Shell – Programmierung

Vollständig Programmiersprache

Kommentare:

```
#       Alles dahinter bis zum Zeilenende wird ignoriert
```

Ausgabe:

```
echo [-neE] [arg ...]

echo a b c
```

für zeilenweise Ausgabe, built-in und eigenes Programm
`echo -e` interpretiert Backslash Escapes wie \n etc.

Formatierte Ausgabe

```
printf FORMAT [ARGUMENT]...

printf "Name %s, Alter %d\n" "John" "30"
(Seltener benutzt, aber standardisiert)
```

Variablen:

```
Definition:    VAR=varia
```

```
Achtung: Kein Leerzeichen zwischen Vabiablennamen und "="!
Keine Variablendeklaration oder Typisierung. Die Shell selbst kann
aber nur mit Strings und Integer umgehen.
Undefinierte Variablen sind leerer String ""
→ Keine Fehlermeldung bei Tippfehler
```

Zugriff: $NAME o. ${NAME}

```
echo $VAR
echo ${VAR}ble
```

```
echo gibt Zeile aus, aber vorher ersetzt die Shell die
Variablen →  Quoting
```

Unterschied "" und ''

Immer wenn $ auftaucht gibt es eine Ersetzung (Expansion) durch etwas anderes. Später mehr...

Erstes Beispiel:

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

Hier wurden schon einige Konzepte genutzt:

Pathname Expansion:

*.JPG wird durch alle Dateien mit Endung .JPG ersetzt

Trennung von Befehlen:

`

for Schleife:

```
for VAR in LIST; do
       Befehl1 $VAR
       Befehl2
        ...
done
```

Block durch do .. done, nicht Einrückung oder Klammern

while und until-Schleifen wie in anderen Programmiersprachen gibt es natürlich auch.

Parameter-Expansion:

Simpelste Form: ${}
viele mehr (Fortgeschritten -> man-page)
z.B.
${i/MUSTER1/MUSTER2} ersetzt in Variable i MUSTER1 durch MUSTER2
${i/%MUSTER1/MUSTER2} ersetzt in Werten die auf MUSTER1 **enden** MUSTER1 durch MUSTER2
${parameter**-**word} Default parameter word

uvm.

Shell-Programme

an Hand von zwei Beispielen. Das genaue Problem ist sehr verschieden, das Grundproblem aber i.d.R. identisch, es sind viele Dateien zu bearbeiten. Die Beispiele sind daher bewusst nicht spezielle physikalische Probleme sondern aus dem Alltag, die vorgestellten Techniken aber leicht auf das aktuelle Problem übertragbar.

Beispiel1:

Bilder sollen in Verzeichnisse entsprechend des Datums einsortiert werden

Datum: exiftool gibt Informationen die im Bild hinterlegt sind aus, u.a. das Datum (Digitalkamera)

```
exiftool $file
exiftool -d "%F_%A" $file
```

Tipp: Best Practice: Datumformat: Jahr, Monat, Tag sorgt dafür das schon alles automatisch chronologisch richtig sortiert ist. Die meisten Applikationen sortieren lexikalisch von links nach rechts.
(ISO 8601, http://xkcd.com/1170, https://speakerdeck.com/jennybc/how-to-name-files)

Zeile extrahieren:

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

Piping
Der Befehl grep filtert nach Suchmuster
grep [OPTION...] PATTERNS [FILE...]

Ausgabe Umleitungen:

Die Ausgabe von Befehlen kann umgeleitet werden.

>	DATEI	DATEI neu erzeugen (d.h. auch überschreiben) und Ausgabe umleiten
>>	DATEI	Ausgabe an DATEI anhängen, nur falls nötig neu erzeugen
|	„Pipe-Operator“	Ausgabe als Eingabe für nächsten Befehl → FILTER

Einschub:

Standarddatenströme (-kanäle, -streams)

Bei Programmen gibt drei Datenströme, über die Programme kommunizieren:

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 Fehlermeldungen und Fehlerausgaben von einem Programm gesendet werden.

