File
# an ein anderes File "anhaengt"
#
# Aufruf: append [von_file] nach_file
```

```
#
case $# in
    1) cat >>$1;;
    2) cat >>$2 <$1;;
    *) echo "Aufruf: append [von] nach";;
esac
```

- allgemeine Form:

  **case** *zk* **in**

  *zk_muster_1*) **cmd_liste_1;;**

  *zk_muster_2*) **cmd_liste_2;;**

  ...

  *zk_muster_n*) **cmd_liste_n;;**

  **esac**

- Bemerkung:

  Zur Bildung der Zeichenkettenmuster *zk_muster_i* können die „pattern matching"-Möglichkeiten der Shell (*****, **?**, **[-]**) benutzt werden sowie mehrere Alternativen, getrennt durch | angegeben werden:

  *zk_muster_1* | *zk_muster_2* | ...)

- Beispiel:

```
#!/bin/sh
#
# komfortablere Schnittstelle fuer /usr/bin/cal
#
# Aufruf:    cal  [ <monatsname>|<monatszahl> [ <jahr> ]]
#
case $# in
0)      set `date`; m=$2; y=$7 ;;   # keine Argumente: heute
1)      m=$1; set `date`; y=$7 ;;   # 1 Arg:  Monat im aktuellen Jahr
2)      m=$1; y=$2 ;;               # 2 Arg: Monat und Jahr
*)      echo "Aufruf: a) $0 [ <monatsname>|<monatszahl> [ <jahr> ]]"
        echo "        b) $0  <jahr> "
        echo ""
        echo "Bsp: $0 5 1990   --> Mai 1990"
        echo "     $0 5        --> Mai des aktuellen Jahres"
        echo "     $0 05       --> Jahr 5 ,d.h fuer die Jahre 1-12 muss die"
        echo "                     Ziffer 0 vor die Jahreszahl gesetzt werden"
        exit 1;;
esac
case $m in
    ja*|Jan*)        m=1 ;;
    fe*|Feb*)        m=2 ;;
    ma[er]*|Ma[er]*) m=3 ;;
    ap*|Apr*)        m=4 ;;
    ma[yi]|Ma[yi])   m=5 ;;
    jun*|Jun*)       m=6 ;;
    jul*|Jul*)       m=7 ;;
    au*|Aug*)        m=8 ;;
    se*|Sep*)        m=9 ;;
    o[ck]*|O[ck]*)   m=10 ;;
    no*|Nov*)        m=11 ;;
    de*|De*)         m=12 ;;
    [1-9]|10|11|12)       ;;          # Monatsnummer
    *)               y=$m;m="";;      # erstes Argument als Jahreszahl
esac
if /usr/bin/cal $m $y
then :
else
    echo "Aufruf: a) $0 [ <monatsname>|<monatszahl> [ <jahr> ]]"
    echo "        b) $0  <jahr> "
    echo ""
    echo "Bsp: $0 5 1990   --> Mai 1990"
    echo "     $0 5        --> Mai des aktuellen Jahres"
    echo "     $0 05       --> Jahr 5 ,d.h fuer die Jahre 1-12 muss die"
    echo "                     Ziffer 0 vor die Jahreszahl gesetzt werden"
    exit 1
fi
```

## 9.8 break-Kommando

- Funktion: Abbruch der umgebenden **for**- oder **while**- Konstruktion bzw. Verlassen um $n$ Stufen

- Syntax: **break** [$n$]

## 9.9 continue-Kommando

- Funktion: Einleiten der nächsten Iteration von **while** bzw. **for**; die wahlweise Angabe $n$ gibt an, daß in der $n$-ten umhüllenden Schleife fortgesetzt werden soll.

- Syntax: **continue** [$n$]

## 9.10 exit-Kommando

Das **exit**-Kommando wird in Shell-Scripts verwendet, um deren Ausführung abzubrechen.

- Beispiel: cmd_name  argument

