

# Shell (bash) als Sprache

Ziel: Erstellung einfacher Scripts mit der Shell

Voraussetzungen: Kenntnisse in der Bedienung

Zeit: Bis Weihnachten, eine Laborarbeit folgt nach den Ferien!

# Shell – Versuch einer Definition

Die Shell ist eine Software des Betriebssystems

Die Shell ist eine wichtige Benutzerschnittstelle zum System (Befehlsinterpreter, Befehlszeile)

Die Shell bietet Funktionen einer Programmiersprache (interpretierte Scriptsprache), geeignet zur Systemprogrammierung

Frage: Welche weitere Sprachen werden häufig zur Systemprogrammierung eingesetzt?

# Shell – Eigenschaften

Im Gegensatz zu DOS-Systemen gibt es unter Unix mehrere Shells (ash, csh, tcsh, bash, ...)

In /etc/passwd liegt der Eintrag der Loginshell (chsh zum Ändern)

Konfigurationsdateien: /etc/profile /etc/profile.d  
/etc/bashrc ~/.profile ~/.bashrc

Frage: Gibt es ein alias in deiner Shell?

# Wichtige Sonderzeichen

/ - Wurzelverzeichnis (root directory)

. - aktuelles Verzeichnis

.. - übergeordnetes Verzeichnis

~ - eigenes Homeverzeichnis

~Benutzer – Homeverzeichnis des angegebenen Benutzers

# Joker

\* - für eine beliebige Zeichenkette, auch die Leere

? - für ein Zeichen

[abc] – für a, b oder c

[a-z] – für ein Kleinbuchstabe

[!a-z] – Negation, also für ein Zeichen, das kein Kleinbuchstabe ist

# Datenstrom – Umleitung (redirect)

< - stdin (prog <Eingabedatei)

> - stdout (prog >Ergebnisse)

2> - stderr (prog 2>Protokoll)

weiter:

>> - Ausgabe an die angegebene Datei anhängen

&> - stdout und stderr in eine gemeinsame Datei umleiten

# Pipe

Die Ausgabe des ersten Prozesses wird zur Eingabe  
des zweiten Prozesses

```
find ~ | grep hallo
```

```
find /usr/include -type f | grep stdio
```

# Kommandosubstitution (command substitution)

Die Ausgabe eines Prozesses wird zum Befehl oder Teil eines Befehls

``Aufruf`` - z.B. `grep printf `find /usr/include -name "*.h" | grep stdio``

`$(Aufruf)` - z.B. `bilder=$(ls *.jpg)` (nur bash)

xargs als Ersatz für die Kommandosubstitution:

`find /usr/include -name "*.h" | grep stdio | xargs grep printf`

# Programmierung

Interaktiv in der Befehlszeile

Script in einer Textdatei

Ausführen mit `sh Datei` oder durch einen direkten Aufruf, nachdem das Recht zur Ausführung auf die Datei gesetzt wurde (z.B. `chmod +x Datei`)

`sh Script`

`./Script`

# Kopf der Scriptdatei

Spezifikation der Shell, damit bei der Ausführung des Skripts die richtige Shell benutzt wird:

```
#!/bin/sh
```

```
#
```

```
# Kommentar
```

```
#
```

```
...
```

# Daten

Man begnügt sich meist mit Zeichenketten!

Gültigkeit der Variable in der aktuellen Umgebung

Definition durch Zuweisung oder mit declare

Zugriff durch  $\$Varname$  oder  $\${Varname}$

# Ein- und Ausgabe

echo Ausdruck-Ausgabe

read Variablenname

# Variablen – Beispiele 1

```
a=Hallo
```

```
echo $a
```

```
b=$aHallo
```

```
echo $b
```

```
b=${a}Hallo
```

```
echo $b
```

# Variablen – Beispiele 2

```
c=Hallo du  
echo $c
```

```
c="Hallo du"  
echo $c
```

```
c=Hallo\ du  
echo $c
```

```
c="Hallo\ du"  
echo $c
```

# Variablen – Beispiele 3

```
d="$a du"
```

```
echo $d
```

```
e='$a du'
```

```
echo $e
```

# Variablen – Beispiele 4

```
f=`ls`
```

```
echo $f
```

```
pfad=`pwd`
```

```
aktuellen Pfad zuweisen
```

# Besondere Variablen 1

\$? - Rückgabewert des letzten Prozesses

#! - PID des zuletzt gestarteten  
Hintergrundprozesses

\$\$ - PID des Shellscripts bzw. der Shell (eigene)

\$0 – Name des Scripts bzw. der Shell

\$# - Anzahl der übergebenen Argumente

\$1 ... \$9 – Argumente 1 bis 9

# Besondere Variablen 2

`$@` - Argumentenliste mit möglichen Leerzeichen

`$*` - Argumentenliste mit Problemen mit  
Leerzeichen

`shift` schiebt die Parameterliste um eine Stelle nach  
links. `$1` geht dabei verloren!

