W e r n i g e r o d e

Fachbereich Automatisierung und Informatik

Vorlesung „Grundlagen der Informatik II“

Vorlesung „Informatikgrundlagen II“

Kapitel Unix-Shell

Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
Friedrichstraße 57 - 59
38855 Wernigerode

Raum: **2.202**
Tel.: **03943/659-338**
Fax: **03943/659-399**
Email: **mwilhlem@hs-harz.de**

Inhaltverzeichnis

Quellcodes

Abbildungen

1 Die Shell

Die Shell ist der ein Hauptbestandteil jedes Unix-Systems. Ohne funktionierende Shell wird das System nicht hochfahren. In der Shell werden Befehle aufgerufen, Variablen gesetzt, Programme gestartet und das System durch Scripte gewartet. Die nächste Tabelle zeigt einige Shell-Varianten. Nicht jede Shell unterstützt alle Eigenschaften. In diesem Script wird der Schwerpunkt auf die Bourne-Shell gelegt.

Name	**Shell**
sh	Bourne Shell
bsh	Bourne Shell
bash	Bourne Again Shell
csh	Berkeley C-Shell
es	Extensible Shell, basierend auf rc
esh	Easy Shell
kiss	Karel´s Interactive Simple Shell
ksh	Korn-Shell
pdksh	Public Domain Korn Shell
rc	Implementierung der AT&T Plan 9 Shell
sash	Stand-Alone Shell
tcsh	Tenex C-Shell
zsh	Z-Shell

Eine Shell erinnert in ihrer Grundfunktion an die Command.com. Sie ist aber wesentlich Leistungsfähiger. So besitzt sie Variablen, Kontrollstrukturen und Funktionen.

1.1 Aufruf der Shell

Mit Hilfe des Befehls „telnet amsel.hs-harz.de“ kann der Unixrechner der Firma SGI angesprochen werden. Der Befehl „bash“ öffnet eine Bourne-Shell.

1.2 Aufruf einer Datei

Um eine Datei zu starten, tippt man „bash datei parameter“.

Eine Shell im Betriebssystem UNIX umfasst jeweils zwei Funktionen:
- Kommandointerpreter
- Programmiersprache

1.3 Kommando

Um eine Datei zu starten, tippt man „bash datei parameter“.

In Abhängigkeit von „*kommandoname*“ werden die „*Argumente*“ definiert. Dabei lässt sich allgemein angeben, dass es sich in der Regel bei dem ersten Argument um Optionen bzw. Schalter handelt, die im allgemeinen mit - beginnen. Danach können weitere Parameter folgen.

Die folgende Abbildung zeigt das Syntaxdiagramm:

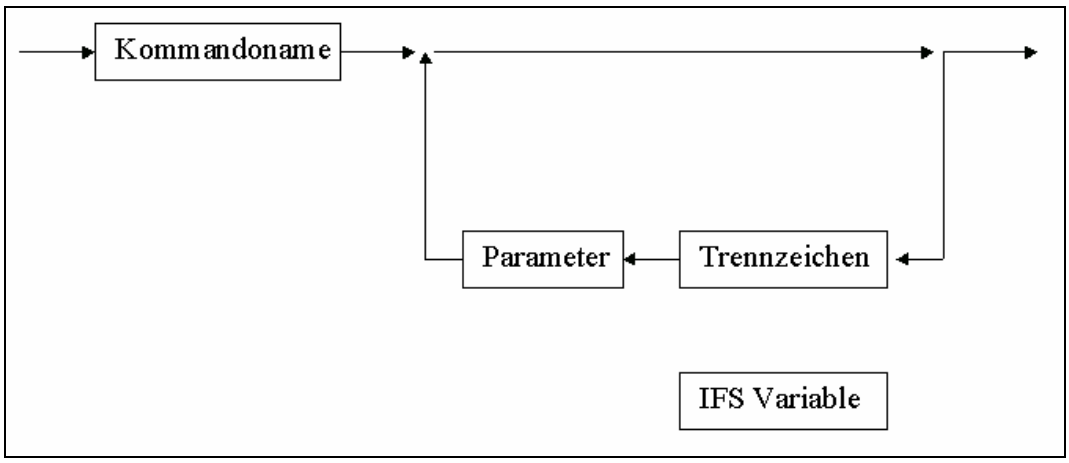

Abbildung 1 Syntaxdiagramm Kommando

Als erstes kommt das Kommando, dann die Parameter mit einem Trennzeichen. Das Trennzeichen wird durch die Variable „IFS“ definiert. Es ist natürlich auch erlaubt, keinen Parameter einzugeben.

Beispiel:

ls	listet alle Dateien und Verzeichnisse auf
ls dat*	listet alle Dateien und Verzeichnisse auf. Die mit „dat“ anfangen
ls -R dat*	listet alle Dateien und Verzeichnisse auf. Die mit „dat“ anfangen. Die Anzeige wird rekursiv auf alle Unterverzeichnisse fortgesetzt.

1.4 Standardein- ausgabe

Um Ergebnisse in Dateien zu speichern oder aus Dateien zu lesen, existieren die Standardeingabe und die Standardausgabe. Normalerweise ist die „Eingabe“ auf die Tastatur und die Ausgabe auf den „Bidlschirm“ gesetzt. Für Fehlermeldungen existiert noch die Fehlerausgabe.

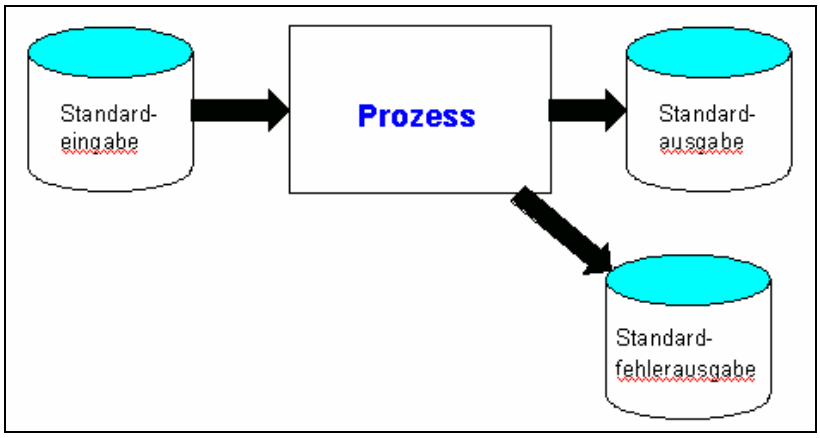

Abbildung 2 Standard-Ein- und ausgabe

1. Beispiel:

Mit dem Befehl „ls > datei1“ wird die Ausgabe in die Datei „datei1“ geschrieben. Diese Datei kann mit dem Befehl „cat“ angezeigt werden.