  ```
  if test $# -ne 1
  then
      echo "Aufruf: $0 argument"
      exit 1      # Exit-Status 1
  fi
  ```

- allgemein:

  **exit** [$n$]

  wobei $n$ den Exit-Status angibt, der geliefert werden soll. Wird diese Angabe weggelassen, so wird der Exit-Status des unmittelbar vorher ausgeführten Kommandos zurückgegeben.

## 9.11 Funktionen

- Definition

  *func_name* () {        *cmd_liste* }

  bzw.

  *func_name* () { *cmd_liste*; }

- Aufruf

  *func_name* [[ *arg_1* [ *arg_2* ... ]]

  Die Werte von $arg\_1$, $arg\_2$ usw. werden den Shell-Variablen 1, 2 usw. zugewiesen und können damit über $1, $2, usw. abgerufen werden.

- Rückkehr aus der Funktion
  - bei Erreichen des letzten Kommandos
  - über das Kommando

    *return* [ *n*]

    Dabei liefert $n$ den Exit-Status der Funktion. Falls $n$ weggelassen wird, ergibt sich der Exit-Status der Funktion aus dem Exit-Status des zuletzt ausgeführten Kommandos.

- Bemerkungen
  - Funktionen existieren nur in der Shell, in der sie definiert wurden. Demzufolge bleiben Veränderungen (z.B. durch Argumentübergabe an die Funktion, d.h. Setzen der Variablen 1 usw. nach Beendigung der Funktion erhalten.
  - Funktionsdefinitionen können in *.profile* gespeichert werden bzw. in einen File *funcdef_file*, welches später (oder in *.profile*) durch

    . *funcdef_file*

    ausgeführt wird. Auf diese Weise können Funktionen auch in Sub-Shells verfügbar gemacht werden (z.B. in einen Shell-Script).
  - Der Aufruf von Funktionen ist (interaktiv) am Terminal möglich.
  - Funktionen sind schneller als Shell-Scripts, da sie intern gespeichert werden.

- Beispiel

```
#
# mcd <verzeichnis>  --> Wechseln in <verzeichnis>
#
# mcd -              --> ins vorherige Verzeichnis
#
mcd ()
{
    AKTDIR=`pwd`
    if [ "$1" = "-" ]
    then
        /bin/echo -n $ALTDIR --
        cd $ALTDIR;
```

```
    else
        /bin/echo -n $1 --
        cd "$1";
    fi
    ALTDIR=$AKTDIR
}
```

# 10 Spezielle Probleme und Kommandos

## 10.1 login-Profile

Für jeden Nutzer muß ein Eintrag im Passwortfile **/etc/passwd** existieren, in dem enthalten sind:

- ein Nutzerkennzeichen
- ein (verschlüsseltes) Passwort
- Nutzernummer (uid)
- Gruppennummer (gid)
- Kommentarfeld (Name, Institution)
- das HOME-Verzeichnis
- ein Programmname (normalerweise **/bin/sh** oder **/bin/csh**)

Beim Anmelden werden diese Informationen geprüft bzw. benutzt. Das im Passwortfile angegebene Programm (also die Shell) wird gestartet. Die Bourne-Shell durchsucht dabei als erstes das HOME-Verzeichnis nach einem File mit dem Namen **.profile**. Dieses File enthält Kommandos, die zu Sitzungsbeginn standardmäßig abgearbeitet werden sollen.

- Beispiel:

```
date
stty erase '^H' kill '^U'
PATH=:/bin:/usr/bin:$HOME/bin
PS1='READY'
echo Viel Erfolg!
```

- Erläuterung zu **stty**-Elementen

```
erase '^H'     # Zeichen loeschen durch <Control H>
kill '^U'      # Zeilen loeschen durch <Control U>
```

## 10.2 Here-Dokumente

Unter einem „here“-Dokument versteht man das temporäre Umlenken der Standardeingabe innerhalb eines Shell-Scripts.

- Notation

  **cmd_name** $arg_1$ … $arg_n$ $<<$*zeichenkette*<br>
  … |<br>
  … $>$ „here“-Dokument<br>
  … |<br>
  *zeichenkette*

- Erläuterung

  $<<$*zeichenkette* bedeutet, daß alle nachfolgenden Zeilen bis zu einer Zeile, auf der nur **zeichenkette** vorkommt, als Standardeingabe für **cmd_name** betrachtet werden sollen.

- Beispiel: Funktion **subst** zur Zeichenkettensubstitution: **subst** *zk_alt zk_neu file*

