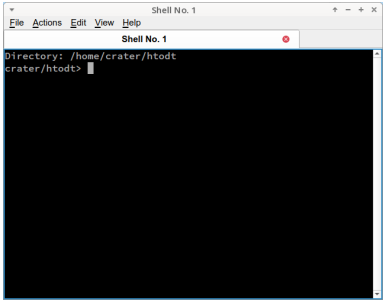
Die UNIX-Shell



Wiederholung und Ergänzung: Linux

Literatur zu Linux & Shell

- Linux / Die wichtigsten Befehle kurz & gut; Daniel J. Barrett, Kathrin Lichtenberg, Thomas Demmig
- Linux in a Nutshell; Ellen Siever, Stephen Figgins, Robert Love
- Unix in a Nutshell; Arnold Robbins
- Linux-Server / Das umfassende Handbuch; Dirk Deimeke, Stefan Kania, Daniel van Soest, Peer Heinlein, Axel Miesen
- Bash - kurz & gut; Karsten Günther

Wiederholung: Linux I

Linux ist ein freies UNIX-Betriebssystem. Seine wesentlichen Komponenten:

Kernel

Programm zur Ansteuerung der Hardware, Verwaltung von Dateien, Netzwerk, etc.
liegt als (bz-komprimierte) Datei unter `/boot/vmlinuz`

Programme

Z.B. zum Editieren von Textdateien (`vi`); Systemdienste

Shell

Benutzerschnittstelle, Eingabe von Befehlen, z.B. `bash`

X/X11

Grafisches System für Fenster, Mausunterstützung etc.
X11-Programme: `gv`, `xemacs`
X11-Umgebungen: `GNOME`, `KDE`
(seit 2008 löst Wayland langsam X11 ab)

Arbeiten mit der Shell, Eingabe von Befehlen:

Befehle

meist Programmnamen gefolgt von *Optionen* und *Argumenten*, z.B. `ls -hl /etc/`

einige Befehle sind auch in die Shell eingebaut, z.B. `set`, `export`

Optionen und *Argumente* sind nicht standardisiert:
`ls -h -l` $\rightarrow$ `h` für human readable
`ps -h` $\rightarrow$ `h` für header

Tipp:

In der `bash` zeigt der Befehl `type` *befehl*, um welche Art von Befehl es sich bei *befehl* handelt (eingebaut, Programm, Alias oder Funktion).
In der `tcsh` heißt der analoge Befehl `which` *befehl*.

Benutzer

Benutzername

Eindeutige Identifikation für Anmeldung (Multiuser)
und Besitzrechte von Dateien
Eigener Datenbereich unter `/home/.../username`

root

Superuser, darf Systembereich (z.B. `/etc/`) bearbeiten,
d.h. auch systemweit Programme installieren
eigener Datenbereich unter `/root/`

Information zum Benutzerstatus

`whoami`	–	Ausgabe der aktuellen Benutzerkennung
`id [user]`	–	Ausgabe der User-ID und Gruppen
`who`	–	Ausgabe der im System eingeloggten User

Wiederholung: Linux IV

Dateien

Jede Datei befindet sich in einem Verzeichnis.

Verzeichnisse

Bilden eine *Hierarchie* bzw. *Baum*:
Verzeichnisse können Unterverzeichnisse enthalten.
Unterverzeichnisse können wieder Unterverzeichnisse enthalten.

Wurzelverzeichnis

Oberstes Verzeichnis, auf jedem Host (Rechner)
nur einmal vorhanden - anders als bei Windows

Mounten

Montieren des *Verzeichnisbaums*, *logische* Bestandteile
liegen *physikalisch* in verschiedener Form vor, z.B.
auf der eigenen Festplatte: `/` und `/boot`
Netzwerk (NFS): `/home/weber/`

Das UNIX-Dateisystem, Anzeige z.B. mit `ls /`

```
/
|
|-- /home/
|    |-- /home/weber/
|        |-- /home/weber/htodt/
|
|-- /etc/
|
|-- /dev/
|
|-- /proc/
|
|-- /bin/
|
...
```

→ Wurzelverzeichnis

→ Homeverzeichnisse
→ Homeverzeichnisse auf weber
→ Helges Homeverzeichnis

→ Konfigurationsdateien

→ Gerätedateien

→ Betriebssytemzustand

→ elementare Programme

Pfad

Schreibweise zur exakten Bezeichung der Lage von Dateien/Verzeichnissen, z.B.

`/etc/fstab`

→ Pfad bezieht sich auf Wurzelverzeichnis /

absoluter Pfad

beginnt mit dem Wurzelverzeichnis /

relativer Pfad

beginnt nicht mit /
bezieht sich auf *aktuelles* Verzeichnis, z.B.
`./Dokumente/`
`../RECO/`
`Dokumente/office`

Aufgabe 4.1 Das Attribut `l`

Im Verzeichnis `/bin` oder `/usr/bin/` (z.B.) befinden sich einige Einträge, deren Dateiattribut mit einem kleinem `l` beginnen.
Lassen Sie sich diese anzeigen.
Betrachten Sie z.B. den Eintrag zu `vi`.
Was steht hinter dem Dateinamen?

Symbolische Links

sind Referenzen auf Dateien/Verzeichnisse oder Links

enthalten einen entsprechenden *Pfad* zum Ziel (hinter ->) z.B.

```
vi -> vim
vim -> /etc/alternatives/vim
```

Anlegen mittels:

`ln -s` *pfad linkname*

Löschen mittels `rm`, löscht nur die Referenz

Wird das Ziel gelöscht, hängt der Link (dangling link).

Exkurs: Hardlinks

Anlegen mittels `ln` *pfad linkname*. Der Linkname ist dabei nur ein alternativer Name für dieselbe Datei (Inodenummer), nützlich für inkrementelle Backups. Funktioniert nur *im selben* Dateisystem. Alternativ bei Copy-on-Write-Dateisystemen (z.B. unter Btrfs): Reflinks mittels `cp -reflink=always`

Wiederholung: Linux IX

In Linux/UNIX ist alles mögliche eine Datei, z.B. auch

- Verzeichnisse (erkennbar am `d` auf der 1. Pos. bei `ls -l`)
- `/dev/sda1`
  $\rightarrow$ erste Partition (1) der ersten (a) SCSI/SATA-Festplatte (sd)
- `/dev/null` $\rightarrow$ Pseudo-Device, schluckt alles
- `/proc/self` $\rightarrow$ *Symbolischer Link* zu Daten des aktuellen Prozesses

Exkurs: Das `/proc/`-Verzeichnis

Dateien in `/proc/` liegen nicht auf der Festplatte, sondern werden vom Kernel im Verzeichnis `/proc/` eingeblendet, Zeitstempel ist daher stets die aktuelle Zeit (z.B. `ls -l`). Beispiele

- `/proc/`*nnn* – Daten zum Prozess mit PID *nnn* der aktuellen Shell
- `/proc/cpuinfo` – Typ/Name des Prozessors (Vgl. `top` $\rightarrow$ `1`)
- `/proc/meminfo` – RAM/Swap-Informationen

Exkus: Dateitypen

In der ersten Spalte wird bei `ls -l` der Dateityp ausgegeben:

