

Shell-Programmierung

In diesem Teil der Veranstaltung wird gezeigt, wie mit der Shell programmiert werden kann. Dazu wird betrachtet:

Erzeugen und Ausführen von Shell-Programmen

Variablen

Kontrollstrukturen

Debuggen

Modifikation der Login Umgebung

Funktionen

Inhalt

1. Erzeugen und Ausführen von Shell-Programmen.....	4
2. Erzeugen eines bin-Verzeichnisses für ausführbare Dateien.....	6
3. Variablen.....	9
3.1. Positionsparameter.....	9
3.2. Spezialparameter.....	12

3.3. Variablen mit Namen.....	13
3.4. Wertzuweisung an Variablen.....	16
4. Shell Kontrollstrukturen.....	21
4.1. Kommentare.....	21
4.2. Here document.....	22
4.3. Return Codes.....	27
4.4. Schleifen.....	29
4.4.1. for.....	29
4.4.2. while.....	33
4.4.3. Die Ablage /dev/null.....	35
4.5. Bedingte Ausführung.....	36
4.6. test Kommando.....	37
4.7. case-esac.....	45
4.8. break, continue.....	47
4.9. exit.....	49
5. Fehlerbehandlung der Shell.....	52

6. Debuggen von Shell-Programmen.....	55
7. Anpassen der Login Umgebung.....	62
8. Funktionen.....	63
9. Ausführen ohne neuen Prozess zu erzeugen.....	65

1. Erzeugen und Ausführen von Shell-Programmen

Gehen wir von einer Datei "dl" mit folgendem Inhalt aus:

```
$ cat dl
pwd
ls -l
echo Dies ist das letzte Kommando des Shellprogramms
$
```

Um ein Shellprogramm auszuführen, kann man der Shell das Programm als Parameter angeben.

```
$ sh dl
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
insgesamt 4
-rw-r--r--  1 as      users      64 Jan 15 10:17 dl
Dies ist das letzte Kommando des Shellprogramms
$
```

Einfacher kann man es machen, indem man die Datei ausführbar macht:

```
$ chmod +x dl
$ dl
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
insgesamt 4
-rwxr-xr-x  1 as      users          64 Jan 15 10:17 dl
Dies ist das letzte Kommando des Shellprogramms
$
```

2. Erzeugen eines bin-Verzeichnisses für ausführbare Dateien

Um Shellprogramme von beliebiger Stelle im Dateisystem ausführen zu können, sollte man ein eigenes Verzeichnis dafür anlegen und dieses Verzeichnis in den Pfad einfügen.

```
$ mkdir $HOME/bin
$ mv dl $HOME/bin
$ PATH=$PATH:$HOME/bin
$
$ cd /tmp/
$ dl
.....
$
```

Die (vordefinierte) Variable HOME gibt das Heimatverzeichnis an, die Variable PATH definiert den Suchpfad, der verwendet wird, um nach ausführbaren Dateien zu suchen.

Achtung

Man kann Shellprogrammen beliebige Namen geben. Vermieden werden sollten Namen, die schon von Systemprogrammen verwendet werden, da dies unerwünschte Effekte haben kann:

Systemkommandos sind nicht mehr wie gewohnt zugänglich.

Es können Unendlichschleifen entstehen.

Dies zeigt folgendes Beispiel:

```
$ mv dl date
$ PATH=./:$PATH
$ date
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
insgesamt 4
-rwxr-xr-x    1 as          users          64 Jan 16 10:00 date
Dies ist das letzte Kommando des Shellprogramms
$
```

Nun kann das Unix date-Kommando nicht mehr wie gewohnt eingegeben werden – man muss jetzt den vollen Pfadname angeben, um Unix date auszuführen.

```
$ /bin/date
Mit Jan 16 10:04:59 CET 2002
$
```

Unendlichschleifen verdeutlicht folgendes Beispiel:

```
$ mv dl ls
$ cat ls
pwd
ls -l
echo Dies ist das letzte Kommando des Shellprogramms
$ ls
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
.....
```

Dies kann verhindert werden, in dem man im Skript den vollen Pfadnamen angibt.

3. Variablen

Variablen sind die grundlegenden Datenobjekte, die außer Dateien von Shellprogrammen verändert werden können.

Zu unterscheiden sind:

- Positionsparameter (positional parameter),

- Spezialparameter (spezial parameter) und

- Variablen mit Namen (named variables)

3.1. Positionsparameter

Ein Positionsparameter ist eine Variable in einem Shellprogramm, deren Wert durch ein Argument bei einem Programmaufruf gesetzt wird.