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

Das | -Symbol verbindet jetzt den Ausgabestrom des ersten Programms mit dem Eingabestrom des zweiten Programms

```
ls foo
ls foo | grep ...
ls foo > file.out
ls foo bar 2> file.err > file.out
ls foo bar 2>&1 | grep
```

Die meisten Unix Befehle können auf Dateien oder STDIN wirken. (Ohne File oder mit "-" als File)

Unix-Philosophie, viele kleine spezialisierte Tools, die über Pipes geschickt kombiniert werden.

(Ähnlich Funktionen die wieder Funktionen aufrufen.)

Weiter im Programm

```
exiftool -d "%F_%A" $file | grep "Create Date" | cut -c 35-
```

Alle Dateien durchgehen

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

Ein "\" am Ende einer Zeile unterdrückt Interpretation des Zeilenumbruchs durch die Shell. Befehl geht in der nächsten Zeile weiter.

Wird länger → besser als Skript speichern, kann dann auch wiederverwendet werden.

Aufruf mit . FILE
Datei wird Zeilenweise gelesen und ausgeführt.
(Source)

oder

chmod o+x und FILE aufrufen

erstes führt die Befehle in der gleichen Shell aus, letztes startet eine eigene Subshell. → Unterschiede bei Variablen.

Ausführbares Programm, besser:

```
#! /bin/sh
for file in *.jpg; do
        exiftool -d "%F_%A" $file \
                | grep "Create Date" | cut -c 35- )
        echo $date
done
```

```
#! Interpreter
```

Interpreter der das Programm ausführen soll.

Wollen die Ausgabe weiterverwenden, dazu muss es in einer Variable stehen.

Command Substitution

$( ): Programm ausführen stdout des Programms einfügen.

```
        a=$( echo "hallo"); echo $a
        b=$( ls foo); echo $b
```

Alternative (alt) ` `, besser nicht benutzen aber noch in Skripts zu finden.

```
#! /bin/sh
for file in *.jpg; do
        date=$( exiftool -d "%F_%A" $file \
                | grep "Create Date" | cut -c 35- )
        echo $date
done
```

Quell- und Zielverzeichnis in Variable, lässt sich dann leichter an einer Stelle ändern:

```
#! /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
```

Dabei benutzt:

1. Umgebungsvariable $PWD ist das aktuelle Verzeichniss
2. \ am Ende der Zeile: Befehl wird in der nächsten Zeile fortgesetzt. Ermöglicht übersichtliche Formatierung auch bei langen Befehlen.
3. | „Pipe“: Ausgabe des Befehls wird als Eingabe für den nächsten Befehl benutzt
4. mkdir -p fehlende übergeordnete Verzeichnisse, keine Fehlermeldung wenn das Verzeichniss schon existiert.

Das geht aber schief wenn keine Info zum Datum existiert!

Bedingung

```
if Testbefehl; then
        Befehle
elif test; then
        Befehle
else
        Befehle
fi
```

Testergebnis= 0 → true
sonst → false

[ ] eingebauter Testbefehl gibt 0 oder 1 zurück, je nach dem ob wahr oder falsch. Hier zum Stringvergleich benutzt.
[-e File] File existiert
[-r File] " existiert und ist lesbar
[-x File] " existiert und ist ausführbar
Aber auch numerische Vergleiche
[1 -lt 2] Kleiner uvm. siehe "man test"

(test ... ist das äquivalent als Befehl, Ergebnis in $?)

```
#! /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
```

Parameter

Das Skript kann man noch öfter brauchen, aber man will doch nicht jedesmal die Variablen ändern:

Aufruf mit Parametern:

```
SOURCEDIR=$1
TARGETDIR=$2

Übergabeparameter (Prog. oder Funktionsaufruf): $1, $2, ….
Alle Parameter in $@, Anzahl in $#, Programmname in $0,


Wenn falsch aufgerufen gibt es Probleme, daher

if [ $# -lt 2]; then
        help()
        exit 1
fi

help() {
        echo "$0 SOURCEDIR TARGETDIR"
}
```

Funktionen

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

Anzahl der übergebenen Argumente in $# , Argumente in $1 ….
Wie beim Programm. Nur diese Variablen sind lokal, alle anderen Variablen gelten global (auch $0)!

