Shell-Programmierung für Fortgeschrittene

Stefan Harinko

4. Juni 2005

Inhaltsverzeichnis

1 unser erstes Shellskript **1**

2 weitere Variablensubstitutionen **2**

3 erweiterte Ausgabeumlenkung **2**

4 Programmierkonstrukte Überblick **3**
4.1 Exit Status und Befehlssequenzen 3
4.2 arithmetische Ausdrücke 4
4.3 Bedingungs-Ausdrücke 4
4.4 Verbindung mehrerer Ausdrücke 4
4.5 if 5
4.6 for 5
4.7 while und until 6
4.8 case 6
4.9 Funktionen 6
4.10 Ausblick 7
4.11 Quellen 7

ProSeminar

1 unser erstes Shellskript

Shellskript Ein Shellskript ist eine Textdatei, die von einer Shell ausgeführt werden kann. In der Datei `shellskript.sh` könnte stehen:

```
#!/bin/bash

echo 'Hallo ProSeminar Unix-Tools !'

exit 0
```

Möglichkeiten die Datei `shellskript.sh` auszuführen:

- Wenn die Datei mit `chmod +x` ausführbar gemacht wurde, kann das Skript mit `./shellskript.sh` gestartet werden.
- Ansonsten benutzt man `"bash shellskript.sh"`

Beispielaufruf

```
$ ./shellskript.sh
Hallo ProSeminar Unix-Tools !
$ bash shellskript.sh
Hallo ProSeminar Unix-Tools !
$
```

2 weitere Variablensubstitutionen

einfache Ersetzungen durch Variablensubstitutionen:

- ${**`Variable#pattern`**} entfernt das erste Pattern vom Anfang des Variableninhalts
- ${**`Variable##pattern`**} entfernt das längste Pattern vom Anfang des Variableninhalts

```
$ Variable=/home/user/datei
$ echo ${Variable#/*/}
user/datei
echo ${Variable##/*/}
datei
```

- ${**`Variable%pattern`**} das Pattern wird vom Ende her entfernt
- ${**`Variable%%pattern`**} das längste Pattern wird entfernt
- ${**`Variable/pattern/string`**} das erste Pattern wird durch string ersetzt
- ${**`Variable//pattern/string`**} alle Muster werden durch string ersetzt
- weitere Möglichkeiten findet man in der Manpage zu bash unter `"Parameter Expansion"`

Variablensubstitutionen erledigen die einfach Aufgaben für die man ansonsten ein externes Programm wie sed, awk, grep, perl, etc benutzen müsste.

3 erweiterte Ausgabeumlenkung

Ausgabeumlenkung von Standardausgabe und Standardfehler

```
2>&1
kopiert Standardfehler auf Standardausgabe
```

Die Auswertung erfolgt von links nach rechts. Will man Standardfehler und -ausgabe eines Programms in eine Datei umlenken muss man die Reihenfolge beachten. Richtig angewendet sieht ein Aufruf so aus:

```
Programm > datei 2>&1
```

Falsch ist:

```
Programm 2>&1 > datei
```

Da hier zuerst der Standardfehler auf die derzeitige Standardausgabe kopiert wird, die momentan der Bildschirm ist und danach erst in die Datei datei umgelenkt wird.

4 Programmierkonstrukte Überblick

- Exit Status und Befehlssequenzen
- Ausdrücke
  - arithmetische Ausdrücke
  - Bedingungsausdrücke
  - Verknüpfung von Ausdrücken
- Kontrollstrukturen
  - if
  - for
  - while und until
  - case
- Funktionen
- Ausblick

4.1 Exit Status und Befehlssequenzen

- Der Exit-Status eines Befehls ist 0, wenn er erfolgreich war und ungleich 0 bei einem Fehler.

- Befehlssequenzen oder Befehlslisten werden mit speziellen Operatoren getrennt und terminiert
  - `;` `&` `&&` oder `||` trennen Listen
  - `;` `&` oder `<newlines>` terminieren Listen

  Beispiel

  ```
  $ befehl1 || befehl2
  # befehl2 wird nur gestartet,
  # wenn befehl1 fehlschlug
  ```

- Wenn Befehl2 ausgeführt wird, war der Exit-Status von Befehl1 ungleich 0.

