EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Dr. Eberle Zentrum für digitale Kompetenzen

DataBASHing - Scripting in der BASH

Martin Raden

Einführung von

- Variablen, `echo` und '..' in Generelles
- Parameter Expansion `${}, $(), $(())` In der BASH
- Loops, if-else, Skriptdateien, ... in BASH-Scripting

Video: (en) Protesilaos Stavrou - BASH Parameter Expansion [12 min]

Video: (en) Pedagogy - Variables In Shell Scripting | store output of a command to a variable | $ [16 min]

Generelles

Im Allgemeinen gilt in (fast) jeder shell:

- **Variablen** sind
  - Platzhalter für sich ändernde Information
  - **ohne Leer- und Sonderzeichen** benannt, d.h. matchen die RegEx "`^[a-zA-Z][\w_]*$`"
  - werden mittels "`$`"**-Präfix** angesprochen, z.b. `echo $SHELL`
  - sind i.d.R. uppercase
- wichtige Standard-/**Umgebungsvariablen** (environment variables) die immer da sind:
  - `$HOSTNAME` - Name des Computers
  - `$HOME` - eigenes Nutzerverzeichnis
  - `$PATH` - Liste von Verzeichnissen ("`:`"-separated), in denen nach ausführbaren Programmen gesucht wird
  - `$PWD` - aktuelles Verzeichnis
- **Eigene Variablen** (local variables) können über "`=`"-Zuweisung (ohne "`$`"-Präfix) erzeugt und anschliessend via "$"-Präfix verwendet werden, z.B. `Heute=Freitag; echo $Heute`
  - Wichtig: **keine Leerzeichen** zwischen Variablenname, "`=`", und Wert bzw. im Wert; **Ansonsten** müssen führende oder zwischenliegende Leerzeichen im Wert via **quoting** eingeschlossen werden, z.B. `Text=" mit Leerzeichen "; echo "> $Text <"`
  - Über "`=`"-Zuweisung können Variablen auch **überschrieben/geändert** werden; sogar erweitert via Selbstreferenz, z.B. `Heute=Freitag; Heute="ist $Heute"; echo $Heute`
- `echo` = Ausgabe von Text und Variablen
  - single `'..'` double `".."` quotes
    - in **double quotes** werden Variablenreferenzen ersetzt, z.B. `echo "Meine shell ist $SHELL"`
    - alles in **single quotes** wird einfach nur ausgegeben, z.B. `echo 'shell Variable = $SHELL'`

  - " `-e` " = Interpretation von Spezialzeichen mit führendem " `\` ", wie z.B. Zeilenumbruch " `\n` " oder Tabulator " `\t` ", z.B. `echo "Erste\nZeile"; echo -e "Nächste\nZeile"`
  - " `-n` " = kein Zeilenumbruch am Ende der Ausgabe
- backticks "`..`" = liefern die Ausgabe des eingeschlossenen Kommandos, z.B. um dieses in einer Variable abzuspeichern, z.B. `meineShell=` ` echo $SHELL ` `; echo "Meine Shell ist $meineShell"`

Achtung MacOs BASH

Apple liefert aus lizenzrechtlichen Gründen nur eine veraltete Version der Bash aus, die teilweise für einige der folgenden Bash-Skripting Teile dieses Kurses nicht ausreicht (Version >= 4 nötig; check via " `bash --version` "). Daher müssen sie ggf. jetzt ihre **bash auf Mac OS aktualisieren** (Alternative Anleitung).

In der BASH

In BASH (sollte man aber Folgendes tun)