2. Beispiel:

Mit dem Kommando wc kann man Zeichen, Wörter und Zeilen zählen. Um die Anzahl der Dateien in einem Verzeichnis zu bestimmen, kann man folgende Befehle eingeben:

ls > liste	Speicherung der Dateien in einer Datei
wc < liste	Kommando „wc“ bezieht die Eingabe aus der Datei „liste“

Diese Arbeitsweise benutzt eine temporäre Datei. Diese sollte natürlich am Ende wieder gelöscht werden. Diese Sequenz ist ein gutes Beispiel für eine Scriptdatei.

Alternativ dazu kann man mit dem Pipekonzept eine automatische Weiterleitung der Ergebnisse an ein Kommando erreichen. Das Pipe-Symbol „|“ definiert die Weiterleitung.

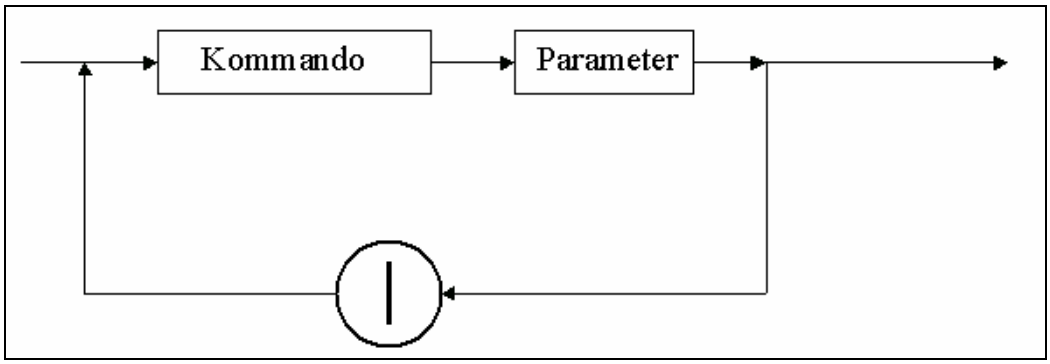

Abbildung 3 Syntax für Pipe (Pipekonzept)

Damit kann man die obigen Befehle durch einen ersetzten:

```
ls | wc
```
bzw.
```
ls –1 | wc -l
```

Durch den ersten Parameter „-1“ werden die Dateien zeilenweise ausgegeben. Der Parameter „-l“ zählt im Kommando „wc“ nur die Zeilen.

Die Umleitung „>“ erzeugt immer eine neue Datei. Mit der Umleitung „>>“ wird die Ausgabe an die vorhandene Datei angehängt. Ist die Datei noch nicht vorhanden, so wird sie angelegt.

Beispiele:

Kommando	**Beschreibung**
kommando < infile > outfile 2>errorfile	Allgemeine Syntax
ls -l > liste	Ausgabe zeilenweise, Umleitung in Datei „liste“
ls -l /bin >> liste	Ausgabe zeilenweise, Verzeichnis „/bin“, Ergebnis wird an die Datei „liste“ angehängt
wc -l < liste	Zählt die Zeilen (lines) in der Datei „liste“
< listing wc –l	Gültiger Befehl! Zählt die Zeilen (lines) in der Datei „liste“
cat /etc/passwd > passwd	Listet die Paßwortdatei auf. Das Ergebnis wird in die Datei „passwd“ geschreiben. Es erfolgt keine Anzeige. Maximal werden Fehler angezeigt.
cat /etc/passwd > passwd 2>/dev/null	Listet die Paßwortdatei auf. Das Ergebnis wird in die Datei „passwd“ geschreiben. Es erfolgt keine Anzeige. Fehler werden nicht angezeigt, sondern in den Papierkorb geschrieben. Die Sequenz „2>/dev/null“ darf keine Leerstellen enthalten.

1.5 Shell-Sonderzeichen

Jede Shell hat bestimmte Zeichen, die die Eingabe steuern.

- Das Semikolon trennt Kommandos
- Das Ambersand-Zeichen „&“ startet das Kommandos als Hintergrundprozess
- Promptstring: $ oder #
- Weitere Eingabe: >
- Abbruch mit Strg+C
- Suchpfad: path=/usr/local/bin/commands
- Stringbegrenzer " und '
- Folgezeichen: \ in einer Scriptdatei

1.6 Shell-Variablen

Jede Shell hat bestimmte Zeichen, die die Eingabe steuern.

