HANDOUT
Proseminar Unix-Tools
Shell-Programmierung SS05

Alles Wissenswerte zur Lösung des Aufgabenblattes. Für weitere Informationen benutze bitte die manpage zu bash: **`man bash`** oder http://www.tldp.org/LDP/abs/html/

1. Command history

Zum Wiederholen und Editieren früherer Kommandos:

- wenn man einen schon einmal eingegebenen Befehl wieder aufrufen möchte drückt man die Pfeiltaste nach oben bis der Befehl erscheint
- zum Editieren der Befehlszeile kann man sich mit den Cursortasten an die entsprechende Stelle bewegen und sie korrigieren
- Expansionsfunktion: ergänzt einen Dateinamen, falls er bereits eindeutig identifiziert weden kann
- nach der Eingabe Tabulator drücken
- gibt es mehrere Dateinamen mit den gleichen Anfangsbuchstaben erhält man durch zweimaliges Drücken des Tabulators eine Auswahlliste

2. Variablen

Variable=Wert	Setzt die Variable auf den Wert.
$Variable	Nutzt den Wert von Variable. Die Klammern müssen nicht mit angegeben werden, wenn die Variable von Trennzeichen umgeben ist.

Vordefinierte Variablen:

$n	Aufrufparameter mit der Nummer n, n<= 9
$*	Alle Aufrufparameter
$@	Alle Aufrufparameter
$#	Anzahl der Aufrufparameter
$?	Rückgabewert des letzten Kommandos
$$	Prozeßnummer der aktiven Shell
$!	Prozeßnummer des letzten Hintergrundprozesses
ERRNO	Fehlernummer des letzten fehlgeschlagenen Systemaufrufs
PWD	Aktuelles Verzeichnis (wird durch cd gesetzt)
OLDPWD	Vorheriges Verzeichnis (wird durch cd gesetzt)

Variablensubstitution / Parametererweiterung:

${Variable:-Wert}	Nutzt den Wert der Variable, falls die Variable nicht gesetzt ist, wird Wert benutzt.
${Variable:=Wert}	Nutzt den Wert der Variable, falls die Variable nicht gesetzt ist, wird Wert benutzt und Variable erhält den Wert.
${Variable:?Wert}	Nutzt den Wert der Variable, falls die Variable nicht gesetzt ist, wird der Wert ausgegeben und die Shell beendet. Wenn kein Wert angegeben wurde, wird der Textparameter null or not set ausgegeben.
${Variable:+Wert}	Nutzt den Wert, falls die Variable gesetzt ist.
${Variable:%pattern}	löscht das erste pattern vom Ende her
${Variable:%pattern}	löscht das längste pattern vom Ende her
${Variable:#pattern}	löscht das erste pattern vom Anfang
${Variable:##pattern}	löscht das längste pattern vom Anfang

3. Wildcards

Die wichtigsten Meta-Zeichen sind:

- **`*`** eine Folge von keinem, einem oder mehereren Zeichen
- **`?`** ein einzelnes Zeichen

- **[abc]** Übereinstimmung mit einem beliebigen Zeichen in der Klammer
- **[a-q]** Übereinstimmung mit einem beliebigen Zeichen aus dem angegebenen Bereich
- **[!abc]** Übereinstimmung mit einem beliebigen Zeichen, das nicht in der Klammer steht
- ∼ Homeverzeichnis des aktuellen Benutzers
- ∼**name** Homeverzeichnis des Benutzers name
- ∼**+** aktuelles Verzeichnis
- ∼**−** vorheriges Verzeichnis

4. Quoting

- Mechanismus der Shell die Bedeutung der Sonderzeichen auszuschalten
- es gibt hierzu 3 verschieden Möglichkeiten:
  1. Maskieren mit Backslash
     - in einem Kommentar oder innerhalb von `` `...` `` (backticks) wird Backslash nicht als Sonderzeichen interpretiert ⇒ hat keine Wirkung auf andere Sonderzeichen
     - innerhalb von `"..."` schaltet er nur die Bedeutung von `' $ " \ ` ` aus
  2. Klammerung mit einzelnen Apostrophen **('...')**
     - alle Metazeichen außer `'` (einfaches Anführungszeichen) verlieren ihre Bedeutung
     - wenn `'` seine Bedeutung verlieren soll, muss es durch Anführungszeichen geklammert werden
  3. Klammerung mit Anführungszeichen **("...")**
     - hierdurch verlieren die meisten aber nicht alle Sonderzeichen ihre Bedeutung
     - `' $ " \ ` ` behalten ihre Sonderbedeutung

5. Befehlsaliase

- für häufig benutzte oder spezielle Kommandos kann man einen sogenannten Alias definieren
- dies kann man manuell direkt auf der Kommandozeile tun
- Sytanx: **alias aliasname='Komanndostring'**

6. Ein- und Ausgabeumlenkung

Kommando > Datei	Falls die Datei bereits existiert wird sie überschrieben ansonsten wird eine neue Datei angelegt
Kommando >> Datei	Falls die Datei bereits existiert wird die Ausgabe an die Datei angehängt, ansonsten wird eine neue Datei angelegt
Kommando < Datei	Eingabe aus der Datei lesen

Bei Umlenkungen kann ein Dateideskriptor unmittelbar vor dem Umlenkungszeichen angegeben werden. Ansonste bezieht sich die Ausgabeumlenkung nur auf die Standardausgabe, nicht auf die Fehlerausgabe.

