Vorlesung Linux-Praktikum

2. Ausgabeumleitung und weitere Befehle

Dirk Frettlöh

Technische Fakultät
Universität Bielefeld

Zusammenfassung heute

`bc`	einfacher Taschenrechner
`>, >>, <`	Aus-/Eingabe umleiten
`sort`	Sortieren
`;`	mehrere Befehle in einer Zeile trennen
`|`	"Pipe", Ausgabe des letzten Befehls als Eingabe des nächsten Befehls nehmen
`cat`	Aneinanderhängen
`Strg-c`	Programm abbrechen
`echo`	Argument ausgeben
`head, tail`	Anfang/Ende einer Datei anzeigen
`wc`	Zählt Zeichen, Worte, Zeilen
Shellskripte	`skript.sh`
`grep`	nach Zeichenkette in Datei suchen
`sed`	Suchen und Ersetzen von Zeichenketten
`join`	Zusammenfügen von "Tabellen"
`cut`	einzelne Einträge aus Zeilen auswählen
`trim`	stutze Zeichenketten zurecht

Ein-/Ausgabe-Umleitung

Ein-/Ausgabeumleitung

bc (basic calculator)

- ein Kommandozeilen - Taschenrechner

```
$ bc
4 + 7
11
9 * 3
27
quit
```

Ein-/Ausgabeumleitung

Grundidee

Ein- und Ausgabe sind hier Text

Der Standard ist:

- `stdin`: (Standard Input) Tastatur
- `stdout`: (Standard Output) Monitor
- `stderr`: (Standard Error, Fehlermeldung) Monitor

Ein-/Ausgabeumleitung

Grundidee

Ein- und Ausgabe sind hier Text

Der Standard ist:

- `stdin`: (Standard Input) Tastatur
- `stdout`: (Standard Output) Monitor
- `stderr`: (Standard Error, Fehlermeldung) Monitor

Man kann Tastatur und Monitor durch Textdateien ersetzen

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

```
$ echo Hallo Welt > ausgabe.txt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

```
$ echo Hallo Welt > ausgabe.txt
$ more ausgabe.txt
Hallo Welt
```

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

Der Befehl `echo` gibt einfach seine Argumente an die Ausgabe.

```
$ echo Hallo Welt
Hallo Welt
```

Also:

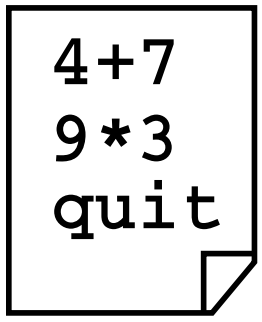
```
$ echo Hallo Welt > ausgabe.txt
$ more ausgabe.txt
Hallo Welt
```

Falls in ausgabe.txt schon etwas steht:

> löscht alten Inhalt
>> hängt an alten Inhalt an

Ein-/Ausgabeumleitung

Beispiel: Eingabeumleitung

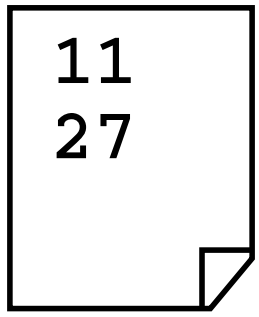



```
$ bc < eingabe.txt
11
27
```

Zeichen für Eingabeumleitung!

Ein-/Ausgabeumleitung

Beispiel: Ausgabeumleitung

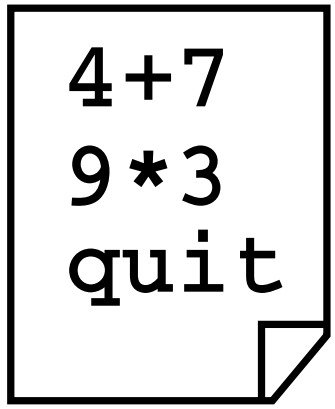


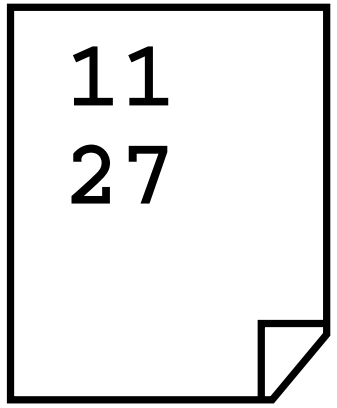

```
$ bc > ausgabe.txt
4+7
9*3
quit
```

Zeichen für Ausgabeumleitung!

Ein-/Ausgabeumleitung

Beispiel: beides gleichzeitig





```
$ bc < eingabe.txt > ausgabe.txt
```

Ein-/Ausgabeumleitung

Textdateien zeilenweise sortieren

sort

```
$ sort planeten.txt
```

- sortiert alphabetisch nach der ersten Spalte

```
$ sort -k 2 planeten.txt
```

- sortiert alphabetisch nach der zweiten Spalte

Ein-/Ausgabeumleitung

Textdateien zeilenweise sortieren

sort

```
$ sort planeten.txt
```

- sortiert alphabetisch nach der ersten Spalte

```
$ sort -k 2 planeten.txt
```

- sortiert alphabetisch nach der zweiten Spalte

```
$ sort -k 2 -n planeten.txt
```

- sortiert numerisch nach der zweiten Spalte

Ein-/Ausgabeumleitung

Verzeichnislisting nach Größe sortieren

