Korn-Shell-Programmierung und Zeichenkettenverarbeitung unter Unix/Linux

FWPF
WS 2005/2006 -- Februar

Überblick

- Einstieg Korn-Shell
  - Konfiguration, History-Funktion, Kommandozeilen-Editing
  - Kommandos und Kommandoverknüpfung
  - Kommandosubstitution
  - Reguläre Ausdrücke
  - Dateinamenexpansion
  - Quoting
  - Debugging

- Unix-Tools (Teil 1)
  - Auflistung von Dateinamen
  - Ausgabe von Zeilen genannter Dateien
  - Ausgabe von manipulierten Zeileninhalten genannter Dateien

- Korn-Shell (Fortsetzung)

Überblick

- Korn-Shell-Skripten
  - Skriptform & -aufruf
  - Skript I/O: print
  - Shellvariablen (Grundlagen), Parametersubstitution, Quoting
  - Datenübergabe
  - Datenrückgabe
  - Variablenmanipulation
  - Bedingte Verzweigung & Bedingungstest
  - Schleifen
  - Arrays & Positionsparameter
  - Variablenattribute
  - Ein-Ausgabe-Umlenkung
  - Benutzermenues
  - Optionsverarbeitung
  - Funktionen
  - zweifache Kommandozeilenverarbeitung
  - Zeitmessung

Überblick

- Unix-Tools (Teil 2)
  - `fgrep`, `grep`, `egrep`
  - `sed`

Literatur

- **Linux-Unix-Shells** — ksh (sh, bash, csh, tcsh)
  Helmut Herold — Bsp. als tgz-File-Download
  Addison-Wesley, 1999

- **Linux-Unix-Kurzreferenz** — ksh (sh, bash, csh, tcsh)
  Helmut Herold — Unix/Linux-Tools
  Addison-Wesley, 1999 — sed/awk

- **Learning the Korn Shell, 2nd Edition** — ksh88, ksh93
  Bill Rosenblatt & Arnold Robbins — Bsp. zum download
  O'Reilly, 2002

- **Skriptprogrammierung für Solaris & Linux** — ksh88/sh, perl, nawk
  Wolfgang Schorn, Jörg Schorn
  Addison-Wesley, 2004

Literatur

- **The New KornShell Command And Programming Language** — ksh93
  Morris Bolksy & David Korn — *„The authoritive reference“*
  Prentice Hall, 1995

- **Hands-On KornShell93 Programming** — ksh93
  Barry Rosenberg — (inkl. ksh93 Binary)
  Addison-Wesley, 1998

- **KornShell Programming Tutorial** — ksh88
  Barry Rosenberg — inkl. Diskette mit Bsp.
  Addison-Wesley, 1991

- **Desktop KornShell Graphical Programming** — ksh93, dtksh
  J. Stephen Pendergrast, Jr — *„The authoritive reference“*
  Addison-Wesley, 1995

Literatur

- **Unix in a Nutshell**
  Arnold Robbins
  O‘Reilly
  ksh
  Unix/Linux-Tools

- **Linux in a Nutshell**
  Ellen Siever et al.
  O‘Reilly
  Unix/Linux-Tools

- **Reguläre Ausdrücke**
  Jefrey E.F. Friedl
  O‘Reilly, 1998
  egrep

- **sed & awk, 2nd Edition**
  Dale Dougherty & Arnold Robbins
  O‘Reilly, 1997 (2nd Ed.)
  sed, awk
  Bsp. als tgz-File-Download

Literatur

- **Using csh & tcsh**
  Paul DuBois
  O‘Reilly, 1995

- **Learning the bash Shell**
  Cameron Newham & Bill Rosenblatt
  O‘Reilly, 2005 (3nd Ed.)

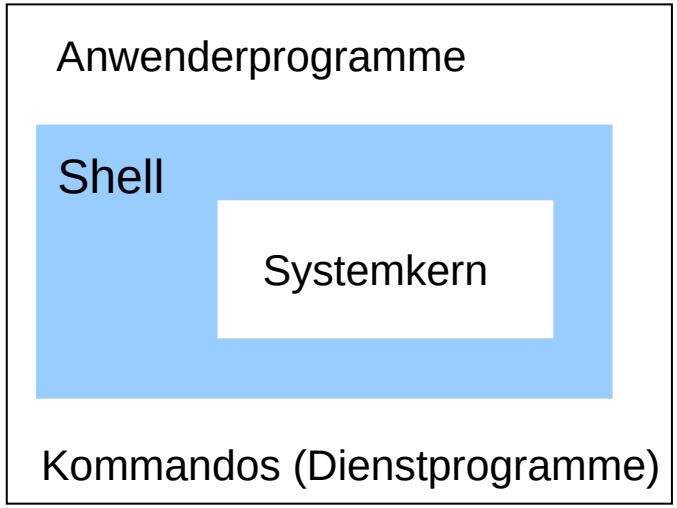
Links & Downloads

- Interessante Links
  - www.kornshell.com
  - www.cs.mun.ca-~michael/pdksh/pdksh/man.html
  - www.cs.princeton.edu/~jlk/kornshell/doc/man88.html
  - www.cs.princeton.edu/~jlk/kornshell/doc/man93.html

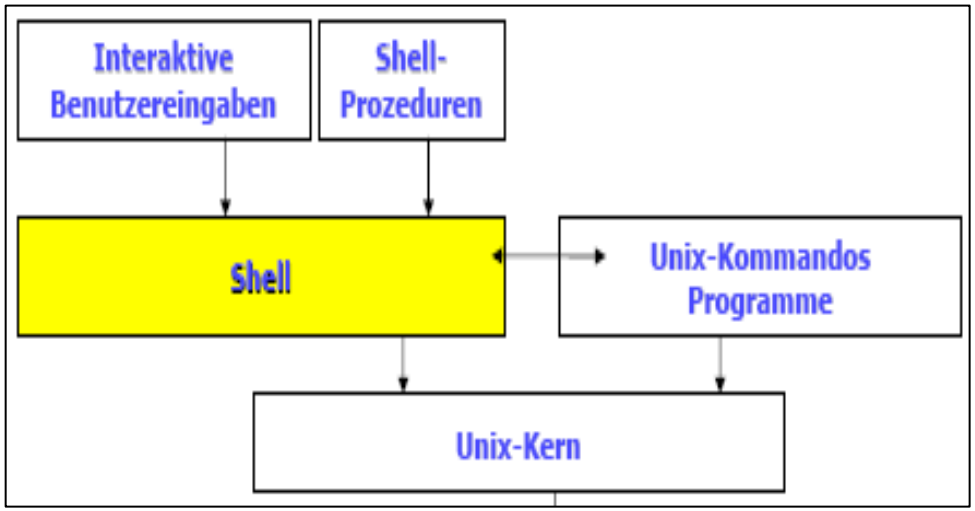
- Interessante Downloads von Beispielen
  - www.addison-wesley.de/service/herold
  - http://examples.oreilly.com/korn2/

Einstieg: Was sind Shells?

- Zugriffsschalen zur Nutzung von Betriebssystemfunktionen



(nach: H.Herold: Linux-Unix-Shells)



(aus: Uwe Wienkop: Foliensatz „Betriebssysteme“, Kap. 2, S. 60

- Kommandointerpreter
- Programmier-Funktionalität

Einstieg: Was ist Shellprogrammierung ?

- Jede Shell verfügt über eine spezifische
  - *Syntax*, nach welcher Kommandos eingegeben werden können, sowie
  - *Semantik*, welche die Bedeutung dieser Kommandoausdrücke, beschreibt.
  - → **Shell-Sprache**

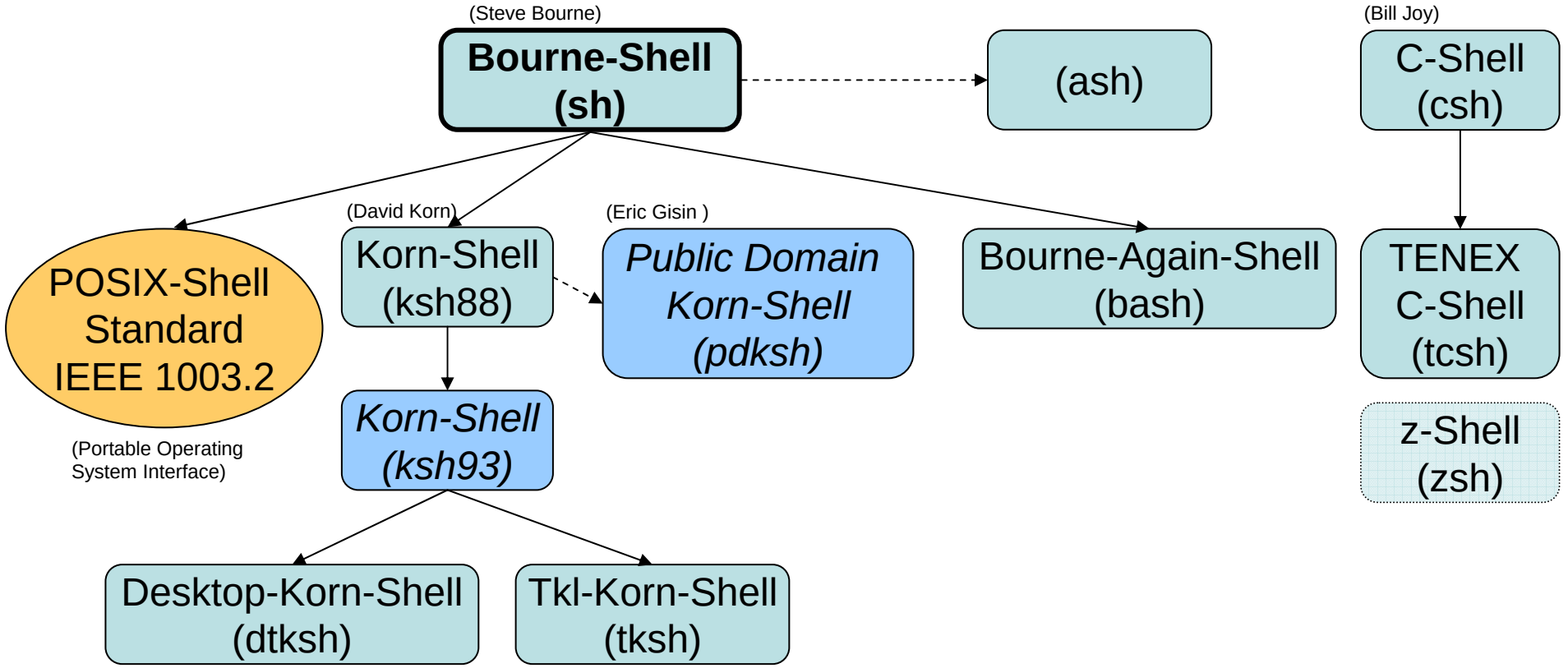
- Neben der Nutzung dieser Sprache direkt per Eingabe über Tastatur, können Kommandoausdrücke auch zu sog. **Skriptprogrammen** (kurz: *Skripten*) zusammengefasst werden.

- Skriptprogramme, welche einer Shellsprache genügen, können durch das zugehörige Shellprogramm korrekt **interpretiert** werden.
  (d.h. müssen nicht compiliert werden)

Einstieg: Shellprogrammierung wozu?

- Effizientes Arbeiten mit System
  - Eigener „Befehlssatz“
  - Customizing
- Vereinfachung/Automatisierung komplexer oder immer wiederkehrender Abläufe der Systemadministration
  - Routineabläufe
- Text/String-orientierte Auswertungen
  - Logfile-Analysen
- Rapid Prototyping

Einstieg: Welche Shell darf es sein?



- Login-Shell: siehe Datei `/etc/passwd`
- Linux: typischerweise: bash
- UNIX: weit verbreitet: ksh

Ableitung — Erweiterung

Einstieg: Warum Korn-Shell ?

- ***csh**: Shell with C-like syntax, standard login shell on BSD systems. The C shell was originally written at UCB to overcome limitations in the Bourne shell. Its flexibility and comfort (at that time) quickly made it the shell of choice until more advanced shells like ksh, bash, zsh or tcsh appeared. Most of the latter incorporate features original to csh.*

  Quelle: http://linux.about.com/cs/linux101/g/csh.htm/

- ***Tcsh** is an enhanced, but completely compatible version of the Berkeley UNIX C shell (csh). It is a command language interpreter usable both as an interactive login shell and a shell script command processor. It includes a command-line editor, programmable word completion, spelling correction, a history mechanism, job control and a C-like syntax.*

  Quelle: http://www.tcsh.org/Welcome

Einstieg: Welche Shell darf es sein?

- *The **KornShell language** was designed and developed by David G. Korn at AT&T Bell Laboratories. It is **an interactive command language** that provides access to the UNIX system and to many other systems, on the many different computers and workstations on which it is implemented. The KornShell language is also a **complete, powerful, high-level programming language** for writing applications, often more easily and quickly than with other high-level languages. This makes it especially suitable for prototyping. There are two other widely used shells, the Bourne shell developed by Steven Bourne at AT&T Bell Laboratories, and the C shell developed by Bill Joy at the University of California. ksh has the best features of both, plus many new features of its own.*

  Quelle: http://www.kornshell.com/info/

- *"Korn shell" was written by "David Korn", around **1982 at AT&T labs**. You can now freely download the full source from AT&T if you're not lucky enough to use an OS that comes with ksh already. **It's "open source",** even!*

  Quelle: http://www.bolthole.com/solaris/ksh.html

Einstieg: Welche Shell darf es sein?

- ***PD-ksh*** *is a* ***clone of the AT&T Korn shell****. At the moment, it has* ***most of the ksh88*** *features,* ***not much of the ksh93 features****, and a number of its own features. pdksh was initially created by Eric Gisin using Charles Forsyth's public domain V7 shell as a base as well as parts of the BRL shell.*

  ...

  it is currently maintained by Michael Rendell. pdksh's strong points are:

  – *its* ***free and quite portable*** *- you should be able to compile it easily on pretty much any unix box.*

  ...

  Quelle: http://www.cs.mun.ca/~michael/pdksh/

Einstieg: Welche Shell darf es sein?

- *The **Desktop Korn Shell** (DtKsh) that comes with the Common Desktop Environment (CDE) is **built on the ksh93 standard** with X, Xt, Motif, ToolTalk and CDE built-in APIs. Unlike Perl and Tcl/Tk, major vendors have built and supported DtKsh through the CDE initiative. Using DtKsh, desktop programmers can develop and/or prototype plug-and-play Graphical User Interface (GUI) applications that are compatible on all CDE-compliant systems without compilation. Although DtKsh applications are interpreted for portability, they can easily be migrated to Motif in C for performance.*

  Quelle: http://www2.linuxjournal.com/article/2643
- ***Tksh** is an implementation of the **Tcl C library written on top of the library for the new KornShell (ksh93)**. Tksh emulates the behavior of Tcl by using the API that is provided for extending ksh93, which is similar to the Tcl library in that it allows access to variables, functions and other state of the interpreter. This implementation allows Tcl libraries such as Tk to run on top of ksh93 unchanged, making it possible to use shell scripts in place of Tcl scripts. ksh93 is well suited for use with Tk because it is backward compatible with sh, making it both easy to learn and easy to extend existing scripts to provide a graphical user interface. Tksh is not yet another port of Tk to another language -- it **allows Tcl scripts to run without modification using the ksh93** internals.*

  (Quelle: http://www.cs.princeton.edu/~jlk/tksh/)

Einstieg: Welche Shell darf es sein?

- *The **ash** shell is a clone of Berkeley's Bourne shell (sh).*

  Ash supports all of the standard sh shell commands, but is considerably smaller than sh. The ash shell lacks some Bourne shell features (for example, command-line histories), but it uses a lot less memory.

  Quelle: http://sparemint.atariforge.net/sparemint/html/packages/ash.html

- ***Bash** is the shell, or command language interpreter, that will appear in the GNU operating system. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.*

  Quelle: http://www.gnu.org/software/bash/bash.html

Einstieg: Warum Korn-Shell ?

- ***Zsh*** *is a shell designed for interactive use, although it is also a powerful scripting language.* ***Many of the useful features of bash, ksh, and tcsh*** *were incorporated into zsh;* ***many original features were added****.*

  Quelle: http://zsh.sourceforge.net

- ***Zsh*** *is a UNIX command interpreter (shell) usable as an interactive login shell and as a shell script command processor. Of the standard shells, zsh* ***most closely resembles ksh*** *but includes many enhancements. Zsh has command line editing, builtin spelling correction, programmable command completion, shell functions (with autoloading), a history mechanism, and a host of other features.*

  Quelle: http://zsh.sourceforge.net/Doc/Release/zsh_2.html#SEC3

Einstieg: Shell-Vergleich

Funktionalität	sh	ksh	bash	csh	tcsh
Interaktive Kommandoeingabe:					
• **History-Mechanismus**		X	X	X	X
• **Directory-Stack**			X	X	X
• **CDPATH-Variable**	X	X	X	X	X
• **Alias-Mechanismus**		X	X	X	X
• **Alias-Argumente**				X	X
• **Kommandozeilen-Editing**		X	X		X
• **variabler Promptstring**		X	X	X	X
• **Spelling-Correction** (user-IDs, Kdos, Dat.namen)					X
• **Aliasnamen-Completion**			X		X
• **Kommandonamen-Completion**			X	X	X
• **Dateinamen-Completion**		X	X	X	X
• **Funktionsnamen-Completion**			X	X	X
• **Hostname-Completion, Variablennamen-Completion**			X		

Einstieg: Shell-Vergleich

Funktionalität	sh	ksh	bash	csh	tcsh
I/O:					
• **Ein-/Ausgabeumlenkung**	X	X	X	X	X
Expansionsmechanismen:					
• **Alias-Erkennung**		X	X	X	X
• **Kommandosubstitution**	X	X	X	X	X
• **Dateinamensubstitution**	X	X	X	X	X
• **Parametersubstitution**	X	X	X	X	X
• **Funktionsnamen-Erkennung**	X	X	X		

Einstieg: Shell-Vergleich

Funktionaltät	sh	ksh	bash	csh	tcsh
Kontrollstrukturen:					
• **bedingte Verzweigung / computed goto**	X	X	X	X	X
• **Schleifenkonstrukte**	X	X	X	X	X
Variablen:					
• **Arrays**		X	X		X
• **formatierte Variablen**		X			
• **Zufallszahlen**		X	X		
• **readonly-Variablen**	X	X	X		X
Operatoren:					
• **arithmetische Operatoren**		X	X	X	X
• **stringmanipulierende Operatoren**		X	X		
Funktionen:					
• **Funktionsdefinitionen**	X	X	X		
• **funktionslokale Variablen**		X	X		
• **Funktionen-Autoload**		X			

Einstieg: Shell-Vergleich

Funktionaltät	sh	ksh	bash	csh	tcsh
Prozesse:					
• **wait**	X	X	X	X	X
• **Koprozesse**		X			
• **Jobkontrolle**		X	X	X	X
Signale:					
• **Signalbehandlung**	X	X	X	X	X
• **Signalnamen**		X	X	X	X

Einstieg: Warum Korn-Shell ?

- Verfügbarkeit auf UNIX *und* Linux
- (weitestgehende) Aufwärtskompatibilität gegenüber Bourne-Shell
- Hohe Portabilität (POSIX compliant)
- Gegenüber C- und gegenüber Bourne-Again-Shell (`bash`) gleichwertige Funktionalität
- z.T. höhere Performanz als Bourne-Shell oder csh
- Korn-Shell 88 (AT&T) üblicherweise auf UNIX-Systemen verfügbar: „**ksh88**“
  - MAC OS, SUN Solaris, HP-UX
- Korn-Shell wird beständig weiterentwickelt.
- Weiterentwicklung Korn-Shell 93 (AT&T): „**ksh93**“
  - FB-IN Linux-System (/usr/local/bin/ksh93): Version M 1993-12-28 r
  - Teil des CDE auf kommerziellen Unix-Systemen (dtksh, z.B: HP-UX)
- Unter Suse-Linux: **pdksh**
  - Open Source Implementierung der Korn-Shell 88 (z.T. mit Ergänzungen u. Lücken)
  - Lokal in FB-IN installierte Version pdksh-5.2.14 99/07/13.2

Einstieg: Warum Korn-Shell ?

- Erweiterte Funktionalität der Korn-Shell ‘88 gegenüber Bourne-Shell (sh), z.B.:
  - Editieren von Kommandozeilen
  - History-Mechanismus
  - Builtin-Arithmetik
  - Attributierung von Variablen
  - Komplexere reguläre Ausdrücke
  - Erweiterte Möglichkeiten der Stringverarbeitung
  - Erweiterte Operatoren-Syntax (insbesondere Arithmetik) in Anlehnung an die Programmiersprache C
  - Z.T. Möglichkeiten der kompakteren/übersichtlicheren Schreibweise von Kommandos

Einstieg: Warum Korn-Shell ?

- Erweiterte Funktionalität der Korn-Shell ‘93 gegenüber Korn-Shell ‘88, z.B.:
  - Arithmetische for-Schleife
  - Gleitkommavariablen, -arithmetik und –funktionen
  - Indirekte Benutzung von Variablen
  - Assoziative Arrays
  - Erweiterte Möglichkeiten des Mustervergleichs (z.B. back reference)
  - Erweiterte Möglichkeiten der Stringverarbeitung (z.B. Selektion von Teilstrings)
  - u.a.

Einstieg: Starten der Korn-Shell

- In aktueller Shell eingeben:
  `ksh`

- Es erscheint neues Promptsymbol **`$`**

- Prüfung, ob wirklich Korn-Shell gestartet wurde

```
$ ps
$ print $$          # Alternative: print $0
```

- Weitere Eingaben

```
$ cd
$ more .kshrc
$ more .mykshrc
```

- Ausstieg aus `ksh`

```
$ exit
```

Einstieg: Starten der Korn-Shell

- Prüfung, welche Korn-Shell verfügbar ist

```
$ which ksh
/bin/ksh                        # Standardinstallation

$ which ksh93
/usr/local/bin/ksh93
```

- Welche Korn-Shell-Version läuft?

  Nochmals in aktueller Shell eingeben:
  ksh bzw. ksh93

```
$ print $KSH_VERSION

$ set –o emacs
$ CTRL-v
```

- Ausstieg aus ksh

```
$ exit
```

Einstieg: Konfigurieren und Starten der Korn-Shell

- In welcher Shell bin ich überhaupt:?

  ```
  <0>$ ps
  ```

- Prüfen, ob folgende Dateien im Homeverzeichnis vorhanden sind:
  `.profile` **`.bashrc`** **`.mybashrc`** (falls `bash` Login-Shell)

- Sichern der Dateien

  ```
  <1>$ cp .profile .profile.orig
  <2>$ cp .bashrc .bashrc.orig
  <3>$ cp .mybashrc .mybashrc.orig
  ```

- Sicherstellen, daß **`.mybashrc`** über die Datei **`.profile`** eingelesen wird.

  ```
  <4>$ cd /afs/informatik.fh-nuernberg.de/projects/Fuhr/korn-shell/config
  <5>$ cp .profile $HOME
  <6>$ cp .bashrc $HOME
  <7>$ cp .mybashrc $HOME
  <8>$ cd
  ```

Einstieg: Konfigurieren und Starten der Korn-Shell

- Prüfen ob folgende Dateien im Home-Verzeichnis vorhanden sind:

  `.kshrc  .mykshrc`

- Wenn nicht vorhanden:

  ```
  <8>$ cp /afs/.../project/korn-shell/config/.kshrc $HOME
  <9>$ cp /afs/.../project/korn-shell/config/.mykshrc $HOME
  ```

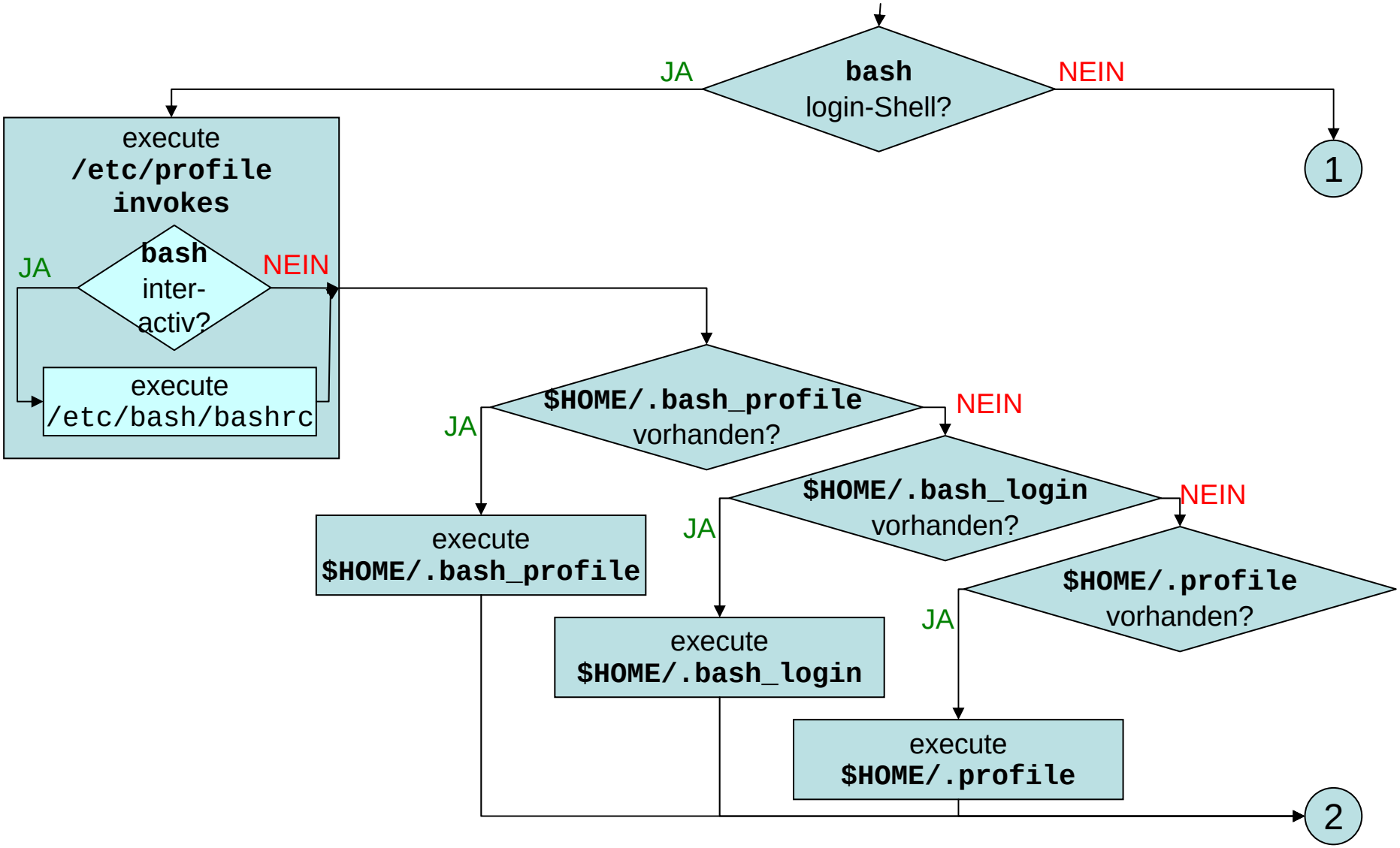
- Wenn vorhanden:
  - **Sichern** und neue aus Projektverzeichnis holen

- Wenn alle Konfigurationsfiles vorhanden nochmals Korn-Shell aus der Login-Shell heraus starten:

