Vorlesung Unix-Praktikum

10. while und read

Dirk Frettlöh

Technische Fakultät
Universität Bielefeld

18. Dezember 2019

Willkommen zur zehnten Vorlesung
Was gab es beim letzten Mal?

- for-Schleifen
- seq, basename
- CSV-Tabellen
- cut, tr, sed

Willkommen zur zehnten Vorlesung
Was machen wir heute?

printf

Zeilenweises Arbeiten
- while
- read

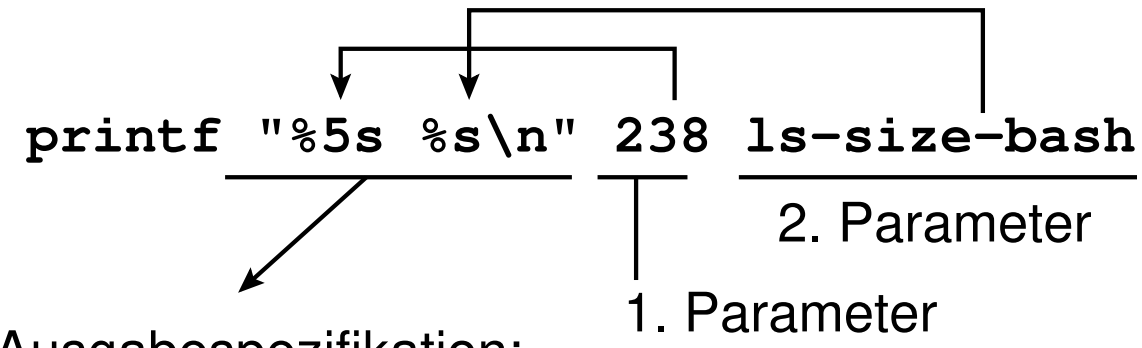
Arrays

read interaktiv

Funktionen
- Wertetabellen
- . . . mit bc
- Funktionen

printf - formatierte Ausgabe

Ansatz:



Ausgabespezifikation:

- Anzahl der Parameter
- Ausgabe der Parameter

`%5s`	String-Parameter, Spaltenbreite 5, rechtsbündig
`%-5s`	String-Parameter, Spaltenbreite 5, linksbündig
`\n`	Zeilenende (-vorschub)

Mehr Elemente von Shellskripten

formatierte Ausgabe: Dateigröße und -name


```
#!/bin/bash

for i in $(ls); do
  if test -f $i; then  # Nur Dateien berücksichtigen
    zeile=$(ls -l $i | tr -s " ")
    groesse=$(echo $zeile | cut -d\  -f 5)
    name=$(echo $zeile | cut -d\  -f 9)
    printf "%5s %s\n" $groesse $name
  fi
done
```

Spaltenweise Ausgabe
Anwendung auf die Wertetabelle


```
#!/bin/bash

printf "%5s %5s\n" x x*x

for i in $(seq 10); do
  printf "%5s %5s\n" $i $((i*i))
done
```

```
    x   x*x
    1     1
    2     4
    3     9
    4    16
  ...   ...
    9    81
   10   100
```

Zeilenweises Arbeiten
Motivation

Gegeben sei eine zweispaltige Datei `zahlen.txt`:

```
90  17
110 201
6   57
20  15
101 99
```

Wie addiert man die Datei zeilenweise?

```
90 + 17 =
110 + 201 =
usw.
```

Zeilenweises Arbeiten
for hilft nicht weiter


```
for i in $(cat zahlen.txt); do
  echo $i
done
```

```
90
17
110
201
...
```

- for-Schleifen arbeiten elementweise
- $\Rightarrow$ hilft uns nicht weiter

Zeilenweises Arbeiten
Umwandlung in CSV-Tabelle wäre möglich


```
$ tr -s " " <zahlen.txt | tr " " ";"
90;17
110;201
6;57
20;15
101;99
```

- `for`-Schleife zerlegt Zeilen nicht mehr elementweise
- Weiterverarbeitung mit `cut` wie bei CSV-Tabellen gezeigt
- aber kein allgemeingültiger Weg
  (z.B. wenn Elemente Leerzeichen enthalten dürfen)

Zeilenweises Arbeiten
while-Schleifen


```
while steuerbefehl; do
  Befehl_1
  Befehl_2
  ...
  Befehl_n
done
```

Solange steuerbefehl wahr ist,
führe $\text{Befehl}_1, \ldots, \text{Befehl}_n$ aus.

