# Shell - Programmierung

Andreas Poenicke

14. April 2025

## Inhaltsverzeichnis

# 1 Erste Sprachelemente

Die Shell ist nicht nur eine einfache Kommandozeile, sondern ein Interpreter der Shell-Skript-Sprache. Wie bei Python kann man mit diesem Interpreter interaktiv arbeiten, d.h. mit der Kommandozeile, aber auch Programme schreiben, die dann von der Shell ausgeführt werden.

Die Shell-Skript-Sprache ist durch den POSIX-Standard standardisiert, allerdings bieten viele Shell-Implementierungen Erweiterungen, die über diesen Standard hinaus gehen. Die folgenden Ausführungen beziehen sich nur auf die standardisierten Sprachelemente.

An einigen Stellen werden in diesem Dokument englische Begriffe in Anführungszeichen verwendet. In diesen Fällen sind es die Begriffe mit denen die Sprachelemente in der original Dokumentation bezeichnet werden. Dies ermöglicht eine einfachere Suche nach dem Thema z.B. in der `man`-Page der `bash`.[1]

## 1.1 Kommentare

Wie bei jeder Programmiersprache gibt es bei der Shell die Möglichkeit den Code zu kommentieren. Kommentare funktionieren wie in Python: Das Kommentarzeichen ist `#`, alles was danach bis zum Zeilenende folgt wird ignoriert.

```
~$ echo a b c
a b c
~$ echo a # b c
a
```

## 1.2 Ausgabe

Eine einfache Ausgabe erreicht man durch den Befehl `echo`.

`echo` gibt einfach die, durch Leerzeichen getrennten, übergebenen Argumente durch *ein* Leerzeichen getrennt aus:

```
~$ echo Dies  ist    ein Text
Dies ist ein Text
```

Hierbei ist zu beachten, dass dieser Befehl als eigenes ausführbares Programm existiert, aber auch in der Shell selbst eingebaut ist. Die Shell verwendet (ohne Pfadangabe) die eigene Implementierung, die sich vom ausführbaren Programm leicht unterscheiden kann. Das Bedeutet aber auch, die korrekte Beschreibung des Befehls findet man z.B. unter `man bash` nicht `man echo`.

Der Befehl `type` zeigt einem wie ein Befehl von der Shell interpretiert werden würde. (`type` selbst ist wieder ein „shell builtin“)

```
~$ type echo
echo ist eine von der Shell mitgelieferte Funktion
~$ type -t type
builtin
```

[1] Nach dem Aufruf von `man bash` kann man mit der Taste `/` einen Suchbegriff eingeben.

## 1.3 Formatierte Ausgabe

Mehr Kontrolle über das Ausgabeformat erhält man bei dem Befehl `printf`. Hier wird die Ausgabe durch einen Formatstring wie bei der C-Funktion `printf()` kontrolliert. Auch Python bietet diese Möglichkeit der Formatierung, die Python f-Strings sind dort aber vorzuziehen. Mehr zu der genauen Formatierung findet man in der Dokumentation der `printf()` C-Funktion (`man 3 printf`).

```
~$ printf "Name %s, Alter %d\n" "John" "30"
Name John, Alter 30
```

## 1.4 Variablen

Natürlich kennt die Shell auch Variablen.[2] Die Shell kennt allerdings keine verschiedenen Variablentypen. Die Werte sind immer Strings, die allerdings in bestimmten Zusammenhängen auch als Integer-Werte interpretiert werden können.

### 1.4.1 Definition

Variablen werden in der Shell Werte mit `=` zugewiesen.

Wichtig: Es dürfen hierbei keine Leerzeichen vor oder nach `=` stehen!

```
~$ VAR=iable
~$ wert=10
~$ wert2 = 3 # Durch das erste Leerzeichen wird wert2 als Befehl interpretiert
bash: wert2: Kommando nicht gefunden.
~$ wert2= 3 # Durch das Leerzeichen wird 3 als Befehl interpretiert
bash: 3: Kommando nicht gefunden.
```

Wichtig: Es gibt in der Shell keine undefinierten Variablen! Variablen, denen kein Wert zugewiesen wurde, werden ohne Fehler als leerer String interpretiert. Tippfehler können hier ohne besondere Fehlermeldung zu überraschenden Ergebnissen führen.

