Jetzt lerne ich

Perl

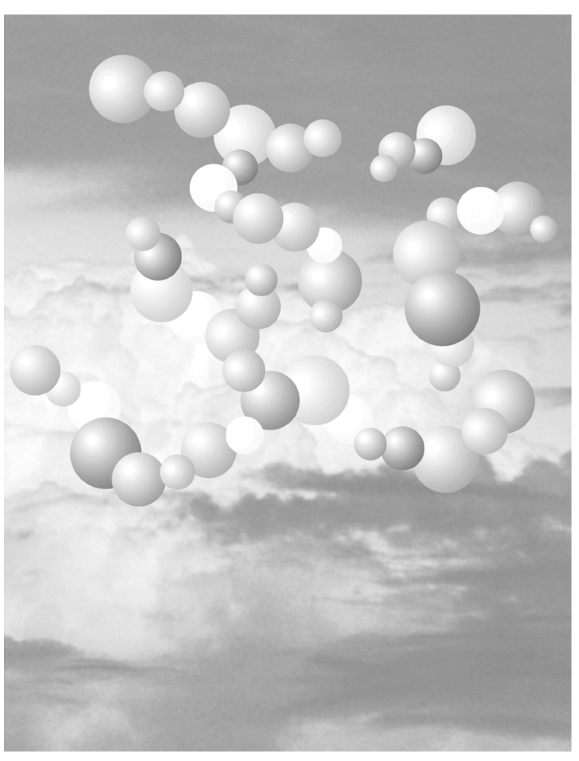
Dirk Louis

eBook

Die nicht autorisierte Weitergabe dieses eBooks an Dritte ist eine Verletzung des Urheberrechts!

Jetzt lerne ich

Perl



Markt+Technik Verlag

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter
Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben
und deren Folgen weder eine juristische Verantwortung noch
irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und
Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der
Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten
ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden,
sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem
und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2

03 02 01

ISBN 3-8272-5841-3

© 2000 by Markt+Technik Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten
Einbandgestaltung: NOWAK werbeagentur & medien, Pfaffenhofen
Lektorat: Erik Franz, efranz@pearson.de
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Satz: text&form, Fürstenfeldbruck
Druck und Verarbeitung: Media-Print, Paderborn
Printed in Germany

Übersicht

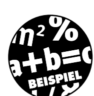
Vorwort 13
Kapitel 1: Von der Installation zum ersten eigenen Programm 15

Teil I: Die Grundlagen 39
Kapitel 2: Daten 41
Kapitel 3: Programmieren mit Zahlen und Strings 75
Kapitel 4: Steuerung des Programmflusses 97
Kapitel 5: Listen, Arrays und Hashes 121
Kapitel 6: Funktionen 153

Teil II: Für Fortgeschrittene 181
Kapitel 7: Referenzen 183
Kapitel 8: Kontext und Standardvariablen 195
Kapitel 9: Ein- und Ausgabe 205
Kapitel 10: Suchen mit regulären Ausdrücken 229
Kapitel 11: Objektorientierte Programmierung in Perl 247
Kapitel 12: Grafische Oberflächen 263

Teil III: Praxis 281
Kapitel 13: Grafik: Diagramme erstellen 283
Kapitel 14: Datenbank: CDs verwalten 295
Kapitel 15: Internet: Informationen aus Webseiten zusammenstellen 303
Kapitel 16: CGI: Perl-Programme für Websites 307
Kapitel 17: CGI: Ein Gästebuch 317

Teil IV: Anhang 329
Anhang A: Synopsis 331
Anhang B: Module 349
Anhang C: Online-Hilfe 353
Anhang D: Der Debugger 357
Anhang E: Lösungen zu den Übungen 359
Anhang F: ASCII-Tabelle 385
Anhang G: Webadressen 387
Stichwortverzeichnis 389



Inhaltsverzeichnis

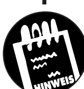
Vorwort 13

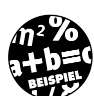
1 Von der Installation zum ersten eigenen Programm 15
1.1 Bezugsquellen für Perl 16
1.2 Installation unter Unix/Linux 20
1.3 Installation unter Windows 23
1.4 Die Installation testen 23
1.5 Ein erstes Perl-Skript aufsetzen und ausführen 25
1.6 Interpreter versus Compiler 30
1.7 Bestandteile eines Perl-Skripts 34

Teil I: Die Grundlagen 39

2 Daten 41
2.1 Kleine Datenkunde 42
2.2 Konstanten 44
2.3 Variablen 49
2.4 Anweisungen und Ausdrücke 55
2.5 Ein- und Ausgabe von Daten 57
2.6 Vertiefung : Zahlen und Strings – ein ungleiches Geschwisterpaar 62
2.7 Vertiefung: Binärdarstellung von Daten 66
2.8 Fragen und Übungen 73

3 Programmieren mit Zahlen und Strings 75
3.1 Der Zuweisungsoperator 76
3.2 Arithmetischen Operatoren und mathematische Funktionen 76
3.2.1 Die arithmetischen Operatoren 77
3.2.2 Die mathematischen Funktionen 78
3.2.3 Die Gleitkommaproblematik 80
3.3 Inkrement, Dekrement und kombinierte Zuweisung 82
3.4 String-Operatoren und String-Funktionen 85
3.4.1 Strings definieren 85
3.4.2 String-Operatoren 87





3.4.3 String-Funktionen 88
3.4.4 Reguläre Ausdrücke 91
3.5 Vertiefung: Auswertung von Ausdrücken 92
3.5.1 Operatorenrangfolge 92
3.5.2 Assoziativität 93
3.5.3 Seiteneffekte 94
3.6 Fragen und Übungen 95

4 Steuerung des Programmflusses 97
4.1 Die if-Bedingung 98
4.2 Wie formuliert man Bedingungen? 100
4.2.1 Arithmetische Ausdrücke in Bedingungen 101
4.2.2 Die Vergleichsoperatoren 101
4.2.3 Die logischen Verknüpfungen 104
4.3 Verzweigungen 105
4.4 Die Schleifen 107
4.4.1 Die while-Schleife 107
4.4.2 Die do-while-Schleife 108
4.4.3 Die for-Schleife 110
4.4.4 Die foreach-Schleife 112
4.5 Abbruchbefehle für Schleifen 112
4.6 Vertiefung: Abkürzungen 114
4.7 Vertiefung: Fallstricke 116
4.8 Fragen und Übungen 118

5 Listen, Arrays und Hashes 121
5.1 Listen 122
5.1.1 Listendefinition 122
5.1.2 Grenzen des Machbaren 123
5.1.3 Furby 124
5.2 Arrays 125
5.2.1 Arrays definieren 125
5.2.2 Mit Arrays arbeiten 126
5.2.3 Elemente hinzufügen oder entfernen 130
5.2.4 Arrays sortieren und durchsuchen 132
5.2.5 Arrays und Strings 139
5.3 Hashes 143
5.3.1 Hashes definieren 143
5.3.2 Mit Hashes arbeiten 144

5.3.3 Wörter zählen 148
5.4 Vertiefung: kombinierte Datenstrukturen 150
5.5 Fragen und Übungen 151

6 Funktionen 153
6.1 Wofür braucht man Funktionen? 154
6.2 Funktionen definieren und aufrufen 157
6.2.1 Funktionen definieren 157
6.2.2 Funktionen aufrufen 158
6.3 Funktionen mit Parametern 158
6.3.1 Allgemein verwendbare Funktionen 159
6.3.2 Argumente und Parameter 160
6.3.3 Listen, Arrays und Hashes als Argumente 162
6.4 Funktionen mit lokalen Variablen 164
6.4.1 Das Schlüsselwort my 167
6.4.2 Unveränderliche Parameter 168
6.5 Funktionen mit Rückgabewerten 168
6.6 Vertiefung: Packages, my und strict 172
6.7 Vertiefung: eigene Module erstellen 177
6.8 Fragen und Übungen 180

Teil II: Für Fortgeschrittene 181

7 Referenzen 183
7.1 Was sind Referenzen? 183
7.2 Wie deklariert man Referenzen? 185
7.3 Wie greift man auf Referenzen zu? 186
7.4 Wofür braucht man Referenzen? 188
7.4.1 Komplexe Datenstrukturen 188
7.4.2 Übergabe von Referenzargumenten an Funktionen 190
7.4.3 Dynamischer Aufbau von Arrays 191
7.5 Fragen und Übungen 193

8 Kontext und Standardvariablen 195
8.1 Kontext 196
8.1.1 Kontext in Perl 196
8.1.2 Wie kann man einen Kontext ändern? 198
8.1.3 Wie implementiert man kontextsensitive Funktionen? 199
8.2 Standardvariablen 201
8.3 Fragen und Übungen 204

9 Ein- und Ausgabe 205
9.1 Ein- und Ausgabe 206
9.1.1 Der Eingabeoperator <> 206
9.1.2 Ausgabe mit print 211
9.1.3 Der Report-Generator write 213
9.2 Dateien 218
9.2.1 Daten in Dateien schreiben 218
9.2.2 Daten aus Dateien einlesen 223
9.2.3 Binärmodus und Binärdateien 225
9.3 Kommandozeilenargumente 227
9.4 Umgebungsvariablen 227
9.5 Fragen und Übungen 228

10 Suchen mit regulären Ausdrücken 229
10.1 Nach Wörtern suchen 230
10.2 Mit Mustern suchen 235
10.3 Suchen und Ersetzen 239
10.4 Gefundenen Text weiter verarbeiten 243
10.5 Fragen und Übungen 245

11 Objektorientierte Programmierung in Perl 247
11.1 Perl und OOP 248
11.2 Grundlagen der objektorientierten Programmierung 250
11.3 Objektorientierte Programmierung in Perl 256
11.4 Fragen und Übungen 261

12 Grafische Oberflächen 263
12.1 Grundlagen und ein erstes Tk-Programm 264
12.2 Furby 270
12.3 Eingabemasken für Programme 277
12.4 Fragen und Übungen 280

Teil III: Praxis 281

13 Grafik: Diagramme erstellen 283

14 Datenbank: CDs verwalten 295

15 Internet: Informationen aus Webseiten zusammenstellen 303

16 CGI: Perl-Programme für Websites 307

17 CGI: Ein Gästebuch 317

Teil IV: Anhang 329

Anhang A: Synopsis 331

Anhang B: Module 349

Anhang C: Online-Hilfe 353

Anhang D: Der Debugger 357

Anhang E: Lösungen zu den Übungen 359

Anhang F: ASCII-Tabelle 385

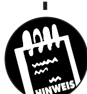
Anhang G: Webadressen 387

Stichwortverzeichnis 389

Vorwort

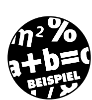
Noch vor wenigen Jahren war Perl eine relativ unbekannte Programmiersprache die hauptsächlich, ja eigentlich ausschließlich, von Systemadministratoren der verschiedenen Unix-Plattformen genutzt und geschätzt wurde. Doch diese Situation hat sich mittlerweile grundlegend geändert.

Durch die Entwicklung des World Wide Web hat Perl einen unglaublichen Boom erlebt. Viele dynamische Webinhalte (Online-Bestellsysteme, Zugriffszähler, Gästebücher etc.) bedürfen der Unterstützung durch serverseitig installierte Skripten. Ein Großteil dieser Skripten ist und wird in Perl aufgesetzt.



Einen weiteren Schub löste die zunehmende Verbreitung des Betriebssystems Linux und die Portierung von Perl auf die Windows-Plattform aus, wodurch die Perl-Gemeinde sprunghaft anstieg. Diese Entwicklung förderte wiederum die Aufstockung des CPAN, einer frei verfügbaren Bibliothek von Perl-Modulen zu praktisch allen denkbaren Programmieraufgaben.

Aber auch Perl selbst hat sich weiterentwickelt. Neue Konzepte (Referenzen, objektorientierte Programmierung, Threads etc.) wurden in die Sprache integriert, so dass Perl den Vergleich zu Programmiersprachen wie C++ oder Java nicht scheuen muss.



Geblieben sind die ursprünglichen Stärken der Sprache. Perl ist unschlagbar, wenn es darum geht, Daten aus Texten und Dateien zu extrahieren, auszuwerten und neu aufbereitet auszugeben. Probleme aus diesem Aufgabenbereich, für deren Lösung man in C++ oder Java mehrere Tage konzentrierter Programmierarbeit und intensiven Nachdenkens investieren muss, lassen sich in Perl häufig in wenigen Minuten und mit wenigen Zeilen Code lösen.

Der Leser darf also gespannt sein.

Saarbrücken, im Juni 2000 Dirk Louis

Kapitel 1

Von der Installation zum ersten eigenen Programm

Nachdem ich Sie im Vorwort bereits ein wenig mit Perl und den Vorzügen dieser außerordentlichen Programmiersprache bekannt gemacht habe, sind Sie nun sicherlich gespannt darauf, Ihr erstes eigenes Perl-Programm zu erstellen. Ich will Sie daher auch gar nicht weiter mit irgendwelchen Vorreden plagen, sondern gleich zur Beschreibung der Installation von Perl und der Erstellung eines kleinen Testprogramms übergehen.

Im Einzelnen erfahren Sie in diesem Kapitel,

- ✘ von wo man die aktuelle Version von Perl herunterladen kann
- ✘ wie man Perl unter Unix/Linux oder Windows installiert
- ✘ wie man ein kleines Testprogramm erstellt und ausführt
- ✘ worin die Unterschiede zwischen interpretierten und kompilierten Programmiersprachen liegen
- ✘ aus welchen typischen Elementen Perl-Skripten aufgebaut werden

Dabei werden Sie folgende Programmierwerkzeuge näher kennen lernen

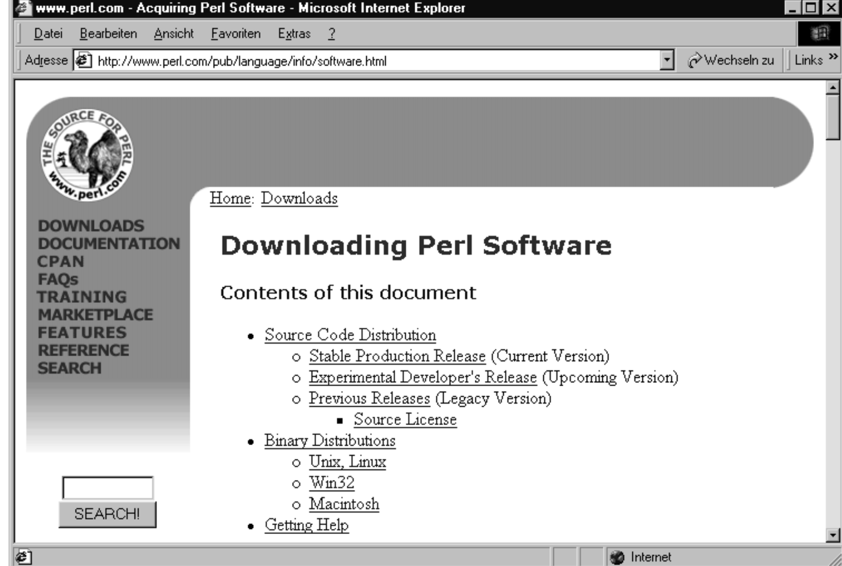
Den Perl-Interpreter perl sowie die wichtigsten Skriptelemente: die Shebang-Zeile, Kommentare, Anweisungen, Symbole, Bezeichner, Whitespace.

1.1 Bezugsquellen für Perl

www.perl.com ist die erste Anlaufstation bei Fragen zu Perl

Die jeweils aktuelle Version von Perl kann man kostenlos vom Perl-Server *www.perl.com* herunterladen. Ein Klick auf den Link DOWNLOADS führt Sie zum Download-Center, wo Sie zwischen verschiedenen Versionen wählen können (siehe Abbildung 1.1).

Abb. 1.1: Perl aus dem Internet herunterladen[1]



Binäre Versionen

Als binäre Versionen bezeichnet man die fertigen Programme (im Gegensatz zu den Quelltext-Versionen, die man zuerst kompilieren muss, siehe nachfolgenden Abschnitt).

Die Binärversionen haben den Vorteil, dass sie einfach zu installieren und danach direkt einsatzfähig sind. Wenn Sie Perl als Binärversion herunterladen wollen, klicken Sie unter »Binary Distributions« auf den zu Ihrem Betriebssystem passenden Link und folgen den weiteren Anweisungen.

---

1 Nichts ist von Dauer – diese Weisheit gilt besonders für Webseiten. Entschuldigen Sie daher bitte, falls die abgebildete Perl-Seite nicht mehr mit der tatsächlichen Perl-Seite übereinstimmten sollte.

