Mitschrift
Open-Source-Programmierwerkzeuge

12. August 2015

Inhaltsverzeichnis

1 Versionsverwaltung

1.1 Warum Versionsverwaltung?

Das Benutzen von Versionsverwaltung erlaubt:

- einen „sauberen“ Stand zu generieren
- Änderungen seit dem x-ten Release zu ermitteln
- zu prüfen, wie eine Lösung früher umgesetzt wurde
- zu prüfen, wann eine Änderung gemacht wurde und von welchem Entwickler

- einen alten Stand zu generieren, um eine Kundenbeschwerde bzgl. einer älteren Programmversion nachzuvollziehen
- verschiedene Entwicklungsstränge zusammenzuführen

1.2 Vergleich mit Kopien von Verzeichnissen

Kopien mit Datum oder Versionsstempel:

- gleiche Dateien werden jedes Mal erneut abgelegt, das ist Platzverschwendung
- Vergleich möglich, aber keine Historie
- für gemeinsame Entwicklung ungeeignet

Archivierung mit Hardlinks:

- Vergleich möglich, aber keine Historie
- für gemeinsame Entwicklung ungeeignet

1.3 Beispiele für Versionsverwaltungssysteme

- „Kopien archivieren“
- SCCS (Source Code Control System)
- RCS (Revision Control System)
- CVS (Concurrent Version System)
- Subversion (svn)
- bitkeeper (bk)
- Git
- Bazaar (bzr)
- Mercurial (hg)
- Microsoft Visual Source Safe
- PVCS/Serena Version Manager

1.4 Terminologie

Das **Repository** ist der Ort, an dem das Versionsverwaltungssystem die versionierten Dateien und alle nötigen Metainformationen ablegt. Man könnte es als „Lagerhaus“ für alle Versionen bezeichnen.

Im Arbeitsverzeichnis liegt eine Versionskopie der Daten aus dem Repository. Jeder Entwickler hat ein eigenes Arbeitsverzeichnis. Hier finden Entwicklungen und Tests statt und von hier werden Änderungen ins Repository übertragen. Daten aus dem Repository ins Arbeitsverzeichnis zu holen bezeichnet man als **checkout**, manchmal als **get**. Sie im Repository zu hinterlegen als **commit** oder als **checkin**. Neuere Versionen aus dem Repository holt man sich mit **update**. Verschiedene „Arten“ desselben Standes im Repository bezeichnet man als **Zweig** oder **Branch**. Wenn zwei Entwickler dieselbe Datei an derselben Stelle ändern und committen, bekommt der zweite einen Konflikt. Um einen bestimmten Entwicklungsstand zu markieren, versieht man ihn mit einem Tag. Hierbei handelt es sich um einen Namen, den man der Dateiversion gibt, z.B. „beta-0.2“, „release-1.0“, etc.

Verschiedene Entwicklungzweige werden durch einen **Merge** zusammengeführt. So lange keine Konflikte vorliegen, kann die Versionsverwaltung bei Textdateien den Großteil der Arbeit automatisch erledigen. Für Binärdateien geht das nicht.

Wird ein Stand aus dem Repository benötigt, der keine Administrationsdaten der Versionsverwaltung enthält, so ist manchmal von einem **Export** die Rede.

Um Konflikte (gerade bei Binärdateien) zu vermeiden, kann man einen **Lock** auf eine Datei setzen, um sie gegen Einchecken anderer zu schützen. Je nach Versionsverwaltung können andere Benutzer das Lock entfernen oder nicht. In jedem Fall ist Kommunikation zwischen den Entwicklern angebracht.

Je nach Versionsverwaltung sehen Versionsnummern unterschiedlich aus. Meist hat eine Datei eine Version wie „1.13“; eine andere Datei kann die Version „1.2“ haben, eine dritte „3.15“. Bei einfachen Branches entstehen Versionsnummern wie „1.10.1.3“’ oder „1.2.2.4“.

Bei Subversion hat nicht jede Datei ihre eigene Versionsnummer, sondern der komplette Stand im Repository. Wenn von zehn Dateien nur zwei geändert und eingecheckt werden, bekommen trotzdem alle Dateien im Repository eine neue Version. Bei acht davon ist der Inhalt der gleiche wie vor dem Einchecken. Die Versionsnummer ist eine einfache (natürliche) Zahl wie „42“ oder „3305“.

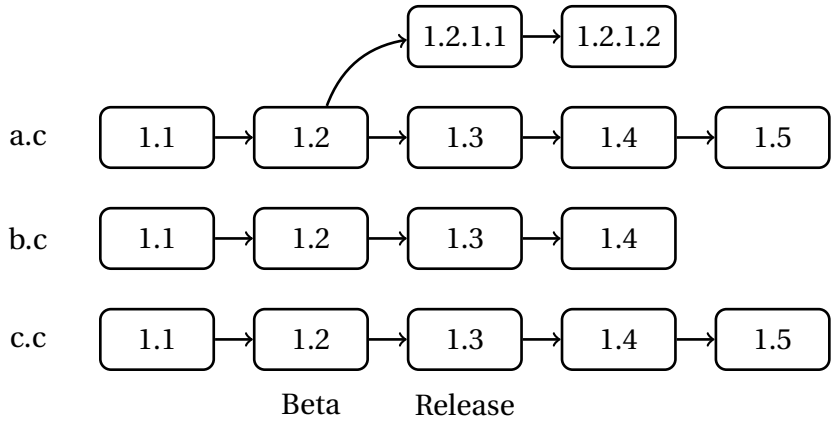

Abbildung 1: Branches am Beispiel von CVS

1.5 Branches

Wie kann man sich Branches vorstellen? Abbildung 1 zeigt es am Beispiel von CVS.

Die Verwendung von Branches kann aus verschiedenen Gründen sinnvoll sein: Bevor eine Version veröffentlicht wird, ist möglicherweise eine Testphase nötig. In dieser Zeit werden keine weiteren Features hinzugefügt, sondern nur Fehler behoben. Wenn dieser Teil von den bzw. nicht von allen Entwicklern, sondern von einer Testabteilung übernommen wird, können nicht alle Parteien auf demselben Stand arbeiten - entweder nur Korrekturen oder auch Erweiterungen. Deswegen wird ein Branch erzeugt: der aktuelle Stand wird innerhalb des Repositorys kopiert und speziell markiert abgelegt. Die Tests und Fehlerkorrekturen werden in diesem Zweig erledigt. Währenddessen dürfen im Hauptzweig neue Features entwickelt und eingecheckt werden. Die Korrekturen aus dem Releasezweig fließen ebenfalls ein. Möglicherweise werden aus dem Releasezweig Patches für die Software erstellt.

Wenn ein Feature größere Änderungen erfordert, kann es sinnvoll sein einen Branch zu erstellen. Dort können problemlos Änderungen durchgeführt und eingecheckt werden, ohne die Stabilität des Hauptzweiges zu beeinträchtigen. Damit die Abweichungen nicht zu groß werden, sollten regelmäßig die Änderungen im Hauptzweig in den Entwicklungszweig übernommen werden, damit keine Konflikte entstehen, bzw. die Konflikte leichter zu beheben sind. Stellt sich im Laufe der Entwicklung heraus, dass das Feature nicht oder nicht auf die vorgesehene Weise umsetzbar ist, kann man den Entwicklungszweig einfach verlassen. Hat man Erfolg, werden die Änderungen in den Hauptzweig über-

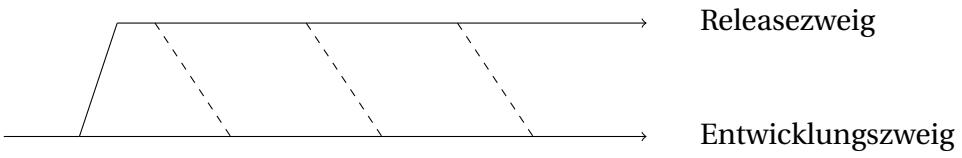

Abbildung 2: Korrekturen fließen vom Releasezweig in den Entwicklungszweig ein, während im Entwicklungszweig neue Features hinzugefügt werden

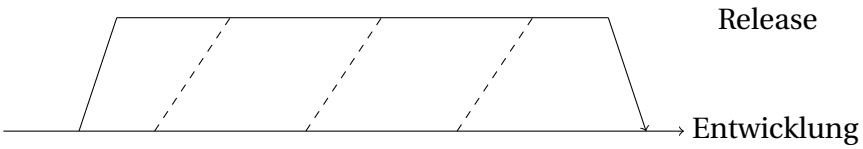

Abbildung 3: Neue Features des Entwicklungszweiges werden in den Releasezweig eingespeist. Nach der Testphase werden beide mit einem Merge zusammengeführt.

nommen und der Entwicklungszweig gelöscht (damit man nicht versehentlich versucht, die Änderungen nochmals zu integrieren). Branches sind manchmal auch sinnvoll, um eine Spezialversion einer Datei zu erstellen, z.B. für eine speziell angepasste Produktversion [Special Edition]. Nicht immer lässt sich so ein Unterschied auf andere geeignete Weise im Buildprozess unterbringen; ein Branch kann einfacher sein.

1.6 Wie Änderungen gespeichert werden (können)

In CVS werden Änderungen als „Patches“ zwischen den Versionen gespeichert. Die neueste Version wird im Klartext gespeichert, ältere werden bei Bedarf aus den Patches erzeugt. Diese Vorgehensweise verkleinert in der Regel den zur Archivierung nötigen Speicherplatz, erlaubt aber dennoch schnellen Zugriff auf die neueren Stände (der viel öfter passiert als der Zugriff auf ältere Stände). Da bei CVS das Repository im Dateisystem einsehbar ist, kann man theoretisch Änderungen an einer Datei nachvollziehen, ohne CVS danach zu fragen. Als Beispiel folgt ein Auszug aus einer solchen Datei: (hw.c, v)

```
head 1.3;
access;
symbols
  initial: 1.1.1.1 current: 1.1.1.1;
locks; strict;
comment @ * @

1.3
date 2009.04.15.21.07.56; author hg 10597; stateExp;
branches;
next 1.2
desc
@@
...
1.3
log
@changedprintf to puts
@
text
@#include <stdio.h>

int main(int argc, char *argv[]) {
    if(argc > 1) {
        printf(stderr, "no arguments allowed\n");
        return 1;
    }
}

    puts("helloworld");
    return 0;
}
@

1.2
log
@added check for arguments
@
text
@d8 1
a8 1
printf("hello world\n");
```

Abbildung 4: Wie kann man sich die Historie von Branches bei CVS vorstellen?

```
@

1.1
log
@initial revision
@
text
@d4 4
@
```

PICTURE: Siehe Skriptabbildung

1.7 Welche Dateien werden archiviert?

Im allgemeinen werden nicht wirklich *alle* Dateien eines Projekts versioniert. Class- und Objektdateien oder mittels JavaDoc generierte Dokumentationen lassen sich leicht aus den Quelltexten erzeugen und müssen (bzw. sollten) nicht im Repository hinterlegt werden. Der Grund dafür ist nicht in erster Linie Speicherplatz oder Zugriffszeit zum Ein- und Auschecken, sondern vielmehr die Gefahr, dass Informationen veralten können. Wenn im Bezug auf die Objektdateien „relevante“ Änderungen am Quelltext vorgenommen werden, die entsprechenden Dateien aber nicht neu generiert oder nicht eingecheckt werden, ist der Stand im Repository inkonsistent. Im Allgemeinen gilt: nur Sachen, die nicht generiert werden können, werden eingecheckt!