### 1.4.2 Zugriff

Der Zugriff auf den Inhalt von Variablen erfolgt über das Zeichen `$`. Der Variablenname kann dabei direkt dem `$`-Zeichen folgen, oder in geschweiften Klammern angegeben werden.

```
~$ VAR=wert
~$ echo $VAR
wert
~$ echo ${VAR}voll
wertvoll
```

Wie im Beispiel oben zu sehen, wird die Form mit geschweiften Klammern insbesondere dann benutzt, wenn der Variable etwas direkt folgt, und der Name damit nicht richtig interpretiert werden kann.

[2] In der Dokumentation werden Sie in diesem Zusammenhang immer wieder der Begriff „Parameter"finden. Unter Parametern werden hier verschiedene Entititäten verstanden, die Werte enthalten können. Variablen sind dann „Parameter“ mit einem Namen.

### 1.4.3 „QUOTING“

Variablen werden von der Shell durch Ihre Werte ersetzt, d.h. mit dem $-Zeichen gibt es ein weiteres Zeichen mit einer besonderen Bedeutung. Auch hier hilft „Quoting“ um dem Zeichen seine besondere Bedeutung zu nehmen. Allerdings gibt es hier einen entscheidenden Unterschied zwischen den verschiedenen Möglichkeiten des „Quoting“.

```
~$ a=1
~$ echo \$a
$a
~$ echo '$a'
$a
~$ echo "$a"
1
```

In Anführungszeichen behalten die Zeichen $, ` und \ ihre besondere Bedeutung, d.h. die Shell gibt die Zeile aus, ersetzt aber vorher die Variablen durch ihren Wert.[3]

Das Zeichen $ leitet, wie wir später sehen werden, auch andere Formen der Ersetzung ein.

# 2 Ein Beispiel

Häufig beschränkt sich der Einsatz der Skript-Sprache auf kurze Ausdrücke in der interaktiven Shell. Ein typisches Beispiel wäre

```
~$ for i in *.JPG; do mv $i ${i/.JPG/.jpg}; done
```

in dem weitere Sprachelemente verwendet wurden:

## 2.1 „Pathname Expansion“

Die Shell kennt Suchmuster, die vor dem Ausführen der Zeile durch passende Filesystem Einträge ersetzt werden. In den Suchmustern können dabei die folgenden Ausdrücke verwendet werden:

***** beliebig viele beliebige Zeichen. (Dies schließt kein Zeichen ein.)

**?** genau ein beliebiges Zeichen.

**[1-9], [a-k],...** genau ein Zeichen aus dem Bereich

Im Beispiel oben wird also `*.JPG` zuerst durch eine Liste aller Dateien mit der Endung `.JPG` ersetzt.

*Hinweis:* Gibt es keine gültige Ersetzung findet keine Ersetzung statt, das Suchmuster bleibt stehen.

```
~$ echo *.dieendunggibtesnicht
*.dieendunggibtesnicht
```

[3] In der `bash` behält auch das `!` seine Bedeutung und erlaubt History-Expansion wenn diese aktiv ist. Dies ist allerdings kein Standardverhalten und im Posix-Modus abgeschaltet.

### 2.1.1 Variablen vs. Pathname Expansion

Bevor die Shell eine Zeile ausführt, ersetzt Sie zuerst alle Variablen durch die entsprechenden Werte, und führt erst danach die Pathname Expansion durch. Die Suchmuster können also auch als Inhalt von Variablen gespeichert werden:

```
~$ a="/*"
~$ echo $a
/bin /boot /dev /etc /home /lib /lib32 /lib64 /libx32 /lost+found /media /mnt /opt
/proc /root /run /sbin /sys /tmp /usr /var
```

## 2.2 Trennung von Befehlen

Die Befehle werden üblicherweise in der Shell durch Zeilenumbrüche getrennt. Will man allerdings mehrere Befehle in einer Zeile ausführen, so können diese auch durch das Zeichen `;` getrennt werden.