Variable	**Beschreibung**
BASH	Der volle Name der bash-Shell
BASH_ENV	Ist dieser Parameter gesetzt, wenn die bash ein Script ausführt, wird der Wert als Name einer Datei interpretiert
BASH_VERSINFO	Ein Array mit Versionsinformationen
BASH_VERSION	Die Versionsnummer der bash-Shell
DISPLAY	Wird mit dem X-Window-System gearbeitet, dann enthält diese Variable den Namen des Ausgabebildschirms für X-Programme.
HISTFILE	Der Name der Datei, in die die Liste mit den Kommandos gespeichert wird. Defaultwert: „~/bash_history“
HISTFILESIZE	Spezifiziert die maximale Anzahl an Zeilen in der History-Datei.
HISTSIZE	Gibt die Anzahl der Befehle in der History-Datei an.
HOME	Enthält den Pfad zum Home-Verzeichnis des Users.
HOSTNAME	Der aktuelle Rechnername
HOSTTYPE	Die verwendete Rechnerarchitektur
IFS	Das intern verwendete Zeichen, um Felder zu trennen. Es ist normalerweise auf Leerzeichen, Tabulator und Zeilenumbruch gesetzt.
LOGNAME	Der Loginname des Users
MACHTYPE	Der Maschinentyp (Betriebssystem, Distributer)
MAIL	Pfad zur Mailbox des Benutzers.
OLDPWD	Das vorige Verzeichnis
OPTERR	Besitzt die Variable den Wert 1, werden Fehlermeldungen angezeigt, die vom eingebauten *getopts* erzeugt werden. Die Variable wird jedesmal auf 1 gesetzt, wenn die Shell aufgerufen oder ein Skript ausgeführt wird.
OSTYPE	Beschreibt das Betriebssystem
PATH	Liste der Verzeichnisse, in denen nach Programmen gesucht wird.
PPID	Beschreibt die Prozess-ID des übergeordneten Prozesses der Shell.
PS1	Enthält die Beschreibung des normalen Prompts. Kann gesetzt werden.
PS2	Enthält die Beschreibung des zweiten Prompts. Kann gesetzt werden.
PS3	Wird als Prompt für *select* verwendet
PS4	Der Wert wird expandiert und vor jedem Kommando angezeigt, wenn die Shell die Ausführung protokolliert. Das erste Zeichen wird mehrfach kopiert, um verschiedene Stufen anzuzeigen.
PWD	Das aktuelle Arbeitsverzeichnis (print working directory)
TERM	Name der verwendetetn Terminalemulation aus der termcap- bzw. terminfo-Datenbank.
RANDOM	Liefert eine Pseudo-Zufallszahl. Wird dieser Variablen eine Zahl zugewiesen, so wird der Zufallsgenerator neu initialisiert.
UID	Die User-ID des aktuellen Benutzers oder Skripts.
SHELL	Ausgabe des Namens der Shell

Beispiele:

Anzeige der Inhalte der Variablen:

Befehl	**Ergebnis**
echo $HOSTTYPE	i386
echo $MACHTYPE	i386-pc-linux-gnu
echo $PS1	\[\]\u@ttyp0\[\W]\$\[\]
echo UID	1000
echo $HOME	/home/knoppix

Setzen von Variablen:

HOME=/home/knoppix/test

Anmerkung:
Wenn man eine Variable anzeigen möchte, muss man ein Dollarzeichen davor setzen.

1.7 Start- Endebefehle

Start: Strg+q
Start / Stop Strg+s
Shellende Strg+d

1.8 Wildcards

In einem Kommando wird ein Wort das die Zeichen *,? oder [..] enthält, als Namensmuster betrachtet und vor der Verwendung expandiert. Dabei bedeuten:

Zeichen	**Beschreibung**
*	eine beliebige Zeichenkette, sie kann kein Zeichen bedeuten
?	ein beliebiges Zeichen. Es muss immer ein Zeichen (A,z,0,9,.) sein.
[...]	ein Satz einzelner Zeichen
[.-.]	ein Bereich von Zeichen
!	benutzt in einer Gruppe. Wird als erstes Zeichen nach dem [eingegeben. Negation der Auswahl

Wildcards sind für folgende Befehle geeignet (Auswahl):
rm, ls, cat, grep, find und cp

Beispiele:

Das aktuelle Verzeichnis enthält die Dateien
prog1, Prog1,

p1.1, p1.2, p1.3, p1.4, p1.5, p1.6

Namensmuster	**Dateien**
prog*	prog1
p???	p1.1, p1.2, p1.3, p1.4, p1.5, p1.6
2	p1.2
p1.[136]	p1.1, p1.3, p1.6
p1.[1-36]	p1.1, p1.2, p1.3, p1.6
p1.[!1-36]	p1.4, p1.5 (nur kornshell)
../*	Dateien oberhalb des aktuellen Verzeichnisses
../.*	../.profile
./*	prog1, p1.1, p1.2, p1.3, p1.4, p1.5 und p1.6

2 Shell-Programmierung

Jeder eingegebene Befehl in einer Shell kann als Shell-Programmierung interpretiert werden. Komplexere Befehle werden nur der Übersicht in einer Datei geschrieben und dann ausgeführt.

2.1 Variablen

Variablen werden automatisch bei der Benutzung definiert. Eine Konvertierung in die unterschiedlichen Datentypen geschieht automatisch. Die Wertzuweisung an eine Variable geschieht mittels Gleichheiszeichen. Es darf kein Leerzeichen enthalten sein.

Es lassen sich grundlegend drei Formen unterscheiden:
- Variablendeklaration
- Wertzuweisung
- Wertreferenzierung (Zugriff auf den Wert einer Variablen)

Zur Verdeutlichung des Sachverhaltes sollen wenige Beispiele dienen:

Form	**Beispiel**	**Bildausgabe**
Deklaration	**$ paul=**	
Wertzuweisung	**$ paul=otto**	
Wertreferenzierung	**$ echo $paul**	**otto**

Beispiele:

Anweisung	**Ergebnis**
Paul =	Deklaration
Paul = Otto	Wertzuweisung
echo Paul	Ausgabe Paul
echo paul	Ausgabe von paul
echo Otto	Ausgabe Otto
echo otto	Ausgabe otto
echo $Paul	Ausgabe der Variable „Paul“ mit Inhalt „Otto“

Folgendes Beispiel erläutert die Klammerung. Gesetzt wird die Variable „my“ mit dem Inhalt „Korn“. Mit der ersten Ausgabe erhält man nur den Satz „I like“. Die Variable „myshell“ existiert noch nicht.

```
my=Korn
echo I like $myshell

my=Korn
print I like ${my}shell
```

Quellcode 1 Kornshell

Erst die zweite Variante zeigt das richtige Ergebnis.