Manchmal gibt es aber Ausnahmen, wenn die Generierung teuer oder umständlich ist, sie z.B. lange dauert oder eine teure Software gebraucht wird, die deswegen nur auf einem bestimmten Computer intsalliert ist, kann es sinnvoll sein, auch die so erzeugten Dateien einzuchecken, damit alle Entwickler darauf zugreifen können.

Idealerweise wird die Generierung automatisiert, damit es keine oder nur kurze Inkonsistenzen im Repository gibt. Das Generieren kann sogar direkt beim Einchecken der betroffenen Quelldateien passieren.

1.8 Properties und Ignorieren von Dateien

Bei jeder Statusabfrage werden auch Dateien angezeigt, über die die Versionsverwaltung nichts weiß. Damit diese nicht den Blick auf das Wesentliche verdecken, kann man sie bei der Anzeige ausblenden lassen. Je nach Versionsverwaltung ist das Vorgehen dafür unterschiedlich. Für CVS wird der Name der Datei (oder ein Muster, z.B. *.class) in die Datei „.cvsignore“ eingetragen. In Subversion gibt es sogenannte *Properties*, die solche und andere Einstellungen erlauben. Die reservierten Namen beginnen mit „svn“. Die Property zum Ignorieren von Dateien heißt `svn:ignore`.

1.9 Grafische Programme für Versionsverwaltung

Bei vielen Entwicklungsumgebungen gibt es heute die Möglichkeit ein Versionsverwaltungssystem zu benutzen. Darüber hinaus existieren verschiedene Programme, die einen grafischen Zugriff auf das Repository ermöglichen. Über spezielle Erweiterungen lassen sich auch Dateimanager wie der Windows-Explorer so nachrüsten, dass versionierte Dateien und Verzeichnisse speziell markiert angezeigt werden und checkout und checkin per Klick möglich sind. Meist passiert das über den Aufruf des entsprechenden Kommandozeilenprogramms.

1.10 Eine Auswahl von Programmen

Standalone Clients:

- wincvs (Windows)
- KDESvn (KDE)
- RapidSvn (Linux, Windows)
- gitk, git-gui (Tcl/Tk-Applikationen)

Desktop integrated clients:

- KSvn (KDE)
- TortoiseCVS (Windows)
- TortoiseSVN (Windows)

Plugin clients:

- Subclipse (Eclipse)
- JGit/EGit (Eclipse)
- VisualSVN (Visual Studio, nutzt TortoiseSVN)

1.11 Vergleich Subversion - Git

Git ist ein *verteiltes* System. Jeder Benutzer hat eine lokale Kopie des ganzen Repositorys. Die meisten Operationen laufen also lokal ab und sind daher sehr schnell. Nur eine Synchronisierung des lokalen Repositorys mit einem nichtlokalen benötigt Netzwerkzugriff. Dagegen kann beim Zugriff auf ein zentrales Subversion-Repository das Netzwerk einen Flaschenhals bilden, besonders wenn viele Operationen zwischen verschiedenen Ständen ausgeführt werden. Ein Checkout zum einfachen Holen der Daten ist aber theoretisch günstiger zu haben, weil keine Historie kopiert wird. Um dieses Problem zu umgehen, kann man Git-Repositorys klein halten. Statt selbst alle Informationen und abhängigen Teile zu erhalten, gibt es Verweise auf andere Repositorys (sogenannte *submodules*).

Umgekehrt liegt in einem Subversion-Repository alles, was zu einem Projekt gehört. Damit ist auch klar, was alles für ein Backup gesichert werden muss. Durch die verteilte Natur von Git ist das dort nicht der Fall. Für eine komplette Sicherung müssten möglicherweise viele andere Repositorys heruntergeladen werden, nur um sicherzustellen, dass es auch von ihnen ein Backup gibt.

Aufgrund der verteilten Natur von Git gibt es keine besonderen Zugriffsrechte auf ein Repository. Jeder entscheidet selbst, welche Änderungen er von wem übernimmt. Eine Ausnahme ist ein möglicherweise *öffentliches* Repository, bei dem man verschiedene Benutzer einrichten kann.

Bei Subversion werden Branches im öffentlichen Repository angelegt. Jeder mit Leserechten auf das Repository kann also den Branch sehen. Da Git immer lokal arbeitet, sind auch die Branches erst einmal lokal; niemand anderes bekommt mit, wenn ein branch angelegt wird, was auch die namensvergabe des Branches deutlich vereinfachen kann. Man kann triviale Namen wie `test` oder `feature-xy-test` verwenden, was in einem öffentlichen Repository weniger leicht möglich ist.

Aktuelle Versionen von Subversion unterstützen automatisches Mergen recht gut, bei früheren Versionen müsste man zum Teil genau aufpassen, welche Versionen bereits früher gemerged wurden. Git ist hier weiter entwickelt, denn Merges werden häufig verwendet, z.B. wenn eine Version von jemand anderem ins eigene Repository eingepflegt wird. Bei Merge in Subversion erscheint außerdem immer die Person, die den Merge durchführt, als Autor der Änderungen - auch wenn jemand anderes die Änderungen im Branch gemacht und eingecheckt hat; in so einem Fall ist diese Information falsch. Abgesehen von zusätzlichen Checkoutinformationen ist auch die Speicherung bei Git effizienter. Das Mozilla-Repository braucht in Subversion etwa 12 GB, in Git fallen nur 420 MB an. Zum Vergleich: Mit CVS ist das Repository etwa 3 GB groß. Allerdings fällt die Größe pro Repository an; die Gesamtgröße bei vielen Gitreposi-

torys wird wahrscheinlich deutlich höher sein als bei einem Subversionrepository mit vielen Checkouts.

In Subversion ist es möglich, nur einen Teil des Repositorys auszuchecken, z.B. wenn man nur ein Verzeichnis oder ein Unterprojekt haben möchte. Bei Git bekommt man immer den kompletten Stand im Repository (das man sich auch erst noch besorgen muss). Es ist aber möglich, solche „Teile“ im Voraus in einzelnen Repositories zu pflegen und als externe Quellen einzubinden.

Die Versionsnummern bei Subversion sind natürliche Zahlen und daher relativ einfach. Selbst beim Mozillaprojekt sind die Revisionsnummern nur sechs Stellen lang. Git versioniert mit SHA-1-Hashes; die Revisionsnummern sind daher 40 Byte lang. Bei der Angabe einer Version reicht aber ein eindeutiger Teil der Revisionsnummer aus. Da Git SHA-1-Hashes zur Versionierung verwendet, kann man leicht prüfen, ob die Daten im Repository denen entsprechen, die eingecheckt wurden: wenn der Hashwert der neuesten Version mit dem beim Einchecken übereinstimmt, stimmt das komplette Repository überein. Das gilt auch dann, wenn man z.B. durch einen Plattencrash das eigene Repository verloren hat und sich eines aus einem Backup oder von woanders holt: gleicher Hashwert, gleicher Inhalt.

Einen Vergleich von Versionsverwaltungssystemen gibt es auf Wikipedia hier.

2 Shellskripts

2.1 Was sind Shellskripts?

Prinzipiell sind Shellskripts eine Folge von Shellbefehlen, die man auch interaktiv eingeben könnte. Im einfachsten Falle ersetzt man also eine Reihe von Befehlen durch den Aufruf des Skripts.

2.2 Warum verwendet man Shellskripts?

Vorteile von Shellskripts:

- relativ leicht zu schreiben, zu verstehen und zu ändern
- Programmierung auf hoher abstrakter Ebene
- einigermaßen portabel (abhängig von der verwendeten Shell und den aufgerufenen Programmen)

Nachteile von Shellskripts:

- weniger effizient als compilierter Code (langsamer)

- dauernd neue Prozesse
- Abhängigkeiten zu vorhandenen Programmen und Programmvarianten

2.3 Entwickeln von Shellskripts

Shell ist im Prinzip eine Programmiersprache, bei der die meisten Anweisungen aus Aufrufen von Programmen bestehen. Die zahllosen existierenden Shells unterscheiden sich u.A. durch ihre Syntax, ihre Fähigkeiten und die Namen von eingebauten Kommandos. Ähnlich der Assemblersprache wird auch hier die Programmierung umso leichter, je mehr Programme/Befehle man kennt.

Im Rahmen dieser Vorlesung sind mit *Shellskripts* Skripts für die Bourne-Again-Shell (`bash`) gemeint, die als Standardshell auf vielen Linux Distributionen installiert ist.

2.4 Ausführen von Skripts

Prinzipiell gibt es zwei Möglichkeiten, Skripts auszuführen. Zum einen kann man den entsprechenden Interpreter starten und ihm den Skriptnamen übergeben. Zum anderen kann man die Datei ausführbar machen und den Namen des Interpreters und ggf. nötige Argumente in die Shebang-Zeile schreiben, z.B.

```
#!/bin/bash  oder
#!/opt/local/bin/gawk -f  oder
#!/usr/bin/expect -- .
```

2.5 Umleitungen

Eingabe, Ausgabe und Fehlerausgabe von Programmen lassen sich auf verschiedene Weisen umleiten. Am leichtesten zu verstehen sind <, > und 2>, die die Streams auf Dateien umleiten: Eingaben werden aus der angegebenen Datei gelesen (statt von der Tastatur), Ausgaben werden in die angegebene Datei geschrieben (statt aufs Terminal). Mit der anweisung `2>&1` wird die Standardfehlerausgabe auf die Standardausgabe umgeleitet, was beim Protokollieren oder Betrachten von Ausgaben (z.B. mit `less`) hilfreich ist. Umgekehrt kann mit `1>&2` die Standardausgabe auf die Fehlerausgabe umgelenkt werden, was z.B. beim Ausgeben von Fehlermeldungen mit echo sinnvoll ist. Wenn es mehr als eine Umleitung für ein Kommando gibt ist die Reihenfolge der Umleitungen wichtig. Eine sehr mächtige Umleitung bietet | (Pipe). Die Ausgabe eines Kommandos wird zur Eingabe des nächsten. Dieses Konzept lässt sich nicht vollständig mit Umleitungen in und aus Dateien simulieren. Abgesehen vom Plattenplatz (z.B. bei Kommandos wie `dd if=/dev/hda1 | gzip -g > dasi.gz`)

ist der gleichzeitige Ablauf der Programme auch deswegen entscheidend, weil Ausgaben des ersten Programms vielleicht gar nicht mehr gebraucht werden. Ein einfaches Beispiel ist `head` als zweites Programm, ein noch besseres ist `yes` als erstes.

2.6 Hilfe!?

Das `help`-Kommando bietet Hilfe zu eingebauten Kommandos und Schlüsselworten. In vielen Fällen findet man damit schneller die gewünschten Informationen als in der Manpage, weil man nicht danach suchen muss. Allerdings sind die Hilfstexte manchmal weniger ausführlich.

2.7 Kontrollstrukturen

Mit einem Semikolon lassen sich Kommandos nacheinander ausführen, als wären sie durch einen Zeilenumbruch getrennt.

Für Verzweigungen gibt es die Schlüsselwörter `if` und `case`, für Wiederholungen `for` und `while`.