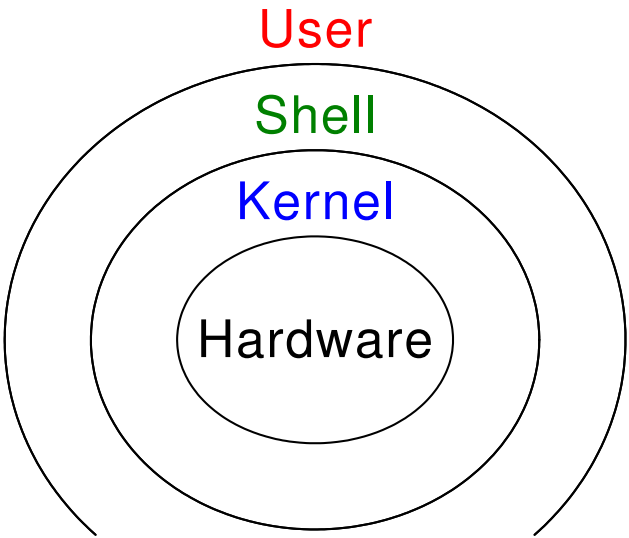
- `-` normale Datei
  ```
  -rw-r--r--   1 htodt users     6018 Mar 14 11:45 wr-pz.pdf
  ```
- `d` Verzeichnis (directory)
  ```
  drwx------   3 htodt users     4096 Mar 14 11:45 .ssh
  ```
- `l` symbolischer Link
  ```
  lrwxrwxrwx 1 root root 12 Feb 28 09:16 /bin/vim -> /usr/bin/vim
  ```
- `b` Blockgerät (in `/dev/`), beliebiger Zugriff, z.B. `/dev/sda1`
  ```
  brw-rw---- 1 root disk 8, 1 Mar 25 14:08 /dev/sda1
  ```
- `c` Zeichengerät (character device, in `/dev/`), serieller Strom I/O, z.B. `/dev/random`
  ```
  crw-rw-rw- 1 root root 1, 8 Mar 25 14:08 /dev/random
  ```
- `p` Pipe für Kommunikation zw. Prozessen
  ```
  prw-r--r-- 1 htodt astro      0 Sep 30 10:32 com.zoom.ipc.confapp__res
  ```
- `s` Socket für bidirektionale Kommunikation zw. Prozessen
  ```
  srwxr-xr-x 1 htodt astro       0 Feb 23 22:56 oauc_pipe
  ```

Der Verzeichnisbaum wird dynamisch erweitert:

Aufgabe 4.2 Mounten

1. Auf welchem Rechner befindet sich Ihr Homeverzeichnis? *Tipp: Der vollständige Pfad Ihres Homeverzeichnisses enthält den Rechnernamen.*
2. Wechseln Sie in das Verzeichnis `/home/`, lassen Sie sich dessen Inhalt im *Langformat* anzeigen.
3. Loggen Sie sich auf dem Host ein, auf dem Ihr Homeverzeichnis liegt und wiederholen Sie Schritt 2. Was fällt Ihnen auf? Loggen Sie sich wieder aus.
4. Falls Sie einen USB-Speicherstick haben und nicht an einem Server sitzen: Stecken Sie ihn in einen freien USB-Port Ihres PCs. Finden Sie heraus, in welches Verzeichnis relativ zu / der Stick gemountet wird. Mounten Sie ihn wieder ab, bevor sie ihn herausziehen. Warum könnte das nötig sein?

Die Shell im Detail


User
Shell
Kernel
Hardware

Shell

- fungiert als *eine* Schnittstelle zwischen Betriebssystem (Kernel) und User
- gibt es in verschiedenen Ausführungen: z.B. `tcsh`, `bash`, `zsh`[†]
- ist ein Programm (wird beim Öffnen eines Terminal/Konsole gestartet)

Aufgabe 4.3 Die Shell als Programm

1. Welche anderen Möglichkeiten der Interaktion mit dem Betriebssystem außer der Shell kennen Sie bereits?
2. Öffnen Sie eine Xterm, stellen Sie mittels `echo $0` fest, welche Shell sie gerade benutzen.
3. Finden Sie mittels des Befehls `type` *shell*, wobei *shell* für den Namen Ihrer Shell steht, heraus, wo im Verzeichnisbaum das Shellprogramm liegt. Ermitteln Sie dessen Dateiattribute (Besitzer, Rechte).

[†] Die `zsh` ist moderner als die `bash`, aber leider kaum dokumentiert (wenig Literatur).

Dateiattribute

```
-rwxr-x--x 1 user group 120798 Oct 24 14:31 .login
```

user group other — Eigentümer — Größe in Byte

Exkurs: Besitzer und Superuser

- jede Datei gehört *einem* Besitzer, der ein User ist; nur dieser kann die Zugriffsrechte ändern (auch die Eigentümerschaft)
- ein User kann mehreren Gruppen angehören → `id [user]`
- der Superuser (`root`, admin) kann sich alle Rechte beschaffen z.B. mittels `su` (substituiere User), `chown` (ändere Besitzerrechte)
- i.d.R. kann nur der Superuser Programme installieren, da nur er Schreibrechte für Systemverzeichnisse hat (oder erhalten kann)

Shellvariablen I

Unterschiede zw. `tcsh` und `bash` gibt es vor allem in der Syntax shellspezifischer Befehle:

Shell-Variablen

- speichern Zustände ab (z.B. Variable `$PWD`)
- werden z.T. von der Shell gesetzt (automatische Variablen)
- können vom User bearbeitet werden:

	tcsh	bash	
neu anlegen	`set VAR`	`set VAR`	
Zugriff auf Wert	`$VAR`	`$VAR`	(z.B. `echo $VAR`)
Wert ändern	`set VAR=wert`	`VAR=wert`	
löschen	`unset VAR`	`unset VAR`	

Aufgabe 4.4 Variablen setzen

1. Legen Sie ein neues Verzeichnis an und wechseln Sie dort hinein. Ermitteln Sie den Wert von `PWD` mittels `echo $PWD`, vergleichen Sie ihn mit dem Ergebnis des Befehls `pwd`.
2. Setzen Sie eine Variable `color`, die als Wert den Namen Ihrer Lieblingsfarbe enthält. Überzeugen Sie sich vom Erfolg des Setzens mittels `echo $color`.
3. Starten Sie im aktuellen Terminal eine neue Shell, in dem Sie den Namen der Shell als Befehl aufrufen.
4. Lassen Sie sich den Wert von `PWD` und `color` ausgeben. Was fällt auf?
5. Beenden Sie die aktuelle Shell und lassen Sie sich erneut den Wert von `color` ausgeben.

Shellvariablen III

Lokale Variablen

Sind nur in der aktuellen Shell gültig.

Setzen:
```
dbx=yes        (bash)
set dbx=yes    (tcsh)
```

Umgebungsvariablen

Werden auf alle Shells vererbt, die von der aktuellen Shell aus gestartet werden (auch `tcsh` ↔ `bash`).

Setzen:
```
export COLOR=blue   (bash)
setenv COLOR blue   (tcsh)
```

Tipp:

Es ist üblich, für lokale Variablen nur Kleinbuchstaben und für Umgebungsvariablen nur Großbuchstaben zu verwenden.