Der Link für Windows (Win32) führt Sie auf einem kleinen Umweg zur Website von ActiveState, von wo Sie ActivePerl – die Windows-Version von Perl – herunterladen können. Wenn Sie dem Download-Link[2] folgen, gelangen Sie zu einer Website mit Links für verschiedene Plattformen, wo Sie die Kombination Windows/Intel auswählen. ActivePerl wird daraufhin heruntergeladen. Speichern Sie es in einem beliebigen Verzeichnis.

Sie können ActiveState auch direkt ansteuern: www.activestate.com

Sofern Sie nicht bereits über Windows 2000 verfügen, ist es damit allerdings noch nicht getan. Die heruntergeladene Datei trägt die Extension .msi und kann nur mit dem neuen Microsoft-Installationssystem entpackt werden. Um das neue Installationssystem auf Ihrem Rechner einzurichten, laden Sie es über den angebotenen Link auf Ihre Festplatte herunter und doppelklicken dann im Windows Explorer auf die gespeicherte EXE-Datei (INSTMSI.EXE). Windows-95-Anwender müssen zudem noch die Bibliotheksdatei MSVCRT.DLL installieren – wofür es wieder einen passenden Link auf der ActiveState-Seite gibt.

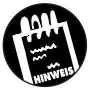
Linux- und Unix-Anwender können sich das Herunterladen unter Umständen ersparen. Sofern Sie eine relativ aktuelle Linux-Version auf Ihrem Rechner installiert haben oder in einem Netzwerkverbund arbeiten, das von einem engagierten Netzwerkadministrator verwaltet wird, haben Sie gute Chancen, dass eine der neueren Perl-Versionen, vielleicht sogar die aktuelle Version, bereits auf Ihrem System installiert ist.

Perl-Version feststellen

Um festzustellen, ob Perl auf Ihrem System installiert ist, öffnen Sie ein Konsolenfenster und geben am Prompt **perl -v** ein. Wenn Perl auf Ihrem System installiert ist, wird Ihnen daraufhin die Kennnummer der installierten Version angezeigt. Zum Zeitpunkt der Drucklegung dieses Buches war 5.6.0 aktuell. Jede Version, deren Kennnummer mit 5 beginnt, sollte aber für den Einstieg vollkommen ausreichend sein. Sie können sich dann erst einmal mit der vorhandenen Version in die Perl-Programmierung einarbeiten und die aktuelle Version später nachinstallieren.

Wenn Sie die aktuelle Perl-Version herunterladen wollen, klicken Sie zuerst auf den Link UNIX, LINUX und danach auf LIST OF PORTS. In der daraufhin angezeigten Liste von Betriebssystemen wählen Sie den Link für Ihr Betriebssystem. Vermutlich wird Sie dies letzten Endes aber zur Website von ActiveState führen, die Sie auch direkt anwählen können: *www.activestate.com/ActivePerl*. Über den Link GET ACTIVEPERL gelangen Sie dann zu einer Liste von Download-Links.

2 Beachten Sie, dass sich die Namen der Links und die Abfolge der Webseiten wegen der Schnelllebigkeit des World Wide Web geändert haben können.



Als Alternative zur ActivePerl-Version können Sie auch auf die Webseite Ihres Linux-Vertreibers gehen und dort nach einem binären Perl-Download Ausschau halten.

Quellcode-Versionen

Kommerzielle Programme werden meist ausschließlich als Binärversion (ausführbare EXE-Datei) ausgeliefert, Perl dagegen kann man auch als reinen Quellcode herunterladen (Links unter dem Punkt »SOURCE CODE DISTRIBUTION«).

Wenn Sie den Quellcode herunterladen, müssen Sie diesen allerdings zuerst auf Ihrem Rechner mit Hilfe eines geeigneten C-Compilers kompilieren und in eine ausführbare Datei übersetzen, bevor Sie mit Perl arbeiten können. Wann sollte man diese Mühe auf sich nehmen?

- ✘ **Die angebotenen Binärversionen sind auf Ihrem System nicht lauffähig**. Ausführbare Dateien wie die Perl-Binärversionen bestehen aus Maschinencode, d.h. aus einer Abfolge von binär codierten Befehlen, wie Sie der Rechner, genauer gesagt der Prozessor, verarbeiten kann. Das Problem besteht nun darin, dass die verschiedenen Prozessoren (genauer gesagt die Prozessorfamilien, beispielsweise die Familie der Intel-kompatiblen Prozessoren) unterschiedliche Maschinenbefehlssätze definieren. Die Folge ist, dass ausführbare Dateien immer nur auf dem Prozessor ausgeführt werden können, für dessen Befehlssatz sie erstellt wurden. Aus diesem Grund können Sie Microsoft Word nur auf Ihrem Windows-Rechner mit Intel-kompatiblem Prozessor ausführen, und aus diesem Grund müssen Sie auch beim Herunterladen einer Perl-Binärversion die gewünschte Plattform (Betriebssystem/Prozessorkombination) auswählen.

  Was macht man aber, wenn man feststellen muss, dass auf der Perl-Website keine passende Perl-Version für die eigene Rechner-Betriebssystem-Kombination angeboten wird?

  In diesem Fall lädt man die Quellcode-Version herunter, die aus reinem ASCII-Text besteht und daher absolut plattformunspezifisch ist. Diesen Programmquelltext muss man nun mit Hilfe eines speziellen Software-Tools, einem Compiler, in Maschinencode übersetzen. Aber Achtung! Der Programmcode der Perl-Werkzeuge ist nicht in Perl geschrieben, sondern in der Programmiersprache C. Man benötigt daher einen C-Compiler, der C-Quelltext in den Maschinencode des eigenen Rechners umwandelt. Unix/Linux-Anwender können hier davon profitieren, dass

ihre Betriebssysteme fast immer zusammen mit einer Version des GNU-C/C++-Compilers installiert werden. Windows-Anwender müssen sich dagegen selbst um einen geeigneten C- (oder C++-)Compiler bemühen[3].

- **Sie wollen Perl selbst anpassen**. Wer den Quelltext der Perl-Programmierwerkzeuge vorliegen hat, kann diesen Quelltext vor der Kompilation überarbeiten und eigenen Bedürfnissen anpassen. Voraussetzung hierfür sind allerdings gute C-Kenntnisse (die Perl-Programmierwerkzeuge wurden in C programmiert) und Verständnis für die Arbeitsweise der Programmierwerkzeuge. Fehler machen ist aber erlaubt. Schlimmstenfalls lädt man sich noch einmal eine lauffähige Binärversion aus dem Internet herunter.
- **Sie sind an der Entwickler-Version von Perl (»Developers Release«) interessiert**. Wenn Sie diesen Quellcode herunterladen, können Sie sich bereits vorab mit den Neuerungen der kommenden Version vertraut machen und sogar selbst an der Weiterentwicklung von Perl mitwirken.

Binärcode schützt kommerzielle Software

Warum werden kommerzielle Programme meist nur als ausführbare Dateien (Binärversionen) vertrieben?

Zur Beantwortung dieser Frage muss man unterscheiden, was ein Programm macht und wie das Programm macht, was es macht. Der normale Anwender ist allein daran interessiert, was das Programm macht, was es leistet. Darum kauft er sich das fertige, ausführbare Programm (oder lässt sich erst einmal eine Demoversion zustellen). Der Industriespion ist dagegen daran interessiert, wie das Programm arbeitet. Vielleicht ist er auf ein Textverarbeitungsprogramm mit einer besonders guten Rechtschreibkorrektur gestoßen und möchte herausfinden, wie diese Rechtschreibkorrektur implementiert wurde. Diese Information kann er kaum dem binären Code des Programms, wohl aber dem Quelltext entnehmen. Genau aus diesem Grund schützen kommerzielle Software-Entwickler ihre Programmquelltexte wie Staatsgeheimnisse.

Man kann dies mit einem Parfumhersteller vergleichen, der sein fertig gemischtes Parfum frei verkauft, während er die Formel für das Mischungsverhältnis der Inhaltsstoffe sicher in einem Safe verwahrt. Ein wenig hinkt der Vergleich zwischen Parfum und Programm jedoch. Parfums werden von unabhängigen Kontrollstellen auf ihre gesundheitliche Verträglichkeit überprüft.

3 Wenn Sie über einen Compiler verfügen, der auf Ihrem System ausgeführt werden kann, können Sie auch sicher sein, dass der Maschinencode, den der Compiler erzeugt, auf Ihrem System ausführbar ist. Manche Compiler, sogenannte Cross-Compiler, erlauben die Erstellung von Programmen für andere als die eigene Plattform.

Wer aber kontrolliert die Programme? Nehmen wir einmal die Kombination Windows-Betriebssystem und Internet Explorer. Ihr Betriebssystem hat praktisch die vollständige Kontrolle über alle Vorgänge und alle Daten auf Ihrem Rechner. Der Internet Explorer stellt die Verbindung zum Internet her. Windows könnte (ich sage »könnte«, denn natürlich kann man einer seriösen Firma wie Microsoft so etwas nicht unterstellen) also ohne Mühe die Daten auf Ihrem PC ausspionieren (beispielsweise welche Programme installiert sind) und diese Informationen beim nächsten Aufruf des Internet Explorer an den eigenen Server zur Auswertung schicken, um Anwender ausfindig zu machen, die Raubkopien von Microsoft-Produkten verwenden.[4] Dies ist wie gesagt nur eine Fiktion, und doch ... An der Entwicklung von Windows 2000 haben angeblich Anhänger der Scientologen mitgewirkt. Es wurde daher von verschiedenen Seiten (beispielsweise auch deutschen Bundesbehörden) gefordert, dass Microsoft Einblick in den Quelltext seines neuen Betriebssystems gewähren soll, damit sich die Behörden vergewissern können, dass die Scientologen keine Viren oder Spionagewerkzeuge im Code versteckt haben.

1.2 Installation unter Unix/Linux

Wenn Sie unter Unix arbeiten, ist Ihr Rechner höchstwahrscheinlich Teil eines größeren Netzwerks, für dessen Verwaltung es einen Systemadministrator gibt. In diesem Falle brauchen Sie sich nicht selbst um die Installation oder Aktualisierung von Perl zu kümmern, sondern können sich vertrauensvoll an Ihren Systemadministrator wenden.

Wenn Sie Linux auf einem privaten Rechner als Betriebssystem verwenden, sind Sie Ihr eigener Systemverwalter und sind selbst für die korrekte Installation von Perl verantwortlich. In aller Regel wird Ihre Linux-Version aber bereits mit einer Perl-Version ausgestattet sein, und vielleicht ist diese sogar noch recht aktuell. Rufen Sie einfach einmal von einem Konsolenfenster aus `perl -v` auf. Wenn Perl installiert ist und im Pfad steht, wird Ihnen daraufhin die Nummer Ihrer Perl-Version angezeigt. Zum Zeitpunkt der Drucklegung dieses Buches war 5.6.0 aktuell. Jede Version, deren Kennnummer mit 5 beginnt, sollte aber für den Einstieg vollkommen ausreichend sein. Sie können sich dann erst einmal mit der vorhandenen Version in die Perl-Programmierung einarbeiten und die aktuelle Version später nachinstallieren.

4 Falls Sie selbst die eine oder andere Raubkopie auf Ihrem Rechner installiert haben, brauchen Sie nicht gleich blass zu werden, denn selbstverständlich spioniert Microsoft seine Anwender nicht aus. Bedenken Sie aber, dass die Verwendung von Raubkopien kein Kavaliersdelikt ist.

Wenn Sie sich für eine Installation entscheiden, haben Sie die Wahl zwischen drei Optionen:

1. Quellcode-Version

Sie können eine Quellcode-Version herunterladen. Diese hat den Vorteil, dass sie auf jeden Fall auf Ihrem System installierbar sein sollte. Nachdem Sie die Datei heruntergeladen haben, gehen Sie folgendermaßen vor:

1. Entzippen Sie die Datei

```
> gzip -d Dateiname.tar.gz
```

2. Entpacken Sie das tar-Archiv

```
> tar xfv Dateiname.tar
```

3. Wechseln Sie in das neu angelegte Verzeichnis, das nach der aktuellen Perl-Versionsnummer benannt ist

```
> cd perl5.005_03
```

4. Konfigurieren Sie die heruntergeladene Perl-Version für die Kompilation auf Ihrem System

```
> rm -f config.sh
> sh ./Configure
```

5. Die daraufhin gestellten Fragen können Sie meist direkt durch Übernahme der Vorgaben beantworten (einfach ↵ drücken.)

Interessant ist vor allem die Frage nach dem Installationsverzeichnis (»Installation prefix to use?«). Vermutlich wird sich die mit Ihrem Betriebssystem installierte Perl-Version unter /USR/BIN befinden. Sie können nun entscheiden, ob Sie diese Version überschreiben oder die aktuelle Version als zusätzliche Perl-Version laden wollen. Ich würde Ihnen empfehlen, die aktuelle Version zusätzlich zu installieren, möglichst unter /USR/LOCAL. Dies hat den Vorteil, dass die alte Version nicht verloren geht und dass die neu installierte Version auch dann erhalten bleibt, wenn Sie einmal das Betriebssystem aktualisieren.

6. Übersetzen Sie den Perl-Code

```
> make
```

7. Testen Sie den Perl-Interpreter

```
> make test
```

8. Schließen Sie die Installation ab (hierfür sind Super-User-Rechte erforderlich)

   ```
   > make install
   ```

2. Perl-Version des Linux-Vertreibers

Unter Umständen bietet Ihr Linux-Vertreiber auf seiner Website eine aktuellere Perl-Version an, mit der Sie die mit dem System installierte Version aktualisieren können.

Wie Sie diese Version installieren, hängt davon ab, wie Ihr Linux-Vertreiber die Version verpackt hat (die großen Linux-Vertreiber haben eigene Pack- und Installationssysteme, wie zum Beispiel RPM von Red Hat Linux) und ob es sich um eine Quellcode- oder eine Binärversion handelt. Lesen Sie dazu die Installationsinformationen auf den entsprechenden Download-Sites.

3. ActivePerl-Version

Schließlich können Sie sich für eine der ActiveState-Versionen entscheiden. Ob Sie dabei eine der angebotenen Binärversionen installieren können, hängt davon ab, ob Ihre Linux-Version über ein entsprechendes Programm zum Entpacken und Installieren der Binärversion anbietet (wer mit Red Hat Linux arbeitet, kann beispielsweise die rpm-Version herunterladen und installieren)

ActivePerl hat den Vorteil, dass die Online-Dokumentation auch im HTML-Format zur Verfügung gestellt wird und dass Module relativ bequem mit dem ppm-Tool aus dem CPAN heruntergeladen und nachträglich installiert werden können.

Der genaue Ablauf der Installation hängt davon ab, welche Version Sie herunterladen und mit welchem Tool Sie die Datei entpacken und installieren. Lesen Sie dazu die Installationsinformationen auf der ActiveState-Site.

Aktuelle Version ausführen

Wenn die neue Perl-Version nicht über die alte Version installiert wurde, müssen Sie noch dafür sorgen, dass Ihre Perl-Programme auch mit der neu installierten, aktuellen Perl-Version ausgeführt werden. Dazu gibt es mehrere Möglichkeiten. Sie können die Umgebungsvariable `PATH` ändern und um das Verzeichnis erweitern, in dem die neu installierten Perl-Werkzeuge stehen, oder Sie kopieren die Perl-Werkzeuge in den Pfad bzw. legen symbolische Links an (den aktuellen Pfad kann man über den Konsolenbefehl **echo $PATH** anzeigen lassen). Wenn Sie einen dieser Wege gehen, sollten Sie darauf achten, dass es zu keinen Überschneidungen mit Programmen älterer Perl-Versionen kommt.

Am einfachsten ist es jedoch, den Verzeichnispfad zu dem Perl-Interpreter in der Shebang-Zeile der Perl-Skripten anzugeben (siehe Kapitel 2.5, Abschnitt »Der Quelltext«).

1.3 Installation unter Windows

Die Installation von ActivePerl ist an sich recht einfach. Sofern Sie nicht bereits über Windows 2000 verfügen, sind allerdings einige Vorarbeiten notwendig.

Windows-95-Anwender müssen die MSVCRT-Bibliotheken in Ihr Windows-Systemverzeichnis kopieren.

1. Doppelklicken Sie dazu im Windows Explorer auf die Datei `msvcrt.exe`, die Sie zusammen mit Perl heruntergeladen haben (siehe oben).

