1 Überblick Unix-Shell

1.1 Was ist Unix und Linux?

Unix rechtlich: Unix ist ein geschütztes Warenzeichen. Ursprünglich wurde es von den Bell Labs registriert, in den letzten Jahren wurden die Rechte für die Bezeichnung "Unix" mehrmals verkauft.

Nur wer diese Rechte besitzt oder an den Besitzer Lizenzgebühren zahlt, darf sein Betriebssystem offiziell "Unix" nennen.

Unix praktisch: Man bezeichnet ein Betriebssystem als Unix-System, wenn es so aufgebaut ist wie Unix und sich so bedienen läßt wie Unix: "It walks like Unix, it talks like Unix!"

Es gibt auch Standards, die festlegen, was ein Unix-System zu können hat: *POSIX* und *XOPEN*

Es handelt sich also um eine ganze Familie von Betriebssystemen (jeder große EDV-Anbieter hat sein eigenes), die einander sehr ähnlich, aber eigenständige Produkte mit jeweils eigenem Namen sind (bei IBM: AIX; bei Sun: Solaris; bei HP: HP/UX, bei Silicon Graphics: Irix, ...).

Linux: Linux ist das heute am weitesten verbreitete Betriebssystem aus der Unix-Familie. Es wurde und wird unabhängig vom Original-Unix von vielen Freiwilligen und eigenen Linux-Firmen auf der ganzen Welt entwickelt und ist (nachdem es keinen Original-Unix-Programmcode der Bell Labs enthält) kostenlos und frei im Quelltext verfügbar.

1.2 Die Geschichte von Unix

- Unix entstand gemeinsam mit der Programmiersprache C um 1970 in den *Bell Labs* (dem Forschungslabor der amerikanischen Telefongesellschaft AT&T, heute Lucent) als Versuch einiger Programmierer (Kernighan, Ritchie, Thompson, ...), ein Minimal-Betriebssystem zu schreiben.

- Die Bell Labs (bzw. USL = Unix System Labs) machten daraus ein kommerzielles Produkt und entwickelten Unix bis vor wenigen Jahren aktiv weiter, dieser Unix-Entwicklungszweig wurde unter *Unix System V* (V = "five") bekannt.

- Schon kurz nach der Entstehung hat sich die Universität *Berkeley* für Unix interessiert, die Quellcode-Lizenzen für einen frühen Entwicklungsstand des Systems gekauft, und das System unabhängig von den Bell Labs ebenfalls weiterentwickelt. Dieser Unix-Zweig wurde unter dem Namen *BSD Unix* bekannt. Er steht unter einer sehr liberalen Lizenz und wurde rasch das weltweit führende Betriebssystem in Universitäten und Forschungslabors.

  Berkeley hat seine Unix-Entwicklung auch schon vor vielen Jahren eingestellt; dieser Zweig lebt aber in den frei verfügbaren Unix-Systemen *FreeBSD*, *NetBSD* und *OpenBSD* weiter, bei denen der wenige in BSD verbliebene Original-Bell-Labs-Code inzwischen restlos durch neu geschriebenen Open Source Code ersetzt wurde.

  BSD Unix lieferte auch die Grundlage für das neue Apple-Mac-Betriebssystem *Mac OS X*.

- Alle großen EDV-Anbieter kauften schon bald Unix-Lizenzen von AT&T oder Berkeley und entwickelten daraus ihren eigenen Unix-Dialekt für ihre eigene Hardware — um 1990 herum gab es mindestens 50 verschiedene, eigenständige Unix-Implementierungen.

- Daneben gab und gibt es immer wieder Versuche, wie Unix funktionierende Betriebssysteme "von Null weg" zu schreiben, ohne eine Lizenz für den Original-Unix-Code zu kaufen.

  Das erfolgreichste dieser Systeme ist *Linux*, begonnen 1991 von Linus Torvalds (damals Student). Auch *Android* baut auf Linux auf.

1.3 Kernel und Shell

Der Kernel: (= das Betriebssystem)

- verwaltet alle Ressourcen (CPU, Speicher, Platten, I/O-Geräte)
- stellt allen anderen Programmen Systemfunktionen (für I/O, für das Starten von Programmen, für das Anfordern von Speicher, ...) zur Verfügung
- bietet selbst aber keinerlei Benutzerschnittstelle (d. h. man kann sich nicht direkt mit dem Kernel unterhalten)

Die Shell: (= der Command (Line) Interpreter)

- ist die textbasierte (nicht grafische) Benutzeroberfläche von Unix
- ist ein eigenständiges Programm, kein Teil des Kernels
- liest Befehle vom Terminal und führt sie entweder selbst aus oder startet das entsprechende Programm
- entspricht `command.com` unter DOS

Unix ist ein Mehrbenutzersystem: Pro Terminalsitzung wird eine Shell gestartet, es können viele Shells desselben oder verschiedener Benutzer gleichzeitig auf einem Unix-System laufen!

1.4 Das X Window System

- Das X Window System (umgangssprachlich XWindows oder nur X) ist die unter Unix übliche grafische Benutzeroberfläche.

- Ebenso wie die Shell ist XWindows ein eigenständiges, vom Kernel unabhängiges Programm (bzw. derer viele), es wurde ursprünglich am MIT entwickelt.

- Die unter Linux übliche Xwindows-Implementierung heißt *Xorg (früher Xfree86)*.

- XWindows besteht im wesentlichen aus folgenden Teilen:

  Der X Server: Der X Server zeichnet die Pixel und kümmert sich um Tastatur und Maus.
  Die GUI-Toolkits: Die Bedienelemente grafischer Anwendungen (Scrollbars, Menüs, Dialogfenster, Buttons, usw.) werden meist unter Verwendung sogenannter Toolkits (großer Libraries) wie Qt (das Toolkit von KDE) oder GTK (das Toolkit von Gnome und Xfce) implementiert. Je nach verwendetem Toolkit sehen diese Elemente bei verschiedenen Anwendungen daher mitunter recht unterschiedlich aus.

Der Window Manager: Der Window Manager verwaltet die angezeigten Fenster und verpaßt ihnen eine Titelleiste.
Früher gab es unter XWindows mindestens ein Dutzend verschiedener Window Manager zur Wahl, die alle ganz verschieden aussahen und funktionierten. Sie stellten oft auch Startmenü, Taskleiste usw. zur Verfügung.
Heute wird dafür unter Linux meist ein komplettes *Desktop Environment* wie *Gnome*, *KDE* oder *Xfce* verwendet, das u. a. auch einen Window Manager enthält.

Die grafischen Anwendungen: Terminalemulation, Uhr, Editor, Textverarbeitung, Zeichenprogram, ...

- Die einzelnen Teile von Xwindows können im Unterschied zu Microsoft Windows auf *verschiedenen* Rechnern laufen, sie kommunizieren über das TCP/IP-Netzwerkprotokoll miteinander:

  * Auf einem anderen Rechner laufende grafische Anwendungen können auf dem lokalen Rechner ihre Fenster anzeigen und mit Tastatur und Maus bedient werden.
  * Auf demselben Schirm können gleichzeitig Fenster von Anwendungen, die auf mehreren verschiedenen Rechnern laufen, angezeigt werden. Ein Systemadministrator startet z. B. üblicherweise auf jedem zu verwaltenden Rechner ein Terminalemulationsprogramm und läßt sich alle diese Terminalfenster auf seinem eigenen Arbeitsplatzrechner anzeigen.
  * Im Extremfall besteht der Arbeitsplatz nur aus einer Minimal-Hardware (ein sogenanntes *X-Terminal*, auch *Thin Client* genannt, ohne Platte, Floppy, ...), auf der gar kein echtes Betriebssystem läuft, sondern die über das Netzwerk ausschließlich einen X Server bootet (nur der muß wirklich auf dem Rechner laufen, an dem Tastatur, Bildschirm und Maus hängen); der Window Manager und alle Anwendungen laufen dann nicht lokal, sondern auf einem zentralen Unix-Server (das war schon vor 25 Jahren die "Frühform" von Cloud Computing).

1.5 Wie kommt man zu einer Shell?

Auf einem Text-Terminal:
(Systembildschirm im Textmodus, serielles alfanumerisches Terminal)
`login` startet nach erfolgreicher Passwort-Prüfung eine Shell.

Auf einem XWindows-Schirm:
Man startet unter XWindows ein Terminal-Emulations-Programm wie `xterm` (bei klassischem XWindows) oder `konsole` (unter KDE), d. h. ein Programm, das ein Textfenster anzeigt und so tut, als ob es ein Text-Terminal wäre, und dieses startet dann eine Shell.

Über das Netz:
Programme wie `telnet` oder `ssh` (Secure Shell) bzw. `putty` emulieren am lokalen Arbeitsplatz unter DOS oder Windows ein Text-Terminal und stellen eine TCP/IP-Netzwerkverbindung zu einem anderen Rechner her. Auch unter Unix gibt es `telnet` und `ssh`, um sich auf einem anderen Rechner anzumelden.

Dort nimmt ein Serverprogramm die Verbindung entgegen, prüft das Passwort, und startet dann für jede einlangende Telnet-Sitzung eine eigene Shell.

Das `telnet` bzw. `ssh` am eigenen Rechner schickt die Tastatureingaben über das Netzwerk zur Shell am entfernten Rechner und zeigt deren Output an (wie bei einem alten alfanumerischen Terminal mit serieller Schnittstelle, nur daß die Daten eben über eine Netzwerkverbindung und nicht über eine serielle Leitung fließen).

Achtung: `telnet` schickt alles (auch Username und Passwort bei der Anmeldung!) im Klartext über das Netz! `ssh` verschlüsselt den gesamten Datenverkehr.

Aus einem anderen Programm: Einige Anwendungen (z. B. der Editor `vi`) erlauben es, einzelne Shell-Befehle auszuführen oder kurz mit einer Shell zu arbeiten, ohne die Anwendung selbst zu beenden. Sie starten dazu eine *neue* Shell und übergeben ihr den eingegebenen Befehl oder die Kontrolle über das Terminal. Nach Ende dieser Shell kann man wieder in der ursprünglichen Anwendung weiterarbeiten.

1.6 Was sind Shell-Skripts?

- Shell-Skripts sind Textfiles, die Befehle enthalten, die von einer Shell abgearbeitet werden. Sie entsprechen `.bat`-Files unter DOS.
- Verwendet werden Shell-Skripts unter anderem für
  - Unix-Befehle bzw. -Programme, die sich eben leichter durch ein Skript als ein compiliertes Programm implementieren lassen,
  - die Steuerung aller beim Starten und Stoppen eines Unix-Systems notwendigen Abläufe,
  - die sogenannten Profiles (`.profile` oder `.bash_profile`), die jedesmal beim Start einer Shell nach dem Login ausgeführt werden, um diverse benutzerspezifische Initialisierungen der Shell vorzunehmen,
  - Installations- und Deinstallations-Skripts,
  - und periodisch auszuführende Aktionen (sogenannte `cron`-Jobs; `cron` ist das Programm, das die zeitgesteuerte Ausführung anderer Programme überwacht).
- Neben der Shell gibt es unter Unix noch einige andere Skript-Sprachen, z. B.:
  - `awk`, die Standard-Unix-Sprache für kleine Hilfsprogramme.
  - `sed`, ein Batch-Editor zur automatisierten Textmanipulation.
  - **Perl** und **Python** (und auch *Ruby* usw.), die in der freien Software-Szene beliebtesten Skript-Sprachen.
  - **Tk/Tcl**, eine Skript-Sprache für Skripts mit grafischer Benutzeroberfläche.

1.7 Shell-Familien

Die Shell ist ein eigenständiges, austauschbares Programm: Jeder kann im Prinzip eine neue Shell schreiben und statt der normalen Shell verwenden.

Im Laufe der Zeit haben sich zwei Familien von Shells etabliert:

`sh`-kompatible:

- `sh`, "Bourne Shell", die Original-Unix-Shell.
- `ksh`, "Korn Shell", die den heutigen POSIX- bzw. XOPEN-Standards entsprechende und bei den meisten kommerziellen Unix-Systemen mitgelieferte Shell.
- `bash`, eine frei verfügbare Shell (GNU Software), üblich bei freien Unix-Systemen.

`csh`-kompatible:

- `csh`, "C Shell", eine Berkeley-Entwicklung, lange Zeit üblich bei allen von BSD abstammenden Unix-Systemen.

- `tcsh`, eine freie Weiterentwicklung der `csh`.

Die Bedienung vom Terminal aus ist bei beiden Familien in etwa gleich, Shell-Skripts hingegen sind nur innerhalb einer Familie aufwärtskompatibel, aber zwischen den beiden Familien inkompatibel (ein `sh`-Skript läuft auch mit einer `bash`, aber nicht mit einer `csh`, ebensowenig läuft ein `csh`-Skript mit einer `bash`: Syntax, Schleifenbefehle usw. sind verschieden!).

2 File-Permissions

- ***Permissions = Zugriffsrechte***
  Für jeden File und jedes Directory sind Zugriffsrechte gespeichert, die vom Betriebssystem bei jedem Zugriff geprüft werden. Anzeigen mit `ls -l`, ändern mit `chmod`.

  Weiters ist bei jedem File der *Besitzer* (Owner) und die *Besitzergruppe* (Group) gespeichert. Anzeigen mit `ls -l`, ändern mit `chown` (darf nur der Systemadministrator) und `chgrp` (darf nur der Besitzer des Files).

  Außerdem ist jeder Benutzer ein oder mehreren Benutzergruppen zugeordnet. Anzeigen mit `id`.

- 3 Arten (in dieser Reihenfolge):

  `r:` File / Directory darf gelesen werden.
  `w:` File / Directory darf geschrieben / geändert werden.
  `x:` File darf ausgeführt werden / ins Directory darf hineingewechselt werden.
  `s, t:` SetUID/SetGID-Bit, Sticky Bit: Interessiert uns nicht.

- Für 3 Arten von Benutzern (in dieser Reihenfolge):

  User: Wenn der zugreifende Benutzer gleich dem Besitzer des Files ist, gelten für ihn die User-Zugriffsrechte.
  Group: Wenn der zugreifende User der Besitzergruppe des Files zugeordnet ist (aber nicht der Besitzer ist), gelten für ihn die Group-Rechte.
  Other: Wenn beides nicht der Fall ist, gelten die Other-Rechte.

- Das macht insgesamt 9 Bits, die auf 3 Arten dargestellt werden können:

  * Als Oktalzahl: `0777` (jeder darf alles), `0644` (der Besitzer darf lesen und schreiben, die anderen nur lesen), `0710` (der Besitzer darf alles, Gruppenmitglieder dürfen ausführen, der Rest darf gar nichts), ...
  * Im `ls`: `rwxrwxrwx`, `rw-r--r--`, `rwx--x---` (jeweils wie oben)...
  * Im `chmod`: Kombination aus `ugoa` ("User", "Group", "Other", "All"; das Default ist "All", wenn man nichts angibt) und `+-=` (dazu, weg, genau so setzen) und `rwx` (oder `ugo` für "gleich wie Userrechte/Grouprechte/Other-Rechte"). Beispiele:

    `+x:` Ausführ-Rechte für alle einschalten.
    `u+w:` Schreibrecht für User einschalten.
    `go-rw:` Lese- und Schreibrechte für Group und Other ausschalten.
    `=rx:` Rechte für alle auf `rw` setzen (d. h. `0555`).
    `o=g:` Für Other die gleichen Rechte wie für Group setzen.