```
$ ls -l > zwischen.txt

$ sort -k 5 -n zwischen.txt > sort.txt

$ more sort.txt

$ rm zwischen.txt sort.txt
```

- das Hantieren mit temporären Dateien ist lästig!

Ein-/Ausgabe -Weiterleitung

Ein-/Ausgabeweiterleitung

Grundidee: Verketten von Programmen

Eingabe

Programm 1

Programm 2

...

Programm n

Ausgabe

Ein-/Ausgabeweiterleitung

Anwendung auf das Sortierproblem

Das “Pipe”-Symbol | verbindet die Programme:

```
$ ls -l | sort -k 5 -n | more
```

- Ausgabe des links von | stehenden Programms
- wird Eingabe des rechts von | stehenden Programms

▶ deutlich effizienter als Zwischenspeichern!

Kommandos zum Bearbeiten von Textdateien

Textdateien zusammenfügen

cat (concatenate files)

```
$ cat eins.txt zwei.txt drei.txt
```

- gibt den Inhalt der Dateien nacheinander aus.

```
$ cat eins.txt zwei.txt drei.txt > sammlung.txt
```

- Ergebnis in neuer Datei speichern.

```
$ cat eins.txt
```

- Nützlicher Spezialfall: Eine kurze Datei anschauen

Kommandos zum Bearbeiten von Textdateien

Textdateien zusammenfügen

cat (concatenate files)

```
$ cat eins.txt zwei.txt drei.txt
```

- gibt den Inhalt der Dateien nacheinander aus.

```
$ cat eins.txt zwei.txt drei.txt > sammlung.txt
```

- Ergebnis in neuer Datei speichern.

```
$ cat eins.txt
```

- Nützlicher Spezialfall: Eine kurze Datei anschauen

`cat` ohne Argument liest von der Standardeingabe.

Nützlich & wichtig: Strg-c bricht Programm ab.

Kommandos zum Bearbeiten von Textdateien

Anfang einer Datei ausgeben

head (show head of file)

```
$ head -3 liste.txt
```

- zeigt die ersten 3 Zeilen einer Datei.

Kommandos zum Bearbeiten von Textdateien

Ende einer Datei ausgeben

tail (show tail of file)

```
$ tail -4 liste.txt
```

- zeigt die letzten 4 Zeilen einer Datei.

```
$ tail +7 liste.txt
```

- zeigt alle Zeilen ab der 7ten Zeile
  (bzw. unterdrückt die Zeilen 1 bis 6)

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Aufgabe: Planeten-Tabelle mit Überschrift sortieren

```
$ sort planeten2.txt
```

- klappt nicht wegen der Überschrift

Ansatz: Überschrift mit tail abschneiden

```
$ tail +3 planeten2.txt | sort
```

- besser, aber Überschrift fehlt jetzt

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Überschrift erhält man mit head:

```
$ head -2 planeten2.txt
```

Alles zusammenfügen:

```
$ head -2 planeten2.txt > teil1.txt
$ tail -n +3 planeten2.txt | sort > teil2.txt
$ cat teil1.txt teil2.txt > sortiert.txt
$ rm teil1.txt teil2.txt
```

- aber es entstehen wieder die unschönen Zwischendateien!

Kommandos zum Bearbeiten von Textdateien

Zusammenfassendes komplexes Beispiel

Es geht auch ohne Zwischendateien:

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```

Semikolon trennt Aufrufe

- man kann mehr als ein Programm pro Zeile ausführen
- Ausführung von links nach rechts
- Ausgaben werden aneinandergehängt

Kommandos zum Bearbeiten von Textdateien

Ausgabeumleitung des Ergebnisses

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
                                                   > ergebnis.txt
```

- liefert nicht das Gewünschte:
  nur die Ausgabe von `tail` wird umgeleitet

Lösung:

```
$ (head -2 planeten2.txt; tail -n +3 planeten2.txt | sort)
                                                   > ergebnis.txt
```

- gesamten Ausdruck in runden Klammern ausführen, dessen Ausgabe umgeleitet werden soll

Kommandos zum Bearbeiten von Textdateien

Ausgabeumleitung des Ergebnisses

```
$ head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
                                                   > ergebnis.txt
```

- liefert nicht das Gewünschte:
  nur die Ausgabe von `tail` wird umgeleitet

Lösung:

```
$ (head -2 planeten2.txt; tail -n +3 planeten2.txt | sort)
                                                   > ergebnis.txt
```

- gesamten Ausdruck in runden Klammern ausführen, dessen Ausgabe umgeleitet werden soll

Aber es gibt eine bessere Lösung.

