Programmierkurs für absolute Anfänger

↳ Unix-Befehle und bash-Skripte

Caren Brinckmann
Sommersemester 2005

http://www.coli.uni-saarland.de/~cabr/teaching.php

Unix-Shells

- Shell:
  - Schnittstelle zwischen Benutzer und Betriebssystem: Die Aktionen des Benutzers werden via Shell an den Betriebssystemkern (*Kernel*) weitergeleitet.
  - Befehlszeileninterpreter, der Eingabezeilen (interaktiv oder aus einem Shell-Skript) auswertet und die entsprechenden Befehle ausführt.
- Unix-Shell-Varianten:
  - sh: "Bourne Shell", klassische Shell von Unix-Systemen
  - csh: "C-Shell", Einsatz von C-ähnlichen Konstrukten bei der Programmierung
  - tcsh: "Tenex C-Shell", Weiterentwicklung der C-Shell
  - ksh: "Korn Shell", verbesserte Version der Bourne-Shell (History-Mechanismus)
  - bash: "Bourne Again Shell", Standard-Shell unter Linux, Kombination der Features der sh, (t)csh und ksh + weitere Features

bash: Grundlagen

- Starten der bash: Terminal-Fenster aufmachen (falls bash nicht die Standard-Shell ist: **`bash`**)
- Version: **`Strg-x Strg-v`** (aktuell: Version 3)
- Prompt (Zeichen am Anfang der Zeile): **`$`**
- Nach dem Prompt kann der Benutzer Befehle eingeben, die dann die Shell durch entsprechende Systemaufrufe ausführt:
  `$ Befehl Optionen Argumente`
  z.B.: **`$ perl -w meinprogramm.pl`**
- Die Online-Hilfe zu jedem Befehl erhält man mit **`man`**, z.B.:
  `$ man perl`
- Eine Folge von Befehlen kann auch in eine Datei geschrieben werden, ein so genanntes Shell-Skript. Beim Aufruf eines Skripts arbeitet die Shell die Befehle Schritt für Schritt ab.
- Ein bash-Skript kann folgendermaßen aufgerufen werden:
  `$ bash meinskript`

mehrere Befehle in einer Zeile

- mehrere Befehle sollen nacheinander ausgeführt werden:
  `$ ersterBefehl ; zweiterBefehl`
- der zweite Befehl soll nur dann ausgeführt werden, wenn der erste Befehl fehlerfrei ausgeführt werden konnte:
  `$ ersterBefehl && zweiterBefehl`
- der zweite Befehl soll nur dann ausgeführt werden, wenn der erste Befehl nicht fehlerfrei ausgeführt werden konnte:
  `$ ersterBefehl || zweiterBefehl`
- Pipeline: die Ausgabe des ersten Befehls wird direkt in den Eingabekanal des zweiten Befehls geleitet:
  `$ ersterBefehl | zweiterBefehl`

Ein- und Ausgaben umlenken

- Standardeingabe aus Datei beziehen:
  `$ Befehl < Datei`
- Standardausgabe in Datei umlenken (die Fehlermeldungen erscheinen weiterhin auf dem Bildschirm):
  `$ Befehl > Datei`
- Standardfehlerausgabe in Datei umlenken:
  `$ Befehl 2> Datei`
- Standardausgabe und Standardfehlerausgabe in dieselbe Datei umlenken:
  `$ Befehl &> Datei`
- Standardausgabe und Standardfehlerausgabe in verschiedene Dateien umlenken:
  `$ Befehl > Ausgabedatei 2> Fehlerdatei`
- Anhängen statt Überschreiben: **`>>`**

Befehle: Dateien und Verzeichnisse (1)

- **`pwd`** (print working directory): aktuelles Verzeichnis anzeigen
- **`ls`** (list contents of directory): Dateinamen auflisten
- **`mkdir`** (make directory): neues Verzeichnis anlegen
  `$ mkdir Programmierkurs`
- **`cd`** (change directory): aktuelles Verzeichnis wechseln
  `$ cd Programmierkurs`
  `$ cd ..`
  `$ cd –`
- **`cp`** (copy files): Dateien kopieren
  `$ cp meinprogramm.pl kopie.pl`
  `$ cp meinprogramm.pl Programmierkurs/`
- **`rm`** (remove files): Dateien und Verzeichnisse löschen
  `$ rm test.txt`
  `$ rm -r Testverzeichnis`

Befehle: Dateien und Verzeichnisse (2)

