

UNIX - Eine Einführung

(FH Hof ; WI I ; DV-Systeme/Shell ; Prof. Dr. Köhler
Version: 1.00 (WS 97/98) Tobias Ott - st0278)

Dieses Skript unterliegt der GNU General Public License, deren Text im Internet nachzulesen ist. Das Copyleft liegt bei dem(n) Autor(en).

Begriffe:

UNIX = wurde Ende der 60er Jahre in den *Bell Laboratories* entwickelt

SHELL = Kommandointerpreter (mit klarer Kommandosprache, die u.a. viele Eigenheiten einer Programmiersprache aufweist)

C = Unix ist stark mit der Programmiersprache *C* verbunden, da bis auf einen Kern von ca. 5% das gesamte Betriebssystem in *C* geschrieben ist.

AT&T = Unix steht unter der Lizenz von *AT&T*, wird aber von vielen weiteren Herstellern unter anderen Namen angeboten (z.B. *Sinix*, *Xenix*, *Solaris*, *Linux*)

BSD = Eine Unix-Distribution der *Berkley Software Distribution*

VT100 = Terminalemulation, als Standard eine gute Wahl

Rlogin = Remote Login, d.h. Login (=Anmelden) über ein Netzwerk

Weiteres: *System V.4* ist der Industrie Standard; Die *Open Group* besitzt das TM von Unix (*OSF*, *X-Open*)

- Besonderheiten von Unix:
- **Swapping** (Prozeß wird vom Hauptspeicher auf die Festplatte ausgelagert) und **Paging** (Ein Teil eines Prozesses und Hauptspeicher werden in gleich große Einheiten (pages) aufgeteilt (typisch 4 KB pro Page))
 - **Multiuser** (=viele Benutzer gleichzeitig) und **Multitasking** (mehrere Prozesse laufen synchron und asynchron ab)
 - **Scheduling** oder **Time-Sharing** (Rechenzeitaufteilung in „Zeitscheiben“)
 - **Pufferung, Caching**
 - **Toolbox-System** (viele versch. Dienstprogramme (Tools))
 - **hierarchisches Dateisystem** (ausgehend von / , dem sog. root-Verz.)
 - **lange Dateinamen** (255 Zeichen; neue Systeme mehr!)
 - **case-sensitive** (Groß- und Kleinschreibung ist zu beachten!)

Möglichkeiten des Einloggens über TCP/IP: ftp, telnet (rlogin)

Hostname in der FH-Hof: *mail* oder *fhh-rz3.fh-hof.de*

man <Stichwort>

Gibt die Hilfe zu diesem Stichwort aus

Sektionen 1-8 gelten auch für das man-Kommando!

NAME	Befehlsname
SYNOPSIS	Wie muß der Befehl aufgerufen werden?
<RETURN>	scrollt Zeilenweise
<SPACE>	scrollt Seitenweise
<PGUP>	Seite nach oben
<PGDOWN>	Seite nach unten
<q>	verläßt die Hilfe

Beispiele: man write
man 2 write

yes <Zeichenkette>

zeigt die Zeichenkette endlos auf dem Bildschirm an

sleep <Zeitintervall>

Der aktuelle Prozeß schläft für <Zeitintervall> Sekunden
Ctrl + C brincht den schlafenden Prozeß ab!

mail

Kontrolliert und Verwaltet Email Funktionen

pine

Ein weiteres Email Programm

To: Adresse des Empfängers
Cc: Kopie der Mail an folgende Adressen
Att: Datei die evtl. angehängt werden soll
Sub: Betreff

write <User>

Schreibt eine Message an den mit <User> angegebenen Benutzer

talk <User>

Plaudern mit einem anderen Benutzer

cat <Dateiname>

zeigt die Datei an (Strom von Bytes an die Standardausgabe)

more <Dateiname>

zeigt die Datei Seite für Seite an
Ctrl + F vorwärts
Ctrl + B rückwärts

tail <Dateiname>

Gibt nur die letzten Zeilen einer Datei aus

tail -n <Dateinamen>

Gibt nur die letzten n Zeilen einer Datei aus

head <Dateiname>

Gibt nur die ersten Zeilen einer Datei aus

wc <Dateiname>

Zählt Zeilen, Worte, Zeichen einer Datei

cp <Quelle> <Ziel>

Kopiert eine Datei

Dateizugriffsrechte:

user	group	other
rwxs	rwxs	rwxs

r = read w=write x=execute s=superuser

chmod u|g|o +/- r|w|x Ändert die Dateizugriffsrechte
 z.B. chmod u+r testdatei.txt

chown Ändert den Eigentümer der Datei (nur root!)

chgrp Ändert die Gruppenrechte der Datei

vi <Dateiname> Startet den Editor vi, optional mit einem Dateinamen

nach dem Start ist der vi im Kommandomodus!