Positionsparameter werden durch

- \$1, \$2, \$3 usw.

angesprochen. In einem Shellprogramm sind bis zu 9 solcher Parameter verwendbar.

```
$ cat pp
echo 1. Positionsparameter:    $1
echo 2. Positionsparameter:    $2
echo 3. Positionsparameter:    $3
echo 4. Positionsparameter:    $4
$
$ pp eins zwei drei vier
1. Positionsparameter: eins
2. Positionsparameter: zwei
3. Positionsparameter: drei
4. Positionsparameter: vier
$
```

Folgende Beispiele verdeutlichen Positionsparameter:

Senden eines Geburtstaggrüßes an einen Benutzer, der beim Aufruf angegeben wird:

```
$ cat bday
echo happy birthday | mail $1
$
$ bday as
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/mail/as": 1 message 1 new
>N 1 as@linux.ki.fbi.fh-d Wed Jan 16 12:12 12/475
& 1
Message 1:
From as@linux.ki.fbi.fh-darmstadt.de Wed Jan 16 12:12:37 2002
Date: Wed, 16 Jan 2002 12:12:37 +0100
From: Alois Schütte <as@linux.ki.fbi.fh-darmstadt.de>
To: as@linux.ki.fbi.fh-darmstadt.de

happy birthday

& d
& q
$
```

Anzeigen der Login-Information des beim Aufruf angegebenen Benutzers:

```
$ cat whoson
who | grep $1
$
$ whoson as
as pts/4 Jan 16 12:04 (trex.fbi.fh-darmstadt.de)
$
```

3.2.Spezialparameter

Durch zwei Spezialparameter

`$#`

`$*`

kann auf die Anzahl der Argumente und auf den gesamten Argumentestring zugegriffen werden.

```
$ cat get.num
echo Anzahl der Argumente: $#
$
$ get.num Test des Programms
Anzahl der Argumente: 3
$
```

```
$
$ cat show.param
echo Die Parameter sind: $*
$
$ show.param Test des Programms 4 5 6 7 8 9 0 11 12
Die Parameter sind: Test des Programms 4 5 6 7 8 9 0 11 12
$
```

3.3.Variablen mit Namen

Eine selbst definierte Variable kann durch

```
Variable=Wert
```

definiert und initialisiert werden. Dabei darf links und rechts vom Gleichheitszeichen **kein** Leerzeichen stehen.

Der Wert einer Variablen wird durch dem Namen vorgestelltes \$-Zeichen ausgedrückt.

Ein Variablenname muss mit einem Buchstaben oder einem Unterstrichstrich beginnen. Der Rest kann aus Buchstaben, Ziffern und Unterstrichstrichen aufgebaut sein. Kommandonamen sind als Namen für Variablen nicht erlaubt.

Vordefinierte Variable können nicht zur eigenen Namensgebung verwendet werden. Folgende (nicht vollständige) Liste beschreibt vordefinierte Variablen:

CDPATH definiert den Suchpfad des cd-Kommandos

HOME beschreibt das Homeverzeichnis

IFS gibt die Trennzeichen ein (internal field separators: space, tab, carriage return)

LOGNAME hat als Wert den Login-Name

PATH ist der Suchweg der Shell zum Finden von ausführbaren Dateien

PS1, PS2 sind die Promptzeichen

TERM definiert den Typ des Terminal

TZ beschreibt die Zeitzone

DISPLAY ist das Display in X-Window-Umgebungen

Den Wert aller Variablen zusammen mit ihre Namen kann man durch das Kommando *env* auflisten.

```
$ env
PWD=/users/as/Vorlesungen/Unix/Skript/Shellprogrammierung
PAGER=less
REMOTEHOST=trex.fbi.fh-darmstadt.de
HOSTNAME=linux
MANPATH=/usr/local/man:/usr/share/man:/usr/X11R6/man:/usr/openwin/man
MACHINE=i686-suse-linux
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
MAIL=/var/mail/as
OLDPWD=/users/as
LANG=de_DE@euro
HOST=linux
INFOPATH=/usr/local/info:/usr/share/info:/usr/info
DISPLAY=141.100.42.131:0.0
LOGNAME=as
SHELL=/bin/bash
PRINTER=lp
HOSTTYPE=i386
WINDOWMANAGER=/usr/X11R6/bin/kde
TERM=xterm
HOME=/users/as
XNLSPATH=/usr/X11R6/lib/X11/nls
no_proxy=localhost
PATH=/users/as/bin:/usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/usr/lib/java/bin
:/usr/games/bin:opt/kde2/bin:
$
```

Den Wert einer Variablen kann man mit dem echo Kommando einfach ausgeben:

```
$ echo $LOGNAME
as
$
```

3.4. Wertzuweisung an Variablen

Wenn man mit dem vi arbeitet, muss die Variable TERM einen Wert enthalten, aus der der benutzte Terminaltyp ableitbar ist. Ein oft verwendeter Typ ist "vt100".