   Aktualisieren Sie die Anzeige des Explorers (F5 drücken) und kopieren Sie die drei MSVCRT-Dateien mit der Extension `.dll` in Ihr Windows-Systemverzeichnis (beispielsweise /WINDOWS/SYSTEM, wenn das Windows-Betriebssystem in das Verzeichnis `/Windows` installiert wurde). Sollten dort bereits entsprechende Dateien vorhanden sein, brechen Sie den Kopiervorgang ab und versuchen Sie, Perl zuerst mit den vorhandenen Dateien zu installieren.

Windows-95- und -98-Anwender müssen als Nächstes das neue Windows-Installationssystem einrichten.

2. Sie brauchen dazu nur im Windows Explorer auf die Datei INSTMSI.EXE, die Sie zusammen mit Perl heruntergeladen haben (siehe vorangehenden Abschnitt), doppelzuklicken.

Schließlich beginnen Sie mit der eigentlichen Perl-Installation.

3. Doppelklicken Sie auf die heruntergeladene MSI-Datei (beispielsweise ACTIVEPERL-5.6.0.613.MSI) und folgen Sie den Anweisungen in den angezeigten Dialogfeldern.

1.4 Die Installation testen

Das Installieren von Programmen ist heute dank der allseits zur Verfügung stehenden Installationsroutinen keine große Schwierigkeit mehr. Man ruft die Installationsroutine auf, spezifiziert eventuell noch die eine oder andere Installationsoption und schaut dann zu, wie die Fortschrittsanzeige von links nach rechts gefüllt wird. Fertig! Oder doch nicht?

Für die Pessimisten unter den Lesern habe ich hier ein paar Test zusammengetragen, mit denen sie sich von der ordnungsgemäßen Installation der Perl-Programmierwerkzeuge überzeugen können:

Perl vorhanden?

Die Windows-Konsole wird über Start/ Programme/ MSDOS-Eingabeaufforderung geöffnet

Öffnen Sie eine Konsole (unter Unix/Linux auch »Terminal«, unter Windows »MSDOS-Eingabeaufforderung« genannt).

Geben Sie

```
perl -v
```

ein, um zu testen, ob Perl gefunden wird und welche Version gefunden wird.

Geben Sie

```
perl -V
```

ein, wenn Sie an weiteren Informationen zur Installation interessiert sind – beispielsweise wissen wollen, in welche Verzeichnisse Perl kopiert wurde.

Wenn Sie Perl von irgendeinem beliebigen Verzeichnis aus aufrufen, kann es sein, dass eine ältere Version gefunden wird, weil diese im Pfad steht, die neu installierte Version dagegen nicht. Wechseln Sie dann in das Verzeichnis der neu installierten Version.

Online-Dokumentation vorhanden?

Die Online-Dokumentation zu Perl liegt in einem speziellen Format vor, POD (Plain Old Documentation), das auf allen Plattformen gelesen werden kann.

Geben Sie beispielsweise

```
perldoc -f print
```

ein, um in der Online-Dokumentation nach der Beschreibung der perl-Funktion `print` zu suchen.

Die ActiveState-Versionen verfügen auch über eine Dokumentation im HTML-Format. Die entsprechenden HTML-Seiten befinden sich nach der Installation im Perl-Unterverzeichnis HTML.

Mehr Informationen zur Perl-Dokumentation finden Sie in Anhang C.

1.5 Ein erstes Perl-Skript aufsetzen und ausführen

Zum Aufsetzen des Perl-Skripts benötigen Sie einen einfachen Texteditor.

Die Auswahl eines geeigneten Texteditors

Unter Windows bietet sich der Notepad-Editor an (START/PROGRAMME/ZUBEHÖR/EDITOR), unter Unix/Linux der VI, EMACS oder – falls Sie mit KDE arbeiten – KEDIT. Wichtig ist vor allem, dass es sich um einen einfachen Texteditor handelt, der den Text als reinen ASCII-Code ohne spezielle Formatierungszeichen speichert (wie es beispielsweise Microsoft Word tut, wenn Sie die Dateien im doc-Format abspeichern).

Formatierte Texte

Wenn Sie einen Text aufsetzen und im Textverarbeitungssystem formatieren, d.h. verschiedene Schriftarten verwenden, einzelne Wörter fett oder kursiv darstellen, ganze Absätze einrücken oder farbig unterlegen etc., so werden alle für die Formatierung des Textes benötigten Informationen zusammen mit dem Text in der Datei gespeichert.[5] Würde man eine solche Datei an den Perl-Interpreter übergeben, der versucht, den Text als Programmcode zu interpretieren und auszuführen, würde dieser fortwährend über die im Text verstreuten Formatierungsbefehle stolpern und in seiner Pein eine Fehlermeldung nach der anderen produzieren. Der Umfang der Formatierungsinformationen kann übrigens recht groß sein. Wenn Sie mit Windows arbeiten und über Microsoft Word verfügen, legen Sie doch einfach mal eine neue Datei an und speichern diese dann ab, ohne nur einen einzigen Buchstaben eingetippt zu haben. Im Windows Explorer können Sie sich über ANSICHT/DETAILS anzeigen lassen, wie groß die Datei ist. Vermutlich dürfte Sie etliche Kbyte umfassen (auf meinem System waren es 19 Kbyte!), was doch recht erstaunlich ist, wenn man bedenkt, dass die Datei ja an sich leer ist und die PCs noch vor wenigen Jahren gerade einmal über 64 Kbyte Arbeitsspeicher verfügten.[6]

Wem die oben genannten Editoren zu spartanisch oder zu kompliziert zu bedienen sind, der sollte zu Editoren greifen, die auf die Bearbeitung von

5 Man kann sich das Ergebnis ungefähr wie eine HTML-Datei vorstellen, bei der die HTML-Tags ja ebenfalls der Gestaltung der Seite und der Formatierung des Textes dienen (Hinweis: die meisten Webbrowser verfügen über einen Befehl, mit dessen Hilfe man sich den Quelltext der geladenen Webseite anschauen kann).

6 Die Erklärung für dieses Phänomen ist, dass Word mit Dokumentvorlagen und Formatvorlagen arbeitet, die den Dateien automatisch zugewiesen werden und die mit den Dateien gespeichert werden. Darüber hinaus werden auch die Dateieigenschaften und eventuell noch weitere Informationen in der Datei gespeichert.

Programmquelltexten spezialisiert sind. Hier wäre zum Beispiel UltraEdit zu nennen, ein Shareware-Programm, das Sie sich zur Evaluierung von der Website *http://www.ultraedit.com* herunterladen können. UltraEdit erlaubt beispielsweise die Syntaxhervorhebung für verschiedene Programmiersprachen (einschließlich Perl) sowie die Konvertierung von Dos zu Unix, von ASCII zu Unicode oder von OEM zu ANSI.

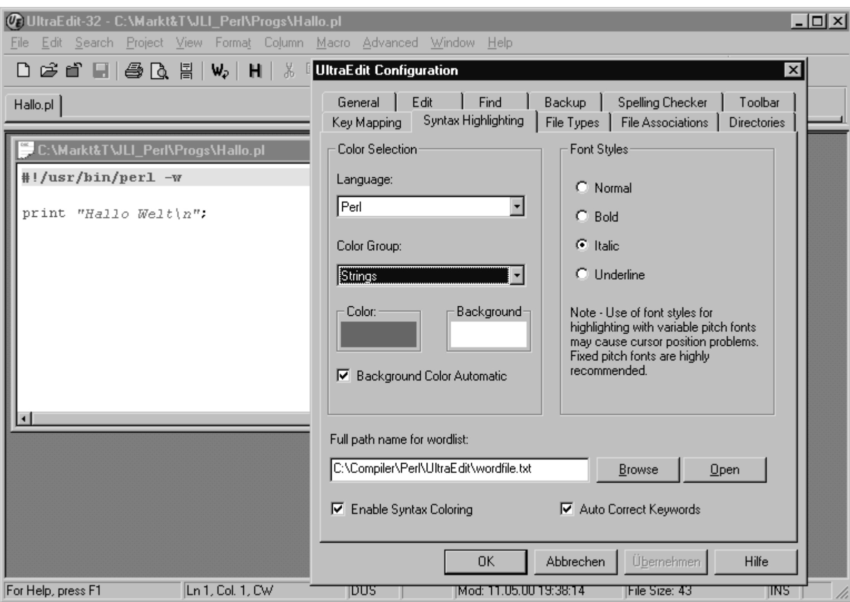


Abb. 1.2: UltraEdit mit Einstellungen zur Syntaxhervorhebung (erreichbar über Advanced/Configuration)

Der Quelltext

Nachdem Sie sich für einen Editor entschieden haben, starten Sie den Editor und geben den folgenden Perl-Quelltext ein (wobei Sie bitte auch auf die Groß- und Kleinschreibung achten):

Listing 1.1: hallo.pl

```
#!/usr/bin/perl -w

print "Hallo Welt!\n";
```

Versuchen wir, in etwa zu verstehen, was dieses Programm macht.

Die erste Zeile ist ein Kommentar. Kommentare beginnen mit einem #-Zeichen und werden vom Perl-Interpreter ignoriert – sie dienen lediglich zur Dokumentation des Quelltextes. Der Kommentar in der ersten Zeile erfüllt allerdings unter Unix/Linux einen besonderen Zweck: Er teilt Unix/Linux mit, mit welchem Interpreter das Skript auszuführen ist, wobei auch – wie

man sieht – Kommandozeilenargumente an den Interpreter übergeben werden können (`-w`). Wenn Sie also unter Unix/Linux arbeiten und Ihr Perl-Interpreter in einem anderen Verzeichnis als `/usr/bin/` steht, ändern Sie die erste Zeile entsprechend ab. Unter Windows wird der Pfad ignoriert und nur die Optionen für den Interpreter werden ausgewertet.

Die Amerikaner bezeichnen die anfängliche Kommentarzeile als Shebang, da sie mit einem # (im Amerikanischen als sharp bezeichnet) und einem ! (dem bang) beginnt.

Unter dem Kommentar folgt eine Leerzeile, die lediglich der besseren Lesbarkeit des Skripts dient.

Der eigentliche Code des Skripts steht in der dritten Zeile. Diese Zeile enthält eine Anweisung, was an dem abschließenden Semikolon zu erkennen ist. Die Anweisung beginnt mit `print`. `print` ist der Name einer Funktion, die in den Perl-Bibliotheken vordefiniert ist und dazu dient, Text auszugeben. Der auszugebende Text folgt direkt hinter dem Namen der Funktion und ist in doppelten Anführungszeichen gefasst, woran der Perl-Interpreter erkennt, dass es sich um Text handelt, den das Programm (und nicht der Interpreter) verarbeitet. Die Anweisung aus der dritten Zeile besteht also aus einem Aufruf der Funktion `print`, der als auszugebender Text der String `"Hallo Welt!\n"` übergeben wird.

Das Skript ausführen

Speichern Sie das Skript unter einem beliebigen Namen mit der Extension `.pl` – beispielsweise als *hallo.pl*.

Um das Skript auszuführen, rufen Sie vom Prompt der Konsole (unter Windows ist dies die MSDOS-Eingabeaufforderung, die über START/PROGRAMME aufgerufen wird) den Perl-Interpreter auf und übergeben ihm das Perl-Skript:

```
perl hallo.pl
```

Danach sollte als Ausgabe

```
Hallo Welt
```

erscheinen.

Wenn Sie mit Windows NT arbeiten und die Dateiextension `.pl` mit dem Perl-Interpreter verbunden haben, können Sie die Skripten auch direkt mit ihrem Namen aufrufen und ausführen lassen.

Auch unter Unix/Linux genügt der Aufruf des Skriptnamens, der zugehörige Interpreter wird dann der auskommentierten ersten Zeile des Skripts entnommen. Voraussetzung ist allerdings, dass das Skript ausführbar ist:

```
> chmod +x hallo.pl
> hallo.pl
```
[7]

Fehler beheben

Selten geht bei der Programmierung alles glatt, ja eigentlich ist es wie im richtigen Leben: Was schief gehen kann, geht schief. Aus diesem Grunde sollten wir kurz die wichtigsten Fehlermeldungen durchgehen:

Wenn Sie eine Fehlermeldung der Art »Command not found« oder »Befehl oder Dateiname nicht gefunden« erhalten, deutet dies daraufhin, dass Ihr Perl-Interpreter nicht gefunden wurde.

Wenn Sie eine Fehlermeldung der Art »Can´t open perl script«, »File not found« oder »Datei oder Verzeichnisname nicht gefunden« erhalten, deutet dies daraufhin, dass Sie den Namen des Perl-Skripts falsch angegeben haben oder sich nicht im gleichen Verzeichnis wie das Skript befinden. Unter Unix/Linux kann eine entsprechende Meldung auch darauf hindeuten, dass der Pfad in der ersten, auskommentierten Zeile des Skripts falsch ist.

Des Weiteren gibt es eine Reihe von Fehlermeldungen, die der Perl-Interpreter ausgibt, wenn er beim Verarbeiten des Skripts auf unzulässigen Code trifft. Meist gehen diese Fehlermeldungen mit der Angabe der betreffenden Zeilenummer einher – beispielsweise:

```
syntax error at hallo.pl line 3, near "prin "Hallo Welt\n""
Execution of hallo.pl aborted due to compilation errors.
```

In diesem Falle wurde beim Eintippen der Name der Funktion `print` verstümmelt. Sie müssen dann den Fehler im Editor beheben, die Datei abspeichern und das Skript erneut von der Konsole aus aufrufen (und hoffen, dass keine weiteren Fehler auftauchen).

Wenn Sie Fehlermeldungen des Perl-Interpreters nachgehen, sollten Sie zwei Dinge beachten:

✘ Die Zeilenangabe der Fehlermeldung verweist auf die Zeile, in der der Interpreter einen Fehler bemerkt hat. Meist werden Sie den Fehler direkt in dieser Zeile finden. Es kann aber auch vorkommen, dass der eigentliche Fehler bereits weiter oben gemacht wurde (meist findet man ihn

7 Wenn das aktuelle Verzeichnis nicht im Ausführungspfad eingetragen ist, müssen Sie explizit mit angeben: `./hallo.pl`.

dann in der vorangehenden Zeile), der Interpreter aber erst in der angezeigten Zeile gemerkt hat, dass mit dem Quelltext etwas nicht stimmt.

- ✘ Wenn der Interpreter mehrere Fehlermeldungen ausgibt, kann es sich bei den letzten Fehlermeldungen um Folgen des ersten Fehlers handeln. Beheben Sie daher zuerst nur die ersten Fehler und versuchen Sie, das Skript zwischenzeitlich auszuführen. Wenn Sie Glück haben, sind die weiteren Fehlermeldungen danach verschwunden.

Syntaxüberprüfung

Grundsätzlich prüft der Interpreter die Korrektheit des Skripts und führt es dann aus. Sie können den Interpreter aber auch dazu verwenden, nur die Korrektheit zu prüfen, indem Sie ihn mit der Option `-c` aufrufen:

```
perl -c hallo.pl.
```

Echte Syntaxfehler führen stets dazu, dass der Interpreter das Skript nicht verarbeiten kann, d.h. der Interpreter weiß nicht, wie er den Quelltext in Maschinencode umwandeln soll. Es gibt aber auch kleinere Fehler oder Unstimmigkeiten, die darauf hindeuten, dass der Programmierer hier einen Fehler gemacht hat, die der Interpreter aber durchaus in Maschinencode übersetzen kann. Wenn Sie ein solches Skript ausführen, gibt es zwei Möglichkeiten:

- ✘ Entweder ist die »Unstimmigkeit«, die der Interpreter gefunden hat, kein Fehler. Das Skript wird dann normal ausgeführt und erledigt seine Arbeit zur vollsten Zufriedenheit des Anwenders.
- ✘ Oder es handelt sich bei der Unstimmigkeit tatsächlich um einen Fehler und das Skript macht nicht das, was man von ihm erwartet. Einen solchen Fehler bezeichnet man als logischen Fehler (womit man ausdrücken will, dass der Programmierer zwar syntaktisch korrekten Code aufgesetzt hat, der aber nicht macht, was er soll) oder als Laufzeitfehler (da der Fehler erst bei Ausführung des Skripts zu Tage tritt).

Laufzeitfehler können recht tückisch sein, denn nicht jeder Laufzeitfehler tritt bei jeder Ausführung des Skripts auf. Es kann durchaus sein, dass Sie ein Skript zehnmal testen, ohne dass ein Fehler auftritt, und beim elften Mal stürzt das Programm unverhofft ab. Ebenso verdrießlich ist die Suche nach den Laufzeitfehlern, denn anders als bei den syntaktischen Fehlern erhalten Sie meist keine Hinweise auf die Zeile, in der der Fehler lokalisiert ist. Ohne die Unterstützung spezieller Fehleranalyseprogramme, sogenannte Debugger (siehe Anhang D), wäre die Suche nach Laufzeitfehlern in größeren Skripten eine Qual.