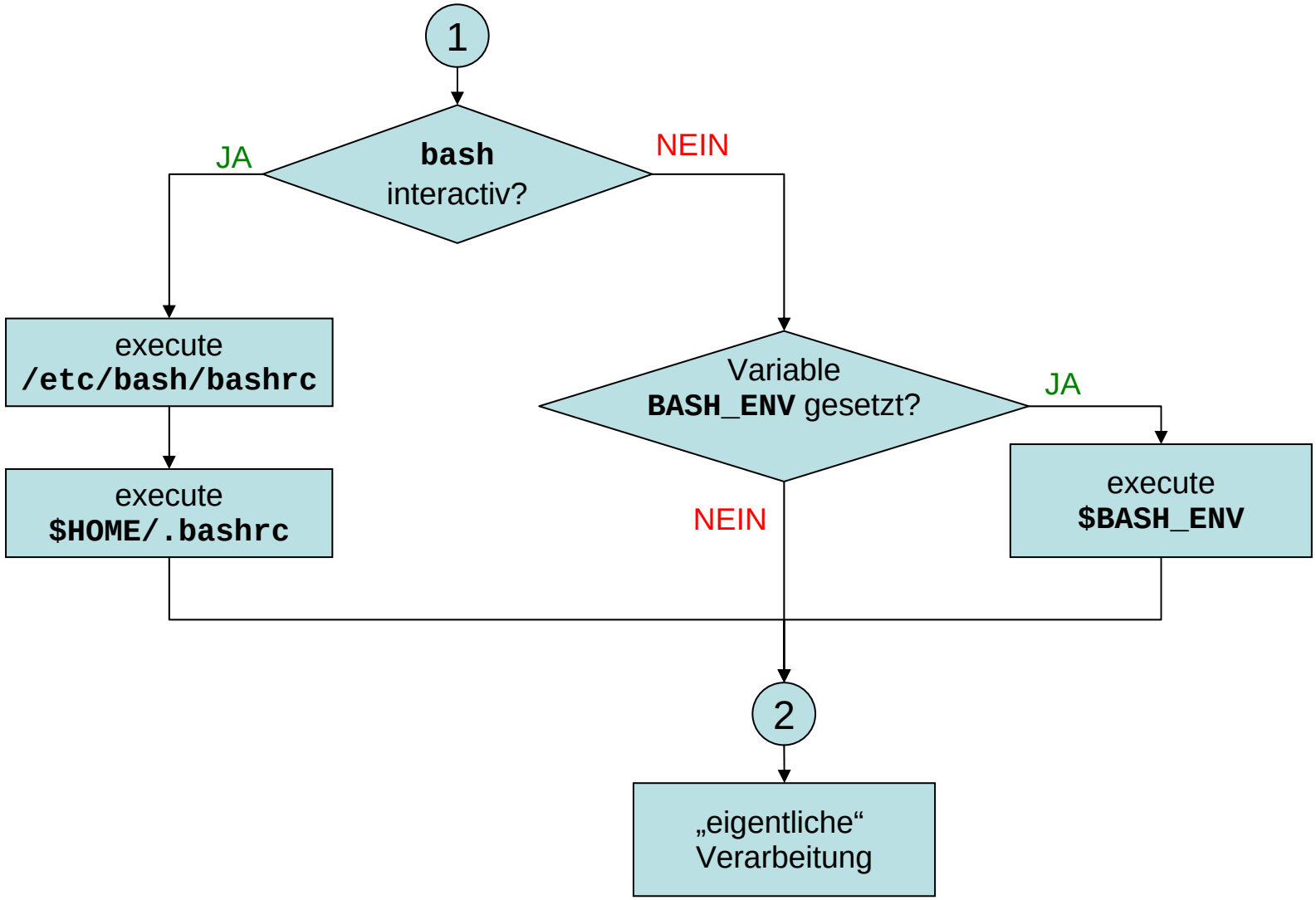
  `<5>$ ksh`

  und dann gleich nochmal starten …

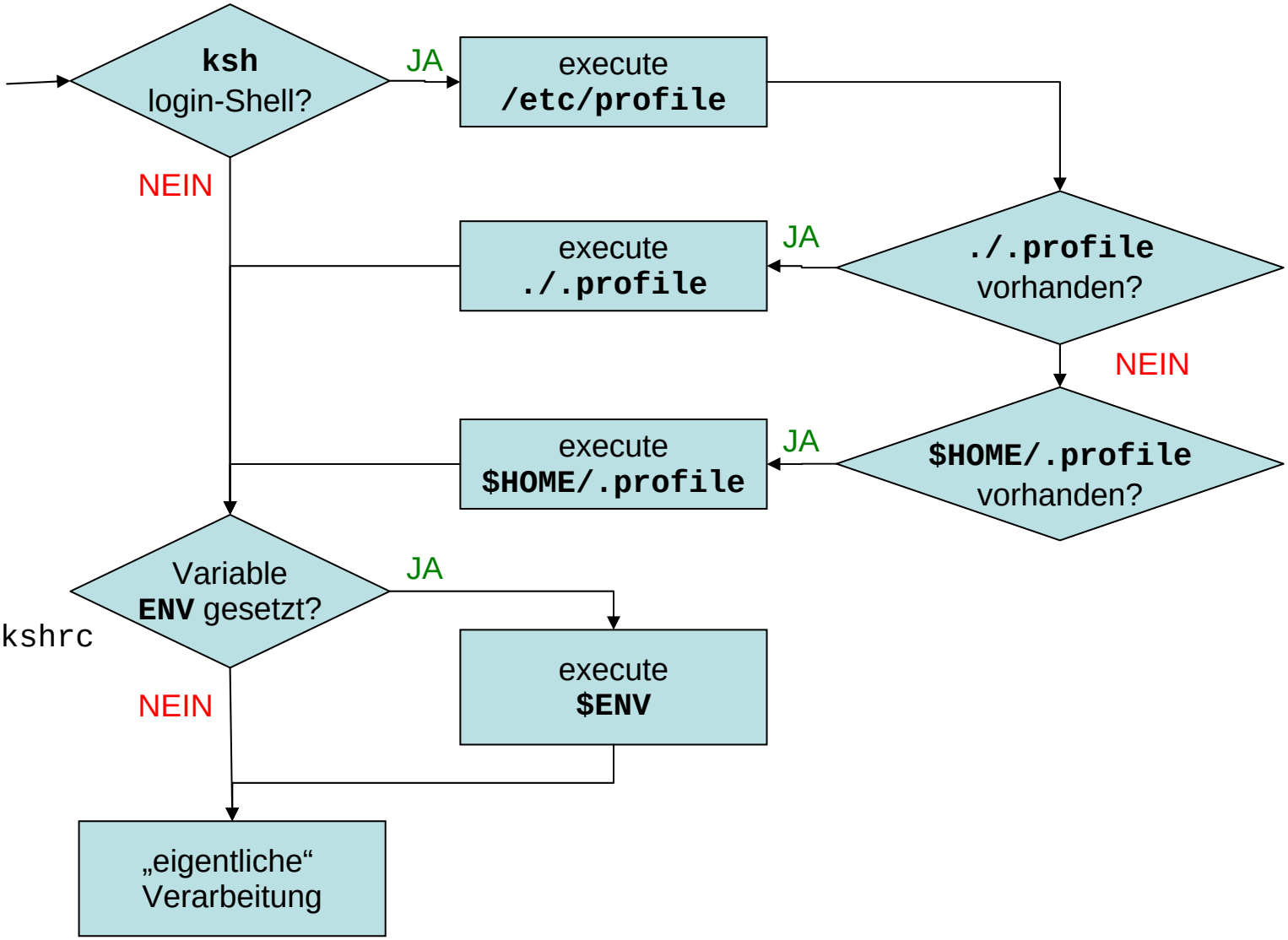
Einstieg: Konfigurieren der Bourne Again Shell (FB IN)



Einstieg: Konfigurieren der Bourne Again Shell (FB IN)



Einstieg: Konfigurieren der Korn-Shell (ksh88, ksh93, pdksh)




```
// bei ksh93
// initial:
// ENV=$HOME/.kshrc
```

Kommandozeile: erster Test

- Kommandos (allgemeiner: Kommandoausdrücke) können als
  - **Einfache Kommandos**
    kdo_name argumente

  In der Kommandozeile aufgerufen werden.

  Bsp:

```
cat /etc/passwd

cat /etc/ksh.kshrc

print Ich bin mal gespannt wie das heute so weiter geht

ls .kshrc .mykshrc .profile .bashrc
```

  usw.

Kommandozeile: Erste Fragen

- Bereits bei Anwendung so elementarer Kommandos wie `ls`, `print` oder `cat` stellen sich dem Benutzer gleich mehrere Fragen:
  - ***Ist eine Kommandoeingabe komfortabel möglich?***
    - Editieren der Kommandozeile
    - Zugriff auf vorhergehende Kommandozeilen
  - ***Wie kann ich Kommandos verknüpfen ?***
    - Sequentialisierung
    - Durchreichen von Ergebnissen
    - Komplexere Kontrollstrukturen
  - ***Wie kann ich mehr als eine Datei ansprechen, ohne jeden Namen tippen oder gar kennen zu müssen ?***
    - Reguläre Ausdrücke (der Shell)

Kommandozeile: emacs-Editierfunktionen

- Auch in der Korn-Shell kann die Kommandozeile editiert bzw. auf die vergangenen Kommandos zurückgegriffen werden. Aktivierung mittels:

```
set –o emacs
```

Emacs-Editierbefehl	Bedeutung in Kommandozeile	
CTRL-b	Ein Zeichen nach links	(„backward“)
ESC-b	Ein Wort nach links	(„backward“)
CTRL-f	Ein Zeichen nach rechts	(„forward“)
ESC-f	Ein Wort nach rechts	(„forward“)
CTRL-a	Zeilenanfang	(„Anfang“)
CTRL-e	Zeilenende	(„Ende“)
CTRL-d	Löschen des Zeichens an Curserposition	(„delete“)
CTRL-h	= Backspace	
CTRL-k	Löschen aktuelle Position bis Zeilenende	

- Alternativ können auch vi-Editierfunktionen gewählt werden:

```
set –o vi
```

Kommandozeile: History-Scrolling

- Auch in der Korn-Shell kann in der History der Befehle gescrollt werden. Diese Möglichkeit wird im Zusammenhang mit den anderen Editierfunktionen aktiviert.

Emacs-History-Scrolling	**Bedeutung in Kommandozeile**
CTRL-p	Scrolle einen Befehl in der History rückwärts („previous“)
CTRL-n	Scrolle einen Befehl in der History vorwärts („next“)
ESC <	Ältester Befehl aus History

- Die früheren in der Korn-Shell eingegebenen Kommandos werden ggf. in einer Datei abgespeichert (***History-Datei***). Siehe jeweilige man-page.
  Bei **`pdksh`** muß dazu die Environment-Variable **`HISTFILE`** gesetzt werden.
  Während des Arbeitens in der Shell, ist die Datei nur über spezielle Kdos inspizierbar.
  Nach Beendigung der Shell, kann die History-Datei auch anders bearbeitet werden.
  GGf. ist hier eine Vorfilterung notwendig.

- Erneute Ausführung des letzten Kommandos mittels
  `r`
  (alias für **`fc -e -`** (in **`ksh88`**) bzw. **`hist -s`** (in **`ksh93`**))

Kommandozeile: History-Kommando – **f**ixed **c**ommand

(analog: **`hist`**-Kommando in **`ksh93)`**

- Listing-Kommandos

  `fc -l`[**n**][**r**] [[-]*from* [*to*]]

Option	**Bedeutung in Kommandozeile**
`-l`	Auflisten der letzen 16 Einträge der History-Datei **`$HISTFILE`**;
`n`	Ohne Nummerierung der Kommandozeilen
`r`	Auflistung rückwärts
from [*to*]	Auflistung ab Kommandozeilennummer *from* [bis Zeile *to*] (*from* bzw. *to* können auch Strings sein)
`-`*from* [*to*]	Auflistung der letzten *from* Kommandozeilen [relativ zur Zeile *to*] (*to* kann auch ein String sein)

Kommandozeile: History-Kommando – **f**ixed **c**ommand

(analog: **`hist`**-Kommando in **`ksh93)`**

- Aufruf früherer einzelner Kommandos ohne Editornutzung

  `fc -s`

  `fc -e -` [*oldstring*=*newstring*] [[-]*pos*]

- Aufruf früherer Kommandos mit Editornutzung

  `fc` [**`-e`** *editor*] [[-]*from* [*to*]]

  der gewünschte Editor kann in der Korn-Shell-Environment-Variablen **`FCEDIT`** (ksh88) abgelegt werden. Default ist `/bin/ed`

Option	**Bedeutung in Kommandozeile**
`-s`	Erneute Ausführung des letzten Kommandos
[-]*pos*	Erneute Ausführung der Kommandozeile Nr. [aktuelle Zeile -] *pos* (*pos* kann auch ein String sein)
oldstring=*newstring*	Ersetze in ausgewählter Kommandozeile *oldstring* durch *newstring*
`-e` *editor*	explizite Angabe des gewünschten Editors
from [to]	Editiere das/die Kommando/s Nummer *from* [bis Nummer *to*] (*from* bzw. *to* können auch Strings sein)
-from [to]	Editiere das/die lezten *from* Kommandozeilen relativ zur aktuellen Zeile [bzw relativ zur Zeile *to*] (*to* kann auch ein String sein)

Kommandozeile: Grundstruktur und einfache Verknüpfung

- Kommandos (allgemeiner: Kommandoausdrücke) können als
  - **Einfache Kommandos (*kdo*)**
    kdo_name argumente

    Bsp: `cat /etc/passwd`
  - als Sequenz unabhängiger Kommandos
    kdo1; kdo2; kdo3; …

    Bsp: `print "Folgende Shells sind festgelegt:" ; cat /etc/passwd;`
  - als Kette per In/Output verknüpfter Kommandos: **Pipeline**
    kdo1 | kdo2 | kdo3 | …

    Bsp: `cat /etc/passwd | grep fuhr`

  in der Kommandozeile abgesetzt werden.

Kommandozeile: Gruppierung von Kommandos

- Ferner können **Kommandos** zu komplexeren Kommandos **gruppiert** werden:

- Variante 1: ***Durchführung*** *der Kommandos* ***in aktueller Shell***
  { *kdo1*; *kdo2*; … ; *kdoN*;**}** bzw. **{** *kdo1* | *kdo2* | … | *kdoN*;**}**

  WICHTIG:
  - *nach* der öffnenden geschweiften Klammer: LEERZEICHEN!
  - Letztes Kommando innerhalb der geschweiften Klammern mit **;** abschliessen!

  Bsp.1 Test auf *aktuelle* Shell:

```
tf-kurs:203$ testvar=1; print outside:$testvar;
outside:1
tf-kurs:204${ print -n "--"; print -n $testvar; print "--";
> testvar=2;     print -n $testvar; print "--";}
1
2
tf-kurs:205$ print outside:$testvar
outside:2
tf-kurs:206$
```

Kommandozeile: Gruppierung von Kommandos

- Variante 2: ***Durchführung*** *der Kommandos* ***in Subshell***
  (*kdo1*; *kdo2*; … ; *kdoN***)** bzw. **(***kdo1* | *kdo2* | … | *kdoN***)**

  Bsp.2 Test auf *Sub*-Shell:

```
tf-kurs:203$ testvar=1; print outside:$testvar;\
> ( print -n "--"; print -n $testvar; print "--";\
>   testvar=2;     print -n $testvar; print "--";);\
> print outside:$testvar
outside:1
1
2
outside:1
tf-kurs:204$
```

- Diese Gruppierungen können beliebig ineinander verschachtelt werden.
  Hinweis:
  Der Output der jeweiligen Kommandos kdo1, …, ist als EIN Output des geklammerten Kommandos zu verstehen! Dies ist bei Hintereinanderschaltung der Kommandos zu beachten.
  Bsp: `>$ { print ksh2; print ksh1; print ksh3; } | sort`

Kommandozeile: Kommandosubstitution

- Ein Mechanismus zur Weitergabe der Ausgaben an ein anderes Kommando haben wir schon kennengelernt
  - die Verknüpfung mittels einer Pipe

  hier wurde der Output eines Kommando als Input dem darauf folgenden Kommando zur Verfügung gestellt

- Ein anderer Mechanismus ist durch die sog. **Kommandosubstitution** gegeben.

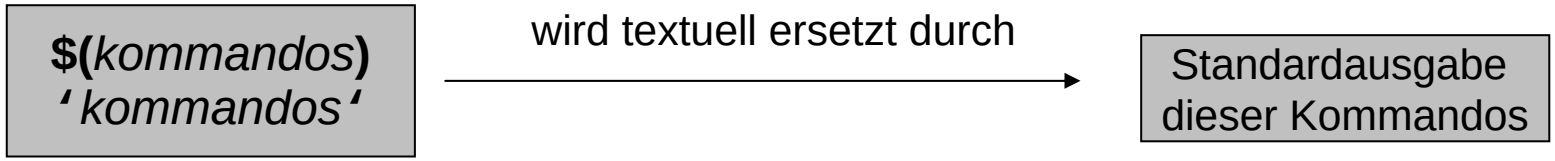
  Hier wird der Output eines Kommandos einem Kommando als Argumentliste zur Verfügung gestellt.

  Syntax:

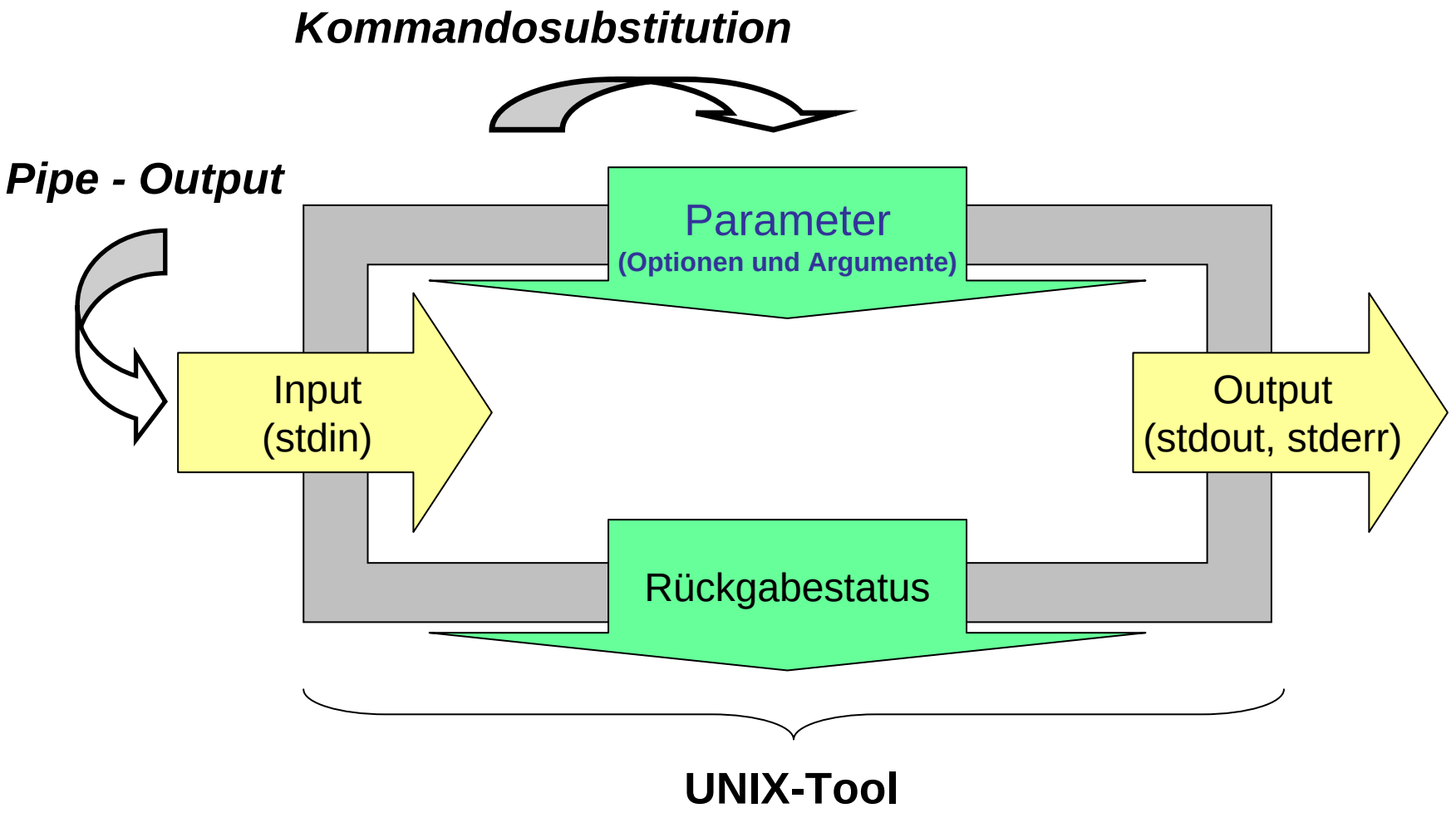
  `$(`*`kommandos`***`)`** bzw. **`` `kommandos` ``** (veraltet)

  Bsp: **`cat $(ls)`** bzw. **`` cat `ls` ``**

- Wirkungsweise:



Kommandosubstitution: Bereitstellung von Parametern



Reguläre Ausdrücke (ksh): einfache „pattern“

- Dateinamen in der Kommandozeile müssen nicht immer explizit ausgeschrieben werden, sondern können kompakt unter Verwendung regulärer Ausdrücke formuliert werden

- Diese können in der Korn-Shell – wie bereits in der Bourne-Shell -- durch Verwendung folgender Metazeichen gebildet werden:

Metazeichen	**Bedeutung**
*****	„eine beliebige Zeichenfolge“
?	„ein beliebiges einzelnes Zeichen“
[…]	„genau eines der in […] angegebenen Zeichen“ (***Zeichenklasse***) Bem: Bereichsangaben erlaubt. Z.B. [A-Z] oder [3-7] oder POSIX-Zeichenklassen
[!...]	„genau ein beliebiges Zeichen, welches nicht in […] vorkommt“ Bem: Bereichsangaben erlaubt. Z.B. [!G-M] oder [!1-6] od. POSIX-Zeichenklassen

Nach: Herold, S.288

- Die mittels dieser Zeichen bildbaren regulären Ausdrücke können miteinander *verkettet* (konkateniert) werden

Reguläre Ausdrücke (ksh): Beispiele

- Alle Dateien, deren Name mit `dat` beginnt:
  ```
  dat*
  ```
- Alle Dateien, deren Name aus zwei beliebigen Zeichen gefolgt von `dat` besteht:
  ```
  ??dat
  ```
- Alle Dateien, deren Name mit `dat` oder `Dat` beginnt:
  ```
  [dD]at*
  ```
- Alle Dateien, deren Name großbuchstabig oder mit kleinem ‘a‘ beginnt:
  ```
  [A-Za]*
  ```
- Alle Dateien, deren Name mit keinem Großbuchstaben beginnt:
  ```
  [!A-Z]*
  ```
- Alle Dateien, deren Name mindestens einen Großbuchstaben beinhaltet und der auf `.cc` endet:
  ```
  *[A-Z]*.cc
  ```

Reguläre Ausdrücke (ksh93): POSIX Zeichenklassen

- Insbesondere um unabhängig bzgl. sprachbezogener Zeichensätze zu sein, ist die Verwendung der sog. POSIX Zeichenklassen nützlich uns sinnvoll.

Metazeichen	**Bedeutung**
`[:alnum:]`	Alphanumerische Zeichen
`[:alpha:]`	Alphabetische Zeichen
`[:blank:]`	Space und Tab Character
`[:cntrl:]`	Control Characterc
`[:digit:]`	Numerische Zeichen
`[:graph:]`	Druckbare/Sichtbare (non-space) Zeichen
`[:lower:]`	Kleinbuchstaben
`[:print:]`	Druckbare Zeichen (inkl. white space)
`[:punct:]`	Punctuation characters
`[:space:]`	White space characters
`[:upper:]`	Großbuchstaben
`[:xdigit:]`	Hexadezimalzeichen

- BEACHTE: Wird von **pdksh** aktuell *nicht* unterstützt!

Reguläre Ausdrücke (ksh93): Beispiele POSIX-Zeichenkl.

- Alle Dateien, deren Name großbuchstabig oder mit kleinem ‘a‘ beginnt:
  ```
  [[:upper:]a]*
  ```
- Alle Dateien, deren Name mit keinem Kleinbuchstaben oder einer Ziffer beginnt:
  ```
  [![:lower:][:digit:]]*
  ```
- Alle Dateien, deren Name mindestens eineZiffer beinhaltet und der auf `.cc` endet:
  ```
  *[[:digit:]]*.cc
  ```

Reguläre Ausdrücke (ksh): Dateinamenexpansion

- Die `ksh` ersetzt jeden in einer Kommandozeile gefundenen regulären Ausdruck durch sämtliche Dateinamen des aktuellen Verzeichnisses, die hierdurch beschrieben werden.

  Werden keine passenden Dateinamen gefunden, *so wird der Pattern als Argument durchgereicht.*

- Diesen Mechanismus nennt man ***Dateinamenexpansion***

- Die Dateinamenexpansion findet *vor* der Kommandoausführung statt!!

Kommandozeile: Dateinamensubstitution

```
>$ touch gerhard hannelore jutta gerd geraldine
>$ ls
 geraldine  gerd  gerhard  hannelore  jutta
>$ rm gerd h* ger*
rm: Entfernen von »gerd« nicht möglich: Datei oder Verzeichnis nicht
   gefunden
>$ ls
  jutta
```

Warum entsteht diese Fehlermeldung??

```
>$ touch gerhard hannelore jutta gerd geraldine

>$ print ger*                # Es kann durch Dateinamen ersetzt werden
geraldine gerd gerhard

>$ print ger*u       # keine passenden Dateinamen - keine Ersetzung!
ger*u
```

Reguläre Ausdrücke (ksh): *pattern*

- Statt von regulären Ausdrücken wird bei der Beschreibung der Korn-Shell auch oft von sog. ***pattern*** gesprochen.
- VORSICHT:
  Nicht nur die UNIX-Shells verfügen über – jeweils eigene – syntaktische und semantische Definitionen von *pattern* (regulären Ausdrücken), sondern auch andere stringverarbeitende UNIX-Tools!
  Der Kontext, in welchem ein regulärer Ausdruck verwendet wird ist also unbedingt zu beachten!
- `/` und `.`
  In regulären Ausdrücken der Korn-Shell sind der Punkt und der Schrägstrich *kein* Metazeichen!
  Trotzdem werden sie eingeschränkter behandelt als andere Nicht-Meta-Zeichen.
  Dies resultiert aus der typischen Konstruktion von Dateinamen unter UNIX:

  ```
  datsubst1/hannelore        .cshrc          /home/fuhr/.kshrc
  ```

  Schrägstriche müssen in *pattern* explizit angegeben werden.
  Punkte *zu Beginn* eines Dateinamens oder nach einem Schrägstrich, müssen explizit in *pattern* angegeben werden.
  Innerhalb von Dateinamen, werden Punkte durch * und ? erfasst.
- **Korn-Shell-*pattern*** können nicht nur zur kompakten Beschreibung von Dateinamen verwendet, sondern auch für Stringvergleiche genutzt werden (siehe später)

Reguläre Ausdrücke (ksh): komplexe *pattern* (ksh88)

- Die Korn-Shell erlaubt die Bildung deutlich komplexerer *pattern* als z.B. die Bourne-Shell

- Einfache *pattern*, können zu komplexeren *pattern* wie folgt zusammengebaut werden – **Besonderheit ksh**

Ausdruck	**Ersetzungsergebnis**
pattern	Einfacher *pattern* (unter Verwendung der Metazeichen **?,*,[,],!**) oder komplexer *pattern* gemäß der folgenden 5 Zeilen
(pattern***[|***pattern***]…)**	Beliebig häufiges Vorkommen eines der *pattern*
+(*pattern***[|***pattern***]…)**	Mindestens einmaliges Vorkommen eines der *pattern*
@(*pattern***[|***pattern***]…)**	Genau einmaliges Vorkommen eines der *pattern*
?(*pattern***[|***pattern***]…)**	Maximal einmaliges Vorkommen eines der *pattern*
!(*pattern***[|***pattern***]…)**	Kein Vorkommen eines der *pattern*

Nach: Herold, S.288

Reguläre Ausdrücke (ksh): Beispiele komplexer *pattern* (ksh88)

```
>$ ls
   abc   abc1a   abcdefabc     abcdefzzz     abczzz     babc  def
   abc1  abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg  cba

>$ ls abc*(def|ggg|zzz)

>$ ls abc+(def)

>$ ls abc*+(def)

>$ ls abc*+(def)*

>$ ls abc@(def|ggg|zzz)

>$ ls abc?(def|ggg|zzz)
```

Reguläre Ausdrücke (ksh): Beispiele komplexer *pattern* (ksh88)

```
>$ ls
    abc   abc1a   abcdefabc     abcdefzzz     abczzz     babc  def
    abc1  abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg  cba

>$ ls abc*(def|ggg|zzz)
    abc                       abcdefzzz     abczzz
          abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg

>$ ls abc+(def)

          abcdef  abcdefdefdef

>$ ls abc*+(def)

          abcdef  abcdefdefdef  abcdefzzzdef

>$ ls abc*+(def)*
                    abcdefabc     abcdefzzz
          abcdef  abcdefdefdef  abcdefzzzdef

>$ ls abc@(def|ggg|zzz)
                                            abczzz
          abcdef

>$ ls abc?(def|ggg|zzz)
    abc                                     abczzz
          abcdef
```

Reguläre Ausdrücke (ksh): Beispiele komplexer *pattern* (ksh88)

```
>$ ls

>$ ls *(abc|def|ggg|zzz)

>$ ls !(*(abc|def|ggg|zzz))

>$ ls @(abc|def|ggg|zzz)

>$ ls !(@(abc|def|ggg|zzz))

>$ ls *+(def)*

>$ ls !(*+(def)*)
```

Vgl. mit: **`ls *!(def)*`** ??? Unterschied???

Reguläre Ausdrücke (ksh): Beispiele komplexer *pattern* (ksh88)

```
>$ ls
    abc   abc1a   abcdefabc     abcdefzzz     abczzz     babc  def
    abc1  abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg  cba

>$ ls *(abc|def|ggg|zzz)
    abc             abcdefabc     abcdefzzz     abczzz          def
          abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg

>$ ls !(*(abc|def|ggg|zzz))
          abc1a                                          babc
    abc1                                                 cba

>$ ls @(abc|def|ggg|zzz)
    abc                                                      def

>$ ls !(@(abc|def|ggg|zzz))
          abc1a   abcdefabc     abcdefzzz     abczzz     babc
    abc1  abcdef  abcdefdefdef  abcdefzzzdef  abczzzggg  cba

>$ ls *+(def)*
                  abcdefabc     abcdefzzz                    def
          abcdef  abcdefdefdef  abcdefzzzdef

>$ ls !(*+(def)*)
    abc   abc1a                               abczzz     babc
    abc1                                      abczzzggg  cba
```

Vgl. mit: **`ls *!(def)*`** ??? Unterschied???

Reguläre Ausdrücke (ksh): Tücken

- Bei der Verwendung der Metazeichen ist ist große Sorgfalt wichtig, da ihre Bedeutung und Wirkung vom Kontext im pattern abhängt:

- Negation:

```
+([!0-9])   vs.   !(+([0-9]))
@([!0-9])   vs.   !(@([0-9]))
```

- Asterisk:

```
*([abc])    vs.   [abc]*        vs.   [abc*]
```

- Fragezeichen

```
?([abc])    vs.   [?abc]
```

- Plus-Symbol

```
+([abc])    vs.   [+abc]
```

u.a.