- Rechte neuer Files und Directories:
  Werden von dem Programm festgelegt, das den File / das Directory anlegt (üblicherweise `0777` bei Directories und Executables, `0666` bei allen anderen Files), ***abzüglich*** der Bits, die mit `umask` festgelegt wurden. `umask` wird normalerweise auf `0022` (lösche die Schreibrechte für Group und Other), im Extremfall auf `0077` (lösche alle Rechte für Group und Other) gesetzt.

- Anmerkung: Der erste Buchstabe bei den Rechten im `ls` ist der Filetyp:

  `-`: Normaler File.
  `d`: Directory.
  `c`: Character Device.
  `b`: Block Device.
  `l`: Soft Link.
  `p`: Named Pipe.
  `s`: Socket.

3 Links

Links sind Verknüpfungen zwischen Files. Unix kennt 2 Arten:

Hard Links:

Erzeugung:
`ln` *bestehender_file neuer_link*

Erkennung:

- Link Count im `ls -l` (zweite Spalte) größer `1`.
- Gleiche Inode-Nummer im `ls -i`.

Funktion:

- Directory-Information wird in Unix zweistufig gespeichert:
  - Im Directory selbst steht nur der Name des Files und eine vom System intern vergebene Filenummer (die Inode-Nummer).
  - Die eigentliche Directory-Information zu jedem File (Größe, Rechte, Besitzer, Änderungsdatum, dazugehöriger Plattenplatz, ...) steht in einem eigenen Systembereich auf der Platte, der Inode-Table: Dort gibt es einen Eintrag pro File, die Inode-Nummer eines Files ist der Index des zu diesem File gehörenden Eintrags in der Inode-Table.
- Mehrere Hard Links auf denselben File sind einfach mehrere Directory-Einträge mit derselben Inode-Nummer: Derselbe File-Eintrag ist unter mehreren Namen ansprechbar.
- Bei Hard Links gibt es kein Original und keinen Verweis darauf, alle Hard Links auf einen File sind gleichberechtigt. Der File verschwindet erst, wenn der letzte darauf verweisende Directory-Eintrag gelöscht wird.

Vorteile:

- Hard Links können nicht "in der Luft hängen", der File kann nicht verloren gehen.
- Hard Links können nicht zyklisch sein.
- Hard Links sind etwas platzsparender und schneller als Soft Links.

Nachteile:

- Hard Links funktionieren nur innerhalb eines Filesystems (einer Platten-Partition, einer Floppy, ...).
- Hard Links darf man nur für Files machen, nicht für Directories (die Directory-Hierarchie arbeitet intern mit Hard Links und würde durcheinanderkommen, wenn es zusätzliche Hard Links auf Directories gäbe).
- Man kann nur schwer (mit `ls -i` und `find`) herausfinden, was die anderen Hard Links auf denselben File sind.
- Hard Links fallen auseinander, wenn der File gelöscht und unter gleichem Namen neu angelegt wird, von einem Backup zurückgespielt wird o. ä. ($\Longrightarrow$ zwei getrennte Files: Der neu angelegte File bekommt eine neue Nummer, die anderen Links haben noch die alte Nummer).

Soft (Symbolic) Links:

Erzeugung:
`ln -s` *bestehender_file neuer_link*

Erkennung:

- `l` in der ersten Spalte eines `ls -l`.
- `->` *Filename* hinten beim `ls -l`.

Funktion:

- Bei einem Soft Link verweist der Directory-Eintrag auf einen anderen Filenamen statt auf den eigentlichen File.
- Um zum eigentlichen File zu kommen, schaut das System unter dem angegebenen Filenamen nach.

Vorteile:

- Soft Links funktionieren auch über getrennte Filesysteme hinweg.
- Soft Links funktionieren auch für Directories (wird viel verwendet!).
- Soft Links werden durch Umkopieren, Neuschreiben usw. des Files nicht beeinflußt.

Nachteile:

- Der Original-File weiß nichts von auf ihn zeigenden Soft Links (und man kann es auch nicht herausfinden!). Man kann den Original-File löschen, obwohl noch Soft Links auf ihn zeigen. Die zeigen dann ins Leere ...
- Soft Links können zyklisch sein.

4 Filename Expansion ("Wildcards")

Wie?

- `*` paßt auf 0 oder mehr beliebige Zeichen (außer `/`).
  Beispiele: `*.c test*.* *2001* letters/*~`
- `?` paßt auf genau **1** beliebiges Zeichen.
  Beispiele: `*.? data???`

- `[]` gibt mehrere Möglichkeiten für einen **einzelnen** Buchstaben an, `-` kann für Bereiche verwendet werden, `!` oder `^` für "nicht".
  Beispiele: `src/*.[ch]` `[Mm]akefile` `[A-Z]*` `*[^0-9]` [1]
- `~` expandiert zum Pfadnamen des eigenen Homedirectories, `~`*user* setzt das Homedirectory des angegebenen Users ein. `~` wird nur am *Anfang* des Filenamens geprüft.
  Beispiel: `~/.bash_profile` wird beispielsweise zu `/home/2dhd15/.bash_profile`

Achtung:

- **Filename Expansion macht die Shell, nicht das aufgerufene Programm!!!**
  Bei `ls *.c` wird in Wirklichkeit z. B. `ls uebung.c test.c myprog.c` gestartet!
- Wenn ich daher ein Programm mit einem `*` oder `?` in einem Argument aufrufen will, muß ich quoten (siehe unten), um die Shell am Expandieren des `*` bzw. `?` zu hindern!
  Wichtigstes Beispiel: `find` (z. B. `find /usr -name "*.h"`)
- Die Command Line ist längenbeschränkt (je nach System 5000–500000 Zeichen) und kann zu kurz werden, wenn es viele passende Files gibt.
  In diesem Fall: Aufspalten oder `find` und `xargs` verwenden!
- Wenn kein File paßt, wird der Filename *unverändert* (mit `*`) an das aufgerufene Programm übergeben (d. h. z. B. `ls *.c`). Im günstigsten Fall beschwert sich das Programm, daß es keinen File mit dem Namen `*.c` (oder was immer) gibt, im ungünstigsten macht es Unfug (legt z. B. einen File mit diesem Namen an!).
- Die `bash` kann man so konfigurieren, daß in diesem Fall *gar nichts* übergeben wird (nach dem Motto "Wenn 0 Files passen, rufe ich den Befehl mit 0 Files auf", also z. B. nur `ls`). Auch nicht viel besser ...
- Alle Files erreicht man mit `*`, nicht mit `*.*` !!!
  Filenamen unter Unix müssen keine Extension enthalten, daher auch keinen Punkt, und die ohne Punkt sind bei `*.*` nicht dabei!
- Files, deren Namen mit `.` beginnen, sind bei `*` *nicht* mit dabei
  (werden also z. B. bei `cp *` ... nict mitkopiert)!!!
  Man muß sie mit `.*` o. ä. extra dazutun!
  Grund: Gemäß Unix-Filenamens-Konventionen gelten mit `.` beginnende Files als *"hidden"*, sie werden z. B. bei `ls` nicht angezeigt. Die meisten internen Konfigurations- und Status-Dateien usw. beginnen daher mit `.` oder liegen in einem eigenen Verzeichnis, das mit `.` beginnt.
  Achtung: `.*` umfasst auch `.` und `..` (das aktuelle Verzeichnis und das Vater-Verzeichnis), was meistens nicht erwünscht ist und oft zu unbeabsichtigten Aktionen führt.
- Filename Expansion prüft alle Namen von Files *und* Unterverzeichnissen, das kann man nicht trennen (d. h. man kann mittels Wildcard nicht nur die Files oder nur die Verzeichnisse ansprechen).
- Wildcards funktionieren nicht nur im Filenamen, sondern auch in Directory-Namen (wobei immer genau *eine* Directory-Ebene geprüft wird, weil der `/` nicht auf `*` paßt):
  `*/Makefile` paßt auf alle Files `Makefile` in den unmittelbaren Unterverzeichnissen.
  `*/*/*.c` betrifft alle C-Files zwei Ebenen unter dem aktuellen Verzeichnis.
- Es gibt keine Möglichkeit, mit Filename Expansion Files oder Verzeichnisse in beliebig bzw. variabel tief geschachtelten Unterverzeichnissen anzusprechen.

---

[1] Das klassische `[a-z]` für Kleinbuchstaben und `[A-Z]` für Großbuchstaben funktioniert seit neuestem nicht mehr bzw. nur mehr in der Locale `C` (d. h. bei ausgeschalteter Internationalisierung): Der POSIX-Standard schreibt vor, daß `-`-Ranges entsprechend der Sorting Order der jeweiligen Locale aufgelöst werden, und die ist üblicherweise `AaBbCc` usw.. In diesem Fall ist `[[:lower:]]`, `[[:upper:]]`, `[[:alpha:]]`, `[[:alnum:]]` usw. zu verwenden (das erfaßt auch Umlaute richtig).

Nur* `bash`*:
In der `bash` kann man Filenamen, die sich nur ein bißchen unterscheiden, mit `{}` abkürzen:
`data/{jan,feb,maerz}2001/Index` wird zu
`data/jan2001/Index data/feb2001/Index data/maerz2001/Index`
und zwar *ohne* Prüfung, ob es diese Files auch wirklich gibt!

5 Quoting

Wozu?

- Um Sonderzeichen, die irgendeine Sonderbedeutung in der Shell haben (`| < ; $ &` usw.), als Zeichen zu erhalten und deren Sonderbedeutung abzuschalten.
- Im besonderen, um Filename Expansion zu verhindern und einem Programm ein Argument mit `*` oder `?` zu übergeben.
- Ebenso, um nicht druckbare Zeichen als solche einzugeben (z. B. `Ctrl/C` oder `Tab`).
- Um ein Argument (z. B. einen Suchstring bei `fgrep`) oder einen Filenamen anzugeben, der Leerzeichen enthält (normalerweise trennt ein Leerzeichen Argumente und Filenamen): Alles, was innerhalb von `"` steht, wird als *ein* Wort behandelt, egal, wie viele Leerzeichen es enthält.
- Um ein leeres Argument (d. h. ein Argument, das ein String der Länge 0 ist) anzugeben: `""`

Wie?

- Der Backslash quotet einen einzelnen, beliebigen Buchstaben *c*: `\`*c* ergibt *c*, egal, welche Bedeutung *c* sonst hätte.
  Achtung: `\n` ergibt den Buchstaben `n`, nicht `Newline`!
  Sonderfall: `\Enter` bewirkt eine Fortsetzungszeile (d. h. man kann in einer neuen Zeile weiterschreiben, ohne daß die alte sofort ausgeführt wird).
  Sonderfall: Manche nicht druckbaren Steuerzeichen (`Ctrl/`*x*) lassen sich mit `\` quoten, andere mit `Ctrl/V` (ausprobieren!).
- Einfache Anführungszeichen quoten einen beliebigen String *str* ohne Ausnahmen, d. h. bei `'`*str*`'` bleibt *str* völlig unverändert (auch ein `\` innerhalb `''` verliert seine Sonderfunktion!).
- Doppelte Anführungszeichen quoten einen beliebigen String *str* teilweise: `\`, `'` und `$` (und auch das Backquote-Konstrukt, kommt später ...) behalten ihre Sonderfunktion innerhalb von `"`*str*`"`, alle anderen Sonderzeichen werden als normale Buchstaben behandelt.

Achtung:
`\`, `"` und `'` werden von der Shell entfernt, d. h. dem aufgerufenen Programm wird das gequotete Argument *ohne* Anführungszeichen und Backslash übergeben! (d. h. statt `"ein\\wort"` sieht das aufgerufene Programm `ein\wort`).

Nur neue* `bash`*-Versionen:
`$'`*str*`'` funktioniert wie `'`*str*`'`, aber in *str* wird der `\` wie in C behandelt: `\n` = `Newline`, `\t` = `Tab`, `\x`*xx* = hexadezimal angegebenes Zeichen, ...

6 Pipes und Redirection

Die Files, die ein Programm offen hat, sind in Unix intern von `0` aufwärts numeriert. 3 Files sind bei jedem Programm standardmäßig geöffnet:

`stdin (0)`: Der normale Input (üblicherweise von der Tastatur).
`stdout (1)`: Der normale Output (üblicherweise auf den Bildschirm).
`stderr (2)`: Output von Fehlermeldungen (üblicherweise auch auf den Bildschirm).

Mit *command1* `|` *command2* kann man 2 Befehle gleichzeitig starten, wobei der `stdout` von *command1* mit dem `stdin` von *command2* verbunden wird (d. h. alles, was *command1* an Output liefert, wird als Input in *command2* gesteckt).

Beispiele:

- `ls -l | less`
- `find . | wc`
- `ps -ef | fgrep -v root | sort`

Man kann `stdin`, `stdout` und `stderr` auch vom Terminal weg auf Files umleiten, indem man hinten an die Befehlszeile eine Umleitung anhängt:

`<`*file*: Der `stdin` des Befehls wird auf *file* umgeleitet, d. h. der Befehl liest seinen Input aus *file* statt vom Terminal (analog für File *n* des Befehls: *n*`<`*file*).

`>`*file*: Der `stdout` des Befehls wird auf *file* umgeleitet, d. h. der Befehl schreibt seinen Output (nicht aber seine Fehlermeldungen!) in *file* statt auf das Terminal.
Analog für File *n* des Befehls: *n*`>`*file*. `2>`*file* leitet daher die Fehlermeldungen des Befehls in *file* um.
Wenn der File noch nicht existiert, wird er angelegt, sonst wird sein alter Inhalt überschrieben. Schmutztrick daher: `>`*file* allein in einer Zeile ohne Befehl (oder mit dem Dummy-Befehl `:`) legt *file* leer an bzw. löscht den Inhalt von *file*.

`>>`*file*: Wie `>`, aber es wird an den bestehenden Inhalt von *file* hinten angehängt.

n`>&`*m*: Verbindet den Outputfile Nummer *n* mit demselben File, mit dem der File Nummer *m* gerade verbunden ist (analog für zwei Inputfiles: *n*`<&`*m*).
Beispiel: `find . >/tmp/files 2>&1` macht ein `find` und speichert sowohl dessen Ergebnis als auch dessen Fehlermeldungen in `/tmp/files`.
Sonderfall: `>&`*m* leitet `stdout` auf File *m* um. Mit `echo Error: ... >&2` kann man also beispielsweise eine Fehlermeldung auf `stderr` statt `stdout` ausgeben.

`&>`*file*: ***Nur*** `bash`***:*** Abkürzung für `>`*file* `2>&1`, sowohl `stdout` als auch `stderr` werden auf *file* umgeleitet.

`<<`*endword*: Für Inputtexte in Shell-Skripts ("Here-Documents"):

1. Die Shell liest zeilenweise die folgenden Zeilen aus dem Skript, bis sie eine Zeile findet, die nur aus *endword* besteht (üblicherweise verwendet man `EOF` als *endword*).
2. Die so gelesenen Zeilen werden als Input für den Befehl verwendet, d. h. der Befehl kann von seinem `stdin` den im Skript enthaltenen Text lesen, als ob er am Terminal eingetippt worden wäre.

