Shell Scripting:

Wo:

- Shell-Befehle *direkt* auf der Befehlszeile *eintippen*
  (werden sofort ausgeführt und dann wieder „vergessen“)
- Shell-Befehle als Body eines *Alias* oder einer *Funktion*
  (typischerweise definiert in **`.bashrc`** ==› werden am Beginn jeder Shellsitzung geladen)
  ==› Ausführen durch Aufruf des Aliases / der Funktion
- Shell-Befehle *in einem eigenen File* speichern (*„Shell-Script“*, wie **`.bat`** unter Windows)

Zu Shell-Scripts:

- Filenamen-*Endung* ist *egal*, meist *keine* oder **`.sh`**
- *Kommentare* mit **#**
- *Erste Zeile* sollte Interpreter-Angabe sein: **`#!/bin/sh`** oder **`#!/bin/bash`**
  (damit es auch dann die **`sh`** oder **`bash`** verwendet, wenn der User **`csh`** oder **`ksh`** nutzt)
- *Einrückung und White Space trennt Worte* ist aber davon abgesehen ist *meist egal*
- *Zeilenvorschübe* sind bei **`for`**, **`if`** usw. *zwingend*, statt Zeilenvorschub ist auch **`;`** erlaubt

2 Möglichkeiten der Ausführung:

- Wie ein „echtes“ Programm mittels *Eingabe des Filenamens oder vollen Pfades*
  ==› *Ausführung in einer neuen Sub-Shell*,
  keine Auswirkung auf das Environment und Working Dir der aktuellen Shell-Session
- Ausführung mit **`.`** *filename* (*Zwischenraum* dazwischen) oder **`source`** *filename*
  ==› *„Sourcen“ des Scripts*, d.h. *einlesen durch die aktuelle Shell* wie wenn eingetippt,
  kann *Environment und Working Dir* der aktuellen Shell-Session *ändern*!
  Achtung auf **`exit`** im Script: *Beendet die aktuelle Shell-Session*, nicht nur das Script!
  Sourcen braucht Lese-Rechte, keine Ausführ-Rechte
  `.` *filename* oder **`source`** *filename* *in einem Script* wirkt ähnlich wie **`#include`**
  (zum *Aufteilen großer Scripts* auf mehrere Files)

Beides *optional mit Argumenten* nach dem Script-Name
Beides macht Suche im **PATH**, wenn *filename* ein Name *ohne* Verzeichnis ist.
Analog:

- **`(`** *befehle* **`)`** führt die *befehle* *in einer Subshell* aus
  Nützlich, wenn man beispielsweise *nur für einen Befehl* ein **`cd`** machen will
  Befehle in einer *Pipeline* oder in **`$(`**…**`)`** laufen *immer* in einer Subshell!
- **`{`** *befehle* **`}`** führt die *befehle* *in der aktuellen Shell* aus

Shell-Skripte mit bestimmten Namen in bestimmten Verzeichnissen
werden *automatisch* in bestimmten Situationen *ausgeführt*:
Beispiel: **`.bashrc`** und **`.bash_profile`** im *Home-Verzeichnis*:
Wird *jedesmal beim Starten* einer interaktiven Shell-Session / beim Einloggen gesourced

Variablen:

- Immer *als Text* gespeichert (auch bei Zahlen)
  (**bash** kennt auch Variablen-Typen, Arrays, …)
- **$***name* oder **${***name***}** wird *textuell durch den aktuellen Inhalt* der Variable *name ersetzt*
  (*unbekannte* Variablen werden *durch nichts* ersetzt, *keine* Fehlermeldung,
  wenn Fehler gewünscht **${***name***:?***fehlermeldung***}** nutzen)
- Viele Möglichkeiten beim Ersetzen, siehe **man bash**:
  – *Default-Werte* bei leerer Variable
  – *Ersetzung von Teilen* (z.B. Filenamen-Extension) des Variablen-Inhalts
  – *Weglassen von Teilen* des Variablen-Inhaltes
  – *Änderung von Groß/Kleinschreibung* des Variablen-Inhalts
- Wenn der Variablen-Inhalt *Zwischenräume* usw. enthält: *Zerfällt in mehrere Worte!*
  (außer in **" "**).
- Spezialtrick **eval** *text*
  Nachdem alle Variablen in *text* ersetzt wurden, wird der Ergebnis-Text
  ein zweites Mal evaluiert (nach **$** und anderen Shell-Konstrukten durchsucht)
- Menge aller gerade gesetzten Variablen: *„Das Environment“* (anschauen mit **env**)
  – Wird vom Betriebssystem *pro Prozess* verwaltet
  – Kann *von allen aufgerufenen Programmen* (Subprozessen) *gelesen* werden
  – Kann von einem Programm oder der Shell *nur für sich und Subprozesse geändert*
    werden (kann *nicht* im Vater-Prozess geändert werden!)
  – Umfasst im weitesten Sinne auch das *Working Dir*, **umask**, **ulimit** usw.