Beachte:

	Syntax:	Beispiel
Setzen:	`VAR`	`export COLOR=blue`
Aufrufen:	`$VAR`	`echo Lieblingsfarbe: $COLOR`

Variablennamen muss beim Aufruf immer ein Dollarzeichen $ vorangestellt werden

Eine Liste aller gesetzten Variablen erhält man mit dem Befehl `set`, bzw. `setenv` oder `export`.

Aufgabe 4.5

Lassen Sie sich alle lokalen und Umgebungs-Variablen anzeigen.

Einige wichtige Variablen

`HOME`	absoluter Pfad zum Homeverzeichnis
`SHELL`	die eigene Login-Shell (Pfad zum Shell-Programm)
`PATH`	Liste von Pfaden für ausführbare Programme/Dateien

Aufgabe 4.6

Lassen Sie sich den Wert von `PATH` ausgeben und vergleichen Sie diesen mit der Ausgabe des Befehls `which` *shell* (wobei *shell* für `bash` oder `tcsh` steht), den Sie bereits in Aufgabe 4.3 benutzt haben.

Exkurs: PATH

- Die Variable $PATH enthält eine *Liste* von Verzeichnissen, in den die Shell nach ausführbaren Dateien sucht, um diese als Programme starten zu können
- Der User darf und sollte diese Liste ergänzen, um z.B. eigene Programme aufrufbar zu machen
- Um z.B. Dateien aus dem jeweils aktuellen Verzeichnis ausführen zu können kann man . in PATH hinzufügen
  → nicht empfehlenswert - Warum?
  besser: Verwendung des Aufrufs *./programm* im aktuellen Verzeichnis

Gebrauch von Variablen

Variablen werden an entsprechender Stelle von der Shell *expandiert* (auflösen), z.B.

```
cd $HOME
```

Aufgabe 4.7 PATH ergänzen

Fügen Sie das Verzeichnis `muCommander` zur `PATH`-Variable hinzu:

- `PATH` wird einfach mit folgendem Wert neu gesetzt (Umgebungs-Variable!):
  `${PATH}:${HOME}/muCommander`
- Wozu könnten die geschweiften Klammern dienen?
- Überzeugen Sie sich, dass das Ändern der `PATH`-Variable erfolgreich war (Tipp: `echo`).
- Nun sollten Sie in der Lage sein, `mucommander.sh` auf der Befehlszeile aufzurufen.

grep, find, Umleitungen und Pipes

Muster suchen mit grep I

Der Befehl `grep` ist einer der mächtigsten Unix-Befehle, man kann mit ihm nach *Mustern* suchen:

`grep` *Muster datei*

Aufgabe 4.8 Dateien durchsuchen

Lassen Sie sich mittels `grep` die Zeile der Datei `/etc/passwd` ausgeben, die den Benutzernamen `SS_08` enthält.

→ `grep` durchsucht die *Datei* `/etc/passwd` zeilenweise nach *SS_08* und gibt die entsprechende *Zeile* aus

Hinweis: Mehrere Dateien durchsuchen

Die Option `-r` ermöglicht das rekursive Durchsuchen aller Dateien im aktuellen Verzeichnis und Unterverzeichnissen. Am besten mit `--include=` kombinieren, z.B.:

```
grep -r --include=.tex openssl
```

→ durchsucht alle Dateien mit der Endung `.tex` im aktuellen Verzeichnis nach dem Muster `openssl`.

Muster suchen mit grep II

Der Befehl `grep` kann noch auf eine zweite Art verwendet werden, mithilfe des Pipe-Symbols | (gerader Strich).
Die Pipe (Röhre) verbindet die Ausgabe (STDOUT) eines Befehls mit der Eingabe (STDIN) eines anderen Befehls:

befehl1 | *befehl2*

Also z.B.

```
ls -l | grep drwx
```

Aufgabe 4.9 grep und Pipe

1. Was gibt die Befehlskombination `ls -l | grep drwx` eigentlich aus?
2. Lassen sie sich eine Liste von Dateien in `/bin/` anzeigen, deren Namen das Muster `sh` enthalten. Worum handelt es sich bei den meisten dieser Dateien?

Umleitungen I

Neben der Pipe | gibt es noch weitere Operatoren zum Umleiten von Befehlsausgaben (STDOUT) und Eingaben (STDIN), z.B.:

> *datei*	leitet STDOUT in *datei* um (überschreiben)
>> *datei*	leitet STDOUT in *datei* um (anhängen)
< *datei*	STDIN aus *datei*
<< *text*	lesen von STDIN(!) bis *text* gefunden

Die Umleitungsoperatoren sind insbesondere zusammen mit `cat` sehr nützlich.

Der Befehl `cat` *datei* schreibt den Inhalt von *datei* auf den Bildschirm (STDOUT). Z.B.

```
cat /etc/fstab
```

Exkurs: Die <<-Umleitung

Die <<-Umleitung wird häufig zusammen mit `cat` benutzt, z.B.:

```
cat > ~/.ssh/config << EOF
Host grieg
Hostname 141.89.178.81
User htodt
EOF
```

wobei jede Zeile mit ENTER abgeschlossen wird. Der Text zwischen dem 1. und 2. "EOF" wird in die Datei `~/.ssh/config` geschrieben.

Umleitungen III

Der Name `cat` leitet sich von *concatenate* (aneinanderhängen) ab. Mithilfe von `cat` und `>>` können Dateien aneinandergehängt werden:

```
cat file2 >> file1
```

Hängt Datei mit Namen *file2* an Datei mit Namen *file1* an.

```
cat file2 file3 >> file1
```

Hängt Datei mit Namen *file2* an Datei mit Namen *file1* an, anschließend wir die Datei mit dem Namen *file3* an *file1* angehängt.

`cat`: Postscript(.ps)-Dateien

Klassische Anwendung für `cat`:

```
cat seite1.ps seite2.ps seite3.ps >> total.ps
```

Für PDF-Dateien muss stattdessen das Tool `pdftk` verwendet werden:

```
pdftk seite1.pdf seite2.pdf seite3.pdf cat output total.pdf
```

Aufgabe 4.10 Passwortfreies Login

Um sich mittels `ssh` ohne Passworteingabe mit einem anderen Rechner im Computerpool verbinden zu können benötigt man ein Schlüsselpaar:

1. Führen Sie den Befehl `ssh-keygen` aus. Bestätigen Sie nur mit <ENTER>, geben Sie keine Passphrase ein. Der Befehl erzeugt ein Schlüsselpaar in `~/.ssh`.
2. Wechseln Sie in Ihr `.ssh`-Verzeichnis, überprüfen Sie, dass nur Sie Schreib-, Ausführ- und Leserechte für dieses Verzeichnis haben und dass das Schlüsselpaar darin angelegt wurde (z.B. `id_rsa.pub`).
3. Hängen Sie den öffentlichen Schlüssel (erkennbar an der `.pub`-Endung) an die Datei `authorized_keys` im `.ssh`-Verzeichnis an (egal ob die Datei existiert).
4. Sie können Sich jetzt passwortfrei auf anderen Computerpool-Rechnern einloggen.

Einloggen von zu Hause

Wiederholen Sie den Schritte 1 aus Aufgabe 4.10 auf Ihrem Unix-Rechner (Linux, MacOS). Kopieren (`scp`) Sie den so erzeugten `pub`-Schlüssel in Ihr Homeverzeichnis hier im Pool. Loggen Sie sich im Pool ein, hängen Sie auch diesen Schlüssel mittels cat und >> an `authorized_keys` an.