Zeilenweises Arbeiten
while-Schleifen: Beispiel


```
#!/bin/bash

zaehler=1
while test $zaehler -le 3; do
  echo $zaehler
  zaehler=$((zaehler+1))
done

$ ./while1.sh
1
2
3
```

read: Eine Zeile aus der Eingabe lesen

```
read line
```

- liest eine Zeile in die Variable `line`
- nimmt den Wert "falsch" an wenn die Eingabe leer ist

Zeilenweises Arbeiten

read ist der perfekte Zuspieler zu while


```
#!/bin/bash

while read line; do
  echo "Zeile: $line"
done
```

```
$ ./while2.sh < zahlen.txt
Zeile: 90  17
Zeile: 110 201
...
```

- zahlen.txt wird als Eingabe in die Schleife umgeleitet
- read stellt sie zeilenweise in der Var. line zur Verfügung

Zeilenweises Arbeiten
Lösung für das Addieren der Zahlen


```
#!/bin/bash

while read line; do
  line=$(echo $line | tr -s " ")
  a=$(echo $line | cut -d\  -f 1)
  b=$(echo $line | cut -d\  -f 2)
  echo $a + $b = $((a+b))
done < zahlen.txt
```

Arrays
Motivation: unterschiedlich lange Zeilen

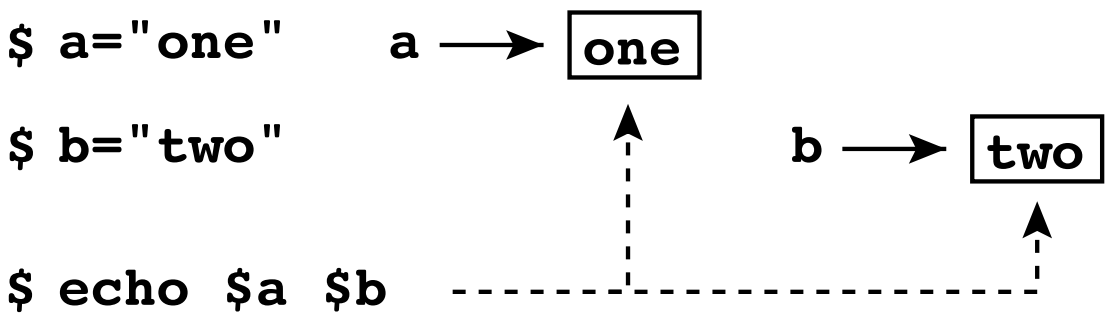
Verschärfte Bedingungen: `zahlen2.txt`

```
1 4 3
8 10 9 7
2 8 1
10 9 12 7 1
1 9
```

- unterschiedlich viele Elemente in den Zeilen!