- **mv** (move files): Dateien und Verzeichnisse verschieben oder umbenennen
  ```
  $ mv meinprogramm.pl deinprogramm.pl
  $ mv deinprogramm.pl Programmierkurs/
  $ mv Programmierkurs Perlkurs
  ```
- **basename**: eigentlichen Dateinamen aus einem Pfadnamen extrahieren
  ```
  $ basename /home/CE/cabr/test.txt
  test.txt
  $ basename /home/CE/cabr/test.txt .txt
  test
  ```
- **dirname**: Verzeichnispfad aus einem Pfadnamen extrahieren
  ```
  $ dirname /home/CE/cabr/test.txt
  /home/CE/cabr
  ```


Befehle: Dateien (1)

- **cat** (concatenate): Dateieninhalt auf die Standardausgabe ausgeben
  ```
  $ cat text1.txt text2.txt > text1und2.txt
  ```
- **cmp** (compare): zwei Dateien vergleichen
  ```
  $ cmp text1.txt text2.txt
  ```
  - Die Dateien dürfen auch Binärdateien sein.
  - Wenn die beiden Dateien identisch sind, wird nichts ausgegeben.
- **diff** (differential file compare): zwei Textdateien vergleichen
  ```
  $ diff text1.txt text2.txt
  ```
  - Es wird ausgegeben, in welchen Zeilen die Dateien sich unterscheiden.
- **wc** (word count): Zeilen, Wörter und Zeichen eines Textes zählen
  ```
  $ wc text1.txt
  ```
  - Optionen:
    - `-l` nur die Zeilen zählen (lines)
    - `-w` nur die Wörter zählen (words)
    - `-c` nur die Zeichen zählen (characters)

Befehle: Dateien (2)

- **sox**: Sounddateien in andere Formate konvertieren
  ```
  $ sox test.wav test.raw
  $ sox -r 16000 -w -s kkos025.raw kkos025.wav
  ```
- (e)**grep**: Suchen in Dateien mit regulären Ausdrücken
  ```
  $ grep 'M[ea][iy]er' namen.txt
  ```
  gibt alle Zeilen aus, die der reguläre Ausdruck matcht
- **sort**: sortieren von Dateien
- **uniq**: aufeinanderfolgende identische Zeilen nur einmal ausgeben
- **tr**: bestimmte Zeichen eines Textes durch andere ersetzen
- **cut**: nur bestimmte Spalten einer Datei ausgeben

Ausgabe

- **echo**
  - Ausgabe auf dem aktuellen Terminal mit abschließendem Zeilenumbruch
  - Option **-e**: Sonderzeichenausgabe
    ```
    $ echo -e "Datum\tUhrzeit"
    ```
  - Option **-n**: kein abschließender Zeilenumbruch
- **printf**
  - formatierte Ausgabe: **printf "***Format***"** *Daten*
  - wie in Perl, nur dass die Daten nicht durch Kommas voneinander getrennt werden (nur durch Leerzeichen), z.B.:
    ```
    $ printf "%.0f\t%s\n" 2.6 hallo
    ```
  - Sonderzeichen im Formatstring funktionieren

Eingaben einlesen

- **read**: liest eine Zeile von der Standardeingabe
- ohne Angabe von Variablen: Eingabe wird in **$REPLY** gespeichert
  ```
  $ while read; do echo $REPLY; done
  ```
- mit Angabe von Variablen: Eingabe wird bei Leerzeichen in einzelne Zeichenketten aufgeteilt
  ```
  $ read a b c; echo $c $a $b
  ```


Variablen (1)

- Variablentypen:
  - interne Variablen der bash, z.B.
    - `$GROUPS` Liste aller Gruppen zu denen der Benutzer gehört
    - `$RANDOM` Zufallszahl zwischen 0 und 32767
  - Variablen, die vom Betriebssystem abhängig sind, z.B.
    - `$PWD` Pfad des aktuellen Verzeichnisses
    - `$HOME` Homeverzeichnis des Benutzers
  - Positionsparameter: aufrufende Argumente
  - frei verwendbare Variablen
- Variablenzuweisung: *Variable*=*Wert*
  - Achtung: keine Leerzeichen vor oder hinter dem Gleichheitszeichen!
  - Falls eine Variable noch nicht existiert, wird sie automatisch erzeugt.

  ```
  $ a=2
  $ b=hallo
  $ c='hallo du'
  ```
- Nicht-initialisierte Variablen enthalten automatisch die leere Zeichenkette.