:q	beendet vi
:q!	beendet ohne zu speichern
:wq	beendet und speichert
:i	Einfügemodus
:a	fügt an aktuelle Position ein
:A	fügt am Zeilenende ein
ESC	beendet den jeweiligen Mode

Ctrl + F	eine Seite vorwärts
Ctrl + B	eine Seite rückwärts
Ctrl + D	Down
Ctrl + U	Up

/ (Slash) schaltet in den Suchmodus

file * zeigt die Dateitypen und Texttypen an

ps Zeigt Informationen zu den einzelnen Prozessen

Einschub Prozesse:

Unix-Programme erzeugen Prozesse (sogenannte Task's).

Jeder dieser Prozesse besitzt:

- eine eindeutige Prozess-ID (PID)
- vom Betriebssystem verwaltete Ressourcen (z.B. Speicher), die vom Zugriff anderer Prozesse geschützt sind.

Neue Prozesse werden im allg. von schon laufenden Prozessen aus gestartet (z.B. einer Shell).
 (Hinweis auf C-Aufrufe: *fork()*, *exec()*)

Jeder Prozeß ist normalerweise mit drei „Geräten“ (sog. *devices*) standardmäßig verbunden:

- Eingabe (Tastatur) 0
- Ausgabe (Bildschirm) 1
- Fehlerausgabe (Bildschirm) 2

Die Descriptoren 0,1,2 werden u.a. in C-Programmen bei *read* oder *write* benötigt.
In C auch: *stdin*, *stdout*, *stderr*.

Jeder Prozeß liefert nach seiner Beendigung einen Return-Code zurück.

In C lauten die Befehle: `exit(0);` oder
 `return 5;` oder
 `return (5);`

Diese Kanäle können unter Unix mit den folgenden Zeichen umgeleitet werden:

< (Eingabe)
> (Ausgabe)
2> (Fehlerausgabe)

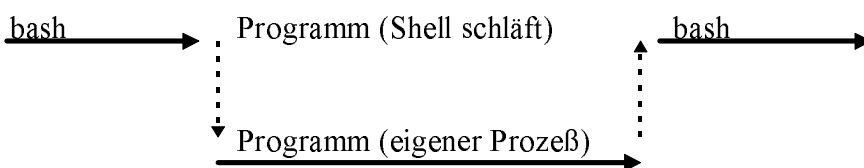
z.B. `mail st000@fh-hof.de <datei.txt`
 `ls /tmp >datei.txt` **TESTEN!**
 `cc hallo.c 2>error.log`
 `cc hello.c <datei 2>&1` (was aus 2 wäre geht hier auf Kanal 1)

Das Symbol „|“ lenkt die Ausgabe eines Prozesses auf die Eingabe eines anderen um. Auch hier zwei Beispiele:

`ls /tmp | more`
`ls /tmp | wc`

Einschub Shell:

Die Shell ist meist ein „normales“ C-Programm. Prinzipiell kann die Shell auch in einer anderen Programmiersprache geschrieben werden.



z.B.



Mit & („Ampersand“) kann man Prozesse im Hintergrund starten.
(„asynchron“) ==> Die Shell wartet nicht auf die Beendigung des Prozesses bzw. Programms

z.B. `sleep 60 &` Als Ausgabe wird die PID zurückgeliefert. Der Prozess läuft im Hintergrund. Bei Beendigung wird der Abschluß bestätigt.