Arten von Variablen:

- Von der *Shell gesetzte oder gelesene Variablen* (**HOME**, **PWD**, **PATH**, **PS1**, **RANDOM**, …)
- Übliche *Variablen mit System-Bedeutung* oder *Bedeutung für ausgeführte Programme*
  (z.B. *Ländereinstellungen*, *Username*, *Verzeichnis für temporäte Files*, …)
- *Selbst definierte* Variablen
- *Variablen mit Shell-Sonderbedeutung* (meist nicht änderbar):
  $*n* … *n*-tes *Argument* von Scripts oder Funktionen (wie **argv[***n***]**)
  $0 … *Filename* des Shell-Scripts (wie **argv[0]**)
  $* … *alle* Argumente außer **$0** (ohne **" "** als *n* Worte, in **" "** als 1 Wort)
  $@ … dasselbe, aber auch in **" "** als *n* Worte
  $# … *Anzahl* der Argumente (wie **argc**)
  $? … *Exit-Status* des letzten ausgeführten Programms (eine Zahl 0 … 255)
  … und ein paar mehr

Setzen von Variablen:

Keine separate Deklaration nötig!

name=*wert befehl …*

> **Achtung:** *Keine Zwischenräume* rund um **=** !
> Setzt *name* auf *wert* und führt dann den *befehl* mit dem neuen Wert aus.
> Die Zuweisung gilt *nur für das Environment von diesem einen befehl*,
> *nicht* für die aktuelle Shell!

name=*wert*

Setzt *name* auf *wert* *für die aktuelle Shell*,
aber *nicht* für die Environments von Subprozessen.

export *name*=*wert*

Setzt *name* auf *wert* *für die aktuelle Shell und alle zukünftigen Subprozesse*.

for *name* **in** *worte*

Schleife: Setzt *name* der Reihe nach auf jedes der *worte*.

read *name* ...

Liest Input vom Terminal (oder aus einem File wenn umgeleitet)
und speichert ihn (wortweise) in *name*
(viele Optionen)

unset *name*

name wird *wieder vergessen*

Shell-Programmstrukturen:

if *testbefehl*
then *befehle*

elif *testbefehl*
then *befehle*

else *befehle*

fi

Selbsterklärend.
Der **else**-Teil ist *optional*, der **elif**-**then**-Teil ist *optional und wiederholbar*.

for *name* **in** *worte*
do *befehle*
done

Schleife über die einzelnen Worte in *worte*

1. *Expandiert* *worte* (Variablen-Ersetzung, **$(**...**)**, Filenamen-Wildcard-Auflösung, ...)
   in *worte* ist *optional*, wenn man es weglässt, wird implizit **in "$@"** verwendet
   ==> Schleife über *alle Worte der Befehlszeile* / *alle Argumente der Funktion*
2. Setzt die *Variable* *name* *der Reihe nach auf jedes einzelne Wort* in *worte*
3. Führt für *jedes* dieser Worte einmal die *befehle* aus

Effekt:

worte ist ein *Filenamen-Pattern*, z.B. ***.c**

==> *befehle* wird *für jeden dazupassenden File* einmal ausgeführt
(mit dem jeweiligen Filenamen in der Variable *name*)

worte ist eine Variable, die *mehrere Worte* enthält, z.B. **"$@"**

==> *befehle* wird für jedes Wort in der Variable einmal ausgeführt
(bei **"$@"** : *Für jedes Argument des Scripts* / der Funktion)

worte ist ein Befehl in **$(…)**, der *mehrere Worte/Zeilen als Output* liefert
Beispiel **$(find . -name Makefile)**
==› *befehle* wird für jeden **Makefile** unterhalb **.** einmal ausgeführt
Bei sehr großem **find** usw.: Problem *maximale Befehlszeilen-Länge*!
Außerdem: Klappt nicht für *Filenamen mit White Space*!
Siehe auch **seq** im nächsten Kapitel

while *testbefehl*
do *befehle*
done
Schleife: Führt *befehle immer wieder* aus solange *testbefehl Erfolg* liefert.

until *testbefehl*
do *befehle*
done
Wie **while**, aber mit *negierter Bedingung* (läuft solange *testbefehl Fehler* liefert).
Achtung: Auch **until** testet *vor* jedem Durchlauf, *nicht* am Ende der Schleife!

Bei **if**, **for**, **while** und **until**:
- Die *befehle* können *beliebig viele Befehle* sein, die *über beliebig viele Zeilen* gehen (bzw. beliebig viele Befehle können mit **;** aneinandergereiht werden)
- Aber man kann auch das ganze Konstrukt mit **;** *in eine Zeile* schreiben:

```
for dir in $(find . -type d) ; do echo -n "$dir: " ; ls $dir | wc -l ; done
```

break oder **break** *n*
Verlasse die innerste / die innersten *n Schleifen*, mach *nach der Schleife* weiter

continue oder **continue** *n*
Mach *sofort den nächsten Durchlauf* der innersten / der *n*-ten umgebenden *Schleife*.

case *word* **in** … **esac**
Fallunterscheidung. Lassen wir weg.
Häufiger Befehl, mächtig (mit Pattern Matching), aber üble Syntax…

Befehle primär für Scripts:

echo oder **printf**
Ausgabe der (expandierten) Befehlszeile
Häufig: **echo -n** … *ohne Zeilenvorschub* am Ende, nächster Output direkt dahinter

test … oder **[** … **]**
Prüft verschiedenste *Bedingungen*, siehe **man test**
Trick, um *name* auf gleich **blabla** zu prüfen, wenn *name* auch *leer* sein kann
[x${*name***} = xblabla]** (Warum? Geht auch eleganter …)
Achtung: *Zwischenräume sowohl innerhalb als auch außerhalb* der **[ ]** *notwendig*!

true und **false**
Tut nichts und endet sofort mit Exit Code **0** bzw. **1**

basename *name* liefert den *Filenamensteil* (ab dem letzten **/**) von *name*
basename *name ext* dasselbe, zusätzlich wird *ext am Ende entfernt* falls vorhanden
dirname *name* liefert den *Directory-Teil* (vor dem letzten **/**) von *name*
liefert **.** wenn *name* keinen **/** enthält

Es wird *nicht* geprüft, ob *name* tatsächlich existiert
Oft in **$(** … **)** verwendet

mktemp (für Files) und **mktemp -d** (für Directories)

Sicheres Anlegen eines *neuen Files* oder eines *neuen Directories*
mit einem neuen, *zufälligen Namen*.
Gibt den neuen Namen auf stdout aus, Verwendung daher meist mit **$(**…**)** :
Beispiele: **tmpname=$(mktemp)** oder **cd $(mktemp -d)**

pushd *dirname* und **popd**

Stack-artiges Hin- und Zurückwechseln zwischen Directories.

exec *befehl …*

Führt *befehl statt der Shell* aus (direkt im Shell-Prozess, *nicht* als Subprozess)
Immer die *letzte ausgeführte Zeile des Scripts*:
Wenn *befehl* endet, *endet der Shell-Prozess*

exec < *filename* oder **exec >** *filename* usw. (*ohne befehl* in der Zeile!)

Startet eine *Eingabe- / Ausgabe-Umleitung* für *alle nachfolgenden Zeilen* des Skripts
(d.h. der gesamte Rest des Scripts liest Tastaturinput von *filename*
oder *speichert seinen Bildschirmoutput* in *filename*)

exit oder **exit** *n*

Beendet das Script (mit dem *Exit Code* des letzten Befehls davor oder Exit Code *n*)

return oder **return** *n*

Beendet die Funktion (analog zu **exit**)

seq *last*
seq *first last*
seq *first step last*

Liefert als *Output* die *Zahlen* von *first* bis *last* (mit Schrittweite *step*)
Oft in Verbindung mit **$(**…**)** und **for** für *Zählschleifen*:
for i in $(seq $#)

shift oder **shift** *n*

Schiebt die Aufrufs-Parameter **$1 $2 $3** … um eins (bzw. um *n*) *nach links*:
$1 geht verloren, **$2** wird **$1**, **$3** wird **$2** usw.
Das letzte **$***i* ist danach undefiniert, **$#** wird um 1 kleiner.
Häufig verwendet zum *Entfernen von Befehls-Optionen* nach deren Erkennung
oder *in Schleifen über alle Argumente*, die pro Durchlauf das **$1** verarbeiten

expr und **bc**

Befehle zum *Rechnen* (in der **bash** selten nötig, **bash** kann selbst rechnen)

Here-Dokumente:

„Here-Document“ =
Gesamter *Tastaturinput* (**stdin**) für einen Befehl *direkt ins Script* geschrieben

befehl **<<***endwort*
inputzeile1
inputzeile2
…
endwort

Als *endwort* wird häufig **EOF** verwendet.

${**..**}** und **$(…**)** im Inputtext wird expandiert,
außer man schreibt **<<"***endwort***"** statt **<<***endwort* (dann wird der Input 1:1 übernommen)

Erstellen eines *neuen Files* samt Inhalt in einem Skript:

cat >*newfile* **<<***endwort*
…
endwort

Für *einzeiligen Input* geht es *einfacher*:

echo *input* **|** *befehl* oder **echo** *input* **>***newfile*

Definition von Funktionen:

function *funcname* **()**
{
…
}

Das **function** ist *optional*.

Keine expliziten Parameter: *Zugriff auf die Argumente* mit **$1**, **$2**, … bzw. **$*** und **$@**
Funktionen haben *keinen Returnwert*, sondern einen *Exit Code* (**0** … ok, nicht **0** … Fehler)
Aufruf wie ein normales Programm oder Script (ohne **()** !)
Eine Funktion hat das Environment mit dem umgebenden Script *gemeinsam*
(in *beide* Richtungen), außer man deklariert eine Variable in der Funktion **local**.

Die bash kann viel mehr!
- *Arrays* (indiziert oder assoziativ) und arithmetische *Berechnungen*, echte *Zählschleifen*, …
- *Stack für Working Dir's* mit **pushd** und **popd**
- *Aufruf-Optionen* mit **getopts** verarbeiten
- …