Einer Variablen kann man einen Wert zuweisen:

```
$ TERM=vt100
$
```

Neben der direkten Zuweisung eines Literals kann eine Variable auch Werte erhalten durch

- read Kommando

- Umlenkung der Ausgabe von Kommandos in Variablen

- Zuweisen von Werten anderer Variable

read Kommando

Um eine Benutzereingabe einzulesen, kann das read Kommando verwendet werden:

```
read variable
```

Die Verwendung wird gezeigt, an einem Programm, das eine Datei mit Telefonnummern nach Zeilen durchsucht, die Namen enthalten, die der Benutzer eingeben kann.

```
$ cat tel_list
Alois Schuette          INF      8435
Ursula Schmunck         INF      8413
$
$ cat num.please
echo ----- Telefonverzeichnis -----
echo -n "Name: "
read name
grep -i $name tel_list
$
$ num.please
----- Telefonverzeichnis -----
Name: Schmunck
Ursula Schmunck         INF      8413
$
```

Hörsaalübung

Ändern Sie das folgende Programm so ab, dass der zu suchende Eintrag als Parameter beim Aufruf des Programms angegeben werden kann.

```
$ cat num.please
echo ----- Telefonverzeichnis -----
echo -n "Name: "
read name
grep -i $name tel_list
$
```

Umlenken der Ausgabe von Kommandos in Variablen

Durch die Kommandoersetzung kann die Ausgabe eines Programms so umgelenkt werden, dass eine Variable den resultierenden Wert zugewiesen bekommt.

Die allgemeine Form ist:

```
Variable=`Kommando`
```

Beispiel:

```
$ cat pt
time=`date|cut -c12-19`
echo aktuelle Zeit: $time
$
$ pt
aktuelle Zeit: 16:16:33
$
```

Zuweisen von Werten von Variablen

Auch auf der rechten Seite des Zuweisungsoperators (=) können Werte von Variablen auftauchen:

```
$ cat prog
prog=$0
echo das Programm $prog wurde gestartet
$
$ prog
das Programm ./prog wurde gestartet
$
$ mv prog Programm
$
$ Programm
das Programm ./Programm wurde gestartet
$
$
```

4.Shell Kontrollstrukturen

Die Shell stellt alle von "normalen" Programmiersprachen bekannte Kontrollstrukturen zur Verfügung:

Kommentare erlauben es, den Programmtext zu dokumentieren,

Durch das "**here document**" ist es möglich, im Programm Zeilen einzufügen, die als Standardeingabe von Kommandos des Programms verwendet werden,

Ein **exit** Kommando zusammen mit return Codes macht es möglich, ein Programm gezielt zu verlassen, bevor das Textende erreicht ist,

Mittels **for**- und **while**-Konstrukten sind Wiederholungen realisierbar,

Durch if- und **case** Anweisungen können Entscheidungen programmiert werden und

break und **continue** können das Abarbeiten von Schleifen beeinflussen.

4.1.Kommentare

Kommentare sind Texte, die einem #-Zeichen folgen; sie enden am Zeilenende.

Allgemeine Form:

```
#Kommentartext<CR>
```

Beispiel:

```
$ cat ptMitKommentar
# File:          ptMitKommentar
# Description:   print actual time
# Author:       as
# Usage:        ptMitKommentar
#
time=`date|cut -c12-19`          # selektiert Zeitangabe
echo aktuelle Zeit: $time
$
$ ptMitKommentar
aktuelle Zeit: 10:02:51
$
```

4.2. Here document

Innerhalb eines Shellprogramms ist es möglich, Zeilen als Eingabe von Kommandos zu markieren. Das Kommando verwendet dann diese Zeilen als Standardeingabe. Dadurch entfällt es, zuerst eine Hilfsdatei zu erzeugen, von der das Kommando dann liest und diese anschließend zu löschen.

Allgemeine Form:

```
Kommando <<Delimiter<RC>
...Eingabezeilen ...
Delimiter<CR>
```

Beispiel (zunächst mit Hilfsdatei):

```
$ cat gbdays
echo Alles Gute zum Geburtstag >> tmp$$
echo Wann gib es Kaffee und Kuchen? >> tmp$$
mail $1 < tmp$$
rm -f tmp$$
$
$ gbdays as
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/mail/as": 1 message 1 new
>N 1 as@pr.fbi.fh-darmsta Thu Jan 17 10:15 13/499
& 1
Message 1:
From as@pr.fbi.fh-darmstadt.de Thu Jan 17 10:15:03 2002
Date: Thu, 17 Jan 2002 10:15:03 +0100
From: Alois Schütte <as@pr.fbi.fh-darmstadt.de>
To: as@linux.ki.fbi.fh-darmstadt.de

Alles Gute zum Geburtstag
Wann gib es Kaffee und Kuchen?

& d
& q
$
```

Nun das Beispiel mit "Here Document"

```
$ cat gbdaysMitHeredoc
mail $1 <<EOF
Alles Gute zum Geburtstag
Wann gib es Kaffee und Kuchen?
EOF
$
$ gbdaysMitHeredoc as
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/mail/as": 1 message 1 new
>N 1 as@pr.fbi.fh-darmsta Thu Jan 17 10:19 13/501
& 1
Message 1:
From as@pr.fbi.fh-darmstadt.de Thu Jan 17 10:19:39 2002
Date: Thu, 17 Jan 2002 10:19:39 +0100
From: Alois Schütte <as@pr.fbi.fh-darmstadt.de>
To: as@linux.ki.fbi.fh-darmstadt.de

Alles Gute zum Geburtstag
Wann gib es Kaffee und Kuchen?

& d
& q
$
```

Diese Art der Eingabeumlenkung wird häufig verwendet, um interaktiven Programmen, vi z.B. Editoren innerhalb von Shellprogrammen benutzen zu können.