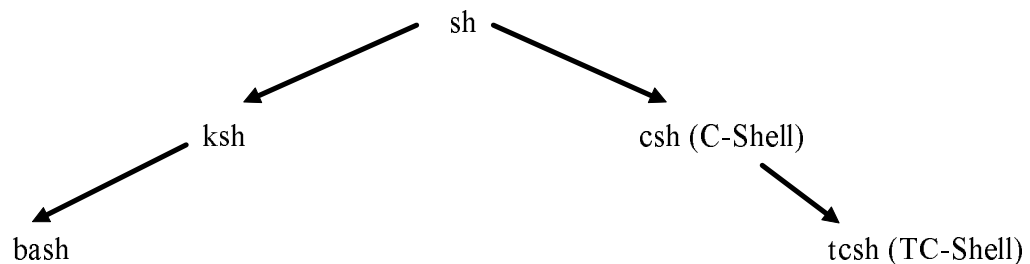
startx & X-Windows wird im Hintergrund gestartet

cc hello.c >hello.log 2>hello.err & siehe oben!

Die sh (bourne-Shell) gibt es auf jedem Unix-Rechner. Sie hat eine eingebaute Programmierbarkeit, ist jedoch im allg. nicht besonders komfortabel.

Die ksh (korn-Shell) ist neueren Datums und z.B. eine integrierte History-Funktion usw. Sie ist schon um einiges komfortabler als die bourne-Shell.

Die bash (bourne again Shell) ist relativ neu und z.B. in LINUX enthalten. Sie ist sehr komfortabel.



Bemerkungen:

- Die Shell ist ein Programm welche interne Befehle (wie z.B. echo, exit, cd, usw.) enthält und die externe Programme bzw. Befehle aufruft. Als erstes wird der eingegebene Befehl unter den internen Befehlen gesucht, dann werden die unter \$PATH genannten Pfade durchsucht (von links nach rechts!)
- Die *bash* ist insgesamt am kompatibelsten
- *tc/tk* ist eine Skriptsprache, die auch als Shell verwendbar ist
- *perl* ist ebenfalls eine *Skriptsprache*

weitere Befehle:

- | | |
|---------------|--|
| kill -9 <PID> | Beendet den Prozeß todsicher! (-9)
Dieser Parameter beendet sogar die bash! Programme können diesen kill-Befehl nicht abfangen! |
| kill <PID> | Sendet ein Löschesignal an den Prozeß. Es gibt jedoch Prozesse die dieses Signal abfangen können! |
| bash | startet die bash-Shell |
| exit | beendet die Shell |

Aufbau von Shell Skripten:

Die erste Zeile kann wie folgt lauten:

```
#!/bin/sh          (oder irgendeine andere Shell!)
```

Dieser Befehl startet die Shell und arbeitet dann das gesamte Skript in dieser Shell ab. Ohne diese Zeile wird die Standardshell benutzt.

```
#                Ist ein Kommentar  
<Befehlsfolge>+  Ist ein oder mehrere Befehle  
exit 5           verläßt das Shell-Skript mit dem Return-Code 5
```

Zum Ausführen von Shell-Skripten stehen folgenden 3 Möglichkeiten zur Verfügung:

1. sh hello oder: bash hello (shellname skriptname)
2. chmod u+x hello (das Skript kann dann wie ein Programm aufgerufen werden!)
3. . hello (Punkt vorangestellt!)

Beispiele und Aufgaben zu Shell-Skripten:

Schreibe drei Shell-Skripte SK1, SK2, SK3 und vergleiche ihre Ausgaben, sowie das Verzeichnis am Ende des Ablaufs! Probiere dabei die 3 oben genannten Möglichkeiten zum Ausführen des Skripts!

```
SK1:           #!/bin/sh  
               ps
```

```
SK2:           #!/bin/bash  
               ps
```

```
SK3:           cd /tmp  
               ps  
               pwd
```

=> Bei *sh SK3* und *chmod u+x SK3*; *SK3* sind Verzeichnisänderungen nicht dauerhaft. Die Shell übergibt das Kommando an *SK3*; *SK3* wird in der in Zeile 1 genannten Shell ausgeführt; nach der Ausführung gibt das Shellskript die Kontrolle wieder an die Shell von der aus aufgerufen wurde!

Bei *. SK3* ist die Ausführung ähnlich einer interaktiven Eingabe. Der Prozeß liest die Eingabedaten für die aktuelle Shell aus der Shell-Skript-Datei.