## 2.3 `for`-Schleife

Wie jede Programmiersprache, kennt die Shell Schleifen. Die `for`-Schleife ähnelt dabei der in Python:

```
for VAR in LISTE; do
    echo $VAR
    Befehl2
    ...
done
```

Die durch Leerzeichen getrennten Elemente der Liste `LISTE` werden nacheinander der Variable `VAR` zugewiesen und jeweils die Befehle zwischen `do` und `done` ausgeführt.

Ein Block wird hierbei also nicht durch Klammern oder Einrückungen sondern durch die beiden neuen Schlüsselworte `do` und `done` gebildet.

## 2.4 „Parameter Expansion“

Bei der „Parameter Expansion“ `${...}` werden Variablen durch Werte ersetzt. Die simpelste Form haben Sie schon kennengelernt: `${VAR}` wird durch den Wert der Variable `VAR` ersetzt.

Es gib allerdings wesentlich mehr Ersetzungen, wichtige Beispiele wären:

```
${i/MUSTER/WORD}  # ersetzt in Wert von Variable i MUSTER durch WORD
${i/%MUSTER/WORD} # ersetzt in Werten die auf MUSTER enden, MUSTER durch WORD
${i=-WORD}        # WORD wenn Variable i leer ist
```

eine vollständige Liste ist in der `man`-Page der bash zu finden.

# 3 Shell-Programme

Anhand eines Praxis-Beispiels soll nun das Entwickeln von Shell-Programmen gezeigt werden. Das Beispiel bezieht sich hier nicht direkt auf Aufgaben aus der Physik, läßt sich aber leicht auf Datendateien übertragen. Das Grundproblem ist meistens identisch, es sind viele Dateien zu bearbeiten.

## Beispiel 1

Ziel ist es, eine große Anzahl von Bilddateien in Verzeichnisse entsprechend des Aufnahmedatums einzusortieren.

Die erste Aufgabe ist es, das Aufnahmedatum zu extrahieren. Unter Linux gibt es hier das praktische Werkzeug `exiftool` das über den Paketmanager installiert werden kann.

```
exiftool FILE
```

gibt die komplette `exif`-Informationen aus, die in eine Bilddatei `FILE` hinterlegt ist. Z.B.:

```
~$ exiftool bild.jpg
ExifTool Version Number         : 12.16
File Name                       : bild.jpg
Directory                       : /home/era
File Size                       : 1313 KiB
File Modification Date/Time     : 2024:02:10 10:54:54+01:00
File Access Date/Time           : 2024:02:10 10:54:54+01:00
File Inode Change Date/Time     : 2024:06:28 23:36:26+02:00
File Permissions                : rw-r--r--
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
X Resolution                    : 72
Y Resolution                    : 72
Resolution Unit                 : inches
Y Cb Cr Positioning             : Centered
Exif Version                    : 0220
Create Date                     : 2023:01:14 11:11:00
Components Configuration        : Y, Cb, Cr, -
Flashpix Version                : 0100
Color Space                     : Uncalibrated
Comment                         : Intel(R) JPEG Library, version [2.0.14.46]
Image Width                     : 1536
Image Height                    : 1024
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:2:0 (2 2)
Image Size                      : 1536x1024
Megapixels                      : 1.6
```

Wir wollen mit dem `Create Date` arbeiten, allerdings dies in einer leicht anderen Form. Glücklicherweise bietet `exiftool` schon eine Option um das Datumsformat vorzugeben.

```
exiftool -d "%F_%A" FILE
```

gibt das Datum in der Form `yyyy-mm-dd_dayofweek` zurück.

## 3.1 Tipp: Datumsangaben

Verwenden Sie ein Datum in einem Datei oder Verzeichnisnamen so ist es empfehlenswert dies immer in der Form `yyyy-mm-dd` zu schreiben. In dieser Form sind lexikalische Sortierung (z.B. durch `ls`) und Datumssortierung identisch.