Wie in C werden Kommandos nach `&&` bzw. || nur ausgeführt, wenn der Befehl davor erfolgreich bzw. erfolglos war.

Für Kommandoabfolgen kann man Definitionen definieren.

2.8 Exitcode von Programmen

Programme teilen über ihren Exitcode mit, ob sie erfolgreich waren: 0 heißt *Erfolg*, ein anderer Wert heißt *Fehler*. Daraus ergibt sich gegenüber den meisten Sprachen eine umgekehrte Logik für Bedingungen, die man aber meist gar nicht bemerkt: Wenn der Befehl erfolgreich war, ist die Bedingung erfüllt, sonst nicht. Mit `!` kann man dieses Ergebnis logisch umdrehen.

2.9 Befehle für Bedingungen

Da jedes Programm einen Exitcode zurückgibt, kann man auch jedes als Bedingung verwenden. Besonders häufig ist hier das Kommando `test`, mit dem man Dateien auf Eigenschaften und Zeichenketten auf Gleichheit testen kann. Aus Gründen der Lesbarkeit eignet sich uach gut das Synonym `[`, das als letztes Argument `]` bekommen muss.

Ob eine Datei existiert, kann man mit `test -f file` prüfen, ob es ein Verzeichnis ist mit `[-d dir]` . Wie üblich bedeutet der Exitcode 0 die Antwort ja.

2.10 Variablen

Die Shell unterscheidet nicht zwischen „normalen“ Variablen und Umgebungsvariablen. Man kann Werte mit = zuweisen und mit vorangestelltem $ auslesen. Die Anweisung `FILE=input.txt` setzt die (Umgebungs-)Variable `FILE` auf den Wert `input.txt`. Mit `echo $FILE` kann man sich den Wert anzeigen lassen.

2.11 Parameter

Auch Parameter werden in der Shell in Umgebungsvariablen abgelegt. Sie sind nummeriert und beginnen mit 0 (dem Skriptnamen; dieses Konzept gleicht `argv` in C).

Den Wert des ersten Parameters bekommt man also mit `$1`.

2.12 Spezielle Variablen

Die Anzahl der übergebenen Argumente erhält man mit `$#`, alle Argumente (in doppelten Anführungszeichen) mit `"$@"`. Auf den Exitcodes des letzten Befehls greift man mit `$?` zu, auf die aktuell eingestellten Optionen mit `$-`. Es gibt noch weitere solcher Variablen, aber sie werden seltener benötigt und können bei Bedarf in der bash-manpage nachgeschlagen werden.

2.13 Einstellungssache

Abgesehen von der Möglichkeit, über Umgebungsvariablen Optionen einzustellen, ist das auch mit dem `set`-Kommando möglich. Für Entwicklung und Debugging sind besonders folgende Optionen hilfreich.

```
set -e
  Wenn ein Kommando fehlschlägt, wird die Shell beendet.
  Das gilt aber nicht für nicht erfüllte Bedingungen:
  if [-f file]; then mv file file.dd; fi
  wird einfach nichts tun, wenn file nicht existiert
  (oder keine Datei ist).
```

Beispiel:

```
$ set -e; for f in *, do test -f "$f" && grip "$f"; done
```

```
set -n
  Nicht existierende Umgebungsvariablen bei der Expansion
  bedeuten einen Fehler.
```

Beispiel:

```
$ set -n; unset X; echo $X
bash: X: unbound variable
```

```
set -v
  Eingegebene Zeichen werden ausgegeben.
```

Beispiel:

```
$ set -v; ls -d *
```

```
set -x
  Kommandos werden ausgegeben bevor sie ausgeführt werden.
  Dies unterscheidet sich dadurch on der Option -v, dass
  Alias, Variablen und Expansion von Dateinamen bereits
  stattegefunden haben. Man erfhärt also, welches Kommando
  wirklich ausgeführt wird.
```

Beispiel:

```
$ set -x; ls $HOME
ls --color=anto /home/hg10597
```

2.14 if

Die einfache Bedingung führt den Test aus und verzweigt entsprechend des Ergebnisses. Weil if mit fi beendet werden muss, gibt es das Folgekommando elif; bei Bedingungen im Elsezweig wären sonst mehrere fis am Ende nötig. Beispiele:

```
if [ -z "$DIR" ]; then DIR=.; fi
if [ ! -e "$DIR" ]; then
  mkdir "$DIR"
elif [ ! -d "$DIR" ]; then
  echo "'$DIR' is not a directory" >&2
  exit 1
else
  echo "'$DIR' already exists"
fi
```

2.15 for-Schleife

Mit for wird eine Anweisungsfolge nacheinander auf die angegebenen Werte angewandt. Häufig macht man soetwas für Dateien oder eine Folge von Zahlen, die oft mit Dateinamen korelliert. Beispiele.

```
for f in *.wav; do
  lane "$f" "${f%.wav}.mp3" && rm "$f"
done

for i in {20..1}; do
  let j=$i+1
  mv -i file$i.txt file$j.txt
done

for f in *.tex; do
  sed -i -e 's/Don Knuth/Donald Knuth/g' "$f"
done

for ((i=0; $i<20; i+=1)); do
done
```

2.16 while-Schleife

Die while-Schleife wiederholt den Anweisungsblock, bis die Bedingung nicht mehr erfüllt ist. Damit sind auch manche Zählweisen möglich. Beispiel.

```
while ! mount /mnt/usb/ ; do
  sleep 1
done

i=1
j=1
while [ $i -lt 100 ]; do
  if [ -f a${i}.txt ]; then
    mv -i a${i}.txt b `printf "%02i" $j`.txt
    let j=$i+1
  fi
  let i=$i+1
done
```

2.17 case-Anweisung

Mittels case wird zwischen verschiedenen Alternativen ausgewählt. Verwendungszwecke sind unter Anderem die Auswertung von Exitcodes und Parametern. Beispiel (Auszug aus bashbug):

```
while [ $# -gt 0 ]; do
  case "$1" in
    --help)    shift; do_help=y ;;
    --version) shift; do_version=y ;;
    --)        shift; break ;;
    -*)        echo "bashbug; ${1}: invalid option" >&2
               echo "$USAGE" >&2
               exit 2 ;;
    *)         break ;;
  esac
done
```

2.18 Lesen von Eingaben

Mit dem Kommando read werden Zeilen von der Standardeingabe gelesen und in Umgebungsvariablen gespeichert. Dabei werden Worte an den Wortgrenzen getrennt (Details zu Wortgrenzen unter IFS in den bash manpages) und nacheinander in die angegebenen Variablen geschrieben. Wenn es mehr Worte als Variablen gibt, landet der Rest der Zeile in der letzten Variablen. Durch Umleitung der Eingabe kann man aus einer Datei lesen. Beispiele:

```
# gibt eine Datei zeilenweise aus und fasst whitespaces zusammen
cat file | while read LINE; do echo $LINE; done

# gibt jedes Wort einer Datei in einer eigenen Zeile aus
while read LINE ; do for WORD in $LINE ; do echo $WORD ; done ; done < file

# erzeugt eine tabellarische Ansicht und leitet die Ausgabe in eine Datei um
echo "Hammer 10.00
Schraube 0.05
Bohrmaschine 100.00" | while read WARE PREIS; do printf "%-15s %6.2f\n" $WAR
---------
Hammer         10.00
Schraube        0.05
Bohrmaschine  100.00
```

2.19 Schreiben von Ausgaben

Zum Schreiben auf die Standardausgabe oder in Dateien kann man echo und printf verwenden und ggf. die Ausgabe umleiten.

3 make

3.1 Was ist make?

make ist ein Programm, das Schritte zum Erzeugen eines Projektes ausführt. Es wird meist bei Programmierprojekten eingesetzt, ist aber nicht darauf beschränkt. make bekommt in der Konfigurationsdatei, dem Makefile, gesagt, welche Dateien (Ziele, z.B. Objektdateien) von welchen (z.B. Quelltextdateien) abhängen, also aus ihnen erzeugt werden. make liest dann die Zeitstempel der Ziele und Voraussetzungen aus dem Dateisystem. Wenn ein Ziel älter ist als seine Voraussetzung, werden die entsprechenden Regeln zur Erzeugung ausgeführt.

Gegenüber einem Skript, das alle Compilieraufrufe für ein Projekt durchführt, hat make den Vorteil, dass nur die nötigen Schritte durchgeführt werden - also die, bei denen eine Voraussetzung neuer ist, als das Ziel. Das ist besonders wichtig, wenn es häufige Änderungen gibt, z.B. während einer Debuggingphase, während der man dauernd neu übersetzt und testet. Wenn das Übersetzen aller Dateien z.B. eine halbe Stunde dauert, kommt man mit der Fehlerkorrektur nicht so recht weiter. Das Übersetzen von nur ein, zwei geänderten Dateien und das Linken des Projektes sind aber vielleicht in einer halben Minute passiert.

3.2 Aufbau von Makefiles

Makefiles enthalten die Abhängigkeiten der Ziele von Voraussetzungen und die Regeln, um die Ziele zu erstellen. Der Aufbau dabei ist

```
Ziel : Voraussetzung
<Tabulator>Kommandos
```

Eine solche Konstruktion könnte z.B. sein:

```
hello : hello.c
    gcc -o hello hello.c
```

Auch mehrere Voraussetzungen sind möglich.

```
prog : a.o b.o
    gcc -o prog a.o b.o
a.o : a.c
    gcc -c a.c
b.o : b.c
    gcc -c b.c
```

Dabei werden zuerst die Voraussetzungen eines Zeils erstellt (falls nötig). Dazu wird beim Aufruf „make“ ein Abhängigkeitsbaum aufgebaut, der anschließend abgearbeitet wird.

Beispiel zu oben:

1. Bisher existieren nur Dateien, `a.c` und `b.c`. Es wird *make* aufgerufen (ohne Argumente). Da kein Ziel angegeben ist, nimmt make das erste, das im Makefile vorkommt, also prog. prog hängt von `a.o` und `b.o` ab. Beide Dateien gibt es nicht, aber es gibt Regeln, um sie zu erstellen. Ziele, die nicht existieren, sind immer „älter“ als ihre Voraussetzungen. Um `a.o` aus `a.c` zu erzeugen, wird `gcc -c a.c` ausgeführt, für `b.o` analog `gcc -c b.c`. Jetzt existieren `a.o` und `b.o`, die Voraussetzungen für prog. Der Aufruf `gcc -o prog b.o` erzeugt prog aus den Objektdateien.

2. Es wird erneut *make* aufgerufen. prog hängt ab von a.o und b.o, die ihrerseits von a.c bzw. b.c abhängen. a.c und b.c sind nicht neuer als a.o und b.o; make braucht nichts dafür zu tun; prog ist aktuell.

3. a.c wird um eine Funktion erweitert; anschließend wird *make* aufgerufen. prog hängt ab von a.o und b.o, die ihrerseits vn a.c bzw. b.c abhängen. b.c ist älter als b.o; hier gibt es nichts zu tun. Beim Speichern von a.c wurde jedoch der Zeitstempel geändert: a.c ist neuer als a.o, also wird gcc -c a.c aufgerufen, um a.o zu aktualisieren. Nun muss auch das Ziel prog neu erzeugt werden, denn seine Voraussetzungen sind nun neuer; es wird also gcc -o prog a.o b.o aufgerufen.

