

Wer hat denn da geSHELLt?

[Christian Hack](#)

Im Rahmen der Betriebssystem-Vorlesung hab ich mich im letzten Semester zum ersten Mal "richtig" mit der Linux/Unix-Shell auseinandergesetzt und war begeistert! Das ist natürlich schon fast eine Schande für jemanden, der sich seit drei Jahren mit Linux beschäftigt. Aber bisher habe ich sämtliche Skripte, die ich brauchte, einfach in Perl geschrieben. Shell blieb außen vor.

Für Routine-Aufgaben ist ein Shell-Skript aber oft besser geeignet oder schneller geschrieben als ein Perl-Skript. Deshalb möchte ich im folgenden die grundlegenden Elemente und den Aufbau anhand von einigen (sinnvollen?) Beispielen kurz erklären.

WICHTIG: TESTET NICHTS IM HOME-VERZEICHNIS!!!

Außerdem muss ich hier natürlich noch einmal darauf aufmerksam machen, dass ich selbstverständlich für keine entstandenen Schäden Haftung oder Verantwortung übernehme! Also Sicherheitskopie von allem was man kaputtmachen kann ...

Routine-Aufgaben sind oft nur eine immer gleiche Folge von verschiedenen Befehlen, diese kann man einfach in einer Datei zusammenfassen. Ein Beispiel hierfür ist die News-Seite des Portals, für die in regelmäßigen Abständen rdf-Dateien von verschiedenen Servern geholt werden müssen, dafür müsste ich eigentlich immer die beiden Befehle:

```
cd /usr/local/httpd/htdocs/news/rdfs/  
wget -N -i ../filelist
```

ausführen. Diese beiden Befehle stehen jetzt in der Textdatei get_rdfs.sh, die ich wiederum per CRON stündlich aufrufe. Wichtig dabei ist noch zu erwähnen, dass diese Datei ausführbar gemacht werden muss:

```
chmod 755 get_rdfs.sh
```

Ja und das war eigentlich schon das erste Shell-Skript. Zwar noch ohne Variablen usw. die kommen aber noch. Zugegebenermaßen ist dieses erste Beispiel noch wenig flexibel. So wäre es z.B. sinnvoll wenn man dem Skript eine Variable mitgibt in der steht, welches Input-File wget benutzen soll. Aber Rom wurde ja auch nicht an einem Tag erbaut ;-)

Und jetzt wird's variabel

Dazu muss hier erst einmal geklärt werden, wie die Shell Variablen verarbeitet bzw. welche Typen sie kennt. Und das ist sehr simpel - es gibt nämlich nur einen Typ: Zeichenkette (Sting). Der (Variablen)Name besteht aus einer Folge von Buchstaben, Ziffern oder Unterstrichen und darf nicht mit einer Ziffer beginnen. Variablen wird folgendermaßen ein Wert zugewiesen:

```
variable=zeichenkette
```

Das war's schon. Falls die Zeichenkette Leerzeichen enthält muss sie mit Anführungszeichen " " geklammert werden. Nun reicht es natürlich nicht aus Variablen zuzuweisen, man möchte ja auch darauf zugreifen. Und das geschieht durch das Vorstellen eines \$ vor den Variablenname:

```
VAR1="zeichenkette mit Leerzeichen"
```

```
echo $VAR1
```

Hier würde nun der Befehl **echo** den Wert von VAR1 ausgeben. Durch die Verwendung des \$-Operators wird die Variable also durch ihren Wert substituiert. Aber Achtung:

```
VAR2 = VAR1           :Inhalt von VAR2: VAR1
VAR2 = $VAR1         :Inhalt von VAR2: zeichenkette mit Leerzeichen
```

Die Shell belegt bereits von sich aus einige Variablen vor und stellt sie dem Benutzer zur Verfügung. Beispiele sind:

\$HOME	Name des Heimatverzeichnis
\$PATH	Suchpfad für ausführbare Dateien
\$MAIL	Name der Mail-Datei
\$?	Rückgabewert des zuletzt ausgeführten Kommandos
\$!	Prozessnummer des zuletzt gestarteten Programms
\$\$	Prozessnummer der betreffenden Shell (findet oft Verwendung zur Benennung von temporären Dateien)

Eine Liste mit allen vorbelegten Variablen erhält man mit dem Befehl **env**. Variablen lassen sich auch noch ganz leicht mit allem füllen, was andere Kommandozeilenprogramme so als Ergebnis liefern. Das geschieht indem das Kommando in Rückwärtsapostrophe eingeschlossen wird:

```
DATUM= `date`
```

Da so den Skripten schon eine gewisse Flexibilität mit auf den Weg gegeben werden kann, interessiert als nächstes natürlich wie man den Programmablauf beeinflusst oder dem Skript noch einige Parameter mit auf den Weg gibt. Siehe hierzu auch Abschnitt Übergabeparameter