Variablen (2)

- Variablenauswertung:
  - Alle Variablen werden durch das Voranstellen eines Dollarzeichens ausgewertet, d.h. durch ihren aktuellen Inhalt ersetzt.
    `$ echo $a $b $c`
  - Der Namen der Variablen kann dafür auch in geschweifte Klammern gesetzt werden.
    `$ meinevariable=66`
    `$ echo ${meinevariable}`

Positionsparameter

- Einem Shell-Skript können beim Aufruf Argumente übergeben werden, die innerhalb des Skripts in speziellen Variablen, den Positionsparametern, zur Verfügung stehen:
  - **`$1`** enthält das erste Argument, **`$2`** das zweite usw.
  - **`$0`** enthält den Namen des aktuell ausgeführten Shell-Skripts
  - mehrstellige Positionsparameternummern müssen in geschweifte Klammern gesetzt werden, z.B. **`${12}`**
- Spezielle Variablen:
  - **`$#`** Anzahl der gesetzten Positionsparameter
  - **`$@`** Liste mit allen Positionsparametern, d.h. **`($1 $2 $3 ...)`**
  - **`$*`** alle Positionsparameter als eine Zeichenkette, d.h. **`"$1 $2 $3 ..."`**
- **`shift`**: entfernt das erste Element aus der Liste der Positionsparameter, d.h. alle Werte werden um eins nach vorne geschoben

Arrays

- erster Index ist 0
  `$ zahlen=(null eins zwei drei)`
  `$ zahlen[4]=vier`
  `$ echo ${zahlen[2]}`
- letzter Index: **`${#zahlen[*]}`**
  `$ echo ${#zahlen[*]}`
- Ausgabe des Arrays als Liste (Elemente getrennt durch Leerzeichen):
  `$ echo ${zahlen[@]}`
- Die Ausgabe eines Befehls kann in einem Array (oder in einer skalaren Variablen) gespeichert werden: **`array=($(`***Befehl***`))`**
  `$ lsausgabe=($(ls))`

Arithmetik

- Die bash kann nur mit ganzen Zahlen rechnen.
- Syntax: **`$((`** *Berechnung* **`))`**
- Beispiel:
  ```
  $ a=5
  $ echo $(( $a/2 )) Rest $(( $a%2 ))
  2 Rest 1
  ```
- mathematische und logische Operatoren und ihre Präzedenz wie in Perl (ggf. Klammern setzen):
  `+ - * / % && || !`
- Post-Inkrement und -Dekrement: **`++`** und **`--`**
  ```
  $ a=4
  $ echo $((a++))
  4
  $ echo $a
  5
  ```


Expandierungen (1)

- Expandierungen: systematisches Ersetzen einzelner Ausdrücke in einer Befehlszeile
- Klammerersetzungen:
  `$ echo B{ee,i,oo,ro}t`
  `Beet Bit Boot Brot`
- Tilde-Ersetzungen:
  `~/` **`$HOME`**
  `~Benutzer/` **`$HOME`** des *Benutzer*s
  `~+/` **`$PWD`**

Expandierungen (2)

- Dateinamensexpandierung:
  - **`*`** beliebig viele Zeichen (auch keines)
  - **`?`** genau ein beliebiges Zeichen
  - **`[abc]`** ein Zeichen aus der Zeichenmenge
  - **`[^abc]`** ein Zeichen, das nicht in der Zeichenmenge enthalten ist
  - Mengen können auch als Bereiche definiert werden, z.B.:
    `[A-Z]` ein Großbuchstabe
  - **`[0-9]`** eine Ziffer
- Bsp: alle Dateinamen auflisten, die auf .pdf enden
  `$ ls *.pdf`

Variablen- und Parameterersetzungen (1)

- Teile der Zeichenkette im Inhalt einer Variablen werden entfernt oder durch eine andere Zeichenkette ersetzt, bevor die Ausgabe erfolgt.
- Der Inhalt der Variablen selbst wird nicht verändert!
- Entfernen non-greedy von links: **${***Variable#Muster***}**
  entfernt den kleinsten Teil vom Inhalt der Variablen von links
- Entfernen greedy von links: **${***Variable##Muster***}**
  entfernt den größten Teil vom Inhalt der Variablen von links
- Entfernen non-greedy von rechts: **${***Variable%Muster***}**
  entfernt den kleinsten Teil vom Inhalt der Variablen von rechts
- Entfernen greedy von rechts: **${***Variable%%Muster***}**
  entfernt den größten Teil vom Inhalt der Variablen von rechts
- Bsp: Dateien umbenennen (Endung **txt** durch **dat** ersetzen):
  $ for f in *.txt; do mv $f ${f%txt}dat; done

Variablen- und Parameterersetzungen (2)