- " `${xyz}` " - **Zugriff** auf **Variable** " `xyz` " und weitere Manipulation (geschweifte Klammern)
  - ermöglicht genauere Verwendung von Variablen in und an Strings z.B. `echo "_${USER}_"` vs. `echo "_"$USER"_"`
- **Bash Parameter Expansion** ermöglicht die **Manipulation von Variablenwerten**! (komplette Online-Dokumentation mit
- `${x:-"schnurps"}` - liefert den Wert von Variable " `x` " oder, wenn Variable nicht verfügbar, den Wert "schnurps"
- `${#x}` - Anzahl Zeichen des in Variable " `x` " gespeicherten Wertes
- `${x//y/z}` - **ersetzt alle** matches von " `y` " durch " `z` " in Variable " `x` ", z.B. `stand=1:1; echo ${stand//1/2}`
  - wenn " `/` " statt " `//` ", dann nur erster match ersetzt, z.B. `stand=1:1; echo ${stand/1/2}`
  - "BASH-regex" bzw. wildcards:
    - " `*` " - ein oder mehrere Zeichen egal was, z.B. `stand=12:3; echo ${stand//*:/x-zu-}`
    - " `?` " - exakt ein Zeichen. z.B. `x="nein neun"; echo ${x//e?/o}`
- `${x:i:l}` - **substring** der Länge " `l` " von Variable " `x` " ab Position " `i` " (**0=Anfang**) z.B. `x=H2O; echo ${x:1:1}`
  - Längenangabe ist optional ==> Suffix ab i, z.B. `x="A=BC"; echo ${x:2}`
- `${x^^}` oder `${x,,}` - alles **upper** oder **lower case**
  - nur Anfang wenn nur ein " `^` " oder " `,` "
- `${!x}` - indirekter Zugriff, d.h. der WERT von Variable " `x` " wird ausgewertet und nach einer Variable gesucht, die so heisst, z.b. `a=b; b=1; echo ${!a}`

Zudem

- `$(xyz)` - **Befehl** " `xyz` " **ausführen** (runde Klammern)
  - z.B. echo "Anzahl lokaler Verzeichnisse = $(ls -l | grep -c '^d')"
  - ersetzt backtick-Notation und ist expliziter
- `$((3+4))` - arithmetische Berechnungen
  - alternativ = " `expr` " Befehl, welcher das Ergebnis direkt ausgibt (Leerzeichen zwischen ALLEN Formelteilen!), z.B. `expr 3 + 4`
  - oder man kann auch den " `let` " Befehl verwenden, welcher das Ergebnis direkt in einer Variable speichert, z.B. `let "A=3+4"` (hier muss i.d.R. gequotet werden!)

> Tutorials <

Schauen sie doch mal in dieses kompakte

- Online-Tutorial zu Parameter Expansion!

BASH-Scripting

Bash scripting direkt in der Kommandozeile:

- fast alle hier vorgestellten bash-Kommandos können auch direkt in der Konsole eingegeben werden und müssen NICHT in einer Datei gespeichert und anschliessend ausgeführt werden.
- Bsp: `for f in /tmp/*; do echo "- $f is a temporary file or folder"; done`

Bashskript/-datei - z.B. Bespieldatei `05-substring.sh` zum Anschauen!

- Textdatei mit Liste von Kommentaren, bash und shell Befehlen
- **1.Zeile = Shebang** mit Ausführungsinfos = `#!/usr/bin/env bash`
- **# - Kommentar**anfang - alles danach wird ignoriert (es sei denn, es ist in einem String, d.h. gequotet!)
- **Argumente** für Aufruf möglich, welche über folgende Variablen zugreifbar sind
  - `$1`, `$2`, .. = 1., 2., ... Aufrufargument
  - `$0` = Skriptname (im Aufruf, d.h. ggf. mit Pfad etc.)
  - `$#` = Anzahl der Aufrufargumente
  - `$@` liefert Array/Liste aller gegebenen Aufrufargumente (z.B. für for-loop Iteration)

Aufruf:

- via **bash call**, z.B. "`bash 05-substring.sh`" (für heute erstmal der Standardfall!)
- direkt (falls als "*executable*" markiert, dazu später mehr). z.B. "`./05-substring.sh`"
  - hierbei wird der Shebang ausgelesen (s.o.) und das entsprechend Programm mit der Skriptdatei als Argument aufgerufen
- via "`source`", z.B. "`source 05-substring.sh`"
  - *ACHTUNG*: `source` kopiert den INHALT des Skriptes und führt die Befehle direkt in der AKTUELLEN bash Konsole aus. Damit kann es zu Nebeneffekten kommen (z.B. ein "`exit`" Kommando im Skript *schliesst die aktuelle shell*!)
  - kann auch einfach nur "`.`" verwendet werden, z.B. "`. 05-substring.sh`"

Prozessstrukturen

- `if` - **Verzweigung** über Ja/Nein Test ala "`if [ TEST ]; then A; else B; fi`", wobei `A` und `B` einzelne Anweisungen oder Anweisungssequenzen ("`;`"-getrennt) sein können
  - *ACHTUNG*: **Leerzeichen** rund **um die eckigen Klammern** sind wichtig!
    - die eckigen Klammern sind eigentlich nur Kurznotation für das Programm "`test`" und seinen Rückgabewert
  - "`else`"-Zweig ist optional
  - **Mehrere Tests** können in **eigene**n `[]`-**Blöcke**n mit "`&&`" (und) bzw. "`||`" (oder) zusammengeführt werden, z.B. "`if [ TEST1 ] && [ TEST2 ]; then ...`"
  - Standardtests (Möglichkeiten für `TEST` in obigem Aufruf)