2.2 Gültigkeitsdauer von Variablen

Der folgende Quellcode zeigt ein Beispiel mit verschachtelten Shells. Das Ergebnis

Befehl	**Ergebnis**	**Beschreibung**
lage=oben		Variable „*lage*“ wird deklariert mit oben
echo $lage	oben	Wert der Variablen wird referenziert
bash		Aufruf einer neuen Shell
echo $lage		Wert der Variablen wird referenziert. Die Variable „*lage*“ ist in der Sub-Shell nicht bekannt
lage=unten		Der Variablen „*lage*“ wird ein neuer Wert zugewiesen
echo $lage	unten	Wert der Variablen wird erneut referenziert
exit		Abmelden von der Sub-Shell, damit ist die alte Shell aktiv
echo $lage	oben	erneut wird der Wert referenziert. Der erste Wert ist erhalten geblieben

Quellcode 2 Verschachtelte Shells

Aus der Befehlsfolge ist zu entnehmen, das Variablen nur in der eigenen Shell Gültigkeit haben.

Abhilfe schafft der Befehl „export“.

Befehl	**Ergebnis**	**Beschreibung**
lage=oben		Variable „*lage*“ wird deklariert mit oben
echo $lage	oben	Wert der Variablen wird referenziert
export lage		Die variable „*lage*“ wird in die neue Shell mit „*übertragen*“
bash		Aufruf einer neuen Shell
echo $lage	oben	Wert der Variablen wird referenziert. Die Variable „*lage*“ ist diesmal der Sub-Shell bekannt
lage=unten		Der Variablen „*lage*“ wird ein neuer Wert zugewiesen
echo $lage	unten	Wert der Variablen wird erneut referenziert
exit		Abmelden von der Sub-Shell, damit ist die alte Shell aktiv
echo $lage	unten	erneut wird der Wert referenziert. Der erste Wert wurde geändert.

Quellcode 3 Verschachtelte Shells

2.3 Ausgabe

Zur Ausgabe von Werten verwendet man den Befehl „echo“.

2.3.1 Steuerungsparameter

-n gib keinen Zeilenvorschub am Ende der Zeile

-e ermöglicht die Interpretation der \-Notation

-E verhindert die Interpretation der \-Notation

Beispiele:

```
echo 1
echo hallo
echo hallo world
```

Enthält ein Text ein oder mehrere Leerzeichen, so ist dieser in Hochkommas zu setzen. Dabei wird zwischen einfachen und doppelten Hochkommas unterschieden. In ein Text in einfachen Hochkommas, so ist dieser statisch und wird nicht weiter ersetzt. Des Weiteren werden alle Trennzeichen durch eine Leerstelle ersetzt.

2.3.2 Zusätzliche Sonderzeichen

Darstellung	**ASCII-Code**	**Bedeutung**
\a	7	Piepston
\b	8	Backspace ←
\c	9	Ausgabe beenden (Kornshell, Bash SGI)
\f	12	Form Feed (Neue Seite)
\n	10	Zeilenvorschub
\r	13	Carriage Return (Wagenrücklauf)
\t	9	Tabulator
\v	11	Vertikaltabulator
\\	92	Backslash \
\0xxx	xxx	Zeichen mit ASCII-Code

Beispiele:

Befehl	**Ergebnis**
echo $HOME	/home/knoppix
echo '$HOME'	$HOME

echo "$HOME"	/home/knoppix
echo eins zwei drei	eins zwei drei
echo * wie geht's *	Der Stern wird als „ls *“ interpretiert. Also werden alle Dateien vorher und nachher angezeigt.
echo '* wie geht's *'	Keine Interpretation. Ausgabe wie geht's
echo "* wie geht's *"	Keine Interpretation. Ausgabe wie geht's
echo 'Pfadvariable: $PATH'	'Pfadvariable: $PATH
echo "Pfadvariable: $PATH"	Ausgabe des vollen Suchpfades

Folgende Variable „text“ wird mit einer Zeichenkette belegt.

text = 'DOS-Verzeichnis: \alpha\beta\gamma'

Kornshell:

Befehl	**Ausgabe**	**Beschreibung**
echo "$text"	DOS-Verzeichnis: lphetaamma	Das Zeichen \g wird ignoriert
echo '$text'	DOS-Verzeichnis: \alpha\beta\gamma'	Keine Erweiterung
echo $text	$text	

Bash:

Der Steuerungsparameter „-e“ erlaubt die Interpretation der \? Notation.

Befehl	**Ausgabe**	**Beschreibung**
echo "$text"	DOS-Verzeichnis: \alpha\beta\gamma	Das Zeichen \g bleibt erhalten
echo -e "$text"	DOS-Verzeichnis: lpheta\gamma	
echo '$text'	$text	Keine Erweiterung
echo –e '$text'	$text	
echo $text	DOS-Verzeichnis: \alpha\beta\gamma'	
echo –e $text	DOS-Verzeichnis: lpheta\gamma	

Die SGI-Bash bw. Kornshell erlaubt die Eingabe mittels:

```
echo "bitte Namen eingeben: \c"
read name
```

Unter Knoppix muss folgende Syntax verwenden:

```
echo –n "bitte Namen eingeben: "
read name
```

Wirkung:

Der Einlesenkursor steht hinter dem Doppelpunkt und nicht wie normal eine Zeile tiefer.

2.4 IFS

Diese Variable bestimmt die Trennung der Parameter bei der Übergabe an ein Kommando. Sie hat folgenden Wert in der Standardeinstellung:

- Leerzeichen

Bei der Ausgabe bzw. bei der Bestimmung der Parameter ein Zeichen aus der variablen IFS ermittelt, so wird dieses durch ein Leerzeichen ersetzt. Mehrere Leerzeichen hintereinander werden durch ein Leerzeichen ersetzt.

Beispiel: („_“ entspricht einem Leerzeichen)

e1='aaa,,,bbb:::ccc_ _ _ ddd'
echo $e1

aaa,,,bbb:::ccc _ ddd

Wird IFS auf ","gesetzt erhält man folgende Ausgabe:
IFS=:,

aaa _ _ _ bbb:::ccc _ ddd

Wird IFS auf ",:" gesetzt erhält man folgende Ausgabe:
IFS=:,

aaa _ _ _ bbb _ _ _ ccc _ ddd

2.5 Parameter für Scripte

Diese Variablen erlauben einen Zugriff auf die übergebenen Positionsparameter.