Shellskripte

Shellskripte

Exkurs: verschiedene Shells

Es gibt verschiedene Kommandozeileninterpreter, also shells:

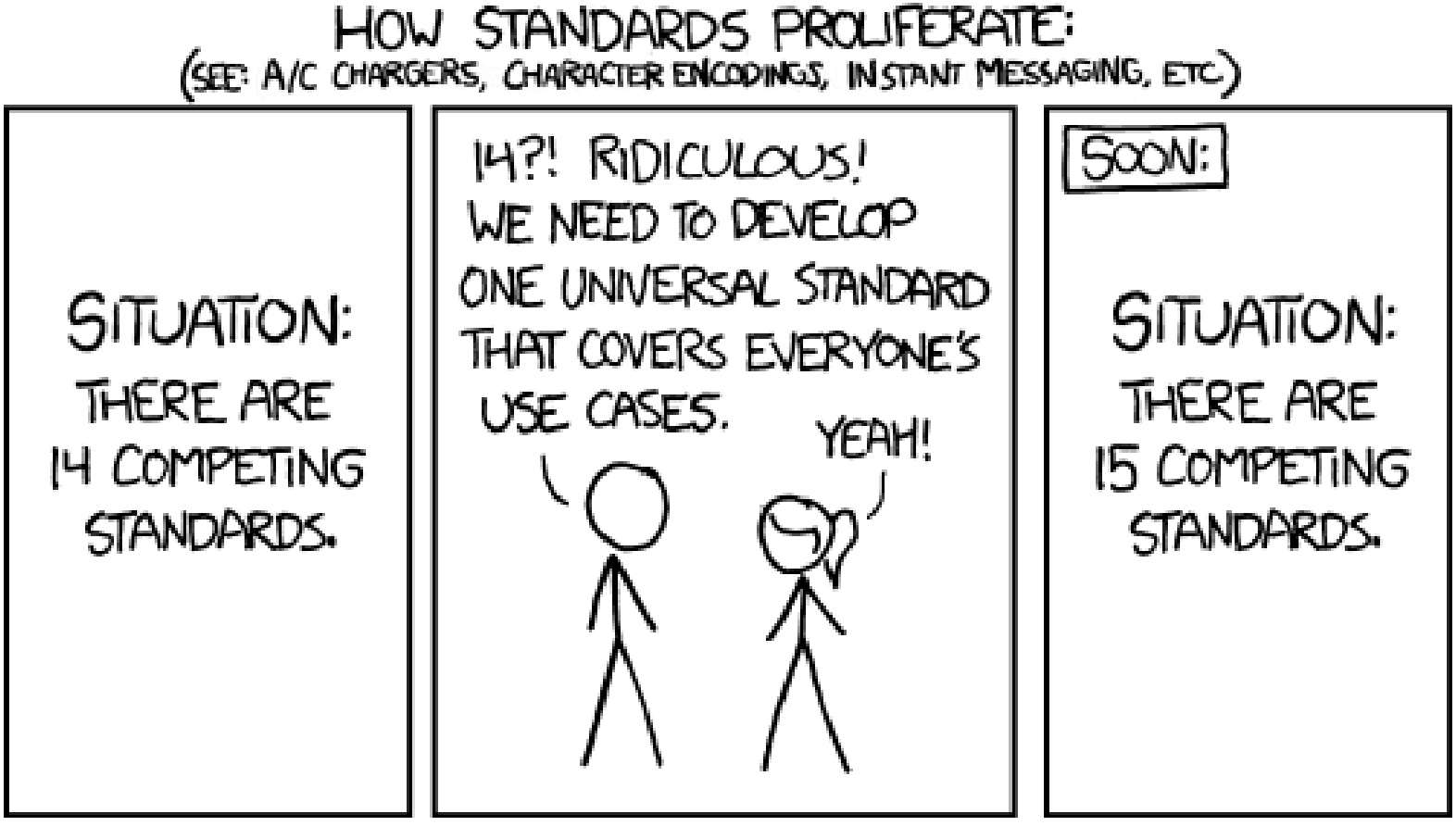
- bash: Bourne-again shell heute DIE shell, auf fast allen Linux- und MacOS-Systemen. Von GNU, seit 1989

- sh: Thompson shell (1971), Bourne shell (1975)
- ksh: Korn shell, pdksh: public domain ksh
- ash: Almquist shell, dash: Debian ash
- zsh: Z shell, csh: C shell,... moderne Erweiterungen

