Shell Scripting — Eine Einführung

Jörg Faschingbauer

Table of Contents

FASCHINGBAUER

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Der Anfang

FASCHINGBAUER

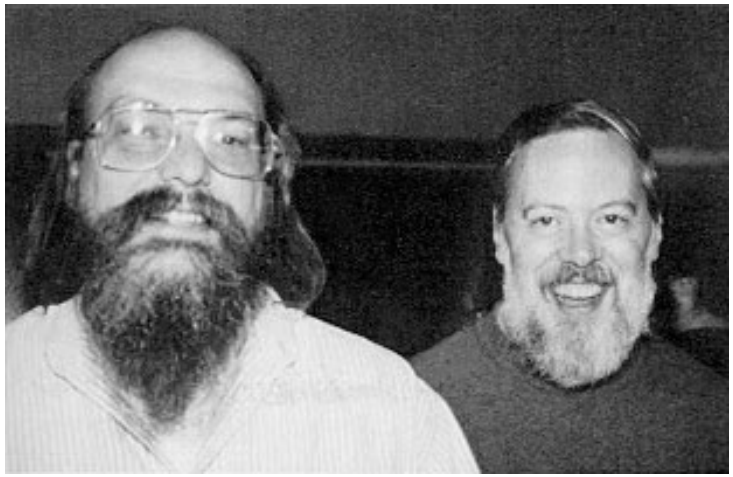
Am Anfang war nichts ...

- ... ausser PDP-7
- ... und ein paar coole Typen
- Brian Kernighan, Dennis Ritchie → C
- Ken Thompson, Dennis Ritchie → Erstes UNIX
- Ken Thompson → Erste Shell

Der Rest ist Geschichte!

Coole Typen und ihr Hobby

FASCHINGBAUER

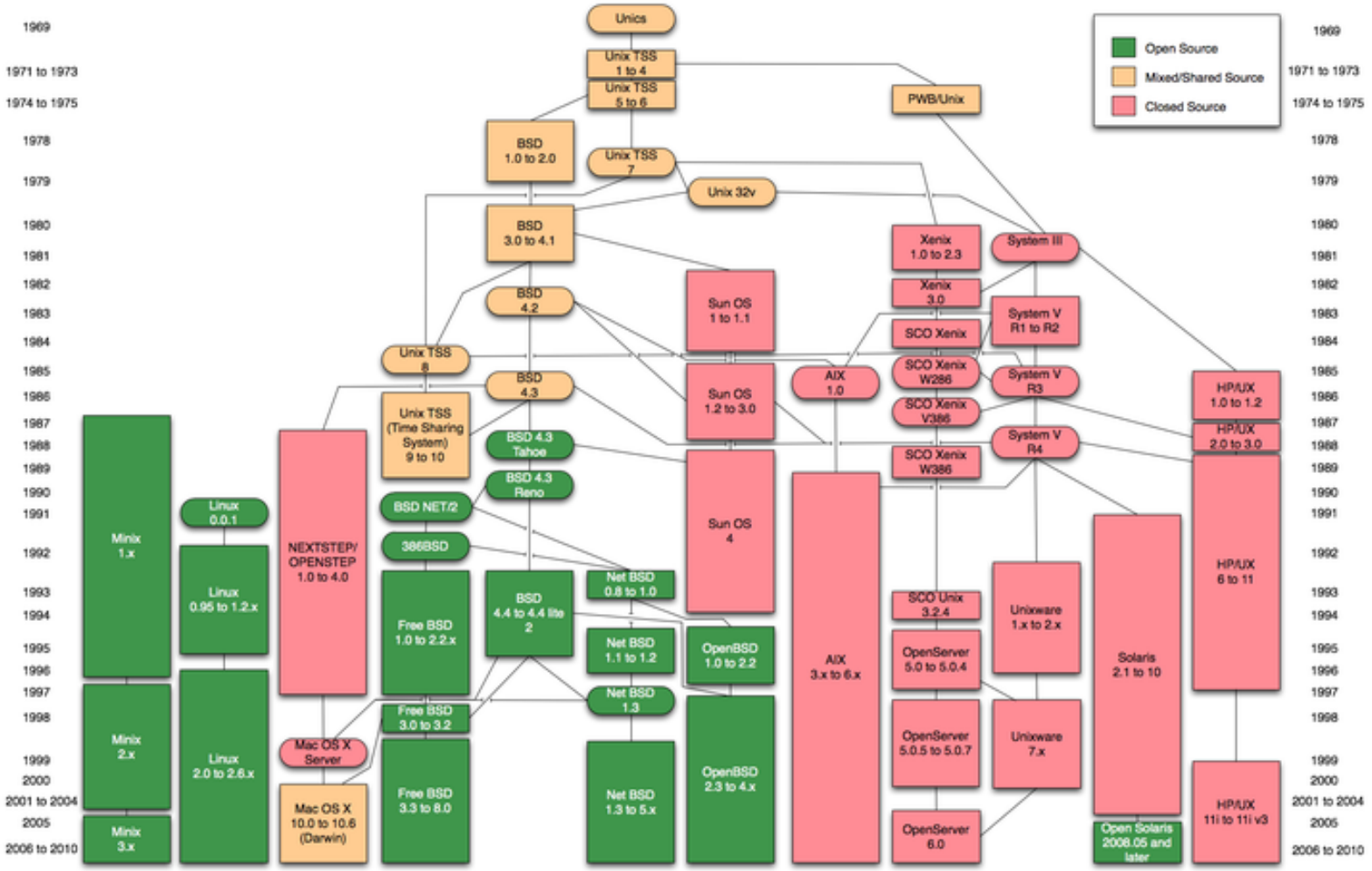


Ken Thompson, Dennis Ritchie

PDP-7

Die Geschichte von UNIX

FASCHINGBAUER



Shell

FASCHINGBAUER

- "Shell": zu deutsch "Schale"
- → damals war das Betriebssystem in der Commandline statt umgekehrt
- z.B. `DOS` (eigentlich nach UNIX erfunden)
- Revolutionär: Shell ist nur ein Programm wie alle anderen
- Erste Shell (Thompson-Shell) wurde mit UNIX zusammen ausgetragen
- Gegenseitige Beeinflussung → UNIX-Konzepte in Shell sichtbar

Bourne Shell

FASCHINGBAUER

- Thompson-Shell war sehr einfach → “featurefrei”
- Hatte IO-Redirection und Pipes (fundamental in UNIX)
- ... aber keine Programmier-Konstrukte
- → Stephen R. Bourne: Bourne Shell (ca. 1977)
- → vollständige Programmiersprache
- ... im wesentlichen Gegenstand dieses Kurses