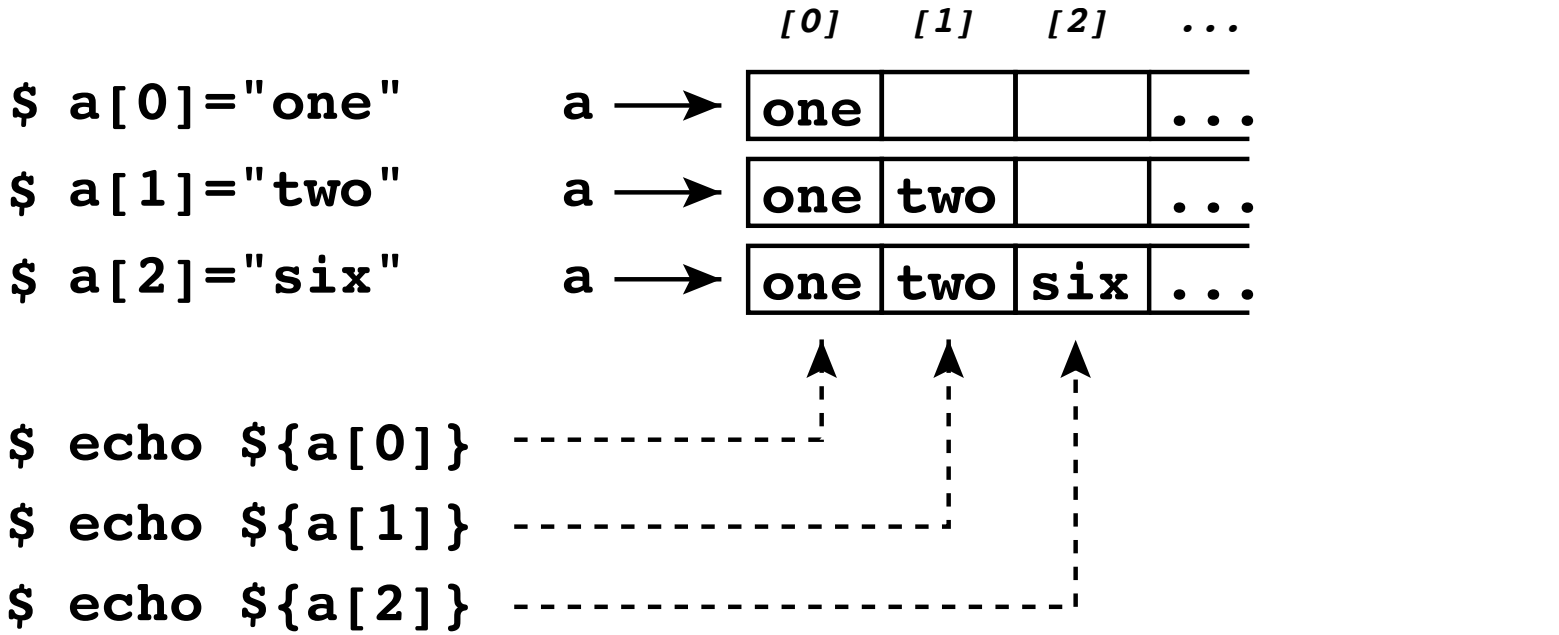
Standardfall:

- jeder Variablen ist ein Speicherbereich zugeordnet
- n Werte $\rightarrow$ n Variablennamen und Speicherbereiche

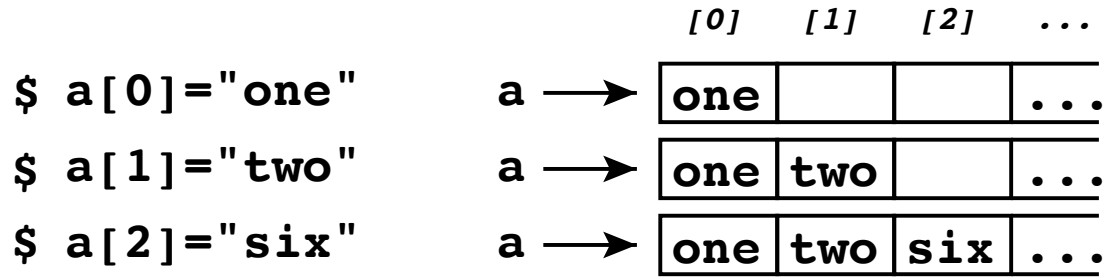


Array-Variablen (Feldvariablen):

- jeder Variablen sind *n* Speicherbereiche zugeordnet
- ein Name für *n* Speicherbereiche
- Zugriff/Unterscheidung der Speicherbereiche durch Index



Arrays
Übersicht: Länge feststellen, Elemente ausgeben



Ein Element ausgeben:

```
$ echo ${a[1]}
two
```

Alle Elemente ausgeben:

```
$ echo ${a[*]}
one two six
```

Anzahl aller Elemente ermitteln:

```
$ echo ${#a[*]}
3
```

Arrays

Arrays: Mehrere Elemente gleichzeitig setzen


```
$ b=(eins zwei drei vier fuenf)
$ echo ${b[*]}
eins zwei drei vier fuenf
```

- Zuweisungsmechanismus `b=(...)` hilft sehr beim Zerlegen von Zeilen!