Exkurs: Asymmetrische Verschlüsselung I

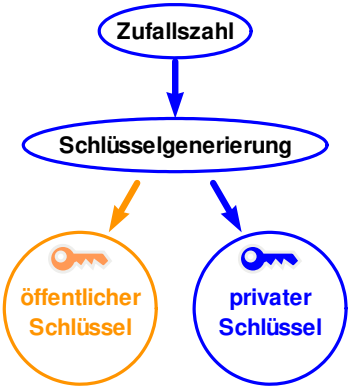
Symmetrische Verschlüsselung

z.B. AES: gleicher Schlüssel zum Ver- und Entschlüsseln
$\rightarrow$ schnell, sehr sicher (auch gegenüber Quantencomputern);
Nachteil: sicherer Schlüsselaustausch erforderlich

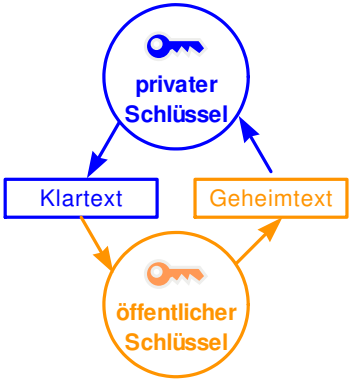
Asymmetrische Verschlüsselung

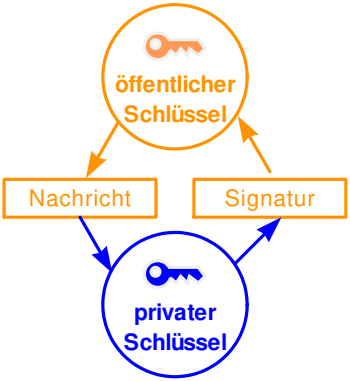
z.B. TLS, SSH mittels RSA, DSA: getrennte Schlüssel für Ver- und Entschlüsselung
$\rightarrow$ kein eigentlicher Schlüsselaustausch nötig, privater Schlüssel zum Signieren (Digital Signature Algorithm);
Nachteile: langsam; unsicherer, z.B. derzeitige Implementierungen (Primfaktoren, diskrete Logarithmen) durch Quantencomputer angreifbar

Exkurs: Asymmetrische Verschlüsselung II



orange: öffentlich, blue: privat





Exkurs: Asymmetrische Verschlüsselung III

Bsp. 1: Sender (Alice) möchte Nachricht `geheim.txt` an Besitzer (Bob) des Schlüssels `id_rsa.pub` (Bobs Schlüssel) schicken:

Bob wandelt seinen `.pub`-Key in ein passendes Format um

```
ssh-keygen -e -f id_rsa.pub -m PKCS8 > id_rsa.pub.pkcs8
```

Erzeugen des Einmal(!)-Passworts für die symm. Verschlüsselung (Alice)

```
openssl rand -out geheim.key 192
```

Symm. AES-Verschlüsselung mit Einmal-Passwort (Alice)

```
openssl aes-256-cbc -pbkdf2 -in geheim.txt -out geheim.txt.enc -pass file:geheim.key
```

Asymm. Verschlüsselung des Passworts (Alice)

```
openssl pkeyutl -encrypt -pubin -inkey id_rsa.pub.pkcs8 -in geheim.key -out geheim.key.enc
```

Exkurs: Asymmetrische Verschlüsselung IV

Empfänger Bob erhält `geheim.txt.enc` und `geheim.key.enc` von Alice

Bob entschlüsselt das Einmal-Passworts mit seinem privatem Schlüssel

```
openssl pkeyutl -decrypt -inkey id_rsa -in geheim.key.enc -out geheim.key
```

Bob entschlüsselt die symm. verschlüsselte Nachricht

```
openssl aes-256-cbc -pbkdf2 -d -in geheim.txt.enc -out geheim.txt -pass file:geheim.key
```

- Bob sollte seinen `.pub`-Key im PKCS8-Format verteilen (z.B. über Website); Sicherstellen der Identität des Schlüsselinhabers (Bob) über Fingerprint:

  ```
  ssh-keygen -l -e -f id_rsa.pub
  ```

- die Verschlüsselung der eigentlichen Nachricht erfolgt mit AES, da asymm. Verschlüsselung langsam und nur kurze Nachrichten verschlüsseln kann

Exkurs: Asymmetrische Verschlüsselung V

Bsp. 2: Sender (Alice) möchte Nachricht `wichtig.txt` an Bob schicken und sicherstellen, dass diese unverfälscht ankommt (Signieren)

Alice erzeugt aus ihrem privaten Schlüssel eine Version des öffentlichen Schlüssels zur Überprüfung der Signatur

```
openssl rsa -in id_rsa -pubout > id_rsa.pub.pem
```

Alice signiert die Nachricht `wichtig.txt` mit ihrem privaten Schlüssel

```
openssl dgst -sign id_rsa wichtig.txt > wichtig.txt.sha
```

Alice schickt `wichtig.txt` und die Signatur des Textes `wichtig.txt.sha` an Bob

Bob verifiziert die Nachricht `wichtig.txt` mittels Alice öffentlichen Keys

```
openssl dgst -verify id_rsa.pub.pem -signature wichtig.txt.sha wichtig.txt
```

Verschlüsseln und Signieren sollten gleichzeitig angewandt werden!

grep, find, Umleitungen und Pipes - Fortsetzung

find - Dateien finden I

Der Befehl `find` durchsucht das Dateisystem nach Dateien, die bestimmte - vom User vergebene - Kriterien erfüllen:

`find` *Verzeichnis Suchkriterium*

Häufigster Einsatz: Finden einer Datei mit einem bestimmten Namen

```
find . -name wichtig.doc
```

Aufgabe 4.11 find

1. Wofür steht der erste Punkt . im obigen Beispiel?
2. Suchen Sie mit Hilfe von `find` in ihrem Homeverzeichnis nach der Datei `init.el`. Wo befindet sie sich?

find - Dateien finden II

Tipp: `find` ist sehr mächtig, z.B. kann es mit allen gefundenen Dateien eine Aktion ausführen. Dazu gibt es die Option `-exec`. Z.B.

```
find ~ -name "*.doc" -exec cp "{}" /tmp \;
```

Dabei steht `"*.doc"` für alle Dateien, die auf `.doc` enden.
Die Option `-exec` wird mit `\;` abgeschlossen, dazwischen steht ein ausführbarer Shell-Befehl, der mit allen Fundstücken ausgeführt wird, als Platzhalter (Variable) für die gefundenen Dateien dient hierbei `"{}"`.

Aufgabe 4.12

Was passiert mit den gefundenen Dateien in diesem Beispiel?

find - Dateien finden III

Darüberhinaus beherrscht `find` viele weitere Suchoptionen, z.B. `-mtime 0` findet Dateien, die innerhalb der letzen 24h modifiziert wurden, mittels `-type l` werden symbolische Links gefunden.