Positionsparameter	**Bedeutung**
$#	Anzahl der Argumente ohne kommandoname
$0	Name des Kommandos (kommandoname)
$1	1. Argument nach dem Kommandoname
:	:
$9	9. Argument nach dem Kommandoname
$@	alle Argumente ohne Kommandoname
$*	alle Argumente ohne Kommandoname
p	alle Argumente einzeln

2.6 Logische Verknüpfungen

Befehle bzw. Aufrufe von Scripten können mittels logischer Operatoren verknüpft werden. Dazu gibt es die „UND“ und „ODER“-Verknüpfung.

2.6.1 UND (&&)

Kommandofolgen werden von links beginnend abgearbeitet, solange der „Exitstatus 0“ liefert.

2.6.2 ODER (||)

Kommandofolgen werden von links beginnend abgearbeitet, bis ein Kommando als „Exitstatus 0“ liefert.

Beispiele:

```
who | grep guest   &&   write guest  < nachricht

(who | grep $1) || (who | grep $2) && echo "$1 oder $2 ist da"

(who | grep meier) || (who | grep schmidt) && write schulz  < nachricht
```

Quellcode 4 Wahrheitstabellen

Das erste Beispiel zeigt alle momentanen Nutzer im System. Diese Liste wird an das Kommando „grep“ weitergegeben. Dieser Befehl sucht den User „guest“. Wenn er gefunden wurde, so gibt es einen Returncode 0, also true, im anderen Fall die 1, also false. Wenn ein Benutzer „guest“ vorhanden ist, erhält dieser die Nachricht aus der Datei „nachricht“.

Das zweite Beispiel sucht die beiden Benutzer, die als Parameter übergeben wurden. Die Oder-Bedingung zeigt an, dass das zweite Kommando auch dann ausgeführt wird, wenn nur einer vorhanden ist.

Das dritte Beispiel sucht die beiden Benutzer „meier“ und „schmidt“. Wenn einer der beiden im Netz ist, so erhält der Benutzer „schulz“ eine Nachricht.

2. Befehl:

(who | grep meier) || (who | grep schmidt) && write schulz < nachricht

Die dazugehörige Wahrheitstabelle:

User	**Kommando1**	**Kommando2**	**Klammerausdruck UND**
?	Result = 0 ⇒ ok	Wird nicht gestartet	Result = 0 ⇒ ok

?	Result = 0 ⇒ ok	Wird nicht gestartet	Result = 0 ⇒ ok
?	Result ⇒ 0 ⇒ nichtok	Result = 0 ⇒ ok	Result = 0 ⇒ ok
?	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok

Aufgabe:

Tragen Sie bitte die passenden User in die erste Spalte.

3. Beispiel:

```
(who | grep meier) || (who | grep schmidt) || echo "schmidt und meier"
```

Quellcode 5 Beispiel Wahrheitstabelle

Die dazugehörige Wahrheitstabelle:

User	**Kommando1**	**Kommando2**	**Kommando3**
?	Result = 0 ⇒ ok	Wird nicht gestartet	?
?	Result = 0 ⇒ ok	Wird nicht gestartet	?
?	Result ⇒ 0 ⇒ nichtok	Result = 0 ⇒ ok	?
?	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok	?

Aufgabe:

Tragen Sie bitte die passenden User in die erste Spalte.

2.7 Read

Mit dem Read-Kommando werden Daten in eine oder mehrere Variablen gespeichert. Dabei wird die Eingabe durch die IFS-Variable getrennt. Normalerweise ein Leerzeichen.

Beispiele:

```
read xx
Mit der Eingabe
   1111

Testausgabe:  echo $xx  liefert 1111


read xx yy zz
Mit der Eingabe
   1111  2222   3333  4444

xx:    ?
```

```
yy:     ?
zz      ?
```

Quellcode 6 Read-Beispiele

Im zweiten Beispiel haben die Variablen folgende Werte:

xx: 1111
yy: 2222
zz: 3333 4444

2.8 Kommandoersetzung

Es besteht auch die Möglichkeit, die Ergebnisse von Kommando direkt in eine Variable zu speichern. Diese wäre eine Kombination aus Umleitung und Variablenersetzung.

Beispiele:

```
liste=$(ls)    # Inhalt des Verzeichnisses in liste
zeit=$(date)
z1=$(grep freundlich verkauf)
z2=$(grep freundlich verkauf | sort)
dateien=$(ls *.java)
```

Quellcode 7 Beispiele Kommandoersetzung

2.9 Vergleiche

2.9.1 Alphanumerische Vergleiche

Die normalen Abfragen werden immer alphanumerisch vorgenommen.

Abfrage	**wahr / falsch**
[[abc < abs]]	# wahr
[[abc > abcd]]	# falsch
[[123 < 147]]	# wahr
[[123 < 0147]]	# falsch
[[0815 > 3]]	# falsch
[[12345 <77]]	# wahr
[[püh < pöh]]	# wahr

2.9.2 Numerische Vergleiche

Für numerische Abfragen existieren folgende Anweisungen:

Abfrage	**Bedeutung**
x -lt -y	x ist kleiner als y
x -le -y	x ist kleiner oder gleich als y
x -eq-y	x ist gleich als y
x -ne-y	x ist ungleich als y
x -gt -y	x ist größer als y
x -ge-y	x ist größer oder gleich als y

Dabei bedeutet „lt“ less than.

Beispiele:

```
T1=23
T2=44
if [  $T1  -lt   $T2  ]
then
  echo okay
else
  echo false
fi
```

Quellcode 8 If-Beispiel

Ausgabe: okay

```
T1=23
T2=14
if [  $T1  -lt   $T2  ]
then
  echo okay
else
  echo false
fi
```

Quellcode 9 If-Beispiel

Ausgabe: false

3 Kontrollstrukturen

3.1 If-Anweisungen

Es sind die bekannten Konstrukte möglich. Wobei mindestens ein Leerzeichen zwischen der Bedingung und den Klammern sein muss.

Syntax:

```
if [  bedingung  ]
then
    kommando_1
else
    kommando_2
fi
```

```
if [  bedingung  ]
then
    kommando_1a
    kommando_1b
else
    kommando_2a
    kommando_2b
fi
```

Des Weiteren existiert noch die Anweisung „elif“, das entspricht der else-if-Anweisung.