Arrays

Vereinfachung des Additions-Skriptes durch Arrays

```
while read line; do
  line=$(echo $line | tr -s " ")
  a=$(echo $line | cut -d\  -f 1)
  b=$(echo $line | cut -d\  -f 2)
  echo $a + $b = $((a+b))
done < zahlen.txt
```

```
while read line; do
  z=($line)
  echo ${z[0]} + ${z[1]} = $((${z[0]}+${z[1]}))
done < zahlen.txt
```

Arrays

Lösung: Datei `zahlen2.txt` bearbeiten


```
while read line; do
  a=($line)
  letztes=$((${#a[*]}-1))

  sum=${a[0]}
  echo -n "$sum "

  for i in $(seq 1 $letztes); do
    sum=$((sum+${a[$i]}))
    echo -n "+ ${a[$i]} "
  done

  echo "= $sum"
done < zahlen2.txt
```

Array aus Zeile zusammenbauen

Index des letzten Elementes
= Länge des Arrays – 1

Ersten Summanden a[0] bearbeiten

Summanden a[1] ... a[letztes]
bearbeiten

Summe ausgeben

read
Shellskripte mit interaktiven Abfragen


```
#!/bin/bash

read -p "Geben Sie einen Dateinamen an: " name
echo "Sie gaben ein: $name"
```

```
$ ./skript.sh
Geben Sie einen Dateinamen an: brief.txt
Sie gaben ein: brief.txt
```

read
Abfrage eines einzelnen Tastendrucks


```
#!/bin/bash

read -p "Drücken Sie eine beliebige Taste: " -n 1 key
echo -e "\nDie Taste war: $key"
```

```
$ ./skript.sh
Drücken Sie eine beliebige Taste: h
Die Taste war: h
```

read
Interaktiv laufende Programme


```
#!/bin/bash

taste="x"
while test "$taste" != "e"; do
  read -p "Würfeln oder Ende (w/e)? " -n 1 taste
  echo
  if test "$taste" == "w"; then
    echo $((1+RANDOM%6))
  fi
done

echo "Spiel beendet."
```

read

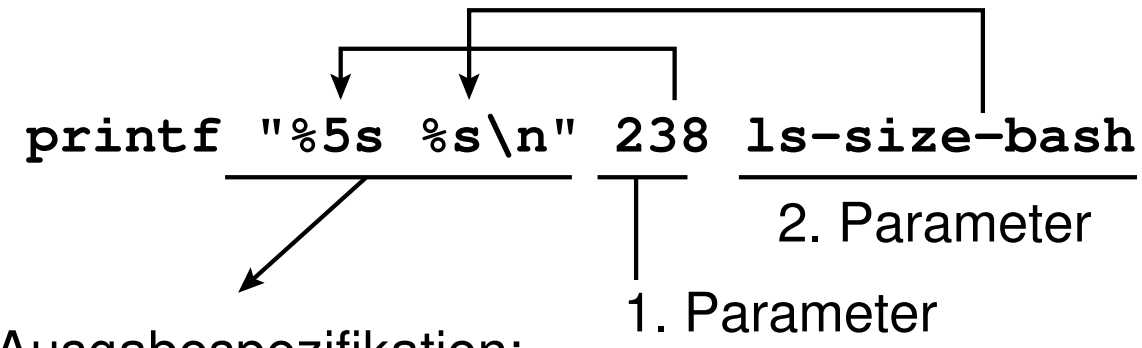
Interaktiv laufende Programme - Beispiellauf


```
$ ./skript.sh
Würfeln oder Ende (w/e)? w
2
Würfeln oder Ende (w/e)? w
6
Würfeln oder Ende (w/e)? w
1
Würfeln oder Ende (w/e)? e
Spiel beendet.
```

Recall: printf
formatierte Ausgabe

printf - formatierte Ausgabe

Ansatz:



Ausgabespezifikation:

- Anzahl der Parameter
- Ausgabe der Parameter

%5s	String-Parameter, Spaltenbreite 5, rechtsbündig
%-5s	String-Parameter, Spaltenbreite 5, linksbündig
\n	Zeilenende (-vorschub)

Wertetabellen

Wertetabellen mit Ganzzahlen können wir schon


```
#!/bin/bash

printf "%5s %5s\n" x x*x

for i in $(seq 10); do
  printf "%5s %5s\n" $i $((i*i))
done
```

```
    x   x*x
    1     1
    2     4
    3     9
    4    16
  ...   ...
    9    81
   10   100
```

Rechnen mit beliebigen Zahlen
Über Ganzzahlen hinaus: bc

Die Kommandozeile rechnet nur ganzzahlig:

```
$ echo $((7/3))
2
```

bc - der Kommandozeilenrechner hilft aus!

```
$ bc
scale=3       # 3 Nachkommastellen zeigen
7/3
2.333
```

Rechnen mit beliebigen Zahlen
Über Ganzzahlen hinaus: bc

bc - der Kommandozeilenrechner hilft aus!

```
$ bc
scale=3; 7/3      # ; ersetzt Zeilenumbruch
2.333
```

Übernahme der Werte in eine Variable:

```
$ ergebnis=$(echo "scale=3; 7/3" | bc)
$ echo $ergebnis
2.333
```

Rechnen mit beliebigen Zahlen

Aufgabe: Wertetabelle erstellen

Erstelle eine Wertetabelle für $\frac{1}{x}$

```
  x     1/x
  1 1.00000
  2 0.50000
  3 0.33333
  4 0.25000
usw.
```

Rechnen mit beliebigen Zahlen
Einfacher Ansatz


```
#!/bin/bash

printf "%3s %7s\n" x 1/x

for i in $(seq 10); do
  a=$(echo "scale=5; 1/$i" | bc)
  printf "%3s %7s\n" $i $a
done
```

```
  x     1/x
  1 1.00000
  2  .50000
  3  .33333
...
```

Rechnen mit beliebigen Zahlen
Erweiterung: mehrere Funktionen


```
printf "%3s %7s %7s %7s\n" x 1/x "1/sqrt x" "ln x"

for i in $(seq 10); do
  a=$(echo "scale=5; 1/$i" | bc -l)
  b=$(echo "scale=5; 1/sqrt($i)" | bc -l)
  c=$(echo "scale=5; l($i)" | bc -l)
  printf "%3s %7s %7s %7s\n" $i $a $b $c
done
```

-l weil bc sonst den Logarithmus nicht kennt

```
  x     1/x 1/sqrt x    ln x
  1 1.00000 1.00000       0
  2  .50000  .70710  .69314
  3  .33333  .57735 1.09861
```

Rechnen mit beliebigen Zahlen

Unschön: Fast identischer Cut&Paste-Programmkode


```
for i in $(seq 10); do
  a=$(echo "scale=5; 1/$i" | bc -l)
  b=$(echo "scale=5; 1/sqrt($i)" | bc -l)
  c=$(echo "scale=5; l($i)" | bc -l)
  printf "%3s %7s %7s %7s\n" $i $a $b $c
done
```

- die blauen Teile sind komplett gleich
- erschwert die Lesbarkeit
- Verbesserungen/Änderungen am blauen Teil müssen 3x gemacht werden
- bei komplexen Programmen ist das fehleranfällig

Funktionen
Funktionen als “Unterprogramme”


```
name()
{
   Befehl 1
   Befehl 2
   ...
   Befehl n
}
```

- Statt `name()` auch `function name`
- erzeugt eine Funktion mit Namen `name`
- Aufruf von `name` führt *Befehl 1*,. . . ,*Befehl n* aus
- (vgl. Shellskript mit Namen name.sh)

Funktionen
Funktionen als “Unterprogramme”


```
#!/bin/bash

f()
{  echo "f wurde mit Wert $1 aufgerufen"
}

echo "Funktion ausprobieren:"
f eins
f zwei
```

```
$ ./func1.sh
Funktion ausprobieren:
f wurde mit Wert eins aufgerufen
f wurde mit Wert zwei aufgerufen
```

Funktionen
Funktionen mit Rückgabewerten

- Ergebnis einfach per echo ausgeben
- (oder durch andere Befehle, die etwas ausgeben)

```
#!/bin/bash

klein()
{  echo $1 | tr [:upper:] [:lower:]
}

k=$(klein $1)
echo "$1 in Kleinschrift: $k"
```

```
$ ./func2.sh HALLO
HALLO in Kleinschrift: hallo
```