Merke: *Prozesse* und *Shell-Skripte* können *Return-Codes* zurückliefern!

weitere Befehle für Shell-Skripte:

\$?	liefert den Return-Code des letzten Kommandos z.B. sh exit 5 echo \$?
\$!	Prozeß-ID des zuletzt gestarteten Hintergrundprozesses
\$\$	Prozeß-ID der aktuellen Shell
\$TERM	Gibt den Namen der Terminalemulation wieder, die im Moment aktiv ist (vt100 ist hier eine gute Wahl, wird kein Wert zurückgegeben sollte einer in gesetzt werden!) zwei weitere wichtige Umgebungsvariablen sind: \$PATH und: \$HOME
TERM=vt100	setzt den Wert für Umgebungsvariable TERM
set	zeigt die Werte aller Umgebungsvariablen an
set XTEST=abcde	legt die Umgebungsvariable XTEST an und belegt sie mit abcde
unset XTEST	löscht die Umgebungsvariable XTEST
type <Kommando>	gibt den Pfad zurück, wo die Shell das Kommando <Kommando> findet
which <Kommando>	wie <i>type</i> , jedoch kürzere Ausgabe
ping <Hostname>	schaut nach ob es den Host <Hostname> gibt

folgende Positionsparameter werden an das Shell-Skript übergeben:

\$0	Name des Shell-Skripts
\$1	1. Übergabeargument
...	
\$9	9. Übergabeargument
\$#	Anzahl der Übergabeparameter
\$*	alle Argutmente als eine Zeichenkette
\$@	alle Argutmente als eine Zeichenkette

pattern matching (Interpretation von Dateinamen durch die Shell):

- * steht für beliebig viele Zeichen (0,1,2,...)
- ? jedes Fragezeichen steht für genau ein Zeichen

Beispiele:

ls a*	zeigt alle Dateien die mit a anfangen und irgendwie weitergehen
ls a*b	a am Anfang, b am Ende und dazwischen versch. viele Zeichen
ls aa*b	aa am Anfang, b am Ende und dazwischen versch. viele Zeichen
ls a?	a am Anfang, 2. Zeichen egal, jedoch nur 2 Zeichen lang!
ls a??	a am Anfang, 2 und 3. Zeichen egal, jedoch genau 3 Zeichen lang!
ls ?a*	
ls ??*	
ls a[bc]	ab oder ac (das in [] Klammern stehende steht als Auswahl für das 2. Zeichen)
ls a[!bc]	a am Anfang <u>und nicht</u> b und <u>nicht</u> c an der zweiten Stelle
ls a[!bc]*b?	

Bei dem folgenden *find* Befehl kann das *pattern matching* u.a. eingesetzt werden:

```
find <Startverzeichnis> -name '<Suchbegriff>' -print
```

z.B.

```
find /usr/lib -name 'libc*.a' -print  
find /etc -name 'p*' -print 2>/dev/null
```

Ab <Startverzeichnis> wird mit dem Suchen begonnen, <Suchbegriff> ist der Suchstring und -name und -print werden als Option benötigt (-print evtl. nur auf älteren Systemen!)

Wichtige devices (Gerätetreiber):

/dev/null	ist der „Papierkorb“ ; alles was hierhin umgeleitet wird, verschwindet
/dev/fd*	Die Dateien /dev/fd* repräsentieren die Diskettenlaufwerke, hier genügen im Normalfall /dev/fd0 und /dev/fd1 (fd = Linuxspezifisch!)
/dev/lp*	/dev/lp* steht für die parallelen Druckerschnittstellen im System z.B. <i>cat <Datei> > /dev/lp1</i> um die Datei <Datei> auf den Drucker umzuleiten

wieder ein paar Befehle:

sort	sortiert Eingaben bzw. Dateien alphanumerisch
sort -r	kehrt die Reihenfolge um, sonst siehe sort!
who sort	sortiert die Ausgabe von <i>who</i> alphanumerisch
ls -l sort +1	sortiert die Ausgabe von <i>ls</i> nach Kriterium 2. Spalte

sort -b +4	sortiert nach 4. Spalte, <i>-b</i> ignoriert führende Leerzeichen (gut für Strings!)
sort -n	sortiert nach 1. Spalte, <i>-n</i> steht für numerisch (gut für Zahlen!)
sort -t:	Doppelpunkt ist Feld- bzw. Spaltentrenner
cut	schneidet bestimmte Teile aus den Zeilen einer Datei

Etliche Zeichen der Shell sind Sonderzeichen, d.h. sie erhalten eine gesonderte Bedeutung:

- *, ?, [] bei *pattern matching*

- \$ Parametersubstitution, z.B.:

TERM=vt100 ; echo \$TERM	
echo \$?	(RC des letzten Bef.)
echo \$\$	(PID d. aktuellen Shell)
echo \$!	(PID d. letzten Hintergrundprozesses)

- `` (ASCII 096) Kommandoersetzung, z.B.:

`<Kommando>`	das Kommando wird ausgeführt und die Ausgabe des Kommandos wird zwischen den beiden `` eingesetzt.
DATUM=`date`	date wird ausgeführt; Ausgabe wird eingesetzt; => Die Variable DATUM enthält jetzt das aktuelle Datum (Sat Dez 19); echo \$DATUM gibt den Inhalt aus; D=\$DATUM belegt die Variable D mit dem Inhalt der Variable DATUM.

- | Pipe-Zeichen (Ausgabe eines Kommandos wird direkt in die Eingabe eines anderen Kommandos umgeleitet), z.B.:

ls -l `find /usr -name vi`	<=>	ls -l /usr/bin/vi
----------------------------	-----	-------------------

ls | more

- & (Ampersand) startet einen Hintergrundprozeß, z.B.:


```
sleep 5 &
sort Adressen | uniq > Adressenliste &
```

- ; erzeugt eine Kommandoliste (die Kommandos werden nacheinander ausgeführt); z.B.:


```
sleep2 ; cd /tmp/ ; pwd ; cd
```

- { <Kommandoliste>; } die geschweiften Klammern fassen Kommandos zusammen. Die so zusammengefassten Kommandos können als Gesamtheit in den Hintergrund geschickt werden um dort nacheinander abgearbeitet zu werden (Bsp. 1). Weiterhin können Ein-/Ausgabekanäle umgelenkt werden (Bsp. 2).
Beachte: die beiden Leerzeichen am Anfang und am Ende, sowie den Strichpunkt (;) am Ende!
z.B.:

```
{ sleep 5 ; echo Hallo ; } &  
{ pwd ; echo Hallo ; date ; } >ausgabe.txt
```

- (<Kommandoliste>) Ausführung der Kommandos in einer separaten Shell, z.B.:

```
(sleep 3 ; echo Hallo ; ps ; exit 5)  
(sleep 3 ; echo Hallo ; ps ; exit 5) &  
(...) >test.dat  
(exit 127) ; echo $? (liefert nur den Return-Code 127 (ESC) zurück)
```

- && Bedingte Kommandoausführung (UND): <Komm. 1> && <Komm. 2>
Kommando 2 wird nur dann ausgeführt, wenn Kommando 1 einen Return-Code von 0 (Null) zurückliefert, also fehlerfrei abgeschlossen wurde. z.B.:

```
(exit 0) && echo fertig!  
(exit 1) && echo fertig!
```

- || Bedingte Kommandoausführung (ODER): <Komm. 1> || <Komm. 2>
Kommando 2 wird genau dann ausgeführt, wenn Kommando 1 einen Return-Code ungleich 0 (Null) zurückliefert. z.B.:

```
(exit 0) || echo fertig!
```

Quoting (Entwerten von Sonderzeichen!)

- \ (Backslash) hebt die Sonderbedeutung des nachfolgenden Sonderzeichens auf, z.B.:
(Ein entwertetes Zeilenende wird ignoriert!)

```
echo $TERM  
echo \$TERM  
echo `pwd`  
echo \  
echo \<Return> (entspricht new line)
```

- „“ (Anführungszeichen) Von den in Anführungszeichen eingeschlossenen Worten erkennt die Shell nur die Sonderzeichen \, \$ und ` als solche, z.B.:

```
echo „ `pwd` \  
$TERM“
```

- ' (Hochkomma) Alle Sonderzeichen bis auf ' (Hochkomma=Quote) verlieren ihre Sonderbedeutung.

wieder einige neue Befehle:

tar (tape archiver) verwaltet Dateiarhive

tar -xf <Dateiname.tar> (x=extract, f=file) extrahiert Dateien aus einem Archiv

grep sucht nach bestimmten Ausdrücken und gibt die Zeile aus in welcher der Ausdruck gefunden wurde

grep <Option> <Ausdruck> <Datei(en)> Optionen:

- n Zeilennummer
- l nur Dateinamen
- i ignoriert Groß-/Kleinschreibung
- c zählt nur Häufigkeit des Vorkommens
- v nur Zeilen ohne <Ausdruck>

z.B.:

- grep ' cafe ' *
- grep '^cafe' * ^ = am Zeilenanfang!
- grep 'cafe\$' * \$ = am Zeilenende!
- grep „[ca][af]fe“ * *pattern matching* mit []!
- grep Cafe * ; echo \$? Return-Code 0 oder 1

shift verschiebt die Postionsparameter, z.B.:

shift \$1 entfällt, die restl. rücken um nach vorne (\$2 wird zu \$1, \$3 zu \$2, usw.)

shift 3 verschiebt um 3 Stellen

set Gibt alle Umgebungsvariablen aus

<Variable> = <Wert> setzt den Wert einer Umgebungsvariable, z.B.:

set \$1=Wert1 ; echo \$1

Erweiterung für Shellskripte (Eingabe, Schleifen, Bedingungen, ...)

`read <Variable> [<Variable>]` liest Werte für Variablen von der Tastatur ein, z.B.

```
read a b          => 12 3 4 5 6
echo $a           => 12
echo $b           => 3 4 5 6
```

`:` Kommando tut nichts, gibt jedoch 0 (Null) als Return-Code zurück!
Man kann dies zum Anlegen einer leeren (Länge 0 Bytes) Datei nutzen:

```
:>Leerdatei.dat
```

`true` liefert den Return-Code 0 (exit 0)

`false` liefert den Return-Code 1 bzw. ungleich 0 (exit 1)

`expr` Rechnet mit ganzen Zahlen (als Rechenoperatoren sind zulässig: + (Plus), - (Minus), '*' bzw. „*“ (Mal), / (Geteilt), % (Modulo))

Hier einige Beispiele:

```
expr 1 + 2          => 3
a=`expr $a + 1`     => erhöht den Wert der Variable a um 1
                    (Leerzeichen beachten!)
```

`if <Liste1>`
`then <Liste2>`
`elif <Liste3>`
`then <Liste4>`
`...`
`else <Liste4>`
`fi`

<Liste *n*> ist eine durch ; oder eine neue Zeile getrennte Kommando-
folge (die selbst auch wieder aus Bedingungen usw. bestehen kann).
if und elif führen den then Zweig aus, wenn der Ret.Code von <Liste *n*>
gleich 0 (Null, True) ist. Ist der letzte elif-Teil nicht gleich 0 (Null, True)
so wird der abschließende else Teil abgearbeitet.
Der Return-Code ist gleich dem des letzten Kommandos, oder Null, wenn
kein Kommando ausgeführt wurde.

z.B.: `if true ; then echo 0 ; else echo 1 ; fi`
`if false ; then echo 0 ; else echo 1 ; fi`

```
if [ -d . ]          => siehe test Kommando
then
  echo Verzeichnis
else
  echo kein Verzeichnis
fi
```

for v in a b c	in \$* (wenn man nichts schreibt), v ist die Laufvariable!
do <Liste>	<Liste> z.B. echo \$v
done	
while [\$# -ne 0] do echo \$1 shift done	Ausführung der Schleife solange Bedingung nicht erfüllt! Der geklammerte [] Ausdruck entspricht dem folgenden <i>test</i> Kommando: <i>test \$# -ne 0</i> Testet ob der Wert von \$# not equal Null ist
while true done	Stellt eine Endlosschleife da, die u.U. mit break verlassen werden muß, oder Ctrl.+C unterbrochen werden muß
break	beendet bzw. verläßt die aktuelle for, until oder while-Schleife
continue	beginnt mit einem neuen Schleifendurchlauf (bei for, until und while-Schleifen)
case ab in b*) echo Fall 1;; a*) echo Fall 2;; *) echo trifft immer zu ;; esac	Verzweigung mit case; falls z.B. Argument b*) zutrifft werden die nachfolgenden Befehle ausgeführt. Beachte:) - schließende Klammer und ;; - Doppelsemikolon !
case ab in a* b*) <Anweisung(en)> esac	...wenn a oder b zutrifft
<Funktionsame> () { <Befehlsliste> }	Definiert eine Funktion mit Namen <Funtionsame> welche die in <Befehlsliste> enthaltenen Befehle ausführt.