Um Dateien zu überprüfen, existieren mehrere zusätzliche Bedingungen:

if [ausdruck datei]

Ausdruck	**Bedeutung**
-r <datei>	datei existiert und man hat Leserechte
-w <datei>	datei existiert und man hat Schreibrechte
-x <datei>	datei existiert und man hat Ausführungsrechte
-f <datei>	datei existiert und ist ein einfache Datei
-d <datei>	datei existiert und ist ein Verzeichnis
-h <datei>	datei existiert und ist ein symbolische Link
-c <datei>	datei existiert und ist ein zeichenorientiertes Gerät
-b <datei>	datei existiert und ist ein blockorientiertes Gerät
-u <datei>	datei existiert und für Eigentümer s-Bit gesetzt
-g <datei>	datei existiert und für Gruppe s-Bit gesetzt
-s <datei>	datei existiert und ist nicht leer

Beispiele:

Datei existiert und ist ein Verzeichnis:
if [-d datei]
then
fi

Datei existiert und ist eine normale Datei:
if [-f datei]
then
fi

Datei existiert und hat Leseberechtigung:
if [-r datei]
then
fi

Datei existiert und hat Schreibeberechtigung:
if [-w datei]
then
fi

Folgende Erweiterungen existieren

datei1 -nt datei2	datei1 ist neuer als datei2
datei1 -ot datei2	datei1 ist älter als datei2
datei1 -ef datei2	datei1, datei2 Links auf dieselbe Datei
-z string	string ist leer (zero)
-n string	string ist nicht leer

Die folgenden Beispiele funktionieren nur, wenn die Befehle in einer Scriptdatei geschrieben worden sind.

1. Beispiel:

```
echo -n "Datei $1 löschen? (ja/nein/abbruch): "
read antwort
[[ $antwort = a* ]] && exit 1
[[ -z $antwort  ||  $antwort = j* ]] && rm $1
```

Quellcode 10 Abfrage löschen Datei (Ja/Nein/ Abbruch)

Mit dem Befehl „echo“ wird die Frage ausgegeben. Das „-n“ verhindert den Zeilenumbruch. Die erste Abfrage bezieht sich auf die Prüfung, ob der este Buchstaben mit einem kleinen „a“ anfängt. Wenn ja,

wird das Script beendet. Ist die Anzwort leer oder fängt sie mit einem kleinen „j“, so wird die Datei gelöscht.

```
wert=815
[[ $wert = 4711  ||  $wert = 0815 ]]  &&  echo wahr
```

Einige Shell liefern für die obige Abfrage den Wert true, da sie automatisch eine Typumwandlung vornehmen. Für die Kornshell ist die Bedingung falsch. Die Bash gibt „wahr“ aus.

3.2 Switch-Bedingung

Statt verschachtelter if-Anweisungen erlaubt die Case-Anweisung eine übersichtlichere Programmierung:

Syntax:

```
case Bed in
(muster1)   kommandoliste
            ;;
(muster2)   kommandoliste
            ;;
esac
```

1. Beispiel:

```
Prompt = $0
echo -e "$prompt (ja/nein): "
read antwort

case $antwort in
 ([jJ])echo ja
       exit 0
       ;;

 ([nN])echo nein
       exit 1
       ;;

 (*)   echo rest
  exit 2
esac
```

Quellcode 11 Case-Beispiel

Aufgerufen wird dieses Script wird „bash case1 wollen Sie wirklich alles formatieren?“.

Der Ausdruck „$*“ gibt die komplette Parameterliste aus. Nach der Ausgabe kann man ja oder nein eingeben.

Die Abfrage [jJ] prüft, ob das erste Zeichen ein J ist. Die Abfrage „sonst“ mit (*) ist hier sehr wichtig, da damit alle Fälle aufgenommen werden, die nicht abgeprüft werden. Prinzipiell sollte dieser Fall nie eintreten.

Ein Alternative zur Syntax ist folgendes Script:

```
Prompt = $0
echo -e "$prompt (ja/nein): "
read antwort

case $antwort in
 j|J)  echo ja
       exit 0
       ;;

 n|N)  echo nein
       exit 1
       ;;

 *)    echo rest
   exit 2
esac
```

Quellcode 12 Case-Beispiel

2. Beispiel:

Gesucht ist ein Skrip zur Bestimmung der Anzahl der Tage eines Monats.

Lösung:

```
read monat
case $monat in
 (1|3|5|7|8|10|12)
        maxtage=31 ;;
 (* )
        maxtage=0;;
esac

echo “Anzahl der Tage ist $maxtage"
```

Quellcode 13 Anzahl der Tage eines Monats

3.3 For-Schleife

Es wird der Zählschleife eine Menge von Werten übergeben. Bei jedem Durchlauf wird einer Variablen der nächste Wert aus der Menge von Werten übergeben. Die Anzahl von Durchläufen ist an die Menge von übergebenen Werten gebunden. Die Anweisungen „Break“ und „Continue“ ändern die Abarbeitung.

Syntax:

```
for name in wert_1 wert_2 wert_3 wert_4
do
        kommandoliste
done
```

Beispiel:

Feste Schleifenanzahl durch:

```
for i in 1 2 3 5 6
do
   echo "Hallo"
done
```

Die Anweisung echo wird fünfmal ausgeführt.

Beispiel:

Wenn der Positionsparameter $* bzw. alle Parameter verwendet werden soll, wird die Zählschleife wie folgt notiert:

```
for name in $*
do
    kommandoliste
done
```

Beispiel:

Dateien Schleife:

```
liste=$(ls *.c)
for datei in $liste
do
    echo $datei
done
```

Quellcode 14 Beispiel For-Schleife

Feste Schleifenanzahl mit Abbruch

```
for i in 1 2 3 4 5 6
do
    if [  $i   -eq  4 ]
    then
       break
   fi
    print "Hallo"
done
```

Quellcode 15 Beispiel For-Schleife

3.4 While-Schleife

Bei einer While-Schleife wird die Bedingung vor jedem Schleifendurchlauf geprüft. Das bedeutet, das die Schleife eventuell keinmal durchlaufen wird. Des Weiteren kann unter Umständen eine Endlos-Schleife programmiert werden.