Aufgabe 4.13 find und STDERR

1. Suchen Sie im Wurzelverzeichnis nach der Datei `mtab`. Dafür sollten Sie unbedingt auch die Option `-maxdepth 2` im Aufruf von `find` hinzufügen (Warum?). Was zeigt `find` außer der Fundstelle noch an?
2. Leiten Sie die Ausgabe von `find` in eine Datei um und vergleichen Sie den Inhalt der Ausgabe mit der vorigen Ausgabe von `find`.

STDERR

manche Befehle schreiben außer nach STDOUT auch in einen speziellen Fehlerkanal, STDERR, der standardmäßig zusammen mit STDOUT ausgegeben wird

Exkurs: Die Dateien mtab und fstab

`/etc/mtab`

Die Datei `mtab` enthält eine Liste der auf dem Host aktuell gemounteten Dateisysteme. Diese Liste wird auch ausgegeben, wenn das Programm `mount` ohne Argumente gerufen wird.

`/etc/fstab`

Die Datei `fstab` enthält Informationen, welche Dateisysteme und wie auf dem Host gemountet werden sollen. Beim Systemstart montiert das Programm `mount` mithilfe dieser Datei den Verzeichnisbaum. Die Datei wird bei der Systeminstallation angelegt und wird ggf. vom Systemadministrator angepasst.

Die Umleitungsoperatoren |, >, >>, usw. leiten nur STDOUT um.
Die Ausgabe von STDERR kann mittels

```
                2>                    (bash)
bzw.   ( befehl ) > &                 (tcsh)
```

umgeleitet werden. Möchte man z.B. die unübersichtliche Fehlerausgabe von `find` loswerden, so kann diese in das NULL-Gerät `/dev/null` umgelenkt werden:

```
find / -name mtab 2> /dev/null                    (bash)
( find / -name mtab > /dev/tty ) > & /dev/null    (tcsh)
```

Aufgabe 4.14 find-STDERR nach /dev/null

Wiederholen Sie die Suche aus Aufgabe 4.13 mit der oben beschriebenen Methode der Fehlerumleitung.

Umleiten von STDOUT UND STDERR

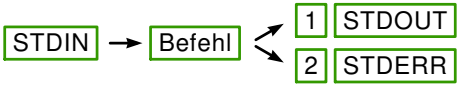
möchte man beide Ausgabekanäle bündeln, z.B. um sie in eine Datei zu schreiben, so hilft `&`:

bash

	befehl `>` *datei* `2>&1`	schreibt STDOUT und STDERR in *datei*
	befehl1 `2>&1` `|` *befehl2*	sendet STDOUT und STDERR
auch:	*befehl1* `|` *befehl2*	von *befehl1* an *befehl2*

tcsh

befehl `>&` *datei*	schreibt STDOUT und STDERR in *datei*
befehl1 `|&` *befehl2*	sendet STDOUT und STDERR von *befehl1* an *befehl2*



tee

liest von STDIN, leitet nach STDOUT weiter UND kopiert gleichzeitig das Gelesene in eine oder mehrere Dateien

→ Anwendung zw. Pipes:

befehl1 datei1 | `tee` *datei[en]* | *befehl2*

`cat` *datei1* | `tee` *datei[en]* | `grep` *huhu*

Aufgabe 4.15 tee

Grepen Sie in der Datei `/var/log/Xorg.0.log` nach `warning`, geben Sie das Ergebnis auf der Kommandozeile aus und schreiben Sie es gleichzeitig in eine Datei `hostfehler`.

Reguläre Ausdrücke I

Die Möglichkeiten von Shellbefehlen können durch Metazeichen und reguläre Ausdrücke beträchtlich erweitert werden.

Metazeichen

stehen in Zeichenketten (Strings) nicht für sich selbst, sondern haben eine andere Bedeutung,
Bsp.: * steht für beliebige Zeichen

Das Metazeichen * ist eine sog. Wildcard (Joker).

Reguläre Ausdrücke

stellen eine Mengenbeschreibung / Filter dar, werden durch Kombination von Metazeichen und normalen Zeichen realisiert
Bsp.: `*.pdf` bedeutet alle Dateien, deren Namen auf `.pdf` enden

Reguläre Ausdrücke II

Weitere Metazeichen und Bereiche für reguläre Ausdrücke:

`?`	genau ein beliebiges Zeichen;	`[a?z]`	a oder z, auch: `[a?b?z]`
`[0-9]`	eine der Ziffern;	`?(a|z)`	bash: a oder z
`[^a-bA-Z]`	keiner der Buchstaben;	`{a,z}`	csh: a oder z

darüber hinaus gibt es (von der Shell) nicht-expandierbare e† Metazeichen, die vor allem für `grep` wichtig sind

`"^a"`	mit a am Anfang
`"z$"`	mit z am Ende
`'eins|zwei'`	`grep`: Zeichenfolge `eins` oder `zwei`

Bsp. `ls -l | grep "^-"` findet alle regulären Dateien

Vorsicht: Metazeichen * in grep

→ beliebige Wiederholungen des vorangehenden Zeichens/Ausdrucks,
z.B. `"[0-9]*"` (Ziffernfolgen), `"*.tex"` (nichts).

†Expandieren: Umwandlung des Metazeichens in normale Zeichen = exakte Namen

Beispiel: Falls folgende Dateien in einem Verzeichnis

```
anfang1.ps   neu1.ps   neu2.ps   neu.ps
anfang1.pdf  neu1.pdf  neu2.pdf  neu.pdf
```

Ausdruck	Ergebnis (Shell-Expansion)
`neu?.ps`	`neu1.ps neu2.ps`
`neu.p*`	`neu.pdf neu.ps`
`neu?.p*`	`neu1.pdf neu1.ps neu2.pdf neu2.ps`
`*u[0-1].ps`	`neu1.ps`

Beispiel: `ls | grep "^b"` filtert alle Dateien, die mit `b` beginnen[†]

Aufgabe 4.16

1. Mit welcher Befehlsfolge kann man sich eine Liste aller Verzeichnisse in `/etc/` anzeigen lassen?
2. Wie kann man sich eine Liste der User aus `/etc/passwd` anzeigen lassen, die *keine* Dämonen (daemon=Systemdienst) sind? *Tipp: Manpage nutzen.*
3. Wie kann man sich alternativ in einem Verzeichnis alle Dateien, die mit b anfangen, anzeigen lassen?

[†] ggf. funktioniert das nicht, wenn der Befehl `ls` so gealiased ist, dass bei der Ausgabe Farben verwendet werden, dann: `/bin/ls | grep "^b"`

Anführungszeichen in der Shell I

Die Shell unterscheidet drei verschiedene Anführungszeichen:

" doppelte Anführungszeichen: Zusammenhalten von Zeichenfolgen (Namen, Text), die Leerzeichen enthalten. Erlaubt Expansion von Variablen ($) - Wildcards (*) werden von der Shell nicht expandiert.

Bsp. `echo "$HOME *"` ergibt `/home/weber/htodt *`

' einfache gerade Anführungszeichen: Stärker als doppelte Anführungszeichen, es werden auch keine `$Variablen` expandiert.

Bsp. `echo '$HOME'` ergibt `$HOME`

` einfache geneigte Anführungszeichen (Backticks): Der enthalte Text wird als Befehl verstanden und ausgeführt:

Bsp. `ls -l `which bash``

Achtung:
Anführungszeichen sehen u.U. auf dieser Seite anders aus als auf der Shell.