Anzeigen mit `$ echo $SHELL`

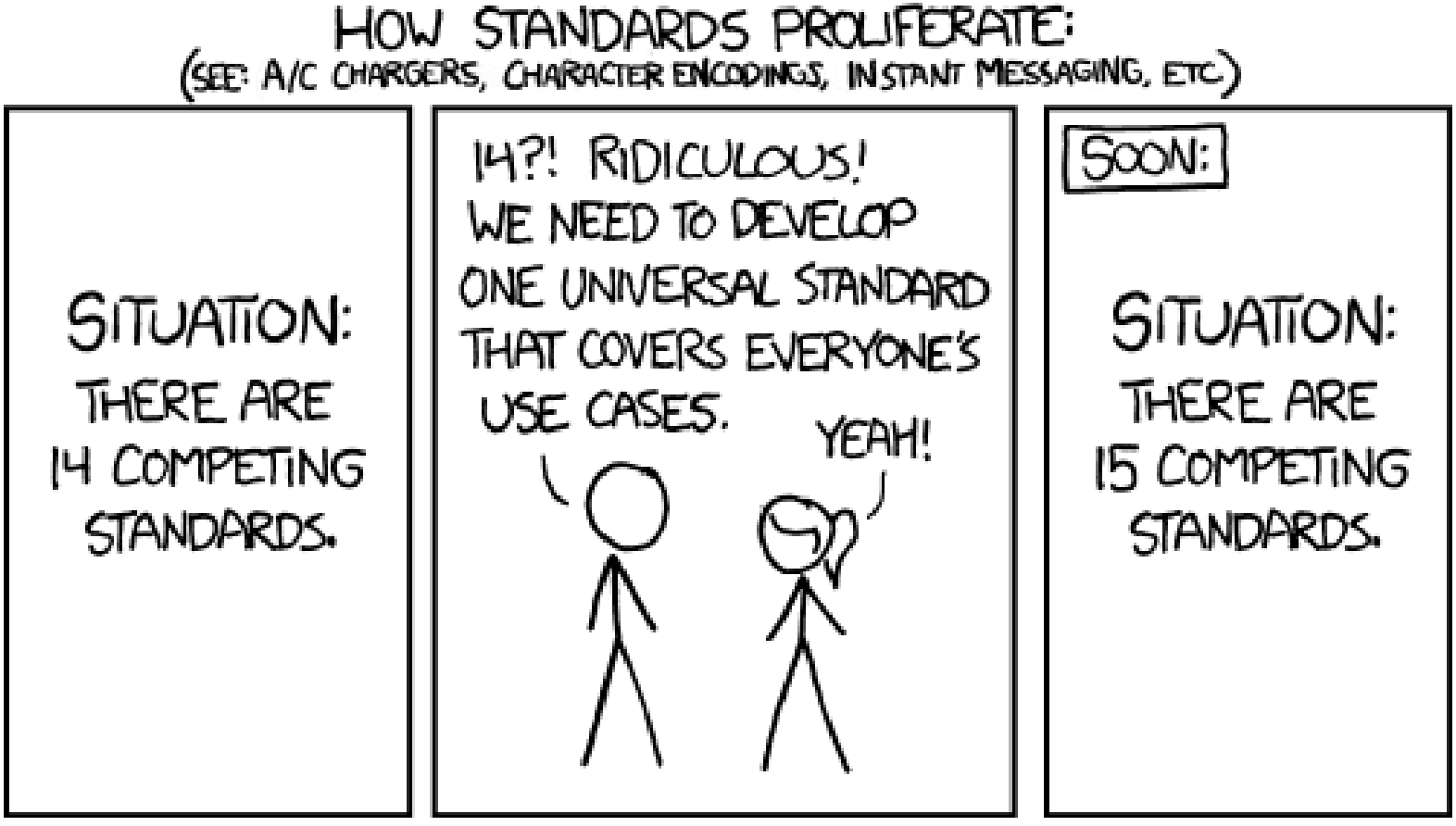
Shellskripte

Exkurs: verschiedene Shells



Shellskripte

Exkurs: verschiedene Shells



...bei shells zum Glück gerade nicht so: bash!

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

`$ echo $i` liefert den Inhalt von `i`.

Umgebungsvariablen

Später sehen wir Variablen (z.B. `i=1`)

Falls `i` eine Variable ist, liefert `echo i` einfach `i`.

`$ echo $i` liefert den Inhalt von `i`.

Es gibt in der bash ganz spezielle Variablen, die **Umgebungsvariablen**. Z.B.

```
$ echo $SHELL
bash
$ echo $USER
frettloe
$ echo $PWD
/homes/frettloe      (vgl pwd!)
```

Umgebungsvariablen

```
$ echo $LANG
de_DE.UTF-8
```

```
$ echo $PS1
\u@\h:\w$
```

`PS1` regelt die Gestalt der Eingabeaufforderung, hier:
`frettloe@hopf57:~$` (user, host, working directory)

...und viele mehr.

Umgebungsvariablen

```
$ echo $LANG
de_DE.UTF-8
$ echo $PS1
\u@\h:\w$
```

PS1 regelt die Gestalt der Eingabeaufforderung, hier:
`frettloe@hopf57:~$` (user, host, working directory)

...und viele mehr.

Alle Anzeigen mit **env**, bzw besser

```
~ $ env | more
TERM_PROGRAM=Apple_Terminal
SHELL=/bin/bash
TERM=xterm-256color
TERM_PROGRAM_VERSION=421.2
USER=frettloe
....
```

Umgebungsvariablen

Suchpfade

which Welchen Pfad hat Programm...

Programme wie `ls`, `grep`, `find`... liegen irgendwo als Dateien auf dem Rechner. Anzeige wo mittels:

```
$ which find
/usr/bin/find

$ which ls
/bin/ls
```

Umgebungsvariablen

Suchpfade

which Welchen Pfad hat Programm...

Programme wie `ls`, `grep`, `find`... liegen irgendwo als Dateien auf dem Rechner. Anzeige wo mittels:

```
$ which find
/usr/bin/find

$ which ls
/bin/ls
```

PATH sagt Linux, wo es `ls` suchen soll.

```
$ echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/homes/frettlo
```

Befehle direkt eingeben

Grenzen dessen

- nur für einzelne Befehle/Pipes praktikabel
- nur ein variabler Eingabewert/-datei pro Alias

  Gegenbeispiel:

```
head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```

- Eingabewert/-datei muss am Ende stehen
- Programmiermöglichkeiten sehr eingeschränkt
  (Nutzung von Variablen, Fallunterscheidungen, Schleifen)

Shellskripte

Wdh: Aufgaben der Kommandozeile

1. Programm ausführen
2. Programme zu mächtigeren Werkzeugen kombinieren (siehe Pipes!)
3. Kommandozeilen-Skripte
   - (1) und (2) abstrahieren und in Datei speichern
   - wiederverwenden statt erneut eintippen

Shellskripte

Aufbau

Prinzipieller Aufbau eines Shellskriptes

- Textdatei mit folgendem Inhalt:

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

Shell zum Ausführen des Skriptes

```
echo Hallo
echo ich bin ein
echo Shellskript
```

Aufrufe, wie Ihr sie auch direkt eintippen würdet

Shellskripte

Aufrufen

Ausführbarkeits-Bit setzen; aufrufen (mit `./`):

```
$ chmod u+x skript.sh
$ ./skript.sh
```

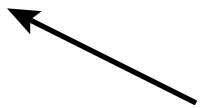
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme “weg”

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

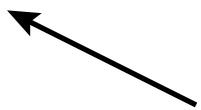
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme "weg"

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

Erste Hilfe: absolute Pfade benutzen, z.B.