Wichtiger File für Umleitung: `/dev/null`
(leer beim Lesen, "vernichtet" alles, was man reinschreibt)

Wichtiger Sonderfall: Eine Redirection in Kombination mit dem Befehl `exec` ohne Argumente (!) leitet den entsprechenden File für die Shell selbst (d. h. für den gesamten Rest des Skripts) um. Beispiel: Nach `exec < script.input` lesen alle nachher ausgeführten Befehle im Skript ihren Input aus dem File `script.input` statt vom Terminal.

`bash` (und nur `bash`) kennt Spezialfälle für *file* (u. a. zum Umleiten auf eine Netzverbindung), siehe `man`-Page.

Achtung:
Redirects werden von links nach rechts durchgeführt, `>out 2>&1` ist etwas anderes als `2>&1 >out`.

Deshalb kann man beispielsweise mit ... `2>&1 |` ... die Fehlerausgaben des ersten Befehls in die Pipe zum zweiten Befehl umleiten ("leite zuerst die Fehlerausgaben in den normalen Output um, und leite dann beides zusammen in die Pipe um").

7 Parameter, Variablen

3 Arten:

Special Parameter: Werden von der Shell belegt, können nur gelesen werden:

- `$`*n*: Das *n*-te Argument (ab 1).
- `$*`: Alle Argumente. `"$*"` ergibt *ein* Wort.
- `$@`: Alle Argumente. `"$@"` ergibt *n* einzelne Worte.
- `$#`: Anzahl der Argumente.
- `$0`: Name des Shell-Skripts.
- `$?`: Letzter Exit Code (siehe unten).
- `$$`: Process Id des Shell-Prozesses.

Shell-Variablen, Environment-Variablen: Können gelesen und geschrieben werden (schreiben ist bei manchen nicht sinnvoll oder gesperrt), werden von der Shell (oder anderen Programmen) intern in vordefinierter Weise verwendet. Siehe `man bash`, Beispiele: `PATH HOME PS1 LANG LC_... TZ USER UID TERM DISPLAY HOSTNAME PWD OLDPWD RANDOM` (es gibt noch viel mehr!): Die Verzeichnisse, in denen Befehle gesucht werden; das Home-Verzeichnis; der Shell-Prompt; Sprache, Ländereinstellungen und Zeitzone; der Name des aktuellen Benutzers, ...

Normale Variablen: Alle anderen, frei gewählten Namen, beliebig für eigene Zwecke verwendbar (Namen sind wie in Programmiersprachen üblich aufgebaut).

Setzen von Variablen:

- *name*`=`*expr*
  Variable *name* wird in der Shell gesetzt.
- *name*`=`*expr command*
  Variable *name* wird nicht in der Shell, sondern nur für den einen Befehl *command* gesetzt.
- `export` *name*`=`*expr* oder `export` *name*
  Variable *name* wird als "zu exportierend" gekennzeichnet, d. h. an alle ausgeführten Befehle und Subshells übergeben (*name* gilt sonst nur für die Shell selbst!).
- `unset` *name*
  Variable *name* wird gelöscht.

Achtung: *Keine Leerzeichen rund um das* `=` *!*

Lesen von Variablen: Früher einfach `$`*name* , laut Standard: `${`*name*`}`

Achtung: Gibt es keine Variable *name*, so kommt *kein* Fehler! Das `$`*name* wird stillschweigend durch nichts (den Leerstring) ersetzt!

Achtung: Falls der Inhalt der Variable *name* leer sein könnte, Sonderzeichen oder Zwischenräume enthalten könnte, usw., ist es meist notwendig bzw. sinnvoll, das Ganze unter `"` zu setzen:
`"${`*name*`}"`

Menge aller gesetzten exportierten Variablen = "Environment" (anzeigen mit `env`)
Anzeigen einer einzelnen Variable: `echo ${`*name*`}`

Variablen in der Shell sind normalerweise typlos, de facto Strings (`bash` kennt auch Arrays und Integer).

Beachte bei Variablen in Shell-Skripts:

- Wird das Skript wie ein Befehl aufgerufen, gilt die Variable nur im Skript: Die aufrufende Shell sieht die Variable nicht.
- Wird das Skript hingegen mit dem Punkt-Befehl eingelesen, gelten die Variablen-Settings im Skript auch für die aufrufende Shell.

8 Der Exit Code

Jeder Befehl (egal, ob Binärprogramm, Shell-Skript, Shell-Funktion oder eingebauter Shell-Befehl) endet mit einem ***Exit Code*** (auch: *Exit Status*, *Return Code*, ...).

`0`**:** Der Befehl wurde erfolgreich ausgeführt.
`1` **bis** `125`**:** Der Befehl wurde erfolglos ausgeführt bzw. es trat ein Fehler auf (z. B. "File nicht gefunden").
`126` **und** `127`**:** Die Shell konnte den Befehl gar nicht starten.
`128` $+ n$**:** Der Befehl wurde durch Signal Nummer n abgebrochen (z. B. `Ctrl/C`).
`128` **und** `256`**:** Keine gültigen Exit-Codes.

Der Exit Code einer Pipe ist der Exit Code des *hintesten* Befehls der Pipe. Auch bei allen anderen Konstrukten mit mehreren Befehlen ist der Exit Code des gesamten Konstrukts gleich dem Exit Code des letzten ausgeführten Befehls.

9 Tests

- Wenn in der Shell irgendwo ein Wahrheitswert (boolscher Wert) gefragt ist, kann dort *jeder beliebige* Befehl verwendet werden: Er wird ausgeführt, und sein Exit Code wird analysiert: `0` gilt als `true` (erfolgreich, Bedingung erfüllt), jeder von `0` verschiedene Wert als `false` (erfolglos, Bedingung nicht erfüllt).

  Das ist genau umgekehrt wie in C!

- Es gibt zwei eigene Befehle `true` und `false`, die immer `0` und `1` (oder `255`) liefern[2].

- Zum Ausführen von Vergleichen oder bestimmten Tests auf Files, Strings usw. gibt es den Befehl `test` *expr*. Statt `test` *expr* kann man auch `[` *expr* `]` (Leerzeichen vor und nach den `[ ]` !) schreiben (ist genau das gleiche).
  Je nach dem Ergebnis von *expr* liefert `test` einen Exit Code `0` oder `1`.

  Die wichtigsten *expr* (Details siehe `man test` oder `man bash`, "Conditional Expressions"):

[2] Weiters gibt es einen "Dummy-Befehl" `:`, der erfolgreich nichts tut (No-Op), also auch stets Exit Code `0` liefert. Man liest daher in Skripts auch immer wieder beispielsweise `while :` statt `while true`.

* `-e` *file*: *file* existiert.
* `-f` *file*: *file* ist ein normaler File.
* `-d` *file*: *file* ist ein Directory.
* `-L` *file*: *file* ist ein Symbolic Link.
* `-r` *file*: *file* existiert und ist lesbar.
* `-w` *file*: *file* existiert und ist schreibbar.
* `-x` *file*: *file* existiert und ist ausführbar.
* `-s` *file*: *file* existiert und ist nicht leer.
* `-O` *file*: *file* existiert und gehört mir.
* `-G` *file*: *file* existiert und gehört meiner Gruppe.
* *file1* `-nt` *file2*: *file1* ist "newer than" *file2*.
* *file1* `-ot` *file2*: *file1* ist "older than" *file2*.
* *file1* `-ef` *file2*: *file1* und *file2* sind Hard Links auf denselben File.
* `-t` *n*: File Nummer *n* geht auf das Terminal.
* `-z` *string*: *string* ist leer (Länge 0).
* `-n` *string* (oder nur *string*): *string* ist nichtleer.
* *string1* `=` *string2*: *string1* gleich *string2*.
* *string1* `!=` *string2*: *string1* ungleich *string2*.
* `!` *expr*: "Not" *expr*.
* *expr1* `-a` *expr2*: *expr1* "and" *expr2*.
* *expr1* `-o` *expr2*: *expr1* "or" *expr2*.
* *int1* `-eq`/`-ne`/`-lt`/`-le`/`-gt`/`-ge` *int2*: *int1* =/<>/</<=/>/>= *int2*.

Klammern sind auch erlaubt, müssen aber *gequotet* werden.
Es gibt keine `<` oder `>` Vergleiche.
Bei Variablen-Zuweisungen dürfen keine Leerzeichen rund um `=` stehen, bei Vergleichen *müssen* Leerzeichen rund um `=` stehen!

Wichtiger Trick: Wenn im Test vorkommende Variablen *leer* sind, kommt ein Syntaxfehler, weil ja dann z. B. auf einer Seite des `=` der Operand fehlt. Man sollte daher entweder alle Variablen quoten oder den `x`-Trick verwenden: `x$1 = xtest.txt`

10 `if`

`if` *befehle*
`then` *befehle*
[`elif` *befehle*
`then` *befehle*] ...
[`else` *befehle*]
`fi`

11 Restliche Statements

`exit` und `exit` *n*:
`exit` *n* beendet die Ausführung des Shellskripts mit Exit Code *n* (siehe Kapitel Exit Code betr. gültiger Werte für *n*) `exit` (ohne *n*) beendet die Ausführung des Shellskripts mit dem Exit Code des letzten ausgeführten Commands.

Achtung: Wird das Skript eingelesen (mit `.`) anstatt als Befehl aufgerufen, wird die Shell und damit die Terminalsitzung beendet, nicht nur das Shellskript!

`shift` und `shift` *n*:
`shift` schiebt die Argumente um eins nach vor: `$1` wird entfernt, `$2` wird zu `$1`, `$3` zu `$2`, usw., das letzte Argument `$`*m* wird undefiniert, und `$#` wird eins runtergezählt. `shift` *n* streicht die ersten *n* Argumente und schiebt die restlichen nach.

Nützlich ist das in Schleifen, um die Argumente eins nach dem anderen abzuarbeiten.

`for`-Schleife:
`for` *name* `in` *word* ...
`do` *befehle*
`done`

Belegt die Variable *name* der Reihe nach mit den einzelnen Worten hinter `in` und führt für jedes dieser Worte einmal den Schleifenrumpf aus. Läßt man `in` *word* ... weg, wird die Variable *name* der Reihe nach mit den einzelnen Argumenten `$1`, `$2` usw. belegt, und für jedes Argument wird die Schleife ein Mal ausgeführt.

In *word* wird Filename Expansion gemacht. Schreibt man zum Beispiel `for f in *.c`, so wird `f` der Reihe nach mit den Namen der `.c`-Files im aktuellen Verzeichnis belegt und die Schleife für jeden `.c`-File einmal durchlaufen.

`while`-Schleife:
`while` *befehle*
`do` *befehle*
`done`

Wie im `if` dient ein *beliebiger Befehl* (meist `[ ]`) als Bedingung, kein boolscher Ausdruck (Vergleich)!

`until`-Schleife:
`until` *befehle*
`do` *befehle*
`done`

Achtung: Die Bedingung wird so wie bei der `while`-Schleife *vor* dem Schleifendurchlauf geprüft (nur eben negiert), nicht nachher.

`break` und `break` *n*:
`break` verläßt die aktuelle Schleife: Die Ausführung wird unmittelbar nach Schleifenende fortgesetzt. `break` *n* verläßt die umgebenden *n* Schleifen.

`continue` und `continue` *n*:
`continue` beendet den gerade laufenden Schleifendurchlauf und springt zum Beginn des nächsten Schleifendurchlaufes der aktuellen Schleife: Die Ausführung wird am Kopf der Schleife (mit dem `while`- bzw. `until`-Test oder der nächsten Belegung der `for`-Variable) fortgesetzt. `continue` *n* bewirkt eine Fortsetzung mit dem nächsten Durchlauf der *n*-ten umgebenden Schleife.

Statementlisten:

- `;` trennt Statements innerhalb einer Zeile: *befehl1* `;` *befehl2* `;` ...
  Die Befehle werden der Reihe nach von links nach rechts ausgeführt, einer nach dem anderen. Ein `;` wirkt also wie ein Zeilenumbruch bzw. ersetzt diesen.
  Möchte man das `then` in dieselbe Zeile wie das `if` schreiben (bzw. das `do` gleich in die `for`-/`while`-Zeile), so braucht man einen `;` davor!
- `()` und `{}` gruppieren Statements bzw. umschließen Listen von Statements (der Unterschied zwischen `()` und `{}` wird später besprochen).
  Achtung: Der letzte Befehl vor einer `}` muß mit einem `;` enden, wenn die `}` nicht allein in einer Zeile steht!

Logische Operationen:

- `!` *befehl* führt *befehl* normal aus und negiert seinen Exit Code logisch: Aus `0` wird `1`, aus jedem anderen Wert `0`.
- *befehl1* `&&` *befehl2* führt *befehl2* nur dann aus, wenn *befehl1* mit Erfolg (`true`, Exit Code `=0`) endete. Der Exit Code des gesamten Konstrukts ist der Exit Code des zuletzt ausgeführten Befehls (welcher immer das auch war).
- *befehl1* `||` *befehl2* führt *befehl2* nur dann aus, wenn *befehl1* mit Mißerfolg (`false`, Exit Code `>0`) endete. Der Exit Code des gesamten Konstrukts ist der Exit Code des zuletzt ausgeführten Befehls (welcher immer das auch war).

Diese logischen Operationen werden üblicherweise nicht im Sinne logischer Operationen, sondern an Stelle von `if` für die bedingte Ausführung von Befehlen (*"Wenn ... dann mach ..."*) verwendet. Beispiele:

- Beende das Skript mit einer Fehlermeldung, wenn das `cp` schiefgeht:
  `cp $from $to || { echo "Copy from $from to $to failed!" >&2 ; exit 2 }`
- Archiviere das Zielverzeichnis, wenn ein `cd` dorthin erfolgreich war:
  `cd $dir && tar cf $tarfile .`

Fallunterscheidung:

`case` *word* `in`
pattern [`|` *pattern*] ...`)` *befehle* `;;`
...
`esac`

pattern sind Patterns wie bei der Filename Expansion (mit `*`, `?` und `[]`), sie werden aber *nicht* expandiert. Das Ergebnis von *word* wird der Reihe (von oben nach unten) nach mit den einzelnen *pattern* verglichen. Die *befehle* beim ersten passenden *pattern* werden ausgeführt, dann geht es hinter `esac` weiter (man braucht kein `break`). Für den Default-Fall schreibt man als letztes Pattern `*)` .

12 Command und Process Substitution

Kommt in einer Shell-Befehlszeile irgendwo ‘*command*‘ vor (‘ ist das verkehrte Anführungszeichen, daher auch der Ausdruck Backquoting statt Command Substitution), so wird — bevor der eigentliche Befehl in der Zeile ausgeführt wird! — zuerst einmal *command* ausgeführt, ‘*command*‘ textuell durch den Output von *command* ersetzt, und dann erst die ganze Zeile normal ausgeführt. Beispielsweise gibt also `echo ‘pwd‘` den Namen des aktuellen Verzeichnisses aus.

`$(`*command*`)` ist die modernere und heute empfohlene Schreibweise für genau das gleiche (also z. B. `echo $(pwd)`), sie läßt sich leichter schachteln.