4. Die Kommando-Zeilen müssen mit einem Tabulator beginnen, anderenfalls wird make die Anweisung nicht verstehen (und möglicherweise eine nichtssagende Fehlermeldung ausgeben). Für jede dieser Zeilen wird eine (neue) Subshell ausgeführt, die die Anweisung abarbeitet.

3.3 Abhängigkeiten

Objektdateien hängen meist nicht nur von ihrem Quelltext ab. Meist werden auch Includedateien verwendet. Wenn sich dort etwas ändert, müssen alle Quelltexte, welche die Includedateien direkt oder indirekt verwenden, ebenfalls neu übersetzt werden. Wenn z.B. Deklaration und Definition einer Funktion geändert werden, müssen auch alle Aufrufe angepasst werden. Passiert das nicht, muss der Compiler beim Übersetzen des Aufrufs einen Fehler ausgeben. Würden die aufrufenden Quelltexte nicht neu übersetzt, wäre „merkwürdiges Programmverhalten“ die Folge.

Headerdateien in Abhängigkeiten aufzuzählen ist schon nach kurzer Zeit nicht mehr vernünftig wartbar. Wenn eine Headerdateie eine andere benutzt, müssten beide aufgezählt werden. Und wenn sich *irgendwo* etwas ändert, müsste man alle Abhängigkeiten von Headerdateien aktualisieren. Aus diesem Grund macht man das nicht selbst, sondern fragt den Compiler, welche Datei von welcher abhängt. Dazu dient die Option -MM (bzw. -M wenn Includedateien des Systems auch ausgeführt werden sollen); die entstehende Ausgabe leitet man in eine Datei um, die man im Makefile verwendet. Ein Beispiel für die entsprechenden Makeregeln:

```
depend.mak : $(wildcard *.c)
    gcc -MM *.c > depend.mak
-include depend.mak
```

3.4 Abbruch bei Fehlern

Wenn beim Erzeugen eines Ziels ein fehler passiert, bricht make ab. Die folgenden Befehle sind in der Regel nicht sinnvoll, weil sie Dateien oder Bedingungen voraussetzen, die wegen des Fehler vermutlich nicht erfüllt sind.

3.5 Pseudoziele

In vielen Fällen sind die Ziele einer Regel gar keine Dateien, sondern werden dazu verwendet, um bestimmte Aktionen auszuführen, wenn beim Aufruf das entsprechende Ziel angegeben ist. Ein bekanntes Ziel dieser Art ist `clean`. Wenn `make clean` aufgerufen wird, wurden z.B. alle Objektdateien gelöscht. Die Regel dafür könnte so aussehen:

```
clean :
    rm -f *.o
```

Häufig verwendet wird auch das Ziel dist-clean, dads außerdem die Abhängigkeiten und das „Endziel“ löscht, um für die Veröffentlichung des Archivs aufzuräumen.

```
dist-clean : clean
    rm -f prog depend.mak
```

Da `make` ohne Angabe von Zielen das erste Ziel im Makefile verwendet, wird dort häufig *all* definiert, das alle „sinnvollen“ Ziele als Voraussetzung aber keine weiteren Ziele hat:

```
all: prog
```

Weil eine vorhandene Datei mit dem Namen eines Pseudoziels das Ausführen der entsprechenden Regeln verbieten würde, gibt es den Konfigurationseintrag `PHONY` (künstlich, unecht), mit dem die Regeln für Pseudoziele auch in solchen Fällen ausgeführt werden. `make clean` würde also auch dann aufräumen wenn eine Datei mit dem Namen `clean` existiert. Beispiel für die Verwendung:

```
PHONY: all tests clean dist-clean
```

3.6 Vordefinierte Regeln

In GNU make gibt es bereits etliche eingebaute Regeln, um Ziele aus Voraussetzungen zu erstellen. Wenn die Datei `hello.c` existiert, wird make mit der Anweisung `make hello` folgende Regel ausführen:

```
%: %.c
#command to execute (built-in):
    $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

`LINK.c` hat dabei den Wert `$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)`. Es wird also der C-Compiler mit entsprechenden Flags aufgerufen. Das Prozentzeichen hat ungefähr die Bedeutung des Sterns in der Shell; allerdings wird auf beiden Seiten des Doppelpunkts die gleiche Zeichenkette dafür eingesetzt. `hello` hängt also ab von `hello.c`, `foo` von `foo.c`. Auch für Objektdateien gibt es entsprechende Regeln:

```
%.o: %.c
# commands zu execute (built-in)
    $(COMPILE.c) $(OUTPUT_OPTION) $C
```

Nach dem gleichen Prinzip existieren Regeln für Ausgaben des Präprozessors und für C++, Assembler-, Pascal- und Fortrandateien.

Man kann sich mit `make -np` alle Regeln (inklusive derer des entsprechenden Makefiles) anzeigen lassen.

3.7 Variablen

Make unterstützt Variablen, deren Wert wie in der Shell mit $ ermittelt wird. Wenn der Variablenname länger als ein Zeichen ist, muss er in runde Klammern geschrieben werden, z.B.: `$(CC)`, `$(CX XFLAGS)` oder `$(SHELL)`. Wenn ein Dollarzeichen an die Shell weitergegeben werden soll, muss es verdoppelt werden, anderenfalls würde make die Variablen expandieren (was meist die leere Zeichenkette ergäbe). Beispiel `printshell`:

```
@echo "the shell's SHELL variable is $$SHELL, make's SHELL variable is '$(SH
```

Das @ am Zeichenanfang bedeutet, dass das Kommando nicht von Make ausgegeben werden soll, bevor es ausgeführt wird.

Variablen können auf verschiedene Arten gesetzt werden. Eine davon sieht aus wie in der Shell:

```
|
    CFLAGS=-Wall -pedantic-g|.
```

Bei den dabei entstehenden rekursiv expandierten Variablen werden evtl. bei der Definition benannte Variablen erst dann expandiert, wenn sie benötigt werden, nicht schon bei der Zuweisung. Alternativ ist eine Zuweisung wie in Pascal möglich.

```
CFLAGS := -Wall -pedantic -g
```

Bei diesen Variablen wird die rechte Seite bei der Zuweisung expandiert. Das ist ein Unterschied, wenn dabei Variablen benutzt werden, die erst später definiert werden oder später einen anderen Wert bekommen, z.B. wenn der Wert die Ausgabe des Programms `date` ist.

```
DATE=‘date‘ # aktuelles Datum
DATE := ‘date‘ # immer gleich: Datum bei der Zuweisung
```

Mit dem Zuweisungsoperator += können Werte an Variablen angehängt werden. Bei ?= erfolgt die Zuweisung nur, wenn die Variable noch nicht existiert.

3.8 Variablen für einzelne Regeln

Gerade bei der Ersetzung mit Prozentzeichen braucht man eine Möglichkeit, sich auf die expandierten Werte zu beziehen. Hierfür gibt es u.A. die folgenden Variablen, die je nach Ziel und Voraussetzungen einen entsprechenden Wert annehmen:

- `$@`: das Ziel
- `$<`: die erste Voraussetzung
- `$^`: alle Voraussetzungen, durch Leerzeichen getrennt
- `$?`: alle Voraussetzungen, die neuer sind als das Ziel, durch Leerzeichen getrennt

Beispiel für die Benutzung:

```
$(BIN): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
%.o: %.c
    $(CC) $(CFLAGS) -o $@ -c $<
```

3.9 Variablen für Programme

Ähnlich wie bei den Zielen gibt es auch für Kommandos vordefinierte Variablen. Der Vorteil ihrer Benutzung ist, dass man den Defaultwert überschreiben kann, um z.B. eine Konfiguration für ein bestimmtes System anzupassen. Einige Beispiele für Variablen und ihren Wert:

- AR = ar
- CC = cc
- CPP = $(CC) -E
- CXX = g++

- LINT = lint
- RM = rm -f

Nach diesem Prinzip gibt es auch Flags, die den Programmen mitgegeben werden, und die man ebenso überschreiben kann. Die Flags passend zu den angegebenen Programmen sind:

- ARFLAGS = rv
- CFLAGS = <leer>
- CPPFLAGS =
- CXXFLAGS =
- LDFLAGS =
- LINTFLAGS =

3.10 Probleme mit make

Da make mit Zeitstempeln arbeitet, kann es Probleme geben, wenn sich an diesen etwas Unvorhergesehenes ändert. So etwas kann passieren, wenn Dateien über mehrere Zeitzonen kopiert werden. Eine Voraussetzung kann neuer sein, als ihr Ziel, aber aufgrund einer Zeitverschiebung ist ihr Zeitstempel „älter“. Manchmal ist die Auflösung von Zeitstempeln ein Problem, weil sie *Sekunden-basiert* ist.

3.11 Neuerzeugung eines Projekts

Um ein Projekt neu zu erstellen, kann man mit `clean` oder `dist-clean` aufräumen und dann make erneut starten. Wenn man nur ein, zwei Dateien übersetzen will, kann man mit touch ihren Zeitstempel aktualisieren. Ein anschließender make-Aufruf wird die entsprechenden Ziele dann neu erstellen.

3.12 Alternativen zu make

- Cons
- ant
- rake

4 Reguläre Ausdrücke

4.1 Hier fehlt noch was ... Björn fragen

...

4.2 27.11.2014

Manche Dialekte erlauben es, auf einen bereits gefundenen Teilstring zuzugreifen, um ihn nochmal im Text zu suchen. Dort kann man mit `(a+)b+\1` sogar das Muster $a^n b^m a^n$ suchen, was über die eigentlichen regulären Ausdrücke hinausgeht. In manchen Dialekten ist der Ausdruck auch gleichbedeutend mit `a+b+a+`, weil dort eine Referenz im Suchstring schon vor dem Suchen im Text ersetzt wird. Welchen Dialekt man verwendet, muss man nachlesen oder ausprobieren.

4.3 Worauf passt ein regulärer Ausdruck?

Das erste passende Textstück wird verwendet; wenn der Ausdruck weiter hinten im Text nochmal passt, spielt das keine Rolle mehr. Für Programme wie `grep` ist das egal: Wenn das Muster in der Zeile vorkommt, wird sie ausgegeben, egal wo es vorkommt. Beim Ersetzen von Text ist es aber ein Unterschied, wenn das Muster mehrfach vorkommt und genau das erste Vorkommen ersetzt wird.

Die normalen Quantifizierer sind „gierig“ (engl. *greedy*), es wird also immer so viel wie möglich von der Zeichenkette verwendet. Der Ausdruck `\d+` wird bei der Zeichenkette `123` auf alle Ziffern passen. `\d.*` wird alle Zeichen ab der ersten Ziffer erkennen. Diese Gier führt dazu, dass Ausdrücke wie `".*"` nicht geeignet sind, um Text in Anführungszeichen zu erkennen. Es kann mehr als einmal Text in Anführungszeichen in einer Zeile geben, und der Ausdruck passt auf alles vom ersten bis zum letzten Anführungszeichen. Für die üblichen Zeichenketten im C-Stil kann man den Ausdruck `"\\.|[^\\"])*"` verwenden. Eine Ausnahme, bei der der Ausdruck nicht passt, wird in der Übung behandelt. Text zu erkennen hat aber höhere Priorität als Gier; Teilausdrücke verzichten auf Text, den sie erkennen können, wenn das zu einer Übereinstimmung führt. Ein Beispiel für soetwas ist der Ausdruck `(.*)a`: Wenn er passt, wird `\1` alle Zeichen bis vor das letzte `a` der Zeile enthalten. Welcher Text wird von dem geklammerten Ausdruck erkannt und in `\1` gespeichert, wenn `^.*([0-9]+)` auf den Text `Copyright 2003` angewandt wird? Die Antwort lautet 3. `^.*` nimmt alles ab dem Zeilenanfang und gibt so lange Zeichen zurück, bis auch der Rest des Ausdrucks passt.

4.4 Kontextsensitive Muster

Manche Dialekte unterstützen Lookaround, mit dem ein Muster nur passt, wenn auch passt, was davor oder danach steht. Der Kontext wird zwar erkannt, ist aber nicht Teil des passenden Texts. Das Prinzip kann man recht gut mit den leichteren Vim-Mustern `\zs` für start-of-match und `\ze` für end-of-match verdeutlichen; dazwischen steht der eigentliche Suchstring, der nur erkannt wird, wenn auch der Kontext passt.

Will man `La` vor `TeX` finden, so ist `La\zeTeX` ein regulärer Ausdruck in Vim-Notation dafür. Für das `T` in `LaTeX` ist `La\zsT\zeeX` geeignet. Da hier keine Wortgrenzen angegeben sind, findet man das Muster aber auch in `abcLaTeXdef`.

4.5 Verwendung regulärer Ausdrücke

Reguläre Ausdrücke werden alltäglich für Textersetzungen mit einem Editor benutzt. Sie werden auch beim Parsen verwendet, u.A. bei Compilern, für Syntaxhighlighting und zum Prüfen von Eingaben auf formale Korrektheit, z.B. auf die Gültigkeit von Emailadressen oder URIs.

Manche Programmiersprachen haben eingebaute Unterstützung für reguläre Ausdrücke. Zu ihnen gehören awk, sed und Perl. Für viele andere Sprachen gibt es Bibliotheken, mit denen man aus regulären Ausdrücken Objekte erzeugen kann, die mit verglichen werden und Ersetzungen darin durchführen können. Beispiele für solche Sprachen sind C/C++, C#, Java und PHP.

5 Einige UNIX-Programme im Überblick

5.1 grep

`grep` dient zum Suchen von Text in Dateien. Es verwendet einen regulären Ausdruck als Suchmuster und gibt gewöhnlich die Zeilen aus, in denen das Muster passt. Das ist bei verschiedenen Anwendungsfällen nützlich. Zwar ist das Suchen innerhalb einer Datei im Editor meist einfacher; manchmal weiß man aber gar nicht, in welcher Datei die gesuchte Stelle vorkommt, oder man will die entsprechenden Zeilen mit einem anderen Programm per UNIX-Pipe weiterverarbeiten. Dann kann man mit grep zum Beispiel in allen Quelltextdateien suchen und ggf. die Dateinmane (mit)ausgeben lassen.

Der Name *grep* kommt vom vi-Kommando g (global) mmit der Anweisung p (print), zusammen `:g/re/p`. Dabei steht `re` für den regulären Ausdruck (regular expression).

Der allgemeine Aufruf lautet:

```
grep [OPTIONS]... PATTERN [FILE]...
```

grep unterstützt etliche Optionen. Die folgenden werden häufig verwendet: `-e` Als nächstes folgt das Suchmuster. Diese Option ist besonders dann hilfreich, wenn das Muste rmit einem Stricht anfängt, damit grep es nicht als Option interpretiert.

`-n` Die Nummern der ausgegebenen Zeile (und ein DOppelpnkt) wird den Zeilen vorangestellt.

`-v` Es werden die Zeilen ausgegeben, in denen das Muster *nicht* vorkommt.

`-i` Groß-/Kleinschreibung wird ignoriert.

`-H-h` Dateinamen werden bzw. werden nicht ausgegeben. Bei mehreren Dateien als Argumente wird der Dateiname normalerweise dem Zeileninhalt (und der optionalen Zeilennummer) vorangestellt; wenn keine order nur eine Datei als Argument angegeben ist, wird kein Name ausgegeben. Dieses Verhalten lässt sich ändern, um Zeilen ohne Dateinamen zu erhalten, oder um bei der Benutzung mit xargs bei nur einer Datei auch den Namen zu erhalten.

`-l` Für jede durchsuchte Datei, in der das Muster vorkommt, wird nur der Dateiname ausgegeben und die Suche in dieser Datei beendet (weil bereits feststeht, dass das Muster vorkommt)

`-q` Es wird nichts ausgegeben, sondern nur der Exitcode verwendet: 0 heißt dabei „Muster gefunden“, 1 heißt „Muster nicht gefunden“ und 2 steht für einen Fehler.

`-c` Statt der gefundenen Zeilen wird angegeben, wie viele Zeilen der einzelnen Dateien das Muster enthalten.

`-B-A-C` Oft will man nicht nur die gefundenen Zeilen, sondern auch Zeilen davor oder danach sehen (wie bei `svn diff`). Das lässt sich mit den Optionen `-B` (before), `-A` (after) bzw. `-C` (context, davor und danach) einstellen. Wenn Zeilen zwischen den Ausgaben „fehlen“, wird – als Trennzeichen eingesetzt.

`--color` Dieser Option kann mit Gleichheitszeichen folgen, wann Farben verwendet werden sollen: never, always und anto. Im letzten Fall entscheidet der Typ des Ausgabedeskriptors ob Farben verwendet werden sollen. Wenn die ausgabe auf ein Terminal kommt, werden Farben verwendet, bei Umeltiungen in eine Datei oder eine Pipe nicht. Oft ist grep ein Alias für `'grep --color=auto'`.

Beim Aufruf von grep aus der Shell ist es ratsam, den regulären Ausdruck in einfache Anführungszeichen einzuschließen, damit die Shell Zeichen wie * und | nicht selbst interpretiert.

5.2 cmp - Compare

`cmp` vergleicht Dateien byteweise und gibt Offset und Zeilennummer des ersten Unterschieds aus, wenn es einen gibt. Der allgemeine Aufruf lautet:

```
cmp [OPTION]... FILE1 [FILE2 [SKIP1 [SKIP2]]]
```

Nützliche Optionen sind:

`-b`

Die Bytewerte des ersten Unterschieds werden mit ausgegeben.

`-i`

Die folgende Zahl gibt an, wie viele Bytes der ersten Datei beim Vergleich übersprungen werden sollen; eine mit Doppelpunkt abgetrennte Zahl gibt das Offset zu der zweiten Datei an (wie bei den SKIP-Argumenten). Ein Suffix gibt entsprechende Faktoren an, z.B. kB für 1,000, k für 1024, mB für 1,000,000, M für 1,048,576 usw. für G, T, P, E, Z, Y.

`-l`

Es wird nicht nur der Bytewert des ersten, sondern von jedem Unterschied angezeigt. Bei verschieden großen Dateien wird nur bis zum Ende der kleineren Datei verglichen,

`-n`

Die der Option folgende Zahl gibt an, wie viele Bytes verglichen werden sollen.

`-s`

Es gibt keine Ausgabe; der Exitcode zeigt die Gleichheit an. 0 heißt gleich, 1 verschieden, ≥ 2 Fehler. Beispielausgabe / erzeugt durch Vergleich von zwei Versionen der Datei `main.c`, die erste von der Standardeingabe; Aufruf `cmp - main.c`):
- main.c differ: char 1504, line 59

5.3 diff - Vergleich von Dateien

`diff` vergleicht zwei Dateien zeilenweise und gibt die unterschiedlichen Zeilen aus. Mittels der Präfixe < und > kann man zwischen Zeilen aus der „linken“ und „rechten“ Datei unterscheiden. Der allgemeine Aufruf lautet:

```
diff [OPTION]... FILES
```

Häufig verwendete Optionen sind:

- -i Groß-/Kleinschreibung wird beim Vergleich ignoriert
- -w Whitespaces werden beim Vergleich ignoriert
- -C Bei Unterschieden wird die angegebene Anzahl von Kontextzeilen mit angegeben
- -u Bei Unterschieden wird die angegebene Anzahl von Kontextzeilen im *Unified Format* mitausgegeben
- -p diff gibt die C-Funktion an, in der sich ein Unterschied befindet.
- -e Erzeugt ein `ed`-Skript, mit dem `ed` die linke in die rechte Datei umwandeln kann
- -y Gibt die Dateien in zwei Spalten aus, sodass man leichter erkennen kann, wie die beiden Dateien aussehen.
- -r vergleicht zwei Verzeichnisse rekursiv (also jeweils gleichnamige Dateien und Verzeichnisse)

Die Ausgabe des `diff`-Kommandos einer Versionsverwaltung wird möglicherweise durch das Programm `diff` erstellt. Beispielsausgabe bei zwei ähnlichen Quelltextdateien (erzeugt durch Vergleich von zwei Versionen der Datei `main.c`, die erste von der Standardeingabe; Aufruf `diff - main.c`).

```
58a59
>  boolean optionNostackcheck
72a74
>  optionNostackcheck = FALSE;
100a103, 105
>  if(strcmp(argv[i], "--nostackcheck") == 0) {
>    optionNostackcheck = TRUE;
>  }
>  else
152c157
<  genCode(progTree, globalTable, outFile, optionNospilling, optionNocodecomm
---
<  genCode(progTree globalTable, outFile, optionNospilling, optionNocodecomme
```

5.4 patch

Mit `patch` kann man die per `diff` ermittelten Unterschiede zwischen zwei Dateien verwenden, um aus der ersten Datei die zweite zu machen. Der übliche Anwendungsfall ist folgender: Man hat eine Datei (z.B. Quelltext) und nimmt Änderungen daran vor. Durch ein `diff` auf die alte und die neue Fassung (in dieser Reihenfolge) ermittelt man die Unterschiede zwischen den Versionen. Diese Unterschiede kann man jemandem zur Verfügung stellen, der ebenfalls die alte Version der Datei besitzt. Mit `patch` kann er daraus die neue Version machen. Der allgemeine Aufruf lautet:

```
patch [OPTION]... [ORIGFILE [PATCHFILE]]
```

- -c Das Patchfile enthält Kontextinformationen
- -e Das Patchfile ist ein `ed`-Skript
- -n Das Patchfile liegt im Unified-Format vor
- -R Beim Erstellen des Patchfiles waren alte und neue Dateien vertauscht; `patch` wird die Änderungen im Patchfile also „rückwärts“ anwenden, um aus der alten Dateie die neue zu erstellen
- -o Die Ausgabe wird in die angegebene Datei geschrieben (statt aus der alten Datei die neue zu machen)
- -b Es werden Backupdateien erzeugt, bevor die Dateien verändert werden. Es gibt auch Optionen, um Backups nur zu erzeugen, wenn die Projektdateien unplausibel sind
- -f Auch unplausible Änderungen werden gemacht; es gibt keine Rückfragen
- -s keine Ausgabe, wenn kein Fehler passiert

Der Exitcode 0 bedeutet, alle Änderungen wurden erfolgreich angewandt. Bei 1 gibt es Teile, die nicht verwendet werden konnten; 2 deutet auf größere Probleme hin. Wenn man Patchfile er- und bereitstellen möchte, sollte man die Empfehlungen aus der manpage beachten.

5.5 diff3

Mit `diff3` kann man drei Dateien zeilenweise vergleichen. Auch hier wird eine Ausgabe erzeugt, die der von `patch` akzeptierten Eingabe ähnelt. Der übliche Fall ist, dass die gleiche Datei von zwei Leuten geändert wird. `diff3` findet die jeweiligen Unterschiede heraus, sodass man durch Vornehmen aller Änderungen aus der alten Datei eine neue machen kann, die all die Änderungen enthält (so lange es keine Konflikte gibt). Der allgemeine Aufruf lautet

```
diff3 [OPTIONS]... MYFILE OLDFILE YOURFILE
```

Der Exitcode 0 bedeutet Erfolg, 1 Konflikt, 2 Probleme.

5.6 sort

`sort` sortiert Dateien zeilenweise und schreibt das Ergebnis auf die Standardausgabe. Die übliche Sortierung erfolgt nach dem Bytewert der Zeichen; das ist aber abhängig von der Spracheinstellung. Mit `LC_ALL=C` bekommt man dieses „normale“ Verhalten. Der allgemeine Aufruf lautet:

```
sort [OPTION]... [FILE]...
```

Häufig verwendete Optionen sind:

- -b Führende Leerzeichen werden ignoriert
- -f Groß-/Kleinschreibung wird beim Vergleich ignoriert
- -i Nichtdruckbare Zeichen werden ignoriert
- -M Es wird nach Monatsnamen sortiert (Schreibweise abhängig von `LC_TIME`)
- -n Es wird nach Zahlenwerten sortiert
- -r Die Sortierung erfolgt absteigend
- -c Es wird nur geprüft, ob die Dateien sortiert sind
- -C Es wird nur geprüft, ob die Dateien sortiert sind, aber es wird nichts ausgegeben
- -k Die angegebene Zahl gibt die Position des ersten Sortierschlüssels an; eine mit Komma abgetrennte optionale zweite Zahl gibt die letzte Spalte des Schlüssels
- -m Die Eingabedateien werden Zeilenweise einsortiert; es findet keine neue Sortierung statt. Die Dateien sollten bereits sortiert sein.
- -s Es wird ein stabiler Sortieralgorithmus verwendet.
- -u Doppelte Zeilen werden entfernt. Mit -c: auf strenge Sortierung prüfen.

Der Exitcode 0 bedeutet Erfolg, 1 bedeutet bei -C, dass die Eingabe nicht sortiert war, 2 Fehler.

5.7 uniq

`uniq` findet doppelte Zeilen in der Eingabe und gibt sie aus oder entfernt sie (wobei *doppelt* hier immer bedeutet, dass die gleiche Zeile mehr als eimal direkt hintereinander kommt; es wird nie sortiert). Der allgemeine Aufruf lautet:

```
uniq [OPTION]... [INPUT [OUTPUT]]
```

Häufig verwendete Optionen sind

-c Gibt vor jeder Zeile aus, wie häufig sie vorkommt

-d Es werden nur Zeilen ausgegeben, die mehr als einmal vorkommen

-f Die angegebene Anzahl Felder wird beim Vergleich übersprungen

-i Groß-/Kleinschreibung wird beim Vergleich ignoriert

-s Die angegebene Anzahl Zeichen wird beim Vergleich übersprungen

-u Es werden nur Zeilen ausgegeben, die nur einmal vorkommen

Der Exitcode 0 heißt Erfolg, alles andere bedeutet Fehler.

5.8 xargs

`xargs` liest Zeichenketten von der Standardeingabe und gibt sie als Argumente an das angegebene Programm weiter. Als Trennzeichen zwischen den Zeichenketten dienen Leerzeichen und Zeilenumbrüche (wenn sie nicht durch Anführungszeichen oder Backslash maskiert sind). In vielen Fällen besteht die Eingabe aus Namen von Dateien, die von einem weiteren Programm verarbeitet werden sollen. Der allgemeine Aufruf lautet:

```
xargs [OPTION]... [COMMAND [INITIAL-ARGUMENTS]]
```

Nützliche Optionen sind:

-a Die Eingabe wird aus der angegebenen Datei gelesen und die Standardeingabe für das gestartete Programm wird nicht nach `/dev/null` umgeleitet.

-0 Die Eingabestrings werden von binären Nullen statt von Whitespaces getrennt; alle Zeichen stehen für sich selbst

-n Es werden höchstens die angegebene Anzahl von Argumenten pro Programmaufruf übergeben.

-p xargs zeigt jeden Programmaufruf und fragt, ob es ihn durchführen soll

-r Wenn es keine Eingabe gibt, dann wird das Programm nicht aufgerufen

-s Es wird höchstens die angegebene Anzahl Zeichen pro Aufruf verwendet (inklusive Programmname, übergebene Argumente und trennende binäre Nullen

-p Es wird maximal die angegebene Anzahl von Programmen gleichzeitig gestartet; Default ist 1, 0 bedeutet *beliebig viele*

Ein anderer Exitcode als 0 deutet auf Probleme him; weitere Informationen gibt es in der manpage.

Beispiele für die Verwendung:

```
grep -l MUSTER1 * | xargs -r grep -l MUSTER2
```

Findet Dateien im aktuellen Verzeichnis, die sowohl MUSTER1 als auch MUSTER2 enthalten.

```
cut -d: -f1,2 < /etc/passwd | sort | xargs echo
```

Gibt die Benutzer des Systems sortiert in einer Zeile aus.

5.9 find

`find` sucht nach Dateien mit bestimmten Eigenschaften und führt Operationen darauf aus. Normalerweise wird der Dateiname ausgegeben. Die Suchkriterien sind vielseitig und kombinierbar, und die Möglichkeit, Programme mit den gefundenen Dateien aufzurufen oder die Liste der Dateien weiterzuleiten machen `find` sehr mächtig.

Die Anweisungen für `find` beginnen mit einem einzelnen Strich, auch wenn sie aus mehreren Buchstaben bestehen. Der allgemeine Aufruf lautet (vereinfacht):

```
find [-L] [path...] [expression]
```

Normalerweise wird symbolischen Links nicht gefolgt, das lässt sich aber mit der Option -L ändern. Die Ausdrücke sind Kombinationen von Operatoren, Optionen, Tests und Aktionen. Operatoren sind runde Klammern für Prioritäten, ! für die negation, Verkettung und -a für UND-Verknüpfung, -o für ODER-Verknüpfung und , für bedingungslose sequentielle Ausführung.

Beispiele für häufig benutzte Befehle sind:

`-name, -iname`	Der Name der Datei entspricht dem angegebenen Shellmuste
`-regex, -iregex`	Der komplette relative Pfad entspricht dem angegebenen reg Ausdruck.
`-print, print0, -printf, -fprint, -fprint0, -fprintf, -ls, -fls`	Der Dateiname oder andere Informationen zur Datei und e lenumbruch bzw. eine binäre Null werden auf die Standarda bzw. in die angegebene Datei geschrieben.
`-user, -group`	Die Datei gehört dem angegebenen Benutzer bzw. der angeg Gruppe.
`-innm, -samefile`	Die Datei hat die angegebene Inodenummer bzw. ist dieselb wie die angegebene (mit -L auch bei symbolischen Links)
`-links`	Die Datei hat die angegebene Anzahl von Hardlinks.
`-mtime, -mmin, -atime, -ltime, -cmin`	Die Datei bzw. ihr Inode wurde vor n Tagen bzw. Minuten bzw. geändert.
`-daystart`	Die Prüfung auf eine Zeit bezieht sich auf das Datum, nicht a Stunden
`-newer, -anewer, -cnewer`	Die Datei ist später als die angegebene Datei geändert, gelese ihr Status geändert worden. Es gibt keine older-optionen, abe tierung mit `!` ist möglich.
`-perm`	Die Zugriffsrechte entsprechen den angegebenen. Symbolisch stellung ist möglich. Rechte mit - sind „mindestens diese“, mit / „irgendeins von diesen“.
`readable, -writable, -executable`	Die Datei ist hier für den suchenden Benutzer les- schreib- bz führbar. Dieser Test berücksichtigt, Access-Control-Lists, kar Probleme beim UID-Mapping von NFS-Servern haben.
`-size`	Die Datei ist *n* Einheiten groß. Die Einheiten *b* (512-Byte- default), *c* (Bytes), *w* (Zwei-Byte-Worte), *k*, *M* und *G* (Kilo-, und Gigabyte) sind erlaubt.
`-type, -xtype`	Die Datei hat den angegebenen Typ; erlaubt sind `b` (block spe (character special), `d` (Verzeichnis), `p` (named pipe, FIFO), `f` (n Datei), `l` (symbolischer link) und `s` (socket)
`-maxdepth`	`find` wird nicht über die angegebene Suchtiefe hinaus in Ve nisse absteigen. 0 bedeutet, es werden nur die Argumente gep
`-exec, -execdir, -ok, -okdir`	Für jede gefundene Datei wird das angegebene Programm g (das möglicherweise selbst eine Prüfung durchführt). Dabei st den Dateinamen, den man dem Programm meist übergebe das Kommando wird mit einem Semikolon abgeschlossen, w für die Shell maskiert werden muss. Bei den dir-Varianten w Programm in dem Verzeichnis ausgeführt, in dem die Datei den wurde. Bei den ok-Varianten wird vor jedem Aufruf gefrag durchgeführt werden soll.
`-depth`	Der Inhalt von Verzeichnissen wird vor dem Verzeichniseint handelt.
`-prune`	Verzeichnisse werden nicht rekursiv behandelt.

Ein anderer Exitcode als 0 weist auf Fehler hin.
Beispiele für die Verwendung:

```
find . -iname "*.mp3" -print
```

Listet Dateien mit der Erweiterung mp3 unterhalb des aktuellen Verzeichnisses aus (d.h. im aktuellen Verzeichnis und allen Verzeichnissen, die sich darin befinden usw.)

```
find /tmp -name core -type f -print | xargs /bin/rm -f
```

Löscht coredumps im Tempverzeichnis.

```
find /tmp -name core -type f -print0 | xargs -0 /bin/rm -f
```

Löscht auch coredumps im Tempverzeichnis. Funktioniert besser, wenn Leerzeichen oder Anführungszeichen im Dateinamen sind, als die Variante von oben.

```
find . -type f -exec file '{}' \;
```

Ruft das Programm `file` für jede normale Datei unterhalb des aktuellen Verzeichnisses auf.

```
find / \( -perm -4000 -fprintf /root/suid.txt "%#m %n %p\n" \) -o \( -size +1
```

Schreibt Informationen über Dateien mit gesetztem suid-Bit bzw. größer als 100 MB in die entsprechenden Dateien. Die Informationen sind oktale Zugriffsrechte, Eigentümer und relativer Dateiname bzw. Dateigröße und relativer Dateiname.

```
find \$HOME -mtime 0
```

Sucht nach Dateien im Homeverzeichnis, die in den letzten 24 Stunden geändert wurden.

```
find /sbin /usr/sbin -executable \! -readable -print
```

Listet Dateien auf die ausführbar aber nicht lesbar sind.

```
find . -perm 664
```

Listet Dateien mit der (oktalen) Zugriffsmaske 664.

```
find . -perm -664
```

Sucht Dateien mit mindestens den Zugriffsrechten 664.

```
find . -perm /222
```

Sucht Dateien, die für irgendjemanden schreibbar sind.

```
cd /source-dir; find . -name .snapshot -prune -o \( \! -name "*~" -print0 \)
```

Kopiert den Inhalt von /source-dir nach /dest-dir, lässt dabei aber Dateien und Verzeichnisse, die .snapshot heißen und Dateien (und Verzeichnisse, aber nicht deren Inhalt), die auf Tilde enden, aus. Die Klammern dienen nur der Klarheit, sind aber nicht nötig.

```
find repo/ -exec test -d {}/.svn -o -d {}/.git -o -d {}/cvs \; -print -prune
```

Listet Verzeichnisse unterhalb von repoauf, die Versionsverwaltungsinformationen enthalten (also Verzeichnisse wie .svn, .git und cvs); die gefundenen Verzeichnisse werden nicht weiter untersucht (-prune).

```
find /foo -maxdepth 1 -atime +366 -print0 | xargs -v0 sh -c mv "$0"
```

Findet Einträge im Verzeichnis /foo, die länger als ein Jahr nicht gelesen wurden und verschiebt sie ins Verzeichnis /archive. „move“ wird zu Argument 0 des Aufrufs; ohne es würde die erste Datei nicht verschoben.

```
find /usr/include -name '*.h' | xargs grep -wl mode_t | xargs -r sh -c 'exec
```

Findet Headerdateien, die den Bezeichner `mode_t` enthalten, und öffnet sie mit Emacs. Ohne die Eingabeumleitung würde die Eingabe aus /dev/null gelesen.

5.10 awk

awk ist eine Programmiersprache, deren Stärken Texterkennung und Spaltenweise Verarbeitung sind. Inzwischen wurde awk weitgehend durch Perl verdängt, für viele Anwendungen eignet es sich aber noch sehr gut. Der Name awk besteht aus den Anfangsbuchstaben der Entwickler: Aho, Weinberger, Kernighan.

Kurzbeispiele zur Verwendung:

```
ls -l | awk '{sum += $5} END {print sum}'
```

Gibt die Gesamtgröße aller (nicht versteckten) Dateien im aktuellen Verzeichnis aus.

```
awk 'BEGIN {FS=";"} {print $1 | "sort"}' < /etc/passwd
```

Gibt eine sortierte Liste der Loginnamen aller Benutzer aus.

```
 awk '{nlines++} END {print nlines}' file
```

Zählt die Zeilen einer Datei und gibt ihre Anzahl aus.

```
awk '/^a/ {nlines++} END {print nlines}'
```

Zählt die Zeilen der Eingabe, die mit a anfangen.

```
awk '/^a/ {alines++} /^b/ {blines++} END {print alines " " blines }'
```

Zählt die Zeilen der Eingabe, die mit a und die mit b anfangen.

```
awk '{print FNR, $0}' file
```

Gibt vor jeder Zeile der Datei file ihre Zeilennummer aus.

```
tail -f access_log | awk '/myhome.html/ {system(" nmap " $1 ">>/logdir/myhome
```

5.11 sed

sed steht für Streameditor. Es handelt sich um ein nicht-interaktives Programm zur Bearbeitung von Datenströmen. Was mit der Eingabe passieren soll, ist im angegebenen Skript bzw. den Argumenten festgelegt. Die Anweisungen ähneln denen des Editors ed; vi-Benutzer werden sie erkennen. Für das Schreiben von Skripts und Details von Optionen lese man in der manpage oder z.B. unter http://www.grymoire.com/Unix/Sed.html (Stand Dezember 2014).

Kurzbeispiele für die Verwendung von sed:

`sed -f sedskript`
Liest von der Standardeingabe, führt das angegebene Skript `sedskript` mit den Daten aus und schreibt auf die Standardausgabe.

`sed -e 's/abc/def/' input.txt`
Liest die Datei `input.txt`, ersetzt die erste Zeichenfolge `abc` pro Zeile durch `def` und schreibt auf die Standardausgabe.

`sed -n '/MUSTER/p' file`
Sucht nach `MUSTER` und gibt Zeilen aus, in denen es vorkommt. Dieses Skript ist effektiv ein `grep`: `p` gibt Zeichen aus, auf die das Muster passt. Die Option `-n` unterdrückt das automatische Ausgeben von Zeilen.

`sed '3,17 s/[0-9][0-9]*//' < file > new`
Entfernt die erste Ziffernfolge in den Zeilen 3 bis 17. Ein- und Ausgabeumleitung wird von der Shell behandelt.

`sed -i '/^ *$/d' inputfilename`
Entfernt Zeilen, in denen nur Leerzeichen vorkommen. Die Option `-i` wird die Ausgabe in die Datei zurückschreiben (in-place). Im Gegensatz zu einem ed-Skript mit der gleichen Funktion zerbrechen dabei eventuelle Hardlinks der Datei.

5.12 wc

wc zählt Zeichen, Worte und Zeilen der Eingabe. Der allgemeine Aufruf lautet:

```
wc [OPTIONEN]... [FILE]...
```

Die häufigsten Optionen sind:

- `-c` Zählt die Anzahl der Bytes.
- `-m` Zählt die Anzahl der Zeichen
- `-l` Zählt die Anzahl der Zeilenumbrüche, also der Zeilen
- `-w` Zählt die Anzahl der Wörter.
- `-L` Gibt die Länge der längsten Zeile aus.

5.13 head

`head` gibt die ersten Zeilen der Eingabe aus. Der allgemeine Aufruf lautet:

```
head [OPTION]... [FILE]...
```

Die relevanten Optionen sind:

- `-n` Die ersten *n* Zeilen jeder Datei werden ausgegeben; Default ist 10. Bei negativen Zahlen werden alle bis auf die letzten *n* Zeilen ausgegeben.
- `-c` Die ersten *n* Bytes jeder Datei werden ausgegeben. Bei negativen Zahlen werden alle außer den letzten *n* Bytes ausgegeben.
- `-q` Die Ausgabe der Dateinamen wird unterbunden.
- `-v` Die Dateinamen werden immer ausgegeben.

5.14 tail

`tail` gibt die letzten Zeilen der Eingabe aus. Der allgemeine Aufruf lautet:

```
tail [OPTION]... [FILE]...
```

Die häufigsten Optionen sind:

`-n`	Die letzten *n* Zeilen jeder Datei werden uasgegeben; Default ist 10. Steht vor der Zahl ein +, wird aber der n-ten Zeile ausgegeben.
`-c`	Die letzten *n* Bytes jeder Datei werden ausgegeben. Steht vor der Zahl ein +, wird ab dem n-ten Byte ausgegeben.
`-q`	Die Ausgabe der Dateinamen wird unterbunden.
`-v`	Die Dateinamen werden immer ausgegeben.
`-f`	`tail` wird versuchen, weitere Daten zu lesen, nachdem das Dateiende erkannt wurde. Das ist sinnvoll, wenn ein anderer Prozess in die entsprechende Datei hineinschreibt.
`-F`	`tail` wird die angegebene Datei erneut öffnen. Das ist nützlich, wenn Dateien umbenannt oder verschoben werden.
`-s`	Gibt die (ungefähre) Intervallzeit zum Prüfen in Sekunden an; es sind Kommazahlen erlaubt.
`--pid=<PID>`	Bei Verwendung mit `-f` wird tail sich beenden, wenn der angegebene Prozess terminiert wird.

5.15 dd

`dd` liest und schreibt Daten. Es ist z.B. dann nützlich, wenn es um eine bestimmte Menge an Daten geht. Beispiele für die Verwendung: `dd if=/dev/sda8 of=sdaf`

Kopiere die Partition `/dev/sda8` in die Datei sda8. Die Partition sollte nicht gemountet sein.
`dd if=sda8 of=/dev/sda8`
Das umgekehrte zum Vorherigen: eine Datensicherung wird zurückgespielt.
`dd bs=1MB const=10`
Kopiert 10MB von Standardeingabe auf Standardausgabe.
`dd bs=1M const=10`
Kopiert 10MiB von Standardeingabe auf Standardausgabe.
`dd conv=ascii`
Konvertiert die Eingabe von EBCDIC nach ASCII.
`dd conv=ucase`
Konvertiert Klein- in Großbuchstaben.
`dd if=/dev/urandom of=random.bin bs=1M count=1`
Kopiert 1MiB Zufallsdaten nach `random.bin`.
`dd if=random.bin of=randomb2.bin bs=1K count=1 skip=2 seek=4`
Kopiert 2048 Byte aus dem Intervall [2048; 3071] von `random.bin` nach [4096; 5119] in `random2.bin`.

5.16 tr

`tr` ändert oder löscht Zeichen aus der Eingabe. Der allgemeine Aufruf lautet:

```
tr [OPTIONS]... SET1 [SET2]
```

Beispiele für die Verwendung:
`tr 'abc' 'def'`
Ersetzt in der Eingabe `a` durch `d`, `b` durch `e` und `c` durch `f`.
`tr -d 'x'`
Entfernt alle x aus der Eingabe, bzw. ersetzt sie mit nichts.
`tr -s ab`
Entfernt doppelte Vorkommen von `a` und `b` in Text. Zum Beispiel wird $a+$ zu a, $b+$ zu b.

5.17 ts

Das Programm `ts` schreibt vor alle Eingabezeichen die jeweils aktuelle Zeit. Man kann das Format wie bei strftime angeben. Mit der Option `-r` wird eine vorhandene Zeitangabe in eine relativ zur aktuellen Zeit umgewandelt.

5.18 match

`match` wiederholt einen Programmaufruf zyklisch und stellt die Ausgabe dar. Auf diese Weise kann man leicht verfolgen, welche Teile sich ändern. Der allgemeine Aufruf lautet:

```
match [OPTIONS] command
```

Nützliche Optionen sind:

- `-d` Die Unterschiede zur vorherigen Ausgabe werden durch Invertierung hervorgehoben. Bei angehängtem `=permanent` werden alle Änderungen seit dem ersten Aufruf hervorgehoben.
- `-n` Gibt die Anzahl Sekunden zwischen den Programmaufrufen an.
- `-g` match wird beendet, wenn sich der Exitcode des Kommandos ändert.

5.19 nl

Mit `nl` werden die Zeilen der Eingabe nummeriert (number lines). Der allgemeine Aufruf lautet:

```
nl [OPTION]... [FILE]...
```

Nützliche Optionen sind:

- `-b` Den angegebenen Stil zur Nummerierung verwenden; Default ist die Nummerierung von nichtleeren Zeilen.
- `-i` Die Zeilennummer jedes Mal um den angegebenen Wert erhöhen.
- `-n` Den angegebenen Stil für Zahlen verwenden; Default ist rechtsbündig ohne führende Nullen.
- `-s` Die angegebene Zeichenkette zwischen Zeilennummer und -text einsetzen; Default ist Tabulator.
- `-v` Die Nummer der ersten Zeile vorgeben.
- `-w` Die angegebene Breite für die Zeilennummer verwenden; Default ist 6. Oft wird man die Option `-b` mit dem Stil `a` verwenden, damit alle Zeilen nummeriert werden.

5.20 cat

„Normalerweise“ dient `cat` zum Aneinanderhängen von Dateien auf die Standardausgabe (*concatenate*). Der häufigere Fall ist aber das schlichte Ausgeben von Dateien auf dem Bildschirm. Der allgemeine Aufruf lautet:

```
cat [OPTION]... [FILE]...
```

Nützliche Optionen sind:

`-E` $ vor Zeilenumbrüchen ausgeben.
`-n` Die Eingabezeilen werden nummeriert.
`-s` Aus mehreren aufeinanderfolgenden Leerzeilen wird eine gemacht.
`-T` Tabulatoren als `^I` ausgeben.
`-v` ^- und M-Notation verwenden (außer für Tabulator und Zeilenenden).

Beispiele für die Verwendung:

```
cat a
```

Die Datei `a` ausgeben. Meist ist `less a` dafür sinnvoller!