```
# globales Ersetzen von <zk_alt> durch <zk_neu>
# in <file>
#
# subst <zk_alt> <zk_neu> <file>
#
subst ()
{
/bin/ed $3 <<%
g/$1/s//$2/g
w
%
}
```

- Bemerkung:
  - keine Prüfung der Argumente im Beispiel
  - innerhalb eines „here“-Dokuments wird die Kommando- und Variablensubstitution durchgeführt; wenn dies verhindert werden soll, so muß nach den Zeichen $<<$ sofort ein Backslash (\ ) folgen:

    $<<$ *zeichenkette*
  - Oft werden „here“-Dokumente angewendet, um längere Nachrichten aus eines Shell-Scripts auszugeben.

### 10.3 set-Kommando

Das built in Kommando **set** dient verschiedenen Zwecken:

- Anzeigen der aktuellen Werte aller Shell-Variablen (außer **$**, **?**, **!**, **#**, **-**) in der Form *varname=wert* durch

```
$ set
DISPLAY=:0
FONTPATH=/usr/openwin/lib/fonts
FRAMEBUFFER=/dev/fb
HELPPATH=/usr/openwin/lib/help
HOME=/home/urz/fs1/wmc
IFS=

LD_LIBRARY_PATH=/usr/openwin/lib
LINES=24
LOGNAME=wmc
MAILCHECK=600
MANPATH=/usr/openwin/share/man:/usr/global/man:/usr/local/man:/usr/man
NEWSSERVER=2255341579.2000;hannibal
OPENWINHOME=/usr/openwin
OPTIND=1
PATH=.:/home/urz/fs1/wmc:/home/urz/fs1/wmc/bin:/usr/openwin/bin/xview:/usr/openwin/bin:/us
PS1=$
PS2=>
PWD=/home/urz/fs1/wmc
SHELL=/bin/csh
TERM=sun-cmd
TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
USER=wmc
WINDOW_PARENT=/dev/win0
WMGR_ENV_PLACEHOLDER=/dev/win3
XAPPLRESDIR=/usr/openwin/lib/X11/app-defaults
$
```

- Setzen der Shell-Variablen **1**, **2**, ... innerhalb eines Shell-Scripts durch

```
set arg_1 arg_2 ...
```

- Beispiel:

```
#!/bin/sh
set `date`   # weekday  mon dd hh:mm:ss mon timezone daylight year

case $1 in
```

```
                0[6-9]:??:??)  echo Guten Morgen!;;
                1[0-7]:??:??)  echo Guten Tag!;;
  1[8-9]:??:??|2[0-1]:??:??)  echo Guten Abend!;;
                          *)  echo Um diese Zeit schlaeft ein normaler Mensch!
                              echo Gute Nacht!;;
esac
```

- Setzen spezieller Shell-Optionen (Auswahl)

```
set -n # Nur Lesen der Kommandos, keine Ausfuehrung (no execution)
set -v # Ausgabe aller Kommandos am Terminal vor der Ausfuehrung (verbose)
set -x # Ausgabe der Kommandos plus Argumente
       # sowie der Variablenwerte nach jeder Kommandoausfuehrung
set -  # Ausschalten -x und -v
```

- Bemerkungen:
  - Kombination der Optionen möglich
  - Debugging von Shell-Scripts auf diese Weise unterstützt
  - Variable - enthält die aktuellen Shell-Optionen

### 10.4 Abfangen von Signalen

Signal: Ereignis, das zum Abbruch der Programmausführung führt, (wenn keine Behandlung vorgesehen ist).

- **Signale**(Auswahl) **Ursache**

| | |
|---|---|
| 0 | Shell-Exit (EOF) |
| 1 | Abhängen der Terminalstation (hangup) |
| 2 | Unterbrechung (DEL-Taste) |
| 3 | QUIT: C |
| 9 | KILL: |
| 15 | Software-Abbruch (**kill**-Kommando) |

- Problem:

  Bei Ausführung eines Shell-Scripts tritt ein Signal auf, d.h. die Ausführung wird abgebrochen. Es sollen jedoch unbedingt noch bestimmte Aktionen durchgeführt werden, z.B. temporär verwendete Files gelöscht werden.

- Lösung: **trap**-Kommando

  **trap** *cmd_liste signal_liste*

  Nach Auftreten eines Signals wird *cmd_liste* ausgeführt und anschließend an der unterbrochenen Stelle fortgesetzt (es sei denn, *cmd_liste* enthält **exit**-Kommando)

- Beispiele:

```
trap 'rm -f temp*;exit' 1 2 3 15

  ...   oder

trap "" 1 2 3 15      # Ignorieren der Signale
```

- Bemerkung:

  Da *cmd_liste* zweimal von der Shell interpretiert wird, sollte *'cmd_liste'* geschrieben werden, damit nur einmal (bei der Ausführung) eine Variablenauswertung erfolgt.

### 10.5 eval-Kommando

Die Argumente des **eval**-Kommandos werden als Kommando betrachtet und ausgeführt.

- Notation:

  **eval** $arg_1$ ... $arg_n$

- Beispiel:

```
#!/bin/sh
#
# Kopieren von Filebaeumen in ein Zielverzeichnis
# Aufruf: copytree verzeichnis1 [verzeichnis2 ... ] ziel_verzeichnis
#
eval 'last=$'$#
while [ $# -gt 1 ]
do
     echo "copy from $1 to $last"
     (cd $1; tar cf - .) | (cd $last; tar xf -)
     shift
done
```

- Bemerkung:
  - **eval** notwendig, um zweistufige Auswertung zu erzwingen:
    1. $# wird ersetzt durch Argumentzahl
    2. Zuweisung des Zeichenkettenwertes des letzten Argumentes an die Variable **last**!!
  - Das Kommando **tar -xf - .** bewirkt, daß von der Standardeingabe (-), d.h. der Pipe gelesen und auf das aktuelle Verzeichnis geschrieben wird.
  - Die Konsequenz dieser Pipeline besteht darin, daß alle Files des Ausgangsverzeichnisses kopiert werden. Das Starten zweier Sub-Shells ist notwendig, um mit „zwei“ aktuellen Verzeichnissen arbeiten zu können.

## 10.6 read-Kommando

Bewirkt das Lesen einer Zeile von der Standardeingabe sowie das Zuweisen der einzelnen Werte an die im **read**-Kommando angegebene Shell-Variablen. Die letzte Variable enthält u.U. den Rest der Zeile.

- Beispiel: scan ⇒ interaktive Kommandoverarbeitung

```
#!/bin/sh
#
# interaktive Kommandoverarbeitung im Filebaum
# ab aktuellem Verzeichnis
#
# Aufruf: scan
#
dir=`pwd`
for i in *
do
    if test -d $dir/$i
    then
cd $dir/$i
        while echo "$i"
            trap exit 2
            read x
        do
            trap : 2
            eval $x
        done
    fi
done
```

## 10.7 Auswertung von Ausdrücken mittels expr

Das Kommando **expr** betrachtet seine Argumente als Ausdrücke, die ausgewertet werden, sowie das Ergebnis auf die Standardausgabe geschrieben wird.

- allgemeine Form:

  **expr** $ausdruck_1$ *operator* $ausdruck_2$

  wobei:

| *operator* | Ergebnis |
|---|---|
| $\vert$ | $a_1$ , falls ungleich 0, sonst $a_2$ |
| & | $a_1$ , falls $a_1$ und $a_2$ ungleich 0 bzw. “”, sonst 0 |
| $<, <=, =, != , >=, >$ | “1”, falls Relation wahr, sonst “0” |
| + | *Addition* von $a_1$ und $a_2$ |
| - | *Subtraktion* $a_2$ von $a_1$ |
| * | *Multiplikation* $a_1$ mit $a_2$ |
| / | *Division* $a_1$ durch $a_2$ |
| % | $a_1$ modulo $a_2$ |
| : | $a_1$ wird mit $a_2$ verglichen, $a_2$ darf ein regulärer Ausdruck sein |

- Bemerkung:
  - Vorrang ist steigend
  - Klammerung von Ausdrücken ist möglich: *(a)*
  - Achtung: * muß auf Grund seiner Sonderbedeutung für die Shell markiert werden, z.B.
- Beispiele:

```
expr 14 + 23+ 112
149

count=`expr $count + 1`    #  Wert von count um 1 erhoehen
```

# 11 Beispiele für Shell-Programmierung

## 11.1 Kopieren von Teilbäumen im Filesystem

```
#!/bin/sh
#
# das ist ein Shell-Skript zum Kopieren von Teilbaeumen des Filesystems
#
# Aufruf: copybaum <von_dir> <nach_dir>
#

if test $# -ne 2
    then
    echo zuwenig oder zuviel argumente
    echo 'Aufruf: copybaum <von_dir> <nach_dir>'
    exit
fi
if test ! -r $1
    then
    echo $1 ist nicht lesbar
    echo 'Aufruf: copybaum <von_dir> <nach_dir>'
    exit
fi
if test ! -d $1
    then
    echo $1 ist kein verzeichnis
    echo 'Aufruf: copybaum <von_dir> <nach_dir>'
    exit
fi
if test `ls $1 | wc -l` -eq 0
    then
    echo $1 ist ein leeres verzeichnis
    echo 'Aufruf: copybaum <von_dir> <nach_dir>'
    exit
fi
if test -r $2
    then
    if test ! -d $2
        then
        echo $2 existiert ist aber kein Verzeichnis
        echo 'Aufruf: copybaum <von_dir> <nach_dir>'
        exit
    fi
    if test ! -w $2
        then
        echo fuer $2 existieren keine schreibrechte
```

```
        echo 'Aufruf: copybaum <von_dir> <nach_dir>'
        exit
    fi
else
    mkdir $2
    if test $? -ne 0
        then
        echo das verzeichnis $2 kann nicht erzeugt werden
        echo 'Aufruf: copybaum <von_dir> <nach_dir>'
        exit
    fi
fi
(cd $1; tar cf - .) | (cd $2; tar xf -)
```

### 11.2 Selektives Kopieren

- Variante 1 (selcopy1)

```
#!/bin/sh
#
# Selektives Kopieren (Version 1)
#
# Aufruf: selcopy1 <quellverz> <zielverz>
#
for i in `ls $1/*`
do
    echo "$i kopieren?"
    read antwort
    if [ "$antwort" = 'j' ]
    then
        cp $1/$i $2
        echo "File $i kopiert"
    else
        echo "File nicht kopiert"
    fi
done
```

- Variante 2 (selcopy2)

```
#!/bin/sh
#
# Selektives Kopieren (Version 2)
#
```

```
# selcopy <quellverz> <zielverz>
#
if [ -d $1 ]
then
    if [ -d $2 ]
    then
        for i in `ls $1/*`
        do
            echo "$i kopieren?"
            read antwort
            if [ "$antwort" = 'j' ]
            then
                cp $1/$i $2
                echo "File $i kopiert"
            else
                echo "File nicht kopiert"
            fi
        done
    else
        echo "$2 ist kein Verzeichnis"
    fi
else
    echo "$1 ist kein Verzeichnis"
fi
```

- Variante 3 (selcopy3)

```
#!/bin/sh
#
# Selektives Kopieren (Version 3)
#
# selcopy <quellverz> <zielverz>
#
if [ $# -ne 2 ]
then
    echo "usage: selcopy_3 <quellverz> <zielverz>"
    exit
fi
if [ -d $1 ]
then
    if [ -d $2 ]
    then
        cd $1
        for i in `ls $1`
        do
            echo "$i kopieren?"
```

```
            read antwort
            if [ "$antwort" = 'j' ]
            then
                if cp $1/$i $2
                then
                    echo "File $i kopiert"
                fi
            else
                echo "File nicht kopiert"
            fi
        done
    else
        echo "$2 ist kein Verzeichnis"
    fi
else
    echo "$1 ist kein Verzeichnis"
fi
```

### 11.3 Varianten des Sieb-Filters

- Schreiben Sie einen Filter namens sieb, der nur jede zweite Zeile von stdin –¿ stdout kopiert.