Wenn das Ergebnis Zwischenräume enthalten kann, aber nicht in mehrere Argumente zerfallen soll, muß man das ganze Konstrukt unter `"` setzen, also z. B. `now="$(date)"`.

Nur neuere `bash`***:***
Wenn in einer Shell-Befehlszeile irgendwo `<(`*command*`)` oder `>(`*command*`)` vorkommt (*ohne* Leerzeichen zwischen `<` und `(` !), so wird *command* gestartet, sein Standard-Output (bzw. -Input) wird mit einer temporären, benannten Pipe verbunden, das ganze Konstrukt wird durch den Namen der Pipe (also einen normalen Filenamen) ersetzt, und die so entstandene Befehlszeile wird ausgeführt. Der ausgeführte Befehl glaubt also, einen ganz normalen File vor sich zu haben, liest aber in Wahrheit den Output von *command* (bzw. schreibt dessen Input).

Mit `diff oldls <(cd /backup ; ls -l)` kann man beispielsweise ein altes, im File `oldls` gespeichertes Directory-Listing mit dem aktuellen Output von `ls -l` vergleichen, ohne diesen eigens in einen File abzuspeichern.

13 `basename` und `dirname`

`basename` und `dirname` sind die am häufigsten mit Backquotes verwendeten Befehle:

`basename` *filename* liefert als Output den eigentlichen Filenamen (ohne davorstehende Verzeichnisse, d. h. alles ab dem hintesten `/`) von *filename*. Dabei wird nicht geprüft, ob es den angegebenen File wirklich gibt.
Beispiel: `basename scripts/hello.sh` liefert `hello.sh`.

`basename` *filename ext* arbeitet wie `basename`, prüft aber zusätzlich, ob *filename* auf *ext* endet: Wenn ja, wird *filename ohne ext* ausgegeben, sonst wie oben.
Beispiel: `basename scripts/hello.sh .sh` liefert `hello`.

`dirname` *filename* liefert als Output den Directory-Teil von *filename* (d. h. alles vor dem hintesten `/`).
Beispiel: `dirname scripts/hello.sh` liefert `scripts`.
Wenn *filename* gar keinen `/` enthält, liefert `dirname .` (in der Annahme, daß *filename* in diesem Fall einen File im aktuellen Verzeichnis bezeichnet).

14 `read` und `echo`

Mit `read` *name* ... kann man aus einem Shell-Skript *eine* Zeile vom Terminal einlesen und in der Variable *name* speichern.

Ist *name* nicht angegeben, wird die eingelesene Zeile in der Variable `REPLY` gespeichert.

Sind mehrere *name* angegeben, kommt das erste Wort der eingelesenen Zeile in die erste angegebene Variable, das zweite Wort in die zweite, usw., und alle verbleibenden Worte in die letzte. Werden weniger Worte eingegeben, als Variablen angeführt sind, werden die verbleibenden Variablen auf den Leerstring gesetzt.

Optionen (meist nur neuere `bash`!):

`-e` erlaubt beim Eingeben das Editieren der Input-Zeile.

`-n` *n* liest maximal *n* Buchstaben (üblicherweise 1) statt einer ganzen Zeile.
`-p` *str* gibt vor dem Einlesen *str* als Prompt aus.
`-s` unterdrückt das Echo der Eingabe am Terminal ("silent").
`-t` *n* setzt für die Eingabe ein Timeout von *n* Sekunden.

Der Exit Code ist `0`, wenn etwas eingelesen wurde, und `1`, wenn das Timeout abgelaufen ist oder beim Lesen End-of-File auftrat.

Wenn man den Prompt lieber mit `echo` ausgeben will oder eine Zeile Output aus mehreren `echo`-Befehlen zusammenstückeln will, ist `echo -n` hilfreich: Es unterdrückt das `\n` am Ende der Ausgabe von `echo`.

`echo -e` erlaubt Steuerzeichen wie in C im auszugebenden Text (Achtung: Der Backslash muß gequotet werden, sonst frißt ihn die Shell!). `\a` ist z. B. das Piepserl, `\e` ist das *esc* für Terminal-Steuersequenzen, `\n` der Zeilenvorschub für mehrzeilige Ausgabe mit einem einzigen `echo`.

Achtung: Es gibt keine allgemein verbreitete und portable Möglichkeit der Terminal-Steuerung von Shell-Skripts aus, da die *esc*-Sequenzen je nach Terminal bzw. Terminal-Emulationsprogramm unterschiedlich sind. Bei `xterm` und der Linux-Konsole (und mit Einschränkungen bei Telnet-Zugang) kann man sich aber auf die ANSI-Steuersequenzen verlassen, also z. B. `\e[2J\e[H` für Terminal löschen und Cursor nach links oben oder `\e[`*n*`m` für das Ein- und Ausschalten von Hervorhebungen (*n*: `0` ist Normalmodus, `7` ist Reverse, `4` ist Underline, `5` ist Blinking, zweistellige Werte setzen die Vorder- oder Hintergrundfarbe).

15 Funktionen

Definition von Funktionen:

funcname `()`
`{`
befehle
`}`

oder

`function` *funcname* `()`
`{`
befehle
`}`

Mit `return` kann man aus der Funktion zurückkehren, `return` *n* gibt zusätzlich den zurückzugebenden Exit Code an.

Funktionen müssen *vor* ihrer Verwendung definiert werden.
Rekursive Funktionsaufrufe sind zulässig.

Argumente und Returnwert:

Innerhalb einer Funktion bezeichnen `$1 $2 $3` usw. sowie `$*` und `$@` nicht die Argumente des Shell-Skripts, sondern die Argumente des Funktionsaufrufes (die Argumente des Shell-Skripts sind daher innerhalb eines Funktionsaufrufes nicht erreichbar). Die Anzahl der Argumente ist variabel und wird in `$#` gespeichert. Auch `shift` wirkt innerhalb der Funktion auf die Funktionsargumente, nicht die Shell-Argumente. Nach der Rückkehr aus der Funktion haben `$1` usw. und `$#` wieder ihren alten Wert.

`$0` bleibt unverändert (Name des Skripts oder Name der Shell) und wird *nicht* auf den Funktionsnamen gesetzt.

Shell-Funktionen haben keinen Returnwert, sondern nur einen Exit Code. Dieser ist gleich dem Exit Code des letzten in der Funktion ausgeführten Befehls oder dem bei `return` angegebenen Wert.

Aufruf von Funktionen:
Wie normale Shell-Befehle, ohne irgendwelche `()` für Argumente:
funcname arg1 arg2 ...

Nicht nur in Skripts möglich, sondern auch beim Eingeben von Befehlen vom Terminal aus!

Variablen in Funktionen:
Funktionen können beliebig Variablen der umgebenden Shell lesen und setzen. In der Funktion neu eingeführte Variablen sind ebenfalls global, bleiben also auch nach Ende des Funktionsaufrufes in der Shell erhalten.

Nur* `bash`*: Verwendet man im Code einer Funktion `local` *name* oder `local` *name*=*wert*, so ist die neue Variable *name* nur innerhalb der Funktion bekannt und verschwindet bei der Rückkehr aus der Funktion wieder. Gibt es schon eine globale Variable dieses Namens, wird eine neue, lokale Variable angelegt, und die globale Variable bleibt von der Funktion unberührt.

Anzeigen von Funktionen:
Mit `declare -f` oder `type`.

Gültigkeit von Funktionen: Gleich wie Variablen:

- Wenn man ein Shellskript, das eine Funktionsdefinition enthält, als Befehl ausführt, für den Rest des Skripts.
- Wenn man ein Shellskript, das eine Funktionsdefinition enthält, mit `.` einliest, für den Rest des Skripts und den Rest der Shell-Sitzung.
- Wenn man die Funktionsdefinition am Terminal eintippt, für den Rest der Shell-Sitzung.
- Normalerweise nur in der aktuellen Shell, nach `export -f` *funcname* auch in Subshells.
- Entfernen der Funktionsdefinition mit `unset -f` *funcname*.

16 `eval`

Nach dem Ersetzen von `$`*name* durch den Inhalt der Variable *name* werden auf dem eingesetzten Text zwar noch einige Prüfungen und Operationen ausgeführt (z. B. Aufteilen in einzelne Worte, je nachdem, ob `$`*name* in `""` stand oder nicht), aber vieles wird nicht mehr erkannt, beispielsweise im Variableninhalt enthaltene Pipes oder Redirects, weitere `$`-Konstrukte usw..

Für diese Fälle gibt es den Befehl `eval`, der seine Argumente noch einmal durch alle "Behandlungen" der Shell schickt und erst dann als Befehlszeile ausführt.

Anwendungen von `eval`:

- Ausführen einer in Variablen enthaltenen Befehlszeile (mit allen Pipes, Redirects und sonstigen Shell-Features), beispielsweise `read ; eval "$REPLY"` oder `eval "$@"`. Auf diese Weise kann man auch dynamisch Skript-Code zusammenbasteln und ausführen, z. B. Funktionsdefinitionen mit variablen Teilen.

- Nochmaliges Ersetzen von Variablen in Variablen-Inhalten: Nach `ls_files='$2 $4 $5 $7'` liefert `ls $ls_files` nicht das Gewünschte, `eval ls $ls_files` aber schon.
- Für "indirekte" Variablen-Zugriffe[3]: Wenn `varname` einen Variablen-Namen (ohne `$`) enthält, so kann man den Inhalt der betreffenden Variable mit `eval echo \$$varname` ausgeben. Mit `eval last=\${$#}` speichert man das letzte Argument in der Variable `last` ab.

17 "Klassische" Shell-Arithmetik

17.1 Der Befehl `expr`

Bis vor kurzem konnten die Shells alle nicht selbst rechnen: Arithmetische Operationen waren ausschließlich durch Aufruf des eigens für diesen Zweck geschaffenen externen Programms `expr` möglich.

`expr` kann Integer-Arithmetik (`+ - * / %`), Klammerung (`( )`), Vergleiche (`= != < <= > >=`) und logische Verknüpfungen (`& |`, einfach, nicht doppelt!). Daneben beherrschen moderne `expr` die String-Operationen `substr`, `index` und `length` sowie Pattern Matching mit Regular Expressions[4] (`:` oder `match`).

Die Rechnung wird als Argumente übergeben. Dabei muß jeder Operand und jeder Operator ein eigenes Wort (also durch *Zwischenraum* getrennt!) sein. Die Operatoren, die eine Bedeutung in der Shell haben (`* < > | & ( )`), müssen *gequotet* werden!

`expr` liefert das Ergebnis als Output, wird daher meist in Backquotes aufgerufen (siehe unten).

Achtung:
Für boolsche Werte gilt die gleiche Vereinbarung wie in C: `false` entspricht dem numerischen Wert `0`, jeder Wert ungleich `0` wird als `true` interpretiert. Das ist genau umgekehrt wie in der Shell selbst!

Dementsprechend ist auch der Exit Code von `expr` definiert:

- `0`, wenn das Ergebnis des Ausdruckes `true` (ein wahrer boolscher Ausdruck), eine von `0` verschiedene Zahl, oder ein nichtleerer String war.
- `1`, wenn das Ergebnis `false`, `0`, oder der Leerstring war.
- `2`, wenn ein Fehler aufgetreten ist (die Argumente beispielsweise keine korrekte Rechnung darstellen).

Beispiele:

- `count=‘expr $count + 1‘`
- `if expr $count \* $len \< 100 > /dev/null`
- `if [ "$(expr substr $fn $(expr $(expr length $fn) - 1) 2)" = ".c" ]`

---

[3] Die `bash` hat dafür ein besseres Feature: `${!`*varname*`}`

[4] Pattern Matching mit Regular Expressions ist eine sehr mächtige, aber oft sehr schwer zu lesende Möglichkeit, einen String auf einen bestimmten Aufbau oder bestimmte Teilstrings zu prüfen (ähnlich wie Wildcards). Aufbau und Bedeutung von Regular Expressions sind in der Man-Page von `grep` oder `ed` und hier in einem späteren Kapitel beschrieben.

17.2 Der Befehl `bc`

`bc` ist ein Programm, das eine C-ähnliche Programmiersprache mit Variablen, Verzweigungen, Schleifen, Arrays und Funktionen interpretiert. `bc` hat den Vorteil, mit *beliebig vielen* Vor- und Nachkommastellen und beliebigen Zahlenbasen rechnen zu können. `bc` liest das auszuführende Programm vom Input und liefert das Ergebnis als Output (`bc` ist als interaktives Programm und nicht als Shell-Tool gedacht).

Beispiele:

- `limit=‘echo ’2^128-1’ | bc‘`
- `echo ‘echo "scale=4; $schilling*13.7603" | bc‘ Euro`
- `bin=‘echo "ibase=16; obase=2; DEADBEEF" | bc‘`

Die meisten Implementierungen von `bc` rufen intern `dc` auf, ein Programm, das in etwa den gleichen Funktionsumfang, aber eine viel primitivere Eingabesyntax hat (nämlich stackorientierte RPN, d. h. *Operand Operand Operation*).

Im Prinzip kann man natürlich jede Programmiersprache, die rechnen kann, von der Shell aus zu diesem Zweck aufrufen, z. B. die Skriptsprache `awk` oder den Makro-Preprozessor `m4`:
`count=‘echo "incr($count)" | m4‘`

18 Arithmetik in der `bash`

Die `bash` (und teilweise auch ganz aktuelle `ksh`-Versionen) hat viele neue Features für Arithmetik (normale `int`-Arithmetik, keine Gleitkommazahlen):

Arithmetische Ausdrücke:

Es stehen alle Konstrukte von C zur Verfügung:

- Arithmetische Operationen `+ - * / %`, zusätzlich `**` für “hoch”, bitweise Operationen `~ & | ^ << >>`, logische Operationen `! && ||`, Vergleiche `== != < <= > >=` (**Achtung:** `==` *statt* `=` !), und der “Conditional Operator” `?:`.
- Increment `++` und Decrement `--`, vorne und hinten.
- Zuweisungen, sowohl mit `=` als auch mit `+=`, `*=` usw., beliebig geschachtelt innnerhalb von Ausdrücken.
- `( )` zur Schachtelung, `,` zur Aneinanderreihung mehrerer Ausdrücke oder Zuweisungen.
- Was mit `0` beginnt, ist eine Oktalzahl, bei `0x` ist es eine Hex-Zahl.

Weitere Vorteile:

- Shell-Sonderzeichen brauchen *nicht* gequotet zu werden!
- Variablennamen können innerhalb arithmetischer Ausdrücke *ohne* einem `$` davor verwendet werden!
- Zwischenräume sind egal, es geht mit oder ohne.

Wie in C und bei `expr` gilt bei dieser `bash`-Arithmetik: `0` entspricht `false`, jeder andere Wert wird als `true` interpretiert!

Arithmetische Ersetzung: `$(( `*expr*` ))`

expr wird als arithmetischer Ausdruck gelesen und ausgerechnet, und dann wird das ganze Konstrukt durch das Ergebnis ersetzt. Beispiel:
`count=$((count + 1))`

Die Befehle `let` und `(( ))` (ohne `$`!): `let` *expr1 expr2* ... und `(( `*expr*` ))`

Beide bewirken, daß die Ausdrücke als arithmetische Ausdrücke gelesen und ausgerechnet werden, also z. b. `let count=count+1` oder `(( count = count + 1 ))`.