4.2 arithmetische Ausdrücke

- **`(( expression ))`** wird als arithmetischer Ausdruck ausgewertet, expression hat eine C-ähnliche Syntax
- einige mögliche Operatoren sind: `+ - * / % ++ -- == != > < >= <= && ||` Beispiel

```
(( 5>10 || 7>3 ))
```

4.3 Bedingungs-Ausdrücke

- **`[[ expression ]]`** ist ein Ausdruck, der eine Bedingung überprüft
- kann Strings, Variablen, Dateiattribute und auch arithmetische Ausdrücke testen
- wichtige Operatoren:
  - **`string1 == string2`**
  - **`string1 != string2`**
  - `>` und `<` testen immer lexikographisch größer oder kleiner
  - **`-n string`** ist wahr wenn die Stringlänge ungleich Null ist.
  - **`-a Datei`** ist wahr wenn Datei existiert
  - **`-d Verzeichnis`** ist wahr wenn Verzeichnis existiert

4.4 Verbindung mehrerer Ausdrücke

- **`( expression )`** gruppiert einen Ausdruck um die Auswertungsreihenfolge zu verändern
- **`! expression`** ist genau dann wahr wenn expression falsch ist
- `&& und ||` funktionieren auch zwischen zwei Ausdrücken

  Beispiel

```
  ! (( 3>10 )) || [[ 3 -gt 1 ]]   # ist true
! ( (( 3>10 )) || [[ 3 -gt 1 ]] ) # ist false
```

  Im zweiten Fall wird die ganze Ausdrucksgruppe negiert ist dann false, das sie vorher true wahr.

Hinweis: **`liste`** ist im Folgenden immer eine Befehlssequenz und oder eine Reihe von Ausdrücken

4.5 if

if Syntax

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

Beispiel

```
if (( "$#" == 0 ))
then
 echo keine Parameter angegeben
 exit 1
fi
```

4.6 for

for Syntax1

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

- word wird erweitert
- name wird nacheinander auf ein Element aus word gesetzt

Beispiel

```
for datei in skripts/*.sh
 do
  ./$datei
done
```

Dieses for ist auch ideal für die shelltypischen Aufgaben der Datei- und Verzeichnismanipulation.

for Syntax2

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

- expr1 wird ausgewertet
- expr2 solange bis sie 0 ergibt
- wenn expr2 != 0 wird liste ausgeführt
- dann expr3

Beispiel

```
for ((i=0; i<5; i++));do echo $i; done
```

gibt die Zahlen 0 bis 4 aus Diese for-Syntax ist die, wie man sie auch aus anderen imperativen Sprachen kennt.

4.7 while und until

Syntax

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

- while führt do-Befehlsliste solange aus bis liste nicht erfolgreich beendet
- until führt do-Befehlsliste solange aus bis liste erfolgreich läuft

Beispiel

```
while true; do echo true true; sleep 1;done
```

While und until sind sich sehr ähnlich, da man zur jeweiligen Umwandlung nur den Negationsoperator `!` vor `liste` stellen muss.

4.8 case

case syntax

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

Beispiel

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

Wie im Beispiel gesehen wird das Schlüsselwort case in vielen Start- und Stoppskripten verwendet.

4.9 Funktionen

Syntax

```
[ function ] name () { liste; }
```

Beispiel

```
#!/bin/bash
function name () {
echo "$# Argumente: $*"
}
name "mit zwei" Parametern
echo Ende
exit
```

Ausgabe 2 Argumente: mit zwei Parametern Ende

Funktionen haben leider keinen Rückgabewert. Man kann sich mit Tricks behelfen, in dem man die Werte, die man zurückgeben will in globale Variablen speichert. Oder man frägt den Exitcode ab, der aber nur einen begrenzten Zahlenbereich annehmen kann. Funktionen sind aber dennoch zur sinnvollen Gliederung größerer Shellskripte notwendig.

4.10 Ausblick

Mithilfe dieser Konstukte kann man einfach und schnell Shellskripte für wiederkehrende Aufgaben programmieren. Außerdem bietet die Bourne Shell unter anderem noch folgende Möglichkeiten :

- lokale Variablen, die mit dem Schlüsselwort localëingeleitet werden
- Arrays ähnlich wie in anderen Programmiersprachen mit initialisiert und verwendet (z.B. arrays[9]=1 weist an der zehnten Stelle den Wert 1 zu, den man mit echo $arrays[9] wieder ausgeben kann
- Signale, die mit **`trap`** abgefangen werden können. Man kann z.B. auf kill Signale reagieren, um vor dem Beenden temporäre Dateien, die eventuell angelegt wurden wieder zu entfernen.
- Debugging mit **`bash -n`** und **`bash -x`**: **`-n`** macht nur einen Syntax check, während **`-x`** jeden Ausdruck, der im Shellskript aufgerufen wird nochmal auf der Console ausgibt.
- named pipes, können benutzt werden damit zwei verschieden Prozesse miteinander kommunizieren können.
- Process Substitution, als Gegenstück zur Command Substitution schickt es die Ausgabe eines Prozesses an einen anderen `cat <(ls)` ist dann dasselbe wie `ls -l | cat` Man kann sich vorstellen, dass intern eine temporäre named pipe verwendet wurde, die zum Lesen benutzt wurde.
- Bash Version 3 kann reguläre Ausdrücke in `[[ ]]` . Eine sinnvolle Erweiterung, da dadurch ähnliche wie bei den verschiedenen Möglichkeiten der Variablensubstitution Aufrufe externer Programme wie perl,sed,awk,grep,etc nicht notwendig sind.

4.11 Quellen

Quellen

- man bash
- http://www.tldp.org/LDP/abs/html/