Zusätzlich ist diese Datumsangabe im Standard ISO 8601 festgelegt und weniger anfällig für Missinterpretation insbesondere im internationalen Austausch. Siehe auch `https://xkcd.com/1179`.

Eine unbedingt empfehlenswerte Abhandlung zur Benennung von Dateien finden Sie unter `https://speakerdeck.com/jennybc/how-to-name-files`.

### weiter im Beispiel

`exiftool` gibt jetzt das Datum im gewünschen Format aus, von der gesammten Ausgabe interessiert uns aber nur die Zeile mit `Create Date`.

Die Zeile kann extrahiert werden, indem wir die Ausgabe durch den Filter `grep` laufen lassen:

```
~$ exiftool -d "%F_%A" FILE | grep "Create Date"
```

Um diese Zeile zu verstehen, allerdings erst ein kleiner Einschub:

## 3.2 Standarddatenströme (-kanäle, -streams)

Es gibt drei Datenströme, über die Programme kommunizieren können:

**STDOUT** Standard Output, Standardausgabe:
: Datenstrom, über den ein Programm seine Ausgabe sendet.

**STDIN** Standard Input, Standardeingabe:
: Datenstrom, über den ein Programm Eingaben empfängt.

**STDERR** Standard Error, Standardfehlerausgabe:
: Datenstrom, über den ein Programm Fehlermeldungen und Fehlerausgaben sendet.

Normalerweise werden `STDOUT` und `STDIN` auf der Konsole ausgegeben, und `STDIN` liest Eingaben vom Terminal (also idR. dem Keyboard).

## 3.3 Ausgabe Umleitungen:

Die Ausgabe Datenströme können umgeleitet werden.

Die Standardausgabe:

- `> Datei` Ausgabe in Datei umleiten, Datei dabei immer neu erzeugen.
- `>> Datei` Ausgabe in Datei umleiten, bei existierender Datei an diese anhängen.
- `| Programm` Ausgabe nach `STDIN` von Programm umleiten.

und die Standardfehlerausgabe:

- `2> Datei` Standardfehlerausgabe in Datei umleiten, Datei dabei immer neu erzeugen.
- `2>> Datei`: Standardfehlerausgabe in Datei umleiten, bei existierender Datei an diese anhängen.
- `2>&1` Standardfehlerausgabe in die Standardausgabe umleiten.

Bei der Standardausgabe ist `> Datei` eine Kurzform für `1> Datei`.

### weiter im Beispiel

```
~$ exiftool -d "%F_%A" FILE | grep "Create Date"
Create Date                     : 2023-01-14_Samstag
```

Mit dem „Pipe Operator“ | wird also die Ausgabe des Befehls `exiftool` zur Eingabe des Befehls `grep`.

Das Programm `grep` filtert nach einem Suchmuster

```
grep [OPTION...] PATTERNS [FILE...]
```

Wie viele Unix-Programme kann der Filter `grep` kann nicht nur auf Dateien, sondern auch auf die Standardeingabe wirken. Wird bei dem Aufruf keine Datei angegeben, erwartet `grep` Eingaben über die Standardeingabe.

Wir haben jetzt die Zeile mit der Datumsangabe extrahiert, benötigen aber nur den letzten Teil der Zeile. Hierfür benutzen wir den nächsten Filter

```
cut OPTION... [FILE]...
```

schneidet Teile von allen Zeilen aus.

```
~$ exiftool -d "%F_%A" FILE | grep "Create Date" | cut -c 35-
2023-01-14_Samstag
```

Hier sieht man gut die Unix-Philosophie: Statt großer Applikationen verwendet man viele kleine sehr spezialisierte Programme, und kombiniert diese geschickt über „Pipes“.