Kurzform: Schleife solange Bedingung erfüllt ist

Syntax:

while [Bed]
do
      kommandoliste
done

Break und Continue ändern die Abarbeitung
Beispiele:

```
i=1
while [ $i -lt 10 ]
do
  echo $i
done
```

Quellcode 16 While-schleife mit einer EndlosBedingung

```
i=1
while [ $i -lt 10 ]
do
  echo $i
  i=$[i+1]
done
```

Quellcode 17 Korrekte While-Schleife

Die folgende Schleife wird niemals durchlaufen:

```
i=12
while [ $i -lt 10 ]
do
  echo $i
  i=$[ i+1 ]
done
```

Quellcode 18 While-Schleife ohne Durchlauf

3.5 Until-Do-Schleife

Die Schleifenbedingung wird hier vor dem Rumpf angegeben und auch ausgewertet, sie muss invers zur While-Schleife angegeben werden.

Kurzform: Schleife bis Bedingung erfüllt ist

```
i=1
until [ $i -ge 10 ]
do
  echo $i
  i=$[i+1]
done
```

Quellcode 19 Korrekte Until-Do-Schleife

Ausgabe:

1
2
3
4
5
6
7
8
9

Die folgende Schleife wird genau einmal durchlaufen:

```
i=12
until [ $i -ge 10 ]
do
  echo $i
```

```
  i=$[i+1]
done
```

Quellcode 20 Until-Do-Schleife mit einem Durchlauf

Hinweis:

In der Bash auf der Knoppix-Version 3.4 wird die Schleife keinmal durchlaufen.

4 Arithmetik

Berechnungen können in verschiedenen Shells eingesetzt werden. Folgender Quellcode erzeugt nicht das korrekte Ergebnis:

```
k=12
k=$k+2
echo $k
```

Ergebnis: 12+2

Hinweis:

In der Kornshell kann man mit der Anweisung typeset –i k die Variable als Integer-Variable definieren. Dann wird die obige Anweisung korrekt ausgeführt.

4.1 expr

Standardmäßig wird folgende Syntax verwenden:

```
k=1
a=`expr $k +1`
```

Hier darf kein Leerzeichen vorkommen.

In der Bourne-Shell gibt es natürlich bequemere Eingaben:

4.2 Dollar-Ausdruck

Mit Hilfe des Dollar-Symbol kann man den obigen Ausdruck vereinfachen.

```
k=1
a=$[ k + 1 ]
```

Es dürfen beliebige Leerstellen eingefügt werden. Zudem wird hier kein Dollar für jede Variable benötigt.

4.3 Klammer-Ausdruck

Der Ausdruck innerhalb einer doppelten Klammer als arithmetische Anweisung interpretiert.

```
k=1
(( a= k + 1 ))
```

Es dürfen beliebige Leerstellen eingefügt werden. Zudem wird hier kein Dollar für jede Variable benötigt.

4.4 Operatoren

Folgende arithmetische Operatoren existieren:

+ Addition
- Substraktion
* Multiplikation
/ Ganzzahlige Division
% Modulo
+= Addition
-= Substraktion
*= Multiplikation
/= Ganzzahlige Division
%= Modulo

Folgende logische Operatoren existieren:

& AND-Operation
| OR-Operation
^ XOR-Operation
! NOT-Operation

Beispiele:

a=15
b=5

Beispiel	Ergebnis (c)	Bemerkung
((c = a + b))	20	
((c = a & b))	5	UND-Verknüpfung
((c = a && b))	1	Hier wird eine bool´sche Operation bzgl. der Kommando-folgen ermittelt.
((c = a | b))	15	ODER-Verknüpfung
((c = a || b))	1	Hier wird eine bool´sche Operation bzgl. der Kommando-folgen ermittelt.
((c = a ^ b))	10	XOR-Verknüpfung
((c = !(a+b)))	Hausübung	Negation
((c = a / b))	3	Ganzahlige Division

4.5 Stringfunktionen

In der Bourne-Shell gibt es folgende Funktionen zur String-Manipulation:

${#dat1} liefert die Anzahl der Zeichen in der Variable.

${dat1#muster}

Überprüfung des Musters in Text. Die Funktion liefert das kürzeste Muster im Text.

${dat1##muster}

Überprüfung des Musters in Text. Die Funktion liefert das längste Muster im Text.

${dat1%muster}

Überprüfung des Musters in Text. Die Funktion liefert das kürzeste Muster im Text bezogen auf das Ende.

${dat1%%muster}

Überprüfung des Musters in Text. Die Funktion liefert das längste Muster im Text bezogen auf das Ende.

Beispiel:

```
path=/usr/local/sbin/list
echo "Relativ zu /usr ${path#/usr/}"
echo "Name ${path##*/}"
echo "Verzeichis  ${path%/*}"
echo "Verzeichis, das sbin enthält  ${path%%/sbin*}"
echo "Leer  ${path%%/*}"
```

5 Felder

Felder werden in den Shells nicht explizit definiert. Bei der Zuweisung an ein Element i wird ein Array von 0 bis i-1 angelegt.

1. Beispiel:

c[3] = 3 # 3.te Element definiert
c[4] = 5 # 4.te Element definiert

Ausgabe:

echo ${c[3]}
3

Ausgabe in einer Schleife:

```
c[3] = 3       # 3.te Element definiert
c[4] = 5       # 4.te Element definiert
for i in 0 1 2 3 4
do
  echo ${c[i]}
done
```

Quellcode 21 Array mit einer For-Schleife

Ausgabe:
Leerzeile
Leerzeile
Leerzeile
3
4

2. Beispiel:

((c[10] = 3)) # 10.te Element definiert

6 Funktionen

Funktionen dienen dazu, Befehle unter einem Namen zusammenzufassen. Damit lassen sie sich mehrfach aufrufen.