Am besten wäre es, gar keine Laufzeitfehler zu produzieren – doch dies wird wohl für immer ein frommer Wunsch bleiben.[8] Realistischer ist es, sich zu bemühen, möglichst wenig Laufzeitfehler zuzulassen. Dabei kann man sich vom Interpreter unterstützen lassen, indem man ihn bittet, sämtliche »Unstimmigkeiten«, über die er bei Durchsicht des Skripts stolpert, als Warnungen auszugeben. Die betreffende Option für den Aufruf des Interpreters lautet `-w`:

```
perl -w skriptname.pl
```

beziehungsweise

```
#!/usr/bin/perl -w
```

für Unix/Linux bei Aufruf ohne direkte Angabe des Interpreters.

Wenn Sie das Perl-Skript gar nicht ausführen, sondern nur auf seine syntaktische Korrektheit hin überprüfen wollen, rufen Sie den Perl-Interpreter mit der Option `-c` bzw. `-cw` auf.

1.6 Interpreter versus Compiler

Im vorangehenden Abschnitt war viel vom Perl-Interpreter die Rede. Auch der Begriff des »Compilers« ist in diesem Kapitel bereits gefallen, allerdings ohne dass bisher erklärt worden wäre, was man sich unter einem Interpreter beziehungsweise einem Compiler konkret vorzustellen hat.

Vom Quelltext zur Binärversion

Prinzipiell sind Programme nichts anderes als eine Folge von Befehlen, die an einen Computer gerichtet sind und von diesem befolgt werden. Im Grunde genommen funktionieren Programme also genauso wie Kochrezepte: Sie als Programmierer sind der Koch, der das Buch schreibt. Jedes Kochrezept entspricht einem Programm, und der Computer, der Ihre Programme ausführt, ist der Leser, der die Rezepte nachkocht.

Leider ist die Realität wie üblich etwas komplizierter als das Modell. Im Falle des Kochrezepts können wir einfach davon ausgehen, dass Schreiber und Leser die gleiche Sprache sprechen. Im Falle des Programmierers und des

8 Es gäbe zwar prinzipiell die Möglichkeit, durch Analyse und Fallunterscheidungen die Korrektheit von Codeabschnitten nachzuweisen, doch sind solche Beweisführungen viel zu komplex (und damit selbst wieder fehleranfällig), als dass man sie in der Praxis sinnvoll anwenden könnte.

Computers ist dies natürlich nicht der Fall, denn Sie als Programmierer sprechen an sich Deutsch und der Computer spricht ... ja, welche Sprache versteht eigentlich ein Computer?

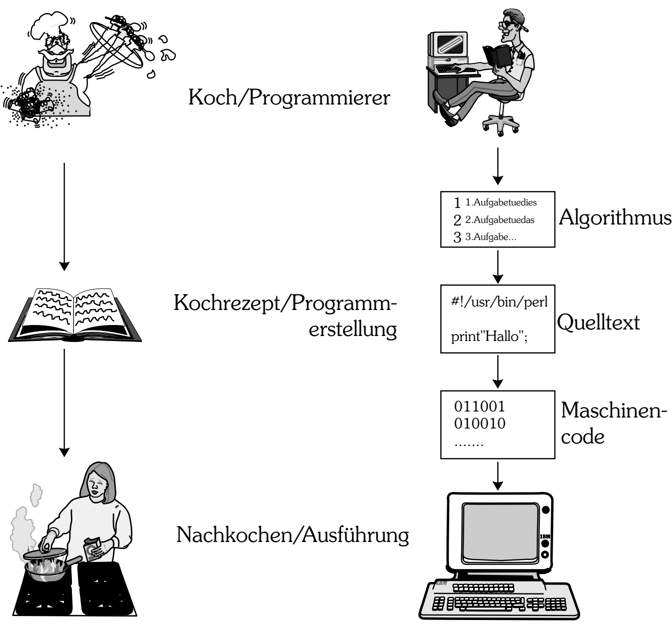


Abb. 1.3: Analogie zwischen Programmen und Kochrezepten

Ich wünschte, Sie hätten diese Frage nicht gestellt, denn die Antwort ist äußerst unerfreulich. Der Computer, in diesem Fall sollte man genauer von dem Prozessor des Computers sprechen, versteht nur einen ganz begrenzten Satz elementarer Befehle – den sogenannten Maschinencode, der zu allem Unglück noch binär codiert ist und daher als eine Folge von Nullen und Einsen vorliegt. Können Sie sich vorstellen, Ihre Programme als Folge von Nullen und Einsen zu schreiben? Wahrscheinlich genauso wenig, wie Ihr Computer in der Lage ist, Deutsch zu lernen. Wir haben also ein echtes Verständigungsproblem. Um dieses zu lösen, müssen Sie – als der Intelligentere – dem Computer schon etwas entgegenkommen. Die Frage ist nur wie?

Stellen Sie sich vor, Sie sind auf einem Treffen der »Anonymen Perl-Programmierer« und treffen dort den berühmten Perl-Guru W. Hadi, den Sie gerne fragen würden, was die Perl-Konstruktion `m/\b\d{3}\b/`[9] bedeutet. Leider ist W. Hadi Inder und spricht nur Indisch. Glücklicherweise findet sich

9 Zur Erklärung dieses Ausdrucks siehe Kapitel 10.2.

jedoch ein Dolmetscher, der vom Englischen ins Indische übersetzen kann. Sie besinnen sich also Ihrer Englischkenntnisse, der Dolmetscher übersetzt diese ins Indische und W. Hadi beantwortet freundlich Ihre Fragen.

Hier wurden zwei Tricks angewendet:

- ✘ Zum einem wurde ein Dolmetscher eingeschaltet.
- ✘ Zum anderen mussten Sie auf eine Sprache ausweichen, für die es einen Dolmetscher gibt.

In gleicher Weise lösen wir das Verständigungsproblem mit dem Computer. Als Dolmetscher verwenden wir ein Programm, das unsere Programmbefehle in Maschinencode übersetzt. Da es jedoch kein Programm gibt, das die deutsche Sprache versteht und in Maschinencode übersetzen könnte, müssen wir auf eine Sprache ausweichen, die so einfach ist, dass ein Programm sie verstehen und übersetzen kann. Menschliche Sprachen sind für solche Zwecke ungeeignet. Sie umfassen einen viel zu großen Wortschatz, haben zu komplizierte Grammatiken und sind zudem noch stark kontextabhängig (so kann man beispielsweise nur aus dem Kontext schließen, dass sich das »Sie« vom Anfang des letzten Satzes auf die »menschlichen Sprachen« und nicht etwa Sie, den Leser, bezieht). Also erzeugt man künstliche Sprachen, mit minimalem Wortschatz, einfacher Grammatik und geringer Kontextabhängigkeit – die sogenannten Programmiersprachen.

Die Programmiersprache unserer Wahl heißt »Perl«. Der Dolmetscher, der für uns die in Perl formulierten Skripten in Maschinencode übersetzt, ist der Perl-Interpreter.

Wie arbeitet der Perl-Interpreter?

Ich habe höchsten Respekt vor der Arbeit des Dolmetschers, insbesondere des Simultandolmetschers. Menschen können zwar ohne Probleme mehrere Dinge gleichzeitig tun (beispielsweise Kaugummi kauend über die Straße schlendern, dabei Musik hören und sich mit einem Freund unterhalten), doch funktioniert dies nur, weil die meisten dieser Tätigkeiten größtenteils unbewusst ablaufen. Tätigkeiten, die Aufmerksamkeit erfordern, können dagegen nicht gleichzeitig verrichtet werden. Versuchen Sie doch einmal, einen längeren Satz zu sagen und gleichzeitig einen ganz anderen Satz aufzuschreiben. Oder klopfen Sie sich mit beiden Händen auf die Oberschenken – allerdings mit unterschiedlichen Rhythmen. Sie werden es erst schaffen, wenn Sie das Klopfen soweit automatisiert haben, dass die Hände automatisch den Rhythmus halten, ohne dass Sie sich darauf konzentrieren müssen. Wie also schafft es der Dolmetscher, einen ihm unbekannten Text zu hören, zu verstehen, zu analysieren und laut zu übersetzen, während sein Gegenüber weiterspricht?

Ohne diese Leistung zu schmälern, wollen wir einmal ganz nüchtern betrachten, was beim Dolmetschen geschieht. Der Dolmetscher nimmt eine Eingabe entgegen, übersetzt diese in Gedanken und gibt die Übersetzung aus. Ähnlich arbeitet ein Interpreter. Er nimmt einen Quelltext entgegen, übersetzt diesen in Maschinencode und lässt den erzeugten Maschinencode vom Prozessor ausführen. Manche Interpreter gehen dabei so vor, dass sie den Quelltext zeilenweise oder Anweisung für Anweisung einlesen, übersetzen und ausführen lassen. Andere Interpreter lesen zuerst den ganzen Text ein. Der Perl-Interpreter verfährt nach letzterem Schema. Er liest erst den gesamten Quelltext des Skripts ein. Danach wird der Quelltext in einen Pseudo-Maschinencode übersetzt, den sogenannten Bytecode. Dieser wird dann Befehl für Befehl in Maschinencode übersetzt und vom Prozessor ausgeführt. Gegenüber dem zeilenweisen Einlesen und Übersetzen hat dies den Vorteil, dass die Qualität der Übersetzung besser ist und das Skript anschließend schneller ausgeführt wird (da die Ausführung nicht fortlaufend für das Einlesen und Übersetzen der nächsten Zeile unterbrochen werden muss).

Was ist ein Compiler?

Als Alternative zum Interpreter gibt es den Compiler. Ein Compiler ist ebenfalls ein Programm, das einen Quelltext in Maschinencode übersetzt. Allerdings lässt der Compiler das Programm nicht ausführen. Stattdessen erzeugt der Compiler eine Ausgabedatei, in der er den erzeugten Maschinencode abspeichert. Lässt man als Ausgabedatei eine ausführbare Datei erstellen, kann man diese danach aufrufen und ausführen lassen.

Ein Vorteil des Compilers ist, dass Sie danach eine Binärversion des Programms vorliegen haben, die man gut kommerziell vertreiben kann. Der Kunde erhält die Binärversion des Programms, die er direkt (ohne Einschaltung eines Interpreters) aufrufen und ausführen kann. Da nur die Binärversion und nicht der Quelltext weitergegeben wird, besteht keine große Gefahr, dass für das Programm entwickelte Algorithmen und Programmtechniken ausspioniert werden.

Ein kleiner Nachteil ist, dass der Compiler plattformspezifischen Maschinencode erzeugt. Wenn Sie ein Programm für Windows-Anwender entwickeln, müssen Sie einen Compiler verwenden, der Maschinencode für die Windows/Intel-Plattform erzeugt. Wenn Sie das gleiche Programm an Linux-Anwender verkaufen wollen, müssen Sie es mit einem weiteren Compiler übersetzen, der Linux-kompatible Programme erzeugt.

Die Beschränkung der Compiler auf eine Plattform hat aber auch Vorteile: Wenn man weiß, auf welcher Plattform ein Programm laufen soll, kann man den Code des Programms gezielt optimieren. Fast alle Compiler bieten

daher die Möglichkeit, Programme hinsichtlich der Codegröße oder der Ausführungsgeschwindigkeit zu optimieren.

Wie schneidet dazu im Vergleich der Interpreter ab?

Perl-Programme werden nicht als Binärcode, sondern als Quelltext – man spricht daher auch von Skripten statt von Programmen – weitergegeben. Dies hat den Nachteil, dass jeder, dem man das »Programm« aushändigt und der mit Perl vertraut ist, sehen kann, wie das Programm implementiert ist. Ein wirklicher Nachteil ist dies aber im Grunde nur, wenn man den anderen Programmierern die Einsicht in das eigene Programmier-Know-how missgönnt bzw. mit seinen Programmen Geld verdienen will. In der Perl-Gemeinde sind solcherlei Gedanken allerdings nicht weit verbreitet, und so stört sich kaum jemand daran.

Ein echter Nachteil ist, dass bei jedem Aufruf des Perl-Skripts der Quelltext neu übersetzt werden muss. Der Perl-Interpreter liest daher zuerst den ganzen Quelltext ein und übersetzt ihn komplett in Bytecode, bevor er mit der Ausführung des Programms beginnt. Somit wird erreicht, dass wenigstens die Ausführung des Programms fast genauso schnell abläuft wie die Ausführung eines kompilierten Programms. Lediglich der Start des Programms ist etwas verzögert, was sich auf schnellen Rechnern aber erst bei größeren Perl-Skripten bemerkbar machen dürfte.

Voraussetzung dafür, dass ein Anwender ein Perl-Skript ausführen kann, ist, dass auf seinem System ein passender Perl-Interpreter installiert ist. Unter Unix/Linux ist dies praktisch immer der Fall, Windows-Anwender müssen den Perl-Interpreter erst zusammen mit den anderen Perl-Werkzeugen aus dem Internet herunterladen und installieren. Doch ist die Installation erst einmal geglückt, sollte es mit der Ausführung der Skripten keine Probleme geben. Für die Weitergabe von Perl-Skripten hat dieses Verfahren weitreichende Folgen. Sofern ein Perl-Skript nicht ausdrücklich plattformspezifische Befehle verwendet, kann man es praktisch auf jede beliebige Betriebssystem/Prozessor-Plattform portieren, für die es einen passenden Interpreter gibt.

1.7 Bestandteile eines Perl-Skripts

Zum Abschluss dieses Kapitels wollen wir uns den Quelltext des eben erstellten Perl-Skripts etwas näher anschauen. Dabei geht es uns weniger um die genaue Syntax (für die eingehende Beschäftigung mit der Perl-Syntax haben wir ja noch ein ganzes Buch lang Zeit), sondern vielmehr um die grundlegenden Komponenten eines Perl-Skripts, die ich Ihnen anhand dieses Bei-

spielprogramms vorstellen möchte. Zur Erinnerung hier noch einmal der Quelltext:

Listing 1.2: hallo.pl

```
#!/usr/bin/perl -w

print "Hallo Welt!\n";
```

Anweisungen

Mit die wichtigsten Elemente eines Perl-Skripts sind die Anweisungen, die eigentlichen Befehle, die vom Interpreter in Maschinencode umgewandelt werden.

Anweisungen werden mit einem Semikolon abgeschlossen und grundsätzlich nacheinander von oben nach unten ausgeführt. Mit Hilfe spezieller Konstrukte (Verzweigungen und Schleifen) kann der Programmierer Einfluss auf die Ausführung der Anweisungen nehmen.

Obiges Skript enthält lediglich eine einzige Anweisung:

```
print "Hallo Welt!\n";
```

Schlüsselwörter, Operatoren und Symbole

Praktisch jede Programmiersprache verfügt über einen bestimmten Satz von Schlüsselwörtern, Operatoren und Symbolen. Die Schlüsselwörter, Operatoren und Symbole bilden quasi das Grundvokabular der Sprache. Der Interpreter kennt diese Vokabeln genau und erwartet von Ihnen, dass Sie diese Vokabeln in Ihren Programmen korrekt verwenden.

In obigem Skript werden beispielsweise folgende Symbole verwendet:

leitet einen Kommentar ein

" " kennzeichnet einen String (=Text, den das Programm verarbeitet)

\ leitet in einem String ein Sonderzeichen ein

; schließt eine Anweisung ab

Bezeichner

Der begrenzte Wortschatz aus Schlüsselwörtern, Operatoren und Symbolen reicht nicht aus, um vernünftige Programme zu schreiben. Perl erlaubt Ihnen daher, den Wortschatz des Interpreters zu erweitern. Dies geschieht im Zuge einer sogenannten Deklaration, bei der Sie dem neuen Element einen Namen (Bezeichner) geben und dem Interpreter mitteilen, um was für ein Element (Variable, Funktion etc.) es sich handelt. Nach der Deklaration

kann der neue Name im Programmcode verwendet werden. (In Kapitel 2 erfahren Sie mehr über Bezeichner, Variablen und Deklarationen.)

Obiges Skript enthält keine Deklaration, wohl aber einen Bezeichner: den Namen der Funktion `print`. Irgendwie widerspricht dies der obigen Aussage, dass Bezeichner vor der Verwendung deklariert werden müssen. Die Erklärung hierzu ist, dass es sich bei `print` um eine vordefinierte Funktion aus der Standard-Perl-Bibliothek handelt, die dem Perl-Interpreter automatisch bekannt ist.