Das folgende Programm verwendet den `ex (vi)`, um in einer Datei Textersetzungen durchzuführen.

```
$ cat st
echo -n Dateiname:
read datei
echo -n Sucht-Text:
read alt
echo -n Ersetz-Text:
read neu
ex $datei <<EoF
%s/$alt/$neu/g
w
q
EoF
echo fertig
$
```

Ein Aufruf von `st` ist nachfolgend dargestellt.

```
$ cat tel_list
Alois Schuette          Informatik      8435
Ursula Schmunck        Informatik      8413
$
$ st
Dateiname:tel_list
Sucht-Text:Informatik
Ersetz-Text:INF
fertig
$ cat tel_list
Alois Schuette          INF            8435
Ursula Schmunck        INF            8413
$
```

4.3.Return Codes

Unix Kommandos sind i.A. so realisiert, dass sie einen Return Kode besitzen, der anzeigt, ob das Kommando erfolgreich oder fehlerhaft beendet wurde.

Unix Konvention ist, dass ein Return Kode 0 anzeigt, dass das Kommando erfolgreich beendet wurde, ein Wert ungleich 0 zeigt einen Fehler an.

In einem Shellprogramm kann der Return Kode durch das exit Kommando an den Aufrufer weitergegeben werden; der Aufrufer kann den Exit Status durch Abfrage des Spezialparameters \$? erkennen.

```
$ head /etc/hosts
##
# Host Database
#
##

127.0.0.1          localhost localhost.localdomain
$ echo $?
0
$
$ cat datXYZ
cat: datXYZ: Datei oder Verzeichnis nicht gefunden
$ echo $?
1
$
```

```
$ cat pt
time=`date|cut -c12-19`
echo aktuelle Zeit: $time
exit 0
$ pt
aktuelle Zeit: 10:55:51
$ echo $?
0
$
```

4.4.Schleifen

For- und while Schleifen erlauben es, eine Kommandosequenz wiederholt auszuführen.

4.4.1.for

Die for-Schleife führt eine Kommandosequenz aus, einmal für jedes Element einer Liste.

Allgemeine Form:

```
for Variable<CR>
    in Variablen_Liste
do<CR>
    Kommando_1<CR>
    Kommando_2<CR>
    ...
    Kommando_n<CR>
done<CR>
```

Bei jedem Durchlauf wird das nächste Element der Variablenliste an die Variable zugewiesen. Referenzen an diese Variable können überall in den Kommandos des do-done-Teils auftauchen.

Von der Shell werden Leerzeichen am Zeilenanfang ignoriert. Deshalb sollte Einrücken zur besseren Lesbarkeit verwendet werden.

Beispiel (mv Dateien in anderes Verzeichnis):

```
$ cat mv.file
echo -n "type in the directory path: "
read path
for file
    in memo1 memo2 memo3
do
    mv $file $path/$file
done
exit 0
$
$ mv.file
type in the directory path: /tmp
$
```

Im letzten Beispiel wurden die Dateinamen (memo1,...) explizit im Programm angegeben. Will man die Dateinamen beim Aufruf angeben können, so kann der in-Teil der for-Schleife entfallen.

```
$ cat mv.file1
echo -n "type in the directory path: "
read path
for file
do
    mv $file $path/$file
done
exit 0
$
$ mv.file1 xy*
type in the directory path: /tmp
$
```

Hörsaalübung

Verdeutlichen Sie sich das letzte Programm:

```
$ cat mv.file1
echo -n "type in the directory path: "
read path
for file
do
    mv $file $path/$file
done
exit 0
$
```

4.4.2.while

Bei einer while Schleife werden zwei Gruppen von Kommandos verwendet: die Kommandos der do-while Gruppe werden solange wiederholt, wie die Auswertung des **letzten** Kommandos der while-Gruppe den Returnstatus true=0 (Kommando erfolgreich ausgeführt) liefert.

Allgemeine Form:

```
while<CR>
    Kommando_1<CR>
    Kommando_2<CR>
    ...
    Kommando_n<CR>
do
    Kommando_n+1<CR>
    Kommando_n+2<CR>
    ...
    Kommando_n+m<CR>
done<CR>
```

Beispiel (Eintrag in Telefon-Datei):

```
$ cat telEintrag
file=telListe
while
    echo -n "Name: "
    read name
    echo -n "Telefon: "
    read telefon
do
    echo $name      INF      $telefon >> $file
done
$
```

Hörsaalübung

Machen Sie sich die Arbeitsweise des o.a. Programms klar.