# Einige Umgebungsvariablen - set

HOME

HOSTNAME

LOGNAME

PATH

PS1

PWD

UID

# Gültigkeitsbereich (scope)

Aufgabe: Schreib ein Script, das ein Verzeichnis zum Suchpfad hinzufügt. Teste das Script.

Exportiere den Suchpfad nach der Änderung

`export var`

???

Eine Umgebungsvariable wird nie an die aufrufende Umgebung zurückgegeben, sie können nur an augerufene Umgebungen übergeben werden.

# Arithmetische Ausdrücke

echo  $$(1+2)$  – alte Schreibweise (deprecated)

echo  $$((1+2))$  – ab der Version 2.0

expr 1 + 2 – Mit einem externen Programm,  
unabhängig von einer Shell!

# Kontrollstrukturen

Alternativen (einfach und mehrfach)

Schleifen (kopfgesteuert, Durchlauf von Listen, ...)

break und continue wie in C

goto gibt es keines

Bedingungen sind Aufrufe von Programme, deshalb  
gibt es das Programm test, um Bedingungen zu  
erstellen

# test

Wertet die Argumente aus und gibt als Wahrheitswert true (wahr) den exit-Status 0 zurück, sonst ein Wert verschiedenen von 0

Logische und relationale Argumente bzw. systembezogene Abfragen

test Ausdruck

[ Ausdruck ] (alternative Schreibweise)

--> man 1 test

# Einfache Alternative

if Befehlsliste1

then

    Befehlsliste2

else (else optional)

    Befehlsliste3

fi

# Beispiel - if

```
if test -e /
```

```
then
```

```
    echo "/ existiert"
```

```
else
```

```
    echo "/ existiert nicht"
```

```
fi
```

# Beispiele - if

```
if [ $# -eq 2 ]; then
    echo "2 Argumente"
elif [ $# -eq 3 ]; then
    echo "3 Argumente"
else
    echo "andere Anzahl von Argumenten"
fi
```

# Befehlslisten

Befehlssequenz

Befehle, getrennt von folgenden Trennzeichen:

;

&

&&

||

Zeilenvorschub

# Befehlslisten

Der Rückgabewert einer Befehlsliste ist der Rückgabewert des letzten Befehls

Befehl1 && Befehl2 (Befehl2 wird ausgeführt, wenn Befehl1 0 zurückgibt)

Befehl1 || Befehl2 (Befehl2 wird ausgeführt, wenn Befehl1 nicht 0 zurückgibt)

# case

case Ausdruck in

pattern1) list ;;

pattern2) list ;;

...

esac

Die Muster zur Auswahl können Joker enthalten.

Es können auch Alternativen angegeben werden:

pattern1 | pattern2 )

# for

```
for var in String
```

```
do
```

```
    Befehlsliste
```

```
done
```

wird auf „in String“ verzichtet, werden die  
Argumente des Skripts durchlaufen!

# Beispiel for

```
for i in 1 "2 3" 4
```

```
do
```

```
  echo $i
```

```
done
```

# while

```
while Befehlsliste1
```

```
do
```

```
    Befehlsliste2
```

```
done
```

Befehlsliste2 wird solange ausgeführt, wie Befehlsliste1 das Ergebnis 0 zurückgibt.

# until

until Befehlsliste

do

    Befehlsliste

done

Befehlsliste2 wird solange ausgeführt, wie Befehlsliste1 ein Ergebnis verschieden 0 zurückgibt.

# Links

<http://www.faqs.org/docs/bashman/bashref.html>

[http://www.lugbz.org/content/sections/workshops/bash-intro-2003-04-26/slide\\_cover.html](http://www.lugbz.org/content/sections/workshops/bash-intro-2003-04-26/slide_cover.html)

<http://www.linuxfibel.de/bash.htm>

[http://www.wachtler.de/skript\\_tools/node23.html](http://www.wachtler.de/skript_tools/node23.html)