Kommentare

Kommentare beginnen mit einem #-Zeichen und gehen bis zum Ende der aktuellen Zeile.

```
# Dieser Kommentar beginnt am Zeilenanfang

print "Hallo";  # Dieser Kommentar geht
                # über zwei Zeilen
```

Grundsätzlich können Sie als Kommentar jeden beliebigen Text einfügen, da die Kommentare bei der Verarbeitung des Programms durch den Perl-Interpreter ja ignoriert werden. In der Praxis sollte man allerdings darauf achten, dass die Kommentare knapp, informativ und sachbezogen sind und dem Programmierer helfen, Aufbau und Funktionsweise des Perl-Skripts besser zu verstehen. Kommentare sind nicht dafür gedacht, den Code eines Skripts Zeile für Zeile zu erklären, so dass selbst Programmierunkundige den Code verstehen können. Kommentare sollen Ihnen oder anderen Perl-Programmierern helfen, sich schneller in den Code eines Skripts, das Sie überarbeiten wollen, einzudenken. In diesem Sinne empfiehlt sich in größeren Skripten die Kommentierung folgender Elemente:

- wichtige Variablen
- komplizierte Codeblöcke
- Code-Verzweigungen
- Funktionen

Whitespace

Als Whitespace bezeichnet man den Leerraum, der entsteht, wenn Sie ein Leerzeichen, einen Tabulator oder einen Zeilenumbruch einfügen. Whitespace erfüllt mehrere Aufgaben:

In bestimmten Fällen ist Whitespace zur Trennung von Programmelementen erforderlich.

- ✘ Kommentare werden mit einem Zeilenumbruch abgeschlossen.

```
# Nachricht ausgeben
print "Hallo";
```

Dies ist eine `print`-Anweisung mit einem vorangehenden Kommentar. Dagegen stellt die folgende Zeile nur einen Kommentar ohne Anweisung dar:

```
# Nachricht ausgeben      print "Hallo"
```

- ✘ Bezeichner werden durch Leerzeichen von Schlüsselwörtern abgegrenzt. Wenn Sie später eigene Funktionen definieren, leiten Sie die Definition mit dem Schlüsselwort `sub` gefolgt von dem Namen für die Funktion ein:

```
sub meineFunktion
```

Ohne Leerzeichen würde der Interpreter nur einen ihm unbekannten Bezeichner `submeineFunktion` erkennen.

- ✘ Whitespace kann die Lesbarkeit eines Programms verbessern..

Zwischen Operatoren, Operanden und Symbolen muss kein Whitespace stehen. Eingefügte Leerzeichen können den Code aber besser lesbar machen.

Auch die Einrückung von Codeblöcken trägt entscheidend zur Lesbarkeit bei:

```
#!/usr/bin/perl -w

print "Geben Sie eine Zahl zwischen 0 und 9 ein.\n";

$eingabe = 1;

while ($eingabe != 0)
  {
  print $text;

  chomp ($eingabe = <STDIN>);
  if ($eingabe < 0 || $eingabe > 9) {
     redo;
     }

  print "Korrekte Eingabe\n";
  }
```

- In Strings bleiben Leerzeichen unverändert erhalten, Tabulatoren und Zeilenumbrüche müssen in Form von Sonderzeichen eingefügt werden (\t für den horizontalen Tabulator, \n für den Zeilenumbruch).

```
print "Hallo Welt";
```

gibt

```
Hallo Welt
```

aus.

```
print "Hallo        Welt";
```

gibt

```
Hallo        Welt
```

aus.

```
print "Hallo\nWelt";
```

gibt

```
Hallo
Welt
```

aus.

Teil I

Die Grundlagen

Kapitel 2: Daten

Kapitel 3: Programmieren mit Zahlen und Strings

Kapitel 4: Steuerung des Programmflusses

Kapitel 5: Listen, Arrays und Hashes

Kapitel 6: Funktionen

Im ersten Teil dieses Buches werden wir uns ganz allgemein mit den Grundlagen der Programmierung sowie im Besonderen mit den Grundlagen der Programmierung in Perl beschäftigen.

- ✘ Ausgangspunkt unserer Betrachtungen sind die Daten. Welche Typen von Daten gibt es, wie werden diese Daten in Perl-Programmen repräsentiert und wie kann man Daten einlesen und ausgeben? All diesen Fragen gehen wir im Kapitel »Daten« nach.
- ✘ Der nächste Schritt ist die Verarbeitung der Daten. Dies geschieht mit Hilfe spezieller Operatoren (siehe Kapitel »Programmieren mit Zahlen und Strings«).
- ✘ Grundsätzlich werden die Befehle eines Perl-Skripts vom Anfang des Skripts bis zum Ende nacheinander ausgeführt. Mit Hilfe spezieller »Kontrollstrukturen« kann man den Programmablauf umlenken, um einzelne Befehle wiederholt oder alternativ den einen oder anderen Befehlsblock ausführen zu lassen (siehe Kapitel »Steuerung des Programmflusses«).
- ✘ Zur Verwaltung komplexerer Daten benötigt man spezielle Datenstrukturen. Diese werden wir im Kapitel »Listen, Arrays und Hashes« kennen lernen.
- ✘ Mit zunehmender Skriptgröße wird es immer wichtiger, den Code zu strukturieren und zu modularisieren. Dazu identifiziert man Teilprobleme, die man in Form kleiner Unterprogramme, den sogenannten »Funktionen«, löst.

Kapitel 2

Daten

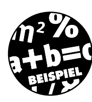
Jedes Programm – sei es nun in Perl, in C oder in Java geschrieben, handele es sich um ein Textverarbeitungs- oder ein Grafikprogramm, um ein Virusprogramm oder einen Virenscanner, um ein 10-Mbyte-Programm oder unser kleines Listing aus dem vorangehenden Kapitel – verarbeitet in irgendeiner Form Daten. Folglich muss jede Programmiersprache über Wege zur Repräsentation und Bearbeitung von Daten verfügen. Wie dies in Perl aussieht, ist Thema dieses Kapitels.

Im Einzelnen lernen Sie,

- ✘ welche Typen von Daten es gibt
- ✘ wie man Konstanten verwendet
- ✘ was skalare Variablen sind
- ✘ wie man Daten und Operatoren zu Ausdrücken zusammenstellt
- ✘ wie man Daten von der Tastatur einliest
- ✘ wie man Daten auf den Bildschirm ausgibt

In den Vertiefungsabschnitten erfahren Sie

- ✘ mehr über die Beziehung von Zahlen und Strings
- ✘ was sich hinter dem skalaren Variablentyp verbirgt
- ✘ wie die elementaren Datentypen auf Maschinenebene dargestellt werden



Folgende Elemente lernen Sie kennen:

Konstanten, skalare Variablen, das Escape-Zeichen `\`, die Funktionen `print`, `printf` und `chomp`, den Eingabeoperator `<>`, die Datei-Handle `STDIN` und `STDOUT`.

2.1 Kleine Datenkunde

Bevor wir uns anschauen, wie Daten in Perl-Skripten eingebaut werden, wollen wir untersuchen, wie diese Daten überhaupt beschaffen sind.

Verschiedene Arten von Daten

Welche Daten würden Sie denn gerne in Ihren Programmen verarbeiten?

- Zahlen, zweifelsohne! Ob Sie nun zwei Zahlenwerte (beispielsweise Längenangaben) vergleichen, ein Integral berechnen oder die Ausgaben für den aktuellen Monat aufaddieren wollen – Zahlen tauchen in unzähligen Programmen auf.
- Text. Natürlich! Schließlich sprechen wir hier über Perl, die Programmiersprache, die sich die Textverarbeitung auf die Fahne geschrieben hat.
- Bilder. Oops, das klingt kompliziert! Wenn man aber die Besonderheiten der verschiedenen Bildformate außer Acht lässt, kommt man zu der Erkenntnis, dass ein Bild letztlich nichts anderes als ein rechteckiger Bereich auf dem Bildschirm ist, dessen einzelne Pixel verschiedene Farben haben[1]. Die Daten eines Bildes bestehen also im Wesentlichen aus den Farbwerten für die einzelnen Pixel des Bildes. Diese Farbwerte werden nicht durch Namen (»Rot«, »Himmelblau« oder »Zartes Altrosa mit einem Stich ins Türkisfarbene«), sondern durch Zahlen angegeben, da Zahlenwerte eindeutig, exakt und besser abzustufen sind.

  Für unsere Betrachtung der verschiedenen Arten von Daten bedeutet dies, dass wir für Bilder keinen neuen Datentyp brauchen, da ein Bild als eine Folge von Zahlen dargestellt werden kann.
- Adressen. Sie wollen eine kleine Datenbankanwendung schreiben, mit der Sie die Adressen Ihrer Freunde und Bekannten verwalten. Die Daten, um die es hierbei geht, sind also Adressen. Eine Adresse ist aber

1 Dies ist im Groben die Definition einer Rastergrafik. Das Pendant zur Rastergrafik ist die Vektorgrafik, die als Sammlung von Linien und anderen geometrischen Figuren definiert ist.

wiederum nichts anderes als eine Gruppe von Textdaten (Name, Vorname, Straße, Stadt etc.) und Zahlen (Alter, Postleitzahl, Hausnummer, Telefon etc.). Wiederum sind die grundlegenden Daten, mit denen wir es zu tun haben, Zahlen und Text, die lediglich zu einer höheren Organisationsebene (der Adresse) zusammengefasst werden.

- ✘ Sound. Es wird Sie sicherlich nicht mehr überraschen, wenn ich Ihnen sage, dass man auch Sounddaten auf Zahlen zurückführen kann. Sound besteht aus Tönen. Töne haben eine definierte Frequenz (Tonhöhe) und eine Lautstärke, die man beide als Zahlenwerte darstellen kann. Die meisten gängigen Soundformate (Spezifikationen für das Speichern von Sound in Dateien) sind zwar um einiges komplizierter (Berücksichtigung des Taktschemas, Nachahmung von Instrumenten, Abspeicherung mehrerer Tracks für verschiedene Stimmen und Instrumente), doch werden alle diese Informationen als Zahlenwerte abgespeichert.

Letztlich haben wir es also mit nur zwei elementaren Datentypen zu tun: Zahlen und Text. Alle anderen Arten von Daten sind aus diesen elementaren Daten aufgebaut oder können durch sie codiert werden.

Die Datentypen von Perl

Im vorangehenden Abschnitt haben wir gesehen, dass man praktisch jede Art von Daten auf zwei grundlegende Datentypen zurückführen kann: auf Zahlen und auf Text. Folglich dürfen wir von Perl erwarten, dass es diese »elementaren« Datentypen direkt unterstützt – und werden in dieser Erwartung auch nicht enttäuscht.

In Form von

- ✘ Konstanten (siehe 3.2) und
- ✘ skalaren Variablen (siehe 3.3)

kann man Zahlen und Text in Programmen repräsentieren und bearbeiten.

Darüber hinaus muss es möglich sein, Daten zu höheren Organisationsformen zusammenzustellen. Perl unterstützt zu diesem Zwecke folgende höhere Datenstrukturen:

- ✘ Listen (siehe Kapitel 5.2) und
- ✘ Hashes (siehe Kapitel 5.3)

Neben Zahlen und Text kennt Perl noch zwei weitere Datentypen: Referenzen (siehe Kapitel 7) und Datei-Handles (siehe Kapitel 9).

Konstanten, Variablen und Ausdrücke

Da alle Programmierer (auch die Programmieranfänger) begnadete Mathematiker sind[2], scheint es mir am einfachsten, die Begriffe »Konstante«, »Variable« und »Ausdruck« am Beispiel einer algebraischen Gleichung einzuführen.

```
y = 2*x + 10
```

In dieser Gleichung werden zwei Terme (die Mathematiker sprechen von »Termen« statt von »Ausdrücken«), `y` und `2*x+10`, gleichgesetzt. Der zweite Term besteht aus zwei Konstanten, `2` und `10`, sowie einer Variablen `x`, für die man beliebige Werte aus der Definitionsmenge der Gleichung einsetzen kann. Ganz allgemein besteht ein Term aus einer sinnvollen Kombination von Konstanten, Variablen und Operatoren. Setzt man für die Variablen in einem Term definierte Werte ein, erhält man einen Ergebniswert (in obiger Gleichung liefert der rechte Term beispielsweise den Ergebniswert `16`, wenn man für `x` den Wert `3` einsetzt).

Soweit die Analogie zur algebraischen Gleichung. Jetzt wollen wir uns anschauen, wie Konstanten, Variablen und Ausdrücke in Perl definiert und verwendet werden.

2.2 Konstanten

Perl kennt zwei Arten von Konstanten:

- ✘ String-Konstanten (für Text) und
- ✘ Zahlenkonstanten.

String-Konstanten

String-Konstanten stehen in Anführungszeichen

String-Konstanten werden in einfache oder doppelte Anführungszeichen gesetzt:

```
'Dies ist eine String-Konstante'
"Dies ist ebenfalls eine String-Konstante"
```

An den Anführungszeichen erkennt der Perl-Interpreter, dass es sich bei dem nachfolgenden Text nicht um einen Programmbefehl handelt, sondern um Text, den der Programmierer bearbeiten, speichern oder ausgeben möchte.

---

2 Dass Programmierer komplexere Rechenaufgaben lieber durch Aufsetzen eines passenden Programms als durch Nachdenken lösen, ist eine böswillige Unterstellung!

Da Beginn und Ende des Strings durch Anführungszeichen gekennzeichnet werden, ergibt sich ein Problem, wenn man im String selbst ein Anführungszeichen verwenden will. Wie würden Sie beispielsweise den folgenden Text ausgeben:

```
Peter sagte "Hallo!"
```

Sicherlich nicht, indem Sie schreiben:

```
print "Peter sagte "Hallo!""
```

Der Perl-Interpreter würde diese Zeile als einen String (`"Peter sagte "`), gefolgt von einem dem Interpreter unverständlichen Programmelement (Hallo!) und einem zweiten, leeren String `""` ansehen.

Der Escape-Operator \

Um innerhalb eines Strings Anführungszeichen auszugeben, muss man dem Interpreter zu erkennen geben, dass das Anführungszeichen nicht das Ende des Strings anzeigt, sondern wie ein ganz normales Zeichen auszugeben ist. Dies erreicht man durch Voranstellung des sogenannten Escape-Operators: `\`.

Die Ausgabe unseres Beispielstrings würde daher wie folgt aussehen:

```
print "Peter sagte \"Hallo!\"";
```
[3]

Nun ist aber der Escape-Operator wiederum identisch mit dem Backslash, den man ab und an vielleicht auch ausgeben möchte (unter Windows beispielsweise bei der Ausgabe von Verzeichnispfaden). In diesem Fall verdoppelt man einfach den Backslash. Der erste Backslash fungiert dann als Escape-Operator, der anzeigt, dass der zweite Backslash als normales Zeichen ausgegeben werden soll.

```
#!/usr/bin/perl -w
print "Wechseln Sie zum Verzeichnis C:\\Perl.";
```

Ausgabe:

```
Wechseln Sie zum Verzeichnis C:\Perl.
```

Mit dem Escape-Operator kann man nicht nur Perl-Symbole (", ', \) in normale Zeichen verwandeln, man kann umgekehrt auch normale Zeichen in Sonderzeichen verwandeln. In Tabelle 2.1 finden Sie eine Zusammenstel-

3 Eine andere Möglichkeit bestünde darin, den String in einfache Anführungszeichen zu setzen: `print 'Peter sagte "Hallo!" ';`. Dies ist allerdings nur dann sinnvoll, wenn der String keine Variablennamen und keine Escape-Sequenzen enthält (siehe Abschnitt »Einfache versus doppelte Anführungszeichen«).

lung der verschiedenen Escape-Sequenzen und der von ihnen erzeugten Ausgabe.

Tabelle 2.1: Escape-Sequenzen

Escape-Sequenz	Bedeutung
\a	Signalton
\b	Backspace
\e	Escape
\f	Neue Seite
\n	Neue Zeile
\r	Wagenrücklauf
\t	Tabulator
\\	\
\"	"
\'	'
\nnn	Zeichen, dessen ASCII-Code der oktalen Zahl nnn entspricht
\xnn	Zeichen, dessen ASCII-Code der hexadezimalen Zahl nn entspricht
\cX	Control-Darstellung
\u	nächstes Zeichen groß
\U	alle nachfolgenden Zeichen groß
\l	nächstes Zeichen klein
\L	alle nachfolgenden Zeichen klein

Einfache versus doppelte Anführungszeichen

Der Einsatz der Escape-Sequenzen bringt mich zu einem Punkt, der schon oben angesprochen wurde, dann aber etwas unterging. Zu Beginn des Kapitels wurde ausgeführt, dass man Strings sowohl in doppelte als auch in einfache Anführungszeichen kleiden kann. Da stellt sich die Frage, ob es egal ist, welche Anführungszeichen man verwendet oder ob es nicht vielleicht doch einen Unterschied gibt.