Jetzt wollen wir alle Bilddateien durchgehen, und verwenden eine Schleife

```
~$ for file in *.jpg; do exiftool -d "%F_%A" $file | grep "Create Date" | cut -c 35; done
```

der Code wird allerdings langsam für einen simplen Einzeiler zu lang und das Skript wird besser in eine Datei abgespeichert. In beliebigen Texteditor erzeugen wir die Datei `exif.sh` mit

```
for file in *.jpg; do
        exiftool -d "%F_%A" $file \
        | grep "Create Date" | cut -c 35-
done
```

Um das Programm übersichtlicher zu halten, wurde ein zusätzlicher Zeilenumbruch nach dem Aufruf von `exiftool` eingeführt. Dieser Zeilenumbruch würde nun eigentlich den Befehl beenden und ausführen. Um dies zu verhindern verwenden wir Quoting und der Zeilenumbruch wird mit `\` am Ende der Zeile seiner besonderen Bedeutung beraubt.

Die so erzeugte Datei können wir jetzt in der Shell ausführen.

## 3.4 Ausführen von Shellprogrammen

Um Shellskripte die als Programm in einer Datei abgelegt sind auszuführen gibt es mehrere Möglichkeiten.

Die erste Möglichkeit ist, die Datei mit `.` direkt in der Shell einzulesen

```
~$ . exif.sh
```

Bei diesem sogenannten „sourcen“ wird die Datei zeilenweise eingelesen und jede Zeile in der laufenden Shell ausgeführt. Variablen, die in der Shell zuvor definiert wurden, sind im Programm bekannt. Variablen, die im Programm definiert wurden, existieren danach in der laufenden Shell.

Meistens will man aber nicht, dass das Ausführen eines Programm die nachfolgenden Befehle beeinflussen könnte. Die zweite Möglichkeit ist, eine neue Shell aufzurufen und das Programm als Argument zu übergebenen

```
~$ sh exif.sh
```

Die neu gestartete Shell führt jetzt die Befehle aus, nach Ende des Programms beendet sich auch die neue Shell.

### 3.4.1 Shebang

Die eleganteste Möglichkeit ist allerdings in der ersten Zeile des Programms die Zeile

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

einzufügen, das Programm ausführbar zu machen und einfach wie jedes andere Programm aufzurufen.

```
~$ chmod u+x exif.sh
~$ ./exif.sh
```

Das Zeichen `#` ist eigentlich das Kommentarzeichen, allerdings gibt es eine Ausnahme:
*Beginnt* die *erste* Zeile einer Datei mit `#!` so gibt der Befehl danach den Interpreter an, mit dem diese Datei ausgeführt werden soll. Dieser Mechanismus ist auch unter dem Begriff „shebang“ oder „hashbang“ bekannt.[4]

*Bemerkung:* Dieses Konstrukt taucht nicht nur in Shellskripten auf. Bei vielen Pythonprogrammen steht in der ersten Zeile `#!/usr/bin/env python3`. Diese Programme können damit auch direkt ausgeführt werden.

## 3.5 „Command Substitution“

Bisher wird `exiftool` aufgerufen und gefiltert, die Ausgabe wird allerdings nur in der Konsole ausgegeben. Um die Ausgabe weiterzuverwenden um sie z.B. einer Variable zuzuweisen greift man auf eine weitere Ersetzung die sogenannte „Command Substitution“ zurück.

`$(BEFEHL)` wird durch die Shell durch die Ausgabe von `BEFEHL` ersetzt.

```
~$ a=$(echo hallo)
~$ echo $a
hallo
```

## 3.6 Beispiel 1 - fast fertig

Damit haben wir alles was für das Programm benötigt wird:

```
#! /bin/sh

SOURCEDIR=$PWD/Pictures
TARGETDIR=$PWD/Ziel

for file in ${SOURCEDIR}/*.jpg; do
    date=$( exiftool -d "%F_%A" $file \
        | grep "Create Date" | cut -c 35- )
    mkdir -p $TARGETDIR/$date
    cp $file $TARGETDIR/$date/
done
```

Bemerkungen:

- Mit dem „Shebang“ wird `/bin/sh` aufgerufen. Dies ist die Standardshell des Systems. In den meisten Fällen wird dies auf die `bash` zeigen. Wird die `bash` unter dem Namen `sh` aufgerufen, schaltet sie in den Posix-Modus, d.h. arbeitet voll kompatibel zum Standard.
- Zur besseren Wartbarkeit wurden Quell- und Zielverzeichnis Variablen zugewiesen.
- Da Variablen häufig Dateinamen oder Verzeichnisse enthalten, beendet `/` den Variablennamen. Bei z.B. `$PWD/Ziel` können die Klammern auch weggelassen werden.
- `mkdir` wird mit der Option `-p` aufgerufen. Damit erstellt `mkdir` auch alle Zwischenverzeichnisse.