    - **Stringvergleiche** - `"$HOME" = "${PWD}"` oder `"${X}" != "lala"`
        - Beachten: Variablenzugriffe i.d.R. quoten, da Leerzeichen, Pfade, etc. schnell zum Problem werden!
        - `=`, `!=`, `<` (lexikographisch), `>`, `-n X` (string `X` ist leer), `-z X` (string `X` ist nicht leer, z.B. `-z "$HOME"`)
    - **Zahlenvergleiche** via
        - `-eq` (==), `-ne` (!=), `-lt` (<), `-gt` (>), `-le` (<=), `-ge` (>=), z.B. "`${#x} -gt 2`"
    - "`!`" - **Negierung** des nachfolgenden Tests z.B. "`! ${#x} -gt 2`" ist das gleiche wie "`${#x} -le 2`"
    - **Datei-/Verzeichnistests**
        - `-e` / `-d` = Datei / Verzeichnis existiert, z.B. `if [ ! -e /tmp/jo.txt ] || [ -s /tmp/jo.txt ]; then`
        - `-r` / `-w` = Datei ist lesbar / beschreibbar
        - `-x` = Datei ist ausführbar
        - `-s` = Datei ist leer (zero size)
- `for` - **Wiederholung** ala "`for x in LISTE; do A; done`"
  - "`LISTE`" ist hierbei eine Liste von Strings (white-space separated, also auch Zeilenumbrüche möglich!), die jeweils als Werte für die Laufvariable (hier "`x`") gesetzt werden, bevor Aufruf(sequenz) "`A`" jeweils ausgeführt wird
  - "`A`" kann wieder Sequenzblock sein
  - **explizit**e Liste: `for i in 1 5 26; do echo "${HOME:0:${i}}"; done`
  - Liste **via call**: `for f in $(ls /tmp); do echo "in tmp liegt ${f}"; done`
  - (Datei)Liste **via wildcards**: `for f in /tmp/*; do echo "$f is a temporary file or folder"; done`
  - Liste **via Array-Variable**: `for ARG in $@; do ...`
- `exit` - **bricht** das Script an dieser Stelle **ab** (implizit am Ende des Skripts aufgerufen)
  - liefert einen **"return" oder "error code"** an das aufrufende Programm, um den **Programmstatus** wiederzugeben
  - **0** = default = "**alles gut**"
  - **!= 0 = error code** = programmspezifische Codierung von Fehlern (im einfachsten Fall einfach immer exit 1 im Fehlerfall)
- Was es sonst noch gibt
  - `while` - loop
  - `until` - loop
  - `read` - liest einen Text von der Kommandozeile in eine Variable (alles bis "ENTER" gedrückt), z.B. `read UserInput`
  - `case` - multiple Verzweigung
  - array-Variablen
- `function` - Funktionsdefinition zur Automatisierung und Programmverkürzung
  - z.B. `function MYNAME { A }`
  - "`A`" steht für eine beliebige Aufrufsequenz
  - die Funktion kann (genau wie ein Skript) eigene Argumente via `$1-$9` etc. aufrufen
  - Aufruf der Funktion (innerhalb des Skripts NACH der Funktionsdefinition) als wäre es ein Programm, z.B. "`MYNAME 'lala' 1`" (hier mit zwei Argumenten aufgerufen)

Input Streams

- Zugriff auf `STDIN` via dummy file "`/dev/stdin`"
  - z.B. NL=$(cat /dev/stdin | wc -l); echo "you piped $NL lines. . . "
  - speichern sie obigen call in einer Datei '`05-countLines.sh`' ab (vielleicht direkt `vi` bzw. `nano`! ;))

  - dann kann dieses Skript in einer pipe verwendet werden
    * z.B. `ls -l | bash ./05-countLines.sh | tr " " "\n"`

> Tutorials <

Dieses umfangreiche

- Online Tutorial zu Bash Scripting von Ryan Chedwick

liefert viele Anwendungsbeispiele, kleine Aufgaben (Activities) und Hintergrundinformationen.