Anführungszeichen in der Shell II

bash: Backticks vs. $( )

In der bash sollte statt Backticks besser die Befehlssubstitution mittels $( ) verwendet werden, da diese geschachtelt werden kann und ggf. auch leichter zu erkennen ist. Bsp.

```
echo $(basename $(dirname $PWD))
```

gibt nur den Namen des übergeordneten Verzeichnis des aktuellen Verzeichnisses aus.

Aufgabe 4.17 Quoting

Welche Ausgabe ergeben wohl folgende Zeilen auf der Shell?

```
echo My "$HOME" is my '$HOME'
echo And `ls *` is all I have in $PWD
```

Shell-Scripting

Shell-Modi I

Die Shell (tcsh/bash) unterstützt zwei Modi:

- interaktiv: Eintippen von Befehlen
- Script: Abarbeiten einer Liste von Befehlen, z.B. aus einer Datei
- Wir werden hauptsächlich mit der `bash` arbeiten.

Aufgabe 4.18 Einfaches Shell-Script

1. Legen Sie ein Verzeichnis `myscripts` an und wechseln Sie in dieses hinein.
2. Schreiben[a] Sie folgendes in eine Datei `hallo.bash`:

```
#!/bin/bash
echo Hallo, Welt!
# Dies ist ein Kommentar ...
exit
```

3. Machen Sie die Datei ausführbar mittels `chmod u+x hallo.bash`.
4. Führen Sie die Datei (Script) aus: `./hallo.bash`

[a] *Hinweis:* Der Editor `emacs` beschwert sich eventuell, dass Ihr Skript nicht mit einer `newline` endet. Lassen sie ihn diese hinzufügen (`yes`).

Ausführbarkeit von Dateien

`ls -l`	zeigt die Ausführbarkeit von Dateien an: `-rwx------` ausführbar für Eigner
`chmod u+x` *datei*	macht *datei* ausführbar
./datei	führt im aktuellen Verzeichnis (!) *datei* aus

Hinweis

Tippt man den Namen einer ausführbaren Datei in die Shell ein, so sucht die Shell nach dieser Datei nur in den unter `$PATH` (anzeigbar mittels `echo $PATH`) gelisteten Verzeichnissen. Befindet sich das aktuelle Verzeichnis nicht in dieser Liste, so ist *./datei* zum Ausführen erforderlich.

Shell-Skripte ausführen

Es gibt verschiedene Arten, Shell-Skripte auszuführen:

- die Datei selbst ist ausführbar, der Shebang `#!` legt die Shell fest
- `source` *script*, ausführen in der aktuellen Shell
- `.` *script*, wie `source` (nur `bash`)
- `bash` *script* oder `tcsh` *script*, starten einer neuen Shell und ausführen des Scripts darin; Bsp. `.cshrc`, `.login` bzw. `.bashrc`, `.profile`

Bezüglich der Struktur von Shellskripten gelten folgende einfache Regeln:

Pro Zeile steht nur ein Befehl.

(mehrere Befehle pro Zeile müssen wie auf der Shell durch ein Semikolon `;` getrennt werden)

In der 1. Zeile steht der Shebang. Z.B.

```
#!/bin/bash
```

→ nur falls das Skript selbst ausführbar sein soll oder unabhängig von der User-Shell gerufen werden soll,

Die Start-Skripte I

Beim Starten einer neuen Shell (z.B. beim Öffnen eines Xterms), können automatisiert bestimmte Startskripte (ohne Shebang) ausgeführt werden.

Unterschied: Nicht-Login-Shell - z.B. durch `bash` gestartet
Login-Shell - z.B. durch `xterm -ls` oder `bash -ls` gestartet

Login-Shell		*Nicht-Login-Shell*	
bash	tcsh	bash	tcsh
`/etc/profile`	`/etc/csh.cshrc` `/etc/csh.login`	`/etc/bash.bashrc`	`/etc/csh.cshrc`
`~/.profile`	`~/.cshrc` `~/.login`	`~/.bashrc`	`~/.cshrc`

Zweck: Setzen von Umgebungsvariablen (z.B. `PATH`) und `alias`

Die Start-Skripte II

`alias`

der Befehl ermöglicht Neu- und Umdefinitionen von Befehlen

alias in der `bash`

`alias` *befehl1=befehl2*

Ausführen von *befehl2*, wenn *befehl1* ausgeführt werden soll

Beispiel:

```
alias ll="ls -lh"
alias la="ls -lah"
```

alias in der `tcsh`

`alias` *befehl1 befehl2*

Aufgabe 4.19 Startskripte anpassen

1. Fügen Sie die beiden oben erwähnten Aliase dem Startskript Ihrer *User-Login*-Shell[a] (Typ steht in der Variablen `SHELL`) hinzu. Achten Sie auf die Syntax und überlegen Sie sich, in welche der Start-Skripte Sie sie schreiben sollten.
2. Aus Sicherheitsgründen sollten Sie auch die Befehle `rm`, `cp` und `mv` so aliasen, dass jeweils der Befehl immer zusammen mit der Option `-i` aufgerufen wird (Bewirkt was?).
3. Mit folgendem Befehl in ihrem Startskript wird automatisch die erste zutreffende Manpage angezeigt:
   `export MAN_POSIXLY_CORRECT=1`
4. Ergänzen Sie ihre Pfad-Variable im Shell-Startskript so, wie in Aufgabe 4.7 beschrieben.

[a] *Achtung, Begriffswirrwarr:* Hier ist die Shell (`bash` oder `tcsh`) gemeint, die gestartet wird, wenn ein `xterm` o.ä. geöffnet wird.

Exkurs: Einstellungen für die Manpages

- Pfade für Manpages in Variable $MANPATH → kann insbesondere für selbstinstallierte Programme (z.B. unter /opt/) ergänzt werden, z.B.:

```
export MANPATH=/opt/intel/comserexe/man/en_US:${MANPATH}   # (bash)
setenv MANPATH /opt/intel/comserexe/man/en_US:${MANPATH}   # (tcsh)
```

- Falls beim Aufruf von `man` mehrere Auswahlmöglichkeiten gezeigt werden, kann aus Komfortgründen die Umgebungsvariable `MAN_POSSIXLY_CORRECT` auf `1` gesetzt werden:

```
export MAN_POSIXLY_CORRECT=1   # (bash)
setenv MAN_POSIXLY_CORRECT 1   # (tcsh)
```

Folgendes Konstrukt ist häufig sehr nützlich:

Einfache Schleife – Listengesteuert

```
for var in liste
do
  ...
done
```

Beispiel

```
for name in Jana Franz Michael
do
 echo Hallo, ${name}!
done
```

Schleife: Wiederholungen von Befehlen

Einfache Schleife (Loop) – Listengesteuert

```
for var in liste
do
  ...
done
```

`for ...`	Schleifenkopf, Steuerung
`var`	Variable, wird bei jedem Schleifendurchlauf mit nächstem Listeneintrag befüllt
liste	Liste mit Einträgen, i.d.R. durch Leerzeichen getrennt

= Für jeden Listeneintrag tue folgendes:

`do ... done`	Schleifenkörper: Abarbeiten von Befehlen

Das o.g. Beispiel lässt sich auch so schreiben:

Beispiel

```
studenten="Jana Franz Michael"
for name in $studenten
do
 echo Hallo, ${name}!
done
```

Tipp: Mit Hilfe der Backticks `` ` ` ``, bzw. `$( )` ist es möglich, die entsprechenden Listen automatisch mittels eines Shell-Befehls zu erzeugen, z.B. `$(ls)`
$\rightarrow$ z.B. praktisch für automatisches Umbenennen von Dateien

Aufgabe 4.20 `for-in`-Schleife

1. Legen Sie mit Ihrem Editor eine Datei namens `forlisteloop1.bash` an. Schreiben Sie den Shebang in die erste Zeile:
   `#!/bin/bash`
2. Schreiben Sie eine `for-in`-Schleife, die für jeden Eintrag im Verzeichnis `/home/weber/` den Befehl `id` *Eintrag* ausführt.
3. Machen Sie die Datei ausführbar und führen Sie sie aus.