Den Ablauf kontrollieren

Wie jede andere Programmiersprache kennt auch die Shell bestimmte Konstrukte um den Ablauf eines Programms zu strukturieren und zu beeinflussen. Bevor ich die verschiedenen Konstrukte im Einzelnen erkläre, muss ich jedoch noch ein paar Programme kurz erwähnen, auf die die Shell in diesem Zusammenhang zurückgreift:

read: liest die Eingabe bis zum Zeilenende ein und verteilt den String auf Variablen

expr: führt logische oder arithmetische Operationen aus

test: wertet einen Ausdruck aus und gibt falls wahr 0 sonst einen Wert !0 zurück (im Gegensatz zu C)

for

```
for WERT [in LISTE]
do
    BEFELHLSFOLGE
```

done

In die mit WERT bezeichnete Variable werden nacheinander alle Werte aus Liste zugewiesen. Für jeden dieser Werte aus LISTE wird dann der Schleifenrumpf (Programmstück zwischen do und done) einmal durchlaufen. Beispiel:

```
for wert in *.html
do
    cp $wert /tmp/backup
done
```

oder als Einzeiler formuliert:

```
for wert in *.html; do cp $wert /tmp/backup; done
```

while

```
while LISTE
do
    BEFELHLSFOLGE
done
```

Das while-Konstrukt ist sehr nützlich zum "Endlos"-Programmieren. So kann zum Beispiel in regelmäßigen Abständen überprüft werden ob irgendwelche Dateien existieren oder Statusinformationen in Log-Files geschrieben werden. Damit solche Endlos-Programme nicht eine Shell blockieren kann das komplette Programm beim Start mit & in den Hintergrund gestellt werden:

```
(while true
    echo date >>zeit.log
    sleep 20
done)&
```

Ein anderes wichtiges Einsatzgebiet für das while-Konstrukt ist das Auslesen von Dateien mit Hilfe des < Operators:

```
while read next
do
    echo $next
done < zeit.log
```

Wichtig für C-Kenner: Die Shell und ihre Kommandos interpretiert **0 als "logisch wahr"** und nicht wie in C die 1. Dies ist auch für das im folgenden behandelte **if** Kommando wichtig!

if -- test

```
if BEDINGUNG
then BEFEHLSFOLGE1
else BEFEHLSFOLGE2
fi
```

Um Bedingungen zu überprüfen benutzt die Shell das Kommando **test**. Mit diesem Kommando ist es möglich sowohl Ausdrücke die mit Zeichenketten und ganzen Zahlen als auch Dateien zu tun haben, auszuwerten. Die wichtigsten Optionen sind:

- **für Dateien:**

-r <datei>	Wahr, wenn <datei> existiert und für Benutzer lesbar
-w <datei>	Wahr, wenn <datei> existiert und für Benutzer beschreibbar
-x <datei>	Wahr, wenn <datei> existiert und für Benutzer ausführbar
-d <datei>	Wahr, wenn <datei> ein Verzeichnis ist
-f <datei>	Wahr, wenn <datei> existiert und eine gewöhnliche Datei ist
-s <datei>	Wahr, wenn <datei> existiert und eine Länge größer als 0 hat

- **für Zeichenketten**

-z <string>	Wahr, wenn <string> die Länge 0 hat
-n <string>	Wahr, wenn <string> eine Länge größer 0 hat
<string1> = <string2>	Wahr, wenn <string1> gleich<string2> sind
<string1> != <string2>	Wahr, wenn <string1> ungleich<string2> sind

- **und für Optionen mit ganzen Zahlen:**

<zahl1> -eq <zahl2>	Wahr, wenn die Zahlen <zahl1> und <zahl2> algebraisch gleich sind
<zahl1> -ne <zahl2>	Wahr, wenn die Zahlen <zahl1> und <zahl2> algebraisch ungleich sind
<zahl1> -gt <zahl2>	Wahr, wenn gilt: <zahl1> größer <zahl2>
<zahl1> -ge <zahl2>	Wahr, wenn gilt: <zahl1> größergleich <zahl2>
<zahl1> -lt <zahl2>	Wahr, wenn gilt: <zahl1> kleiner <zahl2>
<zahl1> -le <zahl2>	Wahr, wenn gilt: <zahl1> kleinergleich <zahl2>

case

```
case WERT in
    MUSTER1) BEFEHLSFOLGE1;;
    MUSTER2) BEFEHLSFOLGE2;;
    *) DEFAULT-BEFEHLSFOLGE;;
esac
```

Wichtig ist vor allem, dass jede Befehlsfolge mit zwei Semikolons abgeschlossen wird! Die aus C bekannte break-Anweisung ist bei Shell nicht nötig. Das folgende Beispiel liest die Datei /etc/httpd/httpd.conf zeilenweise aus und gibt nur die Zeilen aus, die kein Kommentar (=die nicht mit # beginnen) sind:

```
while read zeile
do
    case $zeile in
        [#]*) ;;
        * ) echo $zeile;;
done < /etc/httpd/httpd.conf
```

break, continue, exit

break	bricht eine Schleife ab
continue	überspringt den Rest des Rumpfes bis zum Schleifenende. Anschließend wird die Schleifenbedingung erneut durchgeführt
exit	verlässt eine interaktive Shell

Mit diesen Befehlen sollte sehr vorsichtig umgegangen werden, da sie indirekt Sprungbefehle sind und so die Übersichtlichkeit des Programms stark beeinträchtigen. In jedem Fall sollte die Verwendung gut dokumentiert werden.