C Shell

FASCHINGBAUER

- Bourne Shell zwar funktional, aber kein Commandline-Editing
- → Bill Joy: C Shell (`csh`, ca. 1980)
- Inkompatibel zur Bourne Shell
- Weniger gut zum Programmieren geeignet, aber besser zum Tippen
- → verdrängte die Bourne Shell in den kommenden Jahren
- Weiterentwicklung: TENEX C Shell (`tcsh`)

Bourne Again Shell (`bash`)

FASCHINGBAUER

- GNU Projekt: "Software muss frei sein"
- Bourne Shell nur mit Lizenzierung erhältlich
- Clone namens "Bourne Again Shell": `bash`
- Rückwärtskompatibel mit Bourne Shell
- Commandline Editing
- Arrays
- u.v.a.m.
- *"Die Shell" in Linux*

Weitere Shells

FASCHINGBAUER

Eine Shell ist nur ein Programm wie alle anderen → viele andere Shells mit unterschiedlichem Scope

- Korn Shell (`ksh`)
- Z-Shell (`zsh`)
- Standalone Shell (`sash`)
- Almquist Shell (`ash`)
  - weitgehend kompatibel mit Bourne (Again) Shell
  - Teil der *Busybox* (Embedded Linux)

Overview

FASCHINGBAUER

Minimales Shellscript: “Hello World” (1)

FASCHINGBAUER

Das File `hello`

```
#!/bin/sh

# Das ist ein Kommentar
echo Hello World
```

- `#!` ... “She-Bang”, “Hash-Bang”. Der *Interpreter* für das Script. (Beliebig, je nach Sprache → `/usr/bin/python`, `/usr/bin/perl`)
- ... gefolgt von Shell-Befehlen (und Kommentaren)

Minimales Shellscript: “Hello World” (2)

FASCHINGBAUER

- Shell-Scripts sind Programme → *ausführbar*
- Müssen nicht etwa auf `.sh` enden

```
$ chmod 755 hello
$ ./hello
Hello World
```

Alternative Ausführung

FASCHINGBAUER

- Script hat keinen She-Bang (aber ausführbar) → `/bin/sh` ist *Default Interpreter*
- Script nicht ausführbar → expliziter Shell-Aufruf

Explizit mit der Shell

```
$ ls -l ./hello
-rw-r--r-- ... ./hello
$ bash hello
Hello World
```

Overview

FASCHINGBAUER

Shell Scripts und Errors (1)

FASCHINGBAUER

Missverständliches Shell Script

```
#!/bin/sh
rm -f /etc/passwd
echo "/etc/passwd erfolgreich geloescht"
```

- Löscht `/etc/passwd` nicht (ausser als `root`)
- Macht einfach weiter
- *Exit-Status* ist der des letzten Commands → `echo` war ok

Shell Scripts und Errors (2)

FASCHINGBAUER

Das will man (meistens) nicht!

Missverständliches Shell Script

```
#!/bin/sh

set -e

rm -f /etc/passwd
echo "/etc/passwd erfolgreich geloescht"
```

Shell Scripts und Errors (3)

FASCHINGBAUER

`set -e` **bedeutet genau folgendes**

- Bricht das Shellscript ab, wenn ein Command fehlschlägt
- Der Exitstatus des Scripts ist der des fehlgeschlagenen Commands
- Bricht nur ab, wenn das Command *nicht* als *Bedingung* ausgeführt wird.

Bricht nicht ab

```
if rm -f /etc/passwd; then echo "ok"; fi
```

Debugging: `set -x` (1)

FASCHINGBAUER

Es gibt keinen Shell-Debugger!

- "`printf` Debugging"
- `set -x` → jedes aufgerufene Command wird *auf Standard Error* ausgegeben

`set -x` in Aktion

```
#!/bin/sh
set -x
rm -f /etc/passwd
echo "/etc/passwd erfolgreich geloescht"
```

Debugging: `set -x` (2)

FASCHINGBAUER

`set -x` in Aktion

```
+ rm -f /etc/passwd
rm: cannot remove '/etc/passwd': Permission denied
+ echo '/etc/passwd erfolgreich geloescht'
/etc/passwd erfolgreich geloescht
```

- Sehr "laut" in grossen Scripts
- $\rightarrow$ nur temporär einschalten, mit `set +x` wieder ausschalten

Status- und Debuggingmessages (1)

FASCHINGBAUER

Bitte um Disziplin! Debugoutput geht auf Standard Error!

Nummer	Macro lt. POSIX	`stdio.h` Äquivalent
0	Standard Input	`stdin`
1	Standard Output	`stdout`
2	Standard Error	`stderr`

Error Message, richtig gemacht

```
if ! rm /etc/passwd; then
    echo Jessas 1>&2
fi
```

Status- und Debuggingmessages (2)

FASCHINGBAUER

... oder so ...

```
rm /etc/passwd || echo Jessas 1>&2
```

... oder so

```
message() {
    echo $* 1>&2
}
rm -f /etc/passwd || message Jessas
```

Overview

Overview

FASCHINGBAUER

Variablen

FASCHINGBAUER

Shell ist eine Programmiersprache → hat auch Variablen

Variablen zum Einstieg

```
$ TEXT=Hallo
$ echo $TEXT
Hallo
$ echo "Sag mal $TEXT"
Sag mal Hallo
$ echo 'Sag mal $TEXT'
Sag mal $TEXT
```

Variablen: genauer ...

FASCHINGBAUER

- Links und rechts des Zuweisungsoperators '=' darf kein Space stehen
- Hat der Wert Spaces, muss er gequoted werden
- *Dereferenziert* wird mit '$': `$VARIABLENNAME`
- Variablen werden in Strings expandiert, die mit Doublequotes umschlossen sind
- In Strings mit Singlequotes wird nicht expandiert
- Variablen sind *immer* vom Typ *String*

Variablen: Wissenswertes (1)

FASCHINGBAUER

Ausgeben aller Variablen ...

set ohne Parameter

```
$ set
...
TEXT=Hallo
TMPDIR=/tmp
UID=1000
USER=jfasch
x=1
y=2
...
```

Variablen: Wissenswertes (2)

FASCHINGBAUER

Substituieren von Variablen ohne Leerzeichen ...

```
$ TEXT=mitten
$ echo dasist${TEXT}drin
dasistmittendrin
```

Variablen: Wissenswertes (3)

FASCHINGBAUER

Variablen können als *readonly* markiert werden ...