und wieder ein paar neue Befehle:

test <Ausdruck> [<Ausdruck>] dient zum testen von beliebigen Ausdrücken, z.B.:

test -d .	=> prüft ob das aktuelle Verz. vorhanden ist
[-d .]	=> analog, nur andere schreibweise!
test -r <Datei>	=> prüft ob die Datei lese-Rechte hat
test -w <Datei>	=> prüft ob die Datei schreib-Rechte hat
test -x <Datei>	=> prüft ob die Datei ausführbar ist
test -z „“	
test -z „a“	
test -z „b“ -a -d	=> -a ist eine Verknüpfung mit AND
test -z „b“ -o -d	=> -o ist eine Verknüpfung mit OR
test abc=abc	
test „\$VAR“=abc	
test 1 -eq 2	=> prüft ob 1 gleich 2 ist
test 1 -ne 2	=> prüft ob 1 ungleich 2 ist
test 1 -lt 2	
test 1 -gt 2	
test 1 -le 2	
test 1 -ge 2	

set -x
set +x

schaltet den Debug-Mode ein
schaltet den Debug-Mode aus

Anhang A (Eine Beispielschell in C, nicht optimiert und nicht ganz funktionsfähig!)

Folgendes Programm ist eine „primitiv“-Shell in C:

```
#include <stdio.h>
void main(void)
{
int i;
char eingabe;
while(i=0)
{
printf(„Shellprompt:>„);
gets(eingabe);
system(eingabe);
}
}
```

Das Programm ist jedoch keine „Shell“ im eigentlichen Sinn, sondern läuft nur unter einer anderen Shell (wg. `system`-Aufruf!). Soll eine „richtige“ Shell programmiert werden, ist folgendes anzumerken:

Die `system()` Aufrufe können auch über `fork()` (verzweigt Prozesse) und `exec()` (startet neue Kommandos) gelöst werden!

Return-Codes am Ende des Programms sollten bei C mit `exit(1);` bzw. `exit 1;`
und bei C++ mit `return(3);` bzw. `return 0;`
übergeben werden!

Anhang B (Kurzeinführung ftp)

ftp (file transfer protocol)

Das ftp-Protokoll setzt wie auch telnet, mail oder www auf das TCP-Protokoll auf.

TELNET, FTP, WWW, MAIL
TCP
IP

Das ftp Kommando startet den ftp-Dienst, welcher zur Dateiübertragung genutzt wird. FTP überträgt dabei die Datei byteweise.

Aufruf: ftp <Hostname> , wobei <Hostname> z.B. durch
ftp.uni-bayreuth.de
195.95.193.60
pc1-1115
ftp.mail.fh-hof.de

ersetzt werden kann.

Standardlogins für ftp:

Name: ftp	Passwort: <User>@<Domain>
Name: ftp	Passwort: ftp
Name: guest	Passwort: anonymous
Name: guest	Passwort: guest

einige Befehle:

get <Datei>	holt eine Datei vom Zielrechner <Hostname>
put <Datei>	überträgt eine Datei zum Zielrechner <Hostname>
?	gibt eine Hilfe aus
prompt	schaltet Nachfrage-Option bei Mehrdateiübertragung ein aus
mget <Dateiname>	überträgt mehrerer Dateien (z.B. mget *)
mput <Dateiname>	-analog-

Anhang C (was mir so noch eingefallen ist ...)

Aufbau /etc/passwd : ... :x:USERID:GRUPPENID:NAME ...

gs -sDEVICE=epon <Datei.ps>

| als Pipe für z.B. ls | more , ls | wc -l

PORTS: siehe auch: socket.h
 unsigned-Typ in C
 /etc/services enthält die definierten Ports
 bis 1024 gehen die versch. Portnummern
 netstat
 Port Nr. xx ---> starte News-Server (BBS-System, ...) auf diesem Port
 Internet Programmierung

Wann ist ein Jahr ein Schaltjahr? Die Jahreszahl muß durch 4 teilbar sein!
Die Jahreszahl darf aber nicht durch 100 teilbar sein!
Ist die Jahreszahl durch 400 teilbar ist das Jahr trotzdem ein Schaltjahr!

kurz: /4 => Schaltjahr
 /100 => kein Schaltjahr
 /400 => doch ein Schaltjahr

Anhang D (Dank für Korrektur und Fehlerbeseitigung)

Prof. Dr. Köhler
Stefan Lapenat
Markus Haußner