```
#!/bin/sh
#
# Filter sieb (variante 0)
#
# Aufruf: sieb0
#
a=1
while read n
do
    a=`expr $a '*' -1`
    if test $a -gt 0
    then
        echo $n
    fi
done
```

- Problem: In den durchgelassenen Zeilen gehen Leerzeichen verloren, wenn mehrere zusammenhängende Leerzeichen auftreten (read-Kommando)
- Diese Filter ist zu verallgemeinern. Beim Aufruf sieba +n soll jede n-te Zeile kopiert werden. Beim Aufruf sieba -n soll jede n-te Zeile ignoriert werden. Für n sollen die Werte 2 bis 9 zugelassen sein; dies ist zu überprüfen!

```
#!/bin/sh
#
# Filter sieba
#
# Aufruf: sieba +n    kopieren jeder n-ten Zeile
#         sieba -n    ignorieren jeder n-ten Zeile
#

if test $# -ne 1
then
    echo "Benutze:   sieba +n  oder  sieba -n"
    echo "1 < n < 10"
    exit 1
fi
x=`echo "$1" | cut -c1`
case $x in
    '-' ) : ;;
    '+' ) : ;;
    *   ) echo "Benutze:   sieba +n  oder  sieba -n"
          echo "1 < n < 10"
          exit 1
          ;;
esac
x=`echo "$1" | cut -c2`
case $x in
    [2-9] ) : ;;
    *   ) echo "Benutze:   sieba +n  oder  sieba -n"
          echo "1 < n < 10"
          exit 1
          ;;
esac

a=`echo "$1" | cut -c2`
if test "$1" -gt 0
then
    d=1
else
    d=0
fi
r=$a
while read n
do
    a=`expr $a '-' 1`
    if test $d -eq 0
    then
        if test $a -eq 0
```

```
            then
                a=$r
                continue
            else
                echo $n
                continue
            fi
        fi
        if test $a -eq 0
        then
            echo $n
            a=$r
        fi
    done
```

## 11.4 Primitive Turbo-C-Variante

```
#!/bin/sh
#
# "Turbo-C"
#
# Aufruf: turboc <prog_name>
#
if [ $# -ne 1 ]
then
    echo "usage: turbo <prog_name>"
    exit
fi
while true
do
    vi $1.c
    if cc -o $1 $1.c
    then
        ./$1
        while true
        do
            echo -n "AGAIN? [y/n]:"
            read ans
            case $ans in
                y|Y)    continue 2;;
                n|N)    break 2;;
                *)        continue;;
            esac
        done
    else
```

```
        echo -n ENTER RETURN TO CONTINUE:
        read ans
        continue
    fi
done
```

### 11.5 Anzeigen von Prozeß-Hierarchien

- Schreiben Sie ein Script, das alle Vorfahren eines Prozesses anzeigt. Die Nummer des Prozesses wird dem Script als Argument übergeben. Für jeden Prozeß sind folgende Angaben anzuzeigen: PID, PPID, UID, TTY und das vollständige Kommando.

```
#!/bin/sh
#
# Prozessangaben anzeigen
#
# Aufruf: proz <prozess_id>
#
ps -elf >/tmp/pro$$
a=$1
if test $a -lt 0
then
    a=`expr $a '*' -1`
else
    echo "PID    PPID    UID    TTY    CMD"
fi
trap "rm /tmp/pro$$; exit" 2
(
while read n
do
    x=`echo $n | cut -f4 -d' '`
    if test $x -eq $a
    then
    # die Feldnummern (fuer cut -f) sind vom Listenformat der
    # ps-Implementierung abhaengig
        p=`echo $n | cut -f4 -d' '`
        pp=`echo $n | cut -f5 -d' '`
        u=`echo $n | cut -f3 -d' '`
        t=`echo $n | cut -f12 -d' '`
        c=`echo $n | cut -f14- -d' '`
        echo "$p    $pp    $u    $t    $c"
        if test $pp -ne 0
        then
            pro -$pp
        fi
```

```
        break
    fi
done
) </tmp/pro$$
rm /tmp/pro$$
```