```
$ /bin/ls
$ /usr/bin/emacs ~/.bash_aliases
```

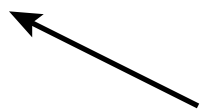
Shellskripte

Suchpfade erweitern

Vorsicht: vermurkster Suchpfad → alle Programme "weg"

(Programme sind noch da, aber die Shell findet sie nicht mehr)

```
> PATH=$PATH:~/shell-skripte
```



nicht vergessen (beliebte Falle ;–)

Erste Hilfe: absolute Pfade benutzen, z.B.

```
$ /bin/ls
$ /usr/bin/emacs ~/.bash_aliases

$ which ls
/bin/ls
```

Shellskripte

Keine Sicherheitslücken aufmachen!

Bitte **nicht** nachmachen:

~~den Punkt . in den Suchpfad aufnehmen, also~~
~~`PATH=.:$PATH` oder `PATH=$PATH:.`~~

Im Verzeichnis `/tmp` gebe es folgendes Skript:

```
#!/bin/bash
rm -rf ~/* # löscht das Benutzerverzeichnis
```

und zwar mit dem Namen "`ls`".

Würdet Ihr dort `./ls` aufrufen? Nein?
Dann nehmt auch `.` nicht in Euren Suchpfad auf!

Shellskripte

Parameterübergabe

Beispiel zur Übergabe von Parametern:

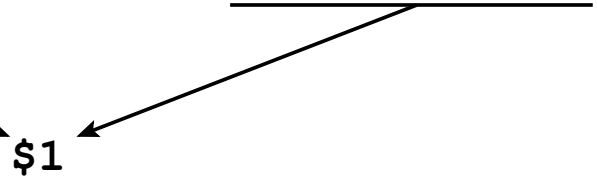
```
#!/bin/bash

echo "Erstes : $1"
echo "Zweites: $2"
echo "Drittes: $3"

echo "Anzahl : $#"
echo "Alle   : $*"
```

Shellskripte

Beispiel

Erinnerung:

```
> head -2 planeten2.txt; tail -n +3 planeten2.txt | sort
```



Abstrahieren und zusammenfassen:

```
#!/bin/bash
head -2 $1 ; tail -n +3 $1 | sort
```

```
$ ./hsort.sh planeten2.txt
```

Shellskripte

zu sortierende Spalte mit angeben

```
#!/bin/bash
head -2 $1 ; tail -n +3 $1 | sort -k $2 -n
```

```
$ ./hsort2.sh planeten2.txt 2
```

Shellskripte

Nützliche Eigenschaften von echo

`echo -n`: unterdrückt Zeilenvorschub

- Ausgabezeile mit mehreren `echo`-Befehlen erzeugen (nur sinnvoll innerhalb von Skripten)

```
#!/bin/bash

echo -n Mehrere Echo-Befehle
echo -n bauen eine
echo Zeile auf
```

```
$ ./test.sh
Mehrere Echo-Befehle bauen eine Zeile auf
```

Shellskripte

echo gibt Variablen aus

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin

$ echo "Mein login ist $USER"
Mein login ist frettloe
```

Shellskripte

echo gibt Variablen aus

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin

$ echo "Mein login ist $USER"
Mein login ist frettloe
```

Oft ist es egal, ob man mit `'...'` quotet oder mit `"..."`.

Hier ist es wichtig! Nur `"..."` zeigt den Inhalt der Variable.

Shellskripte

echo als Anfang einer Pipe

```
$ echo "Linux" | wc -m
6
$ echo -n "Linux" | wc -m
5
```

(`wc -m`: Zählt die Buchstaben)

Shellskripte

Parameter in Eingaben umwandeln

In Shellskripten:

```
#!/bin/bash

wc -m $1                # Fall A
echo $1 | wc -m         # Fall B
```

```
$ ./skript.sh hallo
```

Shellskripte

Parameter in Eingaben umwandeln

In Shellskripten:

```
#!/bin/bash

wc -m $1                # Fall A
echo $1 | wc -m         # Fall B
```

```
$ ./skript.sh hallo
```

Was ist der Unterschied zwischen Fall A und Fall B?

- Fall A: wc zählt Zeichen in der Datei “hallo”
- Fall B: wc zählt Zeichen im ersten Parameter (hier “hallo”)

Shellskripte

Ergebnisse von Programmaufrufen ausgeben

`$(...Aufruf ...)`: liefert Ausgabe des Aufrufs

Beispiel (`date` gibt Datum aus):

```
$ date "+%d. %B %Y"
11. November 2020

$ echo "Log vom $(date "+%d.  %B %Y") für $USER:"
Log vom 11.  November 2020 für frettloe:
```

Shellskripte

Ergebnisse von Programmaufrufen ausgeben

`$(...Aufruf ...)`: liefert Ausgabe des Aufrufs

Beispiel (`date` gibt Datum aus):