[4] Das Verhalten von `#!` an dieser Stelle ist nicht vom Posix-Standard festgelegt. Es ist eine dort definierte mögliche Erweiterung, deren Verhalten aber undefiniert ist. Das beschriebene Verhalten ist allerdings in modernen unixartigen Betriebskernen implementiert. Etwas Hintergrundinformationen findet man in dieser historischen Unix FAQ .

### Beispiel – neue Anforderungen

Das Beispielprogramm erweist sich als nützlich, dabei tauchen aber zwei Probleme auf.

1. Steht in der Bilddatei keine Datumsinformation wird die Datei nach `$TARGETDIR` kopiert.
2. Bei anderen Verzeichnissen muss der Quelltext geändert werden. Es wäre besser, diese angeben zu können.

Steht in der Bilddatei kein Datum, wird der Variable `date` ein leerer String zugewiesen. Dies kann man Testen und darauf reagieren, braucht aber ein neues Sprachelement, Bedingungen

## 3.7 Bedingungen

Bedingungen werden in Shellskript mit `if..else` gebildet:

```
if LIST; then
    Befehle
elif LIST; then
    Befehle
else
    Befehle
fi
```

Das Schlüsselwort `elif` ist eine Kurzform für `else; if ...` und erlaubt übersichtlicheren Code bei vielfachen Verzweigungen. Wieder führt die Shell eigene Schlüsselworte für die Kennzeichnung von Blöcken ein. Für die `if` Bedingung sind dies nun: `then` und `fi`

Wie zuvor erwähnt, kennt die Shell allerdings keine boolschen Variablen, wonach wird nun entschieden ob etwas „wahr“ oder „falsch“ ist? Und was ist hier „`LIST`“?

Interpretiert die Shell Werte als boolsche Werte, so ist (nur) der Wert `0` „wahr“, alle anderen Werte werden als „falsch“ interpretiert.

`LIST` ist eine Liste von Befehlen, wobei nur der Rückgabewert des letzten Befehls dieser Liste ausgewertet wird.

## 3.8 Rückgabewerte von Befehlen

Alle Befehle und Programme die von der Shell aufgerufen werden, geben einen Wert zurück, der den Erfolg oder Misserfolg des Befehls signalisiert. Dieser Rückgabewert wird von der Shell immer für den letzten Befehl in dem speziellen Parameter `$?` gespeichert und in Bedingungen ausgewertet.

```
~$ ls
~$ echo $?
0
~$ ls foobarfoobarfoobar
ls: cannot access 'foobarfoobarfoobar': No such file or directory
~$ echo $?
2
~$ echo $?
0
```

*Wichtig:* Wie in dem Beispiel zu sehen, enthält `$?` immer den Rückgabewert des letzten Befehls. Nach `echo $?` ist dies dann der Rückgabewert des `echo` Befehls. Der Rückgabewert darf auch nicht mit der Ausgabe verwechselt werden.

Da nur der Wert `0` als „wahr“ interpretiert wird, erlaubt dies Programmen im Fehlerfall verschiedene Werte zurückzugeben. Die Programme nutzen dies häufig um für unterschiedliche Fehler unterschiedliche „Errorcodes“ zurückzugeben. Diese sind dann dokumentiert und können auch verwendet werden. Beispiel:

```
~$ man ls
LS(1)
...
Exit status:
0      if OK,

1      if minor problems (e.g., cannot access subdirectory),

2      if serious trouble (e.g., cannot access command-line argument).
```

## 3.9 Testbefehle

Die Shell bietet auch eingebaute Testbefehle, die als Rückgabewert `0` oder `1` geben und häufig in Bedingungen zum Einsatz kommen. Diese Testbefehle stehen immer in eckigen Klammern. `[...]` wobei zwischen den Klammerzeichen und Ausdrucks innerhalb der Klammern unbedingt ein Leerzeichen gefordert wird.