Die Übergabeparameter

Man kennt es ja von "richtigen" Shellprogrammen wie zu Beispiel **cat**, dass dem Programm verschiedene Parameter (bei cat eine Liste von Dateien) übergeben werden, die es dann bearbeiten soll. Das gibt's für Shell-Skripte natürlich auch. Das Verarbeiten von solchen Übergabeparametern soll das folgende Beispiel verdeutlichen dazu wird eine Datei xecho mit folgendem Inhalt erstellt:

```
echo $3
echo $1 $1
echo $0-$1-$2-$5
```

und anschließend mit Parametern aufgerufen:

```
christian@mini:~>xecho erste "zweite dritte" vierte fuenfte
vierte
erste erste
xecho-erste-zweite dritte-
```

daraus wird ersichtlich, dass im Positionsparameter \$0 immer der Skriptname abgelegt ist. In \$1 bis \$x kommen dann alle durch ein Leerzeichen voneinander getrennten Zeichenketten, die beim Aufruf des Programms mit übergeben wurden. Auf \$4 wird überhaupt nicht zugegriffen, \$5 ist leer (zu wenig

Parameter übergeben).

Hier ist auch noch das Kommando **shift** zu erwähnen. Es schiebt die Reihe der Parameter um eine Stelle nach links, beginnend bei \$1, das aktuelle \$1 geht dabei verloren. Folgende Reihe soll dies verdeutlichen:

```
$0 $1 $2 $3 $4 $5 $6 ...
```

shift

```
$0 $2 $3 $4 $5 $6 ...
```

Zugriff durch: \$0 \$1 \$2 \$3 \$4 \$5 ...

Auch in diesem Zusammenhang belegt die Shell wieder einige Variablen vor:

\$# : Anzahl der übergeben Positionsparameter (im echox-Beispiel wäre \$#=4)

\$* : entspricht "\$1 \$2 \$3 \$4 ..." (alle Parameter als ein String)

\$@ : entspricht "\$1" "\$2" "\$3" "\$4" ... (nach Weitergabe an anderes Kommando die urspr. Anzahl der Parameter)

Falls die Positionsparameter \$i mit den Komponenten der Ausgabe eines Programms belegt werden sollen, hilft das Kommando **set** weiter:

```
set `date`
```

belegt die Positionsparameter folgendermaßen:

```
Sat Mar 16 12:15:00 CET 2002  
$1 $2 $3 $4 $5 $6
```

expr - die Shell kann zählen

Viele Programmierer benutzen while-Schleifen um bestimmte Ereignisse zu zählen. Doch so leicht wie in C/C++ (mit var=var+1 oder var++) geht's in der Shell leider nicht. Die Shell benutzt für arithmetische oder logische Operationen das Kommando **expr**. Dieser Befehl wertet Ausdrücke aus. So kann mit Shell-Variablen gerechnet werden:

```
SUMME= `expr $ZAHL1 + $ZAHL2`
```

Wir erinnern uns: Was zwischen den Rückwärtsapostrophen steht wird substituiert.

expr kennt folgende Operationen: + - * / %

Außerdem gibt es einige Besonderheiten:

- Vor und nach Operanten muss ein Leerzeichen stehen
- Da * von der Shell als Namensauflöser benutzt wird, muss bei der Verwendung mit expr ein \ davor stehen:

expr \$ZAHL1 * \$ZAHL2	ergäbe syntax error
expr 9*9	ergäbe 9*9
expr 3 * \(5 + 9 \)	ergäbe 42

the end

Natürlich gibt es noch viel mehr Tricks und Goodies, diese würden den Rahmen einer Einführung aber sprengen. In diesem Zusammenhang möchte ich noch auf die Kombination des hier vorgestellten mit Shell-Redirections (< >> >) und Pipelines (|) hinweisen. Wirklich gute und tricky Shell-Skripts liegen jeder Linux-Distribution bei, ein Blick ins /usr/bin/-Verzeichnis lohnt auf alle Fälle.

Links

- [Bash Reference Manual](#)
- [Advanced Bash-Scripting Guide](#)
- [LINUX IN A NUTSHELL Online-Shell-Befehlsreferenz](#) 

LinuxKP.org 16.03.2002