Am besten beantworten Sie sich die Frage selbst. Schreiben Sie ein Perl-Skript mit den folgenden `print`-Aufrufen:

```
#!/usr/bin/perl -w
print "Hallo Welt!\n";
print 'Hallo Welt!\n';
```

An der Ausgabe

```
Hallo Welt!
Hallo Welt!\n
```

können Sie ablesen, dass Escape-Sequenzen nur in Strings ersetzt werden, die in doppelte Anführungszeichen gefasst sind. Strings in einfachen Anführungszeichen werden direkt so ausgegeben, wie sie im Quellcode stehen.

Die Ersetzung der Escape-Sequenzen bezeichnet man auch als *Expansion*. Neben der Expansion der Escape-Sequenzen gibt es noch die Variablenexpansion, die ebenfalls nur in Strings in doppelten Anführungszeichen stattfindet und zu der wir im nächsten Abschnitt kommen.

Zahlenkonstanten

Zahlenkonstanten werden direkt in den Quellcode geschrieben, beispielsweise:

```
print 333;
```

oder

```
print -333;
```

Beachten Sie, dass die Nachkommaziffern von den Vorkommastellen durch einen Punkt (und nicht wie im Deutschen üblich durch ein Komma) getrennt werden.

Sie können auch Zahlen mit Nachkommastellen angeben:

```
print 333.33;
```

Wenn Sie große Zahlenwerte eingeben, empfiehlt es sich, zumindest die Vorkommastellen in Gruppen zu ordnen. Im Deutschen verwendet man dafür den Punkt, in Perl den Unterstrich _ (der Punkt wird ja schon zur Abtrennung der Nachkommastellen verwendet):

```
print 333_333_333.333;  # Ausgabe: 333333333.333
```

Wer möchte, kann die Zahlen in Exponentialschreibweise angeben, wobei der Exponent durch »e« oder »E« eingeleitet wird:

```
print 1e-3;      # = 1*10^3 = 0.001
```

Die Exponentialschreibweise eignet sich insbesondere für sehr große oder sehr kleine Zahlen:

```
print 1.6021892e-19;     # Ladung eines Elektrons
print 5.976e24;          # Masse der Erde in kg
print 3.0e8;             # Lichtgeschwindigkeit
```

Das Zeichen *e/E* steht für »Exponent« und nicht etwa für die Eulersche Zahl. Der Exponent bezieht sich immer auf die Basis 10.

Bisher haben wir alle Zahlen im Dezimalsystem angegeben. Perl erlaubt darüber hinaus auch die Angabe von Zahlen im Binär-, Oktal- sowie im Hexadezimalsystem:

```
0b0011  # binäre Darstellung von 3 (0*8 + 0*4 + 1*2 + 1*1))
0123;   # oktale Darstellung von 83        (1*64  + 2*8  + 3)
0x123   # hexadezimale Darstellung von 291 (1*256 + 2*16 + 3)
```

Auf den ersten Blick erscheinen diese Zahlendarstellungen sehr unhandlich und man fragt sich zurecht, welcher Programmierer so wahnsinnig wäre, statt mit Dezimalzahlen freiwillig mit Oktal- oder Hexadezimalzahlen zu rechnen? An sich keiner – wenn man mal von meinem Bruder absieht (aber der hat auch Mathematik studiert und in der Schule freiwillig den Griechisch-Leistungskurs belegt, was ihn in meinen Augen irgendwie disqualifiziert). Andererseits stehen das Oktal- und das Hexadezimalsystem dem Binärsystem, in dem der Computer rechnet, sehr nahe und dies kann in bestimmten Fällen ein Vorteil sein. Am Ende dieses Kapitels werden wir einen kleinen Ausflug in die Theorie der Binärdarstellung von Daten unternehmen, in dessen Verlauf diese Nähe anhand der Funktion `chmod` etwas genauer definiert wird.

Konstante Variablen

Neben den hier vorgestellten Text- und Zahlenkonstanten, die als direkte Werte im Quellcode angegeben werden, gibt es noch die symbolischen Konstanten, die mit dem Schlüsselwort `constant` deklariert werden, beispielsweise:

```
use constant PI => 3.1415;
```

2.3 Variablen

Mit Konstanten allein kann man nicht viel anfangen. Was man darüber hinaus benötigt, ist eine Art Zwischenspeicher für veränderliche Werte – die sogenannten Variablen.

Konstanten sind feste, unveränderliche Werte, weswegen sie im Code einfach als Wert angegeben werden können (`-3`, `124`, `"Hallo"` etc.). Der Wert einer Variablen kann sich dagegen im Laufe des Programms ändern, weswegen es schlichtweg nicht möglich ist, eine Variable als Wert anzugeben. Stattdessen gibt man Variablen Namen. Über diesen Namen kann der Programmierer auf die Variable zugreifen – entweder um einen neuen Wert in ihr abzulegen oder um den aktuellen Wert abzufragen.

Selbstverständlich steckt hinter dem Variablenkonzept mehr als nur die Vergabe eines Namens. Intern reserviert der Perl-Interpreter für jede Variable Platz im Arbeitsspeicher und verbindet den Variablennamen mit diesem Speicher. Für den Programmierer sind diese Abläufe aber zweitrangig, er arbeitet ausschließlich mit dem Variablennamen.

Variablen einrichten

Wenn Sie mit einer Variablen arbeiten möchten, besteht der erste Schritt darin, sich einen neuen, eindeutigen Namen für die Variable auszudenken, diesen dem Perl-Interpreter mitzuteilen und der Variablen einen anfänglichen Wert zuzuweisen, beispielsweise:

```
$meineVariable = 123;
```

oder

```
$stadt = "Edinburgh";
```

Ein paar Seiten weiter vorne haben Sie gelernt, dass Perl im Wesentlichen drei Datentypen kennt: Skalare, Listen und Hashes. Für jeden dieser Datentypen gibt es einen eigenen Typ von Variablen. Welchem Variablentyp eine Variable angehört, wird dabei durch das Präfix des Variablennamens festgelegt.

Variable	Datentyp	Präfix
Skalar	Zahlen und Zeichen	$
Array	Liste (von Werten)	@
Hash	Hash	%

Tabelle 2.2: Präfixe der Variablentypen

Im Moment beschränken wir uns auf die Skalare – also Variablen, deren Namen mit einem `$` beginnen und die als Werte einzelne Zahlen oder Strings aufnehmen können.

Regeln für Bezeichner

Auf das Präfix folgt der eigentliche Variablenname. Dieser Variablenname darf aus Buchstaben, Ziffern und Unterstrichen bestehen, aber nicht mit einer Ziffer anfangen.

Gültige Variablennamen wären also:

```
$meineVariable
$meine_variable
$_meineVariable
$meine3teVariable
```

Nicht gültig wären:

```
$3teVariable          # beginnt mit einer Ziffer
$meine%Variable       # enthält ungültiges Zeichen %
```

Variablen initialisieren

Schließlich wird der Variablen noch mit Hilfe des Gleichheitszeichens (des sogenannten Zuweisungsoperators) ein anfänglicher Wert zugewiesen. Dieser Wert muss dem Datentyp der Variablen entsprechen, d.h. einem Skalar kann man eine Zahl oder einen String zuweisen, nicht aber eine Liste von Zahlen (siehe Kapitel »Listen, Arrays und Hashes«).

Eindeutigkeit und Groß- und Kleinschreibung

Zu Beginn dieses Abschnitts wurde kurz erwähnt, dass Variablennamen eindeutig sein müssen. Dies ist unmittelbar einsichtig, denn da Variablen nur über ihre Namen angesprochen werden, können zwei Variablen nur dann unterschieden werden, wenn sie verschiedene Namen haben.

Wenn Sie also am Anfang des Programms eine Variable `$var1` einführen:

```
$var1 = 123;
```

und dann etliche Zeilen weiter unten im Code eine neue Variable benötigen, die Sie versehentlich mit dem gleichen Namen einzuführen versuchen:

```
...
$var1 = 3.5;
```

so haben Sie keine zweite Variable `$var1` erzeugt, sondern lediglich der bereits bestehenden Variablen `$var1` einen neuen Wert (3.5) zugewiesen. (Dies ist syntaktisch vollkommen korrekt, kann aber zu schweren Fehlern führen, wenn Sie `$var1` später in dem Glauben verwenden, die Variable enthalte immer noch den Wert `123`.)

Die Forderung nach der Eindeutigkeit der Variablennamen wirft die Frage auf, ob Perl zwischen Groß- und Kleinschreibung unterscheidet, sprich ob die Bezeichner `$meineVariable` und `$meinevariable` identisch sind oder nicht. Die Antwort zu dieser Frage lautet: »Perl unterscheidet zwischen Groß- und Kleinschreibung und `$meineVariable` und `$meinevariable` bezeichnen zwei verschiedene skalare Variablen.«

Für Programmierer, die bereits mit Sprachen wie C, C++ oder Java gearbeitet haben, dürfte dies keine Schwierigkeit darstellen, da diese Programmiersprachen ebenfalls zwischen Groß- und Kleinschreibung unterscheiden. Für Programmieranfänger oder Programmierer, die bisher nur mit Pascal oder Delphi gearbeitet haben, gibt es hier aber eine teuflische Fehlerquelle.

Wenn Sie nämlich eine Variable `$meinevar` einführen:

```
$meinevar = 123;
```

und dieser später mit folgender Anweisung einen neuen Wert zuweisen wollen:

```
...
$meineVar = 3.5;
```

so haben Sie einen schwerwiegenden Fehler begangen. Statt der alten Variablen einen neuen Wert zuzuweisen, haben Sie eine neue Variable eingeführt! Dies ist syntaktisch zwar vollkommen korrekt (und wird daher vom Perl-Interpreter nicht beanstandet), wird aber mit ziemlicher Sicherheit dazu führen, dass Ihr Programm falsche Ergebnisse berechnet.

Um in größeren Skripten Programmfehler durch die ungewollte Wiederverwendung oder Neueinrichtung von Variablen zu verhindern, empfiehlt es sich, das Pragma `use strict` zu verwenden (siehe Kapitel 6.6).

Variablen verwenden

Nachdem Sie eine Variable eingerichtet haben, wollen Sie sie auch verwenden. Was aber kann man mit einer Variablen machen?

- ✘ Sie können einer Variablen einen neuen Wert zuweisen
- ✘ Sie können den aktuellen Wert einer Variablen abfragen

Werte zuweisen

Um einer Variablen einen Wert zuzuweisen, verwendet man das Gleichheitszeichen = (den sogenannten Zuweisungsoperator). Links des Operators steht der Name der Variablen, rechts der Wert, der der Variablen zugewiesen werden soll.

```
$variable = 123;
```

Auffällig ist dabei, dass die Zuweisung genauso aussieht wie die Einrichtung der Variablen. Schauen wir uns dazu ein kleines Beispielprogramm an.

```
1:  #!/usr/bin/perl -w
2:
3:  $variable = 1;
4:  $variable = 2;
5:
6:  print $variable;
```

Die Zeilennummern gehören nicht zum Quelltext. Sie dienen uns lediglich als Referenz bei der Analyse der Quelltexte.

Was ist der Unterschied zwischen den Anweisungen der Zeilen 3 und 4?

- In Zeile 3 wird eine neue skalare Variable namens `$variable` eingerichtet und mit dem Wert 1 initialisiert. (Für Fortgeschrittene: Intern bedeutet dies, dass für die Variable Speicher reserviert und der Variablenname in die Symboltabelle des Perl-Interpreters eingetragen wird.)
- In Zeile 4 wird der Variablen `$variable` ein neuer Wert zugewiesen. Da der Bezeichner `$variable` dem Interpreter bereits aus Zeile 3 bekannt ist, richtet er keine neue Variable ein, sondern weist lediglich der alten Variablen einen neuen Wert zu.

Bisher haben wir den Variablen stets Konstanten als Werte zugewiesen. Der Nutzen der Variablen wäre allerdings sehr beschränkt, wenn dies die einzige Form der Wertzuweisung wäre. Glücklicherweise gibt es noch eine Reihe anderer Möglichkeiten, die ich Ihnen an dieser Stelle schon einmal kurz vorstellen möchte:

```
$var1 = 1;              # konstanter Wert
$var2 = $var1;          # Wert einer anderen Variablen
$var3 = 3 * $var1;      # berechneter Wert
$var1 = $var1 * 2;      # Wert, der auf Basis des alten Werts
                        # berechnet wird
$var4 = meineFunktion; # Rückgabewert einer Funktion
```

Eine skalare Variable enthält immer genau einen Wert. Wenn man einer Variablen einen neuen Wert zuweist, geht der alte Wert verloren.

Werte abfragen

Variablen sind wie Schubladen, in denen man irgendwelche Gegenstände aufbewahrt. Wenn keiner kommt, der die Schublade aufzieht, um zu sehen, was darin ist, hätte man den Gegenstand auch gleich wegwerfen können. Wenn wir also einen Wert in einer Variablen abspeichern, dann üblicherweise in der Absicht, später noch einmal auf den Wert in der Variablen zuzugreifen.

Das Abfragen des Wertes einer Variablen erfolgt – ebenso wie das Zuweisen eines neuen Wertes – über den Variablennamen. Entscheidend ist dabei, wo der Variablenname auftaucht.

- ✘ Taucht der Variablenname in einem Ausdruck auf, d.h. auf der rechten Seite einer Zuweisung, in einer Bedingung (siehe 4.2) oder in der Argumentenliste einer Funktion (siehe 6.3), so ersetzt der Perl-Interpreter den Variablennamen durch den aktuellen Wert der Variablen.
- ✘ Taucht der Variablenname auf der linken Seite einer Zuweisung auf, ersetzt der Perl-Interpreter den aktuellen Wert der Variablen durch den Wert des Ausdrucks auf der rechten Seite.

```
1:  #!/usr/bin/perl -w
2:
3:  $var1 = 2;
4:  $var1 = $var1 * 3;
5:
6:  print $var1;
```

Ausgabe:

6

In Zeile 3 wird die skalare Variable `$var1` eingerichtet und mit dem Wert 1 initialisiert.

In der darauf folgenden Zeile taucht die Variable `$var1` gleich zweimal auf: einmal auf der linken und einmal auf der rechten Seite des Zuweisungsoperators. Was passiert hier? Der Perl-Interpreter wertet zunächst die rechte Seite aus. Er ersetzt den Variablennamen durch den aktuellen Wert der Variablen (also 2) und multipliziert diesen mit 3. Das Ergebnis (6) weist er dann der Variablen zu, die auf der linken Seite der Zuweisung steht – in unserem Falle also wieder `$var1`.

In Zeile 6 wird der Wert der Variablen `$var1` ausgegeben. `print` ist eine in den Perl-Bibliotheken vordefinierte Funktion, der als Argument die auszugebenden Daten übergeben werden. In unserem Fall übergeben wir die Variable `$var1`. Der Perl-Interpreter ersetzt den Variablennamen durch den aktuellen Wert der Variablen und dieser Wert wird von `print` ausgegeben.

Wenn in einem Ausdruck ein noch nicht definierter Variablenname auftaucht (was durch Tippfehler schnell passieren kann), erzeugt der Perl-Interpreter die zugehörige Variable und setzt sie auf einen undefinierten Wert (siehe Abschnitt zum Löschen von Variablen). Im Ausdruck wird der »Wert« der Variablen dann als 0 oder als leerer String `""` interpretiert.

```
$meineVar = 123;
$meineVar = $meinevar * 2;
print "$meineVar\n";        # Ausgabe: 0
```

Um sich vor solchen Programmfehlern zu schützen, empfiehlt es sich, das Pragma `use strict` zu verwenden (siehe Kapitel 6.6).

Werte löschen

Der Vollständigkeit halber möchte ich an dieser Stelle noch erwähnen, dass man den Wert einer Variablen löschen kann, indem man die Funktion `undef` auf die Variable anwendet.

```
$var = 1;
undef $var;
```

Die Variable existiert danach noch, enthält aber keinen definierten Wert mehr. Wird ihr Wert in einem Ausdruck abgefragt, interpretiert der Perl-Interpreter den undefinierten Wert als 0 oder als leeren String `""`. Sinn ergibt die Anwendung von `undef` allerdings nur, wenn Sie später die Funktion `defined` verwenden, um zu prüfen, ob die Variable einen definierten Wert enthält oder nicht.