Readonly Variablen

```
$ x=1
$ readonly x
$ x=2
bash: x: readonly variable
```

Variablen: Wissenswertes (4)

FASCHINGBAUER

`declare`: Typisierung durch Formatierung

```
$ declare -i x
$ x=irgendwas
$ echo $x
0
$ x=42
$ echo $x
42
$ declare -l x
$ x=UPPERCASE
$ echo $x
uppercase
```

Overview

FASCHINGBAUER

Environment

FASCHINGBAUER

Sichtbarkeit von Shell-Variablen

- Leben nur im Memory des Shell-Prozesses
- $\rightarrow$ Fremde Programme haben keinen Zugriff darauf

Environment-Variablen

- *Vererbte Prozess-Eigenschaft*
- Jeder Prozess erbt die Environment-Variablen seines *Parent*
- *Inhärentes Konzept von UNIX*
- Die Shell zeigt Environment-Variablen wie Shell-Variablen

Environment Variablen in der Shell

FASCHINGBAUER

Durch `export` werden Shell-Variablen *ins Environment aufgenommen* und vererbt

```
$ x='ein wert'
$ export x
```

Oder einfach

```
$ export x='ein wert'
```

Populäre Environmentvariablen

FASCHINGBAUER

Windows: "Systemvariablen" (z.B. `%PATH`) → unklare Definition
UNIX: *Environmentvariablen*, die früh gesetzt werden → *Vererbung in Child-Prozesse* (z.B. in alle Programme des Desktop)

- `PATH`: Suchpfad für Programme. Default gesetzt in `/etc/profile` bei Login (i.a. unterschiedlich für User und `root`). Gerne erweitert in `~/.bashrc`.
- `EDITOR`: Default-Editor für z.B. `svn commit`
- `PAGER`: seitenweises Blättern, als Backend von z.B. `man`
- `HOME`, `USER`, `SHELL`: bei Login gesetzt (→ `/etc/passwd`)
- `TMPDIR`: temporäre Dateien
- `LD_LIBRARY_PATH`: Suchpfad für Shared Libraries

Pfade

FASCHINGBAUER

Achtung: Suchpfade (z.B. `PATH`, `LD_LIBRARY_PATH`) werden durch ':' getrennt!

```
$ PATH=/home/jfasch/scripts:$PATH
```

Achtung - Security-Risiko!

```
$ PATH=.:$PATH
```

Persistentes Setzen von Environmentvariablen

FASCHINGBAUER

Login-Shell evaluiert folgende Files

- `/etc/profile`
- `$HOME/.bash_profile`
- `$HOME/.bash_login`
- `$HOME/.profile`

Interaktive Shell evaluiert

- `$HOME/.bashrc`

($\rightarrow$ Interaktive Login-Shell evaluiert alle)

Ausgeben aller Environmentvariablen

FASCHINGBAUER

`set` gibt Shell- *und* Environmentvariablen aus → `env` nur für Environmentvariablen

```
$ env
...
MANPATH=/usr/share/man:/usr/share/postgresql-9.2/man/:...
SHELL=/bin/bash
TMPDIR=/tmp
USER=jfasch
PAGER=/usr/bin/less
HOME=/home/jfasch
...
```

Overview

FASCHINGBAUER

Positionelle Parameter

FASCHINGBAUER

```
$ ./hello eins zwei
```

→ *drei* Parameter

- `$0`: Programmname selbst, incl. evtl. `PATH` Komponenten. Hier: `./hello`
- `$1`: `eins`
- `$2`: `zwei`
- `$#`: *Index* des letzten Arguments. Hier: 2
- Mehr als Zehn: `${42}`

Spezielle Variablen

FASCHINGBAUER

Spezielle Variablen zum Commandline-Handling

- $*: alle Argumente (ohne $0) als ein String
- $@: alle Argumente (ohne $0) als ein Array von Strings
- $#: Anzahl der Argumente (ohne $0)

Tricks für kurzes Commandline Parsing

FASCHINGBAUER

Checken auf fehlende (eigentlich: *leere*) Parameter

paramcheck

```
#!/bin/sh
echo ${1:?Parameter 1 fehlt}
```

```
$ ./paramcheck
./paramcheck: line 2: 1: Parameter 1 fehlt
```

`shift`: Variabel lange Argumentlisten

FASCHINGBAUER

`shift` vernichtet nächstes Argument und schiebt die folgenden um eins runter

- $2 wird zu $1
- $3 wird zu $2
- ...

```
while [ $# -gt 0 ]; do
    echo $1
    shift
done
```

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Verzweigungen und Bedingungen

FASCHINGBAUER

Shell ist eine vollständige Programmiersprache → **Verzweigungen und Bedingungen**

- Bedingung: Exitstatus eines Prozesses (Commands)
- Verzweigungen mit `if`, `while`, ...
- `$?`: Exitstatus des letzten im Vordergrund ausgeführten Programms

Wahr und Falsch (1)

FASCHINGBAUER

Exitstatus

- Zahl von 0 bis 255
- 0: ok $\rightarrow$ *true*
- $>$0: Fehler $\rightarrow$ *false*
- Programme definieren individuelle Fehlercodes

Wahr und Falsch (2)

FASCHINGBAUER

```
$ cat /etc/passwd
...
$ echo $?
0
$ rm -f /etc/passwd
rm: cannot remove '/etc/passwd': Permission denied
$ echo $?
1
```

if (1)

FASCHINGBAUER

`if` verzweigt in Abhängigkeit des *Exitstatus der Bedingung*

```
if rm -f /etc/passwd
then
    echo Uff, gut gegangen
else
    echo Hoppla, fehlgeschlagen
fi
```

if (2)

FASCHINGBAUER

Negierung der Bedingung durch "!".

```
if ! rm -f /etc/passwd; then
    echo Wahrscheinlich schlechte Permissions
fi
```

Achtung: Space nach ! → ansonsten *History Expansion*

if (3)

FASCHINGBAUER

Allgemeine Form ist *ausschliesslich*

```
if command; then
    true-command
else
    false-command
fi
if ! command; then
    false-command
else
    true-command
fi
```

`if`: Missverständnisse

FASCHINGBAUER

Folgendes ist zwar korrekt und sieht aus wie aus einer “normalen” Programmiersprache ...

```
if (! rm -f /etc/passwd); then
    echo Hoppla, fehlgeschlagen
fi
```

... aber:

- *Subprozess*
- Innerhalb: Sequenz von Commands
- Exitstatus ist der des letzten Commands