Argumente an Skripte übergeben

An Skripte und Funktionen (s.u.) können beim Aufruf Argumente (durch Leerzeichen getrennt) übergeben werden:

```
./skript arg1 ... argn
```

Zugriff auf Argumente im Skript:

z.B. `${1}` → erstes Argument *arg1* usw.

Anzahl der Argumente in Variable `${#}`

Variable `${0}` enthält Skriptnamen (oder Programmnamen) selbst (vgl. `echo ${0}` in der Shell)

Hinweis: Die geschweiften Klammern bei `${0}` usw. sind für einstellige Variablennamen optional, aber ab `${10}` erforderlich, da `$10` = `${1}0`.

Zählschleifen I

Statt einer Liste, kann auch ein Zähler zur Schleifensteuerung dienen:

Zählschleife

```
for (( var=anf ; bedingung ; incr ))
do
  ...
done
```

Beispiel

```
for (( i=1 ; i<=10 ; i+=1 ))
do
 echo $i
done
```

Zählschleifen II

Zählschleife

```
for (( var=anf ; bedingung ; incr ))
do
  ...
done
```

`for (( ... ))`	Zählschleifen, doppelte Klammern!
var	Variablenname (Shell), Ganzzahl(!)
anf	Startwert, Ganzzahl
bedingung	i.d.R. ein Vergleich (s.u.)
incr	Inkrement: wie *var* hochgezählt werden soll

Bedingungen in Zählschleifen

Bed.	Bedeutung	Beispiel
==	ist gleich	`i==$k`
!=	ungleich	`j!=3`
<=	kleiner gleich	`i <= 10`
<	kleiner	`k < 2`
>=	größer gleich	`i>=$k`
>	größer	`r> 45`

Inkremente/Dekremente

Inkrement	Bedeutung	Beispiel
+= *n*	Zähler um *n* erhöhen	`i+=3`
-= *n*	Zähler um *n* verkleinern	`j -= 1`

Aufgabe 4.21 Umbenennen

1. Erzeugen Sie Dateien `neu1.txt ... neu9.txt`.
   Tipp: Mit dem Befehl `touch` *datei* kann eine leere Datei mit dem Namen *datei* erzeugt werden.
2. Benennen Sie von diesen Dateien jeweils die mit ungeraden Nummern mithilfe einer Schleife, möglichst in einer Skriptdatei, in `neu1.dat` usw. um. Sie benötigen den Befehl zum Umbenennen von Dateien.

Der Befehl `let`

`let` *ausdruck*
ermöglicht in der `bash` die Berechnung von arithmetischen Ganzzahl-Ausdrücken, insbesondere Zuweisungen z.B.

```
let i=i+1
```

- keine Leerzeichen um das =-Zeichen
- der Wert der Variablen kann ohne $ abgerufen werden
- synonym können auch wieder doppelte Klammern verwendet werden:

  ```
  ((i=i+1))
  ```

- Klammern in Termen müssen mit doppelten Anführungszeichen geschützt werden:

  ```
  let d="a*(b+c)"
  ```

Aufgabe 4.22 Wenn Gauß einen Computer gehabt hätte ...

Berechnen Sie mithilfe eines Shellskripts die Summe der ersten 100 natürlichen Zahlen (sie benötigen hierfür den Befehl `let` - s.o.):

$$s_n = \sum_{k=1}^{n=100} k \ . \tag{1}$$

Berechnen Sie zum Vergleich s mit der Gaußschen Summenformel

$$s_n = \frac{n(n+1)}{2} \tag{2}$$

im Skript.

Zusatz: Verändern Sie das Skript dahingehend, dass die Zahl n, bis zu der summiert wird, als Argument dem Skript übergeben wird.

Das Programm `bc -l` ermöglicht auch Gleitkomma-Berechnungen innerhalb der Shell:

```
echo "3.4*5 - 10" | bc -l
```

Aufgabe 4.23 Berechnung der Eulerschen Zahl

Berechnen Sie e in einem Shell-Skript mithilfe der Reihe

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \tag{3}$$

Wieviele Terme benötigt man, um e auf 10 Nachkommastellen genau zu bestimmen? Dafür sollte die Anzahl der Terme dem Skript als Argument übergeben werden.

Tipp: Die Zahl e auf 10 Nachkommastellen genau erhält man mittels:

```
echo | awk '{printf "%.10f \n", exp(1)}'
```

Funktionen I

Definition von eigenen Befehlen mittels `alias`:

```
alias ll=ls -lh
```

→ praktisch, aber beschränkt, z.B. Anhängen von & nicht möglich: z.B. `emacs datei &`

Wesentlich flexibeler: Funktionen

bash-Funktionen

```
name ()
{
  ...
}
```

- definiert eine Funktion mit Namen `name`.
- Implementierung erfolgt zwischen den geschweiften Klammern `{ }`
- alternative Schreibweise: `function` *name* `{ ... }`

Funktionen II

An Funktionen können - wie bei einem Shellbefehl - Argumente übergeben werden (Abtrennung durch Leerzeichen).

Aufruf der einzelnen Argumente in der Funktionsdefinition: `$1`, `$2`, ...
Anzahl der Argumente: Variable `$#`

Beispiel

```
saghallo ()
{
 echo "Hallo, $1"
}
```

Aufruf: `saghallo Nutzer`
Resultat: `Hallo, Nutzer`

Hinweis: In Funktionen können Funktionen gerufen werden, sofern diese *vorher* definiert wurden.

Aufgabe 4.24 Einfacherer Editoraufruf

1. Definieren Sie in Ihrem Startskript eine Funktion `em` bzw. `ned`, die das ihr übergebene Argument als Datei im Editor emacs bzw. nedit im Hintergrund (also mittels angehängtem &) öffnet.
2. Falls Sie lieber den Editor `kate` verwenden: Definieren Sie eine gleichnamige Funktion, die `kate` im Hintergrund startet *und außerdem* `kate`s STDERR und STDOUT auf das Nullgerät `/dev/null` umleitet.

Hinweis

Falls Sie eine Funktionsdefinition in nur einer Zeile schreiben, so muss vor der schließenden geschweiften Klammer ein Semikolon oder Ampersand stehen: `; }` oder `& }` .