Vorteile:

- Speicherung innerhalb eines Scriptes
- Schneller Aufruf
- Verwendung der Scriptparameter
- Verwendung der lokalen Variablen
- In der Shell definierte Funktionen können aufgerufen werden

Syntax:

Aufbau:
function name
{
   Anweisungen
}

1. Beispiel:

```
function list
{
   ls -aF
}

# mainprogram
ls > dat1
list
```

Quellcode 22 Beispiel Funktion

Das obige Beispiel speichert die Liste aller Dateien in die Datei dat1. Danach wird die Funktion „list“ aufgerufen.

2. Beispiel:

```
function readname
{
   echo -n "Bitte einen Namen eingeben:"
   read name
}


# mainprogram
readname
```

Quellcode 23 Beispiel Funktion

3. Beispiel:

Das nächste Beispiel durchläuft alle Parameter und gibt sie aus.

```
$ function vii
> {
>   for p
>   do
>     echo $p
>   done
> }
```

Aufruf des Scriptes:

```
$ vii 11   22    33    44        55
11
22
33
44
55
```

4. Beispiel:

Häufig sollen alle Parameter analysiert werden. Die Variablen $1 bis $9 erlauben nur den Zugriff auf die ersten neun Parametern. Abhilfe schafft der „shift-Befehl“. Er verschiebt die Parameter **und** ändert die Anzahl $#.

Beispiel:

```
n=0
while  [  $# -gt  0  ]
do
   ((  n=n+1 ))
   (( dat[n-1] =  $1 ))
   shift
done

for i in n
do
   echo ${dat[i-1]}
done
```

Quellcode 24 Array mit shift-Anweisung

Aufruf: bash loop 111 222 333 444 555 666

Dieses Skript soll die übergegeben Parameter in ein Array einfügen und danach ausgeben. Das Eintragen funktioniert, nur die Ausgabe entspricht nicht dem gewünschten Ergebnis. Es wird nur der letzte Eintrag ausgegebn. Ursache ist die For-Schleife. Sie funktioniert nicht wie in Java oder C. Man muss die For-Schleife in eine While-Schleife umwandlen.

```
n=0
while  [  $# -gt  0  ]
do
   ((  n=n+1 ))
   (( dat[n-1] =  $1 ))
   shift
done

i=0
while [ $i -lt $n ]
do
   echo ${dat[i]}
   (( i=i+1 ))
done
```

Quellcode 25 Array mit shift-Anweisung

Aufruf: bash loop 111 222 333 444 555 666

Ergebnis:

111
222
333
444
555
666

5. Beispiel:

Einlesen einer positiven Zahl. Dabei soll die Eingabe in eine Variable gespeichert werden:

1: Variante

```
function lese
{
   while true
   do
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  retcode=$i
}
```

```
lese
nr1=$retcode
nr2=$retcode
echo  "nr1: $nr1   nr2: $nr2"
```

Quellcode 26 Einlesen einer Zahl mit einer Funktion

Der Quellcode speichert in der Funktion das Ergebnis in einer Variablen. Diese wird nach dem Aufruf ausgewertet. Es wurde aber im obige Beispiel der zweite Aufruf für die Variable „nr2“ vergesssen.

Sinnvoller wäre ein echter Rückgabewert. Dies ist möglich, wenn man den Aufruf in Backtricks (`) setzt.

```
function lese
{
   while true
   do
     echo  -n  Bitte eine Zahl eingeben:
      read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  echo $i
}

# Main
nr1=`lese`       # korrekt
echo  "nr1: $nr1"
```

Quellcode 27 Einlesen einer Zahl mit einer Funktion

Dabei wurde zusätzlich noch ein Ausgabestring zur besseren Eingabe eingebaut.

Aufruf:

bash lese2

Eingabe:
123

Ergebnis: Bitte eine Zahl eingeben: 123

Die komplette Ausgabe wird als Rückgabewert übertragen. Das nächste Beispiel zeigt die korrekte Anwendung:

```
function lese
{
   while true
   do
     read i
     if [ $i -gt 0 ]
     then
        break
     fi
  done
  echo $i
}

# Main
echo  -n  Bitte eine Zahl eingeben:
nr1=`lese`       # korrekt
echo  "nr1: $nr1"
```

Quellcode 28 Einlesen einer Zahl mit einer Funktion

7 Lösungen

7.1 Wahrheitstabelle (meier, schmidt, und)

```
(who | grep meier) || (who | grep schmidt) && write schulz  < nachricht
```

Quellcode 29 Beispiel Wahrheitstabelle

Die dazugehörige Wahrheitstabelle:

User	**Kommando1**	**Kommando2**	**Klammerausdruck UND**
?	Result = 0 ⇒ ok	Wird nicht gestartet	Result = 0 ⇒ ok Echo wird ausgeführt
?	Result = 0 ⇒ ok	Wird nicht gestartet	Result = 0 ⇒ ok Echo wird ausgeführt
?	Result ⇒ 0 ⇒ nichtok	Result = 0 ⇒ ok	Result = 0 ⇒ ok Echo wird ausgeführt
?	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok Echo wird ausgeführt

7.2 Wahrheitstabelle (meier, schmidt, oder)

```
(who | grep meier) || (who | grep schmidt) || echo "schmidt und meier"
```

Quellcode 30 Beispiel Wahrheitstabelle

User	**Kommando1**	**Kommando2**	**Kommando3**
meier	Result = 0 ⇒ ok	Wird nicht gestartet	Wird nicht gestartet
meier schmidt	Result = 0 ⇒ ok	Wird nicht gestartet	Wird nicht gestartet
schmidt	Result ⇒ 0 ⇒ nichtok	Result = 0 ⇒ ok	Wird nicht gestartet
Keiner der beiden	Result ⇒ 0 ⇒ nichtok	Result ⇒ 0 ⇒ nichtok	Wird gestartet

8 Literatur

1) **Stephen R. Bourne**
Das Unix System V
Addison-Wesley
ISBN 3-925118-23-2

2) **Sebastian Hetze et al.**
Linux Anwenderhandbuch und Leitfaden für die Systemverwaltung
LunetiX Softfair
ISBN 3-929764-06-7

3) **Cameron Newham**
Learning the Bash Shell
A Nutshell Handbook
O´Reilly & Associates, 1998
ISBN 1-56592-347-2

Indexverzeichnis