mitp

Patrick
Ditchen

4. Auflage

# Shell-Skript Programmierung

Inklusive CD-ROM

Patrick Ditchen

# Shell-Skript-Programmierung

**Bibliografische Information der Deutschen Nationalbibliothek**
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-8266-8356-5

E-Mail: kundenbetreuung@hjr-verlag.de

Telefon: +49 89/2183-7928
Telefax: +49 89/2183-7620

www.mitp.de



Lektorat: Ernst-Heinrich Pröfener
Fachkorrektorat: Knut Lorenzen
Satz: III-satz, Husby, www.drei-satz.de

# Inhaltsverzeichnis


| | | |
|---|---|---|
| | **Über den Autor** | 11 |
| **1** | **Einführung** | 13 |
| 1.1 | Die Shell als User-Interface | 13 |
| 1.2 | Die Shell als Programmiersprache | 13 |
| 1.3 | Einsatzgebiete von Shell-Skripten | 14 |
| 1.4 | Shell-Skripte, Perl und C-Programme | 15 |
| 1.5 | Die Shell und ihre vielen Varianten | 16 |
| 1.6 | LINUX, SOLARIS, HP-UX, AIX, IRIX ... | 17 |
| 1.7 | Aufbau des Buchs | 18 |
| 1.8 | Buch-CD und Web-Server | 19 |
| **2** | **Shell-Skript-Programmierung in sh, ksh und bash** | 21 |
| 2.1 | Einführung | 21 |
| 2.2 | Ein erster Streifzug | 22 |
| 2.3 | Shell-Skripte schreiben | 29 |
| 2.4 | Shell-Skripte ausführen und testen | 33 |
| 2.5 | Befehle | 38 |
| 2.6 | Umlenkungen und Pipes | 39 |
| 2.7 | Dateinamenexpansion | 46 |
| 2.8 | Variablen | 50 |
| 2.9 | Zahlen | 57 |
| 2.10 | Zeichenketten | 63 |
| 2.11 | Arrays | 71 |
| 2.12 | Kommandosubstitution | 75 |
| 2.13 | Variablen exportieren | 78 |
| 2.14 | Die if-Verzweigung | 85 |
| 2.15 | Zahlen- und String-Tests | 89 |
| 2.16 | Dateitests | 98 |
| 2.17 | Kommandotests | 101 |
| 2.18 | Short-Circuit-Tests | 105 |
| 2.19 | Die case-Verzweigung | 107 |

2.20 Die for-Schleife ... 112
2.21 Die while- und die until-Schleife ... 118
2.22 break und continue ... 127
2.23 Ausgabeoperationen ... 133
2.24 Eingabeoperationen ... 139
2.25 File-Deskriptoren ... 148
2.26 Spezielle Umlenkungen ... 154
2.27 Argumente und Optionen ... 157
2.28 Menüs bilden mit select ... 169
2.29 Passworteingabe und Tastatur ... 176
2.30 Funktionen und Aliase ... 180
2.31 Signalverarbeitung mit kill und trap ... 199
2.32 Mehrere Skripte koordinieren ... 209
2.33 Die Laufzeit von Shell-Skripten ... 216
2.34 Startprozedur und Profildateien ... 219
2.35 Wie die Shell Skripte verarbeitet ... 223
2.36 Weitere Befehle und Techniken ... 227
2.37 Debugging-Methoden ... 233
2.38 Shell-Builtin-Befehle ... 240
2.39 Shell-Optionen ... 244
2.40 Shell-Variablen ... 250
2.41 Kommandozeilen-Editor und History ... 258

**3 Shell-Skript-Programmierung in csh und tcsh ... 263**
3.1 Einführung ... 263
3.2 Ein erster Streifzug ... 266
3.3 Shell-Skripte schreiben ... 273
3.4 Shell-Skripte ausführen und testen ... 276
3.5 Befehle ... 280
3.6 Umlenkungen und Pipes ... 282
3.7 Dateinamenexpansion ... 290
3.8 Variablen ... 295
3.9 Zahlen ... 303
3.10 Zeichenketten ... 307
3.11 Arrays ... 315
3.12 Kommandosubstitution ... 321
3.13 Variablen exportieren ... 323
3.14 Die if-Verzweigung ... 328

3.15 Zahlen- und String-Tests ... 331
3.16 Dateitests ... 338
3.17 Kommandotests ... 343
3.18 Short-Circuit-Tests ... 346
3.19 Die switch-Verzweigung ... 348
3.20 Die foreach-Schleife ... 351
3.21 Die while-Schleife ... 357
3.22 Die repeat-Schleife ... 363
3.23 break und continue ... 365
3.24 Ausgabeoperationen ... 370
3.25 Eingabeoperationen ... 375
3.26 Argumente und Optionen ... 382
3.27 Menüs in der C-Shell ... 392
3.28 Passworteingabe und Tastatur ... 394
3.29 Subroutinen, Aliase, Labels ... 397
3.30 Signalverarbeitung ... 407
3.31 Startprozedur und Profildateien ... 411
3.32 Wie die Shell Skripte verarbeitet ... 414
3.33 Weitere Befehle und Techniken ... 414
3.34 Debugging-Methoden ... 420
3.35 Tabellen und Listen ... 425
3.36 History und Kommandozeilen-Editor ... 437

**4 awk** ... 445
4.1 Einführung ... 445
4.2 awk, nawk und gawk ... 445
4.3 Funktionsweise und Aufruf ... 446
4.4 Aufteilen einer Zeile in Felder ... 448
4.5 Field Separator und Record Separator ... 451
4.6 Ausgaben mit print und printf ... 452
4.7 BEGIN- und END-Block ... 453
4.8 Selektionskriterien ... 453
4.9 Der Umgang mit Variablen ... 456
4.10 Zahlen und Zeichenketten ... 459
4.11 Arrays ... 462
4.12 Assoziative Arrays ... 464
4.13 Abrechnungen und Statistiken ... 465
4.14 Mehrdimensionale Arrays ... 469

4.15 Verzweigungen ... 470
4.16 Schleifen ... 472
4.17 Benutzerdefinierte Funktionen ... 474
4.18 Argumente an awk übergeben ... 476
4.19 Umgebung und Ländereinstellungen ... 478
4.20 Ein- und Ausgabetechniken ... 479
4.21 Mehrzeiler und unregelmäßige Zeilen ... 482

**5 Die wichtigsten UNIX-Tools** ... 485
5.1 Einführung ... 485
5.2 Übersicht über alle Kommandos ... 486
5.3 grep und Regular Expressions ... 490
5.4 sed ... 494
5.5 awk ... 497
5.6 Dateiinhalte: head, tail, sort, tr, cut ... ... 497
5.7 Dateioperationen: cp, ln, ls, find, diff ... ... 503
5.8 Verzeichnisoperationen: mkdir, dirname, dircmp ... ... 508
5.9 Archivierung und Backup: tar, cpio, gzip ... ... 509
5.10 Filesystem-Informationen: df, du, fdisk ... ... 516
5.11 Prozessoperationen: ps, kill, prstat, top ... ... 520
5.12 Systeminformationen: uname, date, vmstat ... ... 522
5.13 Benutzerverwaltung: who, finger, last ... ... 524
5.14 Druck- und Druckeradministration: lp, lpstat ... ... 528
5.15 Software-Installation ... 529
5.16 Netzwerkbefehle: rsh, ftp, mail, ping ... ... 530
5.17 Grafiken erstellen: gnuplot ... 535
5.18 Vermischtes: logger, tput, man ... ... 543
5.19 Einplanung von Befehlen: crontab, at ... 546

**6 Shell-Skripte und Logdateien** ... 547
6.1 Shell-Skripte in der Praxis ... 547
6.2 Logdateien: Eine Übersicht ... 548
6.3 Ereignisse in Logdateien zählen ... 549
6.4 Felder in Logdateien aufsummieren ... 555
6.5 Abrechnungen und Statistiken ... 557
6.6 Statistiken über vorgegebene Gruppen ... 563
6.7 Zeit- und andere Klassen ... 567
6.8 Mehrzeiler im Logfile: Verteilte Informationen ... 571

6.9 Mehrzeiler im Logfile: Zeilenumbrüche ... 575
6.10 Zugriff auf eine laufende Logdatei ... 577
6.11 Logdateien komprimieren und archivieren ... 580
6.12 Logfiles blockweise abarbeiten ... 582
6.13 Große Logfiles und Named Pipes ... 585
6.14 Archivieren über Named Pipes ... 587
6.15 Logs auf Festplatten und Hosts verteilen ... 592

**7 Shell-Skripte in der Systemadministration** ... 595
7.1 Einführung ... 595
7.2 Benutzer effektiv anlegen ... 596
7.3 Benutzer kopieren ... 602
7.4 Benutzer intelligent löschen ... 607
7.5 Dateien auf mehrere Rechner kopieren ... 615
7.6 Skripte auf mehreren Rechnern ausführen ... 621
7.7 Software auf mehreren Rechnern installieren ... 629
7.8 Differenzielle Backups ... 634
7.9 Systemdateien überwachen ... 650
7.10 Ein Papierkorb ... 657

**8 Shell-Skripte zur Systemüberwachung** ... 665
8.1 Einführung ... 665
8.2 Proaktives Systemmanagement ... 666
8.3 Schritt für Schritt zur Systemüberwachung ... 667
8.4 Alarme auslösen ... 671
8.5 Grafiken erstellen ... 676
8.6 History- und Trendreports ... 686
8.7 Festplatten und Partitionen ... 695
8.8 Belegung der Filesysteme ... 702
8.9 Kenndaten eines Verzeichnisses ... 717
8.10 Die größten und die neuesten Dateien ... 722
8.11 Memory- und Swap-Verbrauch ... 723
8.12 CPU-Auslastung, Paging und I/O-Performance ... 736
8.13 Anzahl laufender Prozesse ... 742
8.14 Ausgewählte Prozesse beobachten ... 745
8.15 Ressourcenverbrauch von Benutzern ... 749
8.16 Wer war wann eingeloggt? ... 760
8.17 Accounting: Was hat ein Benutzer wann getan? ... 761

8.18 Netzwerke: Erreichbarkeit von Rechnern . . . . . 762
8.19 Netzwerke: Ein HTML-Netzwerkplan . . . . . 771
8.20 Netzwerke: Verfügbarkeit von Diensten . . . . . 778
8.21 Ein zentrales Überwachungs-Interface . . . . . 782
8.22 Ein Web-Überwachungs-Interface . . . . . 789

**Stichwortverzeichnis** . . . . . 811

# Über den Autor

Patrick Ditchen ist seit 1998 als freier Trainer tätig. Er gibt Schulungen auf den Gebieten der UNIX-System-Administration, UNIX-Shell-Skript-Programmierung und Perl. Für Sun Microsystems Deutschland erstellte er die Unterlagen des Perl-Kurses.

Vor seiner Tätigkeit als Kursleiter arbeitete er sechs Jahre lang am Max-Planck-Institut für Psychiatrie in München. Dort baute er als System- und Netzwerk-Administrator ein komplexes heterogenes Netzwerk mit über 300 Rechnern auf, das er in den folgenden Jahren gemeinsam mit seinen Kollegen betreute. Er implementierte Firewall-Funktionalitäten, half bei der Einführung von SAP und wirkte an vielen weiteren Vernetzungsprojekten mit.

Er besitzt ein Diplom als Physiker, wechselte jedoch gleich nach seinem Studium 1992 in die EDV. Bereits während der letzten Studienjahre hielt er an einer MTA-Schule einen selbst geschriebenen EDV-Kurs, eine Mathematik-Vorlesung, deren Unterlagen er sogleich neu gestaltete, und war Leiter eines Physik-Praktikums, das er ebenfalls an die Bedürfnisse der MTA-SchülerInnen anpasste.

In all seinen Lehrtätigkeiten, gleich ob auf Papier oder bei Schulungen, orientiert er sich in erster Linie an praktischen Fragen und der konkreten Anwendung des Lehrstoffes. Er hofft, dass er auch den Charakter des vorliegenden Buches in dieser Richtung prägen konnte.

Hinweise zu den Schulungen finden Sie unter *www.pditchen.de*.

# Einführung

## 1.1 Die Shell als User-Interface

Die UNIX-Shell wurde als Schnittstelle zwischen dem Benutzer auf der einen und dem UNIX-Kernel auf der anderen Seite geschaffen. Sie nimmt die Wünsche des Benutzers in der Form von Kommandos entgegen und reicht sie an den Kernel – in dessen Sprache – weiter. Die Antworten des Kernels präsentiert sie dem Benutzer im Gegenzug wieder in für ihn verständlichen Meldungen.

Den Übersetzer zwischen diesen beiden Welten spielt die Shell. Wie eine Muschel ihre Perle schirmt sie den Kernel vor uns ab. Oder uns vor dem Kernel? Auf jeden Fall ist diese Funktion als User-Interface die eigentliche Aufgabe der Shell.

Die UNIX-Pioniere der ersten Stunde erkannten, dass man die Shell mit gewissen Schlüssel-Funktionalitäten ausstatten muss, um dem Benutzer ein effektives Arbeiten zu ermöglichen. So erhielt sie die Fähigkeit, die Ausgabe beliebiger Kommandos in Dateien umzuleiten ( > ) oder direkt in die Eingabe eines weiteren Befehls zu schleusen ( | ), Resultate für eine gewisse Zeit zu speichern (Variablen) und mit anderen Werten zu vergleichen (`test`) sowie die Ausführung eines Befehls von gewissen Bedingungen abhängig zu machen (`if`) oder ihn wiederholt auszuführen (Schleifen).

Natürlich merkte man auch sofort, dass es sich lohnen kann, Befehle abzuspeichern, um sie am nächsten Tag erneut ausführen zu können. Man hatte aber niemals eine Programmiersprache vor Augen. Hierfür gab es ja C. Es ging immer in erster Linie darum, dem Benutzer ein Interface zur Verfügung zu stellen, über das er bequem und effektiv mit dem System interagieren kann.

## 1.2 Die Shell als Programmiersprache

Die UNIX-Shell ist in erster Linie ein Interface zwischen Benutzer und UNIX-Kernel: ein Kommando-Interpreter. Aber sie bietet auch einige Möglichkeiten und Konstrukte, die einer Programmiersprache nachempfunden sind. Sie taugt zwar nicht dazu, klassische Programmieraufgaben zu lösen wie etwa die Erzeugung schneller Grafiken oder den Aufbau großer Datenbanken. Aber sie ist beinahe unschlagbar, wenn es um Fragestellungen in ihrem angestammten Terrain geht, dem Vereinfachen, Koordinieren und Automatisieren von UNIX-Kommandos.

Die Shell ist hervorragend dazu geeignet, komplexe Ablaufsteuerungen für eine Vielzahl verschiedener Programme und UNIX-Befehle zu realisieren. Mit Leichtigkeit startet sie fremde Programme, leitet Ein- und Ausgaben um, interagiert mit dem Dateisystem und greift in die Prozessverwaltung ein.

Sie startet einen Befehl, mit dem sie den Inhalt eines Verzeichnisses ausliest (`ls`, `find`), und erlaubt uns anschließend, mit jeder einzelnen der gefundenen Dateien eine beliebige Abfolge von Operationen durchzuführen (`for`). Sie ermittelt die Liste aller Benutzer (`grep`), durchstöbert deren Home-Verzeichnisse oder bestimmt das Datum, an dem die Benutzer zum letzten Mal ihre Mail gelesen haben.

Die UNIX-Shell liefert uns eine Art Werkzeugkasten mit einer Menge Nägel und Schrauben, damit wir aus unserem Baumaterial – das sind die vielen UNIX-Kommandos – komplexe und leistungsfähige Steuerungen konstruieren können. Allerdings finden wir auch einige Hämmer und Sägen in der Kiste, also Vorsicht ;-)!

Wir können die unterschiedlichsten Programme starten und Shell-Mittel dazu benutzen, ihre Ein- und Ausgaben zu verarbeiten. Mit Shell-Befehlen stellen wir Abhängigkeiten her, testen Größe und Art von zurückgelieferten Daten und konstruieren Schleifen und raffinierte Steuerungen. Wir speichern wichtige Werte in Variablen und führen einfache Berechnungen durch. Wir ermitteln Parameter des Datei- oder Prozesssystems, interagieren mit dem Benutzer und machen von seinem Verhalten den weiteren Verlauf abhängig.

Die meisten Programmiersprachen wären mit solchen Aufgaben hoffnungslos überfordert. Die Shell ist dazu prädestiniert. Wie Sie derartige Programme schreiben, um komplexe Abläufe zu koordinieren und zu automatisieren, lernen Sie in diesem Buch!

## 1.3 Einsatzgebiete von Shell-Skripten

Die UNIX Shell-Skript-Programmierung hat ihren festen Platz im Alltag der Systemadministration. Hier dient sie dazu, Tätigkeiten zu vereinfachen und Abläufe zu automatisieren. Das bekannteste und vielleicht auch größte Areal beschäftigt sich mit der Überwachung von Servern, Diensten oder ganzen Netzwerken.

"Proaktives Management" ist ein Schlagwort, das die Situation auf diesem Gebiet beleuchtet. Es besagt, dass es sich lohnt, ein umfassendes Überwachungssystem zu etablieren, das es ermöglicht, *vor* dem Eintreten von Fehlern und Katastrophen aktiv zu werden, um nicht von ihnen überrascht zu werden und Ausfallzeiten zu riskieren.

Doch nicht nur in der Systemüberwachung spielen Shell-Skripte eine wichtige Rolle. Die Auswertung von Logdateien gewinnt zunehmend an Bedeutung, da rechtliche Vorschriften und die juristische Absicherung es zunehmend erfordern,

große Mengen an Verbindungs- und Servicedaten mitzuprotokollieren. Sei es nun, dass die Daten ausgewertet werden sollen, in ein vorgeschriebenes Format konvertiert oder einfach nur ausgedünnt werden müssen – fast immer sind es Shell-Skripte, mit denen man solche Arbeiten erledigt.

Neben den großen Anwendungen in der Systemüberwachung und der Log-Datenanalyse finden sich Shell-Skripte in allen Bereichen des Administrations-Umfeldes. Sie dienen als kleine Helferlein, wenn es darum geht, Benutzer einzurichten oder Drucker zu installieren, automatisieren Backups und Archivierungen, fungieren als Mail-Filter und verrichten Aufräumarbeiten. Und nicht zu vergessen: In Form von Start-Skripten machen sie aus einem nackten Kernel ein komplettes UNIX-System.

Auch der gewöhnliche Benutzer kann von Shell-Skripten profitieren, in dem er sie zum Sichern seiner Daten oder zur Automatisierung immer wiederkehrender Dateioperationen verwendet. Er kann sie einsetzen, um Daten zwischen Formaten verschiedener Programme zu konvertieren oder seine Arbeitsumgebung optimal an seine Bedürfnisse anzupassen.

Die Anwendungsmöglichkeiten von Shell-Skripten haben zwar ihren Schwerpunkt auf dem Gebiet der Systemadministration, gehen aber weit über dieses Feld hinaus.

## 1.4 Shell-Skripte, Perl und C-Programme

Wie lassen sich Shell-Skripte gegenüber Perl- und C-Programmen einordnen bzw. abgrenzen? Welche Sprache hat wann die Nase vorn?

Wie bereits dargestellt, besitzen Shell-Skripte den großen Vorteil, unterschiedliche UNIX-Kommandos integrieren und kombinieren zu können. Die UNIX-Tools wiederum sind auf bestimmte Aufgaben spezialisiert. Immer dann, wenn die anvisierte Arbeit zum größten Teil von solchen UNIX-Kommandos erledigt werden kann und es nur noch darum geht, die einzelnen Tools zu koordinieren, werden Sie ein Shell-Skript schreiben.

Allerdings lässt die Performance von Shell-Skripten häufig zu wünschen übrig oder ist einfach inakzeptabel. Dies kann durch sehr große Datenmengen verursacht werden oder weil die Prozedur, der die Daten unterzogen werden, zu aufwendig ist.

Wie wir noch sehen werden, verfügt die Shell an sich über relativ geringe Mittel zur Verarbeitung von Daten. Sie ist darauf angewiesen, externe Programme aufzurufen. Geschieht dies zu häufig, z.B. innerhalb von Schleifen zur Verarbeitung jeder einzelnen Zeile, bricht die Performance dramatisch ein.

Hinzu kommt, dass die Shell ein Programm *interpretiert* und nicht *kompiliert*. Sie analysiert jede einzelne Zeile, checkt die Syntax, führt zahlreiche Ersetzungen durch und startet erst dann die eigentliche Funktion. Das dauert! Zumal in Schleifen.

In diesen Fällen muss das Design des Shell-Skripts gründlich unter die Lupe genommen werden. Müssen die Tools unbedingt innerhalb der Schleife gestartet werden? Entweder kann man das Skript umschreiben oder man weicht auf Perl aus.

Je mehr die Komplexität der Datenverarbeitung in den Vordergrund tritt und der Nutzen spezieller UNIX-Tools abnimmt, desto stärker kommen die Vorteile von Perl zum Tragen. Im Gegensatz zur Shell besitzt es zahlreiche mathematische und String-verarbeitende Funktionen, so dass es keine externen Tools starten muss und daher um Größenordnungen schneller ist.

Auf ein C-Programm müssen Sie immer dann ausweichen, wenn die Verarbeitung von Daten höchsten Performance-Anforderungen genügen soll und gleichzeitig die gesamte Verarbeitung in einem einzigen Programm erfolgen kann. Grafikprogramme sind ein Paradebeispiel für diesen Fall. C-Programme machen hingegen eine schlechte Figur, wenn ein Zusammenspiel mit mehreren UNIX-Tools gefordert ist.

## 1.5 Die Shell und ihre vielen Varianten

Die Mutter aller heute verwendeten Shells ist die Bourne-Shell (sh). Sie wurde relativ früh entwickelt (1978) und etablierte sich schnell als Standard-Shell auf den meisten UNIX-Systemen.

UNIX-Anwender oder -Administratoren, die intensiv mit der Shell arbeiten mussten, bemängelten aber schon bald den fehlenden Komfort der Bourne-Shell, dass sie die tägliche Arbeit nur schlecht unterstütze.

Bill Joy entwickelte daraufhin die C-Shell (csh), der er nur wegen der C-ähnlichen Syntax einiger Konstrukte diesen Namen gab. Er implementierte in ihr folgende Features:

- einen Kommandozeilen-Editor, der es ermöglicht, Tippfehler bequem zu korrigieren
- eine History, über die man bereits verwendete Befehle erneut hervorzaubern kann
- eine Dateinamenvervollständigung, die automatisch teilweise geschriebene Dateinamen ergänzt

- eine Jobkontrolle zur Verwaltung mehrerer parallel laufender Prozesse, ein Alias-System, mit dem komplexe Befehle abgekürzt werden können
- Arithmetische Fähigkeiten, damit zum Rechnen nicht mehr auf den externen `expr`-Befehl zurückgegriffen werden muss.

Das war toll und viele, viele UNIX-Anwender wurden Fans der C-Shell. Leider war sie aber in ihrer Syntax gänzlich inkompatibel zur Bourne-Shell. Man musste sich also zwei Syntax-Varianten merken. Dies blieb so, bis David Korn 1986 seine Shell vorstellte (ksh), die die neuen Features der C-Shell aufgriff und gleichzeitig die Syntax der sh bewahrte. Leider war sie anfangs kostenpflichtig und stand daher lange Zeit nicht auf allen UNIX-Systemen zur Verfügung.

In den neunziger Jahren wurden schließlich zunächst die tcsh[1] als Nachfolgerin der C-Shell und später die Bash als weitere Nachfolgerin der Bourne-Shell entwickelt. Sie unterscheiden sich in erster Linie dahingehend von C-Shell bzw. Korn-Shell, dass der Kommandozeilen-Editor, die History und die Dateinamenvervollständigung nun über die Pfeil- und Tabulatortasten anstatt über kryptische Befehle funktionieren. Seit einigen Jahren gibt es außerdem die zur ksh kompatible zsh.

Wenn Sie Shell-Skript-Programmierung erlernen wollen, sollten Sie sich für die Bourne-Shell-Familie entscheiden, da in C-Shell und tcsh einige wichtige Programmierkonstrukte fehlen und sie deshalb zunehmend an Bedeutung verlieren. Innerhalb der Bourne-Shell-Familie benötigen Sie wahrscheinlich alle drei Varianten, da die Boot-Skripte als sh-Skripte geschrieben sind, die meisten heutigen Projekte in ksh-Syntax vorliegen und die Bash mehr und mehr an Beliebtheit gewinnt. Allerdings sind die Unterschiede zwischen diesen Shells meistens nicht gravierend.

Der Grundlagenteil dieses Buches ist so geschrieben, dass Sie entweder gemeinsam sh, ksh und bash oder gemeinsam csh und tcsh studieren können, wobei die Shells solange wie möglich parallel und erst bei individuellen Unterschieden getrennt behandelt werden. Der komplette Praxisteil ist dann in Korn-Shell-Syntax geschrieben, in der Syntax der am weitesten verbreiteten Shell.

## 1.6 LINUX, SOLARIS, HP-UX, AIX, IRIX ...

Ein Shell-Skript-Programmierer hat nicht nur mit der Vielfalt an Shells zu kämpfen, sondern teilweise auch mit den vielen verschiedenen UNIX-Versionen.

Die Programmierkonzepte, die wir in diesem Buch behandeln, sind natürlich unabhängig von dem zugrunde liegenden UNIX-System, ebenso fast alle Bestandteile der Shell selbst wie die unterschiedlichen Builtin-Befehle, Schlüsselwörter oder Schleifenkonstrukte.

1 sprich ti-ci-shell (Tenex-C-Shell, manche sagen auch Turbo-C-Shell).

Was uns allerdings das Leben schwer macht, sind die externen UNIX-Kommandos. Wenn auf dem einen System `pkgadd` verwendet wird, um Software zu installieren, benutzt das zweite System `rpm` und das dritte `installp`. Teilweise werden zwar die gleichen Befehle verwendet, diese kennen aber unterschiedliche Optionen für dieselbe Aktion, wie etwa `'ps -ef'` und `'ps aux'`.

Für dieses Problem gibt es keine wirklich befriedigende Lösung. Wenn man ein Buch über Shell-Skript-Programmierung schreibt, muss man sich zwangsläufig für ein oder zwei Systeme entscheiden. Es ist nicht möglich, in jedem Skript auf die Unterschiede zwischen den einzelnen UNIX-Varianten einzugehen.

Wir haben uns hier dazu entschlossen, Solaris 2.8 als Betriebssystem zu verwenden. Alle Erläuterungen und Demonstrationen, Tipps und Tricks werden immer an der Solaris-Version vorgenommen und gezeigt. Alle fertigen Skripte wurden aber zusätzlich auf SUSE LINUX 8.0 getestet. Falls Veränderungen notwendig sind, ist dies in den Anwendungskapiteln als Fußnote oder am Ende des jeweiligen Abschnitts vermerkt. Sie finden die Solaris- und die LINUX-Varianten der Skripte getrennt auf der beiliegenden CD.

Wenn Sie mit einem anderen UNIX-Betriebssystem wie z.B. HP-UX, AIX oder IRIX arbeiten, können Sie dennoch beruhigt sein. Gerade im ersten Teil des Buches, in dem die Grundlagen erklärt werden, wurden fast ausschließlich gängige UNIX-Kommandos verwendet. Sie stimmen in der Regel auf allen Systemen überein, zumal die meisten UNIX-Derivate inzwischen System-V-Abkömmlinge sind.

Was nicht übereinstimmt, wird Ihnen wahrscheinlich sofort ins Auge fallen, da Sie diese Befehle selbst häufig benutzen. Sie haben aber wirklich nur an diesen Stellen mit Schwierigkeiten zu rechnen. Die Hinweise für LINUX können da vielleicht auch für Ihr UNIX-Derivat von Nutzen sein. 80 % aller Skripte sollten ohne Veränderungen auf *allen* gängigen Systemen laufen. Bei den restlichen Skripten müssen an wenigen Stellen die entsprechenden UNIX-Kommandos durch Versionen Ihres Systems ersetzt werden.

## 1.7 Aufbau des Buchs

Das Buch ist in einen Grundlagen- und einen Anwendungsteil gegliedert.

Im ersten Teil werden die Grundlagen der Shell-Skript-Programmierung behandelt, getrennt für die Bourne-Shell- und die C-Shell-Familie. Es wird bereits in diesen ersten Kapiteln viel Wert auf einen engen Praxisbezug aller Befehle, Techniken und Konzepte gelegt. Der Lehrgang ist sehr umfangreich und gründlich angelegt, es werden alle interessanten Themen behandelt und viele Details angesprochen.

In Kapitel 4 wird das für die Programmierung wichtigste UNIX-Tool `awk` besprochen. Es erhält ein eigenes Kapitel, da mit seiner Hilfe die meisten Datenauswertungen und Datenmanipulationen realisiert werden.

Kapitel 5 gibt anschließend einen Überblick über etwa 130 weitere UNIX-Kommandos. Die in der Shell-Skript-Programmierung am häufigsten verwendeten Befehle werden teils kurz, teils ausführlich vorgestellt. Die Benutzung wichtiger Tools wie `find`, `sort`, `tar` oder `cpio` wird anhand einer Fülle von kleinen Beispielen demonstriert. `grep` wird vollständig, `sed` immerhin sehr ausführlich behandelt. Außerdem finden Sie eine umfangreiche Einführung in das Grafik-Tool `gnuplot`, mit dessen Hilfe Sie Ihre Messdaten hervorragend grafisch darstellen können.

Der dritte Teil des Buches, Kapitel 6 bis 8, beschäftigt sich mit umfangreichen, realistischen Anwendungen von Shell-Skripten aus dem Alltag der Systemadministration. Mehr als zehn Skripte befassen sich in Kapitel 6 alleine mit unterschiedlichen Aufgaben rund um die Logfile-Analyse, wobei mit einfachen Auswertungen begonnen wird und man schließlich bei der Behandlung von Logdateien landet, die schneller Daten produzieren, als sie die Festplatte speichern kann.

In Kapitel 7 werden eine Reihe von Skripten zu verschiedenen Themen wie Benutzerverwaltung, Archivierung und Backup geschrieben. Wir gönnen uns auch einige Spielereien und schreiben Funktionen für *Direct Access*-Datenbanken und die Behandlung von assoziativen Arrays in der Shell. Interessant ist sicherlich auch unsere Lösung eines Papierkorbsystems, das durch `rm` gelöschte Dateien rettet.

In Kapitel 8 wagen wir uns an das vielleicht komplexeste Administrationsthema heran, die Systemüberwachung. Wir versuchen, viele einzelne Skripte, die die unterschiedlichsten Werte erfassen, in einem gemeinsamen großen Konzept zusammenzufassen. Wir schreiben Module zur Ausgabe von gesammelten Daten im zeitlichen Verlauf, zur Auslösung von Alarmen bei Grenzwertüberschreitungen und zur Anzeige von Daten in flexiblen und leistungsstarken Grafiken.

Der Anwendungsteil endet und gipfelt schließlich in der Erstellung eines Web-Interfaces, über das wir alle unsere Überwachungs-Skripte von einem beliebigen Browser aus anwenden können.

## 1.8 Buch-CD und Web-Server

Die beiliegende CD enthält alle im Buch vorgestellten Skripten in einer Solaris- und einer LINUX-Fassung, sortiert nach Kapitel-Nummer. Wenn die Skripte auf Dateien mit Messdaten zurückgreifen müssen, sind auch diese hinzugefügt. Sie können also parallel zu dem Studium des Buches alle Skripte gleich testen.

Einige Skripte können direkt von der CD gestartet werden, andere benötigen eine bestimmte Verzeichnisstruktur auf der Festplatte und müssen deshalb zunächst an

den richtigen Ort kopiert werden. Informationen hierzu finden Sie immer in den Fußnoten zu den betroffenen Skripten.

Tools wie `gnuplot` oder `netpbm`, die nicht zum eigentlichen Umfang einer UNIX-Distribution gehören, sind ebenfalls auf der CD enthalten. Außerdem einige Protokolldateien, die aus Platzgründen im Buch nur auszugsweise abgedruckt sind.

Wir beabsichtigen auch, einen Web-Server zu pflegen, der Ihnen die Möglichkeit gibt, an die Skripte des Buches zu gelangen, wenn Sie Ihre CD nicht griffbereit haben. Zudem möchten wir dort interessante Skripte aus dem Internet sammeln und Links zu Servern präsentieren, die ihrerseits Shell-Skript-Sammlungen beherbergen. Außerdem ist es die richtige Stelle für Kritik und Anregungen.

Da zu dem Zeitpunkt, da diese Zeilen geschrieben werden, die URL zu unseren Seiten noch nicht feststeht, bleibt uns nur der Hinweis, dass man über

**www.mitp.de**

auf die Hauptseite des Verlages gelangt und von dort aus ohne Schwierigkeiten den Eintrag für unser Buch findet.

Die vielleicht wichtigste Quelle für Shell-Skripte aus dem Internet ist zur Zeit

**www.freshmeat.net**

Ebenfalls nicht schlecht ist

**www.shelldorado.de**

Schulungen zu den Themen Shell-Skript-Programmierung und Perl bietet der Autor unter

**www.pditchen.de**

# 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.