Overview

FASCHINGBAUER

Kreativität kennt keine Grenzen

FASCHINGBAUER

"Syntaxerweiterung": Shell kennt keine "[]" Metacharacters

- "File Tests"
- Stringvergleiche (alphabetisch und numerisch)

```
if [ -f /etc/passwd -a $USER = root ]; then
    rm /etc/passwd
fi
if test -f /etc/passwd -a $USER = root; then
    rm /etc/passwd
fi
```

test (2)

FASCHINGBAUER

`-e entry`	`entry` existiert
`-f entry`	`entry` ist ein File
`-d entry`	`entry` ist ein Directory
`-r entry`	`entry` ist lesbar
`-w entry`	`entry` ist schreibbar
`-x entry`	`entry` ist ausführbar
`-a`	UND-Verknüpfung
`-o`	ODER-Verknüpfung
`A -lt B`	A ist kleiner als B
`!`	Negation (Achtung: Spaces)

Mehr → `man test`

Overview

FASCHINGBAUER

&& und ||

FASCHINGBAUER

Wie in "normalen" Programmiersprachen:

- `A && B`
  - Wenn nicht `A`, dann ist der ganze Ausdruck nicht
  - Wenn `A`, dann ist der ganze Ausdruck das, was `B` ist
- `A || B`
  - Wenn `A`, dann ist der ganze Ausdruck `A`
  - Wenn nicht `A`, dann ist der ganze Ausdruck `B`
- $\rightarrow$ "abgekürzte Berechnung"
- && zieht vor ||
- Eingebaute Shell-Syntax, *keine Erweiterung*

Boolesche Ausdrücke in Aktion (1)

FASCHINGBAUER

```
$ true && false && echo true || echo false
false
$ true && true && echo true || echo false
true
$ (true || false) && echo true || echo false
true
$ (false || true) && echo true || echo false
true
$ { false || true; } && echo true || echo false
true
```

Boolesche Ausdrücke in Aktion (2)

FASCHINGBAUER

```
if grep jfasch /etc/passwd && rm -f /etc/passwd; then
    echo Kaputte Userdatenbank geloescht 1>&2
fi
```

```
rm -f /etc/passwd && echo ok 1>&2 || echo fehler 1>&2
```

Boolesche Ausdrücke in Aktion (3)

FASCHINGBAUER

```
[ $(stat -c%s /var/log/syslog) -gt 100000 ] && \
    rm -f /var/log/syslog && \
    /etc/init.d/syslog restart || \
    echo Fehler bei Logrotieren 1>&2
```

Overview

FASCHINGBAUER

while (1)

FASCHINGBAUER

while Schleifen laufen, solange die Bedingung wahr ist ...

Endlosschleife

```
while true; do
    echo Murmeltier
done
```

(Erraten: true und false sind ganz normale Programme mit passendem Exit-Status)

while (2)

FASCHINGBAUER

Abbruchbedingung gleich wie in if ...

```
# Warten, bis User jfasch existiert
while ! grep jfasch /etc/passwd 1>/dev/null 2&>1; do
    echo Noch immer nicht da, tss ...
    sleep 1
done

while [ "X" != "U" ]; do
    echo Ich lass mir nichts vormachen!
done
```

Overview

FASCHINGBAUER

for (1)

FASCHINGBAUER

`for` Schleifen laufen über eine vorgegebene Liste → "Schleifenvariable" ...

User `jfasch` und `bheide` anlegen

```
for user in jfasch bheide; do
    useradd -m $user
done
```

for (2)

FASCHINGBAUER

Oft kombiniert mit *Command Substitution* ...

Zeilenanzahl des Kernels

```
for file in $(find /usr/src/linux/ -name \*.h -o -name \*.c);
    cat $file
done | wc -l
```

Vorsicht: Commandlines werden oft lang → Ressourcenengpass!

Overview

FASCHINGBAUER

Fallunterscheidung: case

FASCHINGBAUER

Unterscheidung von mehreren Fällen, basierend auf Strings und Patternmatches ...

```
case $1 in
    ja|Ja|yes|Yes)
        echo JA
        ;;
    nein|Nein|no|No)
        echo NEIN
        ;;
    *)
        echo VIELLEICHT
        ;;
esac
```

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Eine Funktion ist ...

FASCHINGBAUER

Shell ist eine vollständige Programmiersprache → **Funktionen**

- rekursiv aufrufbar
- → Variablen sollten mit `local` "deklariert" sein
- Behandelt wie normale Commands → Pipe und Redirection wie gewöhnlich
- *kein* Subprozess: nur Gruppierung wie mit {}
- Parameter behandelt wie Script-Argumente ($1 ...)

Eine einfache Funktion

FASCHINGBAUER

Simpelster Verwendungszweck: zusammenfassen von immer gleichen Sequenzen ...

Ausgeben von Messages auf Standard Error

```
message() {
    echo $* 1>&2
}
rm -f /etc/passwd || message Jessas
```

Rekursion

FASCHINGBAUER

Fibonacci-Zahlen

```
fibonacci() {
    local sum

    sum=$(($1+$2))
    echo $sum
    fibonacci $2 $sum
}
fibonacci 0 1 | less
```

(Pipe in `less` empfohlen, da sonst sehr schnell Overflow erreicht ist.)

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Uff, waren das noch nicht genug Details?!

FASCHINGBAUER

Wir hatten bisher so ziemlich alles, was eine Programmiersprache braucht — **warum müssen wir jetzt noch tiefer gehen?**

- Shell ist nicht hilfreich beim Debugging → "Draufhauen bis es geht" geht nur bis zu einem gewissen Grad
- Man kann sehr viel mehr machen, wenn die Macht mit einem ist
- Zusammenhänge zu verstehen ist nie schlecht
- ...

Overview

FASCHINGBAUER

Wie interpretiert die Shell die Commandline?

FASCHINGBAUER

Shell Quoting ist die Hölle → ein wenig Verständnis tut Not ...