Als Anfänger könnte man nun auf die Idee kommen, dass man mit `undef` Variablen nach Bedarf ein- und ausschalten kann. Dies ist zwar denkbar, zeugt aber meist von schlechtem Programmierstil. Besser ist es, lokale Variablen zu verwenden (siehe Kapitel 6.6) und ansonsten darauf zu achten, dass alle Variablen bei der Einrichtung einen Wert zugewiesen bekommen.

2.4 Anweisungen und Ausdrücke

Ein Programm ist letztendlich nichts anderes als eine Folge von Befehlen, die an den Prozessor des Rechners gerichtet sind und die dieser der Reihe nach ausführt.

Anweisungen

In Perl bezeichnen wir die Befehle als »Anweisungen« und schließen Sie mit einem Semikolon ab. Betrachten wir dazu noch einmal das kleine Perl-Skript aus dem vorangehenden Abschnitt:

```
1:  #!/usr/bin/perl -w
2:
3:  $var1 = 2;
4:  $var1 = $var1 * 3;
5:
6:  print $var1;
```

Die erste Zeile ist ein Kommentar. Da Kommentare vom Perl-Interpreter ignoriert werden, tauchen Sie im Programmcode überhaupt nicht auf, stellen also keine Anweisungen an den Prozessor dar.

Danach folgen drei typische Anweisungen:

- ✘ In Zeile 3 wird die Variable `$var1` definiert. Der Prozessor wird aufgefordert, Speicher für die Variable `$var1` zu reservieren und der Variablen den anfänglichen Wert 2 zuzuweisen.
- ✘ Zeile 4 ist eine Zuweisung. Hier wird der Prozessor aufgefordert, der Variablen `$var1` den Wert des Ausdrucks auf der rechten Seite des Gleichheitszeichens zuzuweisen (weswegen das Gleichheitszeichen in Perl als Zuweisungsoperator bezeichnet wird).
- ✘ In Zeile 6 wird die Funktion `print` aufgerufen. Der an den Prozessor gerichtete Befehl heißt in diesem Fall einfach: »Rufe die Funktion `print` auf und übergib ihr als Argument den Wert der Variablen `$var1`.«

Betrachtet man das obige Listing, so fällt auf, dass die einzelnen Anweisungen nicht nur mit einem Semikolon abgeschlossen wurden, sondern auch jede in einer eigenen Zeile steht. Dies dient jedoch lediglich der Übersichtlichkeit. Wer will, kann durchaus mehrere Anweisungen in eine Zeile packen.

```
#!/usr/bin/perl -w

$var1 = 1; $var1 = $var1 * 3; print $var1;
```

Anweisungen und Maschinenbefehle

In Kapitel 1 haben Sie erfahren, dass jeder Prozessor nur einen bestimmten Satz von elementaren Maschinenbefehlen versteht und es die Aufgabe des Perl-Interpreters ist, den Code des Perl-Skripts in Maschinenbefehle umzuwandeln. Dem interessierten Leser wird sich dabei unter Umständen die Frage aufdrängen, welche Beziehung denn zwischen den Perl-Anweisungen und den Maschinenbefehlen besteht. Ist es vielleicht gar so, dass jede Perl-Anweisung genau einem Maschinenbefehl entspricht und der Perl-Interpreter lediglich den als ASCII-Text vorliegenden Befehl in Binärformat umwandeln muss. Nein, so einfach ist es nicht. Die Maschinenbefehle des Prozessors sind größtenteils weit elementarer als die Perl-Anweisungen, d.h. einer Anweisung entsprechen meist mehrere Maschinenbefehle.

Für einfache Anweisungen kommt die sprachliche Beschreibung der Anweisung den zugrunde liegenden Maschinenbefehlen schon recht nahe:

```
$var2 = $var1 * 3;
```

1. Hole den Wert der Variablen `$var1`.
2. Multipliziere diesen Wert mit 3.
3. Speichere das Ergebnis in der Variablen `$var2`.

In anderen Fällen – beispielsweise bei Funktionsaufrufen oder in Anweisungen, in denen Strings bearbeitet werden – laufen im Hintergrund weit mehr Operationen ab, als man aus der Anweisung ablesen kann.

Ausdrücke

Ein weiterer programmiertechnischer Begriff, mit dem man vertraut sein sollte, ist der Begriff des »Ausdrucks«. Im letzten Abschnitt von Unterkapitel 3.1 wurde der Begriff bereits anhand der Analogie zur mathematischen Gleichung eingeführt. An dieser Stelle möchte ich Ihnen eine genauere Definition nachreichen.

- ✘ Als Ausdruck bezeichnet man eine Kombination aus Werten und Operatoren (wobei es sich bei den Werten um Konstanten, Variablen oder Rückgabewerte von Funktionen handeln kann).
- ✘ Die Operatoren und Werte müssen so verknüpft sein, dass der Ausdruck berechnet werden kann und als Ergebnis einen Wert liefert.
- ✘ Ausdrücke findet man auf der rechten Seite von Zuweisungen, in Bedingungen (Boolesche Ausdrücke, siehe 4.2) oder als Argumente von Funktionen.

Im einfachsten Fall besteht ein Ausdruck aus einem einzigen Wert. In den folgenden Beispielen stellen die rechten Seiten der Zuweisungsoperatoren Ausdrücke dar:

```
$var1 = 11;
$var2 = $var1;
```
[4]

Komplexere Ausdrücke entstehen durch Verknüpfung der Werte mit Operatoren:

```
$var1 = 11 + 3;
$var2 = (14 * $var1) - 12;
```

Die Ausdrücke werden uns noch weiter beschäftigen, insbesondere in Kapitel 3 und in Kapitel 4.2 zur Bildung Boolescher Ausdrücke.

2.5 Ein- und Ausgabe von Daten

Programme dienen dazu, Daten zu verarbeiten. Meist sieht das so aus, dass das Programm Daten einliest (über die Tastatur, aus einer Datei, von einem Webbrowser), diese in irgendeiner Weise manipuliert und verarbeitet und die Ergebnisse dann ausgibt (auf den Bildschirm, in eine Datei).

Das Einlesen und Ausgeben von Daten ist ein ebenso wichtiges wie umfangreiches Gebiet, dem ich im zweiten Teil des Buches ein eigenes Kapitel gewidmet habe. Gerade aber weil das Thema so interessant und wichtig ist, sollten wir nicht bis zum zweiten Teil des Buches damit warten. Schauen wir uns also vorab schon einmal an, wie man Daten über die Tastatur vom Anwender einliest und Daten auf die Konsole ausgibt.

Daten einlesen

Ein- und Ausgabe erfolgen in Perl grundsätzlich über sogenannte Datei-Handles. Diese Datei-Handles repräsentieren in unseren Perl-Skripten die Quelle oder das Ziel eines Datenflusses. Quelle und Ziel müssen dabei nicht immer wirkliche Dateien sein. Für bestimmte, typische Quellen und Ziele gibt es spezielle, vordefinierte Datei-Handles – so etwa den Datei-Handle `STDIN` für das Standardeingabegerät (sprich die Tastatur) oder `STDOUT` für das Standardausgabegerät (die Konsole).

4 In diesem Beispiel treten links und rechts des Zuweisungsoperators Variablennamen auf, doch nur die rechte Variable (`$var1`) steht für einen Ausdruck, denn nur auf der rechten Seite interpretiert der Perl-Interpreter den Variablennamen als den Wert der Variablen. Auf der linken Seite wird der Variablenname als Speicherplatz interpretiert, in den der Wert des Ausdrucks auf der rechten Seite geschrieben wird.

Wie kann man nun mit Hilfe von STDIN Daten über die Tastatur einlesen? Üblicherweise geschieht dies mit folgender Konstruktion:

```
$meineVar = 0;
$meineVar = <STDIN>;
chomp($meineVar);
```

Zuerst definiert man eine Variable. In dieser Variablen wird im nächsten Schritt die über die Tastatur eingelesene Eingabe abgelegt. (Der Wert, mit dem die Variable initialisiert wird, ist daher nicht so wichtig. Meist initialisiert man die Variablen in solchen Fällen mit dem Wert 0.)

Im nächsten Schritt werden die Daten eingelesen. Dazu wendet man auf den Datei-Handle den Eingabeoperator <> an, was dann wie folgt aussieht: <STDIN>. Der Eingabeoperator liest so lange Daten aus der Datenquelle, die durch den übergebenen Datei-Handle spezifiziert ist, bis er auf ein bestimmtes Trennzeichen trifft. Per Voreinstellung ist dieses Trennzeichen das Neue-Zeile-Zeichen. Die Konstruktion <STDIN> liest also so lange Zeichen über die Tastatur ein, bis der Anwender die ↵ -Taste drückt.

Die eingelesenen Daten speichern wir in der Variablen ab:

```
$meineVar = <STDIN>.
```

Dabei ist zu beachten, dass das Neue-Zeile-Zeichen mit abgespeichert wird. Da das Neue-Zeile-Zeichen allerdings meist unerwünscht ist (es gehört ja im Grunde nicht zur Eingabe, sondern diente lediglich zum Abschicken der Eingabe), ist es Usus, das Neue-Zeile-Zeichen gleich aus der Eingabe zu entfernen. Dies erledigt die vordefinierte Funktion chomp, der die Variable mit der Eingabe übergeben wird.

Statt

```
$meineVar = <STDIN>;
chomp($meineVar);
```

schreibt man meist verkürzt:

```
chomp($meineVar = <STDIN>);
```

Daten mit print ausgeben

Um Daten auf die Konsole auszugeben, verwendet man die Funktionen print und printf. Beginnen wir mit der Funktion print, die Sie schon aus einigen der vorangehenden Beispielskripten kennen.

Die `print`-Funktion erwartet als erste Angabe den Datei-Handle des Ausgabeziels und danach eine Liste der auszugebenden Daten, beispielsweise:

```
print STDOUT "Hallo Welt\n";
```

oder

```
print STDOUT "Hallo Welt", "\n";
```

»Augenblick!«, werden Sie jetzt einwerfen, »Wieso haben wir dann bisher nie den Datei-Handle `STDOUT` angegeben?« Da die Konsole mit Abstand das häufigste Ziel von Datenausgaben ist, wurde `print` so implementiert, dass die Funktion automatisch `STDOUT` als Datei-Handle verwendet, wenn der Programmierer keinen eigenen Datei-Handle angibt. Statt

```
print STDOUT "Hallo Welt\n";
```

kann man also verkürzt schreiben:

```
print "Hallo Welt\n";
```

Auf die gleiche Weise kann man Werte von Variablen ausgeben:

```
print $meineVar;
```

Wer will, kann die Ausgabe von Text und Variablenwerten kombinieren – entweder durch Verwendung des Kommas:

```
print $var1, " ist kleiner als ", $var2, "\n";
```

oder durch Variablenexpansion (siehe Abschnitt »Einfache versus doppelte Anführungszeichen«):

```
print "$var1 ist größer als $var2\n";
```

Umlaute auf der Windows-Konsole

Wenn Sie Text mit deutschen Umlauten auf die Windows-Konsole (MSDOS-Eingabeaufforderung) ausgeben, sehen Sie statt der Umlaute nur grafische Sonderzeichen. Dies liegt daran, dass die Windows-Konsole einen anderen Zeichensatz (OEM-Zeichensatz) verwendet als Windows. Sie können dieses Manko beheben, indem Sie statt der Zeichen den ASCII-Code angeben:

```
print "Umlaute \x84 \x94 \x81 \n";
```

Das folgende Programm demonstriert noch einmal zusammengefasst das Einlesen und Ausgeben von Daten:

Listing 2.1: euro1.pl – rechnet DM-Angaben in Euro um

```
#!/usr/bin/perl -w
$dm = 0;
$euro = 0;

print "Geben Sie einen Betrag in DM ein: ";
chomp ($dm = <STDIN>);                        # Zahl einlesen

$euro = $dm / 1.95583;                        # Umrechnen in Euro
print "$dm DM entsprechen $euro Euro\n";      # Ausgabe
```

Ausgabe:

```
Geben Sie einen Betrag in DM ein: 100.34
100.34 DM entsprechen 51.3030273592286 Euro
```

Formatierte Ausgabe mit printf

Wenn Sie sich die Ausgabe von Listing 3.1 anschauen, wird Ihnen die unschöne Formatierung des errechneten Euro-Betrags auffallen. Zwar ist es immer gut, in Finanzfragen möglichst korrekt zu sein, doch die Angabe des Euro-Betrags mit einer Genauigkeit von 13 Stellen ist doch etwas übertrieben und führt dazu, dass die Ausgabe unleserlich wirkt. Es wäre daher schön, wenn wir eine Möglichkeit hätten, die Ausgabe zu formatieren – etwa um die Ausgabe des Euro-Betrags auf zwei Nachkommastellen zu begrenzen. Dies erlaubt die `printf`-Funktion.

```
printf HANDLE FORMAT DATENLISTE
```

- ✘ Wie `print` erwartet auch `printf` die Angabe eines Datei-Handles, und wie bei `print` gilt, dass die Funktion `STDOUT` verwendet, wenn der Programmierer keinen eigenen Datei-Handle angibt.
- ✘ `FORMAT` steht für einen sogenannten Formatstring, `DATENLISTE` ist die durch Kommata getrennte Liste der Daten, die ausgegeben und dabei gemäß den Angaben im Formatstrings formatiert werden sollen. C-Programmierern wird diese Konstruktion wohl vertraut sein, allen anderen sei der Aufbau anhand eines Beispiels kurz erläutert.

Formatstrings

Grundsätzlich entspricht ein Formatstring einem String mit Variablenexpansion – nur dass im Formatstring statt der Variablennamen spezielle Platzhalter stehen und die Variablennamen, deren Werte bei der Ausgabe die Platzhalter ersetzen, in der Datenliste nachgereicht werden. Aus dem String

```
"$dm DM entsprechen $euro Euro\n";
```

würde dadurch die Konstruktion

```
"%f DM entsprechen %f Euro\n", $dm, $euro;
```

wobei »`"%f DM entsprechen %f Euro\n"`« den Formatstring und »`$dm, $euro`« die Datenliste darstellen.

Wie Sie sehen, besteht jeder Platzhalter aus einem %-Zeichen und einem Buchstaben. Dieser Buchstabe gibt den Datentyp an, den der ausgegebene Wert haben soll. Die wichtigsten Datentypen können Sie Tabelle 2.3 entnehmen.

Tabelle 2.3: Platzhalter für printf

Platzhalter	**Datentyp**
%c	einzelnes Zeichen
%s	String
%d	Ganzzahl
%f	Gleitkommzahl
%e	Gleitkommazahl in Exponentialschreibweise
%o	oktale Ganzzahl
%x	hexadezimale Ganzzahl
%%	kein Platzhalter, sondern Ausgabe des %-Zeichens

Der nächste Schritt besteht darin, im Platzhalter anzugeben, wie der zugehörige Wert formatiert werden soll. Beispielsweise können Sie angeben, wie viele Zeichen der Wert in der Ausgabe mindestens einnehmen soll:

```
printf "DM %7d \n", $dm;    # erzeugt: DM .....10 DM[5]
```

Für Gleitkommazahlen können Sie zusätzlich die Anzahl der Nachkommstellen angeben:

```
printf "DM %7.2f \n", $dm;  # erzeugt: DM ..10.00 DM
```

Sie können auch die Breitenangabe weglassen und nur die Anzahl der Nachkommstellen festlegen:

```
printf "DM %.2f \n", $dm;   # erzeugt: DM 10.00 DM
```

5 Die Punkte erscheinen nicht in der Ausgabe, sie sollen hier nur den Effekt verdeutlichen.

Tabelle 2.4: Formatflags für printf

Formatierung	Effekt	Beispiel
+	Zahlen immer mit Vorzeichen	"%+d"
Leerzeichen	pos. Zahlen mit Leerzeichen statt +-Symbol	"% d"
–	linksbündige Ausrichtung	"%-d"
b	Angabe der Feldbreite der Ausgabe	"%5d"
#	Präfix 0 bzw. 0x für Oktal- bzw. Hexadezimalzahlen	"%#o"

Mit diesen Informationen können wir das Skript aus Listing 3.1 unter Verwendung von `printf` umschreiben.

Listing 2.2: euro2.pl – Formatierte Ausgabe mit printf

```
#!/usr/bin/perl -w
$dm = 0;
$euro = 0;

print "Geben Sie einen Betrag in DM ein: ";
chomp ($dm = <STDIN>);

$euro = $dm / 1.95583;
printf "%.2f DM entsprechen %.2f Euro\n", $dm, $euro;
```

2.6 Vertiefung: Zahlen und Strings – ein ungleiches Geschwisterpaar

Das Besondere an den Variablen ist, dass man ihnen im Laufe des Programms nacheinander verschiedene Werte zuweisen kann, solange der Typ dieser Werte dem Datentyp der Variablen entspricht. Für die skalaren Variablen bedeutet dies, dass man ihnen sowohl Zahlen als auch Strings zuweisen kann, ja man kann sogar ein und derselben Variablen im Laufe des Programms nacheinander Zahlen und Strings zuweisen.

```
$meineVar = 1;
...
$meineVar = 3.1415;
...
$meineVar = "Hallo";
```

und so fort.