Achtung: Bei `let` muß jeder Ausdruck genau *ein* Wort sein, also *keine* Zwischenräume verwenden oder quoten!

Achtung: `let` und `(( ))` setzen den Exit Code so wie `expr`: Liefert *expr* `0` (d. h. `false`), ist der Exit Code `1`, sonst `0`. Daher kann ein `(( ))` als numerischer Test in einem `if`, `while` etc. verwendet werden (ohne `test` oder `[`): `if (( count*len<100 ))`

Die arithmetische `for`-Schleife: `for (( `*expr1* `; `*expr2* `; `*expr3* `))`

Die Schleife funktioniert wie in C. *expr1*, *expr2* und *expr3* müssen arithmetische Ausdrücke wie in `let` oder `$(( ))` sein, keine Befehle oder Shell-Tests! Dementsprechend läuft die Schleife, solange *expr2* von `0` verschieden ist, bei `0` (`false`) endet sie.

Integer-Variablen: `declare -i` *name* und `local -i` *name*
Deklarieren die Variable *name* als Integer-Variable: Das ändert zwar nichts daran, daß die Variable auch nicht-Integer-Werte enthalten kann, aber es bewirkt, daß die rechte Seite einer Zuweisung auf diese Variable auch *ohne* `$(( ))` als arithmetischer Ausdruck behandelt wird!

Und weil wir gerade bei den `bash`-spezifischen Shell-Erweiterungen sind: Die `bash` hat auch einen "verbesserten" nichtnumerischen Test mit `[[ ]]` statt `[ ]` eingeführt:

- Statt `-a` und `-o` kann `&&` und `||` verwendet werden.
- Es wird sowohl `=` als auch `==` akzeptiert.
- `< > <= >=` können ungequotet verwendet werden, sie bewirken einen Vergleich der beiden Operanden als *String* (`(( ))` vergleicht sie hingegen als *Zahl*).
- Und das wichtigste: `*` und `?` werden nicht durch alle passenden Filenamen ersetzt, sondern bewirken in Kombination mit `=` *Pattern Matching* wie bei `case`: `if [[ $file == *.c ]]`

19 Variablen-Einsetzung

Um viele Fälle, die traditionell mit `eval`, `expr`, `basename` usw. gelöst werden müssen, komfortabler zu lösen, hat die `bash` (und einige aktuelle `ksh`-Implementierungen) viele Erweiterungen für das Einsetzen von Variablen mit `$`*name* definiert. Die wichtigeren davon:

Normale Variablen-Einsetzung: `${`*name*`}`
Wie `$`*name*. Die `{ }` sind notwendig, wenn entweder unmittelbar nach dem Namen weitere Buchstaben folgen (z. B. `${progname}log`), oder wenn es sich um eine mehrstellige Zahl handelt (statt `$10` muß man `${10}` verwenden).

Indirekte Variablen-Einsetzung: `${!`*name*`}`
Die Variable *name* enthält den *Namen* der Variable, deren Inhalt eingesetzt werden soll (statt `eval \$$`*name*).

Test auf Belegung: `${`*name*`:?}`
Ist *name* undefiniert oder gleich `""`, so bricht die Einsetzung mit einer Fehlermeldung ab.

Default-Wert: `${`*name*`:-`*wort*`}`
Ist *name* undefiniert oder gleich `""`, so wird statt dessen als Default-Wert *wort* eingesetzt.

Default-Zuweisung: `${`*name*`:=`*wort*`}`
Wie bei `:-`, aber zusätzlich wird *wort* auch in *name* gespeichert.

Alternativ-Wert: `${`*name*`:+`*wort*`}`
Ist *name* undefiniert oder gleich `""`, so wird `""` eingesetzt, hat *name* einen Wert, wird statt diesem *wort* eingesetzt.

Substring: `${`*name*`:`*offset*`}` `${`*name*`:`*offset*`:`*length*`}`
Es wird der Substring der Länge *length* des Inhalts von *name* beginnend an der Position *offset* eingesetzt. *offset* beginnt bei `0`, ist *offset* negativ, wird die Position vom Ende aus gezählt. Ist *length* nicht angegeben, geht der Substring bis zum Ende.
offset und *length* werden von der `bash` als *arithmetische Ausdrücke* behandelt.

Mehrere Argumente: `${@:`*offset*`}` `${@:`*offset*`:`*length*`}`
Setzt *length* Stück Argumente (oder alle restlichen) beginnend bei Argument Nummer *offset* ein.

Stringlänge: `${#`*name*`}`
Setzt die Stringlänge (als Zahl) des Inhalts von *name* ein.

Pattern-Löschung: `${`*name*`#`*pattern*`}` `${`*name*`##`*pattern*`}` `${`*name*`%`*pattern*`}` `${`*name*`%%`*pattern*`}`
pattern kann wie bei Filename-Wildcards `*` usw. enthalten und wird mit dem Inhalt von *name* verglichen. Das Ergebnis ist jener Teil des Inhalts von *name*, der *nicht* auf das *pattern* paßt (bzw. alles, wenn *pattern* überhaupt nicht paßt). Mit anderen Worten: *pattern* wird aus dem Inhalt von *name* gelöscht. Bei `#` wird der kürzeste passende Teil am Anfang gesucht, bei `##` der längste passende Teil am Anfang, bei `%` der kürzeste, passende Teil am Ende, und bei `%%` der längste passende Teil am Ende.
Beispiele: `${filename##*/}` liefert den Basename (ohne Directories davor), `${filename%.*}` schneidet die Extension (falls vorhanden) ab.

Substitution: `${`*name*`//`*pattern*`/`*string*`}` `${`*name*`/#`*pattern*`/`*string*`}` `${`*name*`/%`*pattern*`/`*string*`}`
Wie im vorigen Punkt, aber das Pattern wird im Ergebnis nicht gelöscht, sondern durch *string* ersetzt (überall / nur am Anfang / nur am Ende).
Beispiel: `${cfile/%.c/.h}` liefert den Inhalt von `cfile` mit der Extension `.c` (falls vorhanden) durch `.h` ersetzt.

20 Arrays

Die `bash` (und nur diese) kennt auch Array-Variablen. Details siehe `man bash`, das Wichtigste in Kürze:

- Verwendung: `${`*name*`[`*index*`]}` oder `${`*name*`[*]}` oder `${`*name*`[@]}`.
- Zuweisung: *name*`[`*index*`]=`... oder *name*`=(`*elem1 elem2* ...`)`.
- Deklaration: Implizit durch Zuweisung oder explizit mit `declare -a` *name* oder `local -a` *name*.
- Index: Läuft ab `0`, Größe ist variabel, Löcher sind erlaubt.

Nützliche Sonderfälle:

- Der Exit Code jedes einzelnen Befehls der letzten Pipe ist in der vordefinierten Array-Variable `PIPESTATUS` gespeichert.
- `read -a` *name* liest wortweise in aufeinanderfolgende Elemente des Arrays *name*.

21 Subshells

Viele Operationen macht die `bash`, die den Terminalinput bzw. ein Skript verarbeitet, nicht selbst, sondern startet dafür eine weitere `bash` (im Wesentlichen eine Kopie ihrer selbst, also mit dem gleichen Zustand), und übergibt ihr bestimmte Befehle zur Ausführung. Diese temporär gestarteten Hilfs-Shells nennt man *Subshells*. Sie starten zwar mit einer exakten Kopie der Daten der ursprünglichen Shell, verwalten ihre Daten dann aber unabhängig von der startenden Shell.

Dieser Mechanismus hat folgende Effekte:

- Neue Variablen, Funktionen oder Aliases, die in einer Subshell definiert werden, bleiben in der übergeordneten Shell *unbekannt*. Auch jegliche Zuweisungen auf bereits bestehende Variablen (oder Änderungen bestehender Funktionen und Aliases) in einer Subshell wirken sich *nicht* auf die übergeordnete Shell aus: Die betreffenden Variablen behalten dort ihren alten Wert! Ebenso verändert ein `shift` in einer Subshell die Argumente der übergeordneten Shell nicht!

  Der einzige Weg, Daten von einer Subshell in die übergeordnete Shell zu transferieren, führt über den Exit Code, den Output der ausgeführten Befehle, oder über Files.

- Änderungen im Zustand der Shell in der Subshell wirken sich nicht auf die übergeordnete Shell aus. Zum Zustand einer Shell gehören vor allem das Current Directory (ein `cd` in einer Subshell verändert also das Current Directory der übergeordneten Shell *nicht*!), die Werte von `umask` und `ulimit`, die `trap`-Einstellungen, und die mit `set` oder `shopt` gesetzten Flags und Optionen.

Was wird in einer Subshell verarbeitet?

- Alle Elemente einer Pipeline (eine Subshell pro Element!)[5].

  Variablen-Zuweisungen innerhalb einer Pipeline haben also *nicht* den erwarteten Effekt!

- Die Befehle in Backquotes bzw. `$( )`.

- Alle Befehle innerhalb von `( )`.

  `( )` werden daher oft bewußt verwendet, um die Wirkung eines `cd` oder `umask` auf einen einzigen (oder wenige) Befehle zu beschränken:
  `( cd $fromdir && tar cf - . ) | ( cd $todir && tar xpf - )` war lange Zeit üblich, um Verzeichnisse samt Unterverzeichnissen zu kopieren, ohne das aktuelle Verzeichnis zu verändern.

  Gegenteil: Mit `{ }` zusammengefaßte Befehle werden von der gleichen Shell wie die umgebenden Befehle abgearbeitet, *ohne* Subshell!

- Ein als Befehl (d. h. nur mit Angabe des Namens, ohne `.`) ausgeführtes Shell-Skript.

[5] Früher haben manche Shells den ersten oder den letzten Befehl einer Pipeline selbst ausgeführt.

Gegenteil: Mit `.` *name* oder `source` *name* (bewirkt beides das gleiche) eingelesene Skripts werden von der aktuellen Shell verarbeitet, nicht von einer Subshell.

Wenn ein Skript daher Funktionen oder Variablen bleibend (und nicht nur innerhalb des Skripts) definieren soll, muß es mit `.` eingelesen und nicht als Befehl ausgeführt werden! Auch die Profiles `.bashrc` und `.bash_profile` werden beim Start der Shell von dieser eingelesen, nicht ausgeführt.

Das ist auch der Grund, warum ein `exit` in einem Skript beim Ausführen als Befehl das Ende des Skripts bewirkt (es wird die Subshell beendet), beim Einlesen hingegen das Ende der Terminalsitzung (es wird die Haupt-Shell beendet!).

Ein `.` bzw. `source` kann auch innerhalb von Skripts verwendet werden, um andere Skripts einzulesen, und wirkt wie ein `#include` in C. Man kann also immer wieder benötigte Hilfsfunktionen in einem eigenen File zusammenfassen, und diesen File dann einfach am Anfang jedes Skriptes mit `source` inkludieren.

22 Regular Expressions

Regular Expressions sind etwas ähnliches wie die Filename-Wildcards `*`, `?` usw.. Allerdings sind sie nicht speziell für Filenamen gedacht, sondern für beliebige Textstrings[6].

Man spricht auch von ***Pattern Matching***: Im Englischen heißt es *"a string matches a pattern"*, d. h. "Ein String paßt zu einem Pattern", wenn der String die vom Pattern (= Regular Expression) vorgegebene Struktur hat.

Verwendung:
Die Suche in `vi` verwendet Regular Expressions zur Angabe des zu suchenden Textes, `expr` und `awk` können einen String mit einer Regular Expression vergleichen, `grep` und `egrep` durchsuchen Textfiles nach Zeilen, die zu einem angegebenen Pattern passen, in `sed` und `awk` geben Regular Expressions im Skript die zu bearbeitenden Zeilen an, usw..

Leider unterscheiden sich die Regular Expressions in den einzelnen Programmen in manchen Details und bieten unterschiedliche Fähigkeiten und Erweiterungen; vorsichtshalber `man`-Page lesen. Im wesentlichen gibt es zwei Grundtypen: *Basic Regular Expressions* und *Extended Regular Expressions*, Unterschied siehe unten.

In den meisten Programmen erfolgt das Pattern Matching zeilenweise: Jede Zeile für sich wird mit dem Pattern verglichen, der passende Text kann sich daher nicht über mehrere Zeilen erstrecken. Mit anderen Worten: `\n` gilt *nicht* als beliebiger Buchstabe (in `awk` gilt das nicht immer, und manche Programme erlauben explizite `\n` im Pattern, um mehrzeilige Pattern zu definieren).

Nachdem Regular Expresions meist jede Menge Sonderzeichen enthalten, die normalerweise von der Shell verarbeitet werden (`*`, `?`, `$`, `[]` etc.), sollte man sie auf der Commandline und in Shellskripts tunlichst unter `' '` setzen, damit sie von der Shell unverändert an das jeweilige Programm übergeben werden.

Aufbau von Regular Expressions:

char
: Ein normaler Buchstabe *char* (außer `.`, `*`, `[`, usw.) steht für sich selbst.

[6] Deshalb wird `/` — im Unterschied zu Wildcards — nicht gesondert behandelt, sondern gilt als ganz normaler Buchstabe.

char

Analog, für jene *char*, die sonst eine Sonderbedeutung hätten (`.`, `*`, `[`, usw.).

Achtung: Manche Zeichen bekommen erst in Kombination mit `\` eine Sonderbedeutung (siehe unten)!

`.`

Auf einen `.` paßt ein beliebiger, einzelner Buchstabe (normalerweise außer `\n`).

`[`*range*`]`

Zu diesem Konstrukt passen all jene einzelnen Buchstaben, die in *range* enthalten sind. *range* kann eine Aneinanderreihung einzelner Buchstaben, ein mit `-` angegebener Bereich oder ein Konstrukt wie `[:alnum:]` oder `[:xdigit:]` sein.

`[^`*range*`]`

Analog, für alle Zeichen, die nicht in *range* sind.

Achtung: `^`, nicht `!` verwenden!

regexp1 regexp2

Zu diesem Konstrukt (ohne Zwischenraum!) passen all jene Strings, die einen zu *regexp1* passenden Teil unmittelbar gefolgt von einem zu *regexp2* passenden Teil enthalten.

`^`*regexp*

Ein String paßt zu diesem Konstrukt, wenn er auf *regexp* paßt und am Zeilenanfang steht.

Achtung: `^` selbst entspricht *keinem* Zeichen des entsprechenden Strings, sondern sagt nur "hier muß der Zeilenanfang sein".

regexp`$`

Analog, für *regexp* am Zeilenende.

`(`*regexp*`)`

Zu diesem Konstrukt passen all jene Strings, die auf *regexp* passen. `( )` dienen zur Gruppierung von Teil-Expressions für Operatoren wie `*` oder `|`, oder zur Markierung von Teilen von Regular Expressions für spätere Verweise mit `\1` bis `\9` (wird hier nicht besprochen).

regexp`*`

regexp`+`

regexp`?`

Zu diesem Konstrukt passen all jene Strings, die 0 oder mehr / 1 oder mehr / 0 oder 1 aufeinanderfolgende, zu *regexp* passende Teile enthalten.

regexp`{`*n*`}`

regexp`{`*n*`,}`

regexp`{`*n*`,`*m*`}`

Zu diesem Konstrukt passen all jene Strings, die genau *n* / mindestens *n* / *n* bis *m* aufeinanderfolgende, zu *regexp* passende Teile enthalten.

regexp1`|`*regexp2*

Zu diesem Konstrukt passen all jene Strings, die entweder zu *regexp1* oder zu *regexp2* passen.