```
$ date "+%d. %B %Y"
11. November 2020
```

```
$ echo "Log vom $(date "+%d.  %B %Y") für $USER:"
Log vom 11.  November 2020 für frettloe:
```

Es geht beliebig komplex (mit Pipes):

```
$ echo "Die Sonne hat $(tail -n +3 planeten2.txt | wc -l)
Planeten."
Die Sonne hat 8 Planeten.
```

Kommandos zum Bearbeiten des Inhalts von Textdateien

Kommandos zum Bearbeiten von Textdateien

Texte in Dateien suchen: grep

grep (global regular expression print)

```
$ grep Mars planeten.txt
planeten.txt:  Mars 6.749 210
```

Durchsucht die Datei `planeten.txt`,
ob sie den Text "Mars" enthält.

Man kann mit Wildcards natürlich mehrere Dateien durchsuchen:

```
$ grep Mars *.txt
planeten.txt:  Mars 6.749 210
planeten2.txt:  Mars 6.749 210
```

Kommandos zum Bearbeiten von Textdateien

Texte in Dateien suchen: grep

```
$ grep mars planeten.txt
```

findet keinen Treffer: `mars` $\neq$ `Mars`

Falls Groß-/Kleinschreibung (Datei/datei) egal sein soll:

```
grep -i mars planeten.txt
planeten.txt:4 Mars 6.749 210
...
```

Kommandos zum Bearbeiten von Textdateien

Ausgaben mit grep filtern

Filtern von Programmausgaben mit grep:

```
ls -la | grep 2023
```

- zeigt alle Dateien mit Datum 2023.

Kommandos zum Bearbeiten von Textdateien

Ausgaben mit grep filtern

Filtern von Programmausgaben mit grep:

`ls -la | grep 2023`

- zeigt alle Dateien mit Datum 2023.

`ls -la | grep :`

- zeigt alle Dateien jünger als ein Jahr
  (wegen des speziellen Formats von `ls -l`, ansehen!)

Shellskripte

grep: Suchtext am Zeilenanfang/-ende verankern

"^text" $\Rightarrow$ text muss am Zeilenanfang stehen

"text$" $\Rightarrow$ text muss am Zeilenende stehen

```
$ grep Text text.txt
Der Text steht in der Mitte
Text muss am Anfang stehen
Am Ende steht der Text

$ grep "^Text" text.txt
Text muss am Anfang stehen

$ grep "Text$" text.txt
Am Ende steht der Text
```

Suchen und Ersetzen

```
Bielefled;21243;mittel;Station 44;1.Januar 2020
Herford;5741;hoch;Mast 38;1.Januar 2020
Gütersloh;28759;mittel;Mast 92;1.Januar 2020
Bielefled;12535;hoch;Mast 81;2.Januar 2020
Herford;20885;niedrig;Mast 3;2.Januar 2020
...
```

- wir brauchen “Suchen und Ersetzen” für die Kommandozeile

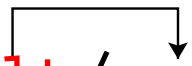
Suchen und Ersetzen

innerhalb von Textdateien

sed: script editor - "Suchen und Ersetzen" per Kommandozeile

Ersetzen des ersten Vorkommens:

```
$ echo "alt alt alt" | sed    "s/alt/neu/"
$ neu alt alt
```



Ersetzen aller Vorkommen:

```
$ echo "alt alt alt" | sed    "s/alt/neu/g"
$ neu neu neu
```

- s - Betriebsart (hier: Ausdruck suchen und ersetzen;
  es gibt noch weitere, aber s ist die häufigste)
- g - Modifier (hier: globale Ersetzung)

Suchen und Ersetzen

Fehler in der Tabelle korrigieren

```
$ sed "s/Bielefled/Bielefeld/g" < messung-typo.csv

Bielefeld;21243;mittel;Station 44;1.Januar 2020
Herford;5741;hoch;Mast 38;1.Januar 2020
Gütersloh;28759;mittel;Mast 92;1.Januar 2020
Bielefeld;12535;hoch;Mast 81;2.Januar 2020
...
```

Suchen und Ersetzen

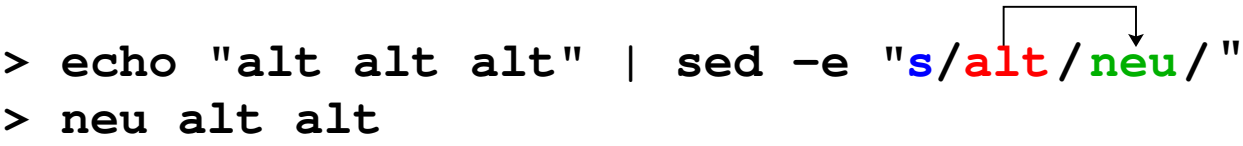
Fehler in der Tabelle korrigieren

```
$ sed "s/Bielefled/Bielefeld/g" < messung-typo.csv

Bielefeld;21243;mittel;Station 44;1.Januar 2020
Herford;5741;hoch;Mast 38;1.Januar 2020
Gütersloh;28759;mittel;Mast 92;1.Januar 2020
Bielefeld;12535;hoch;Mast 81;2.Januar 2020
...
```

Das ist schon sehr, sehr mächtig!

(Zeige: in `demo-x` das jvx $\rightarrow$ obj)

Andere Betriebsart für sed



Betriebsart y:

Buchstaben aus Liste1 durch diejenigen aus Liste2 ersetzen

Suchen und Ersetzen

Beispiel

(den folgenden Ausdruck in eine Zeile schreiben!)