Wann bricht es ab?

Wie müsste es abgeändert werden, dass eine Abbruch möglich ist, ohne das Programm mittels CTR-C zu beenden?

4.4.3. Die Ablage /dev/null

Im Unix Dateisystem existiert eine spezielle Geräte-Datei mit Namen "/dev/null". In diesen "Mülleimer" können ungewollte Ausgaben geschrieben werden.

```
$ find / -name "*.c" -print > resFile 2> /dev/null &  
$
```

Die Verwendung in Shellprogrammen sehen wir in den nächsten Beispielen.

4.5. Bedingte Ausführung

Ein if-Kommando bewirkt, dass die then-Kommandosequenz nur ausgeführt wird, wenn das letzte Kommando der if-Liste erfolgreich abgeschlossen wurde; ansonsten wird die else- Kommandosequenz ausgeführt.

Allgemeine Form:

```
if<CR>
    Kommando_1<CR>
    Kommando_2<CR>
    ...
    Kommando_n<CR>
then
    Kommando_n+1<CR>
    Kommando_n+2<CR>
    ...
else
    Kommando_n+m+1<CR>
    Kommando_n+m+2<CR>
    ...
fi<CR>
```

Beispiel:

```
$ cat search
echo -n "type in the word and file name: "
read word file
if
    grep $word $file > /dev/null 2>&1
then
    echo $word is in $file
else
    echo $word is NOT in $file
fi

$
$ search
type in the word and file name: if search
if is in search
$
$ search
type in the word and file name: do search
do is NOT in search
$
```

Das Shell-Programm serach sucht mittels grep ein Wort in einem File.

4.6.test Kommando

In while- und if-Kommandos sind häufig Tests durchzuführen. Dazu existiert in Unix ein eigenes Kommando, das test-Kommando.

Beispiel:

```
$ cat searchMitTest
echo -n "type in the word and file name: "
read word file
if test -r $file
then
    if
        grep $word $file > /dev/null 2>&1
    then
        echo $word is in $file
    else
        echo $word is NOT in $file
    fi
else
    echo "no such file: $file"
fi
$
```

Die wichtigsten Optionen des test-Kommandos sind:

NAME

test - check file types and compare values

SYNOPSIS

test EXPRESSION
test OPTION

DESCRIPTION

EXPRESSION is true or false and sets exit status.
It is one of:

(EXPRESSION)
 EXPRESSION is true

! EXPRESSION
 EXPRESSION is false

EXPRESSION1 -a EXPRESSION2
 both EXPRESSION1 and EXPRESSION2 are true

EXPRESSION1 -o EXPRESSION2
 either EXPRESSION1 or EXPRESSION2 is true

[-n] STRING
 the length of STRING is nonzero

-z STRING

the length of STRING is zero

STRING1 = STRING2

the strings are equal

STRING1 != STRING2

the strings are not equal

INTEGER1 -eq INTEGER2

INTEGER1 is equal to INTEGER2

INTEGER1 -ge INTEGER2

INTEGER1 is greater than or equal to INTEGER2

INTEGER1 -gt INTEGER2

INTEGER1 is greater than INTEGER2

INTEGER1 -le INTEGER2

INTEGER1 is less than or equal to INTEGER2

INTEGER1 -lt INTEGER2

INTEGER1 is less than INTEGER2

INTEGER1 -ne INTEGER2

INTEGER1 is not equal to INTEGER2

FILE1 -nt FILE2

FILE1 is newer (modification date) than FILE2

FILE1 -ot FILE2

FILE1 is older than FILE2

-b FILE

FILE exists and is block special

-c FILE

FILE exists and is character special

-d FILE

FILE exists and is a directory

-e FILE

FILE exists

-f FILE

FILE exists and is a regular file

-g FILE

FILE exists and is set-group-ID

-G FILE

FILE exists and is owned by the effective group ID

-L FILE

```
FILE exists and is a symbolic link

-r FILE
FILE exists and is readable

-s FILE
FILE exists and has a size greater than zero

-u FILE
FILE exists and its set-user-ID bit is set

-w FILE
FILE exists and is writable

-x FILE
FILE exists and is executable
```

Achtung:

Wenn ein test Kommando in einem Skript nicht so funktioniert, wie es eigentlich definiert ist, kann dies folgende Ursachen haben:

1. es existiert ein Shell Programm mit Namen test
2. es existiert ein ausführbares Programm (z.B. C-Programm) mit Namen test.

Deshalb sollten Sie als Programmnamen **nie** test verwenden.

In Shell-Programmen kann man die Kurzform des Test-Kommandos „[...]“ verwenden; dies macht Programme besser lesbar:

```
echo -n "type in the word and file name: "  
read word file  
if [ -r $file ]  
then  
    if  
        grep $word $file > /dev/null 2>&1  
    then  
        echo $word is in $file  
    else  
        echo $word is NOT in $file  
    fi  
else  
    echo "no such file: $file"  
fi
```

Hörsaalübung

Realisieren Sie ein Programm, das dem Benutzer ein Menü zur Verfügung stellt, bei dem es als Auswahlpunkte gibt:

```
1  Datum anzeigen
2  aktuelles Verzeichnis anzeigen
3  Verzeichnis wechseln
4  Datei anzeigen
0  Programm verlassen
```

Auswahl: _

4.7.case-esac

Durch das case-esac Kommando kann eine Kommandofolge ausgewählt werden, die zu einem Muster passt. Die Muster enden mit der runden Klammer, die dazu gehörenden Kommandos enden mit ";;".

Allgemeine Form:

```
case<CR>
  Muster_1 )<CR>
    Kommando_1_1<CR>
    ...
    Kommando_1_n<CR>
;;<CR>
  Muster_2 )<CR>
    Kommando_2_1<CR>
    ...
    Kommando_2_m<CR>
;;<CR>
  ...
  *)<CR>
    Kommando_*_1<CR>
    ...
    Kommando_*_t<CR>
;;<CR>
esac<CR>
```

In Aufschreibungsreihenfolge wird geprüft, welches Muster mit mit "Wort" übereinstimmt, dann werden die entsprechenden Kommandos ausgeführt. Gibt es keine Übereinstimmung, wird die *-Kommandofolge ausgeführt.

In den Mustern sind die Metazeichen "*?[]" erlaubt.

Beispiel:

```
$ cat set.term
echo -n "type in your terminal type: "
read term
case $term
  in
    vt100)
      TERM=vt100;;
    xterm)
      TERM=xterm;;
    *)
      echo unknown terminal
      TERM=vt100;;
esac
export $TERM
$
```

4.8.break, continue

Das Kommando **break** stoppt die Ausführung einer Schleife oder einer Verzweigung und die Programmausführung geht nach dem korrespondierenden **done**, **fi** oder **esac** weiter.

Das **continue** Kommando verzeigt zur nächsten Iteration einer Schleife und überspringt so die nach ihm stehenden Kommandos des Schleifenrumpfes.

Das folgende Beispiel zeigt zusammenhängend Schleifen und Entscheidungen: ein Auswahl-Menü.

```
$ cat menu.ksh
while :
do
    clear
    echo "  A Auflisten des aktuellen verzeichnisses"
    echo "  W verzeichniss Wechseln"
    echo "  S datei Schreiben"
    echo "  L datei Loeschen"
    echo "  z Datei anzeigen"
    echo "  E Ende"
    echo -n "                Auswahl: "
```

```

read auswahl
  case $auswahl
    in
      [aA])    ls -l | more
              ;;
      [wW])    echo -n "Verzeichnis: "
              read verzeichnis
              cd $verzeichnis
              ;;
      [sS])    echo -n "Datei: "
              read datei ; vi $datei
              ;;
      [lL])    echo -n "Datei: "
              read datei ; rm -i $datei
              ;;
      [zZ])    echo -n "Datei: "
              read datei ; more $datei
              ;;
      [eE])    echo Bye!
              exit
              ;;
      *)       echo  Auswahl nicht erlaubt!
              read
    esac
done
$

```

4.9.exit

Jedes Kommando gibt einen Exit-Status zurück. Der Status 0 bedeutet, dass das Kommando normal (fehlerfrei) beendet wurde. Ein von 0 verschiedener Status zeigt an, dass

das Kommando nicht ausgeführt werden konnte oder

bei der Ausführung ein Fehler aufgetreten ist (z.B. konnte eine Datei nicht geöffnet werden).

Beispiel:

```
$ cat xxx
cat: xxx: Datei oder Verzeichnis nicht gefunden
as@linux> echo $?
1
$
```

Das Kommando exit kann in Shellskripten verwendet werden, um den Exit-Status an die aufrufende Shell zu übergeben.

Beispiel:

```
$ cat zeigeDatei.ksh
echo -n "Datei: "
read datei
if [ -r $datei ]
then
    cat $datei
else
    echo Datei ist nicht lesbar!
    exit 1
fi
exit 0
$
```

```
$ zeigeDatei.ksh
Datei: xxx
Datei ist nicht lesbar!
$ echo $?
1
$ zeigeDatei.ksh
Datei: zeigeDatei.ksh
echo -n "Datei: "
read datei
if [ -r $datei ]
then
    cat $datei
else
    echo Datei ist nicht lesbar!
    exit 1
fi
exit 0
$ echo $?
0
$
```

5. Fehlerbehandlung der Shell

Folgende Fehler können beim Ablauf eines Shellskripts auftreten:

1. eine **E/A Umlenkung** ist nicht möglich, weil z.B. eine Eingabedatei nicht existiert;
2. eine **Kommando existiert nicht** oder ist nicht ausführbar;
3. ein **Kommando terminiert abnormal**, z.B. durch einen Speicherfehler;
4. ein **Kommando** terminiert korrekt, liefert aber **einen Exit-Status verschieden 0**;
5. im Skript ist ein **Syntaxfehler**, z.B. if...else ohne fi;
6. ein **Signal** erreicht die Shell, z.B. durch CTR_C oder kill-Kommando;
7. ein **Fehler** tritt bei einem **Shell-internen Kommando** auf.

Die Shell reagiert per Default wie folgt:

Fehler vom Typ 1-4 werden ignoriert; es wird mit der Bearbeitung des nächsten Kommandos im Skript fortgefahren und (außer bei 4) wird eine Fehlermeldung ausgegeben.

Fehler vom Typ 5-7 bewirken einen Abbruch des Shellskripts.

Eine Shellprozedur wird normalerweise durch die Signale

n	Bedeutung
1	hangup (exit)
2	interrupt (CTR_C)
3	quit
9	kill
15	software termination (kill)

abgebrochen.

Auf diese Signale (außer Signal 9) kann per Programm reagiert werden:

Ignorieren

```
trap '' n
```

Reaktion per Default

```
trap n
```

Ausführen eines Kommandos

```
trap 'Kommando' n
```

Beispiele:

Ein Skript soll bei Abbruch durch CTR_C die Meldung "Bye bye" ausgeben

```
$ cat byeBye.ksh
trap 'echo Bye bye; exit 1' 2
echo Start
while :
do
    date
    sleep 10
done
$
$ byeBye.ksh
Start
Die Mär 12 09:04:04 CET 2002
Die Mär 12 09:04:14 CET 2002
CTR_C
Bye bye
$
```

Hörsaalübung

Verwenden Sie den trap-Mechanismus in Ihrem Programm „Menue“, so dass das kein Abbruch durch CTR-C möglich ist, bei allen anderen Signalen soll nachgefragt werden, ob wirklich abgebrochen werden soll.

6. Debuggen von Shell-Programmen

Zum (elementaren) Debuggen von Shell-Programmen existieren zwei Optionen für die Shell:

- v Programm druckt die Eingabezeilen des Shellprogramms, so wie sie gelesen werden
- x Programm druck Kommandos und ihre Argumente, so wie sie ausgeführt werden

Beispiel:

Ein Programm, das eine Mail versendet, mit der Bitte zu kommen.

```
$ cat pleaceCome.ksh
today=`date`
echo -n "enter person: "
read person
mail $person <<!
Hi $person
When you log off, pleace come into my office.
$today
as
!
$
$ plaeseCome.ksh
enter person: jennifer
$
```

Zunächst ohne Substitution der Variablen:

```
$ sh -v pleaceCome.ksh
today=`date`
date
echo -n "enter person: "
enter person: read person
vs48
mail $person <<!
```

Nun mit Substitution

```
$
$ sh -x pleaceCome.ksh
++ date
+ today=Mon Mär 11 15:36:28 CET 2002
+ echo -n 'enter person: '
enter person: + read person
vs48
+ mail vs48
$
```

Verwendet man **Pipes** beim Programmieren, so sind Fehler schwer zu finden, da man den Inhalt der Pipe nicht sehen kann. Dazu stellt Unix das Kommando "**tee**" bereit, mit dem man Pipes debuggen kann.

Will man beobachten, was in der Pipe bei einem Kommando der Form

```
$ Kommando1 | Kommando 2
```

passiert, dann kann man den Inhalt der Pipe durch das tee Kommando in eine Datei schreiben:

```
$ Kommando | tee Datei | Kommando2
```

Beispiel:

```
$ cat pipe1.ksh
who | grep as | cut -c1-9
$
$ pipe1.ksh
as
$
```

Nun mit dem tee Kommando.

```
$ cat pipe2.ksh
who | tee dat | grep as | cut -c1-9
$
$ pipe2.ksh
as
$
$ cat dat
as      pts/5      Mar 11 14:45 (141.100.42.131:0.0)
root   :0        Mar  8 17:01 (console)
$
```

Die Verwendung der Debugging Möglichkeiten sollte man sich verdeutlichen. Dazu realisieren wir ein Programm, das die Fakultät einer Zahl berechnet.

Um "Rechnen" zu können ist das Kommando "expr" in Unix vorhanden.

```
$ cat fakultaet.ksh
echo -n "Zahl: "
read zahl
res="1"
while [ $zahl -gt 0 ]
do
    res=`expr $res \* $zahl`
    zahl=`expr $zahl - 1`
done
echo $res
$
$
$ fakultaet.ksh
Zahl: 3
6
$
```

Wenn man beim expr-Kommando bei der Multiplikation den \ weglässt, passieren nicht vorher-sagbare Dinge (Zur Erinnerung: hängt vom Inhalt des aktuellen Verzeichnisses ab, die Shell ex-pandiert den *).

Solche Fehler lassen sich mit en o.a. Optionen leicht finden – dies sollte im Praktikum versucht werden!

```
$ cat fakultaet1.ksh
echo -n "Zahl: "
read zahl
res="1"
while [ $zahl -gt 0 ]
do
    res=`expr $res * $zahl`      # \ vergessen
    zahl=`expr $zahl - 1`
done
echo $res
$
```

```
$ sh -x fakultaet1.ksh
+ echo -n 'Zahl: '
Zahl: + read zahl
2
+ res=1
+ '[' 2 -gt 0 ']'
++ expr 1 dat fakultaet.ksh fakultaet1.ksh menue.ksh pipe1.ksh pipe2.ksh please-
Come.ksh 2
expr: Syntaxfehler
+ res=
++ expr 2 - 1
+ zahl=1
+ '[' 1 -gt 0 ']'
++ expr dat fakultaet.ksh fakultaet1.ksh menue.ksh pipe1.ksh pipe2.ksh pleaseCo-
me.ksh 1
expr: Syntaxfehler
+ res=
++ expr 1 - 1
+ zahl=0
+ '[' 0 -gt 0 ']'
+ echo

$
```

Hörsaalübung

Debuggen Sie das Programm zur Berechnung der Fakultät.

```
$ cat fakultaet1.ksh
echo -n "Zahl: "
read zahl
res="1"
while [ $zahl -gt 0 ]
do
    res=`expr $res * $zahl`      # \ vergessen
    zahl=`expr $zahl - 1`
done
echo $res
$
```

7. Anpassen der Login Umgebung

Die Umgebung, in die man nach dem login gelangt, kann individuell angepasst werden. Die Shell (ksh) sucht im Homeverzeichnis eine Datei mit Namen ".login". Diese Datei wird zuerst ausgeführt.

Achtung:

Wenn man die Datei .profile editiert und einen Fehler (z.B. fehlerhaftes Kommando) verwendet, so kann es u.U. vorkommen, dass man sich nicht mehr anmelden kann. Deshalb sollte man immer eine zweite Shell (in einem anderen Fenster) offen haben. Dann kann man die Datei .profile testen und ggf. Änderungen vornehmen.

8. Funktionen

Shell-Funktionen bieten die Möglichkeit, mehrere Kommandos zusammenzufassen bzw. Kommandos mit vielen Parametern anzukürzen.

Shell-Funktionen dürfen nicht mit Shell-Programmen verwechselt werden:

Ein Shell-Programm ist eine Datei, in der Kommandos stehen.

Eine Shell-Funktion wird von der Shell verwaltet und liegt im Speicher, ist also ohne Plattenzugriff abrufbar.

Eine Shell-Funktion wird wie folgt definiert:

```
FunktionName () {  
    Kommandos  
}
```

Eine Shell-Funktion wird wie ein Shell-Programm aufgerufen, die Parameter folgen dem Funktionsnamen.

Mit

```
unset FunktionName
```

kann die Funktion entfernt werden.

Beispiel:

Eine Funktion "c" zum Change Directory mit Ändern des Prompt.

```
$ cat fkt1.ksh
c() {
    cd $*
    cwd=`pwd`          # change dir
    PS1="`basename $cwd` $ " # change prompt
}
$
$ . fkt1.ksh
$
$ c /tmp
tmp $
tmp $
tmp $ c
as $ pwd
/users/as
as $
```

9. Ausführen ohne neuen Prozess zu erzeugen.

Wird ein Kommando ausgeführt, so wird (durch fork) von der Shell ein neuer Prozess erzeugt und der Code des Kommandos dem sh-Code überlagert. Dadurch sind Variablenwerte nach Beendigung des Prozesses, der das Kommando ausführt im Vater (der Shell) nicht mehr wirksam.

```
as@hal:src> cat ohnePunkt
x=1;
export x
echo x = $x
as@hal:src>
as@hal:src> x=0
as@hal:src> echo $x
0
as@hal:src> ohnePunkt
x = 1
as@hal:src> echo $x
0
as@hal:src>
```

Um die Erzeugung eines neuen Prozesses zu verhindern, muss das Kommando aufgerufen durch „. Kommando“.

```
as@hal:src> x=0
as@hal:src> . ohnePunkt
x = 1
as@hal:src> echo$x
-bash: echo1: command not found
as@hal:src> echo $x
1
as@hal:src>
```

Alos müssen nach Änderungen der Datei `.profile` oder `bach_profile` diese stets mit

```
. .bach_profile
```

wirksam gemacht werden.

Hörsaalübung:

Realisieren Sie ein Shellprogramm, in dem es eine Shellfunktion `fibonacci` gibt, die die n-te Fibonacci-Zahl ermittelt.

Die Fibonacci Zahlen sind wie folgt definiert:

- Die n-te Fibonacci Zahl ist die Summe der n-1-ten und der n-2-ten Fibonacci Zahlen, wobei die erste und zweite Fibonacci Zahl eins ist.
- Beispiel: 1,1,2,3,5,8,13,21,...
- D.h. $fibonacci(1) = fibonacci(2) = 1$
 $fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$ für $n > 2$

Verwenden Sie diese Funktion, um eine Liste der ersten 20 Fibonacci-Zahlen auszugeben.