In *Basic Regular Expressions* haben `+ ? | { } ( )` keine Sonderbedeutung, sondern werden als normale Buchstaben behandelt (d. h. passen zu sich selbst). Für die oben angeführte Sonderbedeutung muß man `\+ \? \| \{ \} \( \)` verwenden. In *Extended Regular Expressions* ist es genau umgekehrt: `+ ? | { } ( )` haben die jeweilige Sonderbedeutung, `\+ \? \| \{ \} \( \)` stehen für sich selbst.

23 Allgemeine Anmerkungen zu Unix-Befehlen

- Die folgenden Kapitel sind nur eine Auswahl der unter Unix/Linux zur Verfügung stehenden Befehle. Es gibt viel mehr!
- Die folgenden Kapitel enthalten nur eine überblicksartige Beschreibung der Befehle. Für alle Funktionalitäten, Optionen und Aufruf-Varianten bitte die `man`-Page des jeweiligen Befehls lesen!
- Grundsätzlich gilt in Unix: Optionen beginnen mit `-` und stehen immer **vor** den Filenamen.

  Ursprünglich bestanden alle Optionen aus genau *einem* Buchstaben, mehrere Options-Buchstaben konnten hinter einem einzigen `-` zusammengefasst werden.

  Heute sind die Details der Optionen (längere Options-Namen, Optionen mit Argumenten, ...) leider von Befehl zu Befehl verschieden.
- Viele Befehle akzeptieren beim Aufruf `-` statt eines Filenamens und lesen dann ihren Input von `stdin` (d. h. zum Beispiel aus einer Pipe) statt aus einem File. Ebenso erwarten die meisten Befehle den zu verarbeitenden Input auf `stdin`, wenn gar kein Filenamen angegeben wird[7].
- Ein `--` kann verwendet werden, um das Ende der Optionen anzuzeigen: Alles nach `--` wird als Filename interpretiert und nicht als Option, selbst wenn es mit `-` beginnt. Dieser Trick ist notwendig, falls man es mit Files zu tun hat, deren Namen mit `-` beginnt.

24 Einfache File-Befehle

`ls`: Directory Listing.
`cp`: Files/Directories kopieren.
`mv`: Files/Directories umbenennen oder verschieben.
`rm`: Files/Directories löschen.
`mkdir`: Directories anlegen.
`rmdir`: Directories löschen.
`chown`, `chgrp`, `chmod`: Eigentümer und Rechte ändern.
`touch`: Datum setzen, File leer anlegen.
`ln`: Links erzeugen.
`cd`: Verzeichnis wechseln.
`pwd`: Aktuelles Verzeichnis anzeigen.

25 Einfache Text-Befehle

`cat`: Gibt den Inhalt der angegebenen Files (oder den Input) nacheinander unverändert auf `stdout` aus. Kann mit Optionen Zeilen numerieren und mehrfache Leerzeilen entfernen.
`tac`: Wie `cat`, aber jeder File wird von unten nach oben (letzte Zeile zuerst) ausgegeben.

---

[7] Das ist aber (im Unterschied zur Filename Expansion, d. h. Wildcards mit `*`, `?` usw.) Sache des jeweiligen Programms und nicht der Shell. Ob das funktioniert, ist daher von Programm zu Programm verschieden.

`tail`: Zeigt das hintere Ende jedes Files / des Inputs an (per Default: Die letzten 10 Zeilen, angebbar in Zeilen oder Bytes). Sonderfall `-f`: Schaut einem File / dem Input beim Wachsen zu.

`head`: Analog, aber für das vordere Ende.

`od` ("octal dump"): Zeigt die Files / den Input in Oktal / Hexadezimal / Dezimal / \-Notation / ... an.

`strings`: Zeigt die in Binärdaten enthaltenen ASCII-Strings an.

`file`: Versucht, die Art der angegebenen Files zu erraten.

`split`: Teilt einen großen File in mehrere kleinere.

`wc`: Zählt Zeilen, Worte und Buchstaben; ermittelt die Länge der längsten Zeile.

`seq` *n*: Zählt bis *n*.

26 Komplexe Text-Befehle

`diff` *file1 file2* **oder** `diff` *dir1 dir2*: Vergleicht die angegebenen Files (oder paarweise Files gleichen Namens in den angegebenen Directories) zeilenweise. Versucht auch weit auseinanderliegende gleiche Stücke zu finden. Viele Optionen zur Kontrolle des Vergleiches (Groß/Kleinschreibung ignorieren, Leerzeilen ignorieren, Zwischenräume ignorieren, ...), zur Wahl des Algorithmus (schnell/gründlich) und zur Wahl des Ausgabe-Formates ("normal", als "Kontext-Diff", als Editor-Skript, als File mit C-Preprozessor-Befehlen, ...)

`cmp` *file1 file2*: Vergleicht Files binär.

`grep` *regexp file* ...: Sucht in den angegebenen Files / im Input / in allen Files eines Directories Zeilen, die auf die angegebene *regexp* passen. Viele Optionen zur Kontrolle der Suche und des Outputs (z. B. Groß/Kleinschreibung ignorieren, nur nicht passende Zeilen ausgeben, nur Anzahl der Treffer ausgeben, Zeilennummer mit ausgeben, umgebende Zeilen mit ausgeben, ...).

`egrep`: Analog, aber für Extended Regular Expressions statt Basic Regular Expressions.

`fgrep`: Analog, aber für Strings statt Regular Expressions.

`tr` *charset1 charset2*: Verschiedene Varianten:

- Ersetzt alle Zeichen aus einer Menge durch das entsprechende Zeichen der anderen Menge.
- Ersetzt alle Zeichen (nicht) aus einer Menge durch ein bestimmtes Zeichen.
- Löscht alle Zeichen aus einer Menge.
- Ersetzt aufeinanderfolgende gleiche Zeichen im Output durch ein einziges Zeichen.

Achtung: `tr` akzeptiert keine Input-Filenamen, sondern arbeitet immer von `stdin` nach `stdout`!

`cut`: Extrahiert bestimmte Felder oder Spalten aus den angegebenen Files / dem Input. Angabe entweder durch absolute Spaltennummer oder durch Feldnummer relativ zu einem Feld-Trennzeichen (Zwischenraum, `:`, ...).

`sort`: Sortiert den Inhalt der angegebenen Files / den Input zeilenweise. Viele Optionen für die Art der Sortierung. Kann mehrere bereits sortierte Files effizient zusammenmischen sowie gleiche Zeilen aus dem sortierten Output entfernen.

Achtung: Das Ergebnis wird auf `stdout` ausgegeben, nicht in die jeweiligen Files zurückgeschrieben!

`uniq`: Diverse Operationen auf benachbarten, identen Zeilen in *sortierten* Files.

Achtung: Nimmt nicht mehrere Input-Files als Argument, sondern nur einen Input- und einen Output-File!

`comm` *file1 file2*: Vergleicht *sortierte* Files zeilenweise und liefert nach Wunsch folgende Output-

Spalten: Zeilen nur im ersten File / nur im zweiten File / in beiden Files.

`paste`**:** Fügt Files zeilenweise in mehreren Spalten zusammen (erste Spalte: Zeilen aus erstem File, zweite Spalte: Zeilen aus zweitem File usw.).

`join` *file1 file2*: Konstruiert aus jenen Zeilen der Input-Files, die in einem bestimmten Feld gleich sind, die Output-Zeilen (ähnlich einem Datenbank-Join). Die Input-Files müssen nach dem zu vergleichenden Feld sortiert sein.

27 Weitere Shell-Befehle

`pushd`, `popd` **und** `dirs`**:** Die `bash` verwaltet intern nicht nur ein Current Working Directory, sondern einen ganzen Stack davon:

`pushd` *dir* merkt sich das aktuelle Verzeichnis auf diesem Stack und macht ein `cd` in das angegebene Verzeichnis *dir*.

`popd` holt das oberste Verzeichnis vom Stack und macht ein `cd` dorthin, kehrt also in das Verzeichnis vor dem letzten `pushd` zurück.

`dirs` zeigt alle in diesem Stack gespeicherten Verzeichnisse an.

`exec`**:** Dieser Befehl hat zwei Funktionen:

`exec` *command arg* ... führt das angegebene *command* aus, und zwar *statt* der gerade laufenden Shell. Folglich kehrt ein `exec`-Aufruf nie mehr zum Shell-Skript zurück und ist daher nur als letzter Befehl eines Skripts sinnvoll.

`exec` *redirects* (ohne ein *command*!) führt die angegebenen Redirects für das Shell-Skript selbst aus, d. h. sie gelten für den Rest des Skriptes: Nach `exec > scriptlog` landet alles, was das Skript in Folge an Output produziert, im File `scriptlog`.

`type`**:** Mit `type` *command* kann man anzeigen, ob *command* ein Alias, eine Funktion, ein Shell-Schlüsselwort, ein eingebauter Befehl der Shell, oder ein Programm oder Skript ist. Bei letzterem wird der komplette Pfad angezeigt, d. h. in welchem Verzeichnis sich das Programm oder Skript befindet. Dabei werden die in `PATH` angegebenen Verzeichnisse durchsucht.

`command`**:** Mit `command` *command* wird der Befehl *command* ausgeführt (*command* kann ein eingebauter Befehl oder ein Programm oder Skript sein), auch wenn es eine Funktion mit dem Namen *command* gibt.

Das ist nützlich, wenn man Funktionen mit dem gleichen Namen wie Befehle hat: Würde man nur *command* aufrufen, würde die Shell die Funktion *command* ausführen, und nicht den Befehl *command*.

`printf`**:** `printf` *format arg* ... dient ähnlich der C-Funktion `printf` zur formatierten Ausgabe (schön spaltenweise, in oktal oder hex, ...): *format* ist ein Format-String mit Fixtext und `%`-Formatbeschreibungen, und zu jedem `%` liefert ein *arg* den auszugebenden Wert.

`getopts`**:** Der Befehl `getopts` kann zum Verarbeiten von `-`-Optionen beim Aufruf eines Skripts verwendet werden.

28 Weitere File- und Diskbefehle

`du` *file* ... ("disk usage") zeigt den Platzverbrauch der angegebenen Files und Directories (samt aller Subdirectories) an, wahlweise in Bytes, Disk Blocks (512 Bytes), Kilobytes, oder Megabytes.

Der Output von `du` basiert nicht auf der logischen Länge der Files, sondern auf dem tatsächlich belegten Plattenplatz (dieser wird normalerweise größer als die Länge sein, weil Unix für einen File üblicherweise Plattenplatz in 4-Kilobyte-Stücken reserviert; bei Files mit "Löchern" kann er auch wesentlich kleiner sein).

`df` ("disk free") zeigt alle gemounteten Filesysteme, deren Mount Point, deren Größe sowie deren Füllstand an.

`stat` *file* ... zeigt die gesamte zu den angegebenen Files gespeicherte Directory-Information an.

29 `find` und `xargs`

`find` *dir* ... *opt* ... sucht in den Directories *dir* ... samt Subdirectories nach Files und Directories, die den in *opt* angegebenen Bedingungen entsprechen. Sind keine Directories *dir* angegeben, wird das aktuelle Verzeichnis `.` durchsucht, sind keine Bedingungen angegeben, werden alle in den angegebenen Directories gefundenen Einträge ausgegeben.

Die Aufrufsyntax von `find` weicht von den üblichen Unix-Standards ab: Erstens stehen die zu durchsuchenden Directories vorne und die Optionen hinten, zweitens bestehen die Optionen nicht aus einzelnen Buchstaben, sondern ganzen Worten (obwohl sie hinter einfachen `-` stehen, und nicht hinter `--`), und drittens spielt die Reihenfolge der Optionen eine Rolle: Die Bedingungen werden für jeden File von links nach rechts geprüft.

Die Details von `find` sind in der `man`-Page nachzuschlagen, hier nur ein Auszug:

Bedingungen:

Name: Man kann sowohl den Basename des Files mit Wildcards prüfen (z. B. `-name "*.c"`, bzw. `-iname "*.doc"`, wenn Groß- und Kleinschreibung ignoriert werden soll) als auch den gesamten Pfadnamen (beispielsweise `-path "*/include/*.h"`). Mit `-regex` kann man echte Regular Expressions statt Wildcards verwenden.
Achtung: `*` usw. müssen *gequotet* werden, weil sie ja an das `find` übergeben werden sollen, ohne daß sie die Shell vor dem Aufruf durch passende Filenamen ersetzt!

Filetype (File, Directory, Link, Device, ...): Beispielsweise `-type l`.

Datum der letzten Änderung / des letzten Zugriffs: Z. B. `-daystart -mtime -7`.

Änderungsdatum relativ zu einem File: `-newer timestamp` vergleicht das Änderungsdatum mit dem Änderungsdatum des Files `timestamp`.

Größe: Beispielsweise `-size +1024k`.

Rechte: Beispielsweise `-perm 0777` (man kann auf bestimmte Rechte oder Mindestrechte prüfen).

Besitzer und Gruppe: Beispielsweise `-user k.kusche`.

Anzahl der Hard Links: Beispielsweise `-links +2`.

Inode-Number: Beispielsweise `-inum 4711`.

Zielnamen des Symlinks: `-lname lostfile` zeigt alle Symlinks an, die auf einen File namens `lostfile` verweisen.

Logische Verknüpfungen: Mit `!` oder `-not` kann man Bedingungen negieren, `-o` bewirkt eine Oder-Verknüpfung, einfaches Aneinanderreihen von Bedingungen oder `-a` gilt als Und-Verknüpfung.

Aktionen:

- Per Default werden die Namen der gefundenen Files auf `stdout` ausgegeben, 1 Name pro

Zeile. Man kann sie in einen File umleiten, mit `less` seitenweise anzeigen, mit `xargs` weiterverarbeiten usw..
- Speziell für `xargs` kann man als letzte Option `-print0` angeben. Dann werden die Filenamen durch Null-Bytes (`\0`) statt Newlines (`\n`) getrennt. Das vermeidet Probleme, die sonst auftreten, wenn die Filenamen Zwischenräume, Newlines oder andere Steuerzeichen enthalten.
- Mit `-ls` werden die gefundenen Files wie bei einem `ls -dils` gelistet, mit `-printf` kann man selbst angeben, welche Attribute der gefundenen Files wie ausgegeben werden sollen.
- Mit `-exec` kann man für jeden gefundenen File einen Befehl ausführen (die Syntax ist allerdings etwas kryptisch), `-ok` bewirkt das gleiche, fragt den Benutzer aber für jeden gefundenen File, ob der Befehl wirklich ausgeführt werden soll. Beides wird heute kaum noch verwendet; dasselbe läßt sich mit `xargs` eleganter und effizienter erreichen.

`xargs` *command* ... liest von `stdin` eine Liste von Filenamen (diese können mittels `|` aus dem Output eines `find` kommen, oder z. B. mittels `<` aus einem File) und führt für diese Files das angegebene *command* aus, indem es die Filenamen hinten an das *command* (und eventuell vorhandene Argumente oder Optionen) anhängt.

Normalerweise werden so viele Filenamen bei jedem einzelnen Aufruf von *command* angegeben, wie die durch das Betriebssystem vorgegebene Befehlszeilen-Länge erlaubt, mit `-l` wird *command* für jeden Filenamen einzeln aufgerufen, mit `-n` *n* kann man die Anzahl der Files pro Aufruf angeben.

Weitere Optionen von `xargs` sind `-0` (die Filenamen am Input sind nicht durch Zwischenraum oder Zeilenvorschub getrennt, sondern durch Null-Bytes, für `find ... -print0`) und `-p` (Rückfrage vor jeder Ausführung von *command*).

Beispiele:

- Durchsuche alle `.h`-Files in `/usr/include` nach dem String `printf`:
  `find /usr/include -name "*.h" | xargs fgrep printf`
- Lösche alle Files in `/tmp`, die seit mehr als 3 Tagen nicht mehr geändert wurden:
  `find /tmp -mtime +3 -type f | xargs rm -f`

30 Archive und Komprimierung

Das heute unter Unix am weitesten verbreitete Tool, um Files und Directories in einen einzigen Archiv-File zu verpacken (z. B. zwecks Kopie zwischen Rechnern, Archivierung auf Bändern oder CD's, oder Bereitstellung zum Download im Internet) ist `tar` ("Tape Archiver"). Derartige Files haben üblicherweise auch die Extension `.tar`.

Ein früher ebenso weit verbreitetes Archiv-Programm und -Format war `cpio` (`tar` entstammt dem BSD-Unix-Zweig, `cpio` war das Gegenstück dazu bei den SysV-Unix-Abkömmlingen).

Beide Formate sind an sich genormt (es gibt aber sehr viele nicht genormte Erweiterungen), das Open-Source-Programm `pax` kann beide Formate lesen und schreiben.

Weiters gibt es auch unter Unix ein Open-Source-Programm `zip` bzw. `unzip`, das komprimierte Archive in dem unter Windows üblichen `zip`-Format lesen und schreiben kann (das `zip`-Format ist aber in der Unix-Welt höchst unüblich).

Im Unterschied zu dem in der Windows-Welt verbreiteten Programm `zip` komprimieren `tar` und `cpio` ihre Archive *nicht*, die Komprimierung/Dekomprimierung der Archive wird unabhängig von

deren Erstellung/Entpackung in einem eigenen Schritt von eigenen Programmen erledigt[8]. Diese Komprimierungs-Programme wiederum behandeln immer nur einen File für sich (in vielen Fällen eben ein `tar`-Archiv), ohne selbst irgendwas mit Archiven zu tun zu haben:

- In den Anfängen von Unix wurde mit den Programmen `pack` und `unpack` komprimiert/dekomprimiert, dieses Verfahren kommt heute nicht mehr vor.
- Das Standard-Unix-Programm zur Kompression ist `compress` (bzw. `uncompress` zum Dekomprimieren und `zcat` zum Lesen komprimierter Files). Derart komprimierte Files bekommen üblicherweise die Extension `.Z` angehängt.
  Nachdem das Verfahren von `compress` durch Patente geschützt ist und Lizenzgebühren verlangt werden, wurde `compress` im Bereich der Open-Source-Software in den letzten Jahren fast völlig verdrängt.
- Das erste bei Linux am weitesten verbreitete Kompressionsprogramm war `gzip` (GNU Zip) (beziehungsweise `gunzip` und `gzcat`). Derartige Files erhalten die Endung `.gz`, solcherart komprimierte `tar`-Archive also `.tar.gz` (oder auf CD-Filesystemen, die mehrfache Extensions nicht erlauben, `.tgz`).
  Wie schon erwähnt, kann `gzip` im Unterschied zu seinem Windows-Namensvetter nur *einen* File in einen komprimierten `.gz`-File verwandeln.
- Das effektivste Komprimierungsprogramm war lange Zeit `bzip2` (samt `bunzip2` und `bzcat`), die damit komprimierten Files heißen normalerweise `.bz2`. `bzip2` ist Open Source. Es hat sich in der Open-Source-Welt durchgesetzt, in der kommerziellen Unix-Welt ist es fast völlig unbekannt.
- Aktuell wird im Open-Source-Bereich auf das noch bessere `xz` umgestellt, so komprimierte Files heißen auch `.xz`.

Neben den reinen Archiv-Formaten hat fast jeder Unix-Hersteller sein eigenes Format für Software-Installations-Pakete (die neben den zu installierenden Files auch Installationsscripts, Paketabhängigkeitsinformation usw. enthalten). Unter Linux hat sich weitgehend das `rpm`-Format durchgesetzt, das dazugehörige Programm zur Installation, Verwaltung und Deinstallation von `rpm`-Paketen heißt ebenfalls `rpm`.

Verwendung von `tar`***:***
GNU `tar` ist inzwischen ein umfangreiches Programm mit sehr vielen Optionen (`man`-Page lesen!), aber für den Anfang reicht folgendes:

- Erstellen eines Archivs *tarfile*, das alle Files und Verzeichnisse im Verzeichnis *dir* enthält:
  `tar cf` *tarfile dir*
  (es gibt auch eine Möglichkeit, die Liste der zu archivierenden Files von einem `find` an ein `tar` zu übergeben)
- Anzeigen des Inhaltsverzeichnisses des Archivs *tarfile*:
  `tar tvf` *tarfile*
- Extrahieren aller Files und Verzeichnisse aus dem Archiv *tarfile*:
  `tar xf` *tarfile*
- Dasselbe, für ein `.tar.bz2`-Archiv:
  `bzcat` *tarbz2file* `| tar xf -`

Vorsicht: Wird ein `tar`-Archiv mit absoluten Filenamen erstellt (`/` am Anfang), werden die Files von den meisten `tar`-Programmen auch wieder an dieselbe Stelle (mit demselben absoluten Pfad)

[8] Die GNU-Version von `tar` hat zwar eine Option `z`, um `gzip`-komprimierte `tar`-Archive direkt lesen und schreiben zu können, diese bewirkt aber nichts anderes, als daß `tar` intern `gzip` aufruft (analog mit Option `j` für `bzip2` und `J` für tt xz).

extrahiert. Wird das Archiv mit relativen Pfadnamen erstellt, werden die Files und Verzeichnisse beim Auspacken auch relativ zum aktuellen Verzeichnis angelegt.

31 sed

`sed` (der "Stream Editor") führt Skript-gesteuert (d. h. nicht-interaktiv) Editor-Operationen auf dem Inhalt der angegebenen Files bzw. wenn keine Files angegeben sind auf `stdin` aus und liefert den derart modifizierten Input als Output (d. h. die Input-Files selbst werden nicht verändert).

Normalerweise arbeitet `sed` zeilenweise: Es wird eine Zeile eingelesen, dann werden alle in Frage kommenden Befehle auf die aktuelle Zeile angewendet, und danach wird das Ergebnis ausgegeben[9]. Per default erfolgt das Ausgeben jeder Zeile nach der Verarbeitung automatisch, mit der Option `-n` kann man es unterdrücken: Dann werden nur jene Zeilen ausgegeben, für die man das mit dem Befehl `p` ausdrücklich will.

Die `sed`-Befehle kann man:

- Entweder mit der Option `-e` beim Aufruf von `sed` auf der Commandline angeben (gequotet!): Man braucht pro Befehl (bzw. pro Befehlszeile bei mehrzeiligen Befehlen) ein `-e`. Hat man nur einen einzigen Befehl, kann man das `-e` weglassen und nur den Befehl alleine angeben.
- Oder zeilenweise in einen File schreiben. Dann kann man entweder `sed -f` *scriptfile inputfile* ... aufrufen, oder man gibt dem Skriptfile selbst `x`-Rechte und läßt ihn mit `#!/bin/sed -f` (oder wo immer `sed` liegt) beginnen, dann kann man `sed`-Skripts selbst direkt als Befehl ausführen.

Jede `sed`-Befehlszeile besteht aus zwei Teilen: Keiner, einer oder zwei Adressen und dem eigentlichen Befehl.

- Ist *keine* Adresse angegeben, so gilt der Befehl für jede Input-Zeile.
- Ist *eine* Adresse angegeben, so wird der Befehl nur auf jenen Input-Zeilen ausgeführt, für die die Adresse gilt.
- Sind *zwei* Adressen (mit `,` getrennt) angegeben, so wird der Befehl für alle Zeilen von einschließlich jener, für die die erste Adresse gilt, bis zu einschließlich jener, für die die zweite Adresse gilt, ausgeführt.
- Mit `!` kann man die Wirkung von Adressen umkehren: Bei einer Adresse wirkt der Befehl dann auf alle Zeilen, für die die Adresse nicht gilt, bei zwei Adressen auf alle Zeilen außerhalb des von-bis-Bereiches.
- Eine Adresse kann sein:
  - Eine Zeilennummer (bzw. `$` für die letzte Input-Zeile).
  - `/`*regexp*`/` (d. h. eine Regular Expression zwischen `/ /`). Diese Adresse gilt für alle Input-Zeilen, die zu *regexp* passen.

`sed` kennt eine Vielzahl von Befehlen (bis hin zu `goto`'s für Schleifen), die wichtigsten sind folgende:

`s/`*alt*`/`*neu*`/g`**:** "substitute globally": Ersetzt alle Vorkommen von *alt* durch *neu* (*alt* ist eine Regular Expression, *neu* kennt `&` und `\1` usw. für den Text, der auf die Regular Expression passte).

[9] Es gibt auch Befehle, um mehr als eine Zeile in den Puffer für die aktuelle Zeile einzulesen. Weiters gibt es einen zweiten Zeilenpuffer zum "Merken" von Zeilen, in den man den Puffer für die aktuelle Zeile sichern kann und von dem man ihn wieder holen kann.

`d`: "delete": Löscht die Zeile.
`p`: "print": Gibt die Zeile aus (wenn die automatische Ausgabe mit `-n` abgeschaltet ist).
`q`: "quit": Beendet `sed` (die restlichen Input-Zeilen werden nicht mehr verarbeitet).
`a` *textzeilen*: "append": Gibt die angegebenen *textzeilen nach* der aktuellen Zeile aus.
`i` *textzeilen*: "insert": Gibt die angegebenen *textzeilen vor* der aktuellen Zeile aus.
`c` *textzeilen*: "change": Gibt die angegebenen *textzeilen statt* der aktuellen Zeile aus.

32 `awk` Grundlagen

Name: Nach den Herrn Aho, Weinberger, Kernighan aus den Bell Labs, den Erfindern von `awk`.

Bei uns: `awk` = `gawk` (GNU `awk`)

Anwendung: Hauptsächlich für

- kurze "Ad-hoc-Prögrämmchen"
- und alles, was irgendwie Textfiles verarbeitet.

Vor- und Nachteile:

+ Komfortabel, ergibt kurze Programme.
+ Leicht lesbar.
+ Gut dokumentiert.
– Nicht typsicher, für große Programme schlecht geeignet.
– Ineffizienter als C.

Aufruf: Wie bei `sed` 3 Möglichkeiten:

- `awk` *scriptcode inputfile* ...
- `awk -f` *scriptfile inputfile* ...
- Skript in einen File schreiben, der mit `#!/bin/awk -f` beginnt, diesem File Ausführrechte geben, und ihn direkt als Befehl aufrufen.

Input und Output: Wie unter Unix üblich: `awk` verarbeitet die angegebenen *inputfile* der Reihe nach zeilenweise (ist kein *inputfile* angegeben, wird `stdin` gelesen) und liefert seinen Output auf `stdout` (die Inputfiles werden also nicht verändert).

Grundsätzlicher Aufbau eines `awk`-Skripts: Prinzipiell ähnlich wie ein `sed`-Skript: Ein `awk`-Skript besteht aus ein oder mehreren Konstrukten der Form *pattern* `{` *code* `}`.

Jedesmal, wenn eine Inputzeile gelesen worden ist, geht `awk` alle *pattern* von oben nach unten durch und vergleicht sie mit der Inputzeile. Wenn ein *pattern* paßt, wird der entsprechende *code* ausgeführt (passen mehrere, werden eben mehrere Codestücke für diese Zeile ausgeführt).

Enthält das Skript `{` *code* `}` ohne ein *pattern*, so wird *code* für *jede* Inputzeile ausgeführt. Enthält es ein *pattern* ohne `{` *code* `}`, wird automatisch `print` als *code* verwendet, d. h. die auf *pattern* passenden Inputzeilen werden unverändert ausgegeben.

Daneben kann ein `awk`-Skript noch Kommentarzeilen (beginnend mit `#`) sowie Funktionsdefinitionen (siehe unten) enthalten.

Patterns: `awk` kennt wie `sed` Regular Expressions in Schrägstrichen (`/`*regexp*`/`) und Zeilenbereiche mit Angabe eines Patterns für die Anfangs- und Endzeile (`/`*regexp*`/, /`*regexp*`/`). Daneben erlaubt `awk` auch *boolsche Ausdrücke* statt Patterns: Ergibt der Ausdruck für die aktuelle Inputzeile `true`, wird der dazugehörige *code* ausgeführt.

Statt eines Patterns sind auch die Schlüsselworte `BEGIN` und `END` erlaubt: Der *code* nach `BEGIN` wird beim Start des `awk`-Skripts ausgeführt (bevor die erste Inputzeile gelesen wird), der *code* nach `END` wird am Ende des Programms nach Verarbeitung der letzten Inputzeile ausgeführt.

Der Programmcode: Ist fast ident zu C:

Ausdrücke: `awk` kennt alle arithmetischen und logischen Operatoren von C zuzüglich `^` für Potenzierung. Weiters gilt der Zwischenraum als *String-Konkatenations-Operator*: Stehen zwei Ausdrücke durch Zwischenraum getrennt nebeneinander, werden ihre Ergebnisse beide in Strings verwandelt und zusammengehängt. Weiters gibt es noch einen Operator für Pattern Matching.

Zuweisung: Wie in C, einschließlich der kombinierten Formen.

Statements: `awk` kennt `if`, `while`, `do`, `for`, `break`, `continue`, `return`, `next` (Bearbeite nächste Zeile) und `exit` (Programmende), sowie `{ }` zur Gruppierung von Statements.

I/O-Statements: Siehe `man`-Page:

- `print` gibt die aktuelle Inputzeile aus.
- `print` *expr*`,` ... gibt das Ergebnis von *expr* ... als eine Zeile aus[10].
- `printf` *format*`,` *expr*`,` ... macht das gleiche mit den in C üblichen Formatierungen.
- Alle drei Ausgabebefehle wirken normalerweise auf `stdout`, auf Wunsch auf `stderr`, einen namentlich angegebenen File, oder per Pipeline auf einen Unix-Befehl.
- `getline` liest eine Zeile (vom aktuellen Inputfile, aus einem namentlich angegebenen File, oder aus einem Unix-Befehl per Pipeline).