```
$ echo "HALLO" |
 sed -e "y/ABCDEFGHIJKLMNOPQRSTUVWXYZ
          /abcdefghijklmnopqrstuvywyz/"
hallo
```

Arbeiten mit Tabellen

CSV-Tabellen

CSV: character separated values

Typische Darstellung von Tabellen als Textdateien:

```
Bielefeld;21243;mittel;Station 44;1.Januar 2021
Herford;5741;hoch;Mast 38;1.Januar 2021
Gütersloh;28759;mittel;Mast 92;1.Januar 2021
Bielefeld;12535;hoch;Mast 81;2.Januar 2021
```

Trennzeichen (hier: ;) beliebig wählbar
solange es nicht innerhalb der Daten vorkommt!

Arbeiten mit Tabellen

CSV-Tabellen

CSV: character separated values

Typische Darstellung von Tabellen als Textdateien:

```
Bielefeld;21243;mittel;Station 44;1.Januar 2021
Herford;5741;hoch;Mast 38;1.Januar 2021
Gütersloh;28759;mittel;Mast 92;1.Januar 2021
Bielefeld;12535;hoch;Mast 81;2.Januar 2021
```

Trennzeichen (hier: `;`) beliebig wählbar
solange es nicht innerhalb der Daten vorkommt!

Falls wir andere Trennzeichen brauchen, ist das nun einfach:

Mit `sed`.

Zusammenfügen von Tabellen mit `join`

Manchmal (z.B. für das erste Programmierprojekt) möchte man Tabellen zusammenfügen. Beispiel:

```
$ cat naehrstoffe.txt
1 Protein
2 Kohlenhydrate
3 Fett
4 Alkohol

$ cat speisen.txt
1 Hühnchen
2 Kartoffeln
3 Butter
4 Schnaps
```

Zusammenfügen von Tabellen mit `join`

Manchmal (z.B. für das erste Programmierprojekt) möchte man Tabellen zusammenfügen. Beispiel:

```
$ cat naehrstoffe.txt
1 Protein
2 Kohlenhydrate
3 Fett
4 Alkohol

$ cat speisen.txt
1 Hühnchen
2 Kartoffeln
3 Butter
4 Schnaps
```

Die erste Spalte ist identisch, nach der wird gejoint:

```
$ join naehrstoffe.txt speisen.txt
1 Protein Hühnchen
2 Kohlenhydrate Kartoffeln
3 Fett Butter
4 Alkohol Schnaps
```

Zusammenfügen von Tabellen mit `join`

`join` möchte, dass die Join-Felder (gleich) sortiert sind.
Falls nicht:

```
$ cat naehrstoffe.txt
1 Protein
2 Kohlenhydrate
3 Fett
4 Alkohol

$ cat speisen.txt
2 Kartoffeln
1 Hühnchen
3 Butter
4 Schnaps
```

...dann gibt es eine Fehlermeldung:

```
$ join naehrstoffe.txt speisen.txt
join: speisen.txt:2: is not sorted: 1 Hühnchen
[...]
join: input is not in sorted order
```

Zusammenfügen von Tabellen mit `join`

Falls in beiden Spalten die Reihenfolge 2-1-3-4 ist, klappt's auch.

Gute Praxis ist: erst `sort`, dann `join`.

Zusammenfügen von Tabellen mit `join`

Falls in beiden Spalten die Reihenfolge 2-1-3-4 ist, klappt's auch.

Gute Praxis ist: erst `sort`, dann `join`.

Natürlich kann man die Join-Felder auch explizit angeben:

```
$ cat speisen2.txt
Hühnchen 1
Kartoffeln 2
Butter 3
Schnaps 4
```

Zusammenfügen von Tabellen mit `join`

Falls in beiden Spalten die Reihenfolge 2-1-3-4 ist, klappt's auch.

Gute Praxis ist: erst `sort`, dann `join`.

Natürlich kann man die Join-Felder auch explizit angeben:

```
$ cat speisen2.txt
Hühnchen 1
Kartoffeln 2
Butter 3
Schnaps 4

$ join -1 1 -2 2 naehrstoffe.txt speisen2.txt
1 Protein Hühnchen
2 Kohlenhydrate Kartoffeln
3 Fett Butter
4 Alkohol Schnaps
```

Zusammenfügen von Tabellen mit `join`

Falls in beiden Spalten die Reihenfolge 2-1-3-4 ist, klappt's auch.

Gute Praxis ist: erst `sort`, dann `join`.

Natürlich kann man die Join-Felder auch explizit angeben:

```
$ cat speisen2.txt
Hühnchen 1
Kartoffeln 2
Butter 3
Schnaps 4

$ join -1 1 -2 2 naehrstoffe.txt speisen2.txt
1 Protein Hühnchen
2 Kohlenhydrate Kartoffeln
3 Fett Butter
4 Alkohol Schnaps
```

Wie bei vielen anderen Befehlen: der Rest ist googlen.
(stackoverflow / stackexchange ist oft hilfreich!)

Arbeiten mit Tabellen

Spalten aus CSV-Tabellen auswählen

cut: Spalten aus Tabellen auswählen

Aufruf: `cut -d` *trennzeichen* `-f` *spalten*

Trennzeichen mit Bedeutung in der Shell "entschärfen":