Reguläre Ausdrücke (ksh): Tücken

- Bei der Verwendung der Metazeichen ist große Sorgfalt wichtig, da ihre Bedeutung und Wirkung vom Kontext im pattern abhängt:

- Negation:

`+([!0-9])` vs. `!(+([0-9]))`

Alle (nicht leeren) Zeichenketten, welche **nur aus nicht numerischen** Zeichenketten bestehen, z.B: Ccab, ab%zu, PÜ?§_

Alle Zeichenketten, welche **nicht ausschließlich aus numerischen** Zeichen bestehen, z.B: Cca1b, Ccab, ab%zu (Obermenge des linken Falls)

`[!0-9]` vs. `!([0-9])`

Alle Zeichenketten, welche aus **genau einem nicht numerischen** Zeichen bestehen, z.B: A, b, c, ….%, &, ..

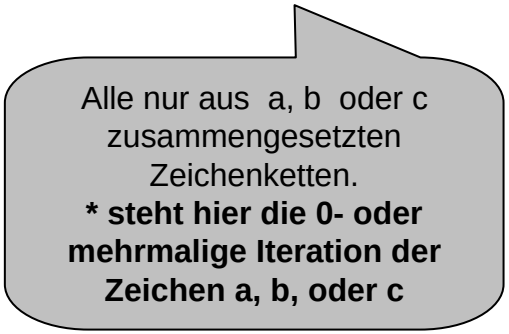
Aber NICHT erlaubt: Aa, a93, 9, …

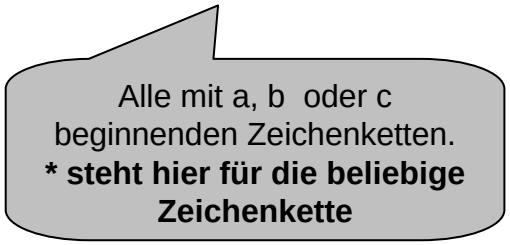
Alle Zeichenketten, welche **nicht aus genau einem numerischen** Zeichen bestehen, z.B.: Aa, a93, 45, %&, …

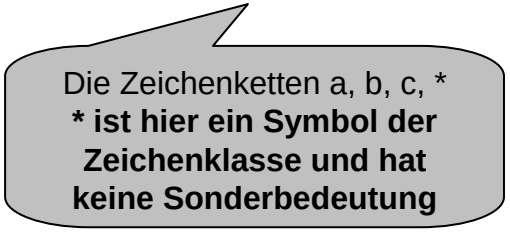
Aber NICHT erlaubt:0, 1, 2, 3, 4, 5, 6, 7, 8, und 9

Reguläre Ausdrücke (ksh): Tücken

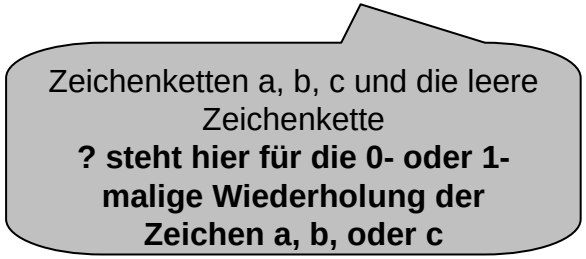
- Asterisk: `*([abc])` vs. `[abc]*` vs. `[abc*]`

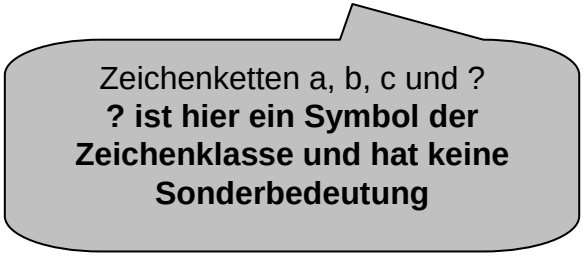




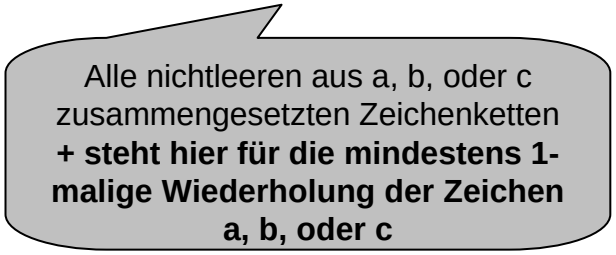


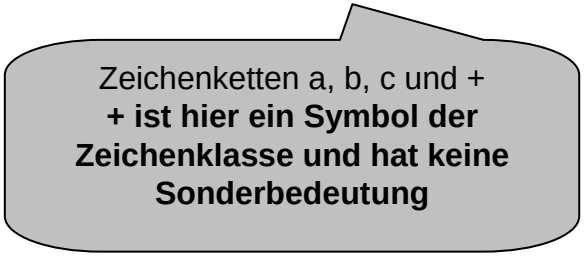
- Fragezeichen: `?([abc])` vs. `[?abc]`





- Plus-Symbol: `+([abc])` vs. `[+abc]`





Reguläre Ausdrücke (ksh): einfache vs. komplexe pattern

- ?, @, + und * können als Iteratoren für die in der folgenden Klammer beschriebene Zeichenkette gelesen werden!!
- Stehen dort ODER-verknüpfte Zeichenketten, so können diese gemäß des Iterators oft in beliebiger Kombination konkateniert werden.
- Damit erlauben diese Operatoren erheblich genauer die Wiederholung von Zeichenketten zu beschreiben, als bei den einfachen Ausdrücken möglich.
- Insbesondere ist mit der Syntax einfacher Pattern *nicht* beschreibbar:
  - Begrenzung auf
    - optionales Vorkommen einer Zeichenkette
    - genau ein (@) Vorkommen einer *bestimmten* Zeichenkette *innerhalb einer aus weiteren* Zeichen bestehenden Zeichenkette *unbestimmter* Länge
    - Komplemente bzgl. ganzer Zeichenketten
- Lediglich mindestens ein Vorkommen (+) kann mit der Syntax der einfacheren Ausdrücke abgebildet werden.
- Insbesondere können nun wg. der Schachtelungsmöglichkeit komplexer Pattern Iterationen über mittels komplexer Pattern charakterisierter Zeichenketten durchgeführt werden

Reguläre Ausdrücke (ksh): Grenzen

- Die regulären Ausdrücke der Korn-Shell sind deutlich ausdrucksstärker als die der Bourne Shell.
- Trotzdem gibt es auch Beschränkungen der Ausdruckskraft:

  Z.B.:
  - Die Zahl der möglichen Wiederholungen einer Zeichenkette kann nicht explizit spezifiziert werden:

    „alle Zeichenketten, in denen def 5-mal vorkommt“

    „alle Zeichenketten, in denen def mindestens 5-mal vorkommt“

    „alle Zeichenketten, in denen def mindestens 5-mal und maximal 7-mal vorkommt“

  - Es gibt nur eine Oder-Verknüpfung von Pattern

  - Referenzierung von (Teil)pattern

    „alle Zeichenketten, in denen eine 3-Zeichenlange Zeichenkette zweimal auftritt“

Ersetzungsmechanismen: Reihenfolge

- Die Reihenfolge der bisher kennengelernten Ersetzungen ist wie folgt:

Kommandosubstitution

Existiert ein String der Form `*kommandos*` oder $(*kommandos*) in der Kommandozeile, so wird dieser durch die Standardausgabe von *kommandos* ersetzt

Dateinamenexpansion

Zuletzt werden vorhandene pattern aufgelöst, durch Einsetzen aller passenden Dateinamen.

Bsp.: `ls *def*` vs. `ls $(print *def)*`

`ls $(print ‘?b?‘ *def ‘def*‘ )`

in Verzeichnis `datsubst2`

Ersetzungsmechanismen: Quoting-Regeln

- Durch geeignetes Klammern, können Ersetzungen/Substitutionen verhindert werden
  Man nennt dieses auch **Quoting**.

Quotingsymbol	**Bedeutung**
'…'	Mittels einfacher Anführungszeichen, werden die Spezialbedeutungen *aller* in der Klammerung stehenden Zeichen unterbunden.
"…"	Die doppelten Anführungszeichen heben die Spezialbedeutung aller Zeichen *außer* von $, `, und \ auf. *Bem1:* Innerhalb von doppelten Anführungszeichen findet also Parameter- und Kommandosubstitution, *nicht aber* die Dateinamenexpansion statt
\c	Unterbindet die Spezialbedeutung von *c*, sofern *c* eine besitzt.

Ersetzungsmechanismen: Quoting-Regeln

(Resultate beziehen sich auf Verzeichnis `verz_ksh_pattern_1.1`)

```
$ ls $(print '*def')*
abcdef
abcdefabc
abcdefdefdef
abcdefzzz
abcdefzzzdef
def
```

Es findet zuerst Kommandosubstitution statt. Anschliessend Dateinamensubstitution für *def*

```
$ touch *def*
```

alle Dateien mit Namensteil `def` erhalten das aktuelle Datum

```
$ touch '*def*'
```

Es wird eine Datei namens `*def*` erzeugt.

```
$ print “'ls *def'“
'ls *def'
```

Es findet keine Dateinamensubstitution statt und die einfachen Anführungszeichen werden als auszugebendes Zeichen angesehen.

Ersetzungsmechanismen: Quoting-Regeln

(Resultate beziehen sich auf Verzeichnis `verz_ksh_pattern_1.1`)

```
$ print “`ls *def`“
abcdef
abcdefdefdef
abcdefzzzdef
def
```

Hier findet die Kommandosubstitution statt: es werden deshalb alle Dateien ausgegeben, deren Namen auf `def` endet.

```
$ print \$\'*def\'
$'*def'
```

alle Sonderzeichen sind explizit einzeln qequotet

Dateinamensubstitution kann nichts finden, da kein Dateiname mit `$'` beginnt und `def'` endet

```
$ touch '*def*' '*'
$ print $(ls)
```

Erklärung?

Korn-Shell-Debugging

- Die `ksh` verfügt über einige Optionen, welche genutzt werden können um Fehler in Anweisungen bzw. Skripten zu finden.

Option	**Bedeutung**
`-n`	Nur Durchführung einer Syntaxprüfung
`-u`	Abbruch und Fehlermeldung bei Zugriff auf nicht gesetzte Variable
`-v`	Ausgabe aller Shell-Eingabezeilen (uninterpretiert!)
`-x`	Jedes Kommando unmittelbar vor seiner Ausführung ausgeben *(damit Resultat der Parameter- und Kommandosubst. sowie der Dateinamenexpansion sichtbar!)*

- Einschalten der Optionen in Korn-Shell mittels des **`set`-Kommandos**:

```
set –x        set –vu
```

- Ausschalten der Optionen mittels des **`unset`-Kommandos**:

```
unset –x     unset –vu        oder
set +x       set +vu
```

Nützliche UNIX-Tools (Teil 1)

- Auflistung von Dateinamen
  - `ls`
  - `find`

- Ausgabe von Zeilen genannter Dateien
  - `cat` (`od`)
  - `head`
  - `tail`
  - `uniq`
  - `sort`

- Ausgabe von manipulierten Zeileninhalten genannter Dateien
  - `cut`
  - `tr`

- Es werden nur ausgewählte Nutzungen beschrieben!
  Für weitere Information verwende das **`man`**- oder das **`info`**-Kommando.

find

- `find` dient dem Suchen von Dateien und Verzeichnissen und erlaubt gleichzeitig die Ausführung von Kommandos auf diesen

- Verwendung zur Erzeugung von Dateilisten unter Verwendung von Namensmustern

- Aufruf:
  `find` [*paths*] [*options*] [*search_criteria*] [*actions*]

- Bsp:
  `find /etc –name “*.txt“ –print`
  Suche alle Dateien im Verzeichnis /etc und dessen Unterverzeichnisses, deren Namen auf .txt enden und gib deren Namen auf der Standardausgabe aus

  `find ~fuhr –name “.*“ -print`
  Suche alle Dateien im Verzeichnis ~fuhr und dessen Unterverzeichnisses, deren Namen mit . beginnt und gib deren Namen auf der Standardausgabe aus

find

- Verwendung zur Erzeugung von Dateilisten unter Verwendung von Namensmustern mit *anschließender Anwendung eines Kommandos* auf diese Dateien

- Bsp:

```
find /etc –name “*.txt“ –exec cat {} \;
```

Suche alle Dateien im Verzeichnis /etc und dessen Unterverzeichnisses, deren Namen auf .txt enden und gib deren Inhalt auf der Standardausgabe aus

```
find ~fuhr –name “.*“ -exec ls –l {} \;
```

Suche alle Dateien im Verzeichnis ~fuhr und dessen Unterverzeichnisses, deren Namen mit . beginnt und gib deren Dateieigenschaften auf der Standardausgabe aus

- Weitere Beschreibung: siehe Manual Pages

cat

Usage: **cat** [options] [file ...]

OPTIONS

-b	Number lines as with -n but omit line numbers from blank lines.
-n	Causes a line number to be inserted at the beginning of each line.
-T	shows TABS explicitly

od

```
Usage: od [ options ] [file ...]

OPTIONS
  -c            select ASCII characters or backslash escapes
```

head

```
Usage: head [OPTION]... [FILE]...

Print first 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file
   name.
With no FILE, or when FILE is -, read standard input.

  -cN                print first N bytes
  -nN                print first N lines
  -q                 never print headers giving file names
  -v                 always print headers giving file names
  --help             display this help and exit
  --version          output version information and exit
```

tail

```
Usage: tail [OPTION]... [FILE]...

Print the last 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file
   name.
With no FILE, or when FILE is -, read standard input.

  -cN          output the last N bytes
  -c+N         output beginning with Byte number N
  -f           output appended data as the file grows
  -nN          output the last N lines, instead of the last 10
  -n+N         output beginning with line number N
  -q           never output headers giving file names
  -v           always output headers giving file names
  --help       display this help and exit
  --version    output version information and exit
```

uniq

Usage: **uniq** [OPTION]... [INPUT [OUTPUT]]

Discard all but one of *successive* identical lines from INPUT (or standard input), writing to OUTPUT (or standard output).

-c	prefix lines by the number of occurrences
-d	only print duplicate lines
-D	print all duplicate lines
-f N	avoid comparing the first N fields
-i	ignore differences in case when comparing
-s N	avoid comparing the first N characters
-u	only print unique lines
-w N	compare no more than N characters in lines

A field is a run of whitespace, then non-whitespace characters.
Fields are skipped before chars.

sort

- `sort` sortiert den Inhalt von Dateien lexikographisch auf Basis ASCII-Codierung
- Die Zeilen des Inputfiles werden als in Felder strukturiert betrachtet.
  Als Default-Feldgrenze gilt der Übergang von *non-white space* zu *white space*

- Aufruf:
  `sort` [*optionen*] *dateinamen*
- Wichtige (ausgewählte) Optionen:

Option	**Bedeutung**
`c`	Prüfung, ob Inputdateien *dateinamen* bereits sortiert (nein: exit 1)
`f`	Gross- und Kleinschreibung ignorieren
`r`	Rückwärts sortieren
`d`	Alphabetisch sortieren
`n`	Numerisch sortieren
`u`	Mehrfachelemente löschen
`t` *c*	Als Trennsymbol *c* festlegen.
`k` *n1* [,*n2*]	Bei Sortierung alle Felder ab Feld *n1* bis Feld *n2* berücksichtigen. Fehlt *n2*, so alle Felder ab *n1* berücksichtigen. (*n1*, *n2* = **1**)
`o` *outfile*	Ausgabe in die Datei *outfile* (kann gleich Eingabedatei sein)

cut

Usage: **cut** [options] [file ...]

OPTIONS

-b*list*	cut based on a *list* of bytes.
-c*list*	cut based on a *list* of characters.
-d*delim*	The field delimeter for the -f option is set to *delim*. The default is the tab character.
-f*list*	cut based on a list of fields separated by delimiter character currently specified
-s	Suppress lines with no delimiter characters, when used with the -f option. By default, lines with no delimiters will be passsed in untouched.
-D*ldelim*	The line delimiter character for the -f option is set to *ldelim*. The default is the newline character.

list = *range1*,*range2*,*range3*,…
range = *num1*-*num2* oder *num1*- oder -*num2*

tr

```
Usage: tr [OPTION]... SET1 [SET2]

Translate, squeeze, and/or delete characters from standard input,
writing to standard output.

  -c          first complement SET1
  -d          delete characters in SET1, do not translate
  -s          replace sequence of characters with one
  -t          first truncate SET1 to length of SET2

Translation occurs if -d is not given and both SET1 and SET2 appear.
-t may be used only when translating.  SET2 is extended to length of
SET1 by repeating its last character as necessary.  Excess characters
of SET2 are ignored.  Only [:lower:] and [:upper:] are guaranteed to
expand in ascending order; used in SET2 while translating, they may
only be used in pairs to specify case conversion.
-s uses SET1 if not translating nor deleting; else squeezing uses
   SET2 and occurs after translation or deletion.
```

tr

SETs are specified as sequences of characters (CHAR). Most represent themselves.

Interpreted sequences are:

CHAR1-CHAR2	all characters from CHAR1 to CHAR2 in ascending order
[CHAR*]	in SET2, copies of CHAR until length of SET1
[CHAR*REPEAT]	REPEAT copies of CHAR
[:alnum:]	all letters and digits
[:alpha:]	all letters
[:blank:]	all horizontal whitespace
[:cntrl:]	all control characters
[:digit:]	all digits
[:lower:]	all lower case letters
[:punct:]	all punctuation characters
[:space:]	all horizontal or vertical whitespace
[:upper:]	all upper case letters

tr

Further interpreted sequences are: (special characters)

\NNN	character with octal value NNN (1 to 3 octal digits)
\\	backslash
\a	audible BEL
\b	backspace
\f	form feed
\n	new line
\r	return
\t	horizontal tab
\v	vertical tab

tr

Lowercase to Uppercase

```
tr a-z A-Z < tr_testinput.1
tr [:lower:] [:upper:] < tr_testinput.1
```

A -> B

```
tr A B < tr_testinput.1
```

Shiftcodierung

```
tr a-z c-zab < tr_testinput.1
```

Alles was nicht Grossbuchstabe ist durch = ersetzen

```
tr -c [:upper:] = < tr_testinput.1
```

Alles was weder Grossbuchstabe noch Newline ist durch = ersetzen

```
tr -c [:upper:]\n = < tr_testinput.1
```

Uups

```
tr -c [:upper:]"\n" =: < tr_testinput.1
```

Entstehen beim Ersetzen der a's Ketten von b's, so b-Ketten auf ein b „zusammenschieben“.

```
tr -s a b < tr_testinput.1
```

tr

Ersetzung von Zeichen durch Newline um Umbrüche zu generieren, kann bei Blankfolgen zu vielen Leerzeilen führen. Das kann mittels der squeeze-Option unterbunden werden

```
tr -s A-Z '\n' < tr_testinput.1
```

Alles was weder Großbuchstabe noch Newline ist durch = ersetzen; =-Ketten stauchen

```
tr -cs A-Z“\n" = < tr_testinput.1
```

alle a's löschen

```
tr -d a < tr_testinput.1
```

alles, was nicht a ist löschen

```
tr -cd a < tr_testinput.1
```

Nützliche UNIX-Tools (Teil 1): unterschiedlicher Feldbegriff

- Die vorgestellten Tools partitionieren den Input standardmäßig sehr unterschiedlich
  - **`uniq, sort`**: Feld=führende sequenz white space + Sequenz non-white space

```
hans    gerda   hannelore       emil    thomas
feld1                   feld2   feld3           feld4
feld1 feld2 feld3               feld4
```

  - **`cut`**: Feld=Sequenz abgegrenzt durch TAB

```
hans    gerda   hannelore       emil    thomas
feld1                   feld2   feld3           feld4
feld1 feld2 feld3               feld4
```

leeres Feld

Nützliche UNIX-Tools (Teil 1): Grenzen

- Mit den vorgestellten Tools
  - `cat`
  - `head` - `tail`
  - `uniq` - `sort`
  - `cut` - `tr`

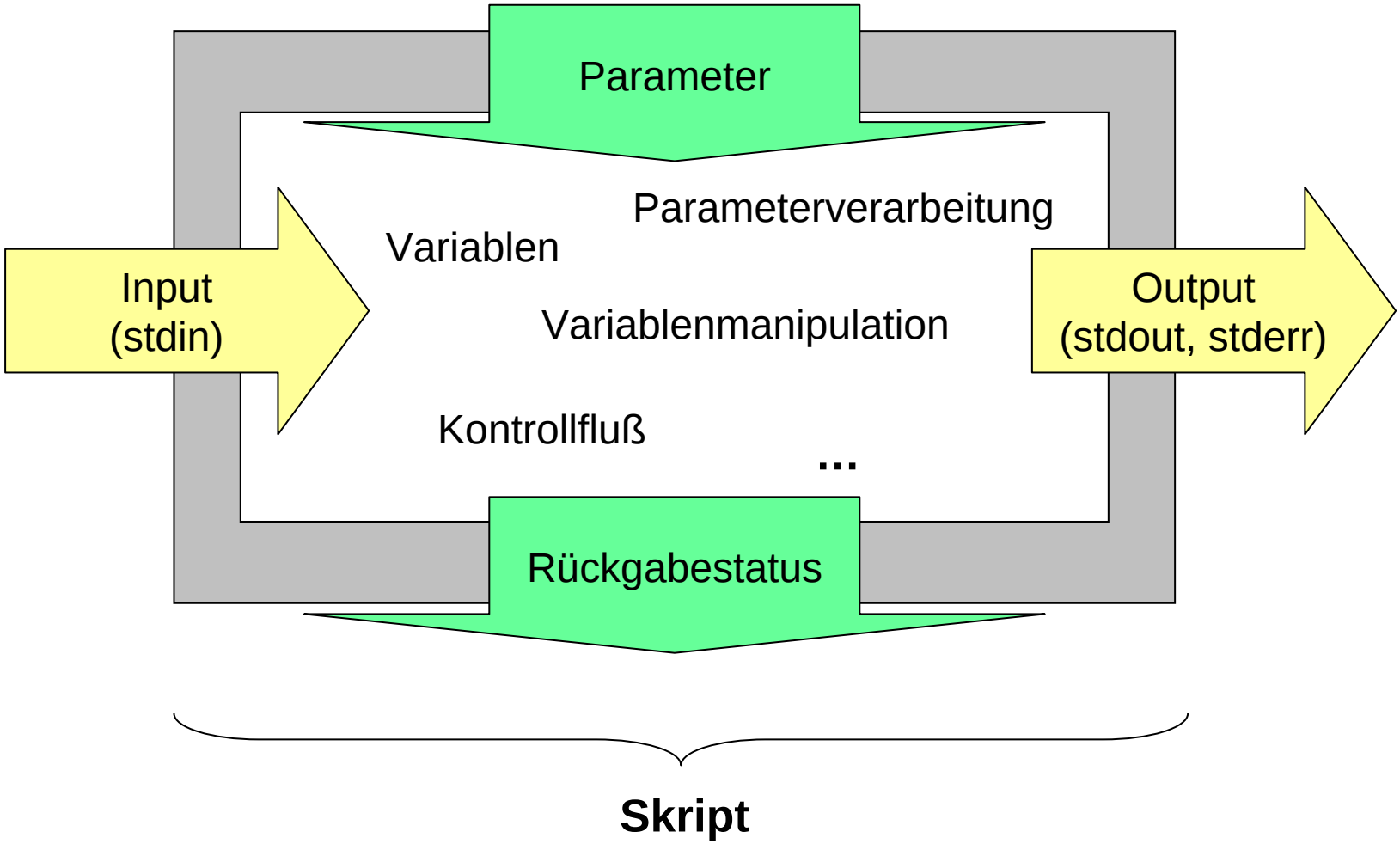
  können

  - Zeilen *nicht inhaltsbezogen selektiert* werden, in dem Sinne, daß alle Zeilen mit bestimmtem Inhalt oder Inhaltsmuster ausgegeben werden.

    „gib alle Zeilen aus, welche mit ‚41' beginnen"
    „gib alle Zeilen aus, in welchen ein Wort zweimal aufeinander folgt"

  - *keine kontextabhängigen Manipulationen* gemacht werden

    „ersetze ‚<H5> … <\H5>' durch ‚<H4> … <\H4>' "
    „ersetze ‚<H5> … <\H5>' durch ‚ … ' "


Parameter
Parameterverarbeitung
Variablen
Input
(stdin)
Variablenmanipulation
Output
(stdout, stderr)
Kontrollfluß
...
Rückgabestatus
Skript

Skripte: Form und Aufrufmechanismen

- Form:
  - Datei, welche syntaktisch den Regeln eines Shellinterpreters genügen muß.

- Aufrufmechanismen (bei denen ksh als Prozeß in Prozeßliste erscheint):

  - `ksh` < *dateiname* // per Eingabeumlenkung

  - `ksh` *dateiname* // als Aufrufparameter

  In all diesen Fällen erscheint in der Prozeßliste des Kommandos `ps` nur ein Prozeß namens `ksh`

Skripte: Form und Aufrufmechanismen

- Aufrufmechanismus, bei denen *dateiname* als Prozeß in Prozeßliste erscheint):
  - *dateiname*

```
// Festlegung des Interpreters in erster Skriptzeile
//          #! Interpreterpfadname
//  Bsp.:   #! /bin/ksh
//  Bsp.:   #! /bin/tcsh

// ACHTUNG: Datei muß ausführbar (executable) sein
```

  - Weiterer Vorteil der Charakterisierung des Shellinterpreters in erster Dateizeile:
    - Dokumentation, welcher Shellsyntax das Skript genügt.

Skript I/O: **`print`**

- Das ksh-builtin-kommando `print` erlaubt formatiert auf die Standardausgabe zu schreiben.

  `print` [**`-n`**] [*argument(e)*]

  `-n` kein Zeilenvorschub nach Ausgabe der Argumente

- In den Argumentstrings können nebenstehende Formatangaben stehen:

  HINWEIS: dies ist nicht die vollständige Beschreibung des Kommandos.
  (→ Manpage oder Literatur)

Formatstring	**Bedeutung**
\a	Akustisches Terminalsignal
\b	Backspace
\c	Wie Option -n
\f	Form feed
\n	Neue Zeile
\r	Carriage Return (ohne newline!)
\t	(horizontaler) Tabulator
\v	Vertikaler Tabulator
****	Backslash
\0*n*	*n* =1, 2, oder 3-zifferige Oktalzahl. Ausgabe des zugehörigen ASCII-Zeichens.

Shell-Variablen: Grundlegendes

- Variablen werden in der Shellprogrammierung i.allg. nicht deklariert
  (wie z.B. aus C oder C++-Programmen bekannt: „char c;“ oder „double value;“

- Die Deklaration erfolgt i.allg. einfach im Rahmen der Initialisierung.

- Die Korn-Shell verfügt jedoch auch über die Möglichkeit der Typdeklaration.
  → siehe später

Korn-Shell-Variablen: Grundlegendes

- **Benutzerdefinierte Variablennamen** müssen folgendem Bezeichner-Schema folgen:

  variablenname ::={_|*Buchstabe*} {_|*Buchstabe*|*Ziffer*}*
  Buchstabe ::={A|B|C|…Z|a|b|c|…|z}\{Ä|Ö|Ü|ä|ö|ü|ß}
  Ziffer ::= {0|1|2|3|4|5|6|7|8|9}

- **Initialisierung** (und damit Deklaration), per einfacher ***Zuweisung***:

  variablenname=string

- **Referenzierung des Variablenwertes** mittels **$:**

  $*variablenname* bzw. **${***variablenname***}**

  steht für den Variablenwert.

  Unterschied: rechte Variante erlaubt Konkatenation des Variablenwerts mit direkt folgendem String, welcher mit Zeichen beginnt, die in Variablennamen verwendet werden dürfen.

  Bsp:

```
>$ var=wasser          >$ var=wasser
>$ print $varhahn      >$ print ${var}hahn
                       wasserhahn
>$                     >$
```

Ersetzungsmechanismen: Parametersubstitution

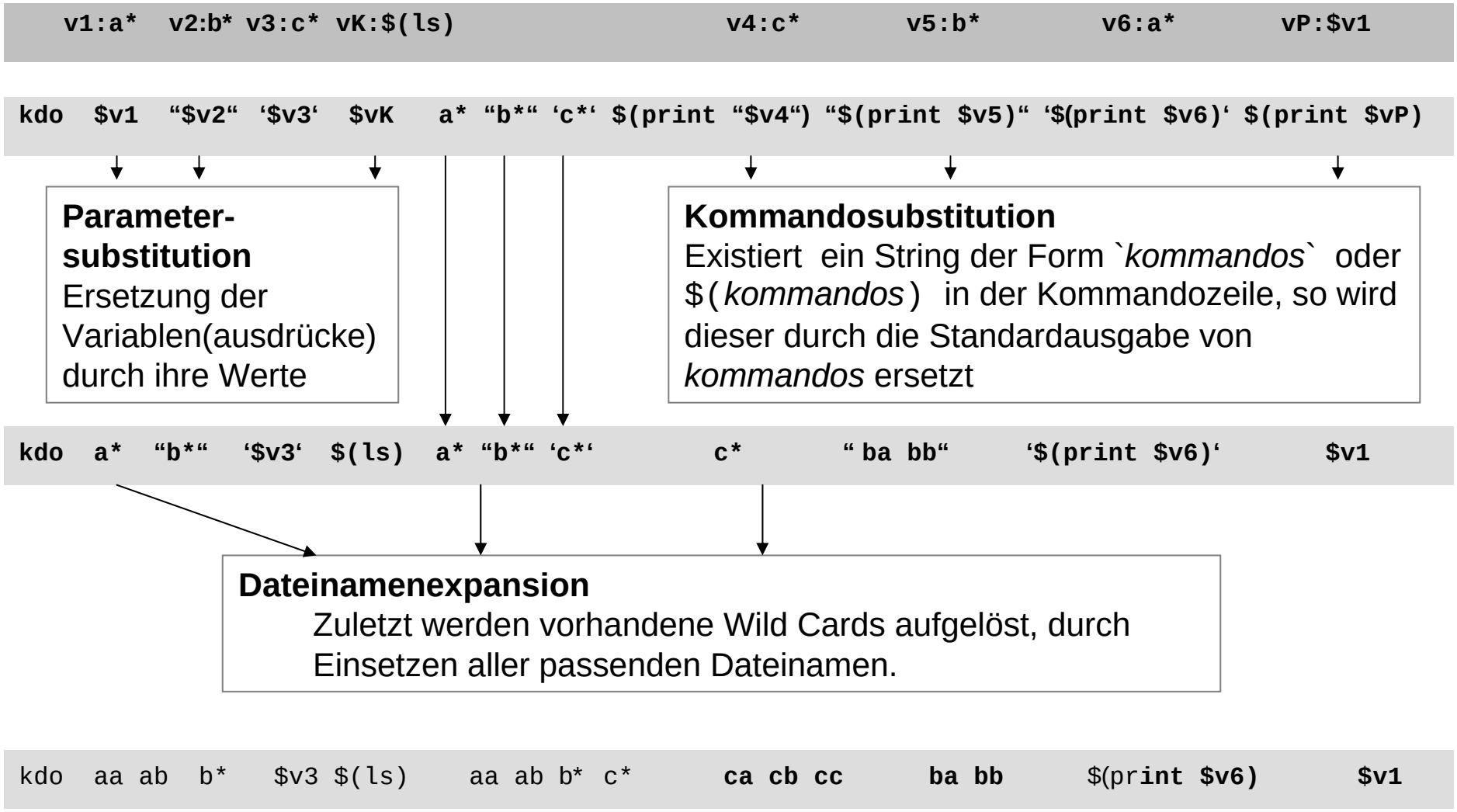
- Tritt eine Variable in einer Kommandozeile auf, so wird sie – durch die Shell -- durch ihren Wert ersetzt. Man nennt dieses auch **Parametersubstitution**.

```
$ myvar=Thomas
$ set –x          # Debugausgabe der ksh einschalten
$ print $myvar    # Kommandozeile vor Bearbeitung durch die Shell
+ print Thomas    # Kommandoaufruf nach Parametersubstitution
Thomas
$ set +x          # Debugausgabe der ksh ausschalten
```

Ersetzungsmechanismen: Quoting-Regeln

```
$ print ls ${myvar4}`print na`*
ls fiona1 fiona2
$
$ print 'ls ${myvar4}`print na`*'      # keine Subst./Expansionen
ls ${myvar4}`print na`*
$
$ print "ls ${myvar4}`print na`*"     # keine Dateinamenexpansion
ls fiona*                             # aber Parameter- u. Kdo-subst.
$
$ print \' ls ${myvar4}`print na`* \'# nur Bedeutung Anf.zeichen aus
' ls fiona1 fiona2 '
$
$ print \'ls ${myvar4}`print na`*\'   # hoppla ?
'ls fiona*'
```

Ersetzungsmechanismen: Reihenfolge

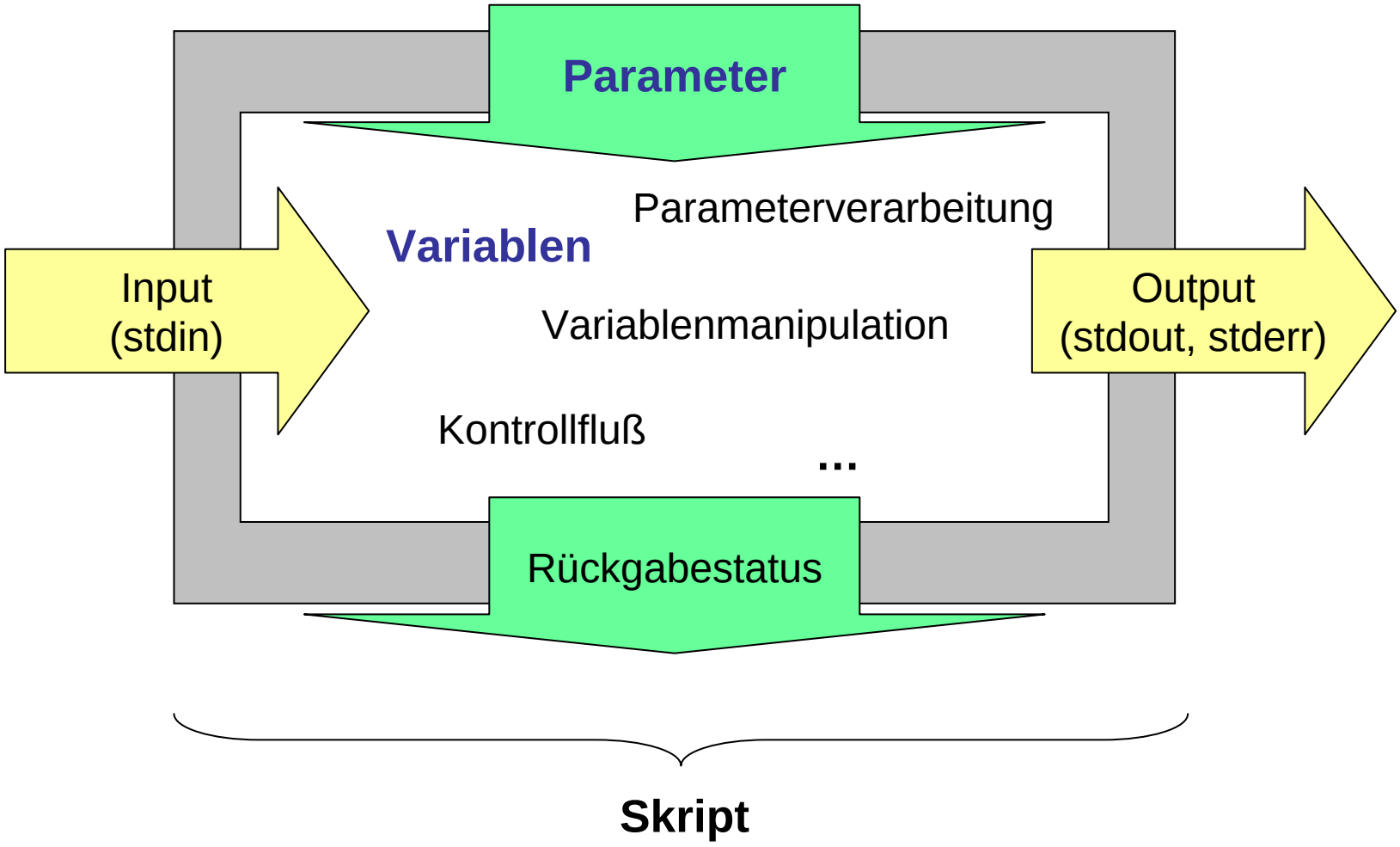


Ersetzungsmechanismen: Reihenfolge

- Links-Nach-Rechtsverarbeitung der Kommandozeile:

```
>$ mkdir neu; cd neu

>$ newvar=s*
>$ print $newvar `print begin; touch showMe; print end` $newvar
s* begin end showMe

>$ rm showMe; cd ..; rmdir neu
```

Wie geht es weiter?



Datenübergabe an ein Skript

- Wie in der C-Programmierung können Daten im Rahmen

  - mittels globaler Variablen bereitgestellt werden. → ***Environmentvariablen***

  oder

  – einer Parameterübergabe explizit in Aufrufzeile → ***Positionsparameter***

    WICHTIG: Es gibt keine implizite Prüfung der Parameterzahl beim Skriptaufruf, d.h. die Kontrolle bzgl. Anzahl und Typ der Parameter muß explizit im Skript programmiert werden.

    M.a.W.: jedes in der Aufrufzeile angegebene Argument wird unkontrolliert durchgereicht!

Environmentvariable: Deklaration

- Mittels des Kommandos

  `export` *variablenname*

  kann eine Variable *variablenname* allen Sub-Shells, welche nach Ausführung dieses Kommandos gestartet werden, bekannt gemacht werden.

Environmentvariable: Gültigkeitsbereich

- Eine mittels des Kommandos

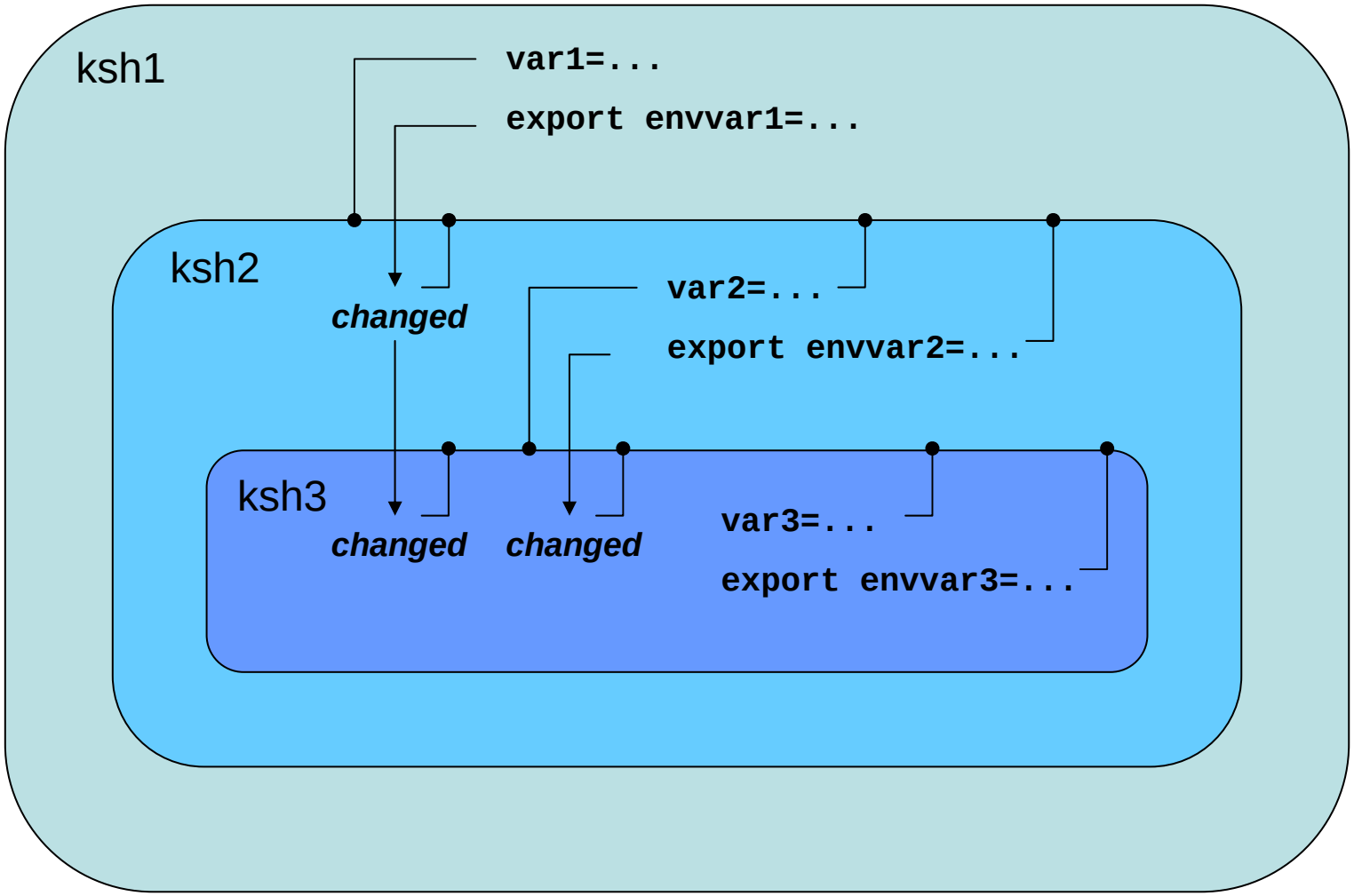
  `export` *variablenname*

  gesetzte Variable *variablenname* ist *allen Sub-Shells* (auch rekursiv!),
  welche nach Ausführung dieses Kommandos gestartet werden, *bekannt*.

- In Sub-Shells adaptierte Werte von Environmentvariablen stehen Sub-Sub-Shells *ohne erneuten* `export` zur Verfügung
  (dies ist anders als z.B. in der Bourne-Shell)

- Anders als z.B. von C-Programmen gewohnt, sind diese „globalen“ Variablen durch die Sub-Shells nicht global wirksam veränderbar!
  D.h. es gibt *keinen Informationstransfer von Sub-Shell zu aufrufender Shell*!

Environmentvariable: Gültigkeitsbereich



Environmentvariable: Beispiel

```
$ newenvvar=new                 # Variable initialisieren
$ print $newenvvar              # zur Kontrolle ausgeben
new
$ ksh                           # Sub-Shell starten; ohne export!
$ print $newenvvar              # Variable in Sub-Shell abfragen
                                # Leerstring, da nicht bekannt
$ exit                          # Subshell verlassen
$ export newenvvar              # exportieren an zukünftige Sub-Shells
$ ksh                           # Sub-Shell starten
$ print $newenvvar              # Variable in Sub-Shell abfragen
new                             # jetzt bekannt als globale Variable
$
```

Environmentvariable. Beispiel (Fortsetzg.)

```
$ ps
PID TTY          TIME CMD
1836 pts/1    00:00:00 bash
1869 pts/1    00:00:00 ksh        # Haupt-Korn-Shell
2174 pts/1    00:00:00 ksh        # Sub-Korn-Shell
2185 pts/1    00:00:00 ps
$ newenvvar=changed               # Änderung des Wertes in Sub-Shell
$ print $newenvvar                # wirksam in Sub-Shell
changed
$ print $$
2174
$ exit
$ print $newenvvar                # ABER: nicht wirksam in Haupt-Shell
new
$ print $$
1869
$
```

Variablen und Environmentvariable: welche sind gesetzt?

- Um herauszufinden, welche Variablen in einer Shell gerade gesetzt sind
  und welche davon Environmentvariablen sind, sind folgende Kommandos nützlich:

- **`env`**

  gibt alle (gesetzten) Environmentvariablen und deren Werte aus

- **`set`**

  gibt alle (gesetzten) Variablen inkl. der Environmentvariablen jeweils mit Wert aus

Gültigkeitsbereiche von (Env.)Variablen: das Kommando .

- Mittels des Kommandos

  . *skriptname*

  werden die Anweisungen von *skriptname* als Teil des Skripts, welches dieses Kommando enthält, ausgeführt.

  Damit funktioniert das Kommando **.** wie eine Makroersetzung, d.h. die Anweisungen aus *skriptname*, kann man sich per copy/paste eingefügt denken.

  Weitere Analogie: #include-Anweisung in C.

- Dieser Mechanismus wird z.B. beim Aufbau von Environments, z.B. beim Aufbau der allen Korn-Shells nach Start zur Verfügung stehenden Environment-Variablen, genutzt.

Vordefinierte Environmentvariablen der **ksh**: änderbar

- Folgende Environmentvariablen sind durch Nutzer manipulierbar:
  (Auszug; weitere im jeweiligen Zusammenhang)

Env.-Variable	Bedeutung
PATH	Suchpfad zur Lokalisierung auszuführender Befehle
CDPATH	Suchpfad zur Lokalisierung relativer Pfadnamen
ENV	Beim Start der `ksh` auszuführendes Skript
HISTFILE	Name der Datei, in der Kommandohistorie abgelegt wird
HISTSIZE	Anzahl gemerkter Kommandos in `$HISTFILE`
TMOUT	Zeitperiode in der Eingabe erfolgen muß, ohne daß Korn-Shell sich beendet
PS1	(Primärer) String für Prompt der Standardeingabezeile. Default: $
PS2	(Sekundärer) String für Prompt in fortgesetzter Standardeingabezeile. Default: >
PS3	Prompt-String für select-Schleifen. Default: #?
PS4	Prompt-String für Debug-Output bei Option ksh –x bzw. set –s. Default: +
	Bem: in den Promptvariablen werden Variablen substituiert und **!** Wird als Platzhalter für die aktuelle Kommandonummer verstanden.

Vordefinierte Environmentvariablen der **ksh**: automatisch

- Folgende Environmentvariablen werden durch die Korn-Shell ständig aktualisiert: (Auszug; weitere im jeweiligen Zusammenhang)

Env.-Variable	Bedeutung
HOME	Home-Verzeichnis
PWD	Aktuelles Arbeitsverzeichnis
OLDPWD	Vorheriges Arbeitsverzeichnis
PPID	Prozeßnummer der Parentshell der aktuell laufenden
SECONDS	Anzahl Sekunden seit der letzten Eingabe (siehe auch TMOUT)

- Bei der Manipulation dieser Variablen ist Vorsicht geboten!

Environmentvariablen: Nutzung

- Environmentvariable dienen i.allg. dazu
  - ggf. unbequem/schwierig zu gewinnende Information auf leichte Weise allen Nutzern zur Verfügung zu stellen:
    - `$USER`
    - `$HOST`
    - `$SECONDS`
  - zu gewährleisten, daß verschiedene Shells und/oder Sub-Shells auf gleiche Weise parametrisiert/konfiguriert werden *und* damit auch die in ihnen laufenden Skripte und Tools:
    - Z.B. alle aufgerufenen Skripten legen ihre Fehlermeldungen in ein und derselben Datei ab.
- Ein Beispiel für die Nutzung von Environmentvariablen ist die Möglichkeit Shell-Optionen zu Debuggingzwecken in aufgerufenen Shell-Skripten zu aktivieren.

  Skript: `test_newskriptOpt`

Datenübergabe an ein Skript

- Wie in der C-Programmierung können Daten im Rahmen

  – mittels globaler Variablen bereitgestellt werden. → ***Environmentvariablen***

  oder

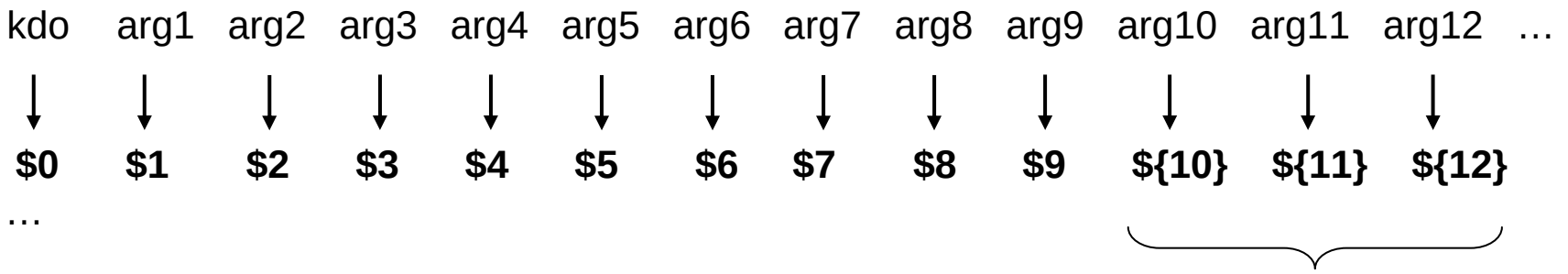
  ➢ einer Parameterübergabe → ***Positionsparameter***

  WICHTIG: Es gibt keine implizite Prüfung der Parameterzahl beim Skriptaufruf, d.h. die Kontrolle bzgl. Anzahl und Typ der Parameter muß explizit im Skript programmiert werden.

  M.a.W.: jedes in der Aufrufzeile angegebene Argument wird unkontrolliert durchgereicht!

Datenübergabe an ein Skript: Positionsparameter

- Jedes einem Kommando übergebene Argument wird dem Skript als sog. ***Positionsparameter*** bereitgestellt:



- Es können in der Korn-Shell *beliebig viele* Positionsparameter addressiert werden. (vorbehaltlich Systemgrenzen)

- Bsp:
  `Skript: showPosParam`

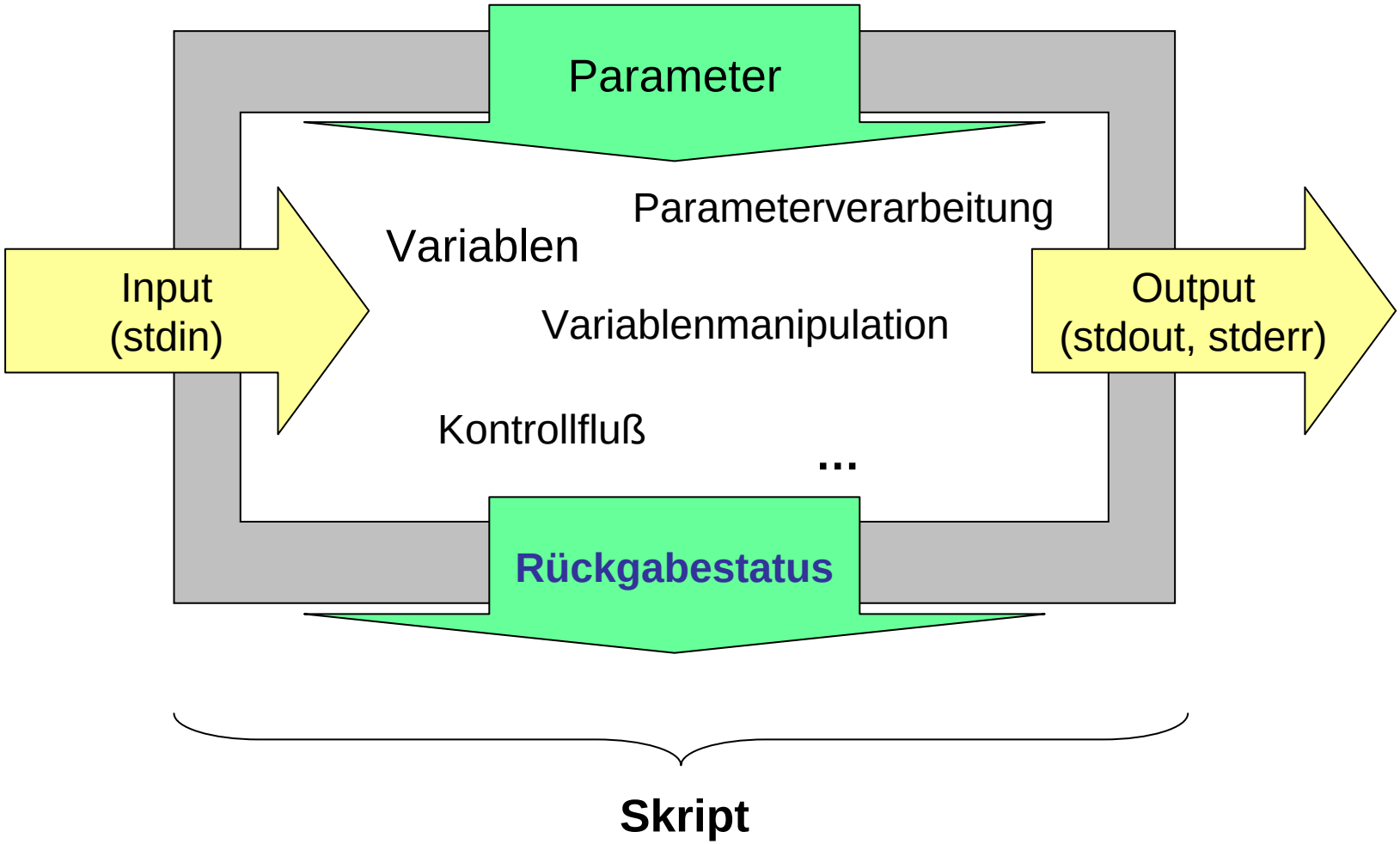
Datenübergabe an ein Skript: weitere vordefinierte Variablen

- Zusätzlich werden jedem Skript folgende vordefinierte Variablen bereitgestellt:

Vordefinierte Variable	**Bedeutung**
#	Enthält die Anzahl der übergebenen Argumente
@	Liste aller Argumente - in einzelne Strings verpackt: "$1" "$2" "$3" "$4" "$5" …
*	Liste aller Argumente - in einen einzigen String verpackt "$1 $2 $3 $4 $5 … "
$	Prozeß-ID des aktuellen Skripts (der aktuellen Shell) *Bem: praktisch z.B. für eindeutige Benennung von temporären Dateien*
!	Prozeß-ID des letzen Hintergrundprozesses

- Bsp:
  `Skript: showStandardVariables`

Wie geht es weiter?



Datenrückgabe durch ein Skript: Exit-Status

- Jedes Skript liefert einen sog. **Exit-Status** zurück.
- Dieser Exit-Status ist standardmäßig, d.h. wenn nicht explizit vor Verlassen des Skripts gesetzt, gleich dem Exit-Status des zuletzt innerhalb des Skripts ausgeführten Kommandos.
- Der Exit-Status ist eine Integerzahl >= 0.
- Hierbei gilt folgende

  WICHTIGE KONVENTION:

  $? **gleich 0** aufgerufenes Programm/Kommando endete **erfolgreich**.

  $? **ungleich 0** aufgerufenes Programm/Kommando endete mit **Fehler**.

  Dieser Konvention sollte man beim Erstellen eigener Skripten unbedingt folgen!

- Der Exit-Status kann folgendermaßen explizit gesetzt werden:
  `exit` [*n*]
  Gleichzeitig wird die umgebende Shell beendet.
- In der aufrufenden Shell ist dieser Exit-Status in der Variable **$?** gespeichert und kann entsprechend abgefragt werden.
  $? wird durch jedes ausgeführte Kommando aktualisiert!!

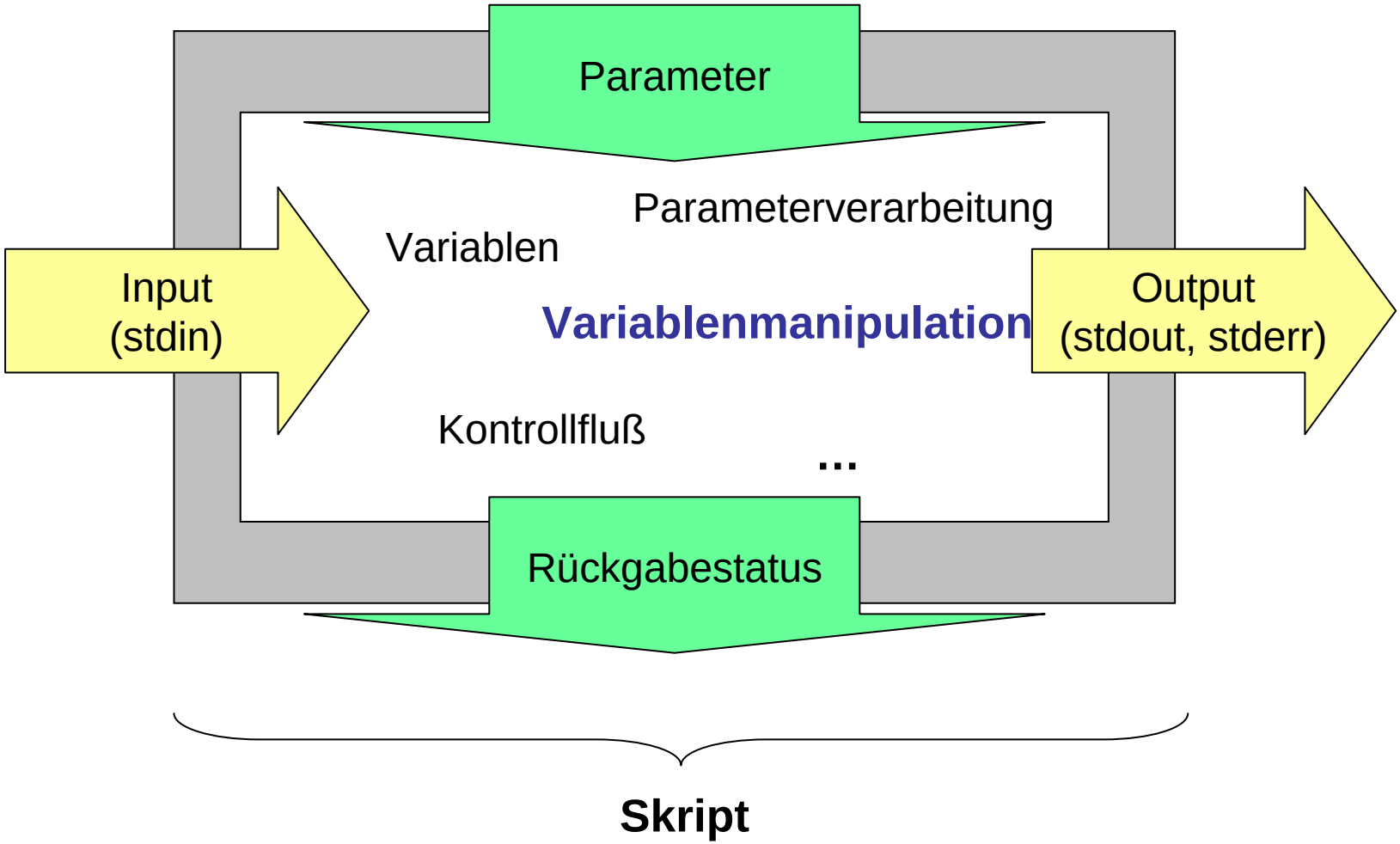
Datenschnittstelle von UNIX-Kommandos

- Das was bzgl. der Datenein/übergabe bzw.Datenrück/ausgabe von Skripten besprochen wurde, lässt sich auch bei Betrachtung von Manual Pages für UNIX-Kommandos nachvollziehen.

- Hier wird i.allg. beschrieben:
  - Welcher Art sollen die Aufrufparameter sein
  - Mittels welcher Environmentvariablen ist das Kommando konfigurierbar
  - Welche Exit-Stati liefert das Kommando zurück und welche Bedeutung haben diese

- Bsp:
```
man cmp
man diff
man grep
man less
```

Wie geht es weiter?



Variablenmanipulation

- Variablen können durch folgende Arten manipuliert werden:
  - Zuweisung eines Strings (siehe „Shell-Variablen: Grundlegendes“)
  - Zuweisung mittels Nutzung Standardoutput anderer Kommandos: Kommandosubstitution
  - Zuweisung mittels Nutzung Standardinput des Skripts: **`read`-Kommando**
  - Zuweisung anderer (ggf. ersetzter oder stringmanipulierter) Variablenwerte: ***Variablenausdrücke***
  - Zuweisung der Ergebnisse **arithmetischer Operationen**

Variablenmanipulation: String-Zuweisung

- *varname*=*string*
  - Erinnerung: keine Leerzeichen dürfen das Gleichheitszeichen umgeben!
  - Die vordefinierten Korn-Shell-Environmentvariablen sind großbuchstabig und bestehen aus mindestens 3 Zeichen. Dies sollte bei der Namenswahl berücksichtigt werden, um nicht unabsichtlich Korn-Shell-Variablen zu überschreiben.
  - *Bei* der *Zuweisung* findet *keine Dateinamensubstitution* statt!

```
>$ testvar=*\‘hal_?o
>$ print “$testvar“
*‘hal_?o
>$
```

Variablenmanipulation: Kommandosubstitution

- *varname*`=$(`*kdo_liste*`)` bzw. *varname*`=\``*kdo_liste*`\``
  - Wiederum: keine Leerzeichen dürfen das Gleichheitszeichen umgeben!
  - der Standardoutput der *kdo_liste* wird in der Variablen *varname* gespeichert

```
>$ ls
geraldine   gerd   gerhard   hannelore   jutta   werner
>$ myfiles=`ls`
>$ print $myfiles
geraldine gerd gerhard hannelore jutta werner
>$

>$ cat sayHello
print “Hallo ich bin gerade aufgerufen worden“
print “was so läuft?  -- Schaut selbst“
ps
print “und tschuess“
>$ mytext=$(< sayHello)     # identisch zu mytext=$(cat sayHello)
>$ print “$mytext“
print “Hallo ich bin gerade aufgerufen worden“
print “was so läuft?  -- Schaut selbst“
ps
print “und tschuess“
>$
```

Skript I/O: **`read`** – Variablenzuweisung durch stdin-Lesen

- Mittels des `ksh`-builtin-Kommandos **`read`**, kann Information von der Standardeingabe in Variablen gespeichert werden:
- **`read`** liest *eine Zeile* von der Standardeingabe.
  - Aufrufvariante 1:

    `read` *variablenname1* [*variablename2 … variablennameN*]

    Die gelesene Zeile wird in (durch white space separierte) Wörter aufgeteilt. Diese werden der Reihe nach den Variablen zugewiesen.

    Enthält die Eingabezeile mehr Wörter als der `read`-Aufruf Variablennamen, so werden die ersten N-1 Wörter 1:1 zugeordnet und die restlichen Wörter alle gemeinsam in die Variable N übernommen.

    Enthält die Eingabezeile weniger Wörter als der `read`-Aufruf Variablennamen, so erhalten die überzähligen Variablen den Leerstring als Wert.

  - Aufrufvariante 2:

    `read`

    Die gesamte gelesene Zeile wird in die Variable **`REPLY`** geschrieben.

Skript I/O: **`read`** – Interaktivität von Skripten

- Das Kommando **`read`** ist nützlich, um Variablenwerte abhängig von Benutzereingaben zu setzen
- Dies ist i. allg. idealerweise zu verknüpfen mit Ausgaben an den Benutzer

Folgende Varianten sind aus Benutzersicht gleichwertig:

- Varianten “print&read“:

1)
```
print -n “Gib die gewünschte Info ein:“
read infovar
```

2)
```
print “Gib die gewünschte Info ein:\c“
read infovar
```

3)
```
print -n “Gib die gewünschte Info ein:“
read
# Info steht in $REPLY
```

- Variante “read“:

```
read infovar?“Gib die gewünschte Info ein:“
```

- Gleichzeitige Zuweisung an mehrere Variablen:

```
read var1?text var2 var3
```

Variablenmanipulation

- Variablen können durch folgende Arten manipuliert werden:
  - Zuweisung eines Strings (siehe „Shell-Variablen: Grundlegendes“)
  - Zuweisung mittels Nutzung Standardoutput anderer Kommandos: Kommandosubstitution
  - Zuweisung mittels Nutzung Standardinput des Skripts: **`read`-Kommando**
  - ➢ Zuweisung anderer (ggf. ersetzter oder stringmanipulierter) Variablenwerte: ***Variablenausdrücke***
  - Zuweisung der Ergebnisse **arithmetischer Operationen**

Variablenausdrücke

- Folgende Ausdrücke charakterisieren **komplexere Parametersubstitutionen**, als die bekannte Ersetzung $*variablenname*

Ausdruck	**Ersetzungsergebnis**
${*varname***-***wort***}**	**Verwendung von Defaultwerten** Falls $*varname* ungesetzt, wird *wort* zurückgeliefert. Sonst wird Inhalt von *varname* zurückgegeben. *Inhalt von* varname *bleibt unverändert.*
${*varname***=***wort***}**	**Zuweisung von Defaultwerten** Falls $*varname* ungesetzt, wird *wort* zurückgeliefert und *varname* erhält den Wert *wort*. Sonst wird Inhalt von *varname* zurückgegeben. *Inhalt von varname wird verändert.*
${*varname***+***wort***}**	**Verwendung eines Alternativwerts** Falls $*varname* gesetzt, wird *wort* zurückgeliefert. Sonst Rückgabe des Leerstrings. *Inhalt von varname bleibt unverändert.*

Variablenausdrücke

- Folgende Ausdrücke charakterisieren **komplexere Parametersubstitutionen**, als die bekannte Ersetzung $*variablenname*

Ausdruck	**Ersetzungsergebnis**
${*varname***:-***wort***}**	**Verwendung von Defaultwerten** Falls $*varname* ungesetzt *oder leer*, wird *wort* zurückgeliefert. Sonst wird Inhalt von *varname* zurückgegeben. *Inhalt von* varname *bleibt unverändert.*
${*varname* **:=***wort***}**	**Zuweisung von Defaultwerten** Falls $*varname* ungesetzt *oder leer*, wird *wort* zurückgeliefert und *varname* erhält den Wert *wort*. Sonst wird Inhalt von *varname* zurückgegeben. *Inhalt von varname wird verändert.*
${*varname* **:+***wort***}**	**Verwendung eines Alternativwerts** Falls $*varname* gesetzt *und nicht leer*, wird *wort* zurückgeliefert. Sonst Rückgabe des Leerstrings. *Inhalt von varname bleibt unverändert.*

Variablenausdrücke: Beispiele – Defaultwerte

```
$ count=“”                  # count ist gesetzt und enthält Leerstring

$ print ${count:-0}         # Prüfung auf gesetzt oder Leerstring
0
$ print ${count-0}          # Prüfung auf gesetzt

$ unset count               # count ist nicht gesetzt
$ print ${count-0}          # Prüfung auf gesetzt
0
$
$
$ print $2                  # Positionsparameter 2 ist nicht gesetzt

$ print ${2-DEFAULTWERT}    # da nicht gesetzt DEFAULTWERT ausgeben
DEFAULTWERT
```

Variablenausdrücke: Beispiele

```
$ print $var1 $var2

$ var1=hans

$ print ${var1-georg}
hans
$ print ${var2-georg}
georg
$ print $var1
hans
$ print $var2

$ print ${var1=georg}
hans
$ print $var1
hans
$ print ${var2=georg}
georg
$ print $var2   # var2 wurde verändert
georg
```

Fortsetzung:

```
$ print ${var1+georg}
georg
$ print $var1
hans
$ unset var2
$ print ${var2+georg}

$
```

Variablenausdrücke

- Eine weitere Möglichkeit:

Ausdruck	**Ersetzungsergebnis**
${*varname***?***wort***}**	**Fehler-Exit, wenn nicht gesetzt** Falls $*varname* ungesetzt, wird das Skript mit Fehlermeldung *wort* abgebrochen.
${*varname***:?***wort***}**	**Fehler-Exit, wenn Nullwert oder nicht gesetzt** Falls $*varname* ungesetzt *oder leer*, wird das Skript mit Fehlermeldung *wort* abgebrochen.

- HINWEIS:

  In allen Variablenausdrücken steht *varname* auch für die Nummern der Positionsparameter!

  Bsp.: Prüfung auf Existenz von $1:

```
${1?falscher Aufruf: erster Parameter ungesetzt}
```

Variablenausdrücke

- Folgende Ausdrücke sind eine **Besonderheit der ksh** (gegenüber Bourne-Shell):
- Länge und Anzahl

Ausdruck	**Gleichwertige Alternative**	**Ersetzungsergebnis**
${*#varname***}**		Länge des Inhalts von *varname*
${*#varname***[***n***]}**		Länge des Inhalts des *n*-ten Array-Elements
${*#varname***[*]}**	**${***#varname***[@]}**	Anzahl der Elemente des Arrays *varname*

- Ersetzung von Teilstrings aus $*varname*:

Ausdruck	**Ersetzungsergebnis**
${*varname#pattern***}**	kleinstes *pattern* genügendes Anfangsstück von $*varname* abschneiden
${*varname##pattern***}**	größtes *pattern* genügendes Anfangsstück von $*varname* abschneiden
${*varname%pattern***}**	kleinstes *pattern* genügendes Endstück von $*varname* abschneiden
${*varname%%pattern***}**	größtes *pattern* genügendes Endstück von $*varname* abschneiden

Variablenausdrücke: Beispiele (ohne Ergebnisse)

```
>$ dateien=ccbmbbmdd

>$ print ${dateien#*b}

>$ print ${dateien##*b}

>$ print ${dateien#c}

>$ print ${dateien##c}

>$ print ${dateien%m}

>$ print ${dateien%m*}

>$ print ${dateien%%m*}
```

Variablenausdrücke: Beispiele

```
>$ dateien=ccbmbbmdd

>$ print ${dateien#*b}
mbbmdd
>$ print ${dateien##*b}
mdd
>$ print ${dateien#c}
cbmbbmdd
>$ print ${dateien##c}
cbmbbmdd
>$ print ${dateien%m}
ccbmbbmdd
>$ print ${dateien%m*}
ccbmbb
>$ print ${dateien%%m*}
ccb
```

Variablenausdrücke: Beispiele (ohne Ergebnisse)

```
>$ pattern=ende
>$ var1=dickesende
>$ print $var1 $pattern
dickesende ende
>$
>$ print ${var1%ende}

>$ print ${var1%$pattern}

>$ print ${var1%`print $pattern`}

>$ print ${var1%$(print $pattern)}
```

Fortsetzung:

```
>$ pattern=es
>$ print ${var1%$pattern}

>$ pattern=es*
>$ print ${var1%$pattern}

>$ print ${var1%`print $pattern`}

>$ print ${var1%$(print $pattern)}

>$ pattern=ende
>$ print ${var1%${pattern#en}}
```

Variablenausdrücke: Beispiele

```
>$ pattern=ende
>$ var1=dickesende
>$ print $var1 pattern
dickesende pattern
>$
>$ print ${var1%ende}
dickes
>$ print ${var1%$pattern}
dickes
>$ print ${var1%`print $pattern`}
dickes
>$ print ${var1%$(print $pattern)}
dickes
```

Fortsetzung:

```
>$ pattern=es
>$ print ${var1%$pattern}
dickesende
>$ pattern=es*
>$ print ${var1%$pattern}
dick
>$ print ${var1%`print $pattern`}
dick
>$ print ${var1%$(print $pattern)}
dick
>$ pattern=ende
>$ print ${var1%${pattern#en}}
dickesen
```

- Variablenausdrücke bieten eine große Flexibilität.
- Insbesondere können Ersetzungspattern dynamisch generiert werden (Kommandosubstitution oder als berechneter Variablenwert)

Variablenausdrücke: Beispiel dirs, pushd, popd

```
function dirs {        # print directory stack (easy)
    print $DIRSTACK
}

function pushd {       # push current directory onto stack
    dirname=$1
    cd ${dirname:?"missing directory name."}
    DIRSTACK="$PWD $DIRSTACK"
    print "$DIRSTACK"
}

function popd {        # cd to top, pop it off stack
    top=${DIRSTACK%% *}
    DIRSTACK=${DIRSTACK#* }
    cd $top
    print "$PWD"
}
```

Nach:
Rosenblatt&Robbins,
Kap. 4

Variablenmanipulation

- Variablen können durch folgende Arten manipuliert werden:
  - Zuweisung eines Strings (siehe „Shell-Variablen: Grundlegendes“)
  - Zuweisung mittels Nutzung Standardoutput anderer Kommandos: Kommandosubstitution
  - Zuweisung mittels Nutzung Standardinput des Skripts: **`read`-Kommando**
  - Zuweisung anderer (ggf. ersetzter oder stringmanipulierter) Variablenwerte: ***Variablenausdrücke***
  - ➢ Zuweisung der Ergebnisse **arithmetischer Operationen**

Arithmetik ganzer Zahlen: **`(( ))`-Kommando**

- Zur Berechnung arithmetischer Ausdrücke gibt es in der Bourne-Shell (sh) das Kommando **`expr`**.

- Die Korn-Shell verfügt über das performantere builtin-Kommando **`let`**:

  `let` *ausdruck*

- Alternativ kann auch geschrieben werden:

  `((` *ausdruck* **`))`**

- Vorteil dieses alternativen Kommandos ist, daß hier Leer-, Tabulator- und sonstige Metazeichen nicht per Quoting ausgeschaltet werden müssen. Denn:

  `((` *ausdruck* **`))`** ist gleich **`let`** "*ausdruck*"

- (geringer) Nachteil: mittels **`(( ))`** kann nur genau ein arithmetischer Ausdruck ausgewertet werden.

Arithmetik ganzer Zahlen: Verknüpfungen

- Die Korn-Shell stellt arithmetische Operatoren für ganzzahlige (!) Werte zur Verfügung. Die Schreibweise orientiert sich an der Programmiersprache C.

Operator	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
ausdr1 **+** *ausdr2*	Addition der Werte der arithmetischen Ausdrücke *ausdr1* und *ausdr2*
ausdr1 **-** *ausdr2*	Subtrakttion der Werte der arithmetischen Ausdrücke *ausdr1* und *ausdr2*
ausdr1 ***** *ausdr2*	Multiplikation der Werte der arithmetischen Ausdrücke *ausdr1* und *ausdr2*
ausdr1 **/** *ausdr2*	Division der Werte der arithmetischen Ausdrücke *ausdr1* und *ausdr2*
ausdr1 **%** *ausdr2*	Wert des *ausdr1* modulo Wert des *ausdr2*

- Ferner existieren für alle Operatoren **+**, **-**, *****, **/** und **%** Kurzformen (wie von C bekannt):
  var **+=** *ausdr* für *var* **=** *var* **+** *ausdruck*
- Werden in arithmetischen Ausdrücken nicht-zahlwertige Variablen verwendet, so wird deren Inhalt durch den Wert 0 ersetzt.
- Es existieren auch Bit-Operationen. Siehe Literatur bzw. `ksh`-Manual Page

  HINWEIS: bei Verwendung von `let` dürfen keine Leerzeichen zwischen Operanden und Operator stehen!)

Arithmetik ganzer Zahlen: **`integer`**-Variablen

- Die `ksh` erlaubt die Attributierung von Variablen. Insbesondere können Variablen als ganzzahlige Variablen attributiert werden:

  `typeset -i` *gz_variable* oder **`integer`** *gz_variable*

- Mit `integer`-Variablen dürfen arithmetische Ausdrücke gebildet werden als:

  ```
  integer a b c
  ```

  - rechte Seite einer Zuweisung an `integer`-Var. `b=a+4*(b/c-7)`
  - Array-Index `y[a*b-10]`
  - Argument des Kommandos `shift` `shift a-b*c`
  - Operand von arithm. Vergleichsoperatoren `[[ a-b –ne b*c ]]`
  - Argumente von `let` bzw. `(( ))` `(( (a-b)/ b *c ))`
- $-Zeichen zur Referenzierung kann hier entfallen

- VORSICHT: ohne `(( ))` ist das Quoting aufgehoben! D.h. die arithmetischen Ausdrücke müssen leerzeichenfrei sein!

Arithmetik ganzer Zahlen: **`integer`**-Variablen (unvollst.!)

```
>$ integer a=1 b=2 c=3 d=4    # ganzzahlige Variablen
>$ print $a $b $c $d
1 2 3 4
>$ a=b+c*d                    # a ist integer -> rS = arithm. Ausdruck
>$ print $a

>$
>$ bb=2 cc=3 dd=4             # KEINE integer-Variablen
>$ print $bb $cc $dd          # nur zahlwertige Strings
2 3 4
>$ aa=bb+cc*dd                # aa ist beliebig -> rS = String
>$ print “$aa“                # Dateinamensubstitution verhindern.

>$ a=bb+cc*dd                 # a ist integer -> rS = arithm. Ausdruck
>$ print $a

>$ dd=quatsch
>$ a=bb+cc*dd                 # a ist integer -> rS = arithm. Ausdruck
>$ print $a
```

Arithmetik ganzer Zahlen: **`integer`**-Variablen

```
>$ integer a=1 b=2 c=3 d=4    # ganzzahlige Variablen
>$ print $a $b $c $d
1 2 3 4
>$ a=b+c*d                   # a ist integer -> rS = arithm. Ausdruck
>$ print $a
14
>$
>$ bb=2 cc=3 dd=4            # KEINE integer-Variablen
>$ print $bb $cc $dd         # nur zahlwertige Strings
2 3 4
>$ aa=bb+cc*dd               # aa ist beliebig -> rS = String
>$ print “$aa“               # Dateinamensubstitution verhindern.
bb+cc*dd
>$ a=bb+cc*dd                # a ist integer -> rS = arithm. Ausdruck
>$ print $a
14
>$ dd=quatsch
>$ a=bb+cc*dd                # a ist integer -> rS = arithm. Ausdruck
>$ print $a
2
```

Arithmetik ganzer Zahlen: **integer**-Variablen (unvollst.!)

```
>$ integer a=1 b=2 c=3 d=4
>$ print $a $b $c $d
1 2 3 4
>$ bb=2 cc=3 dd=4      # KEINE integer-Variablen
>$ print $bb $cc $dd   # nur zahlwertige Strings
2 3 4
>$
>$ aa=b+c*d            # aa ist beliebig -> rS = String
>$ print “$aa“         # Dateinamensubstitution verhindern.

>$
>$ let aa=b+c*d        # mittels let Berechnung des Ausdrucks!
>$ print “$aa“


ODER

>$ (( aa = b + c*d )) # mittels (( )) Berechnung des Ausdrucks!
>$ print “$aa“
```

Arithmetik ganzer Zahlen: **integer**-Variablen

```
>$ integer a=1 b=2 c=3 d=4
>$ print $a $b $c $d
1 2 3 4
>$ bb=2 cc=3 dd=4      # KEINE integer-Variablen
>$ print $bb $cc $dd   # nur zahlwertige Strings
2 3 4
>$
>$ aa=b+c*d            # aa ist beliebig -> rS = String
>$ print “$aa“         # Dateinamensubstitution verhindern.
b+c*d
>$
>$ let aa=b+c*d        # mittels let Berechnung des Ausdrucks!
>$ print “$aa“
14

ODER

>$ (( aa = b + c*d )) # mittels (( )) Berechnung des Ausdrucks!
>$ print “$aa“
14
```

Arithmetik ganzer Zahlen: **`integer`**-Variablen

```
>$ integer a=1 b=2 c=3 d=4
>$ print $a $b $c $d
1 2 3 4
>$
>$ a+=b
>$ print $a
3
>$
```

WARNUNG:

- Arithmetische Ausdrücke können nicht zur Berechnung der Nummern von Positionsparametern verwendet werden:

  `integer a=5 b=6`

  `${b-a} bzw. ${a+b}` sind Variablenausdrücke bzgl. der Variablen `a` und stehen nicht für ${1} bzw. ${11}

- Die Zuweisung von nicht zahlwertigen Strings an Integervariablen führt in der `pdksh` leider *nicht* zu einer Fehlermeldung. Eigentlich Korn-Shell-Standard! Stattdessen ist das Ergebnis der Zuweisung 0.

Arithmetik ganzer Zahlen: Ersetzung auf Kommandozeile

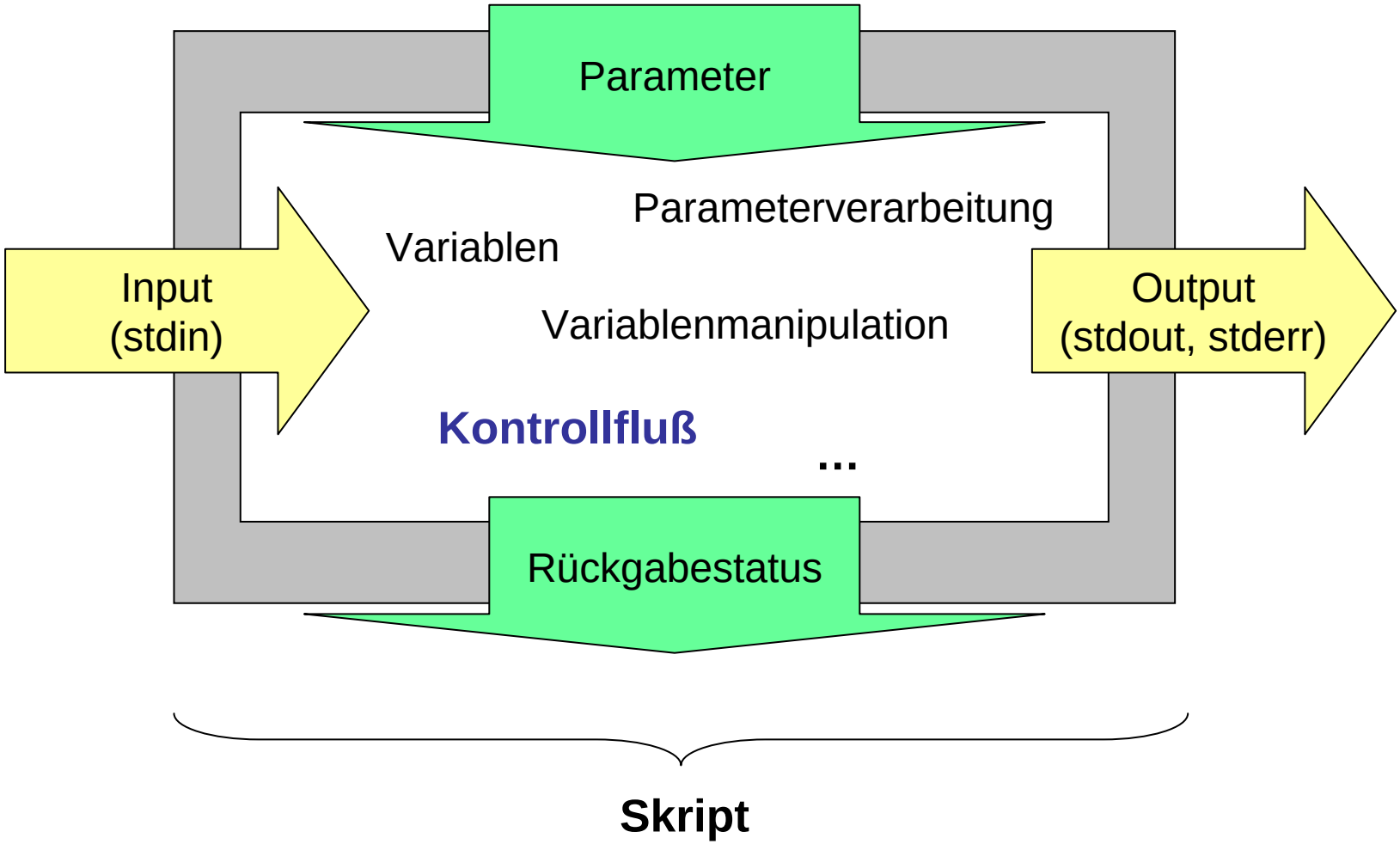
```
>$ integer a=1 b=2 c=3 d=4
>$ print $(( a + b*c + d ))
>$
```

- Arithmetische Ausdrücke können auch innerhalb der Kommandozeile durch $((…)) geklammert stehen. Der entsprechende Teil der Kommandozeile wird durch das Ergebnis des arithmetischen Ausdrucks ersetzt.

Arithmetik ganzer Zahlen: Ersetzung auf Kommandozeile

```
>$ integer a=1 b=2 c=3 d=4
>$ print $(( a + b*c + d ))
>$
```

Wie geht es weiter?



Kontrollstrukturen: if-Anweisung

- Die Struktur dieser Anweisung ist Standard
- Die if-Abfrage gilt als erfüllt, wenn der **Exit-Status** der jeweiligen Kommandoliste 0 ist.
  - D.h. *then_kdoliste2* wird nur ausgeführt, wenn der Exitstatus von *if_kdoliste1* und *if_kdoliste2* 0 ist.
  - *else_kdoliste* wird ausgeführt, wenn alle *if_kdoliste*n Exit-Status 1 liefern

    usw.
- Der Exit-Status der gesamten if-Anweisung ist gleich dem Status der letzten Anweisung. Wurde keine Anweisung ausgeführt, gilt Exit-Status=0.

```
if if_kdoliste1
then
   then_kdoliste1
[ elif if_kdoliste2
  then
     then_kdoliste2 ]

…

[ else
     else_kdoliste ]
fi
```

Nach: Herold, S.321

Kontrollstrukturen: if-Anweisung - Beispiel

```
>$ if print hans >/dev/null
> then
>   print erfolgreich
> else
>   print nicht erfolgreich
> fi
erfolgreich
>$
>$ if cat hans 2>/dev/null 1>&2
> then
>   print erfolgreich
> else
>   print nicht erfolgreich
> fi
nicht erfolgreich
```

Bedingungen prüfen: **[[ ]]**-Kommando

- Alle Strukturen der **ksh**, welche den Kontrollablauf steuern, tun dies in Abhängigkeit des exit-Status einer ausgeführten Kommandoliste

- Um auch vergleichende **Bedingungen** im Rahmen dieses Mechanismus zu erfassen, gibt es das `ksh`-Kommando **[[ ]]**, welches bei
  - **Erfülltheit** der geprüften Bedingung den **exit-Status 0** und bei
  - **Nichterfülltheit** den **exit-Status != 0** zurückliefert.

- Mittels **[[ ]]** können
  - Dateieigenschaften geprüft
  - Zeichenketteneigenschaften geprüft
  - Zeichenketten verglichen
  - Numerische Werte verglichen werden

- Ferner können die einzelnen Prüfungen/Vergleiche logisch verknüpft werden.

- Aufruf:

  [[*testausdruck* **]]** # Leerzeichen beachten!

Bedingungen prüfen: Testausdrücke für Zeichenketten

- Vergleich von Zeichenketten:

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
string1 **=** *pattern*	*string1* dem *pattern* genügt
string1 **!=** *pattern*	*string1* dem *pattern* nicht genügt
string1 **<** *string2*	*string1* **<** *string2* nach ASCII-Code ist
string1 **>** *string2*	*string1* **>** *string2* nach ASCII-Code ist

- Prüfung von Zeichenketten:

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
-z *string*	*string* die Länge 0 hat (**z**ero)
-n *string*	*string* die Länge ungleich 0 hat (**n**ot zero)

HINWEIS: Zwischen Operanden und Operatoren müssen Leerzeichen stehen!

Bedingungen prüfen: Testausdrücke für Dateien

- Prüfung von Dateieigenschaften (nicht vollständig aufgeführt):

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn Datei namens *datname***
-r *datname*	Existiert und gelesen werden darf (**r**eadable)
-w *datname*	Existiert und beschrieben werden darf (**w**ritable)
-x *datname*	Existiert und ausführbar ist (e**x**ecutable)
-f *datname*	Existiert und eine normale Datei ist (**f**ile)
-d *datname*	Existiert und ein Directory ist (**d**irectory)
-s *datname*	Existiert und nicht leer ist (**s**pace)
-L *datname*	Existiert und ein symbolischer Link ist (**L**ink)

Bedingungen prüfen: Testausdrücke für Dateien

- Vergleich von Dateieigenschaften:

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
dat1 **-nt** *dat2*	Die Datei *dat1* neuer ist als die Datei *dat2* (**n**ewer **t**han)
dat1 **-ot** *dat2*	Die Datei *dat1* älter ist als die Datei *dat2* (**o**lder **t**han)
dat1 **-ef** *dat2*	*dat1* eine Datei selben Inhalts, wie *dat2* bezeichnet (**e**qual **f**ile)

Bedingungen prüfen: Vergleich ganzer Zahlen (mittels [[ ]])

- Für den Vergleich ganzer Zahlen stehen folgende Möglichkeiten zur Verfügung:

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
zahl1 **-eq** *zahl2*	Die beiden ganzen Zahlen *zahl1* und *zahl2* gleich sind (**eq**ual)
zahl1 **-ne** *zahl2*	Die beiden ganzen Zahlen *zahl1* und *zahl2* ungleich sind (**n**ot **e**qual)
zahl1 **-gt** *zahl2*	Die ganzen Zahl *zahl1* größer als *zahl2* ist (**g**reater **t**han)
zahl1 **-ge** *zahl2*	Die ganzen Zahl *zahl1* größer oder gleich *zahl2* ist (**g**reater or **e**qual than)
zahl1 **-lt** *zahl2*	Die ganzen Zahl *zahl1* kleiner als *zahl2* ist (**l**ess **t**han)
zahl1 **-le** *zahl2*	Die ganzen Zahl *zahl1* kleiner oder gleich *zahl2* ist (**l**ess or **e**qual than)

Bedingungen prüfen: Verknüpfung von Testausdrücken

- Alle vorher aufgelisteten Testausdrücke können mittels logischer Operatoren verknüpft und mittels Klammerung bzgl..ihrer Auswertungsreihenfolge strukturiert werden:

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
! *ausdr*	Negation des Resultats von *ausdruck*
ausdr1 **&&** *ausdr2*	Logische UND-Verknüpfung
ausdr1 **||** *ausdr2*	Logische ODER-Verknüpfung
(*ausdr* **)**	Klammerung zur Steuerung des Operatorenvorrangs

HINWEIS:

- Diese logischen Operatoren sind nicht mehr konform zur Bourne-Shell-Syntax.
- Die auf den 4 vorangehenden Folien aufgeführten Vergleichsoperatoren gelten auch in der Bourne-Shell (mit entsprechend eingeschränkter *pattern*-Syntax).
- Die aus der Bourne-Shell bekannten Anweisungen **`test`** und **`[ ]`** dürfen auch in der Korn-Shell verwendet werden, werden aber durch **`[[ ]]`** vollständig subsumiert.

  wesentlicher Unterschied: innerhalb von `[[ ]]` findet keine Dateinamenexpansion statt und auch die Klammersymbole `(` bzw. `)` müssen nicht mehr gequotet werden.

Kontrollstrukturen: if-Anweisung -- Kompaktform

- Einseitige if-Anweisungen können kompakter formuliert werden.

```
if if_kdoliste
then
    then_kdoliste
fi
```

==

if_kdoliste **&&** *then_kdoliste*

```
if  ! if_kdoliste
then
   kdoliste
fi
```

==

```
if if_kdoliste
then
    :        # Nullbefehl
else
   kdoliste
fi
```

==

if_kdoliste **||** *kdoliste*

- Ggf. folgende Short-Cut-Kombination sinnvoll:

if_kdoliste **&&** *then_kdoliste* **||** *else_kdoliste*

Beachte: hier gilt, wenn *then_kdoliste* mit Exit-Status ungleich 0 endet, dann wird *else_kdoliste* ausgeführt!

Bedingungen prüfen: Integer-Vergleiche mittels **(( ))**

- Analog zum Kommando `[ [ ] ]` und den dort beschriebenen Vergleichen für zahlwertige Variablen, ermöglicht die Korn-Shell auch den Vergleich von Zahlen mittels des Kommandos **(( ))**

- Die hier verwendbaren Vergleichsoperatoren entsprechen der C-Syntax und sind damit leicht zu merken.

- Vorteil der Nutzung von `( ( ) )`:
  - das einleitende $ vor Variablennamen darf entfallen!
  - Leerzeichen müssen Operatoren und Operanden nicht mehr trennen!

- WARNUNG:

  enthält eine Variable keinen zahlwertigen String, so wird Ihr Wert als 0 angenommen!

Arithmetik ganzer Zahlen: Vergleichsoperationen

- Es sind alle Vergleiche möglich, welche auch das test-Kommando beherrscht. Allerdings orientiert sich die Schreibweise an C.

Testausdruck	**Exit-Status = 0, d.h. Bedingung erfüllt, wenn**
zahl1 **==** *zahl2*	Die beiden ganzen Zahlen *zahl1* und *zahl2* gleich sind (**eq**ual)
zahl1 **!=** *zahl2*	Die beiden ganzen Zahlen *zahl1* und *zahl2* ungleich sind (**n**ot **e**qual)
zahl1 **>** *zahl2*	Die ganzen Zahl *zahl1* größer als *zahl2* ist (**g**reater **t**han)
zahl1 **>=** *zahl2*	Die ganzen Zahl *zahl1* größer oder gleich *zahl2* ist (**g**reater or **e**qual than)
zahl1 **<** *zahl2*	Die ganzen Zahl *zahl1* kleiner als *zahl2* ist (**l**ess **t**han)
zahl1 **<=** *zahl2*	Die ganzen Zahl *zahl1* kleiner oder gleich *zahl2* ist (**l**ess or **e**qual than)

- Es gelten dieselben logischen Operatoren wie für das Kommando **`[[ ]]`**

(HINWEIS: bei Verwendung von `let` dürfen keine Leerzeichen zwischen Operanden und Operator stehen!)

Kontrollstrukturen: case-Anweisung

- Die *pattern* werden der Reihe nach geprüft. Die *kdoliste* des zuerst gefundenen *pattern*, welcher *wort* charakterisiert, wird ausgeführt. Danach ist die case-Anweisung beendet.
  → speziellere *pattern* müssen immer vor gröberen stehen, um das gewünschte Resultat zu erhalten.

- *wort* unterliegt Dateinamenexpansion, Kommandosubstitution und Parametersubstitution

- *pattern* dürfen gemäß aller bereits im Rahmen der Dateinamenexpansion beschriebenen Möglichkeiten gebildet werden

- Der Exit-Status der gesamten case-Anweisung ist gleich dem Status der zuletzt ausgeführten Anweisung. Wurde keine Anweisung ausgeführt, gilt exit-Status=0.

case *wort* **in**
*pattern_1***)** *kdoliste_1***;;**
*pattern_2***)** *kdoliste_2***;;**
*pattern_3***)** *kdoliste_3***;;**

…

*pattern_n***)** *kdoliste_n***;;**
esac

Nach: Herold, S.322

Kontrollstrukturen: case-Anweisung

```
print "Gib Hexaziffer ein"
read hziff
print "$hziff (16) =  $(case $hziff in
                         ([0-9]) print        $hziff;;
                         ([Aa]) print 10;;
                         ([bB]) print 11;;
                         ([cC]) print 12;;
                         ([dD]) print 13;;
                         ([eE]) print 14;;
                         ([fF]) print 15;;
                            (*) print "Keine Hexaziffer";;
                       esac) (10)"
```

Nach: Herold, S.261

- Wird case innerhalb einer Kommandosubstitution des Typs $() verwendet, so ist vor jeden pattern ein öffnende Klammer zu setzen, um die Klammerhierarchie zu erhalten!
- Bem.: Sie dürfen auch immer öffnende Klammern voranstellen.

Kontrollstrukturen: while- und until-Schleife

- Ausführung der **while-Schleife** solange exit-Status der *kdoliste1* **0** ist.
- Der Exit-Status der gesamten while-Schleife ist gleich dem Status der zuletzt ausgeführten Anweisung im Schleifenrumpf.

  Wurde keine Anweisung ausgeführt, gilt Exit-Status=0.

```
while kdoliste1
do
  kdoliste2
done
```

Nach: Herold, S.323

- Ausführung der **until-Schleife** solange exit-Status der *kdoliste1* **1** ist.
- Der Exit-Status der gesamten while-Schleife ist gleich dem Status der zuletzt ausgeführten Anweisung im Schleifenrumpf.

  Wurde keine Anweisung ausgeführt, gilt Exit-Status=0.

```
until kdoliste1
do
  kdoliste2
done
```

Nach: Herold, S.324

Kontrollstrukturen: if, while und until

- Erinnerung:

  Die Kontrollstrukturen if, while und until erlauben die Verwendung beliebiger Kommandofolgen in ihrem „Bedingungsteil“. Dort muß nicht zwingend ein Testkommando (`test`, `[]`, `[[ ]]` oder `(( ))`)aufgerufen werden.

- Bsp.: Einlesen der Zeilen einer Datei

```
while read
do
        …
        … $REPLY …
        …
done < dateiname
```

- Zur Bildung von Endlosschleifen stehen die Kommandos oder Aliase

  `true` und **`false`**

  zur Verfügung, welche immer Exit-Status gleich 0 bzw. ungleich 0 zurückliefern.

Kontrollstrukturen: for-Schleife

- Die for-Schleife ist etwas anders als sonst von Programmiersprachen gewöhnt.
- Sie verfügt über keinen expliziten numerischen Zähler, mit dem z.B. eine Liste durchlaufen wird.
- Stattdessen erhält sie die Liste direkt als Argument.

for *laufvariable* [**in** *wort1 wort2 … wortn*]
do
  kdoliste
done

Nach: Herold, S.325

- Fehlt der optionale Teil, d.h. es wird keine Liste angegeben, so wird implizit die Liste der Positionsparameter benutzt.

- Der Exit-Status der gesamten for-Schleife ist gleich dem Status der zuletzt ausgeführten Anweisung. Wurde keine Anweisung ausgeführt, gilt Exit-Status=0.

Bem: Eine for-Schleife im bekannten Sinne ist in der `ksh` z.B. mittels einer while-Schleife zu realisieren

Kontrollstrukturen: Schleifen

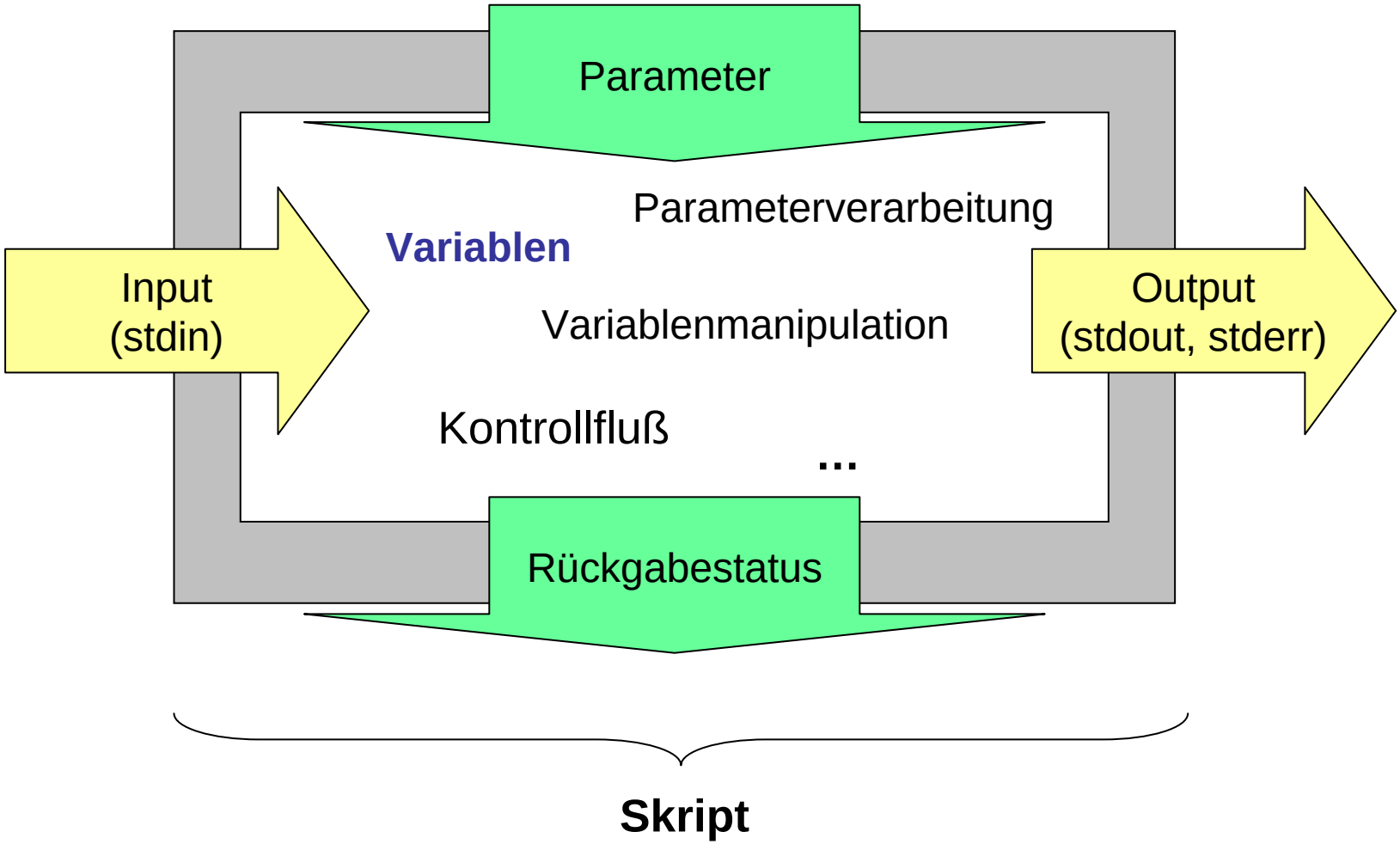
- Zur übersichtlicheren Gestaltung des Programmcodes stellt auch die Korn-Shell Möglichkeiten zum
  - regulären vorzeitigen Verlassen von Schleifen: **`break`**

  oder
  - zum Abbruch der aktuellen Bearbeitung der Schleifenrumpfs und sofortigen Fortsetzung mit der nächsten Iteration: **`continue`**

  zur Verfügung!

- Bem.: mittels **`break`** wird der Schleifenrumpf mit Exit-Status 0 verlassen.

Wie geht es weiter?



Arrays

- Neben einfachen Variablen erlaubt die Korn-Shell auch die Verwendung von Feldern
- Felder können mittels des Statements

  typeset *arrayname***[***anz_el***]**

  deklariert werden.
- Diese Deklaration ist nicht zwingend und führt auch zu keiner Konsequenz bei „Feldüberlauf“. Lediglich die Überschreitung der systemspezifischen erlaubten maximalen Anzahl von Feldelementen wird geprüft.
- Belegung von Feldelementen

  *arrayname***[***index***]=** …

  mittels aller bekannten Zuweisungsmechanismen
- Zugriff auf Feldelemente mittels

  ${*arrayname***[***index***]}**
- *anz_el* und *index* können irgendwelche Korn-Shell-Ausdrücke sein, welche einen numerischen Wert liefern (siehe insb. auch „Arithmetik ganzer Zahlen“).

Arrays

- Auch für individuelle Feldelemente gelten die bekannten Variablenausdrücke

  Z.B.:

  `${textarray[27]##*([0-9])}` repräsentiert Inhalt von `textarray[27]` gekürzt um führende Ziffern

  `${ar1[3]-27}` repräsentiert die Ersetzung durch den Defaultwert 27, falls `ar1[3]` nicht gesetzt.

- Insbesondere bezeichnet
  ${*#arrayname***[***index***]}** die Länge des *index*-ten Arrayelements von *arrayname*

- Die Anzahl der *gesetzten* Feldelemente enthält:
  ${***#arrayname***[*]}** oder **${*#arrayname***[@]}**

- Analog zu Positionsparametern können die Inhalte aller *gesetzter* Feldelemente mittels
  - **${***arrayname***[*]}** in einem String zusammengefasst oder mittels
  - **${***arrayname***[@]}** als Liste von Strings angesprochen

  werden

Arrays

HINWEISE:

- Zugriff mittels `$textarray[27]`geht nicht → Parameter-und Dateinamenexpansion!
- Die Belegung von Feldelementen ausserhalb eines ggf. mittels typeset deklarierten Index-Bereichs führt zu keinen Konsequenzen!
- ${#*arrayname[*]*} liefert *nicht* den mittels `typeset` deklarierten Maximal-Index, sondern nur die Anzahl der Feldelemente denen Werte zugewiesen wurden (das kann auch der Nullstring sein!)
- Arrayelemente, denen nichts explizit zugewiesen wurde, liefern bei Abfrage den Nullstring (sie haben aber nicht den Wert Nullstring!)
- Schleifenbildung:
  ist bequem ohne expliziten Zähler mittels ${a*rrayname*[@]} möglich.
  VORSICHT: es wird dann nur über die belegten Elemente iteriert!
- Das Kommando **`set`** gibt alle belegten Feldelemente mit Index aus

Arrays: Beispiel

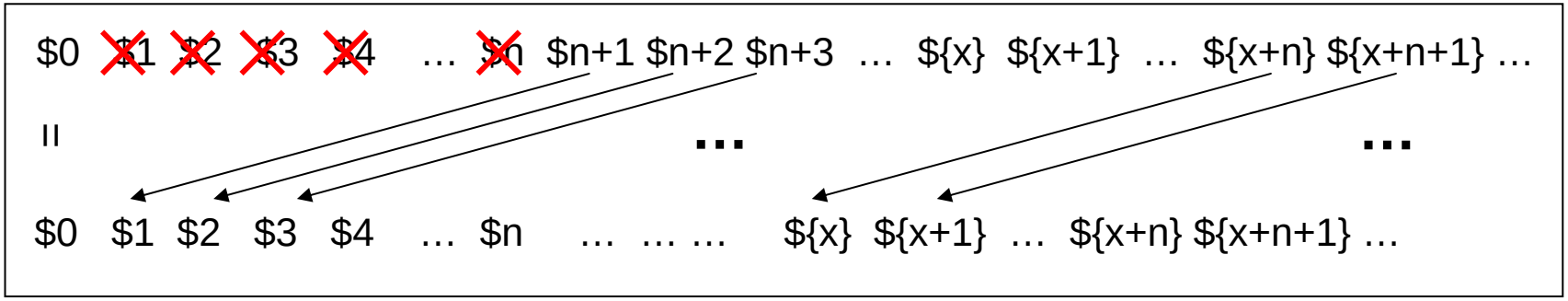
```
# Berechnung von Fibonacci-Zahlen
# $1 enthält die Zahl der zu berechnenden Feldelemente
# Laufvariable und Fibonacci-Zahlen als Integer deklarieren
integer i=2 fib
(( fib[0] = 1 )) ; (( fib[1] =1))

# Berechnung des Feldinhalts
while (( i < $1 ))
do
   (( fib[i]=fib[i-1]+fib[i-2] ))
   (( i+=1 ))
done

# Ausgabe
for el in ${fib[@]}
do
   print $el
done
```

Positionsparameterbelegung ändern: **`shift`**-Kommando

- Mittels des Kommandos

  `shift` *[n]*

  wird jedem Positionsparameter x (x>0) der Wert des Positionsparameters x+*n* zugeordnet (Links-Shift).



HINWEISE:

- Damit „verschwinden“ (unwiderbringlich!!) die bisherigen Belegungen von $1, $2, …, $n.
- $# reduziert sich um den Wert *n* und auch $@ und $* werden entsprechend aktualisiert.

Positionsparameterbelegung ändern: **`shift`**-Kommando

```
# Shiftvalue zufällig bestimmen (zwischen 1 und 4)
integer shiftval
(( shiftval=$RANDOM%4+1 ))

print Es wird jeder $shiftval-te übergebene Positionsparameter ausgegeben
print Dabei wird lediglich '$1' verwendet und das Kdo "'shift'"
print Sie haben $# Argumente übergeben
count=1
while [[ $# -gt 0 ]]
do
        (( count < 10 )) && print '$'$count: $1 \
                         || print '${'$count'}:' $1
        (( count=count+shiftval))
        shift shiftval            # damit verkleinert sich $# um $shiftval
done
```

Positionsparameterbelegung ändern: **`shift`**-Kommando

- Standardverarbeitung von Optionen

```
while [[ $1 = -* ] ; do
   case $1 in
      -a) Bearbeitung Option –a ;;
      -b) Bearbeitung Option –b ;;
      -c) Bearbeitung Option –c ;;
          arg_opt_c=$2
          shift ;;
       -d) Bearbeitung Option –d ;;
      *)  print “usage: $0 [-a] [-b] [-c c_opt_arg] args ...“
   esac
   shift;;
done
```

Positionsparameter neu belegen: **`set`**-Kommando

- Mittels des Kommandos

  `set` *wort2 wort2 wort3 wort4 wort5 …*

  können den Positionsparametern (neue) Werte zugewiesen werden.

  HINWEIS: die Zuweisung erfolgt *nach* Durchführung der Parameter- und Kommandosubstitution sowie Dateinamenexpandierung für *wort1, wort2, …*

  Gleichzeitig werden die Variablenwerte $#, $* und $@ aktualisiert.

  $0 bleibt unverändert!

- Löschen der Positionsparameter:

  `set --`

- Sortierung gemäß ASCII-Code

  `set -s` *wort2 wort2 wort3 wort4 wort5 …*

- BEACHTE: erhält **`set`** kein Argument (nach Subst. und Exp.), dann werden alle Variablenbelegungen ausgegeben.

Positionsparameter neu belegen: **`set`**-Kommando

```
$ ls
geraldine  gerd  gerhard  hannelore  jutta
$ ls
$ set $(ls)            # Neu-Belegung der Positionsparameter
$ print $1 --- $4
geraldine --- hannelore
$
$                      # Umspeichern in Array zum individuellen
$ integer i=0          # Zugriff
$ while (($#>0))
>do
>array[i]=$1
>((i+=1)); shift
>done
$
```

Arrays komplett belegen: **`set`**-Kommando

- Die Korn-Shell erlaubt auch eigene Arrays zu belegen:

```
$ ls
geraldine  gerd  gerhard  hannelore  jutta
$ set –A array $(ls)                # Belegung des Arrays array
$ print ${array[1]} --- ${array[4]}
geraldine --- hannelore
$ print ${#array[*])
5
$
```

- Allgemein lautet der Aufruf:
  `set –A` *arrayname werteliste*

- <u>weitere Beipiele:</u>
  Umspeichern von Arrays:
  `set –A array2 “${array1[@]}“`

Zufallszahlen: die Variable **RANDOM**

- Die Korn-Shell kann Zufallszahlen im Bereich zwischen 0 und 32767 generieren.
- Die aktuelle Zufallszahl steht in der `integer`-Variablen RANDOM
  Ihr kann mittels Zuweisung ein neuer Startwert zugewiesen werden.
- Bei jedem Zugriff mit $RANDOM erhält man eine neue Zahl der Zufallszahlenfolge zurück.

Zufallszahlen: Beispiel

```
RANDOM=$$
(( $# != 2 && $# != 0 )) \
   && { print "usage:\t\t$0  [anz_losungen anz_kugeln]";
        print "default:\tanz_losungen=6 anz_kugeln=49";
        exit 1; }
typeset -i anz_losungen=${1:-6} anz_kugeln=${2:-49} zahl[$anz_kugeln] i=1 z
print "\n=== Auswahl $anz_losungen aus $anz_kugeln ===\n"
while ((i<=$anz_losungen))
do
   ((z=$RANDOM % $anz_kugeln +1))
   while ((zahl[z]==1))
   do
      ((z=$RANDOM % $anz_kugeln +1))
   done
   print $i". Zahl:\t"$z
   zahl[z]=1
   ((i+=1))
done
```

Modifizierte Variante des Herold-Skripts „lotto“

Manipulation des Standardtrennsymbols: Variable **IFS**

- Die Variable **IFS** (Internal Field Seperator) enthält Trennsymbole, welche genutzt werden, um Zeileninhalte in Felder zu strukturieren.
- `IFS` enthält eine Folge von Zeichen, von denen jedes als Trennsymbol betrachtet wird. Standardmäßig ist das die Folge ‚BlankTabNewline'
- `IFS` kann jederzeit neu gesetzt werden

  (es empfiehlt sich den alten Inhalt in einer Variablen zu merken)

  Standardbelegung: `IFS=" \t\n"`

  Beachte Quoting beim Umspeichern: `OLDIFS="$IFS"`

Manipulation des Standardtrennsymbols: Variable **IFS**

- Der Inhalt von IFS wirkt sich aus auf
  - set und read
  - Resultat der Parameterexpansion
  - Resultat der Kommandosubstitution

  Bsp: Einlesen der Zeilen einer Datei datei in das Array array3:

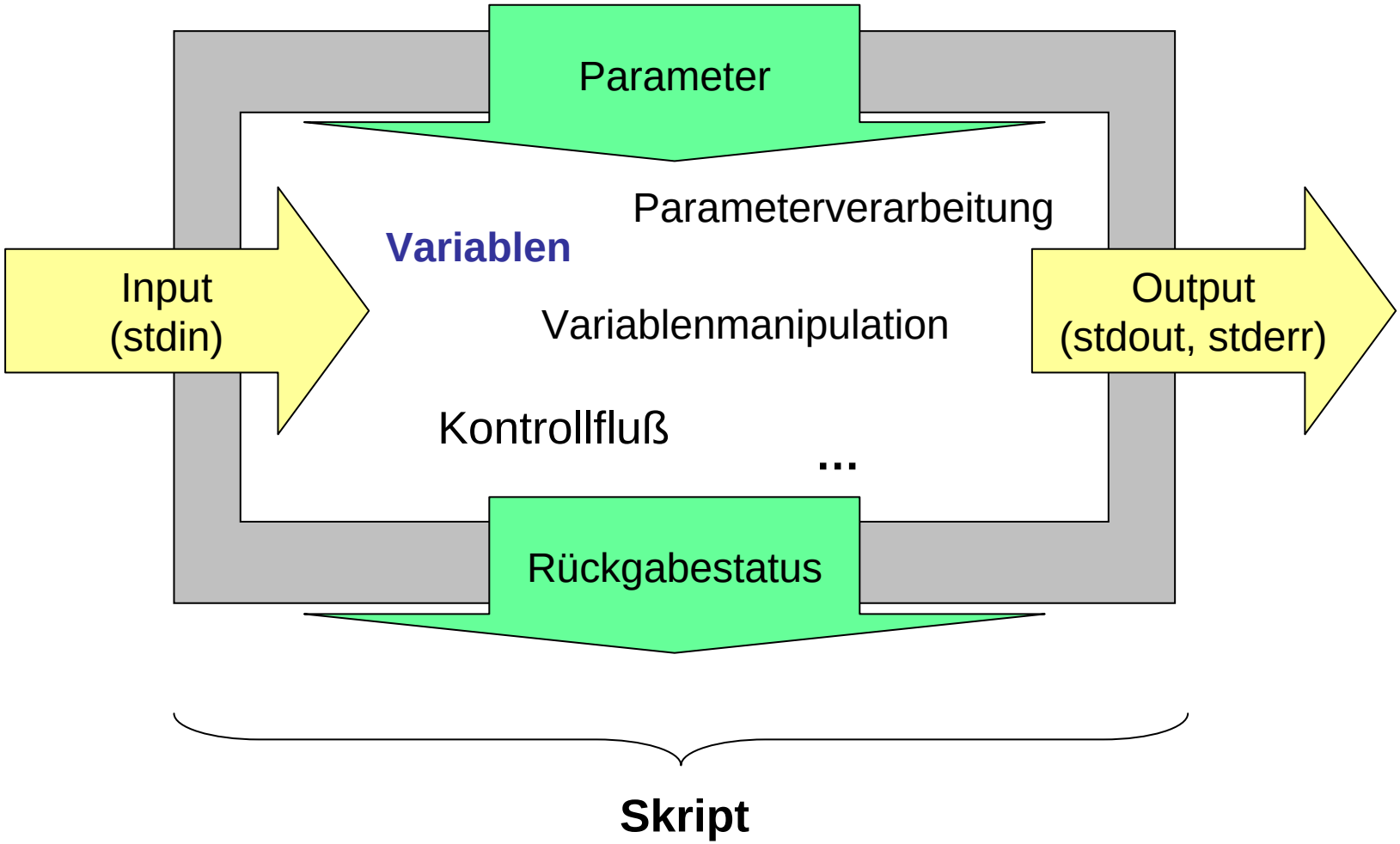
  ```
  >$ OLDIFS="$IFS"; IFS="\n"
  >$ set –A array3 $(< datei)
  >$ IFS="$OLDIFS"; unset OLDIFS
  ```

  Bsp: Ausgabe der Positionsparameter

  (In $* werden die Positionsparameter mittels des ersten Zeichen aus $IFS verbunden)

  ```
  >$ set a b c; IFS=:
  >$ print "$*"
  a:b:c
  ```

Wie geht es weiter?



Variablen: formatierende Attribute – das Kdo **`typeset`**

- Die Korn-Shell erlaubt, wie schon am Beispiel der Integer-Variablen gesehen, Variablen zu typisieren. Dies erlaubt das Kommando

  `typeset` [*optionen*] [*variable*[**=***wert*]…]

Option	**Bedeutung:**
`-u`	$*variable* ist großbuchstabig
`-l`	$*variable* ist kleinbuchstabig
`-i`*n*	$*variable* ist ganze Zahl zur Basis *n*
`-L`[*n*]	$*variable* ist linksbündig und *n* Zeichen breit. Ist ein zugewiesener Wert kürzer als *n*, so wird rechts mit Leerzeichen aufgefüllt. Ansonsten wird auf *n* Zeichen abgeschnitten. Ist *n* nicht gegeben, so wird es auf Basis der ersten Zuweisung an *variable* festgelegt.
`-LZ`[*n*]	Wie `-L`, aber zusätzlich Entfernung führender Nullen
`-R`[*n*]	Analog zu `-L`[*n*], jedoch rechtsbündige Justierung.
`-RZ`[*n*]	Wie `-R`, aber zusätzlich Auffüllen mit führenden Nullen
`-r`	Variable ist read only; die Zuweisung eines Wertes führt zu einer Fehlermeldung: `ksh: `*variable*` : is read only`

Variablen: formatierende Attribute – das Kdo **`typeset`**

- Ausgabe aller Variablen, welche mit Option *option* deklariert wurden:
  `typeset` *option*

- Ausgabe aller attributierten Variablen:
  `typeset`

HINWEIS:

- Positionsparameter sind read-only.
  `${1:=27}` führt zu entsprechender Fehlermeldung!

Variablen: formatierende Attribute – Beispiele

```
$ typeset -u uc
$ typeset -l lc
$ uc="gjhGHJGHGHJlköojUUIUI"
$ print $uc
GJHGHJGHGHJLKöOJUUIUI
$ lc="gjhGHJGHGHJlköojUUIUI"
$ print $lc
Gjhghjghghjlköojuuiui
```

```
$ typeset -i2 binaer
$ typeset -i8 oktal
$ typeset -i16 hexa
$ binaer=13
$ oktal=13
$ hexa=13
$ print $binaer $oktal $hexa
2#1101 8#15 16#d
```

```
$ typeset -L40 lform
$ typeset -LZ40 lformLZ
$ typeset -R40 rform
$ lform="   thomas fuhr  "
$ lformLZ="   thomas fuhr  "
$ rform="   thomas fuhr  "
$ print :"$lform":
:thomas fuhr                                :
$ print :"$lformLZ":
:thomas fuhr                                :
$ print ":$rform:"
:                                thomas fuhr:
$ typeset -LZ20 number=00007890
$ print ":$number:"
:7890                 :
$ print ${#number}
20
$ typeset -r number
$ number=003647300
ksh: number: is read only
```

Korn-Shell-Skripte: aktuelle Unbequemlichkeiten

- Ausgaben aktuell auf einen Kanal (stdout); das ist ggf. zu pauschal!
  - Differenziertere **Ein-Ausgabeumlenkung**

- Ausgabe von Menues zur Abfrage von Benutzerauswahl mühsam!
  Korn-Shell unterstützt dies mittels

  - **`select`**-Kommando

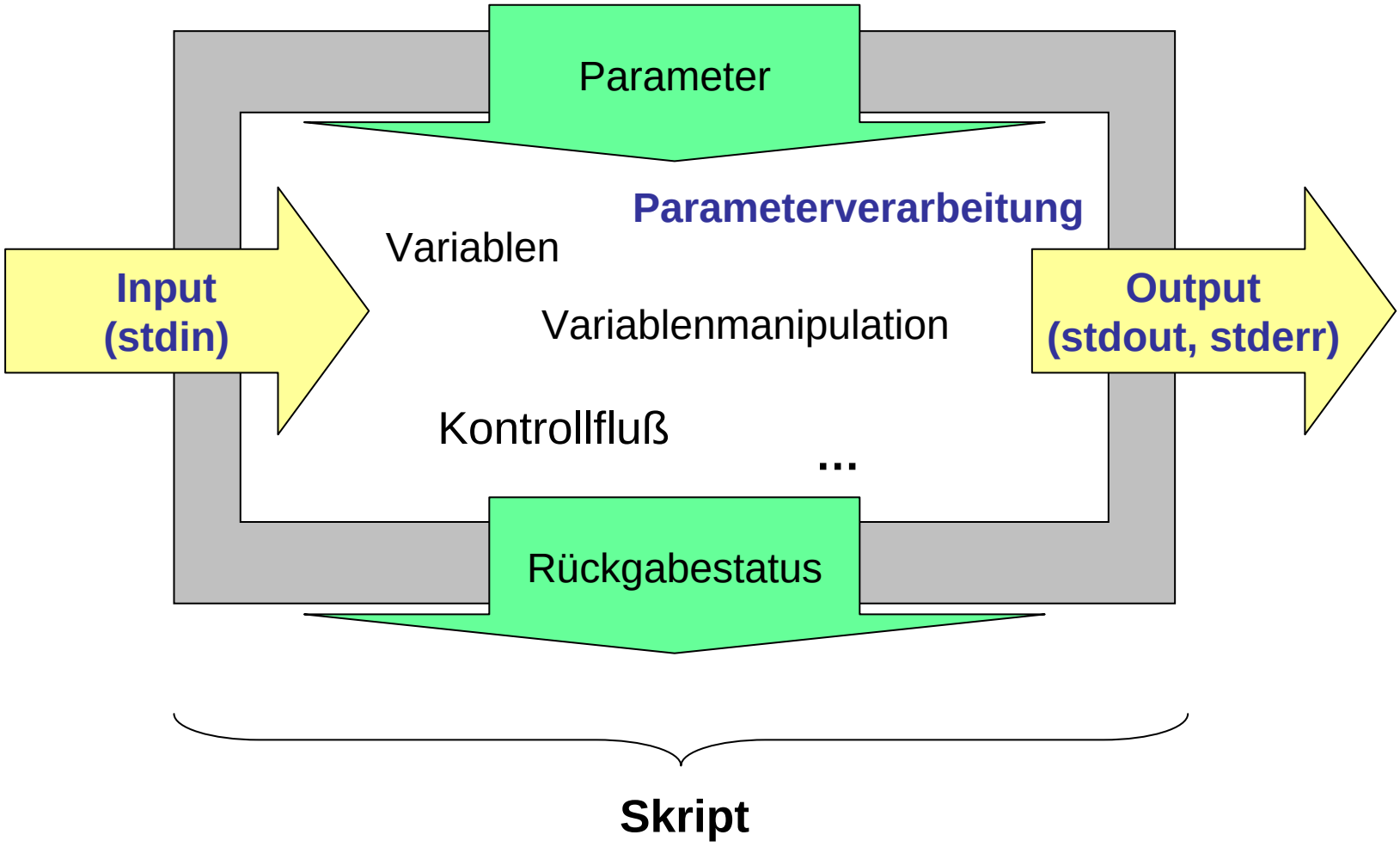
- Analyse der Kommandozeilenstruktur ggf. aufwendig:

  kommando [*option* [*optionsargument*] …] [*argument* …]

  - **`getopts`**-Kommando

- Skripten bisher nicht gut in sich strukturiert
  - **Funktionen**

Wie geht es weiter?



Ein-Ausgabeumlenkung

- Als Standardkanäle für die Ein/Ausgabe sind unter Unix bekannt:
  - Standardeingabe (`stdin`) Tastatur
  - Standardausgabe (`stdout`) Bildschirm
  - Standardfehlerausgabe (`stderr`) Bildschirm

- Z.B.:
  - liest `read` von `stdin`
  - schreibt `print` nach `stdout` und
  - gehen Fehlermeldungen wie „`cat: /home/fuhr ist ein Verzeichnis`“ nach `stderr`

- Typischerweise möchte man aus Dateien lesen, in neue oder bestehende Dateien schreiben, die eigentliche Ergebnisinformation und Fehlermeldungen in getrennte Dateien schreiben.

  ➢ **Umlenkung der Ein- und Ausgabe**

Ein-Ausgabeumlenkung -- Standard

- **Umlenkung `stdin`**

  < *dateiname* # Lesen aus Datei *dateiname* statt von Standardeingabe

- **Umlenkung `stdout`**

  > *dateiname* # Schreiben in Datei *dateiname* statt nach Standardausgabe
  # Datei *dateiname* wird überschrieben, sofern existent
  # wenn ksh-Option `noclobber` gesetzt
  # Fehlermeldung, falls *dateiname* bereits existiert, kein
  # Überschreiben

  >| *dateiname* # Schreiben in Datei dateiname statt nach Standardausgabe
  # *dateiname* wird überschrieben bzw. neu erzeugt

  >> *dateiname* # Schreiben in Datei *dateiname* statt nach Standardausgabe
  # Ausgabe wird an Inhalt von Datei *dateiname* angehängt

- In allen Fällen wird bei Nichtexistenz der Datei *dateiname* diese neu erzeugt.
- Offen: Wie schreibe ich die eigentliche Ergebnisinformation und Fehlermeldungen in getrennte Dateien?

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Eine Umlenkung der Einausgabe ist auch unter Verwendung von Filedeskriptoren möglich. Ein Deskriptor ist/wird immer mit einer Datei verknüpft.
- Standardein und –aus-, bzw. fehlerausgabe haben die feste zugeordnete **Standard File-Deskriptoren: 0** (stdin)**, 1** (stdout) und **2** (stderr)
- Diese Deskriptoren können genutzt werden, um Fehlerausgaben und Ergebnisausgaben gezielt auf stderr bzw. stdout zu tätigen:

  Explizites Schreiben nach stdout:

  ```
  print „Das ist mein Ergebnis“ >&1
  ```

  Explizites Schreiben nach sterr:

  ```
  print „Fehlermeldung“ >&2
  ```

- Andererseits kann man die Ausgaben eines Skripts getrennt umlenken:

  ```
  cat Datei 1>neueDatei 2>fehlerDatei
  ```

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Der Benutzer kann die Deskriptoren **3-9** selbst mit Dateien seiner Wahl verknüpfen (**benutzerdefinierte Filedeskriptoren**)
- Jeder Dateideskriptor *fd* kann „dauerhaft“ mittels des **exec**-Kommandos mit einer Datei *dateiname* verknüpft werden (entspricht öffnen einer Datei mit Deskriptor *fd*)

  exec *fd***<** *dateiname* # *fd* verknüpft mit lesendem Zugriff auf *dateiname*

  exec *fd***>** *dateiname* # *fd* verknüpft mit (über)schreibendem Zugriff auf *dateiname*

  exec *fd***>>** *dateiname* # *fd* verknüpft mit schreibendem Zugriff auf *dateiname*
  # „append-Modus“

  exec *fd***<>** *dateiname* # *fd* verknüpft mit Lese- *und* Schreibzugriff auf *dateiname*
- Geöffnete Dateien (Deskriptoren), werden geschlossen mittels
  exec *fd***<&-**
- HINWEIS: zwischen *fd* und Umlenkzeichen dürfen keine Leerzeichen stehen!
- Bsp.:

```
exec 5<infodatei                exec 7>meine_ausgaben
read –u5 zeile1                 print –u7 ausgabe1
read –u5 zeile2                 print –u7 ausgabe2
exec 5<&-                       exec 7<&-
```

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Dateien können per
  - Namen mittels *dateiname* (bekannter Fall …) oder per
  - Dateideskriptor mittels **&***fd*

  angesprochen werden.

- **Allgemeinere Umlenkungs-Ausdrücke:**

  *fd***<** *dateiname* # lenkt *fd* auf *dateiname* um
  *fd1***<**&*fd2* # lenkt *fd1* auf *fd2* um

  *fd***>** *dateiname* # lenkt *fd* auf *dateiname* um
  *fd1***>**&*fd2* # lenkt *fd1* auf *fd2* um
  # VORSICHT: war *fd2* per append-Modus geöffnet, so wird
  # dieser durch diese Umlenkung ignoriert.

  VORSICHT: keine Leerzeichen zwischen *fd*'s, Umlenkungssymbol und &

- Verwendung von Dateideskriptoren mittels **`read`**-/ **`print`**- Kommando:
  `read –u`*`fd`* **`print -u`*`fd`***

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Problem: `rm` erwartet Antwort auf `stdin`. Da Umlenkung auf `$infile` aktiv, liest `rm` im einfachen Fall („`current_in`“) ebenfalls von `stdin`.
- Lösung1: `rm`-Kommando liest explizit von Filedeskriptor 1.

```
infile=$2
outfile=${infile}.toberemoved

while read zeile
do
   print "Zeile gelesen: $zeile"
   if [[ $1 = "current_in" ]]
   then
      rm -i $outfile
   else # [[ $1 = "stdin" ]] = true
      rm -i $outfile <&1
   fi
   #exit 1
done <$infile
```

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Problem: `rm` erwartet Antwort auf `stdin`. Da Umlenkung auf `$infile` aktiv, liest `rm` im einfachen Fall („`current_in`“) ebenfalls von `stdin`.
- Lösung2: Eingabedatei $2 den Filedeskriptor 4 mit Lesezugriff zuordnen und `read` aus Datei lesen lassen, welche durch Filedeskriptor 4 bezeichnet wird.

```
exec 4<$2  # user-defined-File-Deskriptor für Eingabefile
           # statt: infile=$2
outfile=${2}.toberemoved

while read -u4 zeile  # von File mit Deskriptor 4 lesen
do
   print "Zeile gelesen: $zeile"
   if [[ $1 = "current_in" ]]
   then
      rm -i $outfile
   else # [[ $1 = "stdin" ]] = true
      rm -i $outfile <&1
   fi
   exit 1
done
```

Ein-Ausgabeumlenkung mittels Filedeskriptoren

- Bei mehreren Umlenkungen in einer Kommandozeile ist sorgsam vorzugehen!

- getrennte Abspeicherung der Ausgaben des Skripts `trenneAusgabe`

  `trenneAusgabe 1>datei1 2>datei2`

- Fehlerhafter Versuch einer gebündelten Abspeicherung aller Ausgaben des Skripts `trenneAusgabe` in `datei2`

  `trenneAusgabe 1>&2 2>datei2`

  Nur Fehlerausgaben landen in datei2, der Rest auf dem Bildschirm !

- Erfolgreicher Versuch einer gebündelten Abspeicherung aller Ausgaben des Skripts `trenneAusgabe` in `datei2`

  `trenneAusgabe 2>datei2 1>&2`

Ein-Ausgabeumlenkung – Standard vs. Dateideskriptoren

- **Umlenkung `stdin` auf Datei**

  < *dateiname* = 0**<** *dateiname*

- **Umlenkung `stdout` auf Datei**

  > *dateiname* = 1**>** *dateiname*

  >| *dateiname* = 1**>|** *dateiname*

  >> *dateiname* = 1**>>** *dateiname*

- **Umlenkung `stderr` auf Datei**

  2**>** *dateiname*

  2**>|** *dateiname*

  2**>>** *dateiname*

- **explizites Schreiben nach `stdout`**: *kdo* >&1
- **explizites Schreiben nach `stderr`**: *kdo* >&2

Korn-Shell-Skripte: aktuelle Unbequemlichkeiten

- Ausgaben aktuell auf einen Kanal (stdout); das ist ggf. zu pauschal!
  - Differenziertere Ein-Ausgabeumlenkung

- Ausgabe von Menues zur Abfrage von Benutzerauswahl mühsam!
  Korn-Shell unterstützt dies mittels

  - **`select`-Kommando**

- Analyse der Kommandozeilenstruktur ggf. aufwendig:

  kommando [*option* [*optionsargument*] …] [*argument* …]

  - **`getopts`**-Kommando

- Skripten bisher nicht gut in sich strukturiert
  - Funktionen

Kontrollstrukturen: select-Kommando

- Das **select**-Kommando dient der leichten Erstellung von Auswahlmenues.
- *wort1, wort2, …, wortn* sind die Auswahlalternativen
- Die *kdoliste* enthält die Anweisungen, welche festlegen, was abhängig vom jeweils gewählten *wortx,* welches in $*variable* hinterlegt ist, zu tun ist.

select *variable* [**in** *wort1 wort2 … wortn*]
do
  kdoliste
done

Nach: Herold, S.326

- *kdoliste* ist deshalb häufig eine if-then-else-Kaskade oder durch eine case-Anweisung realisiert.
- Fehlt der optionale Teil, d.h. es wird keine Liste angegeben, so wird implizit die Liste der Positionsparameter benutzt (analog zur for-Anweisung).
- Der Exit-Status des gesamten `select`-Kommandos ist gleich dem Status der zuletzt ausgeführten Anweisung. Wurde keine Anweisung ausgeführt, gilt Exit-Status=0.
- Die Auswahlnummer wird in der Variablen `REPLY` zurückgeliefert

Korn-Shell-Skripte: aktuelle Unbequemlichkeiten

- Ausgaben aktuell auf einen Kanal (stdout); das ist ggf. zu pauschal!
  - Differenziertere Ein-Ausgabeumlenkung

- Ausgabe von Menues zur Abfrage von Benutzerauswahl mühsam!
  Korn-Shell unterstützt dies mittels

  - **`select`**-Kommando

- Analyse der Kommandozeilenstruktur ggf. aufwendig:

  kommando [*option* [*optionsargument*] …] [*argument* …]

  - **`getopts`-Kommando**

- Skripten bisher nicht gut in sich strukturiert
  - Funktionen

Verarbeitung von Optionen: **`getopts`**-Kommando

- Die Analyse der Eingabezeile ist unter Umständen sehr mühselig, wenn viele Optionen, welchen zum Teil auch Optionsargumente folgen dürfen, erlaubt sind.

  Insbesondere, da für Optionen i.allg. keine feste Reihenfolge vorgeschrieben wird und Optionen auch gebündelt werden dürfen.

- Korn-Shell-builtin-Kommando **`getopts`** parsed den Optionsteil von Eingabezeilen der üblichen UNIX-Kommando-Form

  kommando [*option* [*optionsargument*] …] [*argument* …]

  und extrahiert alle Optionen und Optionsparameter. Solange Optionen gefunden werden (egal ob korrekt oder fehlerhaft) wird ein Status = 0 zurückgegeben. Nach Erreichen der *argument*e wird Status ungleich 0 zurückgegeben.

  `getopts` wird deshalb typischer Weise eingebettet in eine `while`-Schleife genutzt.

Verarbeitung von Optionen: **`getopts`**-Kommando

- `getopts` toleriert
  - die Bündelung von Optionen
    Bsp: `ksh -xvn`
  - die Bindung von Optionsargumenten direkt an den Optionsstring
    Bsp: `head -n5; cut -d:`
  - nur in manchen ksh-Varianten (z.B. auf `unixcluster`: Compac Tru64-Unix) die Verwendung sowohl von –o wie auch +o zur Ermöglichung des Ein- und Ausschaltens von Optionen
    Bsp: `set +x -v; cut -d:`
- `getopts` detektiert fehlerhafte Optionen und ermöglicht eine eigene Fehlerbehandlung
- `getopts` stellt die jeweils gefundene Option samt Optionsargument in Variablen bereit
- `getopts` detektiert fehlerhafte Optionen und stellt diese per Variable `OPTARG` bereit. Dies ermöglicht eine eigene Fehlerbehandlung.
- Nach Abschluß des Parsens, stellt `getopts` den Index des Positionsparameters in der Variable `OPTIND` bereit, welcher unmittelbar dem Optionsteil des Kommandos folgt

Verarbeitung von Optionen: **`getopts`**-Kommando

- Aufruf:
  `getopts` *opt_specification optvar*

- *opt_specification* ist gleich
  [**`:`**][[*c*|*c***`:`**] …] mit folgender Bedeutung:
  - **`:`** Keine Shellfehlerausgabe auf stderr, bei detektierten Fehlern
    muß am Anfang der Spezifikation stehen!
  - *c* *c* darf in der Form –c +c oder als Teil von Optionsbündelungen akzeptiert werden
  - *c***`:`** *c* muß ein Optionsargument folgen

- `getopts` liefert in der Variable *optvar*, die jeweils gefundene Option in der Form
  - *c* falls –*c* im Input steht, oder *c* Teil einer mittels – eingeleiteten Options-Bündelung ist.
    Ist für *c* ein Optionsargument spezifiziert, so steht dieses in `$OPTARG`
  - **+***c* falls +*c* im Input steht, oder *c* Teil einer mittels + eingeleiteten Options-Bündelung ist
  - **?** falls eine falsche Option *x* detektiert wurde. Es gilt dann `$OPTARG` =*x*.
  - **`:`** falls zur spezifizierten Option *c* das spezifizierte Optionsargument fehlt.
    Es gilt dann `$OPTARG` = *c*.

Verarbeitung von Optionen: **`getopts`**-Kommando

- Muster der `getopts`-Verwendung:

```
while getopts :of:p  optvar
do
     case $optvar   in
        o)   … ;;
        +o)  … ;;
        f)   … $OPTARG … ;;          # $OPTARG gleich Optionsargument
        +f)  … $OPTARG … ;;          # $OPTARG gleich Optionsargument
        p)   … ;;
        +p)  … ;;
        \?)  … $OPTARG … ;;          # $OPTARG unbekannte Option
        :)   … $OPTARG … ;;          # $OPTARG Option deren Argument fehlt
     esac
done
shift OPTIND-1  # jetzt nur noch alle Argumente in $1, $2, $* usw.
```

Verarbeitung von Optionen: **`getopts`**-Kommando [ksh93]

```
#! /usr/bin/ksh
# shell version of env command
case $(getopts '[-]' opt '--???man' 2>&1) in
version=[0-9]*)
    usage=$'[-?@(#)env (AT&T Labs Research) 1999-05-20\n]
        [-author?David Korn <dgk@research.att.com>]
        [-license?http://www.research.att.com/sw/tools/reuse]
        [+NAME?env - set environment for command invocation]
        [+DESCRIPTION?\benv\b modifies the current environment according to the \aname\a\b=\b\avalue\a arguments, and then
                invokes \acommand\a with the modified environment.]
      [+?If \acommand\a is not specified, the resulting environment is written to standard output quoted as required for
                reading by the \bsh\b.]
      [i:ignore-environment?Invoke \acommand\a with the exact environment specified by the \aname\a\b=\b\avalue\a arguments; inherited
                environment variables are ignored.  As an obsolete feature, \b-\b by itself can be specified instead of \b-i\b.]
      [u:unset]:[name?Unset the environment variable \aname\a if it was in the environment.  This option can be repeated to unset
                additional variables.]
      [name=value]... [command ...]

      [+EXIT STATUS?If \acommand\a is invoked, the exit status of \benv\b
                will be that of \acommand\a.  Otherwise, it will be one of
                the following:]{
               [+0?\benv\b completed successfully.]
               [+126?\acommand\a was found but could not be invoked.]
               [+127?\acommand\a could not be found.]
      }
        [+SEE ALSO?\bsh\b(1), \bexport\b(1)]
    '
    ;;
*)
    usage='iu:[name] [name=value]... [command ...]'
    ;;
esac
```

Download: www.korn-shell.com

Siehe auch ähnliches Beispiel phaser4.ksh aus „Learning the Korn Shell“, Rosenblatt&Robbins, OReilly

Verarbeitung von Optionen: **`getopts`**-Kommando [ksh93]

```
clear=
while getopts  "$usage" var
do    case $var in
      i)       clear=1;;
      u)       command unset $OPTARG 2> /dev/null;;
      esac
done
#[[ $var == "" ]] || exit 1
shift $((OPTIND-1))
if    [[ $1 == - ]]  # obsolete form
then  clear=1
      shift
fi
if    [[ $clear == 1 ]]
then  typeset +x $(typeset +x)
fi
while true
do    case $1 in
      *=*)     export "$1";;
      *) break;;
      esac
      shift
done
if    (( $# >0 ))
then  exec "$@"
else  export
      exit 0
fi
```

Download: www.korn-shell.com

Korn-Shell-Skripte: aktuelle Unbequemlichkeiten

- Ausgaben aktuell auf einen Kanal (stdout); das ist ggf. zu pauschal!
  - ➢ Differenziertere Ein-Ausgabeumlenkung
- Ausgabe von Menues zur Abfrage von Benutzerauswahl mühsam!
  Korn-Shell unterstützt dies mittels
  - ➢ **`select`**-Kommando
- Analyse der Kommandozeilenstruktur ggf. aufwendig:

  kommando [*option* [*optionsargument*] …] [*argument* …]

  - ➢ **`getopts`**-Kommando
- Skripten bisher nicht gut in sich strukturiert
  - ➢ **Funktionen**

Funktionen: Definition und Aufruf

- Auch die Korn-Shell erlaubt die Definition von Funktionen

  `function` *functionname* **{** *kdoliste***; }**

  Alternativ ist auch die Bourne Shell Syntax möglich: *functioname* **() {** *kdoliste***; }**. Da es aber semantische Abweichungen gibt, sollte obide Definitionsform auf jeden Fall immer dann genommen werden, wenn nicht in Bourne-Shell verfügbare Mechanismen innerhalb der Funktion genutzt werden.

  Für *functionname* gelten die für Datei- und Variablennamen üblichen Bezeichnerkonventionen

- Im Rahmen der Definition wird im Funktionskopf die Anzahl der Funktionsparameter nicht explizit deklariert (wie z.B. in C oder C++). Folgerichtig gibt es auch keine automatische Prüfung der Funktionsschnittstelle beim Aufruf. Ein Funktionsnutzer muß also wissen, wieviele Parameter in der kdoliste erwartet werden. Der korrekte Aufruf muß durch den Funktionsentwickler selbst geprüft werden.

- Aufrufschema:

  functionname arg1 arg2 arg3 arg4 …

Funktionen: Gültigkeitsbereich der Definition

- Eine Funktion ist zum Aufrufzeitpunkt automatisch in der Shell bekannt, in welcher sie vorab definiert wurde.
  - In Skripten, welche Funktionsdefinitionen enthalten, muß Funktionsdefinition immer *vor* Aufruf der Funktion im Skript stehen.
  - In externen Dateien stehende Funktionsdefinitionen können mittels des Kommandos `.` eingelesen werden.

    ```
    . dirtree
    ```

- Auflistung aller der Shell aktuell bekannten Funktionsdefinitionen:

  `functions` **`typeset -f`**

- Löschen einer Funktionsdefinition aus der aktuellen Shellumgebung

  `unset -f` *functionname*

Funktionen: Parameterübergabe und Rückgabewert

- Die Aufrufparameter von Funktionen werden analog zu denen eines Skripts in Form von Positionsparametern und den zugehörigen vordefinierten Variablen bereitgestellt:

`$0`	Funktionsname	(Abweichung zur Bourne-Shell)
`$1, $2, $3, …`	Funktionsparameter	
`$*, $@, $#`	wie für Skripte bekannt.	

- Die Positionsparameter etc. der Funktion überdecken die der umgebenden Shell. Diese sind erst nach Verlassen der Funktion wieder sichtbar

- Innerhalb einer Funktion können Positionsparameter (wie auch in Skripten) mittels `set` neu belegt werden. Dies hat keine Auswirkung auf die Positionsparameter der umgebenden Shell!

- Werterückgabe erlaubt der Aufruf

  `return` [*n*]

  D.h. es kann nur ein Status rückgeliefert werden. Hier gilt auch die für Skripte gültige Konvention: return-Wert 0 = erfolgreich, sonst fehlerhafter Funktionsablauf.

  Wird *n* nicht gesetzt so wird der aktuelle Wert von $? zurückgegeben.

Funktionen: Beispiel Positionsparameter

```
$ function scopetest
> {
> print $0 -- $1 -- $2 -- $3 -- $#
> set a b
> print $0 -- $1 -- $2 -- $3 -- $#
> }
$ print $0 -- $1 -- $2 -- $3 -- $#
/bin/ksh --  --  --  -- 0
$ set w x y z
$ print $0 -- $1 -- $2 -- $3 -- $#
/bin/ksh -- w -- x -- y -- 4
$ scopetest thom as fuhr
scopetest -- thom -- as -- fuhr -- 3
scopetest -- a -- b -- -- 2
$ print $0 -- $1 -- $2 -- $3 -- $#
/bin/ksh -- w -- x -- y -- 4
```

```
$ functions
function scopetest
{
print $0 -- $1 -- $2 -- $3 -- $#
set a b
print $0 -- $1 -- $2 -- $3 -- $#
}
$ unset –f scopetest
$ typeset –f
$
```

Funktionen: Gültigkeitsbereich der Definitionen - **`autoload`**

- Auch die Nutzung von Funktionen, deren Definition nicht explizit in der aktuellen Shell erfolgte, ist möglich

  – **Funktionsdatei** *functionname* enthält genau eine Funktionsdefinition der Funktion *functionname* und hat den Dateimodus ‘executable‘

  – Environmentvariable **`FPATH`** ist gesetzt und enthält Verzeichnis, in welchem die Datei *functionname* steht. (Das Bildungsprinzip für `FPATH` ist analog zu `PATH`)

  – Die extern definierte Funktion *functionname* muß in der aufrufenden Shell vor dem Aufruf durch die Anweisung

    `autoload` *functionname*

    bekannt gemacht werden. Die Funktionsdefinition wird dann (erst) zum Zeitpunkt des erstmaligen Funktionsaufrufs gemäß `FPATH` gesucht und geladen.

- HINWEIS: Bzgl. der Auflösung eines Kommandonamens als Funktionsname oder als Skriptname – sofern 2 solche Dateien selben Namens existieren – siehe Namensauflösung

Funktionen: Sonstiges

- Im Rahmen der Analyse der Kommandozeile haben Funktionsnamen in der `ksh` Vorrang vor Skriptnamen oder Namen anderer Executables, sofern die Funktion bereits ins Memory der Shell geladen wurde. D.h. in diesem Fall wird bei Namensgleichheit eine der `ksh` bekannte Funktion vorrangig vor dem gleichnamigen Skript/Executable ausgeführt

- Nach erstmaliger Ausführung einer Funktion bleibt diese im Memory der Shell. Änderungen an der Funktionsdatei werden damit für den Nutzer nicht automatisch nach der Änderung sichtbar. Um dies zu erreichen muß die Funktion zunächst aus dem Memory der Shell gelöscht werden:

  ```
  unset –f functionname
  ```

  Nach erneuter Ausführung wird die – nun geänderte -- Funktion wieder in das Memory der Shell geladen.

Funktionen: Gültigkeitsbereich von Variablen

- Korn-Shell-Funktionen laufen *innerhalb der aktuellen Shellumgebung* ab, nicht in einer Subshell.
  - Alle Variablen der umgebenden (= aufrufenden) Shell – einfache wie auch Environmentvariable -- sind innerhalb der Funktion lesbar *und* veränderbar!

    VORSICHT: Seiteneffekte!
  - Positionsparameter inkl. $*, $@, $# sind *immer* funktionsspezifisch gültig
  - Funktionslokale Variable können mittels

    `typeset` *varname*

    deklariert werden. Ist *varname* zur Bezeichnung einer in der umgebenden Shell bereits deklarierten Variable identisch, so wird diese durch die funktionslokale Variable für die Dauer der Funktionsdurchführung überdeckt (analog zu Positionsparametern).

    HINWEISE:
    - Damit ist *jede* attributierte Variable funktionslokal!
    - Insbesondere bei rekursiven Funktionen berücksichtigen

Funktionen: Beispiel Rekursion

Inhalt der Datei `recursionTest:`

```
function recursionTest {
   typeset localvar=$RANDOM
   ${depth=0}
   print "Akt. Rekursionstiefe: $depth"
   print "Akt. Wert localvar:   $localvar"
   if [[ depth -ne $1 ]]
   then
      (( depth+=1 ))
      recursionTest $1
      (( depth-=1 ))
   else
      print „\n“
      return
   fi
   print "Akt. Rekursionstiefe: $depth"
   print "Akt. Wert localvar:   $localvar"
   return
}
```

Fortsetzung:

```
$ export FPATH=.:$FPATH
$ autoload recursionTest
$ functions
recursionTest()
$
$ recursionTest 2
Akt. Rekursionstiefe: 0
Akt. Wert localvar:   10745
Akt. Rekursionstiefe: 1
Akt. Wert localvar:   7253
Akt. Rekursionstiefe: 2
Akt. Wert localvar:   28148

Akt. Rekursionstiefe: 1
Akt. Wert localvar:   7253
Akt. Rekursionstiefe: 0
Akt. Wert localvar:   10745
$
```

Trick 17: **`eval`**-Kommando

- Bzgl. der Themen Arrays und Positionsparameter stellt sich die Frage, ob man nicht mittels eines Zählers die Positionsparameter ansprechen kann.
- Man benötigt dann so etwas wie eine „Indirekte Adressierung“:
  `count=3; set a b c; print ${$count}` **funktioniert nicht!!**
- Die Expansion der Parameter findet nur einmal in der Kommandozeile statt.
- Das Kommando
  `eval` *kommando*
  führt die Parametersubstitution für *kommando* durch und führt dann *kommando* aus, wobei dann wiederum alle bekannten Substitutionsmechanismen greifen.
  Bsp:

```
$ year=‘$(date +%Y)‘
$ print “$year“
$(date +%Y)
$ eval print $year
2003
```

Trick 17: **`eval`**-Kommando

- Konstruktion und Ausführung einer Kommandozeile:

```
for option in ...
do
        case $option in
        save) out='> $datei_neu';;
        show) out '| more';;
        esac
done

...

eval sort $datei $out
```

Zeitmessung: **`time`**-Kommando

- Die Korn-Shell erlaubt eine Zeitmessung für durchgeführte Kommandos. Hiermit ist ein sehr nützliches Hilfsmittel zur Zeitoptimierung sowohl erstellter Skripten wie auch generell entwickelter unter Unix laufender Applikationen.

- Aufruf:
  `time` *kdoliste*

- Die durchgeführte Zeitmessung erfolgt bzgl. 3 Kategorien.
  - **`real`** die insgesamt benötigte (Uhr)zeit, sog. elapsed time
  - **`user`** die im Benutzermodus verbrauchte Zeit
  - **`sys`** die im Systemmodus verbrauchte Zeit
    (d.h. für die Nutzung von Systemaufrufen, z.B. Dateiin/output etc.)

  Alle Zeitangaben erfolgen in Minuten und Sekunden
- Werden im Aufruf E/A-Umlenkungen genutzt so beziehen sich diese auf die *kdoliste*

Makros: **`alias`**-Kommando

- Die Korn-Shell erlaubt die Bildung von Aliases, d.h. von Kurzschreibweisen für Kommandoaufrufe.
- Aufruf:
  `alias` `[-tx]` [*aliasname*[**=***wert*]]
- Bsp.:
  `alias` cdpro=‘cd /raid/user/hans/projekt; pwd‘
- Ohne Angabe von [*aliasname*[**=***wert*]] werden die aktuell bekannten Aliase angezeigt.
- Ohne Angabe von [**=***wert*] wird die aktuelle Belegung von Aliasname gezeigt
- Bei der Bearbeitung der Kommandozeile, werden durch die Korn-Shell gefundene Aliase (am Anfang der Zeile oder einem Alias nach white space folgend) als erstes ersetzt. Damit ist gewährleistet, daß Aliase die eigentlichen Kommandos „überdecken“

Makros: **`alias`**-Kommando

- Sinnvollerweise definiert man sich als Nutzer beliebte Kürzel in einer beim Shellstart eingelesenen Environment- oder Aliasdatei

- Mittels `-x` können Aliase Subshells bekannt gemacht werden („export“).

- Setzt man in der Kornshell die Option `set -o trackall` , dann werden automatisch Aliase für Kommandoaufrufe generiert, sog. ***tracked aliase***.

  Der Basisname des Kommandos wird auf den vollständigen Pfadnamen per Alias abgebildet. Damit wird erreicht, daß bei erneutem Kommandoaufruf keine Kommandosuche unter Verwendung der Variable **`$PATH`** notwendig ist.

- Die Korn-Shell bietet auch standardmäßig etliche vordefinierte Aliase an. Hierzu zählen u.a.:

  `autoload` `integer` `history` `r` `functions`

  `true` `false` (systemabhängig)

Namensauflösung -- Skripten, Funktionen, Aliase …

- Die `pdksh` versucht in folgender Reihenfolge ein eingegebenes Kommandowort *kdoname* aufzulösen:

1. Keywords (z.B. `if`, `for`, `function` …)
2. Aliase
3. Spezielle Build-In-Kommandos

```
.       :       break   builtin continue        eval    exec
exit    export  readonly        return          set     shift
times   trap    typeset unset
```

4. Funktionen
5. Reguläre Built-In-Kommandos

```
[       alias   bg      cd      command echo    false   fc
fg      getopts jobs    kill    let     print   pwd     read
test    true    ulimit  umask   unalias wait    whence
```

6. Skripte und ausführbare Programme

Namensauflösung -- Skripten, Funktionen, Aliase …

- Bezeichne hier *kdoname* eine Funktion oder ein Skript bzw. ausführbares Programm.
- Die Prüfung von Punkt 4 und 6 wird mit durch die Environment-Variablen `PATH` und `FPATH` sowie durch ggf. vorab getätigte `autoload`-Kommandos beeinflusst:
  - gibt es bereits die undefinierte Funktion *kdoname* (als Ergebnis eines früheren `autoload` *kdoname*), so sucht die ksh nach der vollständigen Definition der Funktion *kdoname* in allen mittels FPATH spezifizierten Verzeichnissen. Die erste gefundene Datei namens *kdoname* wird ausgeführt und geladen. Enthält diese Datei gar keine Funktionsdefinition erscheint die Meldung:

    `ksh: kdoname: function not defined by <pfad>/kdoname`

    Handelt es sich bei der gefundenen Datei um ein Skript namens *kdoname* und kommt es bei der Ausführung zur Ausführung eines `exit`-Kommandos so führt dies zum Verlassen der `ksh` in der man sich gerade befindet.
- Wurde der Name *kdoname* der `ksh` nicht per `autoload` als Funktionsname bekannt gemacht, so geht die Shell zunächst davon aus, daß es ein Skript/ausführbare Programm ist und sucht eine entsprechende Datei gemäß des Inhalts von `PATH`

  Ist diese Suche nicht erfolgreich, so sucht die `ksh` eine entsprechende Datei gemäß des Inhalts von `FPATH`. Findet sie eine, so wird diese als Funktion ausgeführt und geladen.

Namensauflösung -- **whence**-Kommando

- Mittels des Kommandos `whence` kann abgefragt werden, in welcher Weise ein Name durch die `ksh` aufgelöst wird:
- Aufruf:

  whence `-v` *kdoname*

- Bsp.:

  `whence -v test`

- Gibt an, ob das ausgeführte test-Kommando ein Built-in-Kommando, ein Alias, eine Funktion oder eine andere ausführbare Datei (Skript, ausführbares Programm) bezeichnet.

Rückblick: Shell-Vergleich

Funktionalität	sh	ksh	bash	csh	tcsh
Interaktive Kommandoeingabe:					
• **History-Mechanismus**		X	X	X	X
• **Directory-Stack**			X	X	X
• **CDPATH-Variable**	X	X	X	X	X
• **Alias-Mechanismus**		X	X	X	X
• **Alias-Argumente**				X	X
• **Kommandozeilen-Editing**		X	X		X
• **variabler Promptstring**		X	X	X	X
• **Spelling-Correction** (user-IDs, Kdos, Dat.namen)					X
• **Aliasnamen-Completion**			X		X
• **Kommandonamen-Completion**			X	X	X
• **Dateinamen-Completion**		X	X	X	X
• **Funktionsnamen-Completion**			X	X	X
• **Hostname-Completion, Variablennamen-Completion**			X		

Rückblick: Shell-Vergleich

Funktionalität	sh	ksh	bash	csh	tcsh
I/O:					
• **Ein-/Ausgabeumlenkung**	X	X	X	X	X
Expansionsmechanismen:					
• **Alias-Erkennung**		X	X	X	X
• **Kommandosubstitution**	X	X	X	X	X
• **Dateinamensubstitution**	X	X	X	X	X
• **Parametersubstitution**	X	X	X	X	X
• **Funktionsnamen-Erkennung**	X	X	X		

Rückblick: Shell-Vergleich

Funktionaltät	sh	ksh	bash	csh	tcsh
Kontrollstrukturen:					
• **bedingte Verzweigung / computed goto**	X	X	X	X	X
• **Schleifenkonstrukte**	X	X	X	X	X
Variablen:					
• **Arrays**		X	X		X
• **formatierte Variablen**		X			
• **Zufallszahlen**		X	X		
• **readonly-Variablen**	X	X	X		X
Operatoren:					
• **arithmetische Operatoren**		X	X	X	X
• **stringmanipulierende Operatoren**		X	X		
Funktionen:					
• **Funktionsdefinitionen**	X	X	X		
• **funktionslokale Variablen**		X	X		
• **Funktionen-Autoload**		X			

Rückblick: Shell-Vergleich

Funktionaltät	sh	ksh	bash	csh	tcsh
Prozesse:					
• **wait**	X	X	X	X	X
• **Koprozesse**		X			
• **Jobkontrolle**		X	X	X	X
Signale:					
• **Signalbehandlung**	X	X	X	X	X
• **Signalnamen**		X	X	X	X

Nützliche UNIX-Tools (Teil 2)

- Bisher kennengelernte Unix-Tools verfügten nicht über die Möglichkeit einer patternbezogenen Suche, Filterung bzw. Ersetzung

- Die `grep`-Tools sind ein Toolset zur Selektion von Zeilen, welche spezifizierten pattern genügen.

- `sed` ist ein Streameditor, welcher sowohl zur Selektion von Zeilen, wie auch zur patternorientierten Ersetzung von Teilstrings in Zeilen genutzt werden kann.

- `awk` ist eine Skriptsprache, welche über volle Programmierfunktionalität besitzt. Diese Sprache ermöglicht in besonderer Weise die Formulierung patternbezogener algorithmischer Bearbeitung von Textdateien.

`grep`-Tools

- „**g**lobally search for a **r**egular **e**xpression and **p**rint the result“
- Aufrufstruktur:
  `grep` [*options*] [*pattern*|**`-e`** *pattern*|**`-f`** *patternfile*] [*file …*]
- Varianten:

`fgrep`	(gleich **`grep -F`**)	fixed pattern
`grep`		basic pattern
`egrep`	(gleich **`grep -E`**)	extended pattern

- HINWEIS:
  Da in *pattern* üblicherweise Zeichen vorkommen, welche Metazeichen der Kornshell sind, und damit zu einer entsprechenden – ungewollten – Dateinamensubstitutionen führen können, empfiehlt es sich *pattern* immer mittels ‘ zu quoten!

- Mit der mächtigsten Variante `egrep` bzw. `grep -E` und den egrep-*pattern* kann
  - alles beschrieben werden, was auch mittels der ksh-*pattern* möglich war. Darüberhinaus sind
  - Iterationen zahlenmässig spezifizierbar und
  - Rückwärtsreferenzen möglich.

grep-Tools: Verarbeitungsprinzip (Grundidee)

grep pattern datei

```
# nur Prinzip, da natuerlich die Syntax der regulären
# Ausdrücke unterschiedlich und grep Verankerungen an
# Zeilen bzw. Wortgrenzen erlaubt!

patternstring=“$1“
dateiname=“$2“

while read zeile
do
   if [[ $zeile = *patternstring* ]]
   then
      print “$zeile“
   fi
done < $dateiname
```

egrep – erweiterte reguläre Ausdrücke

- Beschreibung *einzelner* Zeichen

Meta-zeichen	**Bedeutung**	**Korn-Shell**
.	„eine beliebiges Zeichen“	**?**
[…]	„eines der in […] angegebenen Zeichen“ **(Zeichenklasse)** Innerhalb von […] können Zeichen als Auflistung $c_1c_2c_3$…, Bereich c_1-c_2 oder als beliebige Kombination von Auflistung und Bereichen angegeben werden.	**[…]**
[^...]	„ein beliebiges Zeichen, welches nicht in […] vorkommt“	**[!...]**
*****c*	Zeichen *c,* falls *c* ein Metazeichen und das Zeichenpaar *c* keine Meta-Bedeutung hat	*****c*

- WICHTIG: Innerhalb von Zeichenklassen verlieren alle Zeichen ausser [,], ^, \ und – ihre Sonderbedeutung und können einfach mit aufgelistet werden.

- [,], ^ und – können wie folgt in Zeichenklassen integriert werden:
  - [,] und – als erstes Zeichen innerhalb von […]
  - ^ und \ muß das Escapezeichen \ vorangestellt werden

egrep – erweiterte reguläre Ausdrücke

- Beispiele

  > Ziffer:
  `egrep '[0-9]'`

  > kein Buchstabe:
  `egrep '[^A-Za-z]'`

  > eine 3 Zeichen lange Zeichenkette bestehend aus einem Großbuchstaben, irgendeinem beliebigen Zeichen und einem Großbuchstaben:
  `grep –E '[A-Z].[A-Z]'`

  > eine 3 Zeichen lange Zeichenkette der Form `-.^` oder `-.]` beinhaltet:
  `grep –E '[-]\.[]\^]'`

`egrep` – erweiterte reguläre Ausdrücke

- Iteratoren (***quantifier***)
  - Beziehen sich immer auf den *direkt vorangehenden* regulären Teilausdruck *RA*!
    Dieser kann ein einzelnes Zeichen eine Zeichenkette oder ein komplexerer Ausdruck sein.

Meta-zeichen	**Bedeutung**	**Korn-Shell**
*RA***?**	maximal einmaliges Vorkommen von RA (d.h. *optional*)	**?(***RA***)**
*RA******	beliebig häufiges Vorkommen von *RA*	***(***RA***)**
*RA***+**	mindestens einmaliges Vorkommen von *RA*	**+(***RA***)**
*RA***{***n***}**	genau *n*-maliges Vorkommen von *RA*	--
*RA***{***n***,}**	mindestens *n*-maliges Vorkommen von *RA*	--
*RA***{***n***,***m***}**	mindestens *n*- und maximal *m*-maliges Vorkommen von *RA*	--

egrep – erweiterte reguläre Ausdrücke

- Beispiele:

  > beliebige ganze Zahlen ohne führende Nullen:
  `egrep '[1-9][0-9]*|0'`

  > fünfstellige Zahl (mit führenden Nullen):
  `egrep '[0-9]{5}'`

  > mindestens 5-stellige Zahl (mit führenden Nullen):
  `egrep '[0-9]{5,}'`

  > mindestens einmal die Sequenz aaa:
  `grep –E 'a{3}+'`

  > 8086, 80286 oder 80386:
  `grep –E '80[23]?86'`

egrep – erweiterte reguläre Ausdrücke

- ***Anker***
  - Beziehen sich auf bestimmte (relative) Positionen

Meta-zeichen	Bedeutung	Korn-Shell
^	Zeilenanfang	--
$	Zeilenende	--
\<	Wortanfang	--
\>	Wortende	--

- ***Wort*** (i.S. des grep-Kommandos)
  - Zeichenkette bestehend aus Buchstaben, Ziffern und Unterstrich

- Beispiele:

  > Zeile, welche nur aus grep-Wörtern besteht, getrennt durch Blanks

  `grep –E '^[A-Za-z0-9_ ]*$'`

  > Wort, welches mit A gefolgt von einem Kleinbuchstaben beginnt

  `grep –E '\<A[a-z]'`

egrep – erweiterte reguläre Ausdrücke

- ***Alternation***
  - Ist die ODER-Verknüpfung regulärer Ausdrücke

Metazeichen	**Bedeutung**	**Korn-Shell**
RA_1 **|** RA_2	Zeichenketten, welche RA_1 oder RA_2 enthalten	RA_1 **|** RA_2

- Beispiele

  > Wochentagskürzel:

```
grep –E ‘Mo|Di|Mi|Do|Fr|Sa|So‘
```

egrep – erweiterte reguläre Ausdrücke

- Klammerung

Metazeichen	**Bedeutung**	**Korn-Shell**
(*RA***)**	Eingrenzung von Alternionen; Gruppierung Quantifier-Anwendung Referenzerzeugung	**( )** **( )** --

- Beispiele:

  > Zeile, welche mit einem grep-Wort oder dem Zeichen ^ beginnt:

  `grep –E '`**`^([A-Za-z0-9_]+|\^)`**`'`

  > kompakte Form für das Datum des amerikanischen Unabhängigkeitstages July fourth, July 4 und July 4th sowie Jul fourth, Jul 4 und Jul 4th.:

  `grep –E '`**`July? (fourth|4(th)?)`**`'`

  > IP-Adressenmuster

  `grep –E '`**`[0-9]+(\.[0-9]+){3}`**`'`

`egrep` – erweiterte reguläre Ausdrücke

- ***Rückwärtsreferenz***
  - erlaubt die Referenzierung von Zeichenketten, welche nur als pattern beschrieben sind

Metazeichen	Bedeutung	Korn-Shell
\n	Rückwärtsreferenz; 0 <= *n* <= 9 Steht für die Zeichenkette, welche als passend zu Teilausdruck in n.tem ()-Klammerpaar gefunden wurde	--

- Beispiele:

  Zeile, welche mit demselben grep-Wort beginnt und endet:

  ```
  grep –E '^([A-Za-z0-9_ ]+).*\1$'
  ```

  Zeile, in welcher eine Sequenz von Großbuchstaben zuerst vor einer Ziffernfolge und später diese Ziffernfolge direkt vor dieser Großbuchstabenfolge steht:

  ```
  grep –E '([A-Z]+)([0-9]+).*\2\1'
  ```

`grep` vs. `egrep`

- Die regulären Ausdrücke von `grep` müssen syntaktisch anders gebildet werden.

- Die Ausdruckskraft (zumindest für GNU grep) ist gleich.
  Ggf. ist in manchen anderen `grep`-Varianten keine Alternation verfügbar.
  Auch müssen ggf. „kein oder einmaliges Vorkommen“ bzw. „mindestens einmaliges Vorkommen“ nicht unter Verwendung von ? und + sondern mittels der geschweiften Klammern ausgedrückt werden.

- Die Zeichen `?`, `+`, `{`, `}`, `|`, `(`, `)` sind für `grep` *keine* Metazeichen.

  Bsp: `grep 'a{3}+'` sucht nach Vorkommen der Zeichenkette `a{3}+` und nicht nach mindestens einmaligem Vorkommen von `aaa`

- Die Bedeutung der in `egrep`-Ausdrücken bekannten Semantik kann durch voranstellen von \ erreicht werden: \?, \+, \{, \}, \|, \(, \).

  Bsp: `grep '[0-9]\+\(\.[0-9]\+\)\{3\}'` = `egrep '[0-9]+(\.[0-9]+){3}'`
  `grep '\([A-Z]\+\)\([0-9]\+\).*\2\1'` = `egrep '([A-Z]+)([0-9]+).*\2\1'`

grep, **egrep**: Grenzen

- Die regulären Ausdrücke von grep und egrep ermöglichen – im Gegensatz zur Korn-Shell - keine Negation über Teilausdrücke.
- Negation (Komplementbildung) kann z.T. mittels der Option –v erreicht werden:

  grep –v *pattern datei*

  gibt alle Zeilen aus, in denen keine Zeichenkette vorkommt, welche *pattern* als Teilstring enthält

- UND-Verknüpfung nicht durch logische Operatoren möglich. Aber in Shell durch Pipe-Sequenz realisierbar:

  Bsp:

  grep *pattern1 datei* | **grep –v** *pattern2* | **grep** *pattern3 …*

`(e|f)?grep`: Optionen

- pattern-Auswahl und -Interpretation

Option	Bedeutung
`-G`	*pattern* ist ein einfacher (basic) regulärer Ausdruck; Default
`-E`	*pattern* ist ein erweiterter (extended) regulärer Ausdruck; = `egrep`
`-F`	*pattern* ist ein fixer String, d.h. wird nicht als regulärer Ausdruck interpretiert, sondern als exakt zu findende Zeichenkette; = `fgrep`
`-e` *pattern*	Verwende *pattern* als regulären Ausdruck. (Nützlich um Bindestrich am *pattern*-Anfang zu maskieren)
`-f` *file*	Lies *pattern* zeilenweise aus der Datei *file.* Der Inhalt der Datei *file* ist als Alternation zu verstehen, bei welcher jede Zeile eine Alternative beschreibt. HINWEIS: • Vorsicht mit Leerzeilen oder Leerzeichen am Zeilenende • *pattern* in file nicht mehr quoten!
`-i`	Ignoriere Groß- und Kleinschreibung
`-w`	Pattern passt nur auf ganze Wörter
`-x`	Pattern passt nur auf ganze Zeile

`(e|f)?grep`: Optionen

- Ausgabekontrolle

Option	Bedeutung
`-v`	Alle Zeilen ausgeben, welche keine *pattern* entsprechende Zeichenkette enthalten.
`-h`	Anzeige des Dateinamens vor jeder Ausgabezeile
`-H`	Anzeige des Dateinamens vor jeder Ausgabezeile
`-n`	Anzeige der Zeilennummer vor jeder Ausgabezeile
`-l`	Statt passender Zeilen nur Dateinamen ausgeben
`-L`	Für nicht passende Zeilen nur Dateinamen ausgeben
`-x`	Pattern passt nur auf ganze Zeile

- Kontextkontrolle

Option	Bedeutung
-A *n*	Auch die *n* Zeilen, nach passender Zeile ausgeben (**A**fter)
-B *n*	Auch die *n* Zeilen, vor passender Zeile ausgeben (**B**efore)
-C *n*	Auch die *n* Zeilen, vor und nach passender Zeile ausgeben (**C**ontext)

Reguläre Ausdrücke: vergleichender Überblick

- Beschreibung *einzelner* Zeichen

Bedeutung	**egrep** **awk**	**grep** **sed**	**Korn-Shell**
„eine beliebiges Zeichen“	**.**	**.**	**?**
„eines der in […] angegebenen Zeichen“ **(Zeichenklasse)** Innerhalb von […] können Zeichen als Auflistung $c_1c_2c_3$…, Bereich c_1-c_2 oder als beliebige Kombination von Auflistung und Bereichen angegeben werden.	**[…]**	**[…]**	**[…]**
„ein beliebiges Zeichen, welches nicht in […] vorkommt“	**[^...]**	**[^...]**	**[!...]**
Zeichen *c,* falls *c* ein Metazeichen und das Zeichenpaar \c keine Meta-Bedeutung hat	*****c*	*****c*	*****c*

Reguläre Ausdrücke: vergleichender Überblick

- Iteratoren (***quantifier***)

Bedeutung	**egrep** **awk**	**grep** **sed**	**Korn-Shell**
maximal einmaliges Vorkommen von RA (d.h. *optional*)	*RA***?**	*RA***?** (implementierungsabhängig)	**?(***RA***)**
beliebig häufiges Vorkommen von *RA*	*RA******	***RA**** (implementierungsabhängig)	***(***RA***)**
mindestens einmaliges Vorkommen von *RA*	*RA***+**	*RA***+** (implementierungsabhängig)	**+(***RA***)**
genau *n*-maliges Vorkommen von *RA*	*RA***{***n***}**	*RA***{***n***\}**	--
mindestens *n*-maliges Vorkommen von *RA*	*RA***{***n***,}**	*RA***{***n***,\}**	--
mindestens *n*- und maximal *m*-maliges Vorkommen von *RA*	*RA***{***n***,***m***}**	*RA***{***n***,***m***\}**	

- Alternation

Zeichenketten, welche RA_1 oder RA_2 enthalten	RA_1 **|** RA_2	RA_1**|** RA_2	RA_1 **|** RA_2

Reguläre Ausdrücke: vergleichender Überblick

- Anker

Bedeutung	**egrep awk**	**grep sed**	**Korn- Shell**
Zeilenanfang	**^***RA*	**^***RA*	*string**
Zeilenende	*RA***$**	*RA***$**	**string*
Wortanfang	**\<***RA*	**\<***RA*	--
Wortende	*RA* **\>**	*RA* **\>**	--

- Klammerung, Rückwärtsreferenzierung

Bedeutung	**egrep awk**	**grep sed**	**Korn- Shell**
Teilstring	**( )**	**\(\)**	--
Rückwärtsreferenz	*n*	*n*	-

sed

- Während mittels der `grep`-Tools eine *musterorientierte* Suche und Filterung durchgeführt werden kann, können mittels des `sed` auch
  - Ersetzungen bzw.
  - Löschungen

  durchgeführt werden.
- Im folgenden wird nur ein Ausschnitt der Möglichkeiten des Tools **`sed`** gezeigt

- Aufruf:
  `sed` [**`-n`**] [[–e] '*sedaction*' …] [**`-f`***scriptfile* …] *inputfiles*

Option	Bedeutung
`-n`	`sed` gibt nur die mittels des p-Befehls bzw. der p-Modifikation des s-Befehls veränderten Zeilen aus
`-e`	*sedaction* folgt. (notwendig bei mehreren sedactions in einem Aufruf)
`-f`	Optionsargument *scriptfile* enthält `sed`-Befehle

sed -- Beispiele

- Aufbau einer *sedaction*
  [*address*[,*address*]] [**!**] *command /pattern1/pattern2/modifier*

- Beispiele:

`s/blabla/sinnvoll/`	ersetze das erste `blabla` in jeder Zeile durch sinnvoll
`s/xxx/yyy/g`	ersetze alle Vorkommen von `xxx` durch `yyy` in jeder Zeile
`3,5 p`	gib die Zeilen 3-5 aus
`/west/p`	gib alle Zeilen aus, welche `west` enthalten (→ grep)
`/^A.*B$/!d`	Lösche alle Zeilen, welche nicht mit `A` beginnen u. mit `B` enden
`/START/, /ende/ s/laut/leise/g`	ersetze für die Zeilen, zwischen dem ersten Vorkommen von `START` und dem nächsten Vorkommen von `ende`, `laut` durch `leise`

sed -- Verarbeitungsprinzip

- Grundverarbeitungsprinzip:
  - Reine Filterung nach `stdout`! *Keine* Veränderung der Inputfiles.
  - Durchführung jeder *sedaction* (per Kommandozeile oder aus Skriptfile) für jede Zeile des Adressbereichs.
  - Betrachtung der *sedactions* von links nach rechts bzw. oben nach unten.
  - Jede *sedaction* verarbeitet das Resultat der vorangehenden *sedaction* weiter.

sed -- Befehlsaufbau

- Aufbau einer *sedaction*:
  [*address*[,*address*]] [**!**] *command* **/***pattern1***/***pattern2***/***modifier*
- als Trennsymbol zwischen den *pattern* darf auch ein beliebig anderes genommen werden
- *pattern* sind reguläre Ausdrücke gemäß der Syntax aus Übersichtstabellen (siehe vorher!)
- In *pattern2* darf das Zeichen & als Rückwärtsreferenz für den Teilstring verwendet werden, welcher dem gesamten *pattern1* entspricht.
- In *pattern2* dürfen \1, \2, … als Rückwärtsreferenz auf in *pattern1* geklammerte Bereiche benutzt werden.

commands	**Bedeutung**	***modifier***	**Bedeutung**
d	delete		
p	print		
s	substitute	**g**	global, d.h. Ersetzung für alle Vorkommen je Zeile im Adressbereich
		p	print, d.h. Ausgabe der Zeilen, in denen tatsächlich eine Ersetzung stattfand. NUR sinnvoll mit Option -n

sed -- Beispiele

- Aufbau einer *sedaction*
  [*address*[,*address*]] [**!**] *command /pattern1/pattern2/modifier*

- Beispiele: (Beispieldatei: `etc.passwd.tab.txt`)

  `s/t.*r/&--&/` dupliziere jeden gefundenen mit `t` beginnenden und `r` endenden Teilstring (mit Trenner `.--`).
  Problematik: longest match hier gut sichtbar

  `s/\(.*\)\1//g` lösche alle Vorkommen von „Doppel-Teilstrings“

  `s/^\(.*\);\(.*\)$/\2/` erhalte nur letzte Spalte in einer csv-Datei

- Gesamtaufruf:

```
sed –e ‘s/t.*r/&--&/‘ etc.passwd.tab.txt
```

Reguläre Ausdrücke (ksh): neue komplexe *pattern* (ksh93)

- Die Korn-Shell 93 verfügt über nochmals erweiterte Mechanismen zur Formulierung regulärer Ausdrücke, welche die eingangs genannten Defizite der ksh88-pattern, aufheben.

Ausdruck	**Ersetzungsergebnis**
{*N***}(***pattern*[**|***pattern*]…**)**	genau N-faches Vorkommen von *pattern*
{*N,M***}(***pattern*[**|***pattern*]…**)**	*N*- bis *M*-faches Vorkommen von *pattern*
*****N**	*back reference* auf Inhalt der *N*-ten Klammer

Reguläre Ausdrücke (ksh): neue komplexe *pattern* (ksh93)

- Im Standardfall wird beim Mustervergleich immer der
  - ***erste*** und der
  - ***längste***

  zum regulären Ausdruck passende Teilstring im Vergleichsstring gesucht.

- In der Korn-Shell 93 (ab Version l) gibt es die explizite Möglichkeit den kürzesten passenden Teilstring zu berücksichtigen (*shortest match*)

Ausdruck	**Ersetzungsergebnis**
-(pattern***[|***pattern***]…)**	wie ***(***pattern***[|***pattern***]…)** aber *shortest match*
+-(*pattern***[|***pattern***]…)**	wie **+(***pattern***[|***pattern***]…)** aber *shortest match*
@**-(***pattern***[|***pattern***]…)**	wie @**(***pattern***[|***pattern***]…)** aber *shortest match*
?-(*pattern***[|***pattern***]…)**	wie **?(***pattern***[|***pattern***]…)** aber *shortest match*
{*N***}-(***pattern***[|***pattern***]…)**	wie **{***N***}(***pattern***[|***pattern***]…)** aber *shortest match*
{*N,M***}-(***pattern***[|***pattern***]…)**	wie **{***N,M***}(***pattern***[|***pattern***]…)** aber *shortest match*

Built-In-Kommandos: Exit-Stati (ksh93)

- Bzgl. der built-in-Kommandos der Korn-Shell 93 wurde die Bedeutung der Exit-Stati vereinheitlicht:

Exit-Status-Wert	**Bedeutung**
1-125	Kommando endete mit Fehler
2	Falscher Kommandogebrauch: usage-Ausgabe
126	Kommando gefunden, aber nicht ausführbar
127	Kommando nicht gefunden
128-255	Externes Kommando endete mit Fehler
>256	Kommandoabbruch mit Signal; Exit-Status-256 = Signalwert