Streng typisierte Programmiersprachen

Was für Perl-Programmierer (fast) ganz normal ist, dürfte bei Programmierern, die bereits mit Sprachen wie Pascal, C/C++ oder Java gearbeitet haben, Erstaunen, ja gar Besorgnis auslösen, denn in diesen Sprachen wird zwischen Zahlen und Strings, ja sogar zwischen verschiedenen Zahlentypen (nämlich zwischen Ganzzahlen und Gleitkommazahlen) streng unterschieden. Diese Unterscheidung kommt nicht von ungefähr, sondern gründet darauf, dass Ganzzahlen, Gleitkommazahlen und Strings von Natur aus verschiedene Datentypen darstellen (mit eigenen Formaten und Operationen), die auch auf Maschinenbefehlebene unterschieden werden. Was die streng typisierten Programmiersprachen machen, ist also nichts anderes, als die Unterscheidung, die auf Maschinenbefehlebene stattfindet, an den Programmierer weiterzugeben.

Wenn Sie in einer dieser Sprachen eine Variable deklarieren, müssen Sie konkret angeben, ob diese Variable für Ganzzahlen (123, -93), Gleitkommazahlen (3.255, 9e5) oder Strings ("Hallo") sein soll. Danach können Sie der Variablen nur Werte des vereinbarten Typs zuweisen, einer Variablen für Ganzzahlen also beispielsweise nur Ganzzahlen. Wenn Sie doch einmal einer Ganzzahl-Variablen eine Gleitkommazahl oder einen String mit einer Zahlenangabe zuweisen wollen, müssen Sie dazu spezielle Funktionen oder Konvertierungsmöglichkeiten in Anspruch nehmen.

Der Perl-Weg

Perl unterscheidet ebenfalls zwischen Ganzzahlen, Gleitkommazahlen und Strings (man beachte beispielsweise die unterschiedlichen Formate für Text- und Zahlenkonstanten), verwischt aber die Grenzen zwischen diesen Datentypen, indem es für alle diese Daten einen einzigen Variablentyp, den Skalar, verwendet.

Die strenge Typisierung fällt damit weg. Der Programmierer kann in seinen skalaren Variablen beliebige Daten (Ganzzahlen, Gleitkommazahlen, Strings) speichern, ohne sich dabei mit Fragen bezüglich der Kompatibilität und Konvertierbarkeit dieser Daten herumschlagen zu müssen.

Ein Variablentyp – drei Datentypen

Für den Anfänger ist diese nicht ganz so strenge Typenunterscheidung auf jeden Fall ein Vorteil. Das einzige Problem ist, dass Sie – sofern Sie nicht schon einmal in C, Pascal oder einer ähnlich typisierten Sprache programmiert haben – diesen Vorteil gar nicht zu schätzen wissen. Lassen Sie mich also Ihre Sinne ein wenig schärfen.

Betrachten wir dazu folgenden Quelltext:

```
$meineVar = 123;
$meineVar = 234.567;
$meineVar = "Hallo";
```

In diesem Code-Fragment wird ein Skalar definiert und mit dem Ganzzahl-Wert 123 initialisiert. In C würde man `meineVar` dazu als Variable vom Typ `int` (für integer = Ganzzahl) deklarieren.

Danach wird der Variablen eine Gleitkommazahl zugewiesen (234.567). In C wäre dies mit einer internen Konvertierung verbunden. Da man in C in einer `int`-Variablen keine Gleitkommazahlen ablegen kann, würde der C-Compiler die Gleitkommazahl 234.567 durch Wegstreichen des Nachkommateils in eine Ganzzahl umwandeln. In `meineVar` stünde danach also der Wert 234 und nicht 234.567 wie in Perl.

In der dritten Zeile wird der Variablen ein String zugewiesen. In Perl ist dies kein Problem. In C wäre dies schlichtweg unmöglich.

Was steckt hinter den Skalaren?

Perl-Neulinge, die von Sprachen wie C oder Java kommen, werden sich sicherlich nicht mit der Erklärung zufrieden geben, dass Perl die elementaren Daten einfach über einen Kamm schert,. »Ja, okay, der Sinn ist klar: Die Programmierung vereinfacht sich durch die Einführung des globalen Skalar-Variablentyps. Aber wie funktioniert das? Wie kann das überhaupt funktionieren?«

Die Perl-Programmierwerkzeuge sind selbst in C programmiert. Ein Blick in den C-Quellcode dieser Werkzeuge verrät uns, was sich hinter dem Datentyp des Skalars verbirgt: eine Struktur namens SV[6], die in der Datei SV.H definiert ist.

```
struct sv {
    void*   sv_any;      /* Zeiger auf Daten */
    U32     sv_refcnt;   /* Referenzzähler */
    U32     sv_flags;    /* Typ des Skalars */
};
```

Interessant für uns sind vor allem die Elemente `sv_any` und `sv_flags`. Wird einem Skalar eine Ganzzahl zugewiesen, wird diese Ganzzahl im Arbeitsspeicher abgelegt und der `sv_any`-Zeiger des Skalars auf diese Ganzzahl gerichtet. Gleichzeitig wird in `sv_flags` festgehalten, dass der Skalar eine Ganzzahl enthält. Wird demselben Skalar später ein String zugewiesen, wird der `sv_any`-Zeiger des Skalars umgebogen, so dass er nicht mehr auf die Ganzzahl, sondern auf den String verweist. In `sv_flags` wird festgehalten, dass der Skalar jetzt einen String beinhaltet.

6 C-Strukturen sind Datentypen, die der Programmierer selbst – als eine Kombination von Daten anderer Datentypen – definieren kann. Von ihrer Bedeutung her entsprechen Sie den Hashes von Perl.

Die Zuweisung von Daten unterschiedlicher Datentypen wird also durch die Zwischenschaltung des `void`-Zeigers erreicht. Mit Hilfe des `sv_flags`-Elements behält der Perl-Interpreter jederzeit die Übersicht darüber, welchem Datentyp die im Skalar abgelegten Daten angehören.

Wer mehr über die interne Datenverwaltung von Perl wissen will, dem seien die Dokumentationen PERLNUMBER und PERLGUTS empfohlen.

Konvertierung von Zahlen in Strings

Perl sieht nicht nur einen gemeinsamen Variablentyp für Ganzzahlen, Gleitkommazahlen und Strings vor, es kann auch automatisch zwischen diesen Datentypen hin- und herkonvertieren.

```
$meineVar = 123;
print $meineVar;
```

Hier wird zuerst in `$meineVar` eine Ganzzahl gespeichert. Danach wird der Wert der Variablen mit `print` ausgegeben. Da auf die Konsole aber nur Text ausgegeben werden kann, muss der Wert von `$meineVar` in einen String ("123") umgewandelt werden. Für Perl ist dies kein Problem.

✘ Zahlen werden in String-Kontexten (Übergabe an String-Funktionen, als Operand zu String-Operatoren) automatisch in Ziffern umgewandelt.

```
$zahl1 = 123;
$zahl2 = 456;
$string = $zahl1 . $zahl2;    # Aneinanderreihung mit
                              # Stringoperator .
print "$string\n";            # Ausgabe: 123456
```

✘ Für C-Programmierer sei angemerkt, dass die Konvertierung üblicherweise mit Hilfe der C-Funktion `sprintf()` und dem Formatspezifizierer `%.15g` vorgenommen wird.

Konvertierung von Strings in Zahlen

Umgekehrt können auch Strings in Zahlen umgewandelt werden:

```
$meineVar = 123;
$meineVar = $meineVar + "420";
print "$meineVar\n";
```

Hier wird zu dem Ganzzahlwert einer Variablen ein String aus Ziffern addiert. Perl wandelt in solchen Fällen den String automatisch in die entsprechende Zahl (im Beispiel also 420) um. In anderen Programmiersprachen muss man für die Umwandlung von String-Darstellungen von Zahlen in echte Zahlen meist spezielle Funktionen aufrufen.

- Strings werden in numerischen Kontexten (Übergabe an mathematische Funktionen, als Operand zu arithmetischen Operatoren) automatisch in Zahlen umgewandelt.

```
$zahl1 = 123;
$zahl2 = 456;
$string = $zahl1 + $zahl2;   # Addition mit
                             # arithmet. Operator +
print "$string\n";           # Ausgabe: 579
```

- Führende Leerzeichen und beliebige nachfolgende Zeichen werden bei der Konvertierung ignoriert.

```
"1234"          wird zu 1234
"1.4e3"         wird zu 1400
"   1234"       wird zu 1234
"1234 Teile"    wird zu 1234
"12e3,4"        wird zu 12000
```

- Ist eine korrekte Konvertierung nicht möglich (weil die Zahlendarstellung im String keinem der unterstützten Zahlenformate entspricht), wird der String zu 0 ausgewertet.

```
"a1234"         wird zu 0
```

- Hexadezimal- und Oktaldarstellungen werden in Strings nicht erkannt.

```
"0123"          wird zu 123 (nicht 83)
"0x123"         wird zu 0
```

- Uninitialisierte Variablen werden zu 0 ausgewertet.
- Für C-Programmierer sei angemerkt, dass die Konvertierung üblicherweise mit Hilfe der C-Funktion `atof()` vorgenommen wird.

2.7 Vertiefung: Binärdarstellung von Daten

Bis jetzt haben wir uns nur Gedanken darüber gemacht, wie man beliebige Daten in einem Perl-Skript repräsentieren kann. Für alle die Leser, für die die Lektüre dieses Buches den Einstieg in die Programmierung überhaupt darstellt, möchte ich noch einen Schritt weiter gehen und die Frage aufwerfen, wie die Daten denn im Computer dargestellt werden?

Codierung von Daten

Computer sind relativ einfach gestrickte Wesen, die leider nicht einmal bis drei zählen können. Das heißt, zählen können sie schon bis drei und auch darüber hinaus, nur die Ziffer 3, die kennen sie nicht. Computer unterscheiden nämlich nur zwischen 1 und 0, was für sie bedeutet, dass an irgendeinem ihrer internen Schaltelemente eine Spannung anliegt oder nicht oder dass das magnetische Material der Festplatte (oder Diskette) in der einen oder anderen Richtung polarisiert ist.

Da Computer darauf beschränkt sind, Nullen und Einsen zu verarbeiten, ist man gezwungen, jegliche Daten, die man im Computer verarbeiten will, als Folgen von Nullen und Einsen darzustellen[7]. Dazu benötigt man für die verschiedenen Arten von Daten passende Codierungsverfahren.

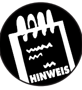
Tabelle 2.5: Codierung der elementaren Datentypen

<table>
<tr><th>Datentyp</th><th>Codierung</th></tr>
<tr><td>Ganzzahlen</td><td>Dezimale Ganzzahlen könnte man prinzipiell nach demselben Verfahren codieren, nach dem man Dezimalzahlen ins Binärsystem umwandelt (durch fortgesetzte Division durch 2, wobei die nicht teilbaren Reste der aufeinander folgenden Divisionen, von rechts nach links geschrieben, die gewünschte Binärzahl ergeben:
<table><tr><th colspan="2">97:2</th><th></th></tr><tr><td>48</td><td>1</td><td></td></tr><tr><td>24</td><td>0</td><td>97=1100001</td></tr><tr><td>12</td><td>0</td><td></td></tr><tr><td>6</td><td>0</td><td></td></tr><tr><td>3</td><td>0</td><td></td></tr><tr><td>1</td><td>1</td><td></td></tr><tr><td>0</td><td>1</td><td></td></tr></table>Diese Codierung bereitet aber Schwierigkeiten, wenn neben positiven auch negative Ganzzahlen codiert und die Rechengesetze erhalten bleiben sollen. Aus diesem Grunde werden Ganzzahlen üblicherweise nach dem sogenannten 2n+1-Komplement codiert, bei dem man das negative Pendant zu einer positiven Zahl dadurch erhält, dass man die Stellen der positiven Zahl invertiert und dann 1 addiert. Während die Zahl 3 also einfach als 011 codiert wird (Zahl 3 im Binärsystem), wird -3 als 101 (100 + 1) dargestellt.</td></tr>
</table>

7 Im Hinblick darauf, dass man Folgen von Nullen und Einsen auch als Zahlen im Binärsystem interpretieren kann, spricht man daher auch von der »digitalen Datenverarbeitung«.

Tabelle 2.5: Codierung der elementaren Datentypen (Fortsetzung)

Datentyp	Codierung
Gleitkommazahlen	Gleitkommazahlen werden in exponentieller Schreibweise, sprich als Kombination aus Vorzeichen, Mantisse und Exponent, codiert. So kann man beispielsweise die Zahl - 300,1 auch als - 0,3001 * 10^3 schreiben. Gespeichert wird die Zahl dann als Vorzeichen (1 für negatives Vorzeichen, 0 für positives Vorzeichen), Mantisse ohne Null vor dem Komma (3001) und Exponent zur Basis 10 (3).[8]
Strings	Strings wie `"Hallo"` oder `'Oops'` werden als Folgen von Zeichen gespeichert. Das Ende der Zeichenkette wird intern mit einem speziellen Terminierungszeichen markiert. Die einzelnen Zeichen werden gemäß einer Tabelle codiert. Früher wurde hierzu die erweiterte ASCII-Tabelle herangezogen, bei der die einzelnen Zeichen durch Folgen von 8 Bit[9] dargestellt wurden (siehe Anhang F). Durch 8 Bit können allerdings nur 2^8 = 256 Zeichen codiert werden. Vor einigen Jahren wurde daher der UNICODE-Standard ins Leben gerufen, bei dem die Zeichen durch 16 Bit codiert werden, wodurch der Zahlencode groß genug ist, um auch die verschiedenen asiatischen und arabischen Schriftzeichen mit einzuschließen. Perl codiert die Zeichen mittlerweile gemäß UNICODE (die arabischen Zeichen aus dem ASCII-Code werden im UNICODE übrigens durch die gleichen Zahlen codiert).



Die digitale Codierung von Daten findet man nicht nur in der Informatik. Als Beispiel aus dem alltäglichen Leben seien die Telefonnummern erwähnt. Wenn Sie im Telefonbuch die Nummer der Inlandsauskunft nachschlagen und dort den Eintrag 11833 finden, werden Sie diese Zahl sicher nicht als Elftausendachthundertdreiundreißig, sondern als nacheinander zu wählende Ziffernfolge interpretieren. Der Datentyp (Telefonnummer) legt dabei fest, wie die Zahlenfolge zu interpretieren und wie die Daten zu verarbeiten sind.

8 Der Computer berechnet und speichert Mantisse und Exponent allerdings im Binärsystem.

9 Ein Bit ist die kleinste Informationseinheit der elektronischen Datenverarbeitung. Es kann einen der Werte Null oder Eins haben. Eine Folge von 8 Bit bezeichnet man als Byte.

Eigene Codierungen

Im vorangehenden Abschnitt haben wir uns angeschaut, wie die elementaren Daten auf Maschinenebene codiert werden. Grundsätzlich haben wir als Programmierer mit dieser Codierung nicht viel zu tun – der Perl-Interpreter nimmt uns die Arbeit ab. Er wandelt die Zahlen und Strings aus unserem Quellcode in die entsprechenden Bitfolgen um und kann bei Bedarf auch Konvertierungen zwischen den Datentypen vornehmen.

Da aber nicht alle Daten aus Zahlen und Text bestehen, ist es häufig erforderlich, der internen Codierung durch den Interpreter eine weitere Codierungsebene vorzuschalten, bei der die Daten (Bilder, Sound, Messkurven, was auch immer) als Zahlenfolgen formuliert werden. Wenn Sie beispielsweise eine Messkurve darstellen wollen, bietet es sich an, die Kurve als eine Folge von Koordinaten von Punkten auf der Kurve abzuspeichern. Ein Perl-Skript kann diese Koordinaten einlesen und daraus die Kurve aufbauen. Dabei wird die Messkurve auf der ersten Ebene als eine Folge von Zahlen (Koordinaten) codiert. Auf der zweiten Ebene codiert der Perl-Interpreter die Zahlen, mit denen das Perl-Skript arbeitet, ins Binärformat.

chmod

Manchmal bietet es sich an, Daten direkt als Bitfolgen zu codieren. Linux-Anