Kapitel 1
Einführung

Als Autor eines Fachbuchs steht man immer vor der Frage, wo man beginnen soll. Fängt man bei null an, so wird eine Menge Raum für interessantere Aufgabenschwerpunkte verschenkt. Fordert man hingegen am Anfang gleich zu viel, läuft man Gefahr, dass das Buch schnell im Regal verstaubt oder zur Versteigerung angeboten wird.

1.1 Was sollten Sie wissen?

Da Sie sich entschieden haben, mit der Shellscript-Programmierung anzufangen, können wir davon ausgehen, dass Sie bereits ein wenig mit Linux bzw. einem UNIX-artigen System vertraut sind. Vielleicht haben Sie auch schon Erfahrungen mit anderen Programmiersprachen gemacht, was Ihnen hier einen gewissen Vorteil verschafft. Vorhandene Programmiererfahrungen sind allerdings keine Voraussetzung, um mit diesem Buch zu arbeiten. Es wurde so konzipiert, dass selbst ein Anfänger recht einfach und schnell ans Ziel kommt. Dies haben wir deshalb getan, weil die Shellscript-Programmierung im Gegensatz zu anderen Programmiersprachen wie beispielsweise C/C++ oder Java erheblich einfacher zu erlernen ist (auch wenn Sie beim ersten Durchblättern des Buchs einen anderen Eindruck haben sollten).

Aber was heißt »mit Linux bzw. UNIX bereits ein wenig vertraut«? Dass Sie folgende Punkte beherrschen, müssen wir einfach von Ihnen erwarten – ansonsten könnte der Buchtitel gleich »Linux/UNIX – Eine Einführung« lauten:

- An- und Abmelden am System.
- Arbeiten mit einem (beliebigen) Texteditor (mehr dazu folgt in Abschnitt 1.8).
- Umgang mit Dateien und Verzeichnissen – sprich Umgang mit grundlegenden Kommandos wie `cp`, `pwd`, `ls`, `cd`, `mv`, `mkdir`, `rmdir`, `cat`, ...: Sind diese Kenntnisse nicht vorhanden, ist das nicht weiter schlimm, denn alle Kommandos werden in diesem Buch mehr als einmal verwendet und auch in einem Crashkurs kurz beschrieben.
- Zugriffsrechte: Wer bin ich, was darf ich, und welche Benutzer gibt es? Oder einfach: Was bedeuten die »komischen« Zeichen `rwx` bei den Dateien und Verzeichnissen? (Auch hierzu gibt es eine kurze Einführung).
- Verzeichnisstruktur: Wo bin ich hier, und wo finde ich was? Wenn Sie nicht wissen, wo Sie »zu Hause« sind oder wie Sie dorthin kommen, wird Ihnen die ganze Umgebung fremd vorkommen (aber auch ein Immigrant kann sich einleben).

- Grundlegende Kenntnisse eines Dateisystems: Wo ist meine Festplatte, mein Netzwerkspeicher oder USB-Stick und vor allem: Wie werden diese bezeichnet?
- Kommunikation: Dieses Buch behandelt auch netzwerkspezifische Dinge, weshalb Sie zumindest den Weg in den »**W**elt**W**eiten**W**älzer« gefunden und den Umgang mit der elektronischen Post beherrschen sollten.

Linux/UNIX-Kommandoreferenz

Sofern Sie mit einigen Shell-Kommandos, die in den Scripts verwendet werden, nicht zurechtkommen oder diese nicht kennen, finden Sie in Kapitel 14 eine Linux/UNIX-Kommandoreferenz, in der Ihnen zumindest die Grundlagen (die gängigsten Optionen) erläutert werden. Detailliertere Informationen bleiben selbstverständlich weiterhin den Manualpages (Manpages) vorbehalten.

1.1.1 Zielgruppe

Mit diesem Buch wollen wir eine recht umfangreiche und keine spezifische Zielgruppe ansprechen. Profitieren von der Shellscript-Programmierung können alle – vom einfachen Linux/UNIX-Anwender bis hin zum absolut überbezahlten (oder häufig auch schlecht bezahlten) Systemadministrator, also einfach alle, die mit Linux/UNIX zu tun haben bzw. vorhaben, damit etwas mehr anzufangen.

Ganz besonders gut eignet sich die Shellscript-Programmierung auch für den Einstieg in die Welt der Programmierung. Sie finden hier viele Konstrukte (Schleifen, Bedingungen, Verzweigungen etc.), die auch in den meisten anderen Programmiersprachen in ähnlicher Form verwendet werden (wenngleich die Syntax häufig ein wenig anders ist).

Da Sie sehr viel mit den »Bordmitteln« (Kommandos/Befehlen) des Betriebssystems arbeiten, bekommen Sie auch ein gewisses Gefühl für Linux/UNIX. So haftet Menschen, die mit Shellscript programmieren, häufig ein Guru-Image an, einfach weil sie tiefer in die Materie einsteigen müssen als viele GUI-verwöhnte Anwenderinnen und Anwender. Trotzdem ist es leichter, die Shellscript-Programmierung zu erlernen als irgendeine andere Hochsprache, beispielsweise C, C++, Java oder C#.

Die primäre Zielgruppe sind aber ganz klar Personen in der Systemadministration und/oder Webmaster (mit SSH-Zugang). In der Systemadministration von Linux/UNIX-Systemen ist es häufig unumgänglich, sich mit der Shellscript-Programmierung auseinanderzusetzen. Letztendlich ist ja auch jeder einzelne Linux/UNIX-(Heim-)Benutzer mit einem PC eine Systemadministratorin oder ein Systemadministrator und profitiert enorm von den neu hinzugewonnenen Kenntnissen.

macOS ist auch UNIX

Natürlich dürfen sich diejenigen, die einen Mac verwenden, ebenfalls angesprochen fühlen. Alles, was wir hier im Buch beschreiben, wurde sorgfältig auf macOS getestet. Auch bei dieser 7. Auflage des Buchs haben wir auf die Mac-Kompatibilität geachtet, und natürlich gehen wir bei Bedarf auf die Unterschiede ein. Mit macOS Catalina (10.15) hat Apple die Z-Shell als Standard-Shell voreingestellt. In Abschnitt 1.5.8 erklären wir Ihnen, wie Sie die Bash-Version 5 auf Ihrem Mac installieren und nutzen können.

1.1.2 Notation

Die hier verwendete Notation ist recht einfach und zum Teil eindeutig aufgebaut. Wenn wir die Eingabe mit der Tastatur (bzw. einer Tastenkombination) beschreiben, kennzeichnen wir das mit einem entsprechenden Tastenzeichen. Lesen Sie beispielsweise Strg+C, bedeutet dies, dass hier die Tasten »Steuerung« (Control oder auch Ctrl) und »C« gleichzeitig gedrückt werden; finden Sie Esc vor, dann ist das Drücken der Escape-Taste gemeint.

Tastenbelegung bei macOS

Während die Tastenbelegung und -verwendung in Windows- und Linux- sowie vielen UNIX-Systemen recht ähnlich ist, hat ein Mac-System eine geringfügig andere Tastatur. Auf die Unterschiede zwischen einer Mac- und einer PC-Tastatur gehen wir kurz in Abschnitt C.1 ein. An dieser Stelle weisen wir daher nur kurz darauf hin, dass eine Tastenkombination wie Strg+C auf dem Mac mit Ctrl+C ausgeführt werden muss und nicht, wie Sie vielleicht annehmen, mit cmd+C. Mac-Veteranen wissen das zwar, aber da das System immer beliebter wird und es somit mehr Umsteiger gibt, sollte dies hier kurz erwähnt werden.

Sie werden sehr viel mit der Shell arbeiten. Als Shell-Prompt des normalen Users in der Kommandozeile wird `you@host >` verwendet. Diesen Prompt müssen Sie also bei der Eingabe nicht mit angeben (logisch, aber es sollte erwähnt werden). Wird hinter diesem Shell-Prompt die Eingabe in fetten Zeichen dargestellt, handelt es sich um eine Eingabe in der Kommandozeile, die vom Benutzer (meistens von Ihnen) vorgenommen und mit einem ↵-Tastendruck bestätigt wurde:

```
you@host > Eine_Eingabe_in_der_Kommandozeile
```

Folgt hinter dieser Zeile eine weitere Zeile ohne den Shell-Prompt `you@host >` und nicht in fetter Schrift, dann handelt es sich in der Regel um die Ausgabe, die die Shell aus der zuvor getätigten Eingabe erzeugt hat (was im Buch meistens den Aktionen Ihres Shellscripts entspricht):

```
you@host > Eine_Eingabe_in_der_Kommandozeile
Die erzeugte Ausgabe
```

Finden Sie stattdessen das Zeichen # als Shell-Prompt, handelt es sich um einen Prompt des Superusers (root). Selbstverständlich setzt dies voraus, dass Sie auch die entsprechenden Rechte haben. Das ist nicht immer möglich und wird daher in diesem Buch so selten wie möglich eingesetzt.

```
you@host > whoami
nomaler_User
you@host > su
Passwort: ********
# whoami
root
# exit
you@host > whoami
normaler_User
```

Vorübergehende Root-Rechte mit sudo

Viele UNIX-artige Betriebssysteme wie Linux oder macOS gehen hier einen anderen Weg und verwenden den Befehl `sudo` (*substitute user do* oder *super user do*). Mit diesem Befehl können Sie einzelne Programme bzw. Kommandos mit den Rechten eines anderen Benutzers (beispielsweise mit Root-Rechten) ausführen. Der Befehl `sudo` wird dabei vor den eigentlich auszuführenden Befehl geschrieben. Wollen Sie für mehr als nur einen Befehl als Superuser arbeiten, können Sie mit `sudo -s` dauerhaft wechseln. Die Einstellungen für `sudo` werden in der Datei */etc/sudoers* gespeichert. Mehr Informationen zu `sudo` finden Sie auf der entsprechenden Manualpage oder auf der offiziellen Webseite (*http://www.sudo.ws*).

1.2 Was ist eine Shell?

Für den einen ist eine Shell nichts anderes als ein besseres »command.com« aus der MS-DOS-Welt. Eine andere wiederum bezeichnet die Shell lediglich als Kommandozeile, und für einige ist die Shell die Benutzeroberfläche schlechthin. Diese Meinungen resultieren daraus, dass die einen die Shell lediglich verwenden, um Installations- bzw. Konfigurationsscripts auszuführen. Andere verwenden auch häufiger mal die Kommandozeile zum Arbeiten, und die letzte Gruppe setzt die Shell so ein, wie andere die grafischen Oberflächen benutzen.

Für die meisten User besteht eine Benutzeroberfläche aus einem Fenster oder mehreren Fenstern, die man mit der Maus oder der Tastatur steuern kann. Wie kann man

(oder wir Autoren) also behaupten, die Shell sei eine Benutzeroberfläche? Genauer betrachtet, ist eine Benutzeroberfläche nichts anderes als eine Schnittstelle, die zwischen der Hardware des Systems (mit der der normale Benutzer nicht gern direkt etwas zu tun haben will und es auch nicht sollte) und dem Betriebssystem kommuniziert. Dies ist natürlich relativ einfach ausgedrückt, denn in Wirklichkeit findet eine solche Kommunikation meistens über mehrere abstrakte Schichten statt. Benutzer geben eine Anweisung ein, diese wird interpretiert und in einen Betriebssystemaufruf umgewandelt – auch *System-Call* oder *Sys-Call* genannt. Jetzt wird auf die Rückmeldung des Systems gewartet und ein entsprechender Befehl umgesetzt – oder im Fall eines Fehlers eine Fehlermeldung ausgegeben.

Kommandozeileneingabe

Noch genauer gesagt, durchläuft eine Eingabe der Kommandozeile einen Interpreter und ist somit praktisch nichts anderes als ein Shellscript, das von der Tastatureingabe aus, ohne es zwischenzuspeichern, in einem Script ausgeführt wird.

Man spricht dabei auch von einem Systemzugang. Einen solchen Systemzugang (Benutzeroberfläche) können Sie als normaler Benutzer wie folgt erlangen:

- über die **Kommandozeile** (Shell): Hierbei geben Sie in einem Fenster (beispielsweise `xterm`) oder häufig auch in einem Vollbildschirm (eben einem echten `tty`) über die Tastatur entsprechende Kommandos ein und setzen so die Betriebssystemaufrufe in Gang.
- über die **grafische Oberfläche** (Windowmanager unter Xorg oder Wayland wie KDE, Gnome etc.): Auch hierbei geben Sie mit der Maus oder der Tastatur Anweisungen (Kommandos) ein, die auf der Betriebssystemebene ebenfalls nicht vorbeikommen.

Kommandozeile und Interpreter

Häufig wird die Kommandozeile als *Shell* bezeichnet. Diese Shell ist aber nicht gleichzusetzen mit dem »Interpreter« Shell. Der Interpreter ist die Schnittstelle, durch die überhaupt erst die Kommunikation zwischen der Ein-/Ausgabe und dem System möglich wird.

Auch wenn es auf den ersten Blick den Anschein hat, dass man über die grafische Oberfläche den etwas schnelleren und bequemeren Weg gewählt hat, ist dies häufig nicht so. Je mehr Erfahrung Anwenderinnen und Anwender haben, desto eher werden sie mit der Kommandozeile schneller ans Ziel kommen als mit dem GUI.

Hurra, die ganze Arbeit war umsonst: KDE und Gnome benötigen wir nicht mehr, und die fliegen jetzt von der Platte! Nein, keine Sorge, wir wollen Sie keinesfalls von der grafischen Oberfläche abbringen, ganz im Gegenteil. Gerade ein Mix aus grafischer

Oberfläche und Kommandozeile macht effektiveres Arbeiten erst möglich. Wie sonst wäre es z. B. zu realisieren, mehrere Shells gleichzeitig zur Überwachung zu betrachten bzw. einfach mehrere Shells gleichzeitig zu verwenden?

Langer Rede kurzer Sinn: Eine Shell ist eine Methode, das System über die Kommandozeile zu bedienen.

Tab-Completion

Wenn Sie mit der Shell arbeiten, wird Ihnen rasch auffallen, dass sie recht viel Tipparbeit von Ihnen verlangt. Besonders schlimm können dabei Pfade zu Dateien sein: Was in der grafischen Oberfläche ein paar Klicks auf einige Ordner sind, muss in der Shell umständlich eingegeben werden.

Als Lösung dafür gibt es die sogenannte Tab-Completion. Mit der Taste ⇆ versucht die Shell, den Rest Ihrer Eingabe zu vervollständigen. Wenn Sie die ersten Zeichen eines Befehls oder einer Datei eingeben und dann ⇆ drücken, ergänzt die Shell – so gut es geht – die restliche Eingabe.

Viele Tools auf der Kommandozeile wie etwa Docker oder Kubernetes bringen Ergänzungsmodule für die Completion mit, um etwa auch die umständlichen Namen von Containern oder anderen Objekten zu vervollständigen. Wenn Sie also schneller und viel komfortabler mit der Shell arbeiten wollen, sollten Sie sich gut mit ⇆ anfreunden und ggfs. weitere Completion-Module installieren.

Historisch interessierte User werden sich wohl fragen, wieso die Bezeichnung *Shell* (was so viel wie »Muschel« oder auch »Schale« bedeutet) verwendet wurde. Der Name bürgerte sich unter Linux/UNIX ein (eigentlich war zuerst UNIX da, aber auf diese Geschichte gehen wir hier nicht ein), weil sich eine Shell wie eine Schale um den Betriebssystemkern legt.

Vereinfacht ausgedrückt, ist die Shell ein vom Betriebssystemkern unabhängiges Programm bei der Ausführung – also ein simpler Prozess. Ein Blick auf das Kommando `ps` (ein Kommando, mit dem Sie Informationen zu laufenden Prozessen ermitteln können) bestätigt Ihnen das:

```
you@host > ps
  PID TTY          TIME CMD
 5890 pts/40   00:00:00 bash
 5897 pts/40   00:00:00 ps
```

Hier läuft als Shell beispielsweise die Bash (mehr zu den verschiedenen Shells und zur Ausgabe von `ps` erfahren Sie in Abschnitt 1.5, »Die Shell-Vielfalt«, und Abschnitt 1.8.4, »Ausführen«).

Natürlich kann man die Shell nicht einfach als normalen Prozess bezeichnen, dazu ist sie ein viel zu mächtiges und flexibles Programm. Selbst der Begriff »Kommandozeile« ist ein wenig zu schwach. Eher schon würde sich der Begriff »Kommandosprache« eignen. Und wenn die Shell schon *nur* ein Programm ist, sollte erwähnt werden, dass sie auch jederzeit gegen eine andere Shell ausgetauscht werden kann. Es gibt also nicht nur die eine Shell, sondern mehrere – aber auch hierzu sagen wir in Kürze mehr.

Shell unter macOS aufrufen

Unter macOS können Sie ein Terminal mit der Standard-Shell über den Ordner */Programme/Dienstprogramme/* aufrufen. Tipp: Schneller wechseln Sie in das Verzeichnis *Dienstprogramme*, wenn Sie im Finder die Tastenkombination cmd+⇧+U betätigen. Eine gute Alternative ist auf jeden Fall `iterm`. Die aktuellste Version finden Sie unter *https://www.iterm2.com/*.

1.3 Hauptanwendungsgebiet

Dass eine Shell als Kommandosprache oder als Scriptsprache eine bedeutende Rolle unter Linux/UNIX spielt, dürfte Ihnen bereits bekannt sein. Wichtige Dinge, wie die Systemsteuerung oder die verschiedenen Konfigurationen, werden über die Shell (und die damit erstellten Shellscripts, also die Prozeduren) realisiert.

Vor allem werden Shellscripts auch dazu verwendet, täglich wiederkehrende Aufgaben zu vereinfachen und Vorgänge zu automatisieren. Dies schließt die Hauptanwendungsgebiete wie die (System-)Überwachung von Servern, das Auswerten von bestimmten Dateien (Logdateien, Protokollen, Analysen, Statistiken, Filterprogrammen etc.) und ganz besonders die Systemadministration mit ein – Dinge, mit denen sich alle IT-Fachleute (ob bewusst oder unbewusst) auseinandersetzen müssen.

Es kann also ganz plausible Gründe dafür geben, ein Shellscript zu erstellen. Im Bereich der Automation des privaten Haushalts mit »smarten« Produkten kann es beispielsweise sinnvoll sein, ein Shellscript einzusetzen, wenn die mitgelieferte App gewünschte Funktionen nicht bietet.

Auch bei einfachsten Eingaben auf der Kommandozeile, die Sie ständig durchführen, können Sie sich das Leben dadurch erleichtern, dass Sie diese Kommandos zusammenfassen und einfach ein »neues« Kommando oder, etwas exakter formuliert, ein Shellscript schreiben.

1.3.1 Was ist ein Shellscript?

Bestimmt haben Sie das eine oder andere Mal schon ein Kommando oder eine Kommandoverkettung in der Shell eingegeben, zum Beispiel:

```
you@host > ls /usr/include | sort | less
af_vfs.h
aio.h
aliases.h
alloca.h
ansidecl.h
a.out.h
argp.h
argz.h
...
```

Tatsächlich stellt diese Kommandoverkettung schon ein Programm bzw. Script dar. Hier leiten Sie z. B. mit `ls` (womit Sie den Inhalt eines Verzeichnisses auflisten können) die Ausgabe aller im Verzeichnis */usr/include* enthaltenen Dateien mit einer Pipe an das Kommando `sort` weiter. `sort` sortiert die Ausgabe des Inhalts von */usr/include*, dessen Daten ja vom Kommando `ls` kommen, lexikografisch. Auch die Ausgabe von `sort` wird nochmals durch eine Pipe an `less` weitergegeben, also an den Pager, mit dem Sie die Ausgabe bequem nach oben bzw. unten scrollen können. Zusammengefasst, könnten Sie jetzt diese Befehlskette in einer Datei (einem Script) speichern und mit einem Aufruf des Dateinamens jederzeit wieder starten (hierauf wird noch ausführlicher eingegangen). Rein theoretisch hätten Sie damit schon ein Shellscript erstellt. Wäre das alles, müssten Sie kein ganzes Buch lesen und könnten gleich auf das Kommando `alias` oder ein anderes bequemeres Mittel zurückgreifen. Für solch einfache Aufgaben müssen Sie nicht extra ein Shellscript schreiben. Ein Shellscript wird allerdings unter anderem dann nötig, wenn Sie

- Befehlssequenzen mehrmals ausführen wollen.
- aufgrund einer bestimmten Bedingung einen Befehl bzw. eine Befehlsfolge ausführen wollen.
- Daten für weitere Bearbeitungen zwischenspeichern müssen.
- eine Eingabe vom Benutzer benötigen.

Für diese und andere Aufgaben finden Sie bei einer Shell noch zusätzliche Konstrukte, wie sie ähnlich auch in anderen Programmiersprachen vorhanden sind.

Natürlich dürfen Sie nicht den Fehler machen, die Shellscript-Programmierung mit anderen höheren Programmiersprachen gleichzusetzen. Aufwendige und schnelle Grafiken, zeitkritische Anwendungen und eine umfangreiche Datenverwaltung (Stichwort: Datenbank) werden weiterhin mit Programmiersprachen wie C oder C++ erstellt.

Ein Shellscript ist also nichts anderes als eine mit Shell-Kommandos zusammengebastelte (Text-)Datei, bei der Sie entscheiden, wann, warum und wie welches Kommando oder welche Kommandosequenz ausgeführt wird.

Grafische Oberfläche mit Shellscripts

Wer jetzt denkt, die Shellscript-Programmierung sei nur eine »graue Maus«, der täuscht sich. Auch mit ihr lassen sich, wie in jeder anderen Sprache, Scripts mit einer grafischen Oberfläche erstellen. Ein Beispiel dazu finden Sie in Kapitel 16, »GUIs und Grafiken«, mit `Zenity` und `YAD`.

Dennoch kann ein Shellscript, je länger es wird und je komplexer die Aufgaben werden, auf den ersten Blick sehr kryptisch und kompliziert sein. Dies sagen wir nur für den Fall, dass Sie vorhaben, das Buch von hinten nach vorne durchzuarbeiten.

1.3.2 Vergleich mit anderen Sprachen

Das Vergleichen von Programmiersprachen war schon immer ein ziemlicher Irrsinn. Jede Programmiersprache hat ihre Vor- und Nachteile und dient letztendlich immer als Mittel, um eine bestimmte Aufgabe zu erfüllen. Somit hängt die Wahl der Programmiersprache davon ab, welche Aufgabe sich Ihnen stellt.

Zumindest können wir ohne Bedenken sagen, dass Sie mit einem Shellscript komplexe Aufgaben erledigen können, für die sich andere Programmiersprachen kaum bis gar nicht eignen. In keiner anderen Sprache haben Sie die Möglichkeit, die unterschiedlichsten Linux/UNIX-Kommandos zu verwenden und so zu kombinieren, wie Sie es gerade benötigen. Einfachstes Beispiel: Geben Sie den Inhalt eines Verzeichnisses in lexikografischer Reihenfolge aus. Bei einem Shellscript geht dies mit einem einfachen `ls | sort` – würden Sie Selbiges in C oder C++ erstellen, wäre dies, wie mit Kanonen auf Spatzen zu schießen.

Auf Umwegen zum Ziel kommen

Aber das soll nicht heißen, etwas Derartiges in C zu schreiben, sei sinnlos – der Lerneffekt ist hierbei sehr gut. Allerdings scheint uns dieser Weg für die, die eben »nur« ans Ziel kommen wollen bzw. müssen, ein ziemlicher Umweg zu sein. Da wir uns selbst sehr viel mit C beschäftigen, müssen wir eingestehen, dass wir natürlich auch häufig mit »Kanonen auf Spatzen geschossen« haben und es immer noch gern tun.

Allerdings haben auch Shellscript-Programme ihre Grenzen. Dies trifft ganz besonders auf Anwendungen zu, die eine enorme Menge an Daten auf einmal verarbeiten müssen. Werden diese Arbeiten dann auch noch innerhalb von Schleifen ausgeführt, kann dies schon mal recht langsam werden. Zwar werden Sie auf einem Einzelplatzrechner recht selten auf solche Engpässe stoßen, aber sobald Sie Ihre Scripts in einem Netzwerk wie beispielsweise auf einem Server schreiben, auf dem sich eine Menge weiterer Benutzer (etwa bei einem Webhoster mit SSH) befinden, dann sollten Sie überlegen, ob sich nicht eine andere Programmiersprache besser eignet. Häufig wird

Kapitel 10
Fehlersuche und Debugging

Sicherlich haben Sie im Verlauf der vielen Kapitel beim Tippen und Schreiben von Scripts genauso viele Fehler gemacht wie wir und sich gefragt: »Was passt denn jetzt wieder nicht?« Auch hier gilt, dass Fehler beim Lernen nichts Schlechtes sind. Aber sobald Sie Ihre Scripts auf mehreren oder fremden Rechnern ausführen, sollten Fehler nicht mehr vorkommen. Daher finden Sie in diesem Kapitel einige Hinweise dazu, wie Sie Fehler vermeiden können und – wenn sie bereits aufgetreten sind – wie Sie ihnen auf die Schliche kommen.

10.1 Strategien zum Vermeiden von Fehlern

Sicherlich, Fehler gänzlich vermeiden können Sie wohl nie, aber es gibt immer ein paar grundlegende Dinge, die Sie beherzigen können, um wenigstens den Überblick zu behalten.

10.1.1 Planen Sie Ihr Script

Zugegeben, bei einem Mini-Script von ein paar Zeilen werden Sie wohl kaum das Script planen. Allerdings haben wir persönlich schon die Erfahrung gemacht, dass ein geplantes Script wesentlich schneller erstellt ist als ein aus dem Kopf geschriebenes. Eine solche Planung hängt natürlich vom entsprechenden »Kopf« ☺ und dem Anwendungsfall ab. Aber das Prinzip ist einfach. Zuerst schreiben Sie auf (oder kritzeln hin), was das Script machen soll.

Ein Beispiel: Das Script soll auf mehreren Rechnern eingesetzt werden und ein Backup der Datenbank erstellen. Das Backup soll hierbei in einem separaten Verzeichnis gespeichert werden, und bei mehr als zehn Backup-Dateien soll immer die älteste gelöscht werden.

Auch wenn man sich bei den ersten Gedanken noch gar nicht vorstellen kann, wie das Script funktionieren bzw. aussehen soll, kann man trotzdem schon mit der Planung beginnen. Zwar haben Sie jetzt noch keine Zeile Code geschrieben, doch Sie werden feststellen, dass diese Liste schon die halbe Miete ist. Wenn Sie dann merken,

dass Sie das Script mit dem derzeitigen Wissensstand nicht realisieren können, wissen Sie wenigstens, wo sich Ihre Wissenslücken befinden, und können entsprechend nachhelfen.

```
#-------------------------------------------------------------#
Plan zum Backup-Script:

1.) Prüfen, ob das Verzeichnis für das Backup existiert (test), und
    gegebenenfalls das Verzeichnis neu anlegen (mkdir).
2.) Da mehrere Dateinamen (zehn) in einem Verzeichnis verwendet
    werden, einen entsprechenden Namen zur Laufzeit mit 'date'
    erstellen und in einer Variablen speichern (date).
3.) Einen Dump der Datenbank mit mysqldump und den entsprechenden
    Optionen erstellen (mysqldump).
4.) Den Dump gleich in das entsprechende Verzeichnis packen und
    archivieren (tar oder cpio).    !!! Dateinamen beachten !!!
5.) Gegebenenfalls Datenreste löschen.
6.) Verzeichnis auf Anzahl von Dateien überprüfen (ls -l | wc -l)
    und gegebenenfalls die älteste Datei löschen.

Notizen: Bei Fertigstellung crontab aktivieren - täglich um
         01.00 Uhr ein Backup erstellen.
#-------------------------------------------------------------#
```

10.1.2 Testsystem bereitstellen

Bevor Sie Ihr Script nach der Fertigstellung dem Ernstfall überlassen, sollten Sie auf jeden Fall das Script lokal oder auf einem Testsystem testen. Gerade Operationen auf einer Datenbank oder schreibende Zugriffe auf Dateien sollte man möglichst vorsichtig behandeln. Denn wenn irgendetwas schiefläuft, ist dies auf Ihrem Testsystem nicht so schlimm, aber bei einem oder gar mehreren Rechnern könnte solch ein Fehler Ihnen eine Menge Ärger einbringen sowie Zeit und vor allem Nerven kosten.

Es wäre auch sinnvoll, wenn Sie mehrere verschiedene Testsysteme zum Ausprobieren hätten – denn was auf Ihrer Umgebung läuft, muss nicht zwangsläufig auch auf einer anderen Umgebung laufen. Gerade wenn man häufig zwischen Linux und UNIX wechselt, gibt es hier und da doch einige Differenzen. Mit den heute zur Verfügung stehenden Rechnern und Virtualisierungslösungen kann man selbst auf einem Notebook problemlos verschiedenen Testumgebungen mit unterschiedlichen Betriebssystemen aufsetzen.

10.1.3 Ordnung ist das halbe Leben

Dass sich nicht alle an das Motto aus unserer Überschrift halten, zeigt folgendes Script:

```
# Eine Datei anlegen
# Name: creatfile

CreatFile=atestfile.txt

if [ ! -e $CreatFile ]
then
touch $CreatFile
if [ ! -e $CreatFile ] ; then
echo "Konnte $CreatFile nicht anlegen" ; exit 1
fi
fi

echo "$CreatFile angelegt/vorhanden!"
```

Zugegeben, das Script ist kurz und im Prinzip auch verständlich, doch wenn Sie solch einen Programmierstil über längere Scripts beibehalten, werden Sie allein bei der Fehlersuche keine große Freude haben. Folgende Punkte fallen auf den ersten Blick negativ auf:

- Es gibt keine Einrückungen bei den `if`-Anweisungen; es fehlt einfach an Struktur.
- Mehrere Befehle stehen in einer Zeile. Bei einer Fehlermeldung wird die Zeilennummer mit ausgegeben. Befinden sich darin mehrere Kommandos, wird die Fehlersuche erschwert.
- In diesem Script wurde die MS-übliche *ungarische Notation* verwendet, die in der Praxis sehr fehleranfällig sein kann. Hat man vergessen, wie die Variable `CreatFile` geschrieben wurde, und schreibt `creatfile` oder `creat_file`, ergibt sich bereits ein Fehler. Wer schon einmal die Win32-API zur Programmierung verwendet hat, der weiß, was wir meinen.
- Die Faulheit hat gesiegt; der Anwender weiß nach dem Script immer noch nicht, ob eine Datei angelegt wurde oder bereits vorhanden ist.
- Kommentarlose Scripts sind schwer zu pflegen. Schreiben Sie besser den einen oder anderen (vielleicht überflüssigen) Kommentar hin als gar keinen.

Hier sehen Sie das Script in einem etwas lesbareren Stil:

```
# Eine Datei anlegen
# Name: creatfile2
```

```
# Datei, die angelegt werden soll
creatfile=atestfile.txt

# Existiert diese Datei bereits ...
if [ ! -e $creatfile ]
then                      # Nein ...
   touch $creatfile       # Datei anlegen ...
   # Jetzt nochmals überprüfen ...
   if [ ! -e $creatfile ]
   then
      echo "Konnte $creatfile nicht anlegen"
      exit 1   # Script erfolglos beenden
   else
      echo "$creatfile erfolgreich angelegt"
   fi
fi

echo "$creatfile vorhanden!"
```

Das sieht doch schon viel besser aus. Natürlich kann Ihnen niemand vorschreiben, in welchem Stil Sie Ihre Scripts schreiben, dennoch wollen wir Ihnen hier einige Hinweise ans Herz legen, mit denen Sie (gerade wenn Sie vielleicht noch Anfänger sind) ein friedlicheres Shell-Leben führen können.

Variablen und Konstanten

Verwenden Sie einen sinnvollen Namen für eine Variable – mit `x`, `y` oder `z` kann niemand so recht etwas anfangen. Verwenden Sie beispielsweise Variablen in einer Funktion namens `removing()`, könnten Sie Variablen wie `rem_source`, `rem_dest` oder `rem_temp` verwenden. Hier gibt es viele Möglichkeiten, nur sollten Sie immer versuchen, dass Sie einer Variablen einen Namen geben, an dem man erkennt, was sie bedeutet – eine Variable benötigt also ein klar zu erkennendes Merkmal.

Außerdem sollten Sie Konstanten und Variablen auseinanderhalten. Wir bezeichnen hier Konstanten nicht als Variablen, die mit `typeset` als »readonly« markiert wurden, sondern meinen Variablen, die sich zur Laufzeit des Scripts nicht mehr ändern. Ein guter Stil wäre es beispielsweise, Variablen, die sich im Script nicht mehr ändern, großzuschreiben (wer hat hier was von C gesagt?) und normale Variablen klein. So kann man im Script schnell erkennen, welche Werte sich ohnehin nicht verändern und welche Werte variabel sind. Dies kann die Fehlersuche eingrenzen, weil konstante Variablen nicht verändert werden und man in diesen auch keine falschen Werte vermutet (es sei denn, man gibt diesen Variablen gleich zu Beginn einen falschen Wert).

Stil und Kommentare

Hier gibt es nur eines zu sagen: Es hat noch niemandem geschadet, einmal mehr auf ↵ und eventuell mal auf die (häufig noch wie neue) ⇆-Taste zu drücken. Wir empfehlen Ihnen, eine Anweisung bzw. ein Schlüsselwort pro Zeile zu verwenden. Dies hilft Ihnen, bei einer Fehlermeldung gleich die entsprechende Zeile anzuspringen und zu lokalisieren. Ebenfalls sollten Sie der Übersichtlichkeit zuliebe Anweisungs- oder Funktionsblöcke einrücken.

Das Problem mit einem kaum kommentierten Code kennt man zur Genüge. Jürgen hatte zum Beispiel früher sehr viel mit Perl zu tun. Nach einem Jahr Abstinenz (und vielen anderen Programmiersprachen) benötigte er nur ein paar Zeilen aus einem Script, das er mal geschrieben hatte. Leider hatte er den Code nicht kommentiert, und so gestaltete sich das »Entschlüsseln« etwas länger – Jürgen benötigte wieder den Rat anderer Personen. Durch die Verwendung von Kommentaren hätte er einiges an Zeit gespart. Bei der Shellscript-Programmierung ist das genauso. Sie werden schließlich noch etwas anderes zu tun haben, als immer nur Shellscripts zu schreiben.

Und: Es gibt auch noch das Backslash-Zeichen, von dem Sie immer dann Gebrauch machen sollten, wenn eine Zeile mal zu lang wird oder wenn Sie Ketten von Kommandos mit Pipes verwenden. Dies erhöht die Übersicht ebenfalls enorm.

10.2 Fehlerarten

Wenn das Script fertiggestellt ist und Fehler auftreten, unterscheidet man gewöhnlich zwischen mehreren Fehlerarten:

- **Syntaxfehler** – Der wohl am häufigsten auftretende Fehler ist ein Syntaxfehler. Hierbei reicht ein einfacher Tippfehler beispielsweise in einem Schlüsselwort aus. Wenn Sie einmal `thn` statt `then` schreiben, haben Sie schon einen Syntaxfehler. Leider lassen sich solche Fehler häufig nicht so einfach finden. Beispielsweise gibt bei uns ein Script, in dem `then` falsch geschrieben ist, folgende Fehlermeldung aus:

  ```
  you@host > ./script1
  ./script1: line 19: syntax error near unexpected token `fi'
  ./script1: line 19: `fi'
  ```

  Ein Syntaxfehler, der sehr häufig vorkommt, ist die Verwendung von ' anstelle eines ` bei einer Kommando-Substitution.

- **Logische Fehler** – Bei logischen Fehlern hat sich ein Denkfehler eingeschlichen. Das Script wird zwar ohne eine Fehlermeldung ausgeführt, führt aber nicht zum gewünschten Ergebnis. Ein einfaches Beispiel:

```
if [ ! -e $creatfile ]
then                      # Nein ...
   touch $creatfile       # Datei anlegen ...
   # Jetzt nochmals überprüfen ...
   if [ -e $creatfile ]
   then
      echo "Konnte $creatfile nicht anlegen"
      exit 1   # Script erfolglos beenden
   else
      echo "$creatfile erfolgreich angelegt"
   fi
fi
```

Sofern die Datei `$creatfile` nicht existiert, gibt das Script Folgendes aus:

```
you@host > ./createfile2
Konnte atestfile.txt nicht anlegen
```

Und das, obwohl die Datei angelegt wurde. Hier haben Sie es nicht – wie man häufig vermutet – mit mangelnden Rechten zu tun, sondern mit einem vergessenen Negationszeichen (`!`) in der zweiten `if`-Anweisung.

```
# Jetzt nochmals überprüfen ...
if [ ! -e $creatfile ]
```

Ein einfacher Fehler, der aber häufig nicht gleich gefunden wird.

- **Fehler der Shell oder der Kommandos** – Dies wird wohl der seltenste Fall eines Fehlers sein. Dabei handelt es sich um einen Fehler, der nicht Ihre Schuld ist, sondern die der Person, die Shell oder Kommando programmiert hat. Hier bleibt Ihnen nichts anderes übrig, als den Autor des Kommandos zu kontaktieren und ihm diesen Fehler mitzuteilen oder auf ein anderes Kommando auszuweichen.

10.3 Fehlersuche

Wenn der Fehler aufgetreten ist, bietet Ihnen die Shell einige Optionen, die Ihnen bei der Fehlersuche helfen. Aber egal welche Fehler denn nun aufgetreten sind, als Erstes sollten Sie die Fehlermeldung lesen und auch verstehen können. Das ist plausibel, aber leider werden immer wieder Fragen danach gestellt, warum dies oder jenes falsch läuft, obwohl die Antwort zum Teil schon eindeutig der Fehlermeldung zu entnehmen ist. Ganz klar im Vorteil sind Sie, wenn Sie das Buch durchgearbeitet und die Scripts abgetippt und ausprobiert haben. Durch »Trial and Error« haben Sie eine Menge Fehler produziert; Sie haben aber auch gelernt, wann welche Fehlermeldung ausgegeben wird.

10.3.1 Tracen mit set -x

Den Trace-Modus (*trace* = verfolgen, untersuchen), den man mit `set -x` setzt, haben Sie schon des Öfteren in diesem Buch eingesetzt, etwa als es darum ging, zu sehen, wie das Script bzw. die Befehle ablaufen. Die Option `-x` wurde bereits in Abschnitt 1.8.9, »Ein Shellscript beenden«, ausführlich behandelt. Trotzdem muss noch erwähnt werden, dass die Verwendung der Trace-Option nur in der aktuellen Shell aktiv ist. Nehmen wir an, Sie wollen wissen, was beim folgenden Script passiert:

```
# Name: areweroot

if [ $UID = 0 ]
then
   echo "Wir sind root!"
   renice -5 $$
else
   echo "Wir sind nicht root!"
   sudo renice -5 $$
fi
```

Wenn Sie das Script jetzt tracen wollen, genügt es nicht, einfach vor seiner Verwendung die Option `-x` zu setzen:

```
you@host > ./areweroot
+ ./areweroot
Wir sind nicht root!
Password:********
8967: Alte Priorität: 0, neue Priorität: -5
```

Das ist ein häufiges Missverständnis. Sie müssen die Option selbstverständlich an der entsprechenden Stelle (oder am Anfang des Scripts) setzen:

```
# Name: areweroot2

# Trace-Modus einschalten
set -x

if [ $UID = 0 ]
then
   echo "Wir sind root!"
   renice -5 $$
else
   echo "Wir sind nicht root!"
   sudo renice -5 $$
fi
```

Das Script bei der Ausführung:

```
you@host > ./areweroot2
+ ./areweroot
++ '[' 1000 = 0 ']'
++ echo 'Wir sind nicht root!'
Wir sind nicht root!
++ su -c 'renice -5 $$'
Password:*******
9050: Alte Priorität: 0, neue Priorität: -5
you@host > su
Password:********
# ./areweroot2
++ '[' 0 = 0 ']'
++ echo 'Wir sind root!'
Wir sind root!
++ renice -5 9070
9070: Alte Priorität: 0, neue Priorität: -5
```

Tracen mit Shell-Aufruf

Alternativ bietet sich hier auch die Möglichkeit, die Option `-x` an das Script mit einem Shell-Aufruf zu übergeben, z. B. `bash -x ./script` oder `ksh -x ./script`, wodurch sich eine Modifikation am Script vermeiden lässt. Oder Sie verwenden die Shebang-Zeile:

```
#!/usr/bin/bash -x
```

Häufig finden Sie mehrere Pluszeichen am Anfang einer Zeile. Dies zeigt an, wie tief die Verschachtelungsebene ist, in der die entsprechende Zeile ausgeführt wird. Jedes weitere Zeichen bedeutet »eine Ebene tiefer«. So verwendet beispielsweise folgendes Script drei Schachtelungsebenen:

```
# Name: datum

# Trace-Modus einschalten
set -x

datum=`date`
echo "Heute ist $datum"
```

Das Script bei der Ausführung:

```
you@host > ./datum
+./script1
+++ date
```



```
++ datum=Mi Mai  4 10:26:25 CEST 2016
++ echo 'Heute ist Mi Mai  4 10:26:25 CEST 2016'
Heute ist Mi Mai  4 10:26:25 CEST 2016
```

10.3.2 Das DEBUG- und das ERR-Signal

Für alle Shells gibt es mit dem `DEBUG`-Signal eine echte Debugging-Alternative. Das `ERR`-Signal hingegen ist nur der Korn-Shell vorbehalten. Angewendet werden diese Signale genauso, wie Sie dies von den Signalen her kennen. Sie richten sich hierbei einen Handler mit `trap` ein:

```
trap 'Kommando(s)' DEBUG
# nur für die Korn-Shell
trap 'Kommando(s)' ERR
```

Im folgenden Beispiel haben wir ein Script, das ständig in einer Endlosschleife läuft, aber wir sind zu blind, um den Fehler zu erkennen:

```
# Name: debug1

val=1

while [ "$val" -le 10 ]
do
   echo "Der ${val}. Schleifendurchlauf"
   i=`expr $val + 1`
done
```

Jetzt wollen wir in das Script mit `trap` einen »Entwanzer« einbauen:

```
# Name: debug1

trap 'printf "$LINENO :-> " ; read line ; eval $line' DEBUG

val=1

while [ "$val" -le 10 ]
do
   echo "Der ${val}. Schleifendurchlauf"
   i=`expr $val + 1`
done
```

Beim Entwanzen wird gewöhnlich das Kommando `eval` verwendet, mit dem Sie im Script Kommandos so ausführen können, als wären diese Teil des Scripts (siehe Abschnitt 9.1):

```
trap 'printf "$LINENO :-> " ; read line ; eval $line' DEBUG
```

Mit `DEBUG` wird nach jedem Ausdruck ein `DEBUG`-Signal gesendet, das Sie »trap(pen)« können, um eine bestimmte Aktion auszuführen. Im Beispiel wird zunächst die Zeilennummer des Scripts ausgegeben, gefolgt von einem Prompt. Anschließend können Sie einen Befehl einlesen und mit `eval` ausführen lassen (beispielsweise Variablen erhöhen oder reduzieren, Datei(en) anlegen, löschen, verändern, auflisten, überprüfen etc.). Das Script bei der Ausführung:

```
you@host > ./debug1
5 :->↵
7 :->↵
9 :->↵
Der 1. Schleifendurchlauf
10 :-> echo $val
1
7 :->↵
9 :->↵
Der 1. Schleifendurchlauf
10 :-> echo $val
1
7 :->↵
9 :->↵
Der 1. Schleifendurchlauf
10 :-> val=`expr $val + 1`
7 :-> echo $val
2
9 :->↵
Der 2. Schleifendurchlauf
10 :->↵
7 :->↵
9 :->↵
Der 2. Schleifendurchlauf
10 :-> exit
```

Der Fehler ist gefunden: Die Variable `val` wurde nicht hochgezählt, und ein Blick auf die Zeile 10 zeigt Folgendes:

```
i=`expr $val + 1`
```


Hier haben wir `i` statt `val` verwendet. Etwas störend ist allerdings, dass nur die Zeilennummer ausgegeben wird. Bei längeren Scripts ist es schwer bzw. umständlich, die Zeilennummer parallel zum Debuggen zu behandeln. Daher ist es sinnvoll, wenn auch hier die entsprechende Zeile mit ausgegeben wird, die ausgeführt wird bzw. wurde. Dies ist im Grunde kein Problem, da die `DEBUG`-Signale in allen drei Shells vorhanden sind, weshalb auch gleich das komplette Script in ein Array eingelesen werden kann. Für den Index nutzen Sie wieder die Variable `LINENO`. Die Zeile `eval` zum Ausführen von Kommandos packen Sie einfach in eine separate Funktion, die beliebig viele Befehle aufnehmen kann, bis eben ↵ gedrückt wird.

```
# Name: debug2

# ------- DEBUG Anfang --------- #

# Die übliche eval-Funktion
debugging() {
   printf "STOP > "
   while true
   do
      read line
      [ "$line" = "" ] && break
      eval $line
      printf " > "
   done
}

typeset -i index=1

# Das komplette Script in ein Array einlesen
while read zeile[$index]
do
   index=index+1
done<$0

trap 'echo "${zeile[$LINENO]}" ; debugging' DEBUG

# ------- DEBUG Ende   --------- #

typeset -i val=1

while (( $val <= 10 ))
do
```

```
   echo "Der ${val}. Schleifendurchlauf"
   val=val+1
done
```

Das Script bei der Ausführung:

```
you@host > ./debug2
typeset -i val=1
STOP >↵
while (( $val <= 10 ))
STOP > echo $val
1
 > val=7
 > echo $val
7
 >↵
echo "Der $val Schleifendurchlauf"
STOP >↵
Der 7. Schleifendurchlauf
val=val+1
STOP >↵
while (( $val <= 10 ))
STOP >↵
echo "Der $val Schleifendurchlauf"
STOP >↵
Der 8. Schleifendurchlauf
val=val+1
STOP >↵
while (( $val <= 10 ))
STOP >↵
echo "Der $val Schleifendurchlauf"
STOP >↵
Der 9. Schleifendurchlauf
val=val+1
STOP >exit
you@host >
```

Shells ohne DEBUG-Signal

Auch in der Bourne-Shell oder in anderen Shells, die eben nicht das DEBUG-Signal unterstützen, können Sie die Funktion `debugging()` hinzufügen. Nur müssen Sie diese Funktion mehrmals hinter oder/und vor den Zeilen einsetzen, von denen Sie vermuten, dass das Script nicht richtig funktioniert. Natürlich sollten Sie es nicht versäumen, sie aus Ihrem fertigen Script wieder zu entfernen.

In der Korn-Shell finden Sie auch das Signal `ERR`, das Sie ebenfalls mit `trap` einfangen können. Allerdings handelt es sich hierbei eher um ein Pseudo-Signal, denn das Signal bezieht sich immer auf den Rückgabewert eines Kommandos. Ist dieser ungleich 0, wird das Signal `ERR` gesendet. Allerdings lässt sich dies nicht dort verwenden, wo bereits der Exit-Code eines Kommandos abgefragt wird (beispielsweise `if`, `while` ...). Auch hierzu zeigen wir Ihnen ein simples Script, das das Signal `ERR` abfängt:

```
# Name: debugERR

error_handling() {
   echo "Fehler: $ERRNO Zeile: $LINENO"
   printf "Beenden (j/n) : " ; read
   [ "$REPLY" = "j" ] && exit 1
}

trap 'error_handling' ERR

echo "Testen des ERR-Signals"
# Sollte dem normalen Benutzer untersagt sein
cat > /etc/profile
echo "Nach dem Testen des ERR-Signals"
```

Das Script bei der Ausführung:

```
you@host > ksh ./debugERR
Testen des ERR-Signals
Fehler: Permission denied  Zeile: 4
Beenden (j/n) : j
```

Das Signal ERR in der Bash

Zwar wird die Bash beim Signal `ERR` in den Dokumentationen nicht erwähnt, aber beim Testen hat es auch unter der Bash funktioniert, nur dass es eben in der Bash nicht die Variable `ERRNO` gibt.

10.3.3 Variablen und Syntax überprüfen

Um wirklich sicherzugehen, dass Sie nicht auf eine nicht gesetzte Variable zugreifen, können Sie `set` mit der Option `-u` verwenden. Wird hierbei auf eine nicht definierte Variable zugegriffen, wird eine entsprechende Fehlermeldung ausgegeben. Mit `+u` schalten Sie diese Option wieder ab.

```
# Name: aunboundvar

# Keine undefinierten Variablen zulassen
set -u

var1=100
echo $var1 $var2
```

Das Script bei der Ausführung:

```
you@host > ./aunboundvar
./aunboundvar: line 7: var2: unbound variable
```

Wollen Sie ein Script nicht ausführen, sondern nur dessen Syntax überprüfen lassen, können Sie die Option `-n` verwenden. Mit `+n` schalten Sie diese Option wieder aus.

10.3.4 Eine Debug-Ausgabe hinzufügen

Die primitivste Form aller Debugging-Techniken ist gleichzeitig wohl die am meisten eingesetzte Variante – nicht nur in der Shell-Programmierung. Sie setzen überall dort, wo Sie einen Fehler vermuten, eine `echo`-Ausgabe über den Zustand der Daten ein (beispielsweise ermitteln Sie, welchen Wert eine Variable hat). Gibt das Script sehr viel auf dem Bildschirm aus, kann man auch einfach hier und da mal ein einfaches `read` einbauen, wodurch auf einen ↵-Tastendruck gewartet wird, ehe die Ausführung des Scripts weiter fortfährt, oder man kann auch die eine oder andere störende Ausgabe kurzzeitig ins Datengrab */dev/null* schicken. Ein einfaches Beispiel:

```
# Name: maskieren

nobody() {
   echo "Die Ausgabe wollen wir nicht!!!"
}

echo "Ausgabe1"
exec 1>/dev/null
nobody
exec 1>`tty`
echo "Ausgabe2"
```

Hier interessieren wir uns nicht für die Ausgabe der Funktion `nobody` und schicken die Standardausgabe ins Datengrab. Natürlich müssen Sie das Ganze wieder rückgängig machen. Wir erreichen das mit einer Umleitung auf das aktuelle Terminal (das Kommando `tty` übernimmt das für uns).

10.3.5 Debugging-Tools

Wenn Sie ein fertiges Debugging-Tool suchen, dann seien Ihnen für die Bash der Debugger `bashdb` (*http://bashdb.sourceforge.net*) und für die Korn-Shell `kshdb` ans Herz gelegt. Der Bash-Debugger ist nichts anderes als eine gepatchte Version der Bash, die ein besseres Debugging ebenso wie eine verbesserte Fehlerausgabe unterstützt.

Für die Z-Shell gibt es ebenfalls einen Debugger, nur ist er nicht Bestandteil der Z-Shell. Sie müssen sich den `zshdb` aus den Git-Quellen selbst bauen. Wenn Sie den `zshdb` nutzen wollen, finden Sie Informationen unter *https://github.com/rocky/zshdb*.

Auch für Entwicklungswerkzeuge wie Visual Studio Code und Atom gibt es integrierte Debugger als Zusatzmodule. Bei sehr umfangreichen Scripts ist es ratsam, auf die Best Practices der Softwareentwicklung zurückzugreifen und die entsprechenden Werkzeuge zu benutzen.

10.4 Fehlerbehandlung

Auch wenn Sie Ihre Anwendung geplant und getestet haben, werden Sie doch feststellen, dass man vor Fehlern nie ganz geschützt ist. Nicht immer liegt es, wenn es zu Fehlern kommt, an der Person, die programmiert hat. Eventuell wurde ein Verzeichnis umbenannt oder gelöscht, die Datenbank für das Backup existiert nicht mehr, oder Ihr Script muss auf einem System mit einer anderen Shell laufen: Es lauern viele Unwegsamkeiten im System. Sie sollten versuchen, solche Probleme bereits beim Entwickeln zu berücksichtigen und abzufangen, damit es nicht zu ungewollten Ergebnissen kommt.

Mit dem Kommando `test` aus Abschnitt 4 können Sie den Status von Dateien und Verzeichnissen sowie die entsprechenden Zugriffsrechte abfragen. Damit ist es möglich, Fehler, die bei dem weiteren Einsatz des Shellscripts eintreten, abzufangen und entsprechende Maßnahme einzuleiten.

Für den Fall, dass es doch zu einem Fehler kommt, sollten Sie eine Notbremse einbauen. Das gilt besonders für Shellscripts, die im Hintergrund laufen.

Angenommen, Sie hätten ein Shellscript, das täglich bestimmte Dateien kopiert, mit dem Jahr und dem Tag des Jahres nummeriert (Sicherung/Versionierung) und anschließend löscht.

```
# kopiere Datei
cp ~/Datenbank/dummy.db ~/Sicherung/dummy.db-$(date '+%Y-%j')
# löschen Datei
rm ~/Datenbank/dummy.db
```

Die Sicherung der Daten würde nicht klappen, da der Zielordner nicht (mehr) existiert. Das Shellscript läuft aber weiter und löscht die Datei. Es würden also keine Sicherungen der Datei mehr vorhanden sein.

Nutzen Sie daher unbedingt den Befehl `set -e`, der bewirkt, dass ein Script sofort beendet wird, wenn ein Fehler auftritt.

```
set -e
# kopiere Datei
cp ~/Datenbank/dummy.db ~/Sicherung/dummy.db-$(date '+%Y-%j')
# löschen Datei
rm ~/Datenbank/dummy.db
```

In diesem Beispiel wäre es natürlich besser, bereits vorab zu prüfen, ob das Verzeichnis und die Quelldatei existieren.

```
# Demonstriert sicherung/kopieren einer Datei
# afilesave
set -e

file=dummy.db                          # zu sichernde Datei
qdir=~/Datenbank                       # Quellverzeichnis
zdir=~/Sicherung                       # Zielverzeichnis

if [ -f $qdir/$file ]                  # existiert die Datei dummy1
then                                   # Ja!
   if [ -d $zdir ]                     # gibt es das Zielverzeichnis?
   then
      cp $qdir/$file $zdir/$file-$(date '+%Y-%j') # kopiere die Datei
      rm $qdir/$file                   # lösche Datei
   else
      echo "Das Zielverzeichnis $zdir ist nicht vorhanden"
      # weitere Maßnahmen
   fi
  else
  echo "$qdir/$file existiert nicht"
  # weitere Maßnahmen
fi
```

Das Zielverzeichnis könnte aber im Netzwerk liegen, und die Verbindung wird unterbrochen – in diesen Fall greift dann `set -e`.

Kapitel 11
Reguläre Ausdrücke und grep

Die Verwendung von regulären Ausdrücken und grep zählt zum Grundwissen eines jeden Linux/UNIX-Anwenders und einer jeden -Anwenderin. In der Systemadministration ist sie sowieso unerlässlich, denn es gibt kein vernünftiges System, in dem reguläre Ausdrücke nicht vorkommen. Eine kurze Einführung zu den regulären Ausdrücken wie auch zum Tool grep (und seinen Nachkommen wie beispielsweise egrep und fgrep) erscheint uns daher notwendig.

11.1 Reguläre Ausdrücke – die Theorie

Reguläre Ausdrücke (engl. *regular expressions*) sind eine leistungsfähige formale Sprache, mit der sich eine bestimmte (Unter-)Menge von Zeichenketten beschreiben lässt. Es muss allerdings gleich erwähnt werden, dass reguläre Ausdrücke kein Tool und keine Sammlung von Funktionen sind, die von einem Betriebssystem abhängig sind; stattdessen handelt es sich in der Tat um eine echte Sprache mit einer formalen Grammatik, in der jeder Ausdruck eine präzise Bedeutung hat.

Reguläre Ausdrücke werden von sehr vielen Texteditoren und Programmen eingesetzt. Meistens verwendet man sie, um bestimmte Muster zu suchen und diese dann durch etwas anderes zu ersetzen. In der Linux/UNIX-Welt werden reguläre Ausdrücke vorwiegend in Programmen wie *grep*, *sed* und *awk* oder den Texteditoren *vi* und *Emacs* verwendet. Aber auch viele Programmiersprachen, unter anderem Perl, Java, Python, Tcl, PHP oder Ruby, bieten reguläre Ausdrücke an.

Die Entstehungsgeschichte der regulären Ausdrücke ist schnell erzählt. Den Grundstein hat ein Mathematiker und Logiker, Stephen Kleene, gelegt. Er gilt übrigens auch als Mitbegründer der theoretischen Informatik, besonders der hier behandelten formalen Sprachen und der Automatentheorie. Stephen Kleene verwendete eine Notation, die er selbst *reguläre Menge* nannte. Später verwendete dann Ken Thompson (der Miterfinder der Programmiersprache C) diese Notationen für eine Vorgängerversion des UNIX-Editors *ed* und für das Werkzeug *grep*. Nach der Fertigstellung von *grep* wurden die regulären Ausdrücke in sehr vielen Programmen implementiert. Viele davon benutzen die mittlerweile sehr bekannte Bibliothek `regex` von Henry Spencer.

Die Grenzen von grep

Sofern Sie Erweiterungen wie Rückwärtsreferenzen verwenden wollen, sei Ihnen Perl empfohlen, weil *grep* hier an seine Leistungsgrenzen kommt. Inzwischen unterscheidet man übrigens verschiedene *regexes* (POSIX-RE, Extended-RE und *pcre*). Die Unterschiede sind in den Manuals *regex* und *perlre* zu finden. Ein Großteil der Scriptsprachen und Programme stützt sich auf die Variante *pcre* (Perl Compatible Regular Expressions), die mittlerweile als die leistungsfähigste gilt. Mit der Option `-P` können Sie die erweiterten Features der *pcre* auch mit *grep* nutzen.

11.1.1 Elemente für reguläre Ausdrücke (POSIX-RE)

Vorwiegend werden reguläre Ausdrücke dazu verwendet, bestimmte Zeichenketten in einer Menge von Zeichen zu suchen und zu finden. Die nun folgende Beschreibung ist eine sehr häufig verwendete Konvention, die von fast allen Programmen, die reguläre Ausdrücke verwenden, so eingesetzt wird. Gewöhnlich wird dabei ein regulärer Ausdruck aus den Zeichen des Alphabets in Kombination mit den Metazeichen gebildet. (Die Metazeichen werden hier gleich vorgestellt.).

Zeichenliterale

Als Zeichenliterale bezeichnet man die Zeichen, die wörtlich übereinstimmen müssen. Diese werden im regulären Ausdruck direkt (als Wort) notiert. Hierbei besteht je nach System auch die Möglichkeit, alles in hexadezimaler oder oktaler Form anzugeben.

Beliebiges Zeichen

Für ein einzelnes beliebiges Zeichen verwendet man einen Punkt. Dieser Punkt kann dann für fast beliebige Zeichen stehen.

Zeichenauswahl

Die Zeichenauswahl mit den eckigen Klammern `[auswahl]` kennen Sie ebenfalls bereits von der Shell (siehe Abschnitt 1.10.6, »Ersatzmuster (Wildcards)«). Alles, was Sie in die eckigen Klammern schreiben, gilt dann exakt für ein Zeichen aus dieser Auswahl. Beispielsweise steht `[axz]` für eines der Zeichen »a«, »x« oder »z«. Dies lässt sich natürlich auch in Bereiche aufteilen. So besteht bei der Angabe von `[2-7]` der Bereich aus den Ziffern 2 bis 7. Mit dem Zeichen `^` innerhalb der Zeichenauswahl können Sie auch Zeichen ausschließen. Beispielsweise schließen Sie mit `[^a-f]` die Zeichen »a«, »b«, »c«, »d«, »e« und »f« aus.

Vordefinierte Zeichenklassen

Manche Implementationen von regulären Ausdrücken bieten auch vordefinierte Zeichenklassen an. Sofern Sie keine solch vordefinierten Zeichenklassen finden, können Sie die Zeichenauswahl auch selbst in eckigen Klammern beschreiben. Die vordefinierten Zeichenklassen sind letztendlich auch nur eine Kurzform der Zeichenklassen. Tabelle 11.1 führt einige bekannte vordefinierte Zeichenklassen auf.

Vordefiniert	Bedeutung	Selbst definiert
`\d`	eine Zahl	`[0-9]`
`\D`	keine Zahl	`[^0-9]`
`\w`	ein Buchstabe, eine Zahl oder der Unterstrich	`[a-zA-Z_0-9]`
`\W`	kein Buchstabe, keine Zahl und kein Unterstrich	`[^a-zA-Z_0-9]`
`\s`	Whitespace-Zeichen	`[ \f\n\r\t\v]`
`\S`	alle Zeichen außer Whitespace-Zeichen	`[^\f\n\r\t\v]`

Tabelle 11.1 Vordefinierte Zeichenklassen

Quantifizierer

Als Quantifizierer bzw. Quantoren bezeichnet man Elemente, die es erlauben, den vorherigen Ausdruck in unterschiedlicher Vielfalt in einer Zeichenkette zuzulassen (siehe Tabelle 11.2).

Quantifizierer	Bedeutung
`?`	Der Ausdruck, der voransteht, ist optional, d. h., er kann einmal vorkommen, muss aber nicht. Der Ausdruck kommt also entweder null- oder einmal vor.
`+`	Der Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrmals vorhanden sein.
`*`	Der Ausdruck darf beliebig oft oder auch gar nicht vorkommen.
`{min,}`	Der voranstehende Ausdruck muss mindestens *min*-mal vorkommen.
`{min,max}`	Der voranstehende Ausdruck muss mindestens *min*-mal, darf aber nicht mehr als *max*-mal vorkommen.
`{n}`	Der voranstehende Ausdruck muss genau *n*-mal vorkommen.

Tabelle 11.2 Quantifizierer

Gruppierung

Ausdrücke können auch zwischen runden Klammern gruppiert werden. Einige Tools speichern diese Gruppierung ab und ermöglichen so eine Wiederverwendung im regulären Ausdruck bzw. bei der Textersetzung über `\1`. Es lassen sich hiermit bis zu neun Muster abspeichern (`\1`, `\2` ... `\9`). Beispielsweise würde man mit

```
s/\(string1\) \(string2\) \(string3\)/\3 \2 \1/g
```

erreichen, dass in einer Textdatei alle Vorkommen von

```
string1 string2 string3
```

umgeändert werden in:

```
string3 string2 string1
```

`\1` bezieht sich also immer auf das erste Klammernpaar, `\2` auf das zweite usw.

Alternativen

Selbstverständlich lassen sich auch Alternativen definieren. Hierfür wird das Zeichen `|` verwendet. Beispielsweise bedeutet

```
(asdf|ASDF)
```

dass nach »asdf« oder »ASDF« gesucht wird, nicht aber nach »AsDf« oder »asdF«.

Sonderzeichen

Da viele Tools direkt auf Textdateien zugreifen, sind gewöhnlich noch die in Tabelle 11.3 aufgelisteten Sonderzeichen definiert.

Sonderzeichen	Bedeutung
`^`	Steht für den Zeilenanfang.
`$`	Steht für das Zeilenende.
`\b`	Steht für die leere Zeichenkette am Wortanfang oder am Wortende.
`\B`	Steht für die leere Zeichenkette, die nicht den Anfang oder das Ende eines Worts bildet.
`\<`	Steht für die leere Zeichenkette am Wortanfang.
`\>`	Steht für die leere Zeichenkette am Wortende.

Tabelle 11.3 Sonderzeichen bei regulären Ausdrücken

Sonderzeichen	Bedeutung
`\d`	Ziffer
`\D`	keine Ziffer
`\s`	Whitespace
`\S`	kein Whitespace
`.`	Zeichen
`+`	voriger Ausdruck mindestens einmal
`*`	voriger Ausdruck beliebig oft
`?`	voriger Ausdruck null- oder einmal

Tabelle 11.3 Sonderzeichen bei regulären Ausdrücken (Forts.)

Jedes dieser Metazeichen lässt sich auch mit dem Backslash (\) maskieren.

11.1.2 Zusammenfassung

Grau ist alle Theorie, und trotzdem ließe sich zu den regulären Ausdrücken noch viel mehr schreiben. Damit das hier Beschriebene für Sie kein Buch mit sieben Siegeln bleibt, greifen wir im nächsten Abschnitt mit `grep` darauf zurück. Auch in den Kapiteln zu `sed` und `awk` hilft Ihnen das Wissen über reguläre Ausdrücke weiter.

Mehr zu den oben aufgeführten regulären Ausdrücken finden Sie im Internet unter der Adresse *https://www.lrz.de/services/schulung/unterlagen/regul/*.

11.2 grep

Um in Dateien nach bestimmten Mustern zu suchen, wird häufig `grep` verwendet. `grep` steht für »**G**lobal search for a **R**egular **E**xpression and **P**rint out matched lines«. Es gibt mittlerweile mehrere verschiedene solcher `grep`-Kommandos:

- `grep` – der Klassiker.
- `egrep` – Kurzform für »Extended Grep«, also ein erweitertes `grep`, das im Gegensatz zu `grep` noch mehr reguläre Ausdrücke versteht.
- `fgrep` – Kurzform für »Fixed Grep« (oder gern auch »Faster Grep«). `fgrep` versteht weniger reguläre Ausdrücke als `grep` und ist dadurch erheblich schneller bei der Suche in großen Dateien als `grep` oder `egrep`.

- rgrep – Kurzform für »Rekursive Grep«. rgrep wird für die rekursive Suche in Unterverzeichnissen verwendet. Es muss allerdings erwähnt werden, dass rgrep die Unterverzeichnisse auch komplett durchläuft.
- agrep – Kurzform für »Approximatives Grep«. Damit ist es möglich, eine unscharfe Suche durchzuführen. agrep findet auch Beinahe-Treffer, sofern die Unterschiede unter einer vorgegebenen Schwelle liegen

11.2.1 Wie arbeitet grep?

Das grep-Kommando sucht nach einem Muster von Zeichen in einer oder mehreren Datei(en). Enthält das Muster *Whitespace*, muss es entsprechend gequotet werden. Das Muster ist also entweder eine gequotete Zeichenkette oder ein einfaches Wort. Alle anderen Wörter hinter dem Muster werden von grep dann als Datei(en) verwendet, in denen nach dem Muster gesucht wird. Die Ausgabe sendet grep an die Standardausgabe (meistens ist das der Bildschirm). grep nimmt auch keinerlei Änderung an der Eingabedatei vor. Die Syntax lautet:

```
grep wort datei1 [datei2] ... [datein]
```

Ein viel zitiertes Beispiel ist:

```
you@host > grep john /etc/passwd
john:x:1002:100:Jonathan Wolf:/home/john:/bin/csh
```

grep sucht hier nach dem Muster »john« in der Datei */etc/passwd*. Bei Erfolg wird die entsprechende Zeile auf dem Bildschirm ausgegeben. Wird das Muster nicht gefunden, gibt es keine Ausgabe und auch keine Fehlermeldung. Existiert die Datei nicht, wird eine Fehlermeldung auf dem Bildschirm ausgegeben.

Als Rückgabewert von grep erhalten Sie bei einer erfolgreichen Suche den Wert 0. Wird ein Muster nicht gefunden, gibt grep 1 als Exit-Code zurück, und wird die Datei nicht gefunden, wird 2 zurückgegeben. Der Rückgabewert von grep ist in den Shellscripts häufig von Bedeutung, da Sie relativ selten die Ausgaben auf dem Bildschirm machen werden. Und vom Exit-Code hängt es häufig auch ab, wie Ihr Script weiterlaufen soll.

grep gibt den Exit-Code 0 zurück, also wurde ein Muster gefunden:

```
you@host > grep you /etc/passwd > /dev/null
you@host > echo $?
0
```

grep gibt den Exit-Code 1 zurück, somit wurde kein übereinstimmendes Muster gefunden:


```
you@host > grep gibtsnicht /etc/passwd > /dev/null
you@host > echo $?
1
```

grep gibt den Exit-Code 2 zurück – die Datei scheint nicht zu existieren (oder wurde, wie hier, falsch geschrieben):

```
you@host > grep you /etc/PASSwd > /dev/null 2>&1
you@host > echo $?
2
```

grep kann seine Eingabe neben Dateien auch von der Standardeingabe oder einer Pipe erhalten:

```
you@host > grep echo < script1
echo "Ausgabe1"
echo "Ausgabe2"
you@host > cat script1 | grep echo
echo "Ausgabe1"
echo "Ausgabe2"
```

Auch grep kennt eine ganze Menge regulärer Ausdrücke. Metazeichen helfen Ihnen dabei, sich mit grep ein Suchmuster zu erstellen. Und natürlich unterstützt auch grep viele Optionen, die Sie dem Kommando mitgeben können. Auf einige dieser Features gehen wir in den folgenden Abschnitten ein.

11.2.2 grep mit regulären Ausdrücken

Dass grep eines der ältesten Programme ist, das reguläre Ausdrücke kennt, haben wir bereits gesagt. Welche das sind, wird in Tabelle 11.4 aufgelistet. Allerdings kennt grep nicht alle regulären Ausdrücke, weshalb es außerdem das Kommando egrep gibt, das noch einige mehr versteht (siehe Tabelle 11.5).

Zeichen	Funktion	Beispiel	Bedeutung
^	Anfang der Zeile	'^wort'	Gibt alle Zeilen aus, die mit »wort« beginnen.
$	Ende der Zeile	'wort$'	Gibt alle Zeilen aus, die mit »wort« enden.
^$	Komplette Zeile	'^wort$'	Gibt alle vollständigen Zeilen mit dem Muster »wort« aus.

Tabelle 11.4 Reguläre Ausdrücke von »grep«

Zeichen	Funktion	Beispiel	Bedeutung
.	Beliebiges Zeichen	'w.rt'	Gibt alle Zeilen aus, die ein »w«, einen beliebigen Buchstaben und »rt« enthalten (beispielsweise »wort«, »wert«, »wirt«, »wart«).
*	Beliebig oft	'wort*'	Gibt alle Zeilen aus mit beliebig vielen (oder auch gar keinen) Vorkommen des vorangegangenen Zeichens.
.*	Beliebig viele	'wort.*wort'	Die Kombination .* steht für beliebig viele Zeichen.
[]	Ein Zeichen aus dem Bereich	'[Ww]ort'	Gibt alle Zeilen aus, die Zeichen im angegebenen Bereich (im Beispiel »Wort« oder »wort«) enthalten.
[^]	Kein Zeichen aus dem Bereich	'[^A-VX-Za-z]ort'	Die Zeichen, die im angegebenen Bereich stehen, werden nicht beachtet. (Im Beispiel kann »Wort« gefunden werden, nicht aber »Tort« oder »Sort« und auch nicht »wort«.)
\<	Anfang eines Worts	'\<wort'	Findet hier alles, was mit »wort« beginnt (beispielsweise »wort«, »wortreich«, aber nicht »Vorwort« oder »Nachwort«).
\>	Ende eines Worts	'wort\>'	Findet alle Zeilen, die mit »wort« enden (beispielsweise »Vorwort« oder »Nachwort«, nicht aber »wort« oder »wortreich«).
\<\>	Ein Wort	'\<wort\>'	Findet exakt »wort« und nicht »Nachwort« oder »wortreich«.
\(...\)	Backreferenz	'\(wort\)'	Merkt sich die eingeschlossenen Muster vor, um darauf später über \1 zuzugreifen. Bis zu neun Muster können auf diese Weise gespeichert werden.
x\{m\}	Exakte Wiederholung des Zeichens	x\{3\}	Exakt 3-maliges Auftreten des Zeichens »x«.

Tabelle 11.4 Reguläre Ausdrücke von »grep« (Forts.)

Zeichen	Funktion	Beispiel	Bedeutung
x\{m,\}	Mindestens Wiederholung des Zeichens	x\{3,\}	Mindestens ein 3-maliges Auftreten des Zeichens »x«.
x\{m,n\}	Mindeste bis maximale Wiederholung des Zeichens	x\{3,6\}	Mindestens ein 3-maliges Auftreten des Zeichens »x« bis maximal 6-maliges Auftreten (nicht mehr).

Tabelle 11.4 Reguläre Ausdrücke von »grep« (Forts.)

Zusätzlich zu diesen regulären Ausdrücken kennt egrep noch folgende:

Zeichen	Funktion	Beispiel	Bedeutung
+	Mindestens einmal	'wort[0-9]+'	Es muss mindestens eine Ziffer aus dem Bereich vorkommen.
?	Null- oder einmal	'wort[0-9]?'	Eine Ziffer aus dem Bereich darf, muss aber nicht vorkommen.
|	Alternativen	'worta|wortb'	Das Wort »worta« oder »wortb«.

Tabelle 11.5 Weitere reguläre Ausdrücke von »egrep«

Hier folgen jetzt einige Beispiele zu den regulären Ausdrücken mit grep. Wir nutzen dazu die Datei mit dem folgenden Muster:

```
you@host > cat mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Einfachstes Beispiel:

```
you@host > grep Libanon mrolympia.dat
Samir Bannout Libanon 1983
```

Hierbei werden alle Zeilen ausgegeben, die den regulären Ausdruck »Libanon« in der Datei *mrolympia.dat* enthalten.

Nächstes Beispiel:

```
you@host > grep '^S' mrolympia.dat
Sergio Oliva USA 1967 1968 1969
Samir Bannout Libanon 1983
```

Hiermit werden alle Zeilen ausgegeben, die mit dem Zeichen »S« beginnen. Das Caret-Zeichen (^) steht immer für den Anfang einer Zeile.

Nächstes Beispiel:

```
you@host > grep '1$' mrolympia.dat
Franco Columbu Argentinien 1976 1981
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
```

Es werden alle Zeilen ausgegeben, die mit dem Zeichen »1« enden. Das Dollarzeichen steht hierbei für das Ende einer Zeile.

Nächstes Beispiel:

```
you@host > grep Sergio Yates mrolympia.dat
grep: Yates: Datei oder Verzeichnis nicht gefunden
mrolympia.dat:Sergio Oliva USA 1967 1968 1969
```

Hier wurde ein Fehler gemacht, da `grep` das dritte Argument, den Namen »Yates«, bereits als eine Dateiangabe behandelt, in der nach einem Muster gesucht wird. Einen Namen wie »Sergio Yates« gibt es nämlich nicht in dieser Datei. Damit das Muster komplett zum Vergleich für `grep` verwendet wird, müssen Sie es zwischen Single Quotes stellen:

```
you@host > grep 'Sergio Yates' mrolympia.dat
you@host > echo $?
1
```

Das nächste Beispiel:

```
you@host > grep '197.' mrolympia.dat
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
```

Wer war in den 1970er-Jahren am besten? Damit geben Sie alle Zeilen aus, in denen sich das Muster »197« und ein weiteres beliebiges einzelnes Zeichen befindet. Sofern Sie wirklich nach einem Punkt suchen, müssen Sie einen Backslash davor setzen. Dies gilt übrigens für alle Metazeichen.

Nächstes Beispiel:

```
you@host > grep '^[AS]' mrolympia.dat
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Samir Bannout Libanon 1983
```

Hier wird jede Zeile ausgegeben, die jeweils mit dem Zeichen »A« oder »S« beginnt.

Nächstes Beispiel:

```
you@host > grep '^[^AS]' mrolympia.dat
Larry Scott USA 1965 1966
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Jetzt werden alle Zeilen ausgegeben, die nicht (`[^AS]`) mit dem Zeichen »A« oder »S« beginnen.

Nächstes Beispiel:

```
you@host > grep '^S.*Libanon' mrolympia.dat
Samir Bannout Libanon 1983
```

Hier liefert `grep` die Zeile zurück, die mit einem Zeichen »S« beginnt, gefolgt von beliebig vielen Zeichen, und die die Zeichenfolge »Libanon« enthält.

Nächstes Beispiel:

```
you@host > grep '^S.*196.' mrolympia.dat
Sergio Oliva USA 1967 1968 1969
```

Ähnlich wie im Beispiel zuvor werden die Zeilen ausgegeben, die mit dem Zeichen »S« beginnen und die Textfolge »196« mit einem beliebigen weiteren Zeichen enthalten.

Nächstes Beispiel:

```
you@host > grep '[a-z]\{14\}' mrolympia.dat
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
```

Gibt alle Zeilen aus, in denen 14 Buchstaben hintereinander Kleinbuchstaben sind.

Nächstes Beispiel:

```
you@host > grep '\<Col' mrolympia.dat
Franco Columbu Argentinien 1976 1981
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Hier werden alle Zeilen ausgegeben, in denen sich ein Wort befindet, das mit »Col« beginnt.

Nächstes Beispiel:

```
you@host > grep 'A\>' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Chris Dickerson USA 1982
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Das Gegenteil vom Beispiel zuvor: Hier werden alle Zeilen ausgegeben, in denen sich ein Wort befindet, das mit »A« endet.

Nächstes Beispiel:

```
you@host > grep '\<Coleman\>' mrolympia.dat
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Hierbei wird nach einem vollständigen Wort »Coleman« gesucht, also nicht nach »AColeman« und auch nicht nach »Colemann«.

Nächstes Beispiel:

```
you@host > grep '\<.*ien.*\>' mrolympia.dat
Franco Columbu Argentinien 1976 1981
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
```

Hier werden alle Zeilen ausgegeben, die ein Wort mit der Zeichenfolge »ien« enthalten. Davor und danach können sich beliebig viele Zeichen befinden.

Nächstes Beispiel:

```
you@host > grep '\<[G].*ien\>' mrolympia.dat
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
```

Hier wird nach einem Wort gesucht, das mit dem Großbuchstaben »G« beginnt und mit der Zeichenfolge »ien« endet. Dazwischen können sich beliebig viele Zeichen befinden.

Natürlich können Sie grep auch dazu verwenden, in einem ganzen Verzeichnis die Dateien nach einem bestimmten Muster abzusuchen. Sie können wieder das Metazeichen * als Platzhalter für alle Dateien im Verzeichnis einsetzen:


```
you@host > grep 'echo' *
...
```

Hier werden im aktuellen Arbeitsverzeichnis alle Dateien nach dem Muster »echo« durchsucht.

11.2.3 grep mit Pipes

Häufig wird grep in Verbindung mit einer Pipe verwendet. Hierbei übergeben Sie grep dann statt einer Datei als drittes Argument für die Eingabe die Daten durch eine Pipe von der Standardausgabe eines anderen Kommandos. Beispielsweise bekommen Sie mit

```
you@host > ps -ef | grep $USER
```

alle Prozesse aufgelistet, deren Eigentümer der aktuelle User ist bzw. die dieser gestartet hat. Dabei gibt ps seine Ausgabe durch die Pipe an die Standardeingabe von grep. grep wiederum sucht dann in der entsprechenden Ausgabe nach dem passenden Muster und gibt gegebenenfalls die Zeile(n) aus, die mit dem Muster übereinstimmen. Natürlich können Sie hierbei auch, wie schon gehabt, die regulären Ausdrücke verwenden. Die Syntax lautet:

```
kommando | grep muster
```

Ebenfalls relativ häufig wird grep mit ls zur Suche bestimmter Dateien verwendet:

```
you@host > ls | grep '^scr.*'
script1
script1~
script2
script2~
```

Im Abschnitt zuvor haben Sie mit

```
you@host > grep 'echo' *
```

alle Dateien nach dem Muster »echo« durchsucht. Meistens – uns ging es zumindest so – haben Sie neben einfachen Scripts auch eine Menge Dokumentationen im Verzeichnis herumliegen. Wollen Sie jetzt auch noch die Dateinamen mithilfe regulärer Ausdrücke eingrenzen, können Sie die Ausgabe von grep an die Eingabe eines weiteren grep-Aufrufs hängen:

```
you@host > grep 'echo' * | grep 'scrip.*'
...
```

Jetzt wird in allen Dateien des aktuellen Verzeichnisses nach dem Muster »echo« gesucht (erstes grep), und anschließend (zweites grep) werden nur die Dateien berücksichtigt, die mit der Zeichenfolge »scrip« beginnen, gefolgt von beliebig vielen weiteren Zeichen.

11.2.4 grep mit Optionen

Natürlich bietet grep Ihnen neben den regulären Ausdrücken auch noch eine Menge weiterer Optionen an, mit denen Sie das Verhalten, insbesondere der Standardausgabe, steuern können. In diesem Abschnitt finden Sie eine Liste mit den interessantesten und gängigsten Optionen (was bedeutet, dass dies längst nicht alle sind). Reichen Ihnen diese Optionen nicht aus, müssen Sie in den Manualpages blättern. Als Beispiel dient wieder unsere Datei *mrolympia.dat*:

```
you@host > cat mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

- **-*n*** – Damit wird die Zeile zurückgegeben, in der das Suchmuster gefunden wurde, und zwar mit zusätzlich *n* Zeilen vor und nach dieser Zeile. In der Praxis:

  ```
  you@host > grep -1 Sergio mrolympia.dat
  Larry Scott USA 1965 1966
  Sergio Oliva USA 1967 1968 1969
  Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
  ```

  Hierbei wurde vor und nach der gefundenen Zeile jeweils eine zusätzliche Zeile ausgegeben. Dies ist in der Praxis bei einem etwas längeren Text mit mehreren Abschnitten sehr sinnvoll, da man häufig mit einem ausgegebenen Teilsatz aus Zeile 343 nicht viel anfangen kann.
- **-A** *anzahl* – ähnlich wie **-*n***, nur dass noch zusätzlich *anzahl* Zeilen mit ausgegeben werden, die nach der Zeile enthalten sind.
- **-B** *anzahl* – wie -A *anzahl*, nur dass *anzahl* Zeilen ausgegeben werden, die vor der Zeile enthalten sind, in der der reguläre Ausdruck abgedeckt wurde.
- **-c** (für *count*) – Damit wird nur die Anzahl von Zeilen ausgegeben, die durch den regulären Ausdruck abgedeckt werden:


```
you@host > grep -c '.*' mrolympia.dat
9
you@host > grep -c 'USA' mrolympia.dat
5
```

Hier wurden zum Beispiel die Daten aller Sportler ausgegeben und beim nächsten Ausdruck nur noch die Teilnehmer aus den USA. 5 von den 9 ehemaligen Titelträgern kamen also aus den USA.

- **-h** – Bei dieser Option wird der Dateiname, in dem der Suchstring gefunden wurde, nicht vor den Zeilen mit ausgegeben:

  ```
  you@host > grep 'Ausgabe1' *
  script1:echo "Ausgabe1"
  script1~:echo "Ausgabe1"
  you@host > grep -h 'Ausgabe1' *
  echo "Ausgabe1"
  echo "Ausgabe1"
  ```
- **-i** – Es wird nicht zwischen Groß- und Kleinschreibung unterschieden. Ein Beispiel:

  ```
  you@host > grep 'uSa' mrolympia.dat
  you@host > grep -i 'uSa' mrolympia.dat
  Larry Scott USA 1965 1966
  Sergio Oliva USA 1967 1968 1969
  Chris Dickerson USA 1982
  Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
  Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
  ```
- **-l** – Bei dieser Option werden nur die Dateinamen aufgelistet, in denen eine Zeile mit dem entsprechenden Suchmuster gefunden wurde:

  ```
  you@host > grep -l echo *
  Kap 1 bis 9.doc
  ksh.2010-02-02.linux.i386
  script1
  script1~
  script2
  script2~
  ```
- **-n** – Hiermit wird vor jeder gefundenen Zeile in der Datei die laufende Zeilennummer mit ausgegeben, beispielsweise so:

  ```
  you@host > grep -n echo *
  script1:4:   echo "Die Ausgabe wollen wir nicht!!!"
  script1:7:echo "Ausgabe1"
  script1:11:echo "Ausgabe2"
  ...
  ```

```
script2:9:   echo "Starte script1 ..."
script2~:7:   echo "Warte ein wenig ..."
script2~:9:   echo "Starte script1 ..."
```

- **-q** – Bei Verwendung dieser Option erfolgt keine Ausgabe, sondern es wird 0 zurückgegeben, wenn ein Suchtext gefunden wurde, oder 1, wenn die Suche erfolglos war. Diese Option wird gewöhnlich in Shellscripts verwendet, bei denen man sich meistens nur dafür interessiert, ob eine Datei einen Suchtext enthält oder nicht.
- **-s** – Mit dieser Option werden keine Fehlermeldungen ausgegeben, wenn eine Datei nicht existiert.

```
you@host > grep test /etc/gibtsnicht
grep: /etc/gibtsnicht: Datei oder Verzeichnis nicht gefunden
you@host > grep -s test /etc/gibtsnicht
you@host >
```

- **-v** – Damit werden alle Zeilen ausgegeben, die nicht durch den angegebenen regulären Ausdruck abgedeckt werden:

```
you@host > grep -v 'USA' mrolympia.dat
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Samir Bannout Libanon 1983
Dorian Yates Grossbritannien 1992 1993 1994 1995 1996 1997
```

- **-w** – ist eine Abkürzung für \<wort\>. Damit wird nach ganzen Wörtern im Suchtext gesucht:

```
you@host > grep 'Lib' mrolympia.dat
Samir Bannout Libanon 1983
you@host > grep -w 'Lib' mrolympia.dat
you@host >
```

11.2.5 grep in Kombination mit anderen Programmen

Eine häufig genutzte Einsatzmöglichkeit von grep ist die Kombination mit anderen Programmen wie find oder ls.

```
find ~ -type f -exec grep -q Suchmuster {} \; -print
```

Mit diesem Befehl werden alle Dateien ab dem Home-Verzeichnis ~ an grep übergeben und nach dem Suchmuster durchsucht. Wenn das erfolgreich ist, wird der Dateiname ausgegeben. Das Kommando durchsucht alle regulären Dateien, also auch Bilder und Audiodateien nach der Zeichenfolge. Es kann daher sein, dass Ihnen in diesem »Binärmüll« Treffer angezeigt werden.

11.2.6 egrep (extended grep)

Wie Sie bereits erfahren haben, lassen sich mit egrep erweiterte und weitaus komplexere reguläre Ausdrücke bilden. Allerdings werden Otto Normalverbraucherinnen und -verbraucher (wie auch die meisten Systemadministratoren) mit grep mehr als zufrieden sein. Komplexere reguläre Ausdrücke gehen natürlich enorm auf Kosten der Ausführgeschwindigkeit. Übrigens können Sie einen egrep-Aufruf auch mit grep und der Option -E realisieren:

```
grep -E regex Datei
```

Einige Beispiele mit egrep:

```
you@host > egrep 'Colombo|Columbu' mrolympia.dat
Franco Columbu Argentinien 1976 1981
you@host > egrep 'Colombo|Columbu|Col' mrolympia.dat
Franco Columbu Argentinien 1976 1981
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Wissen Sie nicht genau, wie man einen bestimmten Namen schreibt, können Sie mit | mehrere Suchmuster miteinander verknüpfen. Im Beispiel wird nach »Colombo« oder »Columbu« und im zweiten Beispiel noch zusätzlich nach einer Zeichenfolge »Col« gesucht. Kennen Sie mehrere Personen mit dem Namen »Oliva« und wollen Sie nach einem »Sergio Oliva« und einem »Gregor Oliva« suchen, können Sie das Ganze folgendermaßen definieren:

```
egrep '(Sergio|Gregor) Olivia' mrolympia.dat
```

Nächstes Beispiel:

```
you@host > egrep 'y+' mrolympia.dat
Larry Scott USA 1965 1966
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
```

Mit dem + wird nach mindestens einem »y«-Zeichen gesucht. Wollen Sie beispielsweise alle Zeilen einer Datei ausgeben, die mit mindestens einem oder mehreren Leerzeichen beginnen, können Sie dies folgendermaßen definieren:

```
you@host > egrep '^ +' mrolympia.dat
```

Nächstes Beispiel:

```
you@host > egrep 'US?' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
```

```
Chris Dickerson USA 1982
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Wenn Sie jetzt nicht wissen, ob die Zeichenfolge für die USA »USA« oder »US« lautet, können Sie das Fragezeichen verwenden. Damit legen Sie fest, dass hier ein Zeichen vorkommen darf, aber nicht vorkommen muss. Somit wird sowohl die Zeichenfolge »US« als auch die Zeichenfolge »USA« gefunden.

11.2.7 fgrep (fixed oder fast grep)

`fgrep` wird vorwiegend für »schnelle greps« verwendet (`fgrep` = *fast grep*). Allerdings ist nur eine Suche nach einfachen Zeichenketten möglich. Reguläre Ausdrücke gibt es hierbei nicht – ein Vorteil, wenn Sie nach Zeichenfolgen suchen, die Metazeichen enthalten.

11.2.8 rgrep

Weil `grep` nur im aktuellen Verzeichnis sucht, wurde `rgrep` entwickelt. `rgrep` sucht mit einer rekursiven Suche in Unterverzeichnissen nach entsprechenden Mustern. Bei einer großen Verzeichnistiefe kann dies allerdings problematisch werden. Wenn Sie beispielsweise nur das aktuelle Verzeichnis und die direkten Unterverzeichnisse durchsuchen wollen, würde auch ein `grep` wie

```
grep regex */*
```

ausreichen. Mehr zu `rgrep` entnehmen Sie bitte der Manualpage.

11.3 Aufgaben

1. Mit welchem Kommando können Sie sich anzeigen lassen, welche Benutzer in der Datei */etc/passwd* die `bash` als Standard-Shell eingetragen haben?
2. Sie haben eine Datei, in der es viele Leerzeilen zwischen dem Text gibt und an manchen Stellen auch mehr als ein Leerzeichen zwischen den Wörtern. Wie können Sie diese Datei so formatieren, dass alle Leerzeilen und alle doppelten Leerzeichen aus ihr entfernt werden?
3. Suchen Sie in der Datei *mrolympia.dat* nach allen Zeilen, in denen die Jahreszahlen 1990 bis 1999 vorkommen.
4. Lassen Sie sich von Ihrer ersten Netzwerkkarte nur die IP-Adresse aus der Information von `ifconfig eth0` anzeigen.

Kapitel 12
Der Stream-Editor sed

sed ist ein relativ altes Tool, wird aber immer noch vielfach eingesetzt, etwa um Zeilen einer Datei oder eines Datenstroms zu manipulieren. Besonders häufig und gern wird sed zum Suchen und Ersetzen von Zeichenfolgen verwendet.

12.1 Funktions- und Anwendungsweise von sed

Bevor Sie sich mit dem Editor `sed` auseinandersetzen, möchten wir Ihnen zunächst die grundlegende Funktionsweise des `sed`-Kommandos beschreiben. `sed` wurde bereits in den 1970er-Jahren von Lee E. McMahon in den Bell Labs entwickelt. Der Name »sed« steht für *Stream EDitor*. Natürlich ist `sed` als Editor nicht vergleichbar mit den Ihnen vielleicht bekannten Editoren *vim* oder *Emacs*. Die Grundfunktionen hat `sed` vom Editor `ed` geerbt, einem der ältesten UNIX-Programme, das sich seit Anfang der 1970er-Jahre nicht wesentlich verändert hat und auf allen unixoiden Systemen verfügbar ist.

12.1.1 Grundlegende Funktionsweise

Mittlerweile hat jede Linux- bzw. UNIX-Distribution eine eigene `sed`-Version. Linux verwendet gewöhnlich GNU-`sed`, was eine erweiterte Variante des Ur-`sed` ist. Beispielsweise bietet GNU-`sed` sogenanntes In-Place-Editing, womit eine Änderung direkt im Eingabefilter möglich ist. Solaris wiederum bietet gleich drei Versionen von `sed` an: eine von AT&T abgeleitete, eine von BSD UNIX und eine weitere, die dem XPG4-Standard entspricht (natürlich kann unter Solaris und BSD auch GNU-`sed` verwendet werden). Generell unterscheiden sich die einzelnen Versionen in der Funktionsvielfalt, was hier allerdings kein Problem sein sollte, da wir vorwiegend auf die grundlegenden Funktionen von `sed` eingehen, die jedes `sed` kennen sollte.

`sed` bekommt seine Kommandos entweder aus einer Datei oder von der Kommandozeile. Meistens handelt es sich aber um eine Datei. Diese Datei (bzw. das Kommando aus der Kommandozeile) liest `sed` Zeile für Zeile von der Standardeingabe ein und kopiert das Eingelesene in einen extra dafür angelegten Puffer (in den sogenannten *Patternspace*), bearbeitet diesen Puffer nach einer von Ihnen bestimmten Regel und gibt den veränderten Puffer anschließend wieder auf die Standardausgabe aus. Bei dem

Puffer handelt es sich um eine Kopie einer jeden Zeile. Das heißt, jegliche Kommandoausführung von sed bezieht sich von nun an auf diesen Puffer. Die Quelldatei bleibt hierbei immer unverändert. Es sollte hier klar geworden sein, dass sed zeilenweise arbeitet – eine Zeile wird eingelesen, bearbeitet und wieder ausgegeben. Natürlich findet dieser ganze Arbeitsvorgang in einer Schleife statt (siehe Abbildung 12.1).

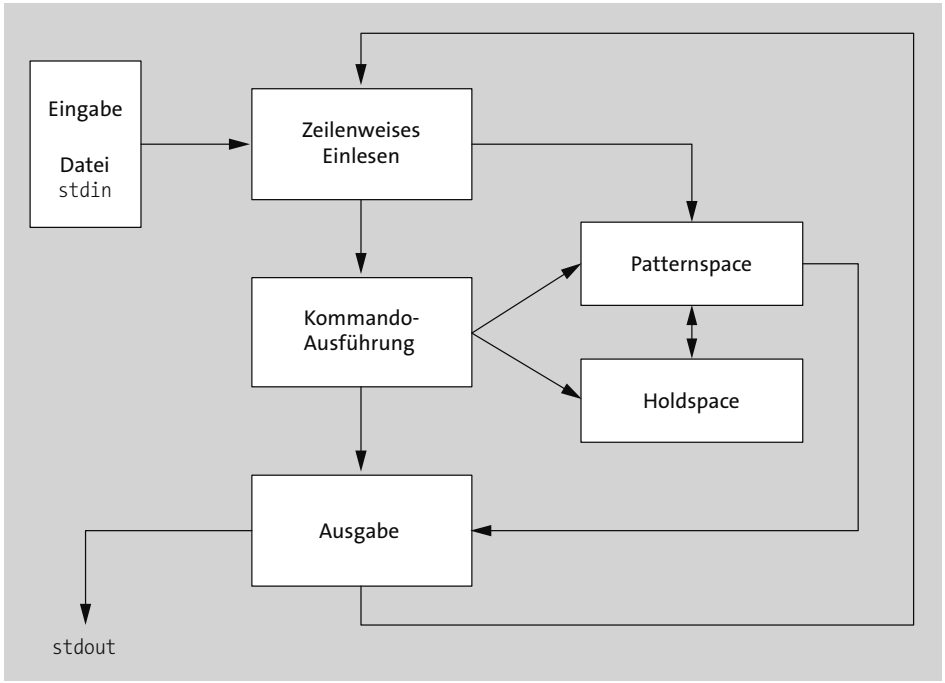

Abbildung 12.1 Die Arbeitsweise von »sed«

Die Quelldatei bleibt unverändert

Dass die Quelldatei unverändert bleibt, sorgt immer wieder für Verwirrung. Aber eigentlich ist dies logisch, denn es ist nun mal nicht möglich, gleichzeitig aus einer Datei zu lesen und in sie hineinzuschreiben. Man muss also die Ausgabe in einer temporären Datei speichern, sie zurückkopieren und die temporäre Datei wieder löschen.

Neben dem Puffer Patternspace gibt es noch einen zweiten Puffer, den *Holdspace*. Diesen können Sie mit speziellen Kommandos verwenden, um Daten untereinander (zwischen dem Pattern- und dem Holdspace) auszutauschen. Die Puffergröße (oder auch die Zeichen, die pro Zeile aufgenommen werden können) ist bei moderneren sed-Versionen unbegrenzt. Bei älteren Versionen gibt es häufig noch eine Begrenzung auf 8.192 Zeichen.

Die Syntax zu sed lautet:

```
sed [-option] Kommando [Datei] ... [Datei_n]
sed -f Kommandodatei [Datei] ... [Datei_n]
```

Damit die Kommandos nicht mit der fleißigen Shell und deren Mühen, Metazeichen vorher zu interpretieren, kollidieren, sollten Sie die sed-Kommandos zwischen Single Quotes setzen:

```
sed [-option] 'Kommando' [Datei] ... [Datei_n]
```

Natürlich können Sie auch mehrere Kommandos auf einmal verwenden. Hierzu müssen Sie entweder die einzelnen Kommandos mit einem Semikolon voneinander trennen, oder Sie schreiben in jede Zeile ein Kommando:

```
sed 'Kommando1 ; Kommando2 ; Kommando3' Datei

sed 'Kommando1
> Kommando2
> Kommando3' Datei
```

Hier sehen Sie als Beispiel drei verschiedene sed-Kommandos, die alle drei zum selben Ziel führen:

```
you@host > sed -n  '/Scott/p' mrolympia.dat
Larry Scott USA 1965 1966
you@host > sed -n  '/Scott/p' < mrolympia.dat
Larry Scott USA 1965 1966
you@host > cat mrolympia.dat | sed -n '/Scott/p'
Larry Scott USA 1965 1966
```

sed liest immer von der Standardeingabe und schreibt standardmäßig auf die Standardausgabe. Mit dem letzten Parameter können Sie einen oder auch mehrere Dateinamen angeben, von denen sed aus dem Eingabestrom lesen soll. Wie Sie im Beispiel sehen konnten, können Sie sich hier auch der Umlenkungszeichen (<, >, |) bedienen. Die nähere Bedeutung der sed-Befehle, die im Beispiel oben verwendet wurden, werden Sie in den folgenden Abschnitten kennenlernen.

12.1.2 Wohin mit der Ausgabe?

Wenn Sie den sed-Editor verwenden, werden Sie wohl im seltensten Fall eine Ausgabe auf dem Bildschirm haben wollen. Es gibt drei gängige Anwendungsmöglichkeiten, um die Ausgabe von sed in die richtigen Bahnen zu lenken.

In diesem Abschnitt behandeln wir erst einmal die gängigsten Methoden von sed, ohne bereits auf die einzelnen Optionen bzw. Features von sed einzugehen.

Hinweis

Die Ausgabe von sed erfolgt auf den Standardausgabekanal, Fehlermeldungen erfolgen auf den Standardfehlerkanal. Der Rückgabewert von sed ist dabei ungleich (meistens größer als) 0.

Mehrere Dateien bearbeiten

Wenn Sie mehrere Dateien auf einmal mit sed bearbeiten wollen bzw. müssen, kann es sinnvoll sein, anstatt eines Kommandos eine Kommandodatei zu verwenden – Sie dürfen hierzu gern auch »sed-Script« sagen. Der Vorteil: Diese Kommandodatei können Sie jederzeit wiederverwenden. Anstatt also eine Datei wie folgt mit sed zu bearbeiten:

```
sed -n 'kommando ; kommando; kommando' Datei > Zieldatei
```

können Sie ein sed-Script verwenden:

```
sed -f ased_script.sed Datei > Zieldatei
```

Damit können Sie de facto mehrere Dateien komfortabel mit einem sed-Script bearbeiten. Theoretisch könnten Sie hierbei vorgefertigte sed-Scripts einsetzen, ohne dass Sie überhaupt Kenntnisse von sed haben müssen (was allerdings nicht sehr empfehlenswert ist). Wie Sie eigene sed-Scripts schreiben, erfahren Sie in Abschnitt 12.5. Gewöhnlich werden Sie das sed-Script auf mehrere Dateien auf einmal in einem Shellscript ansetzen. Dies können Sie dann beispielsweise folgendermaßen machen:

```
# Alle Dateien im aktuellen Verzeichnis
for file in *
do
   sed -f ased_script.sed $file > temp
   # Achtung, hier wird das Original verändert!
   mv tmp $file
done
```

Direkt aus einem Kommando bearbeiten

Im Beispiel oben wurden mehrere Dateien neu bearbeitet. Wenn Sie ein wenig vorausschauender arbeiten, könnten Sie diesen Schritt auch auslassen, indem Sie gleich von vornherein darauf achten, dass man die Datei gar nicht bearbeiten muss (natürlich ist das nicht immer möglich). Dies lässt sich wieder mit einer Pipe realisieren:


```
kommando | sed -f ased_script.sed > Datei
kommando | sed -n 'kommando' > Datei
shellscript | sed -f ased_script.sed > Datei
shellscript | sed -f 'Kommando' > Datei
```

Hierbei schreibt ein Prozess (ein Kommando oder auch ein Shellscript) seine Daten in die Standardausgabe durch die Pipe. Auf der anderen Seite der Pipe wartet sed mit der Standardeingabe auf diese Daten und verarbeitet sie zeilenweise mit dem sed-Script. Die Ausgabe wird mit einer Umlenkung in eine Datei geschrieben.

Direkt aus einer Datei bearbeiten

Selbstverständlich können Sie mit sed auch einen Text aus einer Datei extrahieren, zerlegen und in eine neue Zieldatei speichern, ohne die Originaldatei zu verändern:

```
sed -n '/ein_wort/p' Datei > Zieldatei
```

Damit extrahiert sed alle Zeilen mit dem Wort »ein_wort« aus der *Datei* und schreibt diese Zeilen in die *Zieldatei*.

12.2 Der sed-Befehl

Ein sed-Befehl sieht auf den ersten Blick etwas undurchsichtig aus, ist aber einfacher, als man vermuten würde. Hier ist ein solch typischer sed-Befehl im Rohformat:

```
sed '[adresse1[,adresse2]]kommando' [datei(en)]
```

Alle Argumente, bis auf kommando, sind erst mal optional. Jeder sed-Aufruf benötigt also mindestens einen Kommandoaufruf. Die Operation kommando bezieht sich auf einen bestimmten Bereich bzw. eine bestimmte Adresse einer Datei (solche Adressen werden im nächsten Abschnitt behandelt). Den Bereich können Sie mit adresse1 bestimmen. Geben Sie hierbei noch adresse2 an, so erstreckt sich der Bereich von der Zeile adresse1 bis zur Zeile adresse2. Sie können diesen Bereich auch negieren, indem Sie hinter adresse1 und adresse2 ein !-Zeichen setzen. Dann bezieht sich kommando nur auf den Bereich, der nicht von adresse1 bis adresse2 abgedeckt wird. Ein einfaches Beispiel:

```
you@host > sed -n 'p' file.dat
```

Hiermit geben Sie praktisch die komplette Datei *file.dat* auf dem Bildschirm aus. Das Kommando p steht für *print*, also für die Ausgabe (auf den Bildschirm). Damit sed die Zeile(n) nicht doppelt ausgibt, wurde die Option -n verwendet. Näheres dazu folgt später.

12.3 Adressen

Adressen sind, wie eben erwähnt, entweder fixe Zeilen, ganze Bereiche oder aber Zeilen, die auf einen bestimmten regulären Ausdruck passen. Hier sehen Sie einige Beispiele dafür, wie Sie bestimmte Adressen definieren und welche Zeilen Sie damit selektieren. Zur Demonstration wird immer die Option `-n` verwendet, mit der Sie den Default Output von `sed` abschalten, sowie das Kommando `p`, mit dem Sie die selektierte(n) Zeile(n) auf dem Bildschirm ausgeben lassen können.

Hier wird nur die vierte Zeile der Datei *file.dat* selektiert und ausgegeben:

```
you@host > sed -n '4p' file.dat
```

Nachfolgend werden die Zeilen 4, 5, 6 und 7 aus der Datei *file.dat* selektiert und ausgegeben. Wenn Sie für die »bis«-Adresse einen niedrigeren Wert angeben als für die »von«-Adresse, wird dieser Wert ignoriert:

```
you@host > sed -n '4,7p' file.dat
```

Hiermit werden alle Zeilen von Zeile 4 bis zum Ende der Datei ausgegeben – das Dollarzeichen steht für die letzte Zeile:

```
you@host > sed -n '4,$p' file.dat
```

Damit werden alle Zeilen selektiert und ausgegeben, in denen sich das Wort »wort« befindet:

```
you@host > sed -n '/wort/p' file.dat
```

Hier werden alle Zeilen ausgegeben, die die Zeichenfolge »197« und eine beliebige Zahl von 0 bis 9 enthalten (1970, 1971 ... 1979):

```
you@host > sed -n '/197[0-9]/p' file.dat
```

Neben den Möglichkeiten mit

```
[Adresse1, Adresse2]Kommando
```

und

```
[Adresse]Kommando
```

gibt es auch noch eine dritte Möglichkeit, wie Sie Adressen angeben können. Man kann nämlich durch die Verwendung von geschweiften Klammern mehrere Kommandos zusammenfassen:


```
Adresse{
   Kommando1
   Kommando2
}
```

Oder auch als Einzeiler:

```
Adresse{ Kommando1 ;  Kommando2 ; ... }
```

Beispiel:

```
you@host > sed -n '1{p ; s/USA/ASU/g ; p }' mrolympia.dat
Larry Scott USA 1965 1966
Larry Scott ASU 1965 1966
```

Hiermit bearbeiten Sie die erste Zeile der Datei *mrolympia.dat*. Zuerst geben Sie diese Zeile aus, anschließend führen Sie mit `s/.../.../g` eine globale (`g`) Ersetzung mittels `s` (*substitute*) der Zeichenfolge »USA« durch »ASU« durch und geben daraufhin diese Zeile (gegebenenfalls verändert) nochmals aus. Natürlich können Sie eine solche Ersetzung auch ohne Angabe einer Adresse auf die ganze Datei ausdehnen:

```
you@host > sed -n '{/USA/p ; s/USA/ASU/g ; /ASU/p; }' mrolympia.dat
```

Da hierbei keine direkte Adressierung verwendet wird, können Sie das Ganze auch gleich ohne geschweifte Klammern schreiben:

```
you@host > sed -n '/USA/p ; s/USA/ASU/g ; /ASU/p' mrolympia.dat
```

Natürlich können Sie mit den geschweiften Klammern auch einen Adressbereich verwenden:

```
you@host > sed -n '1,5{/USA/p ; s/USA/ASU/g ; /ASU/p; }' \
> mrolympia.dat
```

Hierbei wurden die Zeilen 1 bis 5 zusammengefasst, um alle Kommandos in den geschweiften Klammern darauf auszuführen.

12.4 Kommandos, Substitutionsflags und Optionen von sed

In den obigen Beispielen haben Sie bisher immer die Option `-n` und das Kommando `p` verwendet. `sed` bietet neben `p` eine interessante Fülle von Kommandos an. Bevor Sie die einzelnen Kommandos und Optionen in der Praxis einsetzen, sollten Sie sich die Tabelle 12.1 bis Tabelle 12.3 ansehen, die die Optionen, die Kommandos und die sogenannten Substitutionsflags zusammenfassen.

Tabelle 12.1 listet die wichtigsten Kommandos von sed auf.

Kommando	Bedeutung
a	(für *append*) Fügt eine oder mehrere Zeilen an die selektierte Zeile an.
c	(für *change*) Ersetzt die selektierte Zeile durch eine oder mehrere neue.
d	(für *delete*) Löscht Zeile(n).
g	(für *get »buffer«*) Kopiert den Inhalt des temporären Puffers (*Holdspace*) in den Arbeitspuffer (*Patternspace*).
G	(für *GetNewline*) Fügt den Inhalt des temporären Puffers (*Holdspace*) an den Arbeitspuffer (*Patternspace*) an.
h	(für *hold »buffer«*) Gegenstück zu g. Kopiert den Inhalt des Arbeitspuffers (*Patternspace*) in den temporären Puffer (*Holdspace*).
H	(für *HoldNewline*) Gegenstück zu G. Fügt den Inhalt des Arbeitspuffers (*Patternspace*) an den temporären Puffer (*Holdspace*) an.
i	(für *insert*) Fügt eine neue Zeile vor der selektierten Zeile ein.
l	(für *listing*) Zeigt nicht druckbare Zeichen an.
n	(für *next*) Wendet das nächste Kommando statt des aktuellen Kommandos auf die nächste Zeile an.
p	(für *print*) Gibt die Zeilen aus.
q	(für *quit*) Beendet sed.
r	(für *read*) Datei integrieren; liest Zeilen aus einer Datei ein.
s	(für *substitute*) Ersetzt eine Zeichenfolge durch eine andere.
x	(für *eXchange*) Vertauschen des temporären Puffers (*Holdspace*) mit dem Arbeitspuffer (*Patternspace*).
y	(für *yank*) Zeichen aus einer Liste ersetzen; Ersetzen eines Zeichens durch ein anderes.
w	(für *write*) Schreibt Zeilen in eine Datei.
!	Negation; wendet die Kommandos auf Zeilen an, die nicht zutreffen.

Tabelle 12.1 Gängige Basiskommandos von »sed«

Wenn Sie eine Substitution mittels s/.../.../ verwenden möchten, können Sie zusätzliche Flags (Substitutionsflags) angeben, die am Ende des Befehls notiert werden. Beispielsweise findet mit s/.../.../g eine globale Ersetzung statt. Das heißt, es werden alle Vorkommen eines Musters in einer Zeile ersetzt. Ohne g würde nur das erste Vorkommen ersetzt. Tabelle 12.2 enthält einige dieser Flags und deren Bedeutung.

Flag	Bedeutung
g	Globale Ersetzung (aller Vorkommen eines Musters in der Zeile).
p	Ausgabe der Zeile.
w	Tauscht den Inhalt des Zwischenspeichers gegen die aktuell selektierte Zeile aus.

Tabelle 12.2 Einige Substitutionsflags

Tabelle 12.3 enthält die Schalter-Optionen, die Sie mit sed verwenden können.

Option	Bedeutung
-n	Schaltet »Default Output« aus. Mit dem Default Output ist die Ausgabe des Puffers (*Patternspace*) gemeint.
-e	Mehrere Befehle nacheinander ausführen. Man gibt praktisch ein Script bzw. die Kommandos direkt in der Kommandozeile ein.
-f	Die sed-Befehle in einem Script (sed-Script) zusammenfassen und dieses Script mittels -f übergeben. Praktisch wie die Option -e, nur dass hier anstatt des Scripts bzw. Kommandos in der Kommandozeile der Name eines Scriptfiles angegeben wird.

Tabelle 12.3 Gängige Schalter-Optionen für »sed«

Noch mehr Optionen

Die meisten sed-Versionen (GNU-sed, BSD-sed, FreeBSD-sed und ssed) bieten neben diesen Optionen noch einige weitere an, deren Bedeutung Sie bei Bedarf der Manualpage von sed entnehmen können.

Wie grep versteht auch sed eine Menge regulärer Ausdrücke. Auch hier bieten die unterschiedlichen sed-Versionen die eine oder andere Erweiterung an. Allerdings begnügen wir uns hier wieder mit den grundlegenden Ausdrücken. Die regulären Ausdrücke, die sed versteht, sind in Tabelle 12.4 aufgelistet.

Ausdruck	Bedeutung	Beispiel	Erklärung
^	Anfang einer Zeile	/^wort/	Behandelt alle Zeilen, die mit der Zeichenfolge wort beginnen.
$	Ende einer Zeile	/wort$/	Behandelt alle Zeilen, die mit der Zeichenfolge wort enden.
*	Keine, eine oder mehrere Wiederholungen des vorhergehenden Zeichens (oder Gruppe)	/*wort/	Behandelt alle Zeilen, in denen vor wort kein, ein oder mehrere Zeichen stehen.
.	Ein Zeichen	/w.rt/	Behandelt alle Zeilen mit den Zeichenfolgen wort, wert, wirt, wart etc. (Ausnahme ist das Newline-Zeichen.)
[]	Ein Zeichen aus der Menge	/[Ww]ort/	Behandelt alle Zeilen mit der Zeichenfolge wort oder Wort.
[^]	Kein Zeichen aus der Menge	/[^Ww]ort/	Behandelt alle Zeilen, die nicht die Zeichenfolge wort oder Wort enthalten.
\(...\)	Speichern eines enthaltenen Musters	S/\(wort\)a/\1b/	Die Zeichenfolge wort wird in \1 gespeichert. Diese Referenz auf das Muster wort verwenden Sie nun, um in der Zeichenfolge worta das Wort wortb zu ersetzen. Damit lassen sich bis zu 9 solcher Referenzen definieren, auf die Sie mit \1, ... \9 zugreifen können.
&	Enthält das Suchmuster	S/wort/Ant&en/	Das Ampersand-Zeichen repräsentiert den Suchstring. Im Beispiel wird jeder Suchstring wort durch Antworten ersetzt.

Tabelle 12.4 Grundlegende reguläre Ausdrücke, die »sed« versteht

Ausdruck	Bedeutung	Beispiel	Erklärung
\<	Wortanfang	/\<wort/	Findet alle Zeilen mit einem String, der mit wort beginnt, also wortreich, wortarm, nicht aber Vorwort oder Nachwort.
\>	Wortende	/wort\>/	Findet alle Zeilen mit einem Wort, das mit wort endet, also Vorwort, Nachwort, nicht aber wortreich oder wortarm.
x\{m\} x\{m,\} x\{m,n\}	m-fache Wiederholung des Zeichens x mindestens m-fache Wiederholung des Zeichens x mindestens m-, maximal n-fache Wiederholung des Zeichens x		

Tabelle 12.4 Grundlegende reguläre Ausdrücke, die »sed« versteht (Forts.)

Nachdem Sie bisher eine Menge Theorie und viele Tabellen gesehen haben, folgt nun ein Praxisteil zu der Verwendung der einzelnen Kommandos von sed. Als Beispiel nutzen wir wieder die Datei *mrolympia.dat*:

```
you@host > cat mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Des Weiteren muss erwähnt werden, dass alle Beispiele auf dem Patternspace-Puffer ausgeführt werden. Somit bezieht sich die Änderung nur auf die Ausgabe auf dem Bildschirm (Standardausgabe). Sofern Sie hier gern eine bleibende Änderung vornehmen wollen (was in der Regel der Fall sein sollte), können Sie sich auf Abschnitt 12.1.2 beziehen. Die einfachste Lösung dürfte hier wohl die Verwendung des Umlenkungszeichens am Ende des sed-Befehls sein.

12.4.1 Das a-Kommando – Zeile(n) anfügen

Die Syntax lautet:

```
a\
neue Zeile
# oder in neueren sed-Versionen auch
a neue Zeile
```

Mit dem a-Kommando (*append*) können Sie eine neue Zeile hinter einer gefundenen Zeile einfügen, beispielsweise so:

```
you@host > sed '/2004/ a Jay Cutler 2005' mrolympia.dat
...
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
Jay Cutler 2005
```

Sollte die Zeile länger werden, können Sie der Übersichtlichkeit zuliebe einen Backslash verwenden:

```
you@host > sed '/2004/ a\
> Prognose für 2005: J.Cutler; R.Coleman; M.Rühl' mrolympia.dat
...
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
Prognose für 2005: J.Cutler; R.Coleman; M.Rühl'
```

Das Kommando »a« bei älteren sed-Versionen

Dass das Kommando a und die neue Zeile in derselben Zeile stehen, ist eine Erweiterung neuerer sed-Versionen. Ältere sed-Programme kennen häufig nur das a-Kommando, gefolgt von einem Backslash, und die *neue Zeile* folgt dann in der nächsten Zeile. Dies sollten Sie wissen, falls Sie auf einem etwas betagteren Rechner mit sed arbeiten müssen.

Beachten Sie außerdem, dass jedes Whitespace nach dem a-Kommando hinter dem Backslash ebenfalls als ein Zeichen interpretiert und verwendet wird.

12.4.2 Das c-Kommando – Zeilen ersetzen

Die Syntax lautet:

```
c\
neuer Text
# oder in neueren sed-Versionen
c neuer Text
```

Wollen Sie eine gefundene Zeile komplett durch eine neue Zeile ersetzen, können Sie das c-Kommando (*change*) verwenden. Die Funktionsweise entspricht der des a-Kommandos.

```
you@host > sed '/\<Oliva\>/ c \
> Sergio Oliva Cuba 1967 1968 1969' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva Cuba 1967 1968 1969
...
```

Im Beispiel wurden alle Zeilen mit dem Vorkommen des Worts »Oliva« durch die in der darauffolgenden Zeile vorgenommene Eingabe ersetzt. Hier wurde etwa die Nationalität verändert. Dem c-Kommando müssen wie beim a-Kommando ein Backslash und ein Newline-Zeichen folgen – obgleich auch hier die neueren sed-Versionen ohne Backslash und Newline-Zeichen auskommen.

12.4.3 Das d-Kommando – Zeilen löschen

Zum Löschen von Zeilen wird das d-Kommando (*delete*) verwendet. Damit wird die entsprechende Adresse im Puffer gelöscht. Im Folgenden sehen Sie ein paar Beispiele.

- Löscht die fünfte Zeile:

  ```
  you@host > sed '5d' mrolympia.dat
  ```

- Löscht ab der fünften bis zur neunten Zeile:

  ```
  you@host > sed '5,9d' mrolympia.dat
  ```

- Löscht alle Zeilen ab der fünften Zeile bis zum Ende der Datei:

  ```
  you@host > sed '5,$d' mrolympia.dat
  ```

- Löscht alle Zeilen, die das Muster »USA« enthalten:

  ```
  you@host > sed '/USA/d' mrolympia.dat
  ```

- Löscht alle Zeilen, die nicht das Muster »USA« enthalten:

  ```
  you@host > sed '/!USA/d' mrolympia.dat
  ```

12.4.4 Die Kommandos h, H, g, G und x – Arbeiten mit den Puffern

Mit den Kommandos `g` (*get*), `G` (*GetNewline*), `h` (*hold*), `H` (*HoldNewline*) und `x` (*eXchange*) können Sie mit den Puffern (Patternspace und Holdspace) arbeiten.

Mit dem Kommando `h` können Sie den aktuellen Inhalt des Zwischenpuffers (Patternspace) in einen anderen Puffer (Holdspace) sichern. Mit `H` hängen Sie ein Newline-Zeichen an das Ende des Puffers, gefolgt vom Inhalt des Zwischenpuffers.

Mit `g`, dem Gegenstück zu `h`, ersetzen Sie die aktuelle Zeile des Zwischenpuffers durch den Inhalt des Puffers, den Sie zuvor mit `h` gesichert haben. Mit `G` hängen Sie ein Newline-Zeichen an das Ende des Zwischenpuffers, gefolgt vom Inhalt des Puffers.

Mit dem Kommando `x` hingegen tauschen Sie den Inhalt des Zwischenpuffers gegen den Inhalt des anderen Puffers aus.

Ein Beispiel:

```
you@host > sed -e '/Sergio/{h;d}' -e '$G' mrolympia.dat
```

Hierbei führen Sie zunächst eine Suche nach dem Muster `Sergio` in der Datei *mrolympia.dat* durch. Wird eine entsprechende Zeile gefunden, legen Sie diese in den Holdspace zum Zwischenspeichern (Kommando `h`). Im nächsten Schritt löschen Sie diese Zeile (Kommando `d`). Anschließend führen Sie (Option `-e`) das nächste Kommando aus. Hier hängen Sie praktisch die Zeile im Holdspace (`G`) mit einem beginnenden Newline-Zeichen an das Ende des Patternspace. Diese Zeile wird an das Ende angehängt (`$`-Zeichen). Wollen Sie diese Zeile beispielsweise in die fünfte Zeile platzieren, gehen Sie wie folgt vor:

```
you@host > sed -e '/Sergio/{h;d}' -e '5G' mrolympia.dat
Larry Scott USA 1965 1966
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Sergio Oliva USA 1967 1968 1969
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Bitte beachten Sie, dass hierbei die gelöschte Zeile beim Einfügen mitberechnet wird. Was passiert nun, wenn Sie die Zeile im Holdspace ohne Newline-Zeichen, wie dies mit `g` der Fall ist, im Patternspace einfügen? Es geschieht Folgendes:

```
you@host > sed -e '/Sergio/{h;d}' -e '5g' mrolympia.dat
Larry Scott USA 1965 1966
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
```



```
Franco Columbu Argentinien 1976 1981
Sergio Oliva USA 1967 1968 1969
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Hier wird praktisch gnadenlos eine Zeile (hier »Chris Dickerson«) überschrieben. Beachten Sie dies bitte bei der Verwendung von `g` und `G` bzw. `h` und `H`.

Noch ein Beispiel zum Kommando `x`:

```
you@host > sed -e '/Sergio/{h;d}' -e '/Dorian/x' \
> -e '$G' mrolympia.dat
Larry Scott USA 1965 1966
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Sergio Oliva USA 1967 1968 1969
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
Jay Cutler 2005
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
```

Hier suchen Sie zunächst nach dem Muster `Sergio`, legen dieses in den Holdspace (Kommando `h`) und löschen daraufhin die Zeile im Patternspace (Kommando `d`). Dann suchen Sie nach dem Muster `Dorian` und tauschen den Inhalt des Patternspace (Zeile mit »Dorian«) gegen den Inhalt des Holdspace (Zeile mit »Sergio«) aus. Jetzt befindet sich im Holdspace die Zeile mit »Dorian«. Diese hängen Sie mit einem weiteren Befehl (Option `-e`) an das Ende des Patternspace.

12.4.5 Das Kommando i – Einfügen von Zeilen

Die Syntax:

```
i\
Text zum Einfügen
# oder bei neueren sed-Versionen
i Text zum Einfügen
```

Bei diesem Kommando gilt all das, was schon zum Kommando `a` gesagt wurde. Damit können Sie eine neue Zeile vor der gefundenen Zeile einfügen.

```
you@host > sed '/Franco/ i \
> ---Zeile---' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
---Zeile---

Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
...
```

12.4.6 Das p-Kommando – Patternspace ausgeben

Dieses Kommando wurde schon zur Genüge verwendet. Mit `p` geben Sie den Patternspace aus. Wollen Sie beispielsweise einfach eine komplette Datei ausgeben lassen, können Sie folgendermaßen vorgehen:

```
you@host > sed -n 'p' mrolympia.dat
```

Die Option `-n` ist hierbei nötig, da sonst neben dem Patternspace-Puffer auch noch der Default Output mit ausgegeben würde und Sie somit eine doppelte Ausgabe hätten. Mit `-n` schalten Sie den Default Output aus. Dies wird häufig vergessen und führt zu Fehlern. Wollen Sie beispielsweise die letzte Zeile in einer Datei ausgeben lassen und vergessen dabei, den Default Output abzuschalten, bekommen Sie folgende Ausgabe zurück:

```
you@host > sed '$p' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Hier wird trotzdem die komplette Datei ausgegeben, da der Default Output weiterhin über den Bildschirm rauscht. Die letzte Zeile hingegen ist doppelt vorhanden, weil hier zum einen der Default Output seine Arbeit gemacht hat und zum anderen das `p`-Kommando auch noch einmal die letzte Zeile mit ausgibt. Richtig wäre hier also:


```
you@host > sed -n '$p' mrolympia.dat
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Im Folgenden sehen Sie noch einige typische Beispiele mit dem `p`-Kommando:

```
you@host > sed -n '4p' mrolympia.dat
Franco Columbu Argentinien 1976 1981
```

Hier wird die vierte Zeile ausgegeben.

```
you@host > sed -n '4,6p' mrolympia.dat
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
```

Hiermit werden die Zeilen 4 bis 6 ausgegeben.

```
you@host > sed -n '/USA/p' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Chris Dickerson USA 1982
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Hierbei werden alle Zeilen mit der Textfolge »USA« ausgegeben.

12.4.7 Das Kommando q – Beenden

Mit dem Kommando `q` können Sie das laufende `sed`-Script durch eine Verzweigung zum Scriptende beenden. Vorher wird aber noch der Patternspace ausgegeben. Ein einfaches Beispiel:

```
you@host > sed -n '/USA/{p;q}' mrolympia.dat
Larry Scott USA 1965 1966
```

Hier wird der `sed`-Befehl nach dem ersten gefundenen Muster »USA« beendet. Zuvor wird noch der Inhalt des Patternspace ausgegeben.

12.4.8 Die Kommandos r und w

Mit dem Kommando `r` können Sie einen Text aus einer Datei einschieben. In den seltensten Fällen werden Sie einen Text in nur einer Datei einfügen wollen. Hierzu sei folgende einfache Textdatei gegeben:

```
you@host > cat header.txt
--------------------------------
Vorname Name Nationalität Jahr(e)
--------------------------------
```

Der Inhalt dieser Datei soll jetzt mit dem Kommando `r` in die letzte Zeile eingefügt werden:

```
you@host > sed '$r header.txt' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
...
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
Jay Cutler 2005
--------------------------------
Vorname Name Nationalität Jahr(e)
--------------------------------
```

Natürlich können Sie auch die Zeilennummer oder ein Muster als Adresse für das Einfügen verwenden:

```
you@host > sed -e '/Arnold/r header.txt' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
--------------------------------
Vorname Name Nationalität Jahr(e)
--------------------------------
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
...
```

Ebenso können Sie mehr als nur eine Datei mithilfe der `-e`-Option aus einer Datei einlesen lassen und in die entsprechenden Zeilen einfügen:

```
you@host > sed -e '3r file1.dat' -e '7r file2.dat' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
----------Ich bin file1------------
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
```



```
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
----------Ich bin file2------------
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

Das entsprechende Gegenstück zum Kommando `r` erhalten Sie mit dem Kommando `w`, mit dem Sie das Ergebnis von `sed` in einer Datei speichern können:

```
you@host > sed -n '/USA/w USA.dat' mrolympia.dat
you@host > cat USA.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Chris Dickerson USA 1982
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
```

12.4.9 Das Kommando s – substitute

Bei dem `s`-Kommando handelt es sich wohl um das meistverwendete Kommando in `sed`, und es ist auch einer der Hauptgründe, `sed` überhaupt zu verwenden. Die Syntax lautet:

```
sed -e 's/altes_Muster/neues_Muster/flag' datei
sed -e 's?altes_Muster?neues_Muster?flag' datei
```

Damit können Sie das Muster »altes_Muster« (im Patternspace) von *datei* durch »neues_Muster« ersetzen. Als Trennzeichen (hier je einmal mit / und ?) können Sie praktisch jedes beliebige Zeichen verwenden (außer den Backslash und das Newline-Zeichen), es darf nur nicht Bestandteil des Musters sein.

In der Grundform, ohne Angabe von `flag`, ersetzt das Kommando `s` nur das erste Vorkommen eines Musters pro Zeile. Somit können Sie mit der Angabe von `flag` auch das Verhalten von `s` verändern. Am häufigsten verwendet wird wohl das Flag `g`, mit dem alle Vorkommen eines Musters in einer Zeile nacheinander ersetzt werden – also eine globale Ersetzung. Mit dem Flag `p` wird nach der letzten Ersetzung der Patternspace ausgegeben, und mit dem Flag `w` können Sie den Patternspace in die angegebene Datei schreiben. Allerdings muss diese Option die letzte im Flag sein, da der Dateiname bis zum Ende der Zeile geht.

Selbstverständlich werden gerade zu diesem Kommando jetzt massenhaft Beispiele folgen:

```
you@host > sed 's/USA/Amerika/g' mrolympia.dat
Larry Scott Amerika 1965 1966
Sergio Oliva Amerika 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson Amerika 1982
Samir Bannout Libanon 1983
Lee Haney Amerika 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman Amerika 1998 1999 2000 2001 2002 2003 2004
```

Hier wurden global alle Zeichenfolgen »USA« durch »Amerika« ersetzt. Natürlich können Sie hierbei auch einzelne Zeilen mithilfe einer Adresse verändern:

```
you@host > sed '/Oliva/ s/USA/Cuba/' mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva Cuba 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
...
```

Hier wurde nur die Zeile ersetzt (»USA« durch »Cuba«), in der sich die Textfolge »Oliva« befindet. Wollen Sie nicht, dass hier die komplette Datei ausgegeben wird, sondern nur die Zeile(n), die ersetzt wurde(n), müssen Sie das Flag `p` (mit der Option `-n`) verwenden:

```
you@host > sed -n '/Oliva/ s/USA/Cuba/p' mrolympia.dat
Sergio Oliva Cuba 1967 1968 1969
```

Das nächste Beispiel sieht so aus:

```
you@host > sed 's/[12][90]/-/' mrolympia.dat
Larry Scott USA -65 1966
Sergio Oliva USA -67 1968 1969
Arnold Schwarzenegger Österreich -70 1971 1972 1973 1974 1975 1980
Franco Columbu Argentinien -76 1981
Chris Dickerson USA -82
Samir Bannout Libanon -83
Lee Haney USA -84 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien -92 1993 1994 1995 1996 1997
Ronnie Coleman USA -98 1999 2000 2001 2002 2003 2004
```

Hier konnten Sie zum ersten Mal den Einfluss des `g`-Flags erkennen. Es sollten wohl alle Zeichenfolgen »19« und »20« durch ein Minuszeichen ersetzt werden. Dasselbe nochmals mit dem Flag `g`:


```
you@host > sed 's/[12][90]/-/g' mrolympia.dat
Larry Scott USA -65 -66
Sergio Oliva USA -67 -68 -69
Arnold Schwarzenegger Österreich -70 -71 -72 -73 -74 -75 -80
Franco Columbu Argentinien -76 -81
Chris Dickerson USA -82
Samir Bannout Libanon -83
Lee Haney USA -84 -85 -86 -87 -88 -89 -90 -91
Dorian Yates Großbritannien -92 -93 -94 -95 -96 -97
Ronnie Coleman USA -98 -99 -00 -01 -02 -03 -04
```

Hier werden auch die Zeichenfolgen »10«, »20« und »29« beachtet. Wollen Sie statt einer Ersetzung eine Ergänzung vornehmen, können Sie das Ampersand-Zeichen (`&`) beim Ersetzungsstring verwenden:

```
you@host > sed -n 's/Franco Columbu/Dr. &/p' mrolympia.dat
Dr. Franco Columbu Argentinien 1976 1981
```

Hier wurde der Suchstring »Franco Columbu« exakt an der Position des Ampersand-Zeichens beim Ersetzungsstring verwendet und ein Titel davor gesetzt. Sofern Sie etwas Derartiges global durchführen müssen und nur die ersetzten Pattern ausgeben wollen, können Sie auch die Flags `g` und `p` gleichzeitig verwenden:

```
you@host > sed -n 's/USA/& (Amerika)/gp' mrolympia.dat
Larry Scott USA (Amerika) 1965 1966
Sergio Oliva USA (Amerika) 1967 1968 1969
Chris Dickerson USA (Amerika) 1982
Lee Haney USA (Amerika) 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA (Amerika) 1998 1999 2000 2001 2002 2003 2004
```

Mit der Option `-e` können Sie mehrere Ersetzungen auf einmal durchführen lassen:

```
you@host > sed -e 's/USA/& (Amerika)/g' \
> -e 's/[12][90]/-/g'  mrolympia.dat
Larry Scott USA (Amerika) -65 -66
Sergio Oliva USA (Amerika) -67 -68 -69
Arnold Schwarzenegger Österreich -70 -71 -72 -73 -74 -75 -80
Franco Columbu Argentinien -76 -81
Chris Dickerson USA (Amerika) -82
Samir Bannout Libanon -83
Lee Haney USA (Amerika) -84 -85 -86 -87 -88 -89 -90 -91
Dorian Yates Großbritannien -92 -93 -94 -95 -96 -97
Ronnie Coleman USA (Amerika) -98 -99 -00 -01 -02 -03 -04
```

Gleiches können Sie übrigens auch mit einer Referenz wie folgt machen:

```
you@host > sed -n 's/\(USA\)/\1 (Amerika)/gp' mrolympia.dat
Larry Scott USA (Amerika) 1965 1966
Sergio Oliva USA (Amerika) 1967 1968 1969
Chris Dickerson USA (Amerika) 1982
Lee Haney USA (Amerika) 1984 1985 1986 1987 1988 1989 1990 1991
Ronnie Coleman USA (Amerika) 1998 1999 2000 2001 2002 2003 2004
```

Hier wird `\1` als Referenz auf »USA« beim Ersetzungsstring verwendet.

Betrachten Sie mal folgendes Beispiel:

```
you@host > sed -n 's/\([12][90]\)\(..\)/\/\1\2\//gp' \
> mrolympia.dat
```

Beim Anblick dieser Zeile fällt es schon schwer, den Überblick zu behalten. Hier sollen alle Jahreszahlen zwischen zwei Schrägstriche gesetzt werden (/2000/, /2001/ etc). Bei derartigen Zeichenorgien empfiehlt es sich, häufiger mal ein anderes Trennzeichen zu verwenden:

```
you@host > sed -n 's#\([12][90]\)\(..\)#\/\1\2\/#gp' \
> mrolympia.dat
```

Jetzt kann man wenigstens leichter zwischen dem Such- und dem Ersetzungsstring unterscheiden.

Man könnte noch extremere Auswüchse zum Suchen und Ersetzen mit `sed` schreiben, doch erscheint es uns an dieser Stelle wichtiger, ein Problem mit den regulären Ausdrücken zu erwähnen, das Jürgen zur Weißglut gebracht hat, als er einen Artikel zur Verwendung von `sed` in Verbindung mit HTML-Seiten schrieb. Erst ein Artikel im Internet, den Thomas Pircher (*https://tty1.net/sed-tutorium_de.html*) verfasste, hat dann für klare Verhältnisse gesorgt. Das Problem: Ein regulärer Ausdruck sucht sich immer den längsten passenden String. Nehmen wir als Beispiel diese Textfolge in einem HTML-Dokument:

```
Dies ist eine <strong>verflixte</strong> Stelle in einem
<strong>verflixten</strong> HTML-Dokument.
```

Jürgen wollte alle HTML-Tags aus dem Dokument entfernen, um so eine normale Textdatei daraus zu machen. Mit folgendem Konstrukt wollte er dies realisieren:

```
sed -e 's/<.*>//g' index.html
```


Das Ergebnis sah so aus:

```
Das ist ein HTML-Dokument.
```

Leider hatte Jürgen vergessen, dass `.*` so gefräßig ist (eigentlich war es ihm aus Perl bekannt), dass es die längste Zeichenfolge sucht, also alles von

```
<strong>verflixte</strong> Stelle in einem <strong>verflixten
</strong>
```

haben will. Um aber nur die Zeichen bis zum ersten Auftreten des Zeichens > zu löschen, muss man mit folgender Bereichsangabe arbeiten:

```
sed -e 's/<[^>]*>//g' index.html
```

Hieran können Sie erkennen, dass die Verwendung regulärer Ausdrücke recht kompliziert werden kann und dass man häufig um eine intensivere Beschäftigung mit ihnen nicht herumkommt. Gerade wenn Sie reguläre Ausdrücke beim Suchen und Ersetzen mehrerer Dateien verwenden, sollten Sie wissen, was Sie tun, und gegebenenfalls einen Probelauf vornehmen bzw. ein Backup zur Hand haben. Zum Glück – und vielleicht auch deswegen – wurde `sed` so konzipiert, dass die Manipulation erst mal nicht an der Originaldatei vorgenommen wird.

Manch einer mag jetzt fragen, wie man Dateien vom DOS-Format ins UNIX-Format und umgekehrt konvertieren kann. Mit `sed` ist dies leichter, als Sie denken. Gehen wir davon aus, dass eine DOS-Zeile mit CR und LF endet. Wir haben dieses Beispiel der `sed`-FAQ von Eric Pement (*http://sed.sourceforge.net/sedfaq.html*) entnommen:

```
# 3. Under UNIX: convert DOS newlines (CR/LF) to Unix format
sed 's/.$//' file    # assumes that all lines end with CR/LF
sed 's/^M$// file    # in bash/tcsh, press Ctrl-V then Ctrl-M
```

Allerdings stellt sich die Frage, ob Sie hierbei nicht lieber auf die Tools `dos2unix` bzw. `unix2dos` zurückgreifen. Beim `vim` können Sie auch mittels `set` (`set fileformat=dos` oder `set fileformat=unix`) das Dateiformat angeben.

Konvertierung von DOS-Zeilen in UNIX-Zeilen

Vielleicht fragen Sie sich, warum in diesem Beispiel zur Konvertierung von DOS-Zeilen in UNIX-Zeilen nicht folgendes Konstrukt verwendet wurde:

```
's/\r\n/\n/g'
```

Die Antwort ist: Weil es so nicht überall (beispielsweise bei Debian) zu funktionieren scheint.

12.4.10 Das Kommando y

Ein weiteres Kommando, das neben dem Kommando `s` gern verwendet wird, ist `y` (*yank*), mit dem Sie eine Ersetzung von Zeichen aus einer Liste vornehmen können. Die Syntax lautet:

```
y/Quellzeichen/Zielzeichen/
```

Hiermit werden alle Zeichen in `Quellzeichen` in die entsprechenden Zeichen in `Zielzeichen` umgewandelt. Ist eine der Listen leer oder unterschiedlich lang, wird `sed` mit einem Fehler beendet. Natürlich können Sie auch (wie schon beim Kommando `s`) als Trenner ein anderes Zeichen als / verwenden. Beispielsweise können Sie mit folgendem `sed`-Befehl eine Datei nach der »rot-13«-Methode verschlüsseln (hier nur auf die Kleinbuchstaben beschränkt):

```
you@host > cp mrolympia.dat mrolympia.bak
you@host > sed -e \
> 'y/abcdefghijklmnopqrstuvwxyz/nopqrstuvwxyzabcdefghijklm/' \
> mrolympia.bak > mrolympia.dat
you@host > cat mrolympia.dat
Lneel Spbgg USA 1965 1966
Sretvb Oyvin USA 1967 1968 1969
Aeabyq Spujnemrarttre Öfgrervpu 1970 1971 1972 1973 1974 1975
Fenapb Cbyhzoh Aetragvavra 1976 1981
...
```

Hiermit werden alle (Klein-)Buchstaben um 13 Zeichen verschoben. Aus »a« wird dadurch »n«, aus »b« wird »o«, aus »c« wird »p« usw. Wollen Sie das Ganze wieder rückgängig machen, brauchen Sie `Quellzeichen` und `Zielzeichen` aus dem Beispiel nur auszutauschen:

```
you@host > cp mrolympia.dat mrolympia.bak
you@host > sed -e \
> 'y/nopqrstuvwxyzabcdefghijklm/abcdefghijklmnopqrstuvwxyz/' \
> mrolympia.bak > mrolympia.dat
you@host > cat mrolympia.dat
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
...
```


12.5 sed-Scripts

Neben der Methode, den `sed`-Befehl in den Shellscripts wie einen üblichen Befehl oder `sed` in der Kommandozeile zu verwenden, haben Sie noch eine dritte Möglichkeit, nämlich echte `sed`-Scripts zu schreiben. Der Vorteil ist, dass Sie so immer wieder verwendete `sed`-Kommandos gleich zur Hand haben und vor allem bei etwas längeren `sed`-Kommandos den Überblick behalten. Das `sed`-Script können Sie natürlich trotzdem in einem Shellscript nutzen, nur benötigen Sie dann neben dem Shellscript auch noch das `sed`-Script.

Damit `sed` weiß, dass es sein Kommando aus einer Datei erhält, müssen Sie die Option `-f` (*file*) verwenden. Die Syntax lautet:

```
sed -f sed_script.sed Datei
```

Folgendes müssen Sie beim Verwenden von `sed`-Scripts beachten:

- Mehrere Kommandos in einer Zeile müssen durch ein Semikolon getrennt werden.
- Es dürfen keine Leerzeichen, Tabulatoren etc. vor und nach den Kommandos stehen.
- Eine Zeile, die mit # beginnt, wird als Kommentar behandelt.

Hier sehen Sie ein einfaches `sed`-Script, mit dem Sie eine Textdatei in eine HTML-Datei einbetten können:

```
# Name: text2html.sed

# Sonderzeichen '<', '>' und '&' ins HTML-Format
s/&/\&amp;/g
s/</\&lt;/g
s/>/\&gt;/g

# Zeile 1 selektieren wir mit insert für den HTML-Header
1 i\
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"\
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\
<html>\
<head>\
<title>\
Converted with txt2html.sed\
</title>\
</head>\
<body>\
<pre>
```

```
# Hier kommt der Text der Datei hin

# Mit append hängen wir den Footer ans Ende
$ a\
</pre>\
</body>\
</html>
```

Dieses Script können Sie nun wie folgt anwenden:

```
you@host > sed -f text2html.sed mrolympia.dat > mrolympia.html
you@host > cat mrolympia.html
<head>
<title>
Converted with txt2html.sed
</title>
</head>
<body>
<pre>
Larry Scott USA 1965 1966
Sergio Oliva USA 1967 1968 1969
Arnold Schwarzenegger Österreich 1970 1971 1972 1973 1974 1975
Franco Columbu Argentinien 1976 1981
Chris Dickerson USA 1982
Samir Bannout Libanon 1983
Lee Haney USA 1984 1985 1986 1987 1988 1989 1990 1991
Dorian Yates Großbritannien 1992 1993 1994 1995 1996 1997
Ronnie Coleman USA 1998 1999 2000 2001 2002 2003 2004
</pre>
</body>
</html>
```

Dieses HTML-Dokument können Sie sich jetzt mit einem beliebigen Webbrowser Ihrer Wahl ansehen. In einem Shellscript mit vielen Dateien auf einmal können Sie das Ganze wie folgt realisieren:

```
# Alle Dateien aus der Kommandozeile; alternativ könnte hier auch
# das Metazeichen * verwendet werden, sodass alle Dateien aus dem
# aktuellen Verzeichnis bearbeitet werden
for file in "$@"
do
   sed -f txt2html.sed $file > temp
   # Achtung, hier wird das Original verändert!
   mv tmp $file
done
```


Die Eingabe überschreibt die Ausgabe

An dieser Stelle müssen wir nochmals darauf hinweisen, dass eine Umleitung wie `sed -f script datei > datei` deshalb nicht funktioniert, weil die Eingabe die Ausgabe überschreiben würde.

Jetzt haben Sie noch eine weitere Möglichkeit, `sed` zu verwenden, und zwar als eigenständiges Programm. Hierzu müssen Sie nur die Ausführrechte entsprechend setzen und in der ersten Zeile Folgendes eingeben:

```
#!/usr/bin/sed -f
```

Jetzt können Sie (im Beispiel verwenden wir nochmals das `text2html`-Script) das `sed`-Script wie ein gewöhnliches Shellscript ausführen:

```
you@host > chmod u+x text2html.sed
you@host > ./text2html.sed mrolympia.dat > mrolympia.html
```

Sammlungen von sed-Scripts

Eine interessante Sammlung von `sed`-Scripts (auch von weiteren Dokumentationen, Links etc.) finden Sie unter *http://sed.sourceforge.net/grabbag/*.

Sprungziele mit sed verwenden

`sed` unterstützt übrigens auch Sprungziele (`label`) – und zwar einen unbedingten Sprung (es wird also immer gesprungen). Diesen Hinweis wollten wir geben, falls Sie sich noch intensiver mit den `sed`-Scripts auseinandersetzen wollen.

12.6 Aufgaben

1. Ermitteln Sie mithilfe von `who` und `sed` die Anmeldenamen aller momentan angemeldeten Benutzer.
2. Erstellen Sie mithilfe von `sed` eine Liste aller Einträge mit den Berechtigungen in Ihrem Heimatverzeichnis. Es sollen nur die Rechte und der Dateiname angezeigt werden.
3. Lassen Sie sich mithilfe von `sed` die PID des `sshd` anzeigen.
4. Erstellen Sie ein `sed`-Kommando, mit dem Sie sich alle Kommentare aus einem Shellscript anzeigen lassen können. Denken Sie daran, dass Kommentare sowohl in einer ganzen Zeile stehen können als auch nur am Zeilenende nach dem Shell-Befehl.