Variablen:

- `awk` kennt Variablen mit Namen wie in C.
- Variablen werden automatisch angelegt, wenn sie das erste Mal verwendet werden, man braucht sie nicht deklarieren.
- Variablen in `awk` sind typlos: Kommt eine Variable in einem arithmetischen Ausdruck vor, wird sie als Gleitkommazahl betrachtet (und automatisch in eine solche verwandelt, falls die Variable bisher ein String war), kommt sie in einem String-Ausdruck vor, wird sie als String betrachtet (und ev. von einer Gleitkommazahl in einen String zurückverwandelt).
- Wird eine Variable das erste Mal verwendet, ist sie mit `""` bzw. `0` initialisiert.
- Variablen sind global und leben bis zum Programmende. Möchte man lokale Variablen in selbstdeklarierten Funktionen, gibt man diese als zusätzliche Parameter im Funktionskopf an (`awk` beschwert sich nicht, wenn man eine Funktion mit weniger Argumenten aufruft als deklariert wurden).

String-Konstanten: Wie in C: Unter `"`, mit `\` für Sonderzeichen.

Vordefinierte Funktionen: Siehe `man`-Page:

- Mathematische Funktionen wie `sin`.
- String-Funktionen wie `substr`, `length`, ... (anders als in C!).

---

[10] Mehrere durch Zwischenraum getrennte `expr` werden direkt aneinandergehängt, mehrere durch `,` getrennte `expr` werden mit je einem Zwischenraum dazwischen ausgegeben.

- Zeit-Funktionen.

Selbstdefinierte Funktionen: Man kann mit `function` *funcname* `(` *params* `)` `{` *code* `}` selbst Funktionen definieren. In *params* werden die Namen der Parameter (und der lokalen Variablen) der Reihe nach durch `,` getrennt ohne Angabe eines Typs angeführt.

Achtung: Beim Aufruf einer Funktion darf man in `awk` zwischen dem Funktionsnamen und der `(` ***keinen*** Zwischenraum lassen!

Input-Fields, vordefinierte Variablen, Arrays: Siehe nächstes Kapitel.

Beispiel: Gib die Gesamtanzahl der Zeilen aller Inputfiles aus:

```
    { nlines++ }
END { print nlines }
```

33 `awk` Sonderthemen

Vordefinierte Variablen:

In `awk` gibt es einige Variablen mit vordefinierem Namen und fixer Bedeutung (siehe `man`-Page):

- Manche dieser Variablen werden von `awk` automatisch mit bestimmten Werten belegt: `ARGC` und `ARGV` (die Argumente des `awk`-Aufrufs), `ENVIRON` (die Environment-Variablen), `FILENAME` (Name des aktuellen Inputfiles), `NR`, `FNR` und `NF` (Zeilennummer und Feldanzahl, siehe unten), ...
- Andere dieser Variablen steuern das Verhalten von `awk`: `RS` und `FS` (Zeilen- und Feldtrennzeichen beim Einlesen, siehe unten), `ORS` und `OFS` (analog beim Ausgeben), `IGNORECASE` (Berücksichtigung der Groß- und Kleinschreibung beim Pattern Matching), ...

Input:

`awk` liest und verarbeitet Input zeilenweise. `$0` liefert (wie eine Variable) den gesamten Inhalt der aktuellen Zeile, die Variable `FNR` enthält die Zeilennummer der aktuellen Zeile innerhalb des aktuellen Inputfiles, `NR` enthält die Zeilennummer der aktuellen Zeile insgesamt.

Weiters zerlegt `awk` jede Zeile beim Einlesen automatisch in Felder (wobei die Felder durch ein oder mehrere Leerzeichen oder Tabs getrennt sind, und Leerzeichen und Tabs am Anfang und Ende der Zeile ignoriert werden). Den Inhalt der einzelnen Felder der aktuellen Zeile kann man mit `$1`, `$2` usw. ansprechen, die Anzahl der Felder in der aktuellen Zeile steht in der Variable `NF`. Ein Zugriff auf ein Feld mit höherer Nummer liefert keinen Fehler, sondern `""`.

Die Zahl nach dem `$` muß keine Konstante sein, es ist beispielsweise auch `$(i+1)` erlaubt (`$NF` spricht daher das letzte Feld an). Durch Zuweisung auf `$1` usw. kann man das entsprechende Feld auch ändern, bei der Ausgabe der Zeile wird dann der neue Inhalt des Feldes ausgegeben.

Das Trennzeichen zum Lesen der einzelnen Zeilen steht in `RS`, die Regular Expression zum Trennen der Felder in `FS`. Beides kann man bei Bedarf umsetzen, beispielsweise bei durch `:` getrennten Feldern (Details siehe `awk`-Handbuch!).

Arrays:

Das mächtigste und ungewöhnlichste Feature von `awk` sind seine Arrays:

- Die Notation zum Zugriff auf Arrayelemente ist ident zu C: *feldname*`[`*index*`]`
- Allerdings kann der *index* in `awk` ein ***beliebiger String*** sein, auch Zahlen als Indexwerte werden intern in einen String konvertiert.

  De facto ist ein Array in `awk` also kein Array, sondern eine Menge von Schlüssel/Wert-Paaren, auf die über den Schlüssel direkt zugegriffen werden kann: Der Index ist der Schlüssel, und der Inhalt des betreffenden Elements ist der zu diesem Schlüssel gehörige Wert.

  Das bedeutet auch, daß die Elemente eines Arrays in `awk` *nicht* in Index-Reihenfolge geordnet sind: Sie haben keinerlei Reihenfolge, selbst wenn nur Zahlen als Index verwendet wurden (dafür können die Index-Werte auch negative oder "löchrige" Zahlenfolgen sein)!
- Multidimensionale Arrays mit beliebig vielen Dimensionen sind ebenfalls möglich, alle Indices werden intern mit einem Trennzeichen (per Default das nicht druckbare Zeichen `'\034'`) zu einem einzigen Index-String verkettet. `a[1,2]` wird also als `a["1\0342"]` gespeichert.
- Arrays in `awk` müssen nicht deklariert werden und wachsen bei Bedarf dynamisch.
- Greift man auf ein Element zu, das es nicht gibt (d. h. verwendet man einen String als Index, mit dem noch nie ein Element im Array gespeichert wurde), so liefert dieser Zugriff keinen Fehler, sondern `""`.

  Als Nebeneffekt wird dieses Element dabei allerdings mit dem Wert `""` tatsächlich im Speicher angelegt (Speicherplatzverschwendung!).
- Um zu prüfen, ob ein Array einen Wert für einen bestimmten Index enthält, sollte man daher nicht *feldname*`[`*index*`]!=""` verwenden.

  Für diesen Zweck gibt es das Konstrukt *index* `in` *feldname* (bzw. `(`*index1*`,` *index2*`)` `in` *feldname* für mehrdimensionale Arrays).
- Um alle Elemente eines Arrays zu bearbeiten, gibt es das Schleifenkonstrukt `for (` *var* `in` *feldname* `)`. Dabei wird *var* bei jedem Durchlauf mit einem neuen in *feldname* vorkommenden Indexwert (nicht Elementwert!) belegt (in beliebiger Reihenfolge); die Schleife endet, wenn sie alle Elemente durchlaufen hat.
- Normalerweise bleiben einmal angelegte Array-Elemente bis zum Programmende erhalten; mit `delete` *feldname*`[`*index*`]` kann man sie gezielt löschen. Dabei wird nicht nur der Wert des Elementes gelöscht, sondern das ganze Paar (d. h. *index* kommt nachher in *feldname* nicht mehr vor).

34 Prozesse, Jobs, Signale

Prozesse:

Was ist ein Prozeß?

- Ein Prozeß ist ein in Ausführung befindliches Programm (Code + Daten) (läuft oder wartet).
- Ein Prozeß ist die Verwaltungseinheit des Betriebssystems für CPU-Zuteilung, Speicherplatz, offene Files, Rechte, ...

Daher:

- Ein Programm wird erst mit dem Starten ein Prozeß (oder mehrere), ein Programmtext oder Executable File an sich ist noch kein Prozeß.
- Jeder Prozeß befindet sich zu jedem Zeitpunkt an genau einer Codestelle genau eines Programmes, aber ein Programm kann zu einem Zeitpunkt von mehreren Prozessen ausgeführt werden (entweder, weil es mehrmals unabhängig voneinander gestartet wurde, oder weil es sich nach dem Start intern in mehrere gleichzeitige Abläufe aufgespalten hat). Jeder Prozeß hat dabei seine eigene aktuelle Codestelle im Programm, unabhängig von den anderen.
- Prozesse können auf Mehrprozessor-Maschinen echt gleichzeitig (parallel) ausgeführt werden, auf Einprozessor-Maschinen werden sie durch das Betriebssystem gesteuert reihum stückerlweise nacheinander ausgeführt.

Jeder Prozeß ist gekennzeichnet durch eine eindeutige, vom Betriebssystem beim Starten des Prozesses zugeteilte Nummer (***"Pid"*** = *"Process Identifier"*).

Weiters merkt sich das Betriebssystem für jeden Prozes auch die Pid des "Vaterprozesses", d. h. jenes Prozesses, der diesen Prozeß gestartet hat (*"PPid"* = *"Parent Pid"*).

Außerdem merkt sich das System für jeden Prozeß den Eigentümer (dieser bestimmt u. a., was der Prozeß darf und was nicht) und einige weitere Systeminformationen, z. B. das Controlling Terminal.

Die Befehle `ps`, `pstree` und `top`:

`ps` zeigt Prozesse in Listenform an (`ps` hat dutzende Optionen, häufige Aufrufe sind `ps -ef` oder `ps aux`).

`pstree` zeigt die Prozesse als Baumstruktur an (d. h. in der Hierarchie, in der sie erzeugt wurden).

`top` zeigt eine periodisch aktualisierte, nach CPU-Verbrauch, Speicherverbrauch oder anderen Kriterien sortierte Prozeßliste (sowie den aktuellen Maschinenstatus) an.

Signale:

Ein Signal ist ein (meist nicht vorhersagbares, von außen ausgelöstes) Ereignis, das einen einzelnen Prozeß betrifft. Normalerweise beendet ein Signal den betreffenden Prozeß (in einigen Fällen mit `core`, d. h. Speicherdump), es kann aber auch ignoriert werden oder die Ausführung einer eigens für dieses Signal geschriebenen Funktion im Programm bewirken.

Hat ein Befehl durch ein Signal geendet, gibt die Shell eine entsprechende Meldung aus.

Beispiele für Signale:

`SIGINT` **(2):** Am Terminal wurde `Ctrl/C` gedrückt ("`interrupted`").
`SIGHUP` **(1):** Das Terminalfenster bzw. die Terminalverbindung wurde beendet.
`SIGTERM` **(15):** Der Prozeß soll beendet werden ("`terminated`").
`SIGKILL` **(9):** Das gleiche, nicht abfangbar ("`killed`").
`SIGSEGV` **(11),** `SIGBUS` **(10) und** `SIGILL` **(4):** Das Unix-Gegenstück zur 'Schutzverletzung': Programmabsturz...
`SIGPIPE` **(13):** Das Programm hat versucht, in eine Pipe zu schreiben, die keiner mehr liest ("`broken pipe`").

Der Befehl `kill`:

Mit `kill` kann man einem oder mehreren Prozessen (nur eigenen Prozessen!) händisch ein Signal schicken. Die Argumente von `kill` sind die Pid's der betreffenden Prozesse; als Option kann das Signal angegeben werden, z. B. `-9` oder `-INT`. Gibt man nichts an, wird `SIGTERM` verwendet.

Der Shell-Befehl `trap`:

Mit `trap` kann man angeben, wie die Shell (bzw. ein Shellscript) auf das Eintreffen eines Signals reagieren soll:

`trap ""` *signal ...*
: bewirkt, daß die angegebenen Signale völlig ignoriert werden (beispielsweise, um heikle Programmbereiche vor `Ctrl/C` zu schützen).

`trap` *signal ...* **oder** `trap -` *signal ...*
: bewirkt, daß für die angegebenen Signale wieder die Default-Reaktion (normalerweise Abbruch des Skripts) eingerichtet wird.

`trap` *command signal ...*
: bewirkt, daß beim Eintreffen eines der angegebenen Signale (nachdem der gerade laufende Befehl geendet hat) das angegebene *command* ausgeführt wird. Das kann Aufräum-Code sein, ein `exit` mit einem bestimmten Exit-Code, eine Rückfrage, oder was immer. Nach der Ausführung von *command* endet das Skript.

Shell Jobs:

So, wie das Betriebssystem Prozesse verwaltet, verwaltet die Shell Jobs: Ein Job entspricht einem gerade ausgeführten Befehl (und kann aus einem oder mehreren Prozessen bestehen).

- Wird eine normale Befehlszeile eingetippt (oder in einem Script ausgeführt), so läuft sie normalerweise im Vordergrund, d. h. erst wenn der Befehl fertig ist, gibt die Shell wieder den Prompt aus und nimmt den nächsten Befehl entgegen (oder führt den nächsten Befehl im Skript aus).

- Endet die Befehlszeile hingegen mit `&`, wird sie als *Hintergrund-Job* ausgeführt: Die Shell gibt sofort wieder den Prompt aus (bzw. nimmt sofort den nächsten Befehl im Skript in Angriff); der Hintergrund-Job und die Shell arbeiten unabhängig voneinander und gleichzeitig weiter.

  Die Shell schreibt eine Meldung (mit der Jobnummer), wenn ein Job im Hintergrund gestartet wird oder endet. In `$!` steht die Pid des letzten gestarteten Hintergrund-Prozesses.

- Es kann immer nur einen Vordergrund-Job geben, aber beliebig viele Hintergrund-Jobs.

- Der Vordergrund-Job kontrolliert das Terminal, und bekommt den Tastaturinput sowie die von der Tastatur ausgelösten Signale (z. B. `Ctrl/C`). Die Hintergrund-Jobs bekommen keinen Tastaturinput und sind von `Ctrl/C` nicht betroffen.

  Bei den meisten Unix-Systemen werden Hintergrund-Jobs defaultmäßig automatisch *angehalten*, wenn sie versuchen, auf das Terminal zu schreiben, bis sie in den Vordergrund geholt werden. Unter Linux dürfen sie per Default auch aus dem Hintergrund am Terminal ausgeben[11]. Beim Lesen vom Terminal werden sie immer gestoppt.

[11] Mit `stty tostop` läßt sich das steuern.

- `jobs -l`
  Listet die gerade existierenden Jobs: Jobnummer, Pid, Status, Befehl.
- `Ctrl/Z`
  Stoppt den gerade im Vordergrund laufenden Job und macht einen *angehaltenen (!)* Hintergrund-Job daraus. Die Shell gibt einen Prompt aus, man kann normal weiterarbeiten. Mit `fg` kann man den angehaltenen Job wieder in den Vordergrund holen, mit `bg` kann man ihn im Hintergrund weiterlaufen lassen.
- Jobs können mit *%jobnumber* angesprochen werden:
  * `fg` *%jobnumber*
    holt den angegebenen (statt den zuletzt verwendeten) Job in den Vordergrund.
  * `bg` *%jobnumber*
    läßt den angegebenen Job im Hintergrund weiterlaufen.
  * `kill` *[signal]* *%jobnumber*
    schickt dem angegebenen Job das angegebene Signal.
  * `wait` *%jobnumber*
    wartet, bis der angegebene Job endet. `wait` allein wartet, bis *alle* gestarteten Jobs fertig geworden sind.