```
cat a b c > d
```

Die Dateien `a`, `b` und `c` aneinanderhängen und in die Datei `d` umleiten.

```
(echo -en "obase=16\ntime=";
 date 't%s';
 echo -e "\ntime+(25567*60*60*24)" )
 | bc | (echo -n "00000000: "; cat)
 | xxd -r
```

Big-Endian-Wert der Sekunden seit 1900-01-01T00:00:00Z ausgeben.

```
{foo; bar; cat mumble; baz;} | whatever
```

Leitet die Ausgaben der Programme weiter; das dritte Teilkommando gibt die Datei mumble aus; ihr Inhalt kommt also zwischen den Ausgaben der anderen Befehle.

Siehe zur Benutzung (bzw. zur Nichtbenutzung) auch den „useless use of cat award“ hier.

5.21 netcat

`netcat` macht über eine IP-Verbindung, was `cat` lokal macht: Dateien lesen bzw. schreiben. Beispiele für die Verwendung:

```
nc mail.server.net 25
```

Öffnet eine Verbindung zu Port 25 des angegebenen Rechners (ähnlich telnet).

```
nc -v2n 192.168.0.1 80-90
```

Führt einen gesprächigen Scan der UDP-Ports 80 bis 90 des angegebenen Rechners durch.

```
nc -l -p 12345 > file
```

```
nc hostname 12345 < file
```

Überträgt die Datei file von einem Rechner auf einen anderen.

5.22 tac

Als Gegenteil von `cat` dreht |verb|tac| die Eingabe um: Es wird zuerst die letzte, und als letztes die erste Zeile ausgegeben.

5.23 rev

Mit `rev` wird die Eingabe zeilenweise umgedreht: das letzte Zeichen jeder Zeile kommmt zuerst, das erste zuletzt.

5.24 pv

Mit `pv` (pipe view) kann man sich die Datenrate einer Pipe anschauen und sie begrenzen. Wenn die Gesamtgröße der Daten bekannt ist, kann außerdem die Zeit bis zum Ende geschätzt werden. `pv` ist praktisch für Test auf geringe Datenrate und bei einigen Szenarien sogar notwendig. Der allgemeine Aufruf lautet:
`pv [OPTION]... [FILE]` Beispiele für die Verwendung:

```
pv file | nc -w 1 somewhere.com 3000
```

Die Geschwindigkeit einer netcat-Übertragung ausgeben.

`<file pv -s 12345 | nc -w 1 somewhere.com 3000` Das gleiche, aber eine Dateigröße vorgeben.

`pv -L 5k | nc -n somewhere.com 3000`

Die Datenrate auf 5kB pro Sekunde begrenzen; bei UDP können leicht Daten verloren gehen.

5.25 tee

Mit tee kopiert man die Standardeingabe in jede angegebene Datei und auf die Standardausgabe. Das dient dem Speichern von Daten, die durch eine Pipeline laufen. Der Name kommt von einem T-Stück, das den Datenstrom darstellt.

Der allgemeine Aufruf lautet:

`tee [OPTION]... [FILE]...` Die relevanten Optionen sind:

`-a` Die Dateien werden nicht überschrieben, sondern die Daten werden angehängt.

`-i` Das Interrupt-Signal wird ignoriert.

Beispiele für die Verwendung `make | tee make.output` make wird ausgeführt und die Ausgabe gleichzeitig ausgegeben und gespeichert.

`make | tee make.output | grep 'warning\|error'` Make wird ausgeführt, die Ausgabe gespeichert und nach Fehlermeldungen und Warnungen gefiltert.

5.26 pee

Auf die gleiche Weise wie mit `tee` kann man mit `pee` Eingaben an mehrere Prozesse weiterleiten. Der allgemeine Aufruf lautet:

`pee ['command']...` Wenn man die Ausgabe auch sehen möchte, dann hilft das Kommando `cat`.

Beispiel für die Verwendung: `tr -d '\r' file | pee 'sort -n > sorted' 'sort -R >` CRs aus file entfernen, Datei sortieren und „zufällig sortiert“ speichern.

6 Verwendung eines Debuggers

6.1 Was ist ein Debugger?

Ein Debugger ist ein Programm, mit dem man ein anderes Programm schrittweise ausführen und beobachten kann. Das untersuchte Programm sollte Debug-Informationen enthalten, da man sonst nur auf Ebene des Assemblercodes arbeiten kann, ohne den Quelltext zu sehen.

6.2 Wann verwendet man einen Debugger?

Im allgemeinen wird ein Debugger zur Fehlersuche verwendet. Man kann prüfen, welche Zweige des Programms ausgeführt werden, und ob Variablen die erwarteten Werte enthalten. Das schrittweise ausführen erlaubt aber auch, an bestimmten Stellen anzuhalten, um Raceconditions auf die zu untersuchenden Wege „hinzubiegen“.

6.3 Welche Programme kann man debuggen?

Da die zu debuggenden Programme vom Compiler mit Debug-Informationen versehen werden müssen, ist „sinnvolles“ Debuggen meist auf Programme beschränkt, deren Quelltext man hat. Da der Quelltext in den Debug-Informationen enthalten ist, wird kaum ein Hersteller entsprechende Binärdateien ausliefern, wenn der Quelltext nicht bereits veröffentlicht ist. Das Debuggen auf Ebene des Assembler-Codes ist umständlich, langwierig und wird meist nicht viel bringen, weil man auf unterster Ebene nachvollziehen muss, was bereits im Quelltext schwierig war (sonst hätte man den Fehler gar nicht gemacht).

Viele Compiler erlauben nicht, sowohl Debuginformationen zu hinterlegen, als auch Optimierungen anzuwenden (beim GNU-Compiler ist das möglich). Optimierungen des Compilers haben aber Einfluss auf den Code, damit wird vielleicht anderer Code gezeigt als ausgeführt, oder man kann Werte nicht ausgeben, weil der Compiler die Variablen bei der Optimierung entfernt hat.

Außerdem müssen Compiler und Debugger zueinander passen. Meist gibt es beide Programme vom selben Hersteller und sie sind gut aufeinander abgestimmt.

6.4 Vorgehensweise

Die `gcc`-Option zum Einbetten von Debug-Informationen ist `-g`. Sie muss beim Compileraufrauf angegeben werde; beim Linkeraufruf ist sie nicht nötig. Das erzeugte Programm kann anschließend durch einen Debugger gestartet werden. Je nach Debugger läuft es sofort los (und wird meist bei `main` oder zu Beginn der Startup-Routine abgehalten), oder man kann noch Einstellungen vornehmen und Informationen abrufen, bevor es startet.

6.5 Schrittweises Ausführen

Es gibt mehrere Arten, den Code schrittweise auszuführen. Zum einen kann man in aufgerufene Funktionen hineinspringen und sie ebenfalls schrittweise ausführen; zum anderen kann man aufgerufene Funktionen als einzelnen

Schritt ausführen. Die Aufrufe sind zumeist Zeilenweise; wenn mehrere Anweisungen in einer Zeile stehen, werden sie also alle ausgeführt. Außerdem ist es möglich, den Code auf der Ebene von Maschineninstruktionen abzuarbeiten. Das ist mühsam, erlaubt aber die Werte in den Prozessorregistern zu untersuchen. Manche Fehler findet man nur auf diese Weise (das sind aber nur wenige).

6.6 Breakpoints

Oft möchte man eine bestimmte Stelle untersuchen und dafür nicht zeilenweise vom Programmbeginn vorgehen; beim Debuggen von Callbackfunktionen funktioniert diese Vorgehensweise außerdem nicht. In diesen Fällen kann man einen Breakpoint auf eine Zeile setzen. Wenn das Programm während des Ausführens dort vorbeikommt, wird der Debugger es anhalten und auf weitere Benutzeranweisungen warten.

Zusätzlich kann man Bedingungen für Breakpoints vergeben, damit das Programm dort nur unter bestimmten Umständen angehalten wird oder z.B. erst ab dem n-ten Ausführen der Zeile (intern passiert das in der Regel dadurch, dass der Debugger das Programm wie gewöhnlich anhält, die Bedingung prüft und bei Nichterfülltsein das Programm fortsetzt).

Je nach Debugger und Hardwarearchitektur kann die Anzahl der Breakpoints begrenzt sein, wenn Prozessorregister dafür verwendet werden. Dieses Problem tritt besonders beim Debuggen in externer Hardware auf. Bisweilen hat man z.B. nur zwei Breakpoints zur Verfügung, (bzw. drei, wenn man auf schrittweises Ausführen verzichtet). Das ist spätestens bei Switch-Anweisungen problematisch, weil für jede case-Anweisung ein (impliziter) Breakpoint nötig ist.

Sogenannte Watchpoints ermöglichen das Anhalten des Programms, wenn eine Variable oder ein bestimmter Speicherbereich beschrieben oder gelesen wird. Nicht alle Debugger unterstützen Watchpoints.

Bei manchen Systemen muss man das Programm anhalten, um Break- oder Watchpoints zu setzen; nicht alle Debugger tun das automatisch.

6.7 Anzeigen von Werten

Der Debugger hat Kontrolle über den Speicher des gestarteten Programms und kann Variablen, Speicheradressen und Prozessorregister lesen und schreiben. Dazu existieren Kommandos oder Menüs für die einmalige bzw. dauerhafte Darstellung der Werte. Oft ist es möglich, andere Darstellungen als dezimale zu verwenden, z.B. hexadezimale, Interpretation als Zeichen oder als Zeichenkette.

6.8 Setzen von Variablen

Während des Debuggens ist oft interessant, wie sich das Programm verhielte, wenn ein anderer Wert in einer Variablen oder einem Register stünde. Für solche Tasks kann man Werte orgeben und (unabhängig davon) die Ausführung an einer bestimmten Stelle fortsetzen, um Teile des Programms zu wiederholen oder zu überspringen. Es ist dabei nicht empfehlenswert, eine Stelle außerhalb der aktuellen Funktion anzugeben. Manche Debugger erlauben es sogar, Funktionen aufzurufen, um ihren Rückgabewert auszugeben bzw. ihre Seiteneffekte zu verursachen.

6.9 Beschränkungen von Debuggern

Neben der bereits erwähnten zum Teil begrenzten Anzahl von Breakpoints, gibt es oft keine native Darstellung von enums oder boole'schen Werten, sondern es wird der Zahlenwert angegeben. Außerdem haben Debugger meist Schwierigkeiten mit Makros: sie können die Werte entsprechender Ausdrücke normalerweise nicht berechnen.

6.10 Verwendung des GNU-Debuggers

Der GNU-Debugger hat den Namen `gdb`. Der übliche Aufruf ist `gdb <Programm>`.
Vor dem Start des Programms kann man Argumente einstellen: `set args <Argumente>`.
Das Kommando `run` startet das Programm. Nimmt man stattdessen `start`, wird im Programm bei `main` angehalten.