Funktionen
Wertetabelle mit Berechnungs-Funktion


```
calc()
{ y=$(echo "scale=5; $1" | bc -l)

  echo $y
}

printf "%3s %7s %7s %7s\n" x 1/x "1/sqrt x" "ln x"

for i in $(seq 10); do
  a=$(calc "1/$i")
  b=$(calc "1/sqrt($i)")
  c=$(calc "l($i)")
  printf "%3s %7s %7s %7s\n" $i $a $b $c
done
```

Funktionen

Zentrale Verbesserung an der Funktion - globale Wirkung


```
calc()
{ y=$(echo "scale=5; $1" | bc -l)
  if echo $y | grep -q "^\."; then
    echo 0$y
  else
    echo $y
  fi
}
```

---

x	1/x	1/sqrt x	ln x
1	1.00000	1.00000	0
2	0.50000	0.70710	0.69314
3	0.33333	0.57735	1.09861

Funktionen
Vergleich Lesbarkeit mit/ohne Funktionen


```
calc()
{ y=$(echo "scale=5; $1" | bc -l)

  if echo $y | grep -q "^\."; then
      echo 0$y
  else
      echo $y
  fi
}

printf "%3s %7s %7s %7s\n"\
       x 1/x "1/sqrt x" "ln x"

for i in $(seq 10); do
  a=$(calc "1/$i")
  b=$(calc "1/sqrt($i)")
  c=$(calc "l($i)")
  printf "%3s %7s %7s %7s\n" $i $a $b $c
done
```

```
printf "%3s %7s %7s %7s\n"\
        x 1/x "1/sqrt x" "ln x"

for i in $(seq 10); do
  a=$(echo "scale=5; 1/$i" | bc -l)
  if echo $a | grep -q "^\."; then
    a="0$a"
  fi
  b=$(echo "scale=5; 1/sqrt($i)" | bc -l)
  if echo $b | grep -q "^\."; then
    b="0$b"
  fi
  c=$(echo "scale=5; l($i)" | bc -l)
  if echo $c | grep -q "^\."; then
    c="0$c"
  fi
  printf "%3s %7s %7s %7s\n" $i $a $b $c
done
```

- Lesbarkeit, Struktur
  (was ist gleich/verschieden?)
- Erweiterbarkeit
  (wo/wie oft muss man Änderungen vornehmen?)

Funktionen

Funktionen als Ersatz für (kurze) Skripte

```
#!/bin/bash

head -2 $1 ; tail -n +3 $1 | sort -k $2 -n
```

Umgeschrieben als Funktion:

```
hsort2()
{
    head -2 $1
    tail -n +3 $1 | sort -k $2 -n
}

$ hsort2 planeten2.txt 2
```

Funktionen
Funktionen per .bash_aliases nutzen

Vorteile von Funktionen in der `.bash_aliases`

- $PATH muss nicht verändert werden
- alle Funktionen in einer Datei zusammengefasst
- geringfügig schnellerer Aufruf

Nachteile:

- `.bash_aliases` wird mit der Zeit unübersichtlich
- insbesondere wenn Funktionen Unterfunktionen nutzen
- Syntaxfehler in Funktion bricht `.bash_aliases`
- erst testen, dann in `.bash_aliases` einbauen

Funktionen
Fehlerbehandlung in Funktionen


```
function hsort2()
{
    if test $# != 2; then
      echo "Aufruf: hsort <datei> <spalte>"
      return 1
    fi

    head –2 $1
    tail –n +3 $1 | sort –k $2 –n
}
```

- Vorzeitiges Beenden mit `return` statt ~~`exit`~~
- `exit` würde aufrufende Sitzung beenden

Übersicht

- `printf` formatierte Ausgabe
- `while` Schleife
- `read` zeilenweises Lesen einer Datei
- `a[0],a[1],a[*]` Arrays
- `function` Funktionen (Unterprogramme)

Ende der heutigen Vorlesung

Vielen Dank fürs Zuhören!

Schöne Feiertage! Bis zum 9. Januar