```
cut -d \;   ...
cut -d \␣   ...
```

Arbeiten mit Tabellen

Spalten aus CSV-Tabellen auswählen

cut: Spalten aus Tabellen auswählen

Aufruf: `cut -d` *trennzeichen* `-f` *spalten*

Trennzeichen mit Bedeutung in der Shell "entschärfen":

```
cut -d \;   ...
cut -d \␣   ...
```

typische Spaltenauswahlen:

`cut -f 2,5,9`	Spalten 2,5,9 auswählen
`cut -f 2-4,7`	Spalten 2 bis 4 und 7
`cut -f 5-`	alle Spalten ab der 5.

Arbeiten mit Tabellen

Beispiele

Spalten 1,2 und 4 auswählen:

```
$ cut -d\; -f1,2,4 messung.csv
Bielefeld;21243;Station 44
Herford;5741;Mast 38
Gütersloh;28759;Mast 92
```

Arbeiten mit Tabellen

Beispiele

Spalten 1,2 und 4 auswählen:

```
$ cut -d\; -f1,2,4 messung.csv
Bielefeld;21243;Station 44
Herford;5741;Mast 38
Gütersloh;28759;Mast 92
```

Spalten 1,2 und 5 nur für Bielefeld auswählen:

```
$ grep Bielefeld messung.csv | cut -d\; -f 1-2,5
Bielefeld;21243;1.Januar.2021
Bielefeld;12535;2.Januar.2021
Bielefeld;24817;3.Januar.2021
...
```

- Spalten vertauschen $\rightarrow$ Übungen

Tabellen mit Leerzeichen als Spaltentrennern

Ausgabe von ls spaltenweise zerlegen

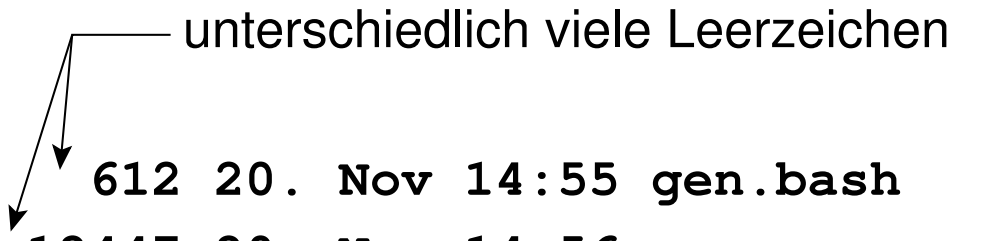
Ziel: In der Ausgabe von `ls -l` Größe und Namen von Dateien (Spalten 5,9) extrahieren.

Tabellen mit Leerzeichen als Spaltentrennern

Ausgabe von ls spaltenweise zerlegen

Ziel: In der Ausgabe von `ls -l` Größe und Namen von Dateien (Spalten 5,9) extrahieren.

Problem: `cut` betrachtet 3 Leerzeichen als 3 leere Spalten!




```
> ls -l
-rwxr--r-- 1 cg cg  612 20. Nov 14:55 gen.bash
-rw-r--r-- 1 cg cg 12447 20. Nov 14:56 messung.csv
```

```
$ ls -l | cut -d\  -f 5,9
 Nov
12447 messung.csv
```

Tabellen mit Leerzeichen als Spaltentrennern

einzelne Zeichen umwandeln oder zusammenfassen

tr: Zeichen umwandeln oder zusammenfassen

Zeichen komprimieren:

```
$ echo "abxxxbacxxxxxxb" | tr -s "x"
abxbacxb
```

Zeichen umwandeln:

```
$ echo "abxxxbaccxxxxxxb" | tr "xc" "yd"
abyyybaddyyyyyyb
```

Groß-/Kleinschreibung konvertieren:

```
$ echo GROSS | tr [:upper:]  [:lower:]
gross
```

Tabellen mit Leerzeichen als Spaltentrennern

Lösung zum Auswählen von Spalten aus ls -l

```
ls -l | tr -s " " | cut -d\ -f 5,9
```

```
612 gen.sh
238 ls-size.sh
12447 messung.csv
283 rechner.sh
4502 verbrauch.txt
4096 verzeichnis
```

Zusammenfassung heute

`bc`	einfacher Taschenrechner
`>, >>, <`	Aus-/Eingabe umleiten
`sort`	Sortieren
`;`	mehrere Befehle in einer Zeile trennen
`|`	"Pipe", Ausgabe des letzten Befehls als Eingabe des nächsten Befehls nehmen
`cat`	Aneinanderhängen
`Strg-c`	Programm abbrechen
`echo`	Argument ausgeben
`head, tail`	Anfang/Ende einer Datei anzeigen
`wc`	Zählt Zeichen, Worte, Zeilen
Shellskripte	`skript.sh`
`grep`	nach Zeichenkette in Datei suchen
`sed`	Suchen und Ersetzen von Zeichenketten
`join`	Zusammenfügen von "Tabellen"
`cut`	einzelne Einträge aus Zeilen auswählen
`trim`	stutze Zeichenketten zurecht

Ende der heutigen Vorlesung

Sie probieren das alles gleich in den Tutorien aus.

Viel Spaß! Bis morgen!