Folgende Schritte, einer nach dem anderen:

1. Auftrennung der Zeile in "Wörter" → Metacharacters und deren Quoting
2. Auftrennung in "Compound Commands"
3. Verschiedene "Expansions"
4. Redirections der einzelnen Commands

Erster Schritt: Word Splitting, Quoting (1)

FASCHINGBAUER

Eine Zeile wird nach sogenannten “Metacharacters” aufgespalten; sie markieren ein Wortende (ausser wenn *gequoted*):

Shell Metacharacters

```
| & ; ( ) < > space tab
```

Erster Schritt: Word Splitting, Quoting (2)

FASCHINGBAUER

Beispiel:
`$ echo Hallo && echo Hello`
ergibt die Worte

1. `echo`
2. `Hallo`
3. `&&` → Spezielles Wort: *Operator*
4. `echo`
5. `Hello`

Erster Schritt: Word Splitting, Quoting (3)

FASCHINGBAUER

Compound Command

```
$ echo Hallo && echo Hello
Hallo
Hello
```

"Quoting" nimmt Zeichen spezielle Bedeutung:

&&, literal genommen

```
$ echo Hallo \&\& echo Hello
Hallo && echo Hello
```

Erster Schritt: Word Splitting, Quoting (4)

FASCHINGBAUER

Weitere Möglichkeiten ...

- `'&&'`
- `"&&"`

Beispiel: `echo $PATH`

- `echo \$PATH`
- `echo '$'PATH`
- `echo '$P'ATH`
- `echo '$PATH'`

Erster Schritt: Word Splitting, Quoting (5)

FASCHINGBAUER

Nun endlich die Quoting-Definition:

- **Backslash**, '\'
  - Nimmt dem nächsten Zeichen seine spezielle Bedeutung → "literal"
  - Ausser, das nächste Zeichen ist ein Newline, dann werden '\' und Newline ignoriert → "Line Continuation"
- **Single Quotes**, ' ': Nehmen allen Zeichen ihre spezielle Bedeutung. Achtung: selbst Backslash-Escapen von ' hilft nix, da der Backslash auch seine Bedeutung verliert.
- **Double Quotes**, "": Nehmen allen Zeichen ihre spezielle Bedeutung, ausser `$`, `\` und `!`

Zweiter Schritt: Brace Expansion

FASCHINGBAUER

Geschwungene Klammern ("Braces") werden "kreuzweise" ausmultipliziert ...

Brace Expansion, sinnvoll eingesetzt

```
$ touch SomeClass.hpp SomeClass.cpp
# oder kürzer ...
$ touch SomeClass.{hpp,cpp}
```

Brace Expansion, sinnfrei und rekursiv

```
$ echo a{b,c,d{x,y}}e
abe ace adxe adye
```

Weitere Schritte (1)

FASCHINGBAUER

- Tilde Expansion: z.B. `~jfasch` → `/home/jfasch`
- Command Substitution: `grep $(whoami) /etc/passwd`
- Rechnen: `echo "Wie viel ist $((1+1))?"`

Pathname Expansion ("Globbing")

FASCHINGBAUER

Shell Globs, Beispiele

```
$ echo *.tex
5010-Bash-Commandline.tex 5020-Bash-Variables.tex ...
$ echo 5[01]?0-*.tex
5010-Bash-Commandline.tex ... 5110-Paths.tex ...
$ echo *xxx
*xxx
```

Overview

FASCHINGBAUER

Standard I/O Streams

FASCHINGBAUER

- Drei Standard I/O *Filedescriptoren*
- Alle drei sind an das Terminal gekoppelt

Bedeutung	Name	Descriptor-Nummer	C-Macro
Standard Input	`stdin`	0	`STDIN_FILENO`
Standard Output	`stdout`	1	`STDOUT_FILENO`
Standard Error	`stderr`	2	`STDERR_FILENO`

Religiös eingehaltene Tradition unter Unix:

- Debug/Error-Output geht nach *Standard Error* und *nicht* nach Standard Output. Standard Output ist für die Pipe da.
- Programme sind nur in Ausnahmefällen nicht für die Pipe gedacht.

I/O Redirection

FASCHINGBAUER

Redirection mit Hilfe der Shell:

`command < file`	`command` bekommt `file` auf Standard Input
`command > file`	`command` schreibt Standard Output auf `file`
`command 2> file`	`command` schreibt Standard Error auf `file`

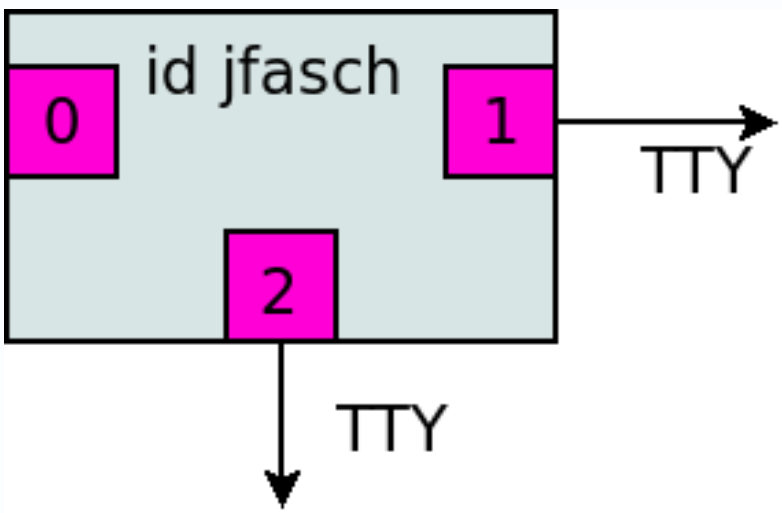
- '>' löscht den Inhalt des Files vorher, falls es existiert
- $\rightarrow$ '>>', um anzuhängen

Keine Redirection

FASCHINGBAUER

Keine Redirection

```
$ id jfasch
uid=1000(jfasch) gid=1000(jfasch) groups=...
```

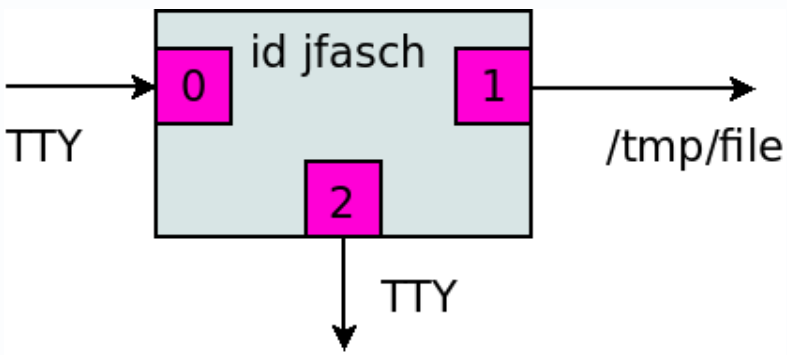


Output Redirection

FASCHINGBAUER

Output Redirection

```
$ id jfasch > /tmp/file
# oder ...
$ id jfasch 1> /tmp/file
```

