# Shell-Skripte in der Bash

## *Starten von Skripten*

Mit den Skripten **.bashrc** und **.bashrc_user** haben wir die ersten Bash-Skripte bereits kennen gelernt. Diese werden **automatisch** (beim Starten der Shell) aufgerufen.

Skripte können wie folgt auch **explizit** gestartet werden:

1. **source <*file*>**: Liest die Kommandos aus der Skript-Datei **<*file*>** und führt es in der aufrufenden Shell aus. Das Skript muss dafür nicht das Execute-Flag in den Berechtigungen gesetzt haben! (siehe z.B. auch **.bashrc**)
2. **<*file*>**: Ist das Execute-Flag (kann mit **chmod** geändert werden) gesetzt, können Skripte wie andere Kommandos auch einfach durch Aufruf ihres Namens gestartet werden. Das Skript blockiert während der Ausführung die aufrufende Shell. Das Skript muss im Suchpfad des Systems gefunden werden oder der Dateiname muss als vollständig qualifizierter Pfad angegeben werden.
3. **<*file*>&**: Ist das Execute-Flag der Datei gesetzt, kann ein Skript auch im Hintergrund laufen, d.h. die aufrufende Shell läuft im Vordergrund weiter.

## *Einfache Shell-Skripte*

Ein Skript kann von verschiedenen Shells aufgerufen werden, z.B. aus der Bash oder aus einer C-Shell. Da die Syntax dieser Shells unterschiedlich ist, muss in einem Skript zu verwendende Kommandointerpreter (also die Shell, unter der das Skript laufen soll) stehen. Aus diesem Grunde wird als erste Zeile eines Skripts immer der Kommandointerpreter erwartet. Der Syntax lautet:

**#!<*interpreter*>**

Die Einstellung wird als "*Shebang*"-String bezeichnet. Soll die Bash benutzt werden (was im Rahmen des Praktikums immer der Fall ist), lautet die erste Zeile:

```
#!/bin/bash
```

Diese Einstellung setzt voraus, dass die Bash in dem Verzeichnis **/bin** installiert wurde. Der Pfad ist entsprechend anzupassen, wenn die Installation in einem anderen Verzeichnis erfolgte.

Die Verwendung der Syntax einer spezifischen Shell in einem Skript ist somit unabhängig von der das Skript aufrufenden Shell.

Ein erstes Skript könnte wie folgt aussehen:

```
#!/bin/bash
echo "Bitte geben Sie den Vornamen ein: "
read VORNAME
echo "Bitte geben Sie den Nachnamen ein: "
read NACHNAME
echo "Hallo $VORNAME $NACHNAME!"
```

# Hilfsblatt zu Kontrollstrukturen in der Bash

Die Bash unterstützt folgende Kontrollstrukturen:

## *if / else / fi:*

Eine **`if`**-Abfrage hat eine bedingte Ausführung zur Folge.

**Syntax:**

```
if condition
  then
    statements
[elif condition2
  then
    statements ]
[else
  statements ]
fi
```

ebenfalls möglich ist die folgende Syntax:

```
if condition; then
  statements
fi
```

## *case:*

Wählt eine Befehlsfolge in Abhängigkeit von einer Variablen aus.

**Syntax:**

```
case expression in
  pattern1 )
     statements;;
  pattern2 )
     statements;;
  * )
    defaultStatements
esac
```

## *select:*

**`select`** baut eine Menüstruktur auf und führt eine Auswahl aus diesem Menü aus.

**Syntax:**

```
select name [in list]
 do
   statements, can use $name
```

```
done
```

Ein Beispiel:

```
#!/bin/bash
# lese Inhalt des aktuellen Verzeichnis
# in die Variable DIR ein
DIR=$(ls)
# Baue ein Menue mit allen Dateien auf:
select CHOICE in $DIR
do
  echo $CHOICE
  if [ -f $CHOICE ]; then
    less $CHOICE
    break
  else
    echo "Select a regular file"
  fi
done
```

## *for-Schleife:*

In einer for-Schleife wird eine Befehlsfolge so oft ausgeführt, wie Elemente in einem Array von Strings gefunden werden. Als Variable wird ein Element aus dem String-Array genommen.

**Syntax:**

```
for name [in list]
  do
    statements
  done
```

**`list`** ist in dem obigen Beispiel das String-Array, **`name`** ist ein Element mit dem in der Schleife gearbeitet werden kann.

Die Angabe der Liste ist optional. Wird nichts angegeben, so wird als Standardwert für **`list`** das Array **`$@`** (die *Liste der Eingabeparameter* des Skriptes) angenommen.

## *while-Schleife:*

In einer **`while`**-Schleife wird eine Befehlsfolge wiederholt, solange eine Bedingung erfüllt ist.

**Syntax:**

```
while condition; do
    statements
```

```
done
```

## *until-Schleife:*

In einer **`until`**-Schleife wird eine Befehlsfolge wiederholt, bis eine anzugebende Bedingung erfüllt ist.

**Syntax:**

```
until condition; do
    statements
done
```

**`while`**- und **`until`**-Schleifen sind also äquivalent, wenn die **`condition`** negiert wird.

## *conditions:*

Die Bedingungen in der **`if`**-Abfrage und in den **`while`**- und **`until`**-Schleifen sind Rückgabewerte anderer Befehle (anderes Shell-Skript oder eines Executables). Deshalb sollten auch Ihre Skripts immer definierte Rückgabewerte besitzen. Im fehlerfreien Fall **muss eine 0** zurückgegeben werden, sonst ein Wert ungleich 0. Dies ist auch die Konvention für den Exit-Code von Executables (also z.B. C-Programme).

Der Rückgabewert des letzten Befehlsaufrufes wird in der Variable **`$?`** gespeichert. Er kann bei der Skript-Ausführung auch bei der Ausgabe von Rückgaben von anderen Programmen, Skripten und Befehlen verwendet werden.

In der Regel muss bei der Auswertung von Bedingungen das Kommando **`test`** verwendet werden, welches eine Vielzahl von Überprüfungen ermöglicht: z.B.

- Vergleichen von Strings,
- Auswertung und Behandlung von Dateiattributen,
- arithmetische Vergleiche.

Beispiel: Prüfen, ob der Pfad **`file1`** existiert:

```
test –a file1
echo $?
```

Ist die **`test`**-Bedingung erfüllt, existiert also die Datei, wird von **`test`** der Wert **`0`** zurückgegeben, sonst eine **`1`**. Der Rückgabewert kann durch Verwendung des **`echo`**-Befehls sichtbar gemacht werden.

Man könnte also auch folgendes formulieren:

```
if test –a file1; then
  echo file1 existiert.
else
  echo file1 existiert nicht
fi
```

Siehe dazu den *Bash Guide for Beginners*, Kap. 7.1 oder das integrierte Manual

```
man test
```

Hinweis: Achten Sie darauf, ausführbare Dateien (Systemprogramme und Skripte) nicht **`test`** zu nennen. Wollen Sie Ihr Programm aufrufen ohne den vollständigen Pfad anzugeben, wird womöglich (je nach Einstellung der Variable **`PATH`**) der Befehl **`test`** aufgerufen, der keine Ausgabe erzeugt!

Der Befeh **`test`** …. kann durch die Schreibweise **`[`** … **`]`** (eckige Klammern) abgekürzt werden.

D.h.:

```
if test –a file1
```

ist gleichbedeutend mit:

```
if [ –a file1 ]
```

Achtung: Hinter [ und vor ] muss ein Leerzeichen stehen, sonst gibt es einen Syntax-Fehler!

Wie von anderen Programmiersprachen bekannt, können Rückgabewerte auch logisch verknüpft werden:

| | |
|---|---|
| `statement1 && statement2` | (logische Und-Verknüpfung) |
| `statement1 \|\| statement2` | (logische Oder-Verknüpfung) |
| `! statement` | (Negierung) |

## *functions:*

Lange Skripte können schnell unübersichtlich werden.
In einem Bash-Skript können für wiederkehrende Abläufe Funktionen gemäß dem Motto DRY (*Don't Repear Yourself*) geschrieben

```
function myfunc {
   ls -als $1
}
```

und dann an einer beliebigen Stelle im Skript aufgerufen werden:

```
myfunc /bin/bash
```

Funktionen werden in der Shell ausgeführt, in der auch das Skript läuft, das die Funktion beinhaltet. Die übergegebenen Parameter werden in der Funktion mit **$1**, **$2**, … referenziert. Mit dem Befehl **return** kann man die Funktion verlassen und einen Wert zum Aufrufer zurückgeben. Die Auswertung erfolgt analog zu der von Bedingungen über die Variable **$?**.

Alternativ kann ein Resultat auch per globaler Variable an den Aufrufer zurückgeliefert werden oder per **echo**-Befehl in eine Datei geschrieben und von dort ausgelesen werden.

In einer Funktion können Skript-interne Variablen (z.B. **LOCVAR="value"**) und globale Variablen (z.B. **export GLOBVAR="value"**) manipuliert werden. Eine globale Variable im Skript, die in der Funktion verändert wird, hat nach Verlassen der Funktion den Wert, den sie in der Funktion erhalten hat. Innerhalb von Funktion sind lokale Variablen (**local LOCFUNCVAR="value"**) möglich.

## *Arithmetische Ausdrücke:*

Die Variablen der Shell sind Zeichenketten und nicht typisiert. Trotzdem können in der Shell arithmetische Operationen ausgeführt werden. Dabei werden intern Integer-Werte im Long-Format verwendet, wobei etwaige Überläufe nicht abgefangen werden.

Zur Auswertung stehen die Shell-Befehle **expr** (externes Shell-Kommando) und **let** (internes Shell-Kommando) zur Verfügung.

```
let y=y+2        # Aufruf des internen Kommandos let
x=`expr $x + 1` # Aufruf des externen Kommandos expr
let "z += 4"     # Quotes fuer Leerzeichen & spez. Operatoren
```

Man beachte die Nutzung der Anführungszeichen: `…` (*accent grave*) bedeutet, dass das Ergebnis das das Kommando in den Anführungszeichen 1:1 durch dessen Ergebnis ersetzt wird. Hingegen ist "…" eine Zeichenkette, die in dem Bsp. oben als Eingabe für den Befehl **let** dient.

Daneben gibt es noch eine verkürzte Schreibweise:

```
z=$(($z-5))   # verkuerzte Schreibweise (ab Bash 2.0)
```

# Referenzen

Siehe z.B.:

M. Garrelts: Bash Guide for Beginners, GNU Public Licence (via Stud.Ip im Bereich der Vorlesung verfügbar)

Da das Thema Shell-Programmierung zum üblichen Curriculum akademischer Einrichtungen im Informatik-Bereich gehört, findet man auch zahlreiche kurze und gute Anleitungen im Netz, die als Orientierungshilfe dienen können.