Shell-Skript-Programmierung in sh, ksh und bash

2.1 Einführung

Shell-Skript-Programmierung wird heute in erster Linie mit Korn-Shell und Bash betrieben. Während die Korn-Shell im klassischen UNIX-Bereich seit Jahren etabliert ist, wird die Bash von Jahr zu Jahr beliebter und droht der Korn-Shell langsam den Rang abzulaufen. Grund hierfür ist einerseits die Tatsache, dass die Bash auf Linux-Systemen als Standard-Shell eingerichtet ist, was ihr naturgemäß viel Popularität beschert. Andererseits ist sie durch ihren leicht zu bedienenden Kommandozeilen-Editor auch wirklich hervorragend für die tägliche interaktive Arbeit geeignet – weitaus besser als die Korn-Shell.

Was die Skript-Programmierung betrifft, sind die Unterschiede jedoch minimal. Die eine der beiden Shells hat hier einen kleinen Vorteil zu bieten, die andere da. Es gibt für ein Unternehmen kaum einen Grund, bei der Programmierung auf die Bash zu wechseln, wenn bisher mit der Korn-Shell gearbeitet wurde. Anders herum wäre es zwar genauso; da die Korn-Shell aber wesentlich älter ist, stellt sich die Frage in diese Richtung praktisch nicht.

Wenn Sie die freie Wahl haben, sollten Sie die Syntax beider Shells erlernen, denn in Zukunft wird es sicherlich mehr und mehr Projekte im Linux-Bereich geben, die in Bash-Syntax geschrieben sind, andererseits arbeiten die meisten Projekte heutzutage nach wie vor mit der Korn-Shell. Im Übrigen sind die Unterschiede nicht so gravierend, dass man sich darüber Sorgen machen müsste.

Die Syntax der Bourne-Shell bekommen Sie gleich gratis mitgeliefert, denn Korn-Shell und Bash sind Erweiterungen der Bourne-Shell. Letztere ist also quasi in den beiden anderen enthalten. Lediglich die Abgrenzungen müssen Sie sich merken, also welche der verschiedenen Features die Bourne-Shell beherrscht und welche nicht. Die Bourne-Shell ist deshalb interessant, weil sie nach wie vor als Basis-Shell dient. Sie ist die erste Shell, die auf UNIX-Systemen gestartet wird. Boot-Skripte sind deshalb in Bourne-Shell-Syntax geschrieben!

Wir werden in unserem Programmierkurs alle drei Shells parallel behandeln, denn die wichtigsten Konzepte und Techniken sind allen dreien gemeinsam. Erst da, wo die Syntax oder zusätzliche Features dies erfordern, werden wir in eine Beschrei-

bung der individuellen Shells verzweigen. Auf diese Art werden Sie deutlich die Unterschiede zwischen den individuellen Shells erkennen und in *einem* Schwung den Umgang mit allen drei Shells erlernen.

In den folgenden vierzig Abschnitten erwartet Sie ein umfassender, detaillierter und praxisorientierter Kurs in der Kunst der Shell-Skript-Programmierung. Für jeden Befehl und jedes Konstrukt werden Sie ausführliche Erläuterungen und eine detaillierte Beschreibung der Syntax finden. Alle Kommandos und wichtigen Optionen werden in Mini-Beispielen demonstriert, und in jedem Abschnitt erwarten Sie längere konkrete Beispiele, die zeigen, wie man die vorgestellten Methoden in der Praxis einsetzt und wie man mit den dabei auftretenden Schwierigkeiten fertig wird.

Wir wünschen Ihnen viel Freude beim Erlernen der Shell-Skript-Programmierung mit Bourne-Shell, Korn-Shell und Bash.

2.2 Ein erster Streifzug

UNIX-Shell-Skript-Programmierung ist einfach und kompliziert zugleich. Einfach, weil man es mit einer simplen Skriptsprache zu tun hat, mit nur wenigen Befehlen, Variablentypen und Schlüsselwörtern. Kompliziert, weil man in diesem kleinen Repertoire oft vergeblich nach einem passenden Kommando für die gestellte Aufgabe sucht und man daher ständig Umwege gehen und raffinierte Tricks anwenden muss.

Bevor wir uns diesen Details zuwenden, begeben wir uns auf einen kleinen Streifzug durch die wichtigsten Elemente dieser Sprache. Wir werden sehen, wie man Werte in Variablen speichert, UNIX-Befehle aufruft, if-Konstruktionen und Schleifen einsetzt, Daten einliest und einiges mehr. Das Allerwichtigste sozusagen auf einen Blick, um Ihnen einen ersten Überblick zu geben und Sie davor zu bewahren, in den Einzelheiten verloren zu gehen.

- `#!/bin/sh` In der ersten Zeile steht die aufzurufende Shell.
- `mars# ./myscript` Skript im aktuellen Verzeichnis ausführen.
- `befehl >datei` Ausgabe eines Befehls in eine Datei umlenken.
- `befehl >>datei` Ausgabe eines Befehls an eine Datei anhängen.
- `befehl1 | befehl2` Eine Pipe aus zwei Befehlen bilden.
- `file* * ? [ ]` Sonderzeichen im Umgang mit Dateien.
- `alter=47` Werte in Variablen speichern.
- `echo $alter` Auf Variablen zugreifen.
- ``datum=`date` `` Ausgabe von Befehlen in Variablen speichern.

- `if [ bedingung ] ; then befehle ; fi` — if-Verzweigung
- `for var in liste ; do befehle ; done` — for-Schleife
- `while [ bedingung ] ; do befehle ; done` — while-Schleife
- `echo "Name: $name \t Alter: $alter"` — echo: Einfache Ausgabe
- `printf "format" var1 var2 ...` — printf: Formatierte Ausgabe
- `read zeile` — Zeile von Tastatur lesen.
- `read zeile < datei` — Zeile aus Datei lesen.
- `$1, $2, $3` — Argumente, die beim Aufruf übergeben wurden.

Skripte schreiben und ausführen

Wie geht man vor, wenn man sein erstes Shell-Skript schreiben möchte?

Ein Shell-Skript zu schreiben bedeutet, Befehle, die man sonst auf der Kommandozeile eingibt, in einer Datei abzulegen. Die Datei wird abgespeichert und kann anschließend beliebig oft ausgeführt werden. Sie können die Befehle mit irgendeinem Editor schreiben und unter einem selbst gewählten Namen einfach im ASCII-Textformat speichern.

In der ersten Zeile legen Sie über das Kürzel `#!` fest, welche Shell das Skript ausführen soll.

```
#!/bin/sh
echo "Datum:"
date
echo "Rechner:"
hostname
```

Nachdem Sie Ihr Skript abgespeichert haben (hier unter dem Namen `myscript`), müssen Sie ihm noch über `chmod` Ausführungsrechte zuweisen. Anschließend können Sie es wie ein gewöhnliches UNIX-Kommando aus dem aktuellen Verzeichnis heraus aufrufen.

```
mars# chmod 744 myscript
mars# ./myscript
Datum:
Tue Feb 11 16:25:27 MET 2002
Rechner:
mars
```

Befehle umlenken

Die Ausgabe von Befehlen in Shell-Skripten erfolgt normalerweise auf dem Bildschirm. Das ist nett anzusehen, hat aber die unangenehme Eigenschaft des Vergänglichen. Deshalb wollen wir in aller Regel die gewonnenen Daten in einer Datei speichern. Hierzu lenkt man die Ausgabe der Befehle mit einem >-Zeichen in eine Datei um.

```
date > datei
```

Das Gleiche funktioniert auch pauschal für die Ausgabe des gesamten Skriptes.

```
mars# ./myscript >datei
```

Der Inhalt der Datei wird dabei überschrieben. Möchte man ihn bewahren und die neue Ausgabe an das Ende der Datei anhängen, wählt man die Umlenkung >>.

```
echo "Fehler 13: keine Leserechte"  >> datei
```

Nicht nur die Ausgabe, auch die Eingabe für Befehle lässt sich umlenken. Ihre Eingabe beziehen Befehle normalerweise von der Tastatur. Soll stattdessen aus einer Datei gelesen werden, lenkt man die Eingabe mithilfe eines <-Zeichens um.

```
befehl < datei
```

Wenn wir nun die Ausgabe eines ersten Befehls direkt auf die Eingabe eines zweiten Befehls umleiten, erhalten wir eine so genannte Pipe, geschrieben als senkrechter Strich.

```
tail[1] -100 file.log | grep[2] "error"
```

In diesem Beispiel liefert der erste Befehl die letzten 100 Zeilen einer Logdatei; der zweite Befehl sucht aus diesen dann diejenigen heraus, die das Wort "error" enthalten. Die Pipe ist eines der wichtigsten Mittel, um UNIX-Befehle miteinander zu verknüpfen; Sie werden sie in praktisch jedem Ihrer Skripte einsetzen.

Dateien

Wenden wir uns als Nächstes dem Umgang mit Dateien zu. Da sie die Daten enthalten, die wir verarbeiten wollen, tauchen Dateinamen in jeder zweiten Zeile unserer Skripte auf. Konkrete Namen werden einfach hingeschrieben, das ist klar; aber häufig verwenden wir Sonderzeichen * oder ? , um gleich mehrere Dateien auf einen Streich zu verarbeiten. Wie funktioniert das?

1 tail gibt die letzten (hier 100) Zeilen einer Datei aus.
2 grep sucht aus einer Datei oder der Standardeingabe Zeilen heraus, die einen bestimmten String enthalten.

Die Sonderzeichen sorgen dafür, dass die Shell nach allen Dateinamen sucht, die auf das beschriebene Muster passen. Diesen Vorgang nennt man Dateinamenexpansion. Sehen wir uns die einzelnen Metazeichen einmal etwas genauer an.

```
ls file*
grep error file?.log
cat file[a-d].log >log.tmp
```

Das Metazeichen * steht dabei für eine beliebige Zeichenkette, so dass im ersten Beispiel alle Dateien aufgelistet werden, die mit "file" beginnen.

Das Fragezeichen `?` steht für genau ein Zeichen. `grep` sucht also nach "error" in allen Dateien, die nach "file" genau ein Zeichen vor dem folgenden Punkt besitzen, wie etwa `file1.log`, `file2.log` oder `filea.log`.

In eckigen Klammern `[]` kann man einen Bereich von Zeichen angeben. Statt eines beliebigen Zeichens muss deshalb im dritten Beispiel entweder ein a, b, c oder d vor dem Punkt stehen. Alle hierauf passenden Dateien werden aneinander gehängt und nach `log.tmp` geschrieben.

Variablen

Um leistungsfähige Skripte schreiben zu können, müssen wir auch den Umgang mit Variablen beherrschen. Variablen dienen dazu, Werte zu speichern, damit man sie wiederholt verwenden oder weiterverarbeiten kann. Was gilt es hierbei zu beachten?

Erstens: Das Speichern erfolgt über eine einfache Zuweisung.

```
name="Peter Lustig"
alter=47
dir="/etc"
```

Es wird in der Shell zunächst nicht zwischen Zeichenketten und Zahlen unterschieden, schon gar nicht zwischen Zahlen unterschiedlichen Typs. Alle Werte werden als Zeichenketten gespeichert. Enthält eine Zeichenkette Leer- oder Sonderzeichen, muss sie in Anführungszeichen eingeschlossen werden.

Zweitens: Der spätere Zugriff auf die Variablen erfolgt über ein `$`-Zeichen, das vor den Variablennamen geschrieben wird.

```
echo $name
ls -l $dir
```

Drittens: Mit dem Befehl `echo` geben Sie Daten jeglicher Art auf die Standardausgabe aus, das ist normalerweise der Bildschirm. Um einen längeren Text zu produzieren, setzen Sie ihn in Anführungszeichen. Wählt man hierbei Double Quotes, wird das `$`-Zeichen innerhalb der Anführungszeichen erkannt und der Wert der Variablen eingesetzt.

```
echo "Ihr Name lautet: $name. Sie sind $alter Jahre alt."
```

Für den Anfang soll uns dieser Einblick in die Welt der Variablen genügen. In späteren Abschnitten werden wir uns ausführlich damit beschäftigen, wie man den Inhalt von Variablen in einem Skript verarbeiten kann, das heißt, wie man mit Zahlen rechnet und wie man Zeichenketten manipuliert. Ein sehr umfangreiches Thema, wie Sie noch sehen werden.

Kommandosubstitution

Nun wissen wir zwar, wie man fixe Daten wie Zahlen oder Dateinamen in Variablen speichert, aber wie steht es eigentlich mit dynamisch erzeugten Daten? Wenn ich in meinem Skript Befehle wie `date`, `grep` oder `ls` aufrufe, wie kann ich dann deren Ausgabe in einer Variablen speichern? Umlenkungen und Pipes helfen hier nicht! Stattdessen verwendet man eine Technik, die Kommandosubstitution genannt wird. Dabei wird der Befehl in rückwärts gerichteten Anführungszeichen geschrieben.

```
datum=`date`
prozesse=`ps -ef`
anzahl=`ls -l *.log | wc -l`
```

Besonders interessant ist das zweite Beispiel: Eine Variable hat überhaupt keine Probleme damit, mehrzeilige Ausgaben zu speichern! Im weiteren Verlauf des Skriptes können die so gespeicherten Daten dann wie gewohnt weiter verarbeitet werden.

if-Verzweigungen

In unserem ersten Streifzug durch die Shell-Skript-Programmierung wenden wir uns als Nächstes einem Thema zu, das bei *jeder* Programmiersprache eine zentrale Stellung einnimmt: die `if`-Verzweigung. Immer wenn Sie Befehle nur dann ausführen möchten, wenn eine bestimmte Bedingung zutrifft, verwenden Sie eine `if`-Konstruktion.

```
if [ "$name" = "Peter Lustig" ]
  then
  zeilen=`wc -l lustig.dat`
  echo "Ihre Datei enthaelt $zeilen Zeilen."
fi
```

Die zu testende Bedingung wird in eckige Klammern geschrieben. Die Test-Operatoren = != < und > kann man allerdings in gewohnter Form nur für String-Vergleiche verwenden. Möchte man stattdessen Zahlen vergleichen, benötigt man spezielle Operatoren wie z.B. `-gt` für "größer als" oder `-lt` für "kleiner als".

```
if [ $filesize -gt 1000000 ]
  then
  echo "Alarm: Logdatei groesser als 1 MB."
fi
```

Reicht uns eine einfache Verzweigung nicht aus, weil wir mehrere Bedingungen testen müssen, greifen wir auf Verzweigungen wie `elif`, `else` oder `case` zurück. Mehr dazu dann später in den jeweiligen Spezial-Abschnitten.

Schleifen

Auf die Verwendung von Schleifen dürfen Sie sich wirklich freuen! Es ist wie Schokolade essen: Endlich wird man für seine Mühen belohnt und kann sein Ergebnis so richtig genießen. Denn Schleifen stellen sozusagen die Quintessenz der Idee des Shell-Skript-Programmierens dar: einmal schreiben und vielfach anwenden. Schleifen werden immer dann benötigt, wenn eine Reihe von Befehlen *mehrmals* wiederholt werden soll. Die Shell bietet uns hierfür zwei Varianten an, die `for`- und die `while`-Schleife.

Die `for`-Schleife dient dazu, *Listen* abzuarbeiten, also einen Block von Befehlen auf eine *Reihe* von Dateien, Rechnernamen, Benutzer oder andere gleichartige Einheiten anzuwenden.

```
for datei in file1 file2 file3 file4
do
  cp $datei /backup/$datei.old
  ls -l /backup/$datei.old
done
```

Hier wandert jedes Element der Liste sukzessive in die angegebene Variable (hier `datei`), auf welche die Befehle dann innerhalb der Schleife zugreifen können.

Die `for`-Schleife ist ungemein ergiebig – nicht nur in Programmen, sondern auch bei der täglichen Arbeit des Administrators auf der Kommandozeile. Sie hilft Ihnen, Massenarbeiten auf effektive Art zu verrichten, wie allen Benutzern Dateien ins Homeverzeichnis zu kopieren, Rechte gleichermaßen an vielen Dateien zu verändern oder ein Sortiment an Logdateien auf die gleiche Art auszuwerten.

Die `while`-Schleife funktioniert anders. Sie prüft eine Bedingung und führt einen Block von Befehlen solange aus, wie diese Bedingung wahr ist.

```
zahl=1
while [ $zahl -lt 10 ]
do
  cp file$zahl /backup/file$zahl.old
  ls -l /backup/file$zahl.old
  zahl=`expr $zahl + 1`
done
```

Wir lassen also die Variable `zahl` von 1 bis 9 laufen und sprechen die Dateien über einen Namen an, der sich aus "file" und dieser Zahl zusammensetzt. Das Hochzählen um Eins erledigen wir über den `expr`-Befehl (`bash` und `ksh` können rechnen, die `sh` aber leider nicht, die benötigt hierzu `expr`).

Neben den beiden hier erwähnten Schleifen gibt es in der Shell außerdem eine `until`- und eine zum Aufbau von Menüs gedachte `select`-Schleife. Wie man die verschiedenen Schleifentypen bei typischen Problemstellungen einsetzt, erfahren Sie dann in den entsprechenden Abschnitten. Freuen Sie sich darauf!

Eingabe- und Ausgabeoperationen

Das Ausgeben von Daten aus einem Skript heraus geschieht meistens mit Hilfe des `echo`-Befehls.

Möchten Sie die Ausgabe in Spalten schreiben, hilft oft das Einfügen eines Tabulator-Sprungs, ausgedrückt als `\t`. Eine neue Zeile erhalten Sie durch `\n`.

```
mars# echo "Name: $name \t Alter: $alter"
Name: Peter Lustig      Alter: 47
```

Weitergehende Möglichkeiten der Formatierung bietet der Ausgabebefehl `printf`. Er kommt immer dann zum Einsatz, wenn `\t` nicht zum gewünschten Ergebnis führt. Da seine Syntax aber recht aufwendig ist, werden wir ihn erst sehr viel später besprechen.

Möchte man Daten in ein Skript *einlesen*, verwendet man entweder die bereits erwähnte Eingabeumlenkung < oder man benutzt den `read`-Befehl.

```
echo "Geben Sie bitte einen Benutzernamen ein: "
read name
```

Das `read`-Kommando liest eine Zeile von der Tastatur ein und speichert sie in der angegebenen Variable (hier `name`).

Um eine Zeile aus einer *Datei* zu lesen, leitet man den Befehl `read` in diese Datei um.

```
read zeile < datei
```

In Abschnitt 2.24 werden wir sehen, wie man ganze Dateien auf diese Weise ausliest oder den Benutzer so lange Daten von der Tastatur eingeben lässt, bis er z.B. mit "quit" das Ende signalisiert. Außerdem werden wir lernen, Daten gleichzeitig aus mehreren Dateien auszulesen, und und und ...

Argumente verarbeiten

Als Letztes wollen wir in unserem Streifzug darauf eingehen, wie man Argumente verarbeitet, die dem Skript beim Aufruf übergeben werden. Hier ein Aufruf, bei dem zwei solcher Parameter mitgegeben werden:

```
mars# ./myscript montag.dat 300
```

Die übergebenen Argumente, hier der Dateiname `montag.dat` und die Zahl `300`, landen innerhalb des Skriptes automatisch in den Variablen `$1`, `$2`, `$3`, etc. Auf diese Variablen können Sie nun wie gewohnt zugreifen.

```
datei=$1
anzahl=$2
tail -$anzahl $datei | grep error
```

Wir suchen nach dem String "error" in den letzten 300 Zeilen der Datei `montag.dat`. Wie Sie sehen, erhöhen wir die Flexibilität unserer Skripte enorm durch den Einsatz und die Verarbeitung von Argumenten.

2.3 Shell-Skripte schreiben

Im vorangegangenen Abschnitt haben wir einen Überblick über die wichtigsten Elemente der Shell-Skript-Programmierung erhalten, können daher Details besser einordnen und haben uns bereits mit der Denkweise bei der Programmierung vertraut gemacht. Ich hoffe, Sie haben dabei Lust auf mehr bekommen und sind gespannt darauf, wie man wirklich professionelle Skripte gestaltet.

In den folgenden Abschnitten werden wir uns dieses professionelle Know-how Schritt für Schritt systematisch aufbauen, immer ausgerichtet auf die konkrete Anwendung im Alltag. Sollte in Ihren Augen die eine oder andere Fragestellung dabei zu ausführlich behandelt werden, so zögern Sie nicht, sie einfach nur zu überfliegen. Sie können ja jederzeit später noch einmal an die entsprechende Stelle zurückkehren, um die benötigten Details nachzulesen.

Wir beginnen mit der Frage, was es beim Schreiben eines Shell-Skriptes noch alles zu beachten gilt.

- Shell-Skripte sind ausführbare Dateien (oft ASCII, aber zunehmend auch Unicode), die Befehle enthalten.
- Sie können mit einem beliebigen Editor erstellt werden.
- Der Name sollte aus den Zeichen a-Z 0-9 . – und _ zusammengesetzt sein.
- Ein Shell-Skript erhält Ausführungsrechte.
- Es wird wie ein gewöhnlicher UNIX-Befehl aufgerufen.
- Kommentare werden durch # gekennzeichnet.
- Überlange Zeilen bildet man durch ein \ vor dem Zeilenende.
- Einrückungen sind jederzeit erlaubt.

Befehle abspeichern

Ein Shell-Skript zu schreiben, bedeutet schlicht, die Befehle, die man normalerweise auf der Kommandozeile ausführt, in eine Datei zu schreiben. Die Datei wird anschließend unter einem sinnvollen Namen abgespeichert, so dass man sie in Zukunft bequem und beliebig oft aufrufen kann.

```
echo "Datum und Uhrzeit:"
date
echo "Rechnername:"
hostname
```

Listing 2.1: Inhalt einer Skript-Datei

Der geeignete Editor

Sie können Ihre Shell-Skripte mit einem beliebigen Editor erstellen. Geübte Administratoren verwenden oft den berüchtigten `vi`, da er überall zur Verfügung steht und keine grafische Oberfläche benötigt. Je größer das Skript, desto mehr lernt man aber den Komfort zu schätzen, den modernere Editoren wie `xemacs`, `nedit`, `kedit` oder `kwrite` in Sachen Navigation und Übersichtlichkeit mit sich bringen. Zur Not genügt aber auch der einfache Texteditor Ihrer grafischen Oberfläche. Die angesprochenen Editoren finden Sie auf Ihrer UNIX/Linux-CD und im Internet.

Der `emacs` und seine grafische Variante `xemacs` sind wegen ihrer enormen Vielfalt an Features in der UNIX-Welt weit verbreitet. Für Neueinsteiger ist seine Mächtigkeit allerdings eher eine Hürde denn eine Hilfe.

`nedit`, `kedit` und `kwrite` bieten ebenfalls die Möglichkeit der grafischen Navigation, Cut & Paste, Syntax-Highlighting (Schlüsselwörter, Daten, Befehle, etc.

jeweils in einer eigenen Farbe) und einiges mehr. Im Gegensatz zum `emacs` sind hier die Befehle jedoch intuitiver und damit für einen Neuling leichter zu merken.

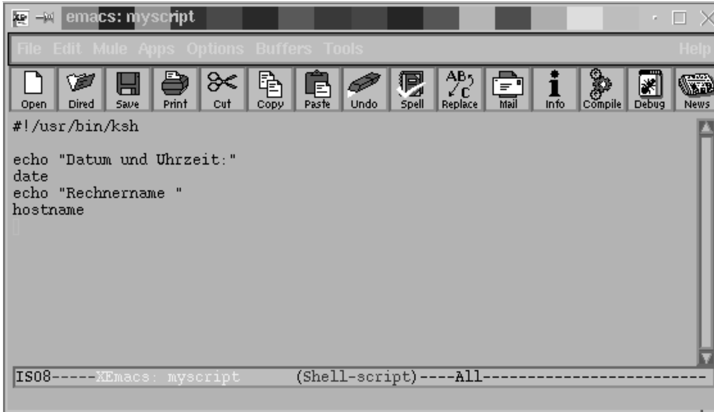

Abb. 2.1: xemacs - ein mächtiges Werkzeug

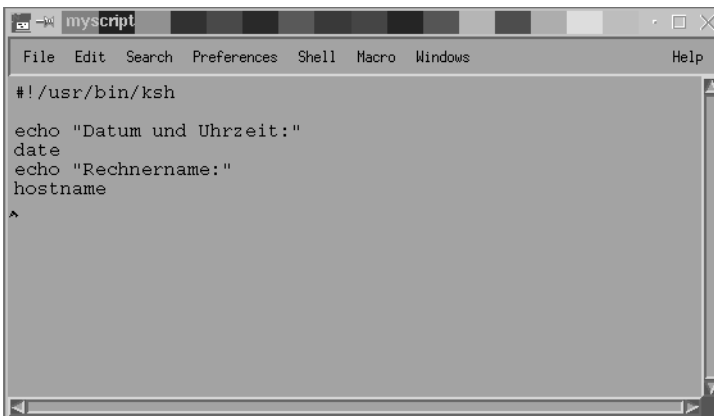

Abb. 2.2: nedit - hervorragend geeignet, um Shell-Skripte zu schreiben

Der richtige Name

Nachdem Sie einen passenden Editor gewählt haben, können Sie Ihr erstes Shell-Skript schreiben. Versuchen Sie es einfach mit den Zeilen, die Sie oben im `nedit` abgebildet sehen. Anschließend speichern Sie Ihr Skript ab.

Was gilt es bezüglich des Namens zu beachten?

- Der Name sollte die Funktion des Skriptes andeuten.
- Verwenden Sie nur die Zeichen A-z 0-9 - _ . und vermeiden Sie Sonderzeichen.

- Nennen Sie Ihr Skript nicht `test` . Es gibt bereits ein Kommando, das so heißt.
- Es sollte keinen gängigen UNIX-Befehl gleichen Namens geben.

Mit den Kommandos `which myscript`, `whence -v myscript`, `type myscript` oder `man myscript` können Sie testen, ob es bereits einen Befehl gleichen Namens gibt.

Das Skript ausführen

Nun erhält das Skript noch Ausführungsrechte.

```
chmod 744 myscript
```

oder

```
chmod u+x myscript
```

Fertig! Führen Sie es aus.

```
mars# ./myscript
Datum und Uhrzeit:
Tue Feb 12 15:19:01 MET 2002
Rechnername:
mars
```

Wie Sie sehen, ist es sehr einfach, kleine Shell-Skripte zum Laufen zu bringen. Wenn wir aber professionelle Skripte schreiben und pflegen wollen, müssen wir genau verstehen, was bei der Ausführung eigentlich geschieht. Welche Shell aktiv wird, wie sie mit den einzelnen Befehlen umgeht usw. Außerdem benötigen wir Testwerkzeuge, die uns beim Aufspüren von Fehlern helfen.

Mit diesen Fragen werden wir uns in einem eigenen Abschnitt gleich im Anschluss beschäftigen. Zunächst wenden wir uns noch einigen Aspekten zu, die beim Schreiben des Skriptes eine Rolle spielen.

Kommentare einfügen

Kommentare fügen Sie mit Hilfe eines Hash-Symbols # ein. Das #-Zeichen kann am Anfang oder mitten in einer Zeile stehen. Alles hinter einem # wird von der Shell ignoriert.

```
# procanz    Gibt die Anzahl der laufenden Prozesse aus
#            Hans Meier  25.07.2001
```

oder

```
ls -l | awk '{print $5}'    # Dateigroesse
```

Es empfiehlt sich, seine Skripte reichlich zu kommentieren. Das spart eine Menge Zeit, wenn man selbst oder die Kollegen es später wiederverwenden oder weiterentwickeln möchten.

Üblicherweise fügt man Kommentare am Anfang des Skriptes hinzu, um den Skriptnamen, den Autor, das Datum der Erstellung, eventuelle Versionsnummern und die Zielsetzung des Skriptes anzugeben. Außerdem sollten Sie logische Abschnitte und schwierige oder trickreiche Stellen kommentieren.

Zeilen richtig formatieren

Sie dürfen mehr als einen Befehl in eine Zeile schreiben. Befehle werden durch ein Semikolon voneinander getrennt. Bedenken Sie jedoch, dass hierdurch die Übersichtlichkeit Ihres Skriptes leidet.

```
echo "Geben Sie den Dateinamen ein: \c" ; read file
```

Einrückungen sind jederzeit erlaubt. Vor allem bei Verzweigungen oder Schleifen dienen Einrückungen der besseren Strukturierung.

```
if [ $1 -gt $zahl ]
   then
   echo "Ihre Zahl ist zu gross."
fi
```

Zeilen können durch einen abschließenden Backslash verlängert werden. Die Shell arbeitet ein Skript Zeile für Zeile ab, aus ihrer Sicht muss jede Zeile eine abgeschlossene Einheit bilden. Diese Einheit würde durch ein Newline (Eingabetaste) in der Mitte des Befehls zerstört werden. Was tut man also bei überlangen Befehlen, die nicht in eine Zeile passen? Entweder man schreibt einfach über das Zeilenende hinaus, was jedoch unmöglich aussieht, oder man nimmt dem *Newline* seine Bedeutung als Zeilentrenner, indem man ihm einen Backslash \ voranstellt. Mit der zweiten Variante erreicht man eine deutlichere Formatierung.

```
ls -l /data/docs/2002/*.txt | \
   awk '{print "Groesse:" $5, "\t", "Name: ", $NF}'
```

2.4 Shell-Skripte ausführen und testen

Wie Sie bereits wissen, ruft man Shell-Skripte wie gewöhnliche UNIX-Befehle auf. Doch welche Shell führt unser Skript eigentlich aus? Wie stellt man sicher, dass es sich um die richtige Shell-Variante handelt? Was heißt es, ein „Korn-Shell-Skript“ zu schreiben? Mit diesen Fragen müssen wir uns nun befassen. Anschließend werden wir einige einfache Methoden kennen lernen, die uns beim Testen der Skripts und bei der Fehlersuche helfen können.

- `./myscript` Aufruf durch Angabe des Namens.
- `ksh myscript` Aufruf durch Übergabe als Argument.
- `. myscript` Öffnen einer Subshell verhindern.
- `#!/bin/sh` Ausführende Shell in der ersten Zeile festlegen.
- `-n` Skript nur auf Syntax prüfen, nicht ausführen.
- `-v` Zeilen vor der Ausführung unverändert anzeigen.
- `-x` Zeilen vorher, doch nach Substitutionen anzeigen.
- `echo $var` Ausgabe von Variablen an interessanten Stellen.
- `exit` Skripte vorzeitig verlassen.

script

Sehen wir uns die einfachste Aufruf-Methode noch einmal näher an. Wir können den Skriptnamen mit absolutem Pfadnamen angeben oder relativ. Das Präfix `./` steht für das aktuelle Verzeichnis.

```
mars# /var/scripts/myscript
```

oder

```
mars# ./myscript
```

oder

```
mars# myscript
```

Geben wir überhaupt keinen Pfad an, wird das Skript in allen Verzeichnissen gesucht, die in der System-Variablen PATH gespeichert sind. Das aktuelle Verzeichnis ist in dieser Liste übrigens nicht automatisch enthalten.

Hinweis

Hinzufügen des aktuellen Verzeichnisses zur Variablen PATH:

`PATH=.:$PATH ; export $PATH`

Aus Sicherheitsgründen sollte man aber besser darauf verzichten, da es sonst leicht zur ungewollten Ausführung fremder Skripten kommen kann.

(Details dazu in meinem Buch auf Seite 340.)

Es ist wichtig zu verstehen, dass die aktuelle Shell, die unseren Befehl entgegennimmt, das Skript nicht selbst ausführt. Stattdessen startet sie eine *neue* Shell, eine so genannte Subshell, die dann ihrerseits das Skript ausführt.

Unterschiede zwischen sh, ksh und bash

Die sh ruft immer auch eine sh als Subshell auf, die bash eine bash und die ksh je nach Version eine ksh oder eine sh. (Testen Sie es einfach durch ein `ps` in ihrem Skript.)

sh script

In der Regel möchte man selbst festlegen, von welcher Shell-Variante ein Skript ausgeführt werden soll. Das Skript ist in der Syntax einer bestimmten Shell-Variante geschrieben, also soll es auch unter dieser Shell laufen. Selbst wenn der Benutzer, der das Skript aufruft, gerade unter der `bash` arbeitet, sollte ein für die `ksh` geschriebenes Skript auch von der `ksh` ausgeführt werden.

Es gibt zwei Wege, den Typ der ausführenden Subshell festzulegen. Der erste – etwas umständlichere – besteht darin, sie direkt aufzurufen und ihr das Skript als Argument zu übergeben. Auf diese Weise ist es auch möglich, Shell-Optionen, beispielsweise für die Fehlersuche (`-x`), mit anzugeben.

```
mars# ksh /var/scripts/myscript
mars# ksh -x /var/scripts/myscript
```

#!/bin/sh

Der zweite – und übliche – Weg besteht darin, die ausführende Shell im Skript selbst festzulegen. Sie muss hierfür mit dem Präfix `#!` versehen und gleich in der ersten Zeile mit vollem Pfad angegeben werden.

```
#!/usr/bin/ksh
echo "Datum und Uhrzeit:"
date
echo "Rechnername:"
hostname
```

Listing 2.2: Skript myscript: Korn-Shell in der ersten Zeile festgelegt

```
mars# ./myscript
```

Damit ist nicht nur die ausführende Shell festgelegt, das Skript ist auch wieder einfacher, nämlich nur über seinen Namen, aufzurufen. Wieso funktioniert das? UNIX-Shells schauen bei einer Datei, die man ihnen zur Ausführung direkt übergibt, grundsätzlich in die erste Zeile. Finden sie dort den Anfang `#!` , starten sie das im Folgenden angegebene Programm und übergeben diesem die Datei als Argument.

. script

Manchmal möchte man das automatische Öffnen einer Subshell verhindern. Um ein Skript direkt von der *aktuellen* Shell verarbeiten zu lassen, benützt man den einfachen Punkt als Befehl. Das Skript übergibt man als Argument.

```
mars# . /var/scripts/myscript
```

oder

```
mars# . ./myscript
```

Ein solches Vorgehen wird nötig, wenn das Skript Veränderungen an der aktuellen Shell vornehmen soll, also z.B. Variablen setzen oder verändern. Ohne den Punkt wären die Veränderungen nur in der Subshell bekannt (mehr hierzu in Abschnitt 2.13.)

Hinweis

Wenn man ein Shell-Skript als Argument übergibt oder durch einen Punkt in der aktuellen Shell ausführt, braucht es keine Ausführungs-, sondern lediglich Leserechte. Jeder Benutzer kann also beliebige Shell-Skripte ausführen, solange er wenigstens Leserechte besitzt.

Testen und Debuggen

Bevor wir mit dem konkreten Programmieren beginnen, widmen wir uns noch kurz einem unangenehmen Thema. Wir werden unweigerlich Fehler in unsere Skripte einbauen und wahrscheinlich viel Zeit mit der Fehlersuche verbringen. Deshalb lohnt es sich, die Hilfsmittel kennen zu lernen, die die Shell hier bietet. Von einem echten Debugger, wie ihn C-Programmierer kennen, kann allerdings keine Rede sein.

```
mars# sh -n myscript
```

Die Shell-Option `-n` (noexec) schaltet die Ausführung ab und testet ausschließlich auf die Korrektheit der Syntax. Dies hilft, wenn man zum Beispiel die richtige Verwendung von geschachtelten Strukturen wie Verzweigungen und Schleifen prüfen möchte.

```
mars# sh -v myscript
```

Die Option `-v` (verbose) führt dazu, dass vor der Ausführung jeder Zeile diese zunächst ausgegeben wird, exakt so wie sie im Skript steht. Um den verbose-Schalter nicht für das ganze Skript, sondern nur für einen bestimmten Teil zu aktivieren, fügen Sie den Befehl `set -v` an der gewünschten Stelle direkt im Skript ein. Mit `set +v` schalten Sie den Modus wieder ab.

```
mars# sh -x myscript
```

Die Option -x (xtrace) arbeitet ähnlich wie -v. Wieder wird jede Zeile vor ihrer Ausführung zunächst ausgegeben. Hier werden jedoch bereits Variablennamen wie `$user` durch konkrete Inhalte (z.B. `otto`) und Sonderzeichen wie `*.dat` durch passende Dateinamen (z.B. `hosts.dat`) ersetzt sowie Kommando-Substitutionen durchgeführt (hierzu mehr in Abschnitt 2.12). Die resultierende Zeile wird dann vom System mit vorangestelltem + ausgegeben und anschließend ausgeführt. Hier folgt ein Beispiel. Die Hinweise rechts wurden von mir zur Verdeutlichung hinzugefügt, sie erscheinen nicht wirklich in der Ausgabe.

```
#!/usr/bin/ksh
# showdbg
echo $LOGNAME
ls -l *.txt
```

```
mars# sh -x showdbg
+ echo root                   <- -x
root                          <- ausgeführt
+ ls -l brief.txt             <- -x    unten: ausgeführt
-rw-r--r--   1 root  root   856 Feb 12 15:31   brief.txt
```

Die Optionen -v und -x können auch gemeinsam verwendet werden: -vx

Wenn unser Skript nicht an Syntaxfehlern krankt, sondern an logischen, läuft es anscheinend fehlerfrei ab, liefert aber falsche Ergebnisse. In diesem Fall ist es oft sinnvoll, sich an kritischen Stellen den Inhalt von Variablen anzusehen.

```
echo $var
```

Komplizierte Zeilen vorübergehend auszukommentieren oder das Programm durch den Befehl `exit` vorzeitig zu verlassen, sind ebenfalls einfache Hilfen zur Fehlersuche.

In Abschnitt 2.37 werden wir uns noch eingehender mit dem Thema Debugging beschäftigen. Für unsere ersten Shell-Skripte werden die vorgestellten Methoden aber sicherlich genügen.

Shell-Skripte beenden

Ein Shell-Skript beendet sich automatisch, nachdem die letzte Zeile ausgeführt worden ist. Andererseits können Sie Ihr Skript auch vorzeitig beenden, wenn wichtige Dateien nicht gefunden werden, Rechte nicht ausreichen oder andere notwendige Bedingungen nicht gegeben sind. Oder, wie gerade erwähnt, zum Testen. Zum vorzeitigen Abbruch des Skriptes benutzen Sie den Befehl `exit`.

```
exit
```

oder

```
exit 1
```

Man kann `exit` eine ganze Zahl von 0 bis 255 als Argument mitgeben, um einen bestimmten Fehler zu signalisieren. `exit 0` heißt "alles OK". `exit 1` bedeutet Fehler Nr. 1 etc. Dieser `exit`-Status kann dann von außen über die Variable `$?` abgefragt werden, doch dazu mehr in Abschnitt 2.17.

2.5 Befehle

Wenden wir uns endlich dem eigentlichen Programmieren zu! Wir haben viel über den Umgang mit Skripten erfahren, wie man sie schreibt, ausführt und testet, doch nun wird es Zeit, sich mit den Befehlen innerhalb der Skripte zu beschäftigen.

- Befehle werden normalerweise nacheinander ausgeführt.
- Jeder Befehl wird zuerst syntaktisch geprüft, dann ausgeführt.
- Hintergrundprozesse werden parallel zu anderen Prozessen ausgeführt.
- Hintergrundprozesse startet man durch Anhängen von & an den Befehl.

Einfache Befehle

Ein Shell-Skript wird Zeile für Zeile von oben nach unten abgearbeitet. Dabei wird jeder Befehl zunächst auf seine syntaktische Korrektheit geprüft und anschließend ausgeführt. Stehen mehrere Befehle mit Semikolon getrennt in einer Zeile, werden auch diese nacheinander ausgeführt.

```
befehl1 ; befehl2
```

Hintergrundprozesse

```
befehl &
```

```
tail -f db.log & [3]
tail -f error.log
```

In der Regel wartet die Shell auf das Ende eines Befehles, bevor sie mit dem nächsten fortfährt. Wird ein Befehl jedoch mit einem nachfolgenden Ampersand (&) gestartet, führt die Shell ihn im Hintergrund aus, das heißt, sie wartet nicht auf sein

3 tail liefert die letzten Zeilen einer Datei. tail -f beobachtet dauerhaft das Dateiende und liefert neu entdeckte Zeilen.

Ende. Dadurch wird in unserem Beispiel der zweite `tail`-Prozess unmittelbar nach dem *Start* des ersten ausgeführt; beide werden also parallel verarbeitet.

Eine weitere Möglichkeit, Prozesse parallel verarbeiten zu lassen, besteht in der Bildung einer Pipe. Diese werden wir im nächsten Abschnitt kennen lernen.

Hinweis

Bei einer Kette von Hintergrundprozessen `befehl1 & befehl2 & ...` dient das &-Zeichen selbst bereits als Trenner, so dass kein zusätzliches Semikolon geschrieben wird.

2.6 Umlenkungen und Pipes

Normalerweise erwartet ein Befehl seine Eingabe von der Tastatur und gibt seine Ausgabe und eventuelle Fehlermeldungen auf dem Bildschirm aus. Diese so genannten Standardkanäle werden dem Befehl beim Aufruf von der Shell automatisch zugewiesen. Häufig möchten Sie die Daten jedoch nicht auf dem Bildschirm bewundern, sondern lieber in einer Datei speichern. Wie Sie dies erreichen und auf welche Details Sie dabei achten müssen, erfahren Sie im folgenden Abschnitt.

- cmd > datei — Umlenkung von stdout in eine Datei.
- cmd >> datei — Umlenkung von stdout an das Ende einer Datei.
- cmd 2> datei — Umlenkung von stderr in eine Datei.
- cmd 2>> datei — Umlenkung von stderr an das Ende einer Datei.
- cmd >datei 2>&1 — Umlenkung von stdout und stderr in die gleiche Datei.
- cmd < datei — Umlenkung von stdin, Eingabe durch eine Datei.
- echo Text — Ausgabe nach stdout.
- echo Text >&2 — Ausgabe nach stderr.
- read var — Einlesen aus stdin.
- { ... } — Befehle in *aktueller* Shell gruppieren.
- (...) — Befehle in *Subshell* gruppieren.
- cmd | tee datei — Ausgabe auf stdout und in Datei lenken.

Der Normalfall

Als Eingabekanal dient die Tastatur. Ausgabe und Fehlermeldungen gehen an den Bildschirm.

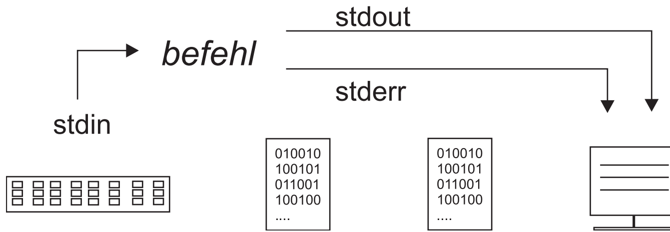

Abb. 2.3: befehl (ohne Umleitung)

Umlenkung der Standardausgabe

```
df -k > filesys.dat   4
myscript > myscript.out
```

Die Standardausgabe (`stdout`) eines Befehls oder eines Skriptes lenken Sie durch ein >-Zeichen in eine Datei um. Die Daten, die Sie ansonsten auf dem Bildschirm sähen, landen stattdessen in der angegebenen Datei. Wenn die Datei bereits existiert, wird ihr bisheriger Inhalt überschrieben. Gab es die Datei noch nicht, wird sie neu angelegt. Achtung: Fehlermeldungen sind von der Umlenkung *nicht* betroffen!

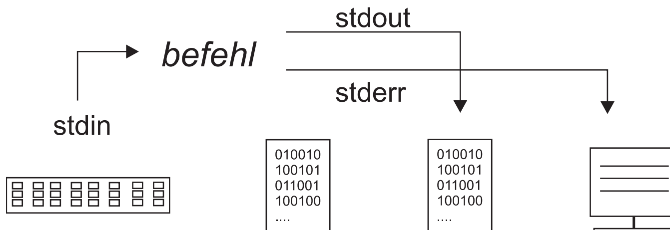

Abb. 2.4: befehl >datei

```
df -k >> filesys.dat
```

Auch durch >> wird die Standardausgabe umgeleitet, dieses mal wird sie jedoch an das Ende der Datei angehängt, so dass deren ursprünglicher Inhalt unversehrt bleibt.

Die Standardausgabe wird auch als Kanal 1 bezeichnet. Wenn Sie selbst etwas auf Kanal 1 ausgeben möchten, verwenden Sie einfach den `echo`-Befehl.

```
echo "Start des Programms"
```

4 df -k listet die Belegung der Partitionen auf.

Umlenkung der Standardfehlerausgabe

```
rm *.txt 2> error.log
```

Durch die Syntax 2> lenken Sie die Standardfehlerausgabe (stderr) eines Befehls oder eines Skriptes in eine Datei um. Im obigen Beispiel werden wir keine Fehlermeldung auf dem Bildschirm finden, wenn z.B. keine Dateien *.txt existieren.

Jedes Programm hat die Möglichkeit, seine Fehler getrennt von den eigentlichen Daten auszugeben. Hierfür öffnet die Shell den Standardfehlerkanal, auch Kanal 2 genannt. Normalerweise zeigt er wie Kanal 1 auf den Bildschirm. 2> datei bedeutet also: Lenke Kanal 2 in datei um. Ebenso wie bei Kanal 1 kann die Ausgabe auch hier an eine bestehende Datei *angehängt* werden.

```
rm *.txt 2>> error.log
```

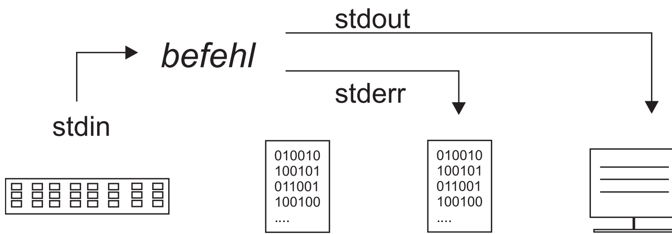

Abb. 2.5: befehl 2>datei

Möchte man störende Fehlermeldungen aus der eigentlichen Datenausgabe herausfiltern, ohne sie ausgeben oder abspeichern zu wollen, kann man sie auf /dev/null umlenken. Daten, die dort landen, werden einfach verworfen.

```
rm *.txt 2> /dev/null
```

Wenn Sie selbst aus ihrem Skript heraus eine Fehlermeldung ausgeben möchten, verwenden Sie den echo-Befehl mit Angabe des Fehler-Kanals.

```
echo "Fehler beim Öffnen der Datei." >&2
```

Wenn Sie Ihr Skript in Zukunft aufrufen, haben Sie dadurch wieder die Möglichkeit, Ihre Fehlermeldungen über 2> von der Ausgabe zu trennen.

Warum heißt Kanal 2 eigentlich umständlich *Standard*fehlerausgabe? Weil Sie über gewöhnliche Umlenkungen natürlich jederzeit Ihre Fehler schreiben können, wohin Sie wollen. Standardmäßig bietet Ihnen UNIX hierfür aber Kanal 2 an :-).

Getrennte und gekoppelte Umlenkung

Wenn Sie sowohl Standardausgabe als auch Standardfehlerausgabe in Dateien umlenken möchten, müssen beide Umlenkungen explizit angegeben werden. Sollen beide Ausgaben in der gleichen Datei landen, verwendet man oft die Abkürzung `2>&1` , was soviel heißt wie: Lenke Kanal 2 auf das gleiche Ziel wie Kanal 1 um.

```
grep meier acc.dat > found.tmp 2> found.err
grep meier acc.dat > found.tmp 2>&1
```

Im ersten Beispiel werden die Ausgaben getrennt, im zweiten gekoppelt umgeleitet.

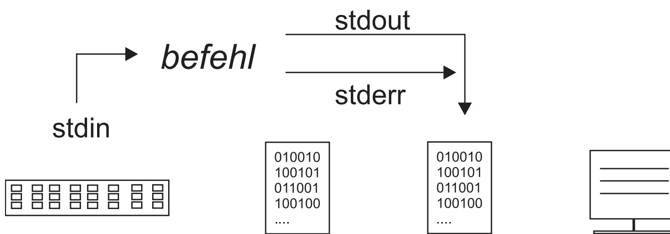

Abb. 2.6: befehl >datei 2>&1

Auf diese Weise bekommen Sie übrigens auch die Programme in den Griff, deren Ausgaben über den Bildschirm rauschen, ohne sich durch den Befehl `more` regulieren zu lassen. Es sind die Ausgaben auf den Fehlerkanal, die `more` nicht zu fassen bekommt! Versuchen Sie statt

```
befehl | more
```

doch einmal

```
befehl 2>&1 | more
```

Umlenkung der Standardeingabe

Auch die Standardeingabe (`stdin`), Kanal 0, kann umgelenkt werden. Hierfür verwenden Sie ein < -Zeichen. Den Text für einen automatischen Alarm, den wir als E-Mail versenden wollen, werden wir sicherlich nicht per Tastatur eingeben. Stattdessen lesen wir ihn aus einer Datei.

```
mail root < alarm.txt
```

Um aus einem Skript heraus von der Standardeingabe zu lesen, verwendet man den Befehl `read`. In Abschnitt 2.24 werden wir mehr darüber erfahren.

```
echo "Geben Sie bitte Ihren Namen an: " ; read name
```

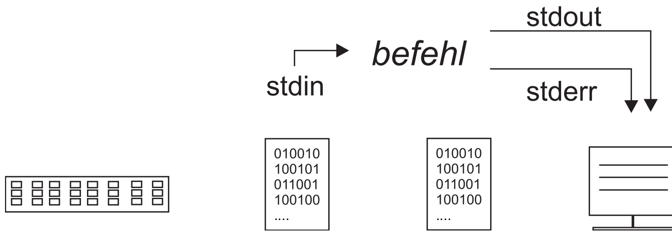

Abb. 2.7: befehl <datei

Alle Umlenkungen auf einen Blick

Kanal	Bezeichnung	Voreinstellung	Umlenkung
0	Standardeingabe	Tastatur	cmd <datei
1	Standardausgabe	Bildschirm	cmd >datei cmd >>datei
2	Standardfehlerausgabe	Bildschirm	cmd 2>datei cmd 2>>datei cmd >datei1 2>datei2 cmd >datei 2>&1

Tabelle 2.1: Umlenkung der Standardkanäle

Die drei Standardkanäle werden von der Shell automatisch geöffnet. Darüber hinaus können wir selbst zusätzliche Kanäle öffnen, so genannte File-Deskriptoren. Da man sie eher selten benötigt, werden wir uns erst in Abschnitt 2.25 mit ihnen beschäftigen. Die Umlenkung << hat ebenfalls eine Sonderbedeutung und wird in Abschnitt 2.24 besprochen.

Pipes

```
tail -f db.log | grep error
```

Pipes entstehen, wenn Sie mehrere Befehle durch ein Pipe-Symbol | miteinander verbinden. Dabei wird `stdout` des ersten mit `stdin` des zweiten Prozesses verbunden. Die Ausgabe des ersten Prozesses wird also direkt vom zweiten gelesen, ohne in einer Datei zwischengespeichert zu werden.

Alle Prozesse in einer Pipe werden parallel verarbeitet.

Hinweis

Eine Pipe leitet ausschließlich die Standard*ausgabe* an den nächsten Befehl weiter. Die Standardfehlerausgabe wird nicht berücksichtigt.

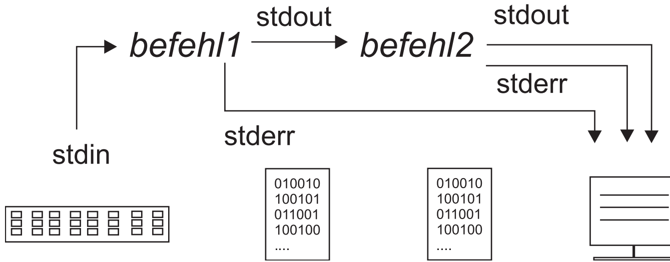

Abb. 2.8: befehl1 | befehl2

In unserem Beispiel "hängt" sich `tail -f` an das Ende der Datei `db.log` und liefert uns alle Zeilen, die dort neu hinein geschrieben werden. `grep` filtert aus dieser Ausgabe Zeilen heraus, die "`error`" enthalten. `tail` läuft so lange, bis wir es mit `ctrl-c` abbrechen, es beendet sich also nicht von selbst. Nur weil beide Prozesse sofort parallel gestartet werden, erhalten wir einen Output.

Pipes können aus beliebig vielen Befehlen gebildet werden.

```
tail -200 db.log | grep error | wc -l
```

Gruppierung

Sollen mehrere Befehle in die gleiche Datei umgelenkt werden, kann man sie zu einer Gruppe zusammenfassen und die gesamte Gruppe umleiten. Dies ist übersichtlicher, als die Umlenkung hinter jeden einzelnen Befehl zu setzen.

```
{
  date
  df -k
  ps -ef
} > status.out
```

oder

```
(
  date
  df -k
  ps -ef
) > status.out
```

Bildet man die Gruppe durch umschließende *geschweifte* Klammern, werden die Befehle von der aktuellen Shell ausgeführt. Benutzt man stattdessen *runde* Klammern, wird eine Kopie der aktuellen Shell als Subshell gestartet, die sich dann um die Befehle kümmert.

Bei Verwendung von runden Klammern können Fehler entstehen, da Veränderungen an Variablen, die in einer Subshell durchgeführt werden, auf die aktuelle Shell keinen Einfluss haben (mehr zum Export von Variablen in Subshells erfahren Sie in Abschnitt 2.13).

Mit dieser Methode können auch die anderen Standardkanäle für ganze Gruppen von Befehlen umgeleitet werden.

Wenn Sie {...} in Einzeilern benutzen möchten, müssen Sie darauf achten, dass hinter der öffnenden Klammer ein Leerzeichen folgt und vor der schließenden Klammer ein Semikolon steht.

```
{ befehl1 ; befehl2 ; } >datei
```

Hinweis

Durch `(...;...)&` oder `{ ...;...; }&` können Sie ganze Gruppen von Befehlen in den Hintergrund stellen.

Output duplizieren mit tee

Manchmal möchte man die Ausgabe eines Befehls sowohl auf dem Bildschirm sehen als auch in einer Datei speichern. Oder man möchte ihn gleich in zwei verschiedene Dateien schreiben, eine Tages- und eine Wochen-Logdatei zum Beispiel. Hierfür gibt es das Kommando `tee` (T-Stück).

```
befehl | tee datei [datei ...]
```

```
df -k | tee status.log
df -k | tee day.log week.log >year.log
df -k | tee -a week.log
```

Im ersten Beispiel erhalten wir Ausgaben auf den Bildschirm und nach `status.log`. Im zweiten erhalten wir ausschließlich Ausgaben in die drei Dateien, da wir die Standardausgabe zusätzlich mit > umlenken. In Beispiel Nr. 3 wird die Ausgabe an `week.log` angehängt (`-a` für *append*), statt es zu überschreiben.

`tee` verarbeitet ausschließlich `stdin`. Um auch Fehlermeldungen zu duplizieren, müssen diese zunächst auf `stdin` umgeleitet werden:

```
grep meier acc.dat 2>&1 | tee found.out
```

Wenn Sie sich nun fragen, wie man die Fehler alleine, ohne die Standardausgabe, an `tee` weitergeben kann, sind Sie auf einen der wenigen Fälle gestoßen, in denen File-Deskriptoren benötigt werden (s. Abschnitt 2.25).

2.7 Dateinamenexpansion

Viele Befehle, die wir in unseren Skripten verwenden, arbeiten mit Dateien. Oft geben wir die gewünschten Dateinamen nicht explizit an, sondern verwenden Platzhalter wie `*.txt`. Es ist wichtig, die genaue Bedeutung dieser Wildcards zu kennen, um nicht versehentlich falsche Dateien zu verarbeiten.

- Die Shell ersetzt Wildcards durch eine Liste von Dateinamen.
- Die Dateinamenexpansion findet *vor* der Ausführung des Befehls statt.
- * steht für beliebig viele beliebige Zeichen.
- ? steht für genau ein beliebiges Zeichen.
- [ ] stehen für genau ein Zeichen aus einer bestimmten Gruppe.
- \ hebt die Sonderbedeutung von Metazeichen auf.

Wildcards

```
grep meier *txt
```

Wenn die Shell bei der Analyse eines Befehls auf eines der Zeichen `* ? []` stößt, interpretiert sie die entsprechende Zeichenkette als Muster für Dateinamen. Sie sucht im aktuellen oder im angegebenen Verzeichnis nach Dateien, deren Namen auf das gewünschte Muster passen, und ersetzt in der Befehlszeile die Zeichenkette durch die Liste der gefundenen Dateien.

Die Dateinamenexpansion erfolgt *vor* der Ausführung des Befehls. Der Befehl bekommt das Metazeichen nicht zu sehen. In unserem Beispiel könnte die Shell die Befehlszeile z.B. folgendermaßen expandieren:

```
grep meier apr.txt feb.txt jan.txt mar.txt
```

Wie Sie sehen, werden die Dateinamen in alphabetischer Reihenfolge geliefert. Das gilt genauso für Dateilisten, die durch `?` oder `[]` gebildet werden. Es ist wichtig zu verstehen, dass es sich hier wirklich nur um Muster für *Dateinamen* handelt. Der Parameter `meier` in unserem Beispiel könnte nicht etwa durch `m*` angegeben werden!

Hinweis

Mit der Debugging-Option `-x` sehen Sie Ihre Befehlszeilen, nachdem die Dateinamenexpansion durchgeführt worden ist.

```
mars# set -x
mars# grep meier *.txt
```

```
+ grep meier apr.txt feb.txt jan.txt mar.txt
feb.txt: 14.02.2002 16:00-17:15 meier
```

* Eine beliebige Zeichenkette

Sehen wir uns die exakte Bedeutung der einzelnen Sonderzeichen einmal genauer an. Das Spezialzeichen * ist die am häufigsten verwendete Wildcard und steht für eine beliebige Anzahl beliebiger Zeichen im Dateinamen. Ausgenommen ist der führende Punkt für versteckte Dateien, den muss man explizit angeben. Die Angabe "beliebig" beinhaltet auch die Option, dass eventuell kein Zeichen passt. Entspricht überhaupt keine Datei dem gewünschten Muster, wird die Zeichenkette unverändert, also inklusive des Sternchens, zurückgegeben.

```
mars# echo *dat
drucker.dat monitore.dat rechner.dat
mars# echo *dat .*dat
drucker.dat monitore.dat rechner.dat .config.dat
mars# echo *.datt
*.datt
```

? Genau ein beliebiges Zeichen

Das Metazeichen ? steht für genau ein beliebiges Zeichen im Dateinamen. Genau eines, nicht mehrere und auch nicht keines! So passt `file?.txt` zwar auf `filea.txt` oder `file2.txt`, nicht aber auf `file.txt`. Wie bei * muss auch hier der führende Punkt für versteckte Dateien explizit angegeben werden. Auch hier wird die Zeichenkette unverändert zurückgegeben, wenn kein passender Dateiname gefunden wird.

```
mars# echo file*.txt
file.txt file1.txt file23.txt filea.txt fileb.txt
mars# echo file?.txt
file1.txt filea.txt fileb.txt
mars# echo ?config.dat
?config.dat
mars# echo .?onfig.dat
.config.dat
```

[ ] Bereich für genau ein Zeichen

Mit Hilfe der eckigen Klammern geben Sie ebenfalls ein Muster für genau ein Zeichen im Dateinamen an. Im Gegensatz zum Fragezeichen darf an der entsprechenden Stelle jedoch nicht ein beliebiges Zeichen stehen; vielmehr muss es aus der Gruppe von Zeichen in den eckigen Klammern stammen.

`file[abx].txt` liefert z.B. `filea.txt`, `fileb.txt` und `filex.txt`, wenn es solche Dateien im aktuellen Verzeichnis gibt. Nicht jedoch `filey.txt`, `file.txt` oder `fileab.txt`.

Auch hier gilt die Ausnahme für den Punkt bei versteckten Dateien, auch hier wird das Muster unverändert gelassen, wenn ihm keine Datei entspricht.

Anstatt alle möglichen Zeichen in den eckigen Klammern aneinander zu reihen, können Sie mit Hilfe eines Minuszeichens auch einen Zeichenbereich angeben, vorausgesetzt, die Zeichen liegen in einem fortlaufenden Bereich der ASCII-Tabelle.

`file[0-9].txt` oder `file[a-zA-Z].txt`

Es ist auch möglich, an der gewünschten Stelle des Dateinamens bestimmte Zeichen auszuschließen, indem man ein `!` an den Beginn der Zeichenklasse setzt:

`file[!ab].txt` liefert Dateien, die auf das angegebene Muster passen, jedoch kein a oder b als fünftes Zeichen besitzen.

```
mars# echo file*.txt
file.txt file1.txt file23.txt filea.txt fileb.txt
mars# echo file[123].txt
file1.txt
mars# echo file[0-9][0-9].txt
file23.txt
mars# echo file[!0-9].txt
filea.txt fileb.txt
```

\ Sonderbedeutung aufheben

Setzt man einen Backslash vor `* ? []` oder `!`, verliert das Zeichen seine Sonderbedeutung und wird wie jedes andere Zeichen behandelt. Das gilt nicht nur für Wildcards, sondern für *alle* Sonderzeichen. Wozu benötigt man eine solche Maßnahme?

Zum einen kann man auf diese Weise Dateien finden, die Metazeichen im Namen enthalten (was unbedingt vermieden werden sollte).

```
mars# echo *dat
drucker.dat monitore.dat rechner.dat scan*.dat
mars# echo scan\*dat
scan*.dat
mars# mv scan\*.dat scanner.dat
```

Andererseits ist es manchmal nötig, Sonderzeichen an Kommandos zu übergeben. Durch \ kann man die Shell daran hindern, das Zeichen zu interpretieren, so dass es überhaupt erst bei dem gewünschten Kommando ankommen kann. Das Sternchen hat zum Beispiel auch für den Befehl `grep` eine Sonderbedeutung, allerdings eine andere als in der Shell. Nur durch ein vorangestelltes \ erhält `grep` den Stern.

Ohne den Backslash erhielte `grep` stattdessen alle Dateinamen aus dem aktuellen Verzeichnis.

```
mars# set -x
mars# grep user.\*02 backup.log
grep user.*02 backup.log
userdat_koeln_02   gesichert   12.01.02
userdat_koeln_02   gesichert   16.01.02
userdat_koeln_02   gesichert   20.01.02
```

Musteralternativen

Korn-Shell und Bash

Nur in Korn-Shell und Bash gibt es die Möglichkeiten, Alternativen für Namens-Muster zu formulieren. Zur Angabe von *Datei*namen werden sie allerdings eher weniger gebraucht. Wir werden solchen Konstruktionen noch einmal in Abschnitt 2.15 begegnen, wo wir sie für Mustervergleiche von Zeichenketten verwenden werden.

Konstruktion	**Bedeutung**
@(*muster1*|*muster2*|...|*mustern*)	genau eines der Muster
+(*muster1*|*muster2*|...|*mustern*)	mindestens eines der Muster
*(*muster1*|*muster2*|...|*mustern*)	keines, eines oder mehrere der Muster
?(*muster1*|*muster2*|...|*mustern*)	keines oder eines der Muster
!(*muster1*|*muster2*|...|*mustern*)	keines der Muster

Tabelle 2.2: Konstruktionen für Musteralternativen in der Korn-Shell

Hinweis

Falls Ihre Bash diese Muster nicht erkennt, ist die erweiterte Option `extglob` ausgeschaltet. Um sie zu aktivieren, gibt man ein: `shopt -s extglob`.

Bash

Die Bash verfügt über eine weitere Form, mit der alternative Muster formuliert werden können:

```
praefix{alt1,alt2,...}suffix
```

Man schreibt die Alternativausdrücke also in geschweifte Klammern und trennt sie mit Komma. Die Shell generiert daraus die Dateinamen `praefixalt1suffix`, `praefixalt2suffix` oder `praefixalt3suffix`, falls diese existieren.

Die Bourne-Shell kennt keine Syntax für Alternativmuster.