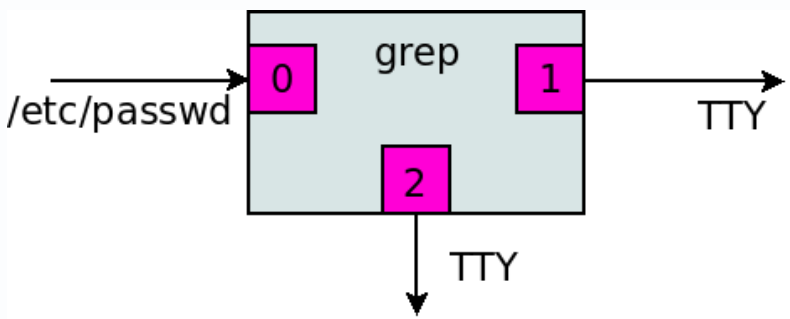


Input Redirection

FASCHINGBAUER

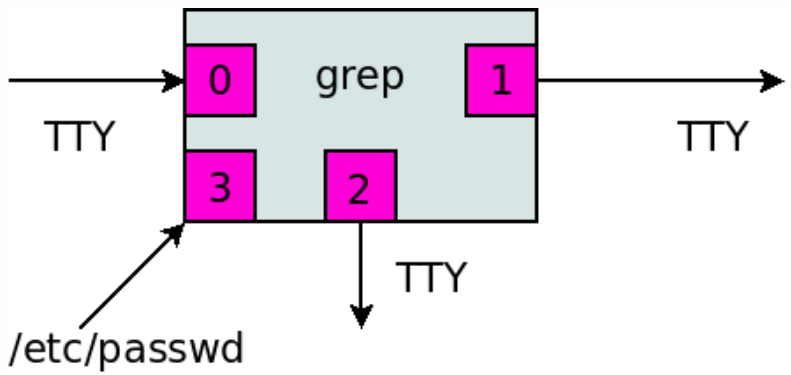
Input Redirection

```
$ grep jfasch < \
    /etc/passwd
```



grep ohne Redirection

```
$ grep jfasch /etc/passwd
```



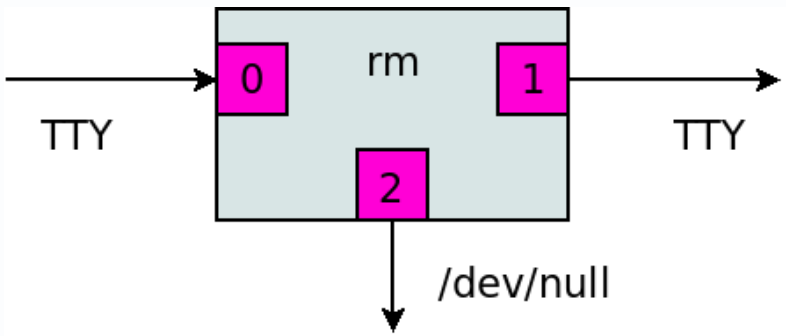
Error Redirection

FASCHINGBAUER

Fehlermeldungen unterdrückt man so ...

Error Redirection

```
$ rm -f /etc/passwd 2> /dev/null
```

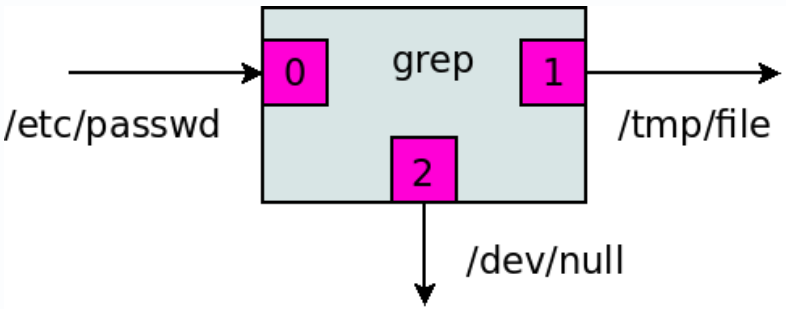


Alles Redirection

FASCHINGBAUER

Alles Redirection

```
$ grep jfasch < /etc/passwd > /tmp/file 2> /dev/null
```



Jonglieren (1)

FASCHINGBAUER

Programme, die auf `stdout` schreiben, sind gute Mitglieder einer Pipe ...

Zählen der Einträge in `/etc`

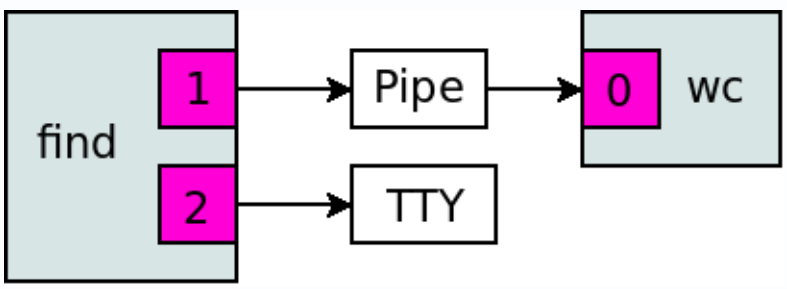
```
$ find /etc | wc -l
find: ‘/etc/cron.daily’: Permission denied
find: ‘/etc/sudoers.d’: Permission denied
find: ‘/etc/cron.weekly’: Permission denied
...
1558
```

Jonglieren (2)

FASCHINGBAUER

Aktionen der Shell:

- Alloziert Pipe ($\rightarrow$ `man 2 pipe`)
- Dupliziert `stdout` von `find` auf den Input der Pipe
- Dupliziert `stdin` von `wc` auf den Output der Pipe ($\rightarrow$ `man 2 dup`)



Jonglieren (3)

FASCHINGBAUER

Problem: wie zählt man Fehlermeldungen?
→ Vertauschen von `stdout` und `stderr`

Zählen der Fehlermeldungen

```
$ find /etc 3>&1 1>&2 2>&3 | wc -l
```

Häh?!

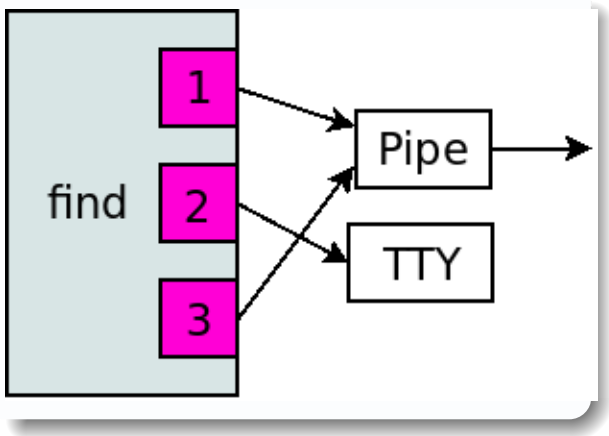
- Evaluierung von links nach rechts (`3>&1` vor `1>&2` ...)
- `A>&B` heisst: "Mach, dass Filedeskriptor A auf das gleiche zeigt wie Filedeskriptor B!"