7. Pipelining

- Verknüpfung der Standardausgabe eines Kommandos mit der Standardeingabe eines anderen Kommandos
- Pipe Operator Befehl1 | Befehl2 wirkt z.B. wie ein Filter: Kommando, das eine Standardeingabe einliest, sie verändert und das Ergebnis wieder ausgibt

8. Programmierkonstrukte der Shell

Eine Befehlsliste (kurz liste) ist eine Reihe von Befehlen oder Ausdrücken, die durch **; & && ||** oder <newline> getrennt werden.
Ein Befehl beendet immer mit einem Exitstatus, er ist 0 wenn der Befehl erfolgreich war und ungleich 0 wenn er fehlschlug.

Trennzeichen	Beispiel
; trennt Befehle & startet einen Befehl im Hintergrund && UND-Verbindung zweier Befehle || ODER-Verknuepfung zweier Befehle <newline>: in der interaktiven Shell startet es einen Befehl, in einem Shellskript trennt es die Syntaxelemente	ls ; cd / # ruft zuerst ls, dann cd / auf cp DATEI /tmp & # kopiert DATEI im Hintergrund ls /tmp/DATEI && rm DATEI # wenn ls erfolgreich beendet, wird DATEI entfernt cd DIR || mkdir DIR # wenn cd nicht nach DIR wechseln kann, wird DIR erstellt if true then echo if ist wahr fi

8.1. Ausdrücke

- **((expression))** wird als arithmetischer Ausdruck ausgewertet. Beispiel dazu unter if. C ähnliche Syntax. Es stehen die grundlegenden mathematischen Operatoren zur Verfügung: **+ - * / % VAR++ VAR--**
- **[[expression]]** ist ein bedingter Ausdruck, mit dem man Strings (also auch Variablen), Dateiattribute und arithmetische Ausdrücke testen kann. Siehe dazu auch **man test**. **Achtung > und < testet hier auf lexikographische Ordnung**. Jedem Ausdruck kann ein **!** zum Negieren vorangestellt werden (Leerzeichen vor und nach dem **!**). Beispiele:

  [[! -d /home/user/DIR]] testet ob Verzeichnis DIR nicht existiert.
  [["$VAR == "YES"]] testet ob in der Variablen VAR "YES" steht.

8.2. if

Syntax:

```
if liste; then liste; [ elif liste; then liste; ] ... [ else liste; ] fi
```

die Ausdrücke in eckigen Klammern sind optional.
Wenn der Exitstatus von liste 0 liefert (erfolgreich läuft) wird die liste im then Teil ausgeführt. Wenn nicht, wird jede elif liste überprüft, wenn erfolgreich wird die zugehörige then liste gestartet. Ansonsten die else liste.

Beispiel testet ob 3 größer als 1 ist und gibt bei Erfolg 3>1 ist wahr aus:

```
if (( 3 > 1 ))
 then
  echo '3>1' ist wahr
fi
```

8.3. for

Syntax1:

```
for name [ in word ] ; do liste ; done
```

word wird erweitert und bildet eine Liste mit Elementen. Der Variablen name wird dann nacheinander jedes Element von word zugewiesen. Damit wird dann liste im do Teil ausgeführt.

Beispiel benennt alle .txt in .doc um:

```
for i in *.txt
do
 mv $i ${i%.*}.doc
done
```

Syntax2:

```
for (( expr1 ; expr2 ; expr3 )); do liste ; done
```

expr1-3 sind jeweils arithmetische Ausdrücke. Zuerst wird expr1 ausgewertet, dann expr2 solange bis sie 0 ergibt. Jedes mal wenn expr2 nicht 0 ist wird liste ausgeführt. Und dann wird expr3 evaluiert.

Beispiel gibt die Zahlen von 0 bis 9 aus:

```
for (( i=0 ; i < 10 ; i++ ))
 do
  echo $i
done
```

8.4. while und until

Syntax:

```
while liste; do do-Befehlsliste; done
until liste; do do-Befehlsliste; done
```

while führt die do-Befehlsliste solange aus bis liste keinen Exitstatus von 0 liefert, also nicht erfolgreich beendet.
until führt die do-Befehlsliste solange aus bis liste einen Exitstatus von 0 liefert, also erfolgreich läuft

Beispiel gibt solange die Benutzerin angemeldet ist alle 10 Sekunden "Benutzerin ist angemeldet" aus:

```
while who|grep Benutzerin >> /dev/null
 do
  echo Benutzerin ist angemeldet
  sleep 10
done
```

8.5. case

Syntax:

```
case word in [ [(] pattern [ | pattern ] ... ) liste ;; ] ... esac
```

case expandiert word und vergleicht es mit jedem pattern. Bei Übereinstimmung wird liste ausgeführt. Kein weiterer Vergleich wird durchgeführt.

Beispiel überprüft ob $1 gleich start oder stop ist und führt die jeweilige liste aus:

```
case "$1" in
start)
echo STARTE DAEMON
    /sbin/daemon
;;
    stop)
echo STOPPE DAEMON
killall daemon
;;
esac
```