Einige Beispiele wären:

- `[ "$VAR" = "string" ]` Stringvergleich: Wert von Variable `$VAR` ist Zeichenkette `string`
- `[ $VAR -eq 2 ]` Arthmetischer Vergleich: Wert von Variable `$VAR` ist gleich 2
- `[ $VAR -lt 2 ]` Arthmetischer Vergleich: Wert von Variable `$VAR` ist kleiner 2
- `[ -e FILE ]` Datei FILE existiert
- `[ -r FILE ]` Datei FILE existiert und ist lesbar
- `[ -x FILE ]` Datei FILE existiert und ist ausführbare

Eine vollständige Liste finden Sie in der `man`-Page der `bash` unter `CONDITIONAL EXPRESSIONS`.

### Beispiel – erweitert

Man kann also ein leeres Datum abfangen und z.B. durch `Unknown` ersetzen.

```
#! /bin/sh

SOURCEDIR=$PWD/Pictures
TARGETDIR=$PWD/Ziel

for file in ${SOURCEDIR}/*.jpg; do
        date=$( exiftool -d "%F_%A" $file \
                | grep "Create Date" | cut -c 35- )
```

```
        if [ "$date" = "" ]; then
                date=Unknown
        fi

        mkdir -p $TARGETDIR/$date
        cp $file $TARGETDIR/$date/
done
```

## 3.10 Parameter

Das Skript soll jetzt noch mit Quell- und Zielverzeichnis aufgerufen werden. Die Aufrufparameter des Shellskripts werden von der Shell in besonderen Parametern gespeichert:

- `$1`,`$2`,... Aufrufparameter 1, Aufrufparameter 2, ...
- `$0`: Name des aufgerufenen Programms
- `$@`: Liste der Aufrufparameter
- `$#`: Anzahl der Aufrufparameter

## 3.11 Funktionen

Wie in allen Programmiersprachen, kann man auch eigene Funktionen definieren. Die definition einer Funtion `Name` erfolgt dabei durch

```
    [function] Name [()] {
        Befehle
    }
```

Die eckigen Klammern symbolisieren hierbei, dass `function` oder `()` weggelassen werden können. Üblich ist es das Schlüsselwort `function` nicht zu verwenden und die runden Klammern zu schreiben.

Bei der Funktionsdefinition werden keine Aufrufparameter angegeben. Trotzdem können Funktionen auch mit Paramtern aufgerufen werden. Diese Argumente stehen dann innerhalb der Funktion in den Paramtern `$1`,`$2`,... bzw. `$@`. Dies sind also die gleichen Parameter wie beim Skript selbst, und ihre Werte innerhalb und ausserhalb der Funktion unterscheiden sich.

## Finales Beispiel

```
#! /bin/bash

SOURCEDIR=$1
TARGETDIR=$2

usage() {
cat << EOT
Usage: $0 SOURCEDIR TARGETDIR

Kopiert Dateien aus Sourcedir
in Datumsunterverzeichnisse von TARGETDIR
EOT
}

if [ "$1" = "-h" -o "$1" = "--help" ]; then
        usage
        exit 0
elif [ $# -ne 2 ]; then
        echo "Error: Script needs two parameters"
        usage
        exit 1
fi

if [ ! -e $SOURCEDIR ]; then
        echo "Quellverzeichnis $SOURCEDIR existiert nicht"
        exit 1
fi

for file in ${SOURCEDIR}/*.jpg; do
        date=$( exiftool -d "%F_%A" $file | grep "Create Date" | cut -c 35- )
        if [ "$date" = "" ]; then
                date=Unknown
        fi
        mkdir -p $TARGETDIR/$date
        cp $file $TARGETDIR/$date/
done
```