Jonglieren (4)

FASCHINGBAUER

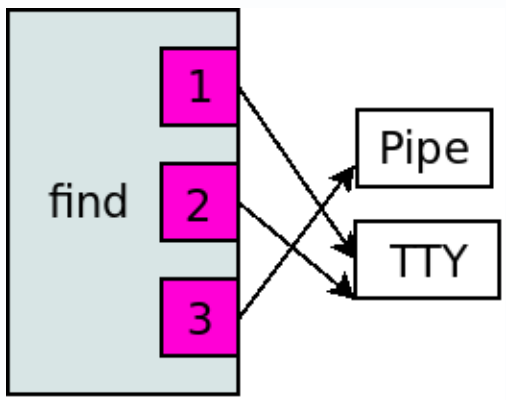
`3>&1`

Wegspeichern von `stdout` nach 3



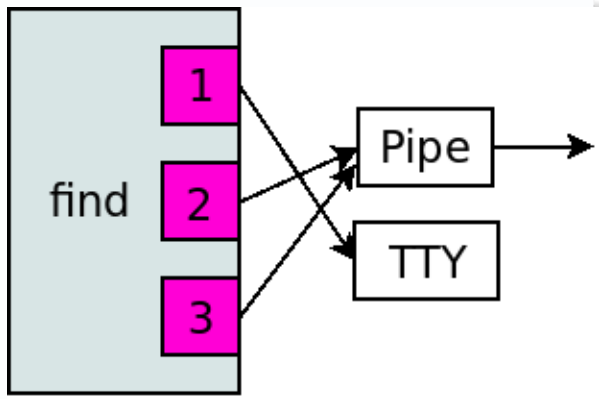
`1>&2`

`stderr` nach `stdout` zeigen lassen



`2>&3`

`stderr` mit Ex-`stdout` überschreiben



- Optional: Schliessen von 3: `3>&-`

Overview

FASCHINGBAUER

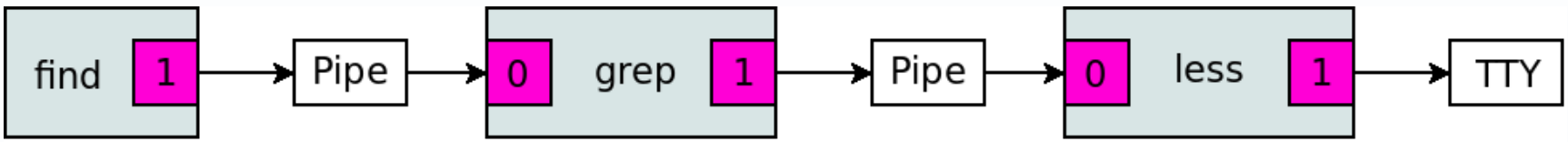
Pipes (1)

FASCHINGBAUER

- Weiterer UNIX-Leitsatz: jedes Werkzeug soll eine Sache machen, und das gut
- Pipe kombiniert Werkzeuge

Alle Kernel-Files, die nicht compiliert werden

```
$ find /usr/src/linux/|grep -v \*.c|less
```



Pipes (2)

FASCHINGBAUER

- *Keine* temporären Files involviert (unter Doze schon)
- Kommunikationsmechnismus
- Pipe = Buffer von beschränkter Größe
- Linker und rechter Prozess arbeiten *gleichzeitig* ($\rightarrow$ Scheduling)
- Schreibender Prozess wird suspendiert, wenn Pipe voll
- Lesender Prozess wird suspendiert, wenn Pipe leer

Pipes: Beispiele (1)

FASCHINGBAUER

Alle User im System, alphabetisch

```
$ cat /etc/passwd|cut -d : -f 1|sort
# effizienter:
$ cut -d : -f 1 < /etc/passwd|sort
```

..., mit Gruppen und IDs

```
$ cut -d : -f 1 < /etc/passwd| \
    sort | \
    while read user; do id $user; done
```

Pipes: Beispiele (2)

FASCHINGBAUER

..., zwischengespeichert in `/tmp/users.txt`

```
$ cut -d : -f 1 < /etc/passwd| \
    sort | \
    tee /tmp/users.txt | \
    while read user; do id $user; done
```

`tee` ... *T-Stück*, als Verbinder zwischen zwei Rohren (Pipes)

Named Pipes

FASCHINGBAUER

- Rendezvous wie (unnamed) Pipe, nur im Filesystem
- Wird behandelt wie ein File, aber: Inhalt nicht persistent
- → `man mkfifo`, `man 3 mkfifo`, `man 7 fifo`

In einem Terminal ...

```
$ mkfifo /tmp/fifo
$ echo Hallo > /tmp/fifo
```

In einem anderen Terminal ...

```
$ cat /tmp/fifo
```

Overview

FASCHINGBAUER

IO-Redirection, Pipes: Übungen (1)

FASCHINGBAUER

- Kopieren Sie mit Hilfe von `cat` (ohne Argumente) `/etc/passwd` nach `/tmp`.
- Erstellen Sie mittels mehrerer Aufrufe von `echo` ein File, das Ihren Namen und Ihre Adresse enthält.
- Geben Sie eine sortierte Liste aller User in Ihrem System aus (`/etc/passwd` enthält alle User samt deren Einzelheiten, durch ':' getrennt).
- Leiten Sie `stdout` *und* `stderr` des Commands `find /etc` nach `/tmp/output` um.
- Warum ist das (nichtleere) File `/tmp/output` nach dem Command `cat < /tmp/output > /tmp/output` leer?
- Erklären Sie den Effekt des Commands (`/tmp/output` ist wiederum nicht leer) `cat < /tmp/output >> /tmp/output`!

IO-Redirection, Pipes: Übungen (2)

FASCHINGBAUER

- Wieviele Directories enthält Ihr Homedirectory? (Hinweis: kombinieren Sie `find` und `wc` mit einer Pipe)
- Statten Sie das Command `find /etc | wc` mit geeigneten Umleitungen aus, sodass die Anzahl der Fehler gezählt wird und `stdout` von `find` nach `/dev/null` geht.
- Schicken Sie unter Zuhilfenahme von `echo`, `cat` und `mkfifo` das Wort "Hallo" im Kreis herum.

Overview

FASCHINGBAUER

Die Shell, Pipe, und Filedeskriptoren

FASCHINGBAUER

Die Shell kann sehr gut mit Pipes, Redirections und Filedeskriptoren

- *Es gibt meistens einen besseren Weg, als Temporärfiles zu schreiben!*
- Redirections und Pipes überall anwendbar, nicht nur bei regulären Commands
  - Schleifen
  - Funktionen
  - Blöcke
  - ...

Zum Beispiel (1)

FASCHINGBAUER

```
get_syslog() {
    ssh $machine 'cat /var/log/syslog'
}

MACHINES="192.168.1.2 192.168.1.20 192.168.1.23"
for machine in $MACHINES; do
    get_syslog $machine > \
      ~/logs/syslog.$machine.$(date %Y-%m-%d-%H:%M:%S)
done
```

Zum Beispiel (2)

FASCHINGBAUER

Funktion als Pipe-Input

```
PATTERN='funky-driver'
get_syslog 192.168.1.2 | grep $PATTERN > \
    ~/logs/funky-driver.192.168.1.2.log
```

Pipe zwischen Funktionen

```
PATTERN='funky-driver'
get_syslog 192.168.1.2 | funky_filter > \
    ~/logs/funky-driver.192.168.1.2.log
```

Zum Beispiel (3)

FASCHINGBAUER

```
hiccup=false; message=
get_syslog 192.168.1.2 | while read line; do
    if echo $line | grep 'funky-driver'; then
        hiccup=true; message=$message+$line
    else
        if [ $hiccup = true ]; then
            echo $message | \
               mail -s 'Another hiccup seen!' \
               developer@company.com
            hiccup=false; message=
        fi
    fi
done
```

Overview

FASCHINGBAUER

Overview

Subshell: weitermachen in einem *eigenen Prozess*

FASCHINGBAUER

Problem: man möchte Shell-Code ausführen, sich dadurch aber nicht den Scope verschmutzen
Lösung: Subshells

Wie gross wärs?

```
$SRCDIR=$1
(
    cd $SRCDIR;
    tar -jcf - .
) | wc -c
```

Subshell: wie geht das?

FASCHINGBAUER

Folgende Tatsachen:

- Der Code innerhalb der Klammern wird in einem eigenen Prozess ausgeführt
- → alle Variablen des umschliessenden Scopes sind sichtbar
- → keine Änderung, die innerhalb gemacht wird, ist heraussen sichtbar
  - Änderung des CWD
  - Änderung an Variablen
  - ...
- Exitstatus ist der des zuletzt ausgeführten Commands
- Redirection- und Pipe-Operatoren gelten wie bei einzelnen Commands

Overview

FASCHINGBAUER

Blöcke: Gruppieren von Commands

FASCHINGBAUER

Problem: man möchte Redirection- und Pipe-Operatoren auf eine Gruppe von Commands anwenden

Lösung: Blöcke — geschweifte Klammern

- Änderungen innerhalb der Klammern ausserhalb sichtbar
- Redirection- und Pipe-Operatoren gelten für jedes Command innerhalb der Klammern
- Alle Schleifen und `if` Statements sind von sich aus Blöcke

Blöcke: Beispiel

FASCHINGBAUER

```
workdir=$(pwd)
echo Variable workdir vorher ist $workdir
echo CWD vorher ist $(pwd)
{
    cd /
    workdir=$(pwd)
    ls -1
} | wc -l
echo Variable workdir nachher ist $workdir
echo CWD nachher ist $(pwd)
```

Overview

FASCHINGBAUER

Shell-Code “inkludieren”

FASCHINGBAUER

Problem: Verwenden von gemeinsamem Code in *verschiedenen* Scripts

- Setzen von Variablen und “Konstanten”
- Definieren von gemeinsam verwendeten Funktionen

Lösung: inkludieren mit “.”

Library

FASCHINGBAUER

common.shlib

```
message() {
    echo $* 1>&2
}
```

einscript

```
#!/bin/sh

. ./common.shlib
message Hurra
```

Overview

FASCHINGBAUER

Overview

Stringmanipulation leicht gemacht (1)

FASCHINGBAUER

Der Zauberbegriff **“Parameter Expansion”** vereint folgende nicht endenwollende Liste von praktischen Obskuritäten. Entnommen direkt aus `info bash`, bitte vor dem geistigen Auge `PARAMETER` durch `VARIABLE` ersetzen.

- `${PARAMETER:OFFSET}`: Teilstring von `$PARAMETER` ab `OFFSET`
- `${PARAMETER:OFFSET:LENGTH}`: Teilstring ab `OFFSET` mit Länge `LENGTH`
- `${PARAMETER#WORD}`: das kürzeste Stück vom Anfang weg, das dem Pattern `WORD` entspricht, wird weggeschnitten.
- `${PARAMETER##WORD}`: das längste Stück vom Anfang weg ...

Stringmanipulation leicht gemacht (2)

FASCHINGBAUER

- `${PARAMETER%WORD}`: das kürzeste Stück vom Ende weg ...
- `${PARAMETER%%WORD}`: das längste Stück vom Ende weg ...
- `${PARAMETER:-WORD}`: wenn `$PARAMETER` leer, dann ist der Ausdruck `WORD`
- `${PARAMETER:=WORD}`: wie "-", nur wird gleichzeitig `WORD` an `PARAMETER` zugewiesen

Falls noch nicht genug: bitte weiterlesen in der Bash-Dokumentation

Overview

Inline Dokumente — “Here Documents”

FASCHINGBAUER

Problem: man will direkt im Shellscript längeren Multiline-Text per Standard-Input an ein Command übergeben

```
wc -l <<EOF
Das ist eine Zeile
Das ist noch eine Zeile
EOF
```

Overview

FASCHINGBAUER

Spezielle Variablen

FASCHINGBAUER

Die Shell setzt eine Vielzahl interner Variablen, hier eine Auswahl

- `$$`: PID der ausführenden Shell
- `$?`: Exitstatus des letzten im Vordergrund ausgeführten Commands
- `$!`: PID des letzten im Hintergrund gestarteten Prozesses

Nachzulesen in `info bash`, “Special Parameters”

Overview

FASCHINGBAUER

Schlusswort

FASCHINGBAUER

Danke für euer Verständnis!

Falls es noch nicht reicht: hier Links zum weiterlesen

- `http://tldp.org/LDP/abs/html/`
- `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`