Aufgabe 4.25 Sichereres Löschen

1. Definieren Sie eine Funktion `rm` in Ihrem Startskript, die statt Dateien zu löschen, diese in das Verzeichnis
   `$HOME/.local/share/Trash/files/`
   verschiebt.
2. Damit diese Dateien auch auf dem Desktop korrekt angezeigt werden, sollten mithilfe der Funktion `mktrinfo` entsprechende Info-Dateien erzeugt werden.
3. Um mehrere Dateien zu löschen, empfiehlt sich eine `for arg`-Schleife. Dabei sollte auch geprüft werden, ob das Argument eine Datei oder eine Option (beginnend mit -) ist.

In der Zählschleife: Überprüfung einer Bedingung im Schleifenkopf (sonst Endlosschleife), z.B.

```
i <= 5
```

Bedingung erfüllt? $\rightarrow$ $\begin{cases} \text{ja} & \rightarrow \text{Abarbeitung des Schleifenkörpers} \\ \text{nein} & \rightarrow \text{Gehe hinter Ende der Schleife} \end{cases}$

Ausführung eines *bedingten Sprungs*

In den meisten Programmiersprachen: bedingte Sprünge explizit möglich durch `if-then-else`-Konstrukte

`if-then-else` in der bash

```
if [ bedingung ]
then
 ...
elif [ bedingung2 ]
then
 ...
else
 ...
fi
```

Die eckigen Klammern `[ ]` sind ein Alias für den bash-Befehl `test`.

Einfaches Beispiel

```
if [ $(whoami) == 'root' ]
then
 echo "Hallo, Meister!"
else
 echo "Watt, wer bist Du denn?!"
fi
```

test bzw. [ ]

gibt 0 (wahr) oder 1 (falsch) als Rückgabewert
Beispiele (mehr auf der Manpage von `test`):

Dateien	
`-e` *datei*	Datei *datei* existiert
`-d` *datei*	Datei *datei* ist ein Verzeichnis

Strings und Ganzzahlen	
t1 `==` *t2*	Strings *t1* und *t2* sind identisch
t1 `!=` *t2*	Strings *t1* und *t2* sind ungleich
a `-gt` *b*	Ganzzahl $a > b$
a `-lt` *b*	Ganzzahl $a < b$

Verknüpfungen von Bedingungen	
bed1 `-a` *bed2*	Bedingungen 1 UND 2 erfüllt
bed1 `-o` *bed2*	Bedingungen 1 ODER 2 erfüllt

Bedingte Ausführung

die Ausführung eines Befehls kann vom Erfolg eines vorhergehenden Befehls abhängig sein:

befehl1 `&&` *befehl2*

→ führt *befehl2* nur aus, falls *befehl1* erfolgreich war

befehl1 `||` *befehl2*

→ führt *befehl2* aus, falls *befehl1* mit einem Fehler endet

Beispiel:

```
gcc -c prog.c && gcc -o prog prog.o -lX11
```

→ Linken nur nach erfolgreichem Compilieren, statt

```
gcc -c prog.c ; gcc -o prog prog.o -lX11
```

Exkurs: Variablen und -substitution

`${var}`	Wert der Variablen; `echo ${HOME}` → `/home/weber/htodt`
`$0`	Name des Programms/Skripts
`$n`	n-tes Argument ($1 \leq n \leq 9$); `file=$1`
`$#`	Anzahl der Argumente; `if [ $# -gt 0 ] ...`

folgende Befehle wirken sich nur auf die *Ausgabe* der Variablen aus:

`${var#Muster}`	kürzester zu *Muster* passender Teil der Variabl. wird am Anfang gelöscht; `echo ${HOME#*/}` → `home/weber/htodt`
`${var##Muster}`	längster zu *Muster* passender Teil der Variablen wird am Anfang gelöscht; `echo ${HOME##*/}` → `htodt`
`${var%Muster}`	kürzester zu *Muster* passender Teil der Variablen wird am Ende gelöscht; `echo ${HOME%/*}` → `/home/weber`
`${var%%Muster}`	längster zu *Muster* passender Teil der Variablen wird am Ende gelöscht; `echo ${HOME%%/*}` → (schneidet alles weg)
`${var:n}`	entfernt die ersten n-Zeichen aus der Variablen
`${var:n:m}`	entfernt ab dem n-ten Zeichen m Zeichen aus der Variablen (1. Zeichen: Index $n = 0$!)

In der csh/tcsh gibt es leider keine Funktionen, aber Schleifen und Verzweigungen wie in der bash, allerdings mit anderer Syntax:

`foreach`-Schleife in der csh/tcsh

```
foreach var (liste)
 ...
end
```

z.B. zeilenweise Ausgabe einer Datei mit Abschneiden des Zeileninhalts vor einem Punkt:

```
foreach line (" `cat $inputfile`")
 echo $line | cut -d. -f2
end
```

Exkurs: Die csh/tcsh-Syntax II

In der `csh/tcsh` wird für Zählschleifen i.d.R. folgendes `while`-Konstrukt genutzt:

Zählschleife in der csh/tcsh

```
@ var = anf
while (bedingung)
 ...
@ var = $var + incr
end
```

z.B. Ausgabe der ersten 10 Zahlen:

```
@ i=1
while ( $i <= 10 )
 echo $i
 @ i = $i + 1  # oder: @ i++
end
```

Die Syntax für Verzweigungen ist sehr ähnlich der der `bash`, man beachte, dass die zu prüfende Bedingung in runden Klammern (`bash`: eckige Klammern) steht und das `then` in derselben Zeile (`bash`: nächste Zeile).

`if-then-else` in der `csh/tcsh`

```
if ( bedingung ) then
 ...
else if ( bedingung ) then
 ...
else
 ...
endif
```

Zusammenfassung

Stärken der Shell (bash/tcsh):

- Dateioperationen (z.B. `cp`, `grep`)
- auf jedem System verfügbar

Nachteile der Shell:

- für mathematische Berechnungen eher ungeeignet
- sehr langsam (Interpreter-Sprache)
- kein grafisches Toolkit
- keine Erweiterungsmöglichkeiten

Weitere Skript-Sprachen I

Einige der Beschränkungen der Shell werden von anderen Skript-Sprachen überwunden

`awk/sed`	erweitern die Shell um String-Manipulationen
`Perl`	Die "Schweizer Offiziers-Kettensäge" der Programmiersprachen
`Python`	objektorientiert, gut lesbar, numpy
`Tcl/Tk`	funktional, mit grafischem Toolkit (Tk), wird z.B. von macPorts genutzt

awk/sed

Beispiele:

```
ls -l | awk '{print $9 " im Besitz von " $3}'
```

druckt nur die 9. und 3. Spalte von `ls -l` zusammen mit dem angegebenen Text aus (Herausschneiden von Substrings, Vertauschen von Reihenfolgen)

```
sed s/falsh/richtig/g datei > datei.neu
```

ersetzt in Datei `datei` den String `falsh` durch `richtig` und schreibt die neue Datei nach `datei.neu`

Perl

- Perl-Einzeiler:

```
out='(16**.25)'
perl -le "print $out"
```

  berechnet die vierte Wurzel aus 16 (während `bc -l` nur Ganzzahl-Exponenten akzeptiert)

- einige Systembefehle sind Perl-Skripte, z.B. `vcnserver` (`head $(which vncserver)`)

Tcl/Tk

Beispiel

→ Blinkender Button: flashbutton.tcl

Python

Beispiel

→ Plotte Sinus-Funktion: sinplot.py