Ausführlicher Hilfe

Brauchbares Skript, aber wenig später weiß man nicht mehr was es tut:

```
usage() {
        cat << EOT
Usage: $0 SOURCEDIR TARGETDIR
Kopiert jpg-Dateien aus SOURCEDIR in entsprechende
Datums-Unterverzeichnisse  von TARGETDIR
EOT
}
```

Falls mehr Parameter

```
case "$1" in
        --help) usage(); exit 0 ;;
esac
```

Tipp:
Um Optionen zu benutzen getopts(builtin) oder getopt (extern) verwenden.

Beispiel2:

Dateien sollen umkopiert und dabei in umbenannt werden. Im Namen sollen die Dateien durchnummeriert werden.

```
#! /bin/sh
SOURCEDIR=$1
TARGETDIR=$2
SUFFIX=jpg
TMPFILE=${SOURCEDIR}/file.lst

if [ -e $TMPFILE]; then
        echo "Tempfile $TMPFILE exists..."
        echo "exiting..."
        exit 1
fi

echo "Filename?"
read NAME


ls $SOURCEDIR/*.$SUFFIX > $TMPFILE

COUNTER=0
while read FILE; do
        COUNTER=$(( COUNTER + 1 ))
        TARGET=$TARGETDIR/${NAME}-${COUNTER}.$SUFFIX
        cp $FILE $TARGET
done < $TMPFILE
rm $TMPFILE
```

Neu:

read Variable Eingabe
$(( )) Arithmetic Evaluation (integer) (auch expr als Befehl)
Schleife, while Bedingung; do Befehle; done
<,> Ein-/Ausgabe umleiten

Hint temporäre Dateien immer mit **mktemp** erstellen

sed - Stream-Editor

zeilenbasiert
Modifiziert (ohne explizite Option `-i`) nicht die Datei sondern gibt auf stdout aus Ausgabe muss dann bei Bedarf in eine **neue** Datei umgeleitet werden.
Achtung idR. muss gequotet werden damit die Shell nichts interpretiert -> '... '
-n unterdrückt die automatische Ausgabe

Bsp:

- `sed 5p    FILE`
- `sed -n 5p FILE`
- `sed -n 5,10p     FILE`
- `sed    '/regex/d' FILE`
- `sed -n '/regex/p' FILE`
- `sed -n '/BEGIN/,/END/p' FILE`

Suchen `/regex/`

`Wichtigster Einsatz:` Suchen und Ersetzen

- einmal pro Zeile `s/regex/replacement/`
- mehrfach `s/regex/replacement/g`

Stream: Hauptnutzung zum Editieren von vielen oder grossen Dateien
sowie dynamisches Editieren der Ausgabe anderer Programme

awk

(Autoren: **A**ho, **W**einberger, und **K**ernighan)

Problem:
Dateien iv...dat Strom-Spannungskennlinien von Berechnungen der des Transports durch DNA verschiedener Länge (Basenpaare). Jede Datei ist das Ergebnis der Berechnung für eine Länge. Jetzt soll aber der Strom bei fester Spannung als Funktion der Länge, genauer
`log( |I(V=-2.)| ) vs. -log(Länge)`
geplottet werden:

Lösung: **awk** + **xmgrace**

```
awk '/no_sites/ {x=$2} ($1==-2.0) {print -log(x) ''\t''
log( sqrt($2*$2) )}' * | sort -n | xmgrace -
```

Hint:

1. Alte Programme wiederverwenden
2. Optionen der Tools ansehen
3. Debuggen -xv
4. Fehlerquelle Nr. 1: Quoten