- erster Treffer von links wird ersetzt:
  ${*Variable/Muster/Ersatz***}**
- alle Treffer werden ersetzt:
  ${*Variable//Muster/Ersatz***}**
- Ersetzung nur wenn Muster am Anfang der Zeichenkette:
  ${*Variable/#Muster/Ersatz***}**
- Ersetzung nur wenn Muster am Ende der Zeichenkette:
  ${*Variable/%Muster/Ersatz***}**
- Beispiel:
  $ name="Anna.Lena.Meyer"
  $ echo ${name//./:}

Verzweigungen

- Syntax:
  if *Bedingung1*
  then *Befehl1*
  elif *Bedingung2*
  then *Befehl2*
  else *Befehl3*
  fi
- **elif**- und **else**-Zweige sind optional
- **elif**-Konstrukte dürfen beliebig oft vorkommen
- nach **then** muss immer ein Befehl stehen – notfalls der Dummy-Befehl **:**

Tests (1)

- Ein Test prüft eine Bedingung und führt abhängig vom Ergebnis dieser Prüfung bestimmte Befehle aus.
- Syntax: **[** *Bedingung* **]**
- Tests für
  - gleich: == bzw. = / -eq
  - ungleich: != / -ne
  - kleiner: < / -lt
  - kleiner gleich: <= / -le
  - größer: > / -gt
  - größer gleich: >= / -ge
- komplexe Bedingungen:
  - NICHT: **!** *Ausdruck*
  - UND: *Ausdruck1* **-a** *Ausdruck2*
  - ODER: *Ausdruck* **-o** *Ausdruck2*

Tests (2)

- Test für Dateien: wahr wenn *Datei*
  - **-e** *Datei* existiert
  - **-s** *Datei* existiert und nicht die Größe 0 hat
  - **-d** *Datei* ein Verzeichnis ist
  - **-r** *Datei* für den Benutzer lesbar ist
  - **-w** *Datei* für den Benutzer schreibbar ist
  - **-x** *Datei* für den Benutzer ausführbar ist
- Testen, ob das Skript mit zwei Argumenten aufgerufen wurde:

```
if [ $# -eq 2 ]
then …
else echo "Aufruf: …"
fi
```


Tests (3)

- Statt eines Tests können auch Befehle in die Bedingung geschrieben werden, z.B.:

```
$ if ! ls | grep "pl$"; then echo "kein pl"; fi
```

- Variante: **[[** *Argument1 Vergleich Argument2* **]]**
- *Argument2* wird als Muster interpretiert:

```
$ if [[ $dateiliste == *.tmp ]]; then …; fi
$ if [[ $dateiliste != *.tmp ]]; then …; fi
```

Mehrfachverzweigungen: **`case`**

- Syntax:
  `case` *Variable* **`in`**
  Muster1 **`|`** *Muster2* **`...`** **`)`** *Befehle* **`;;`**
  Muster3 **`)`** *Befehle* **`;;`**
  `*)` *Befehle*
  `esac`
- ```
  for arg in $@
  do
    case "$arg" in
      -*) ...
      *) ...
    esac
  done
  ```


while- und until-Schleife

- Syntax while-Schleife:
  `while` *Test/Befehl*
  `do` *Befehle*
  `done`
- Syntax until-Schleife:
  `until` *Test/Befehl*
  `do` *Befehle*
  `done`
- Wahrheitswerte:
  - **`true`** ist immer wahr
  - **`false`** ist immer falsch
- Endlosschleifen:
  ```
  $ while true; do ...; done
  $ until false; do ...; done
  ```


for-Schleife (1)

- Syntax:
  `for` *Variable* **`in`** *Liste*
  `do` *Befehle*
  `done`
- ```
  for i in 1 2 5 6; do echo $i; done
  ```
- Ausgabe aller Arrayelemente (auf separaten Zeilen):
  ```
  $ for i in ${zahlen[@]}; do echo $i; done
  ```
- Alle Postscriptdateien in einem Verzeichnis drucken:
  ```
  $ for d in *.ps; do echo $d; lpr $d; done
  ```
- Alle Perlskripte in einem Verzeichnis ausführen:
  ```
  $ for skript in *.pl; do perl $skript; done
  ```


for-Schleife (2)

- Syntax:
  `for ((` *Ausdruck1* **`;`** *Ausdruck2***`;`** *Ausdruck3* **`))`**
  `do` *Befehle*
  `done`
- Beispiel:
  ```
  for (( i=30 ; i>=20 ; i--))
  do echo $i
  done
  ```


was noch fehlt...

- Vorzeitiges Beenden:
  - Schleife vorzeitig beenden: **`break`**
  - Skript vorzeitig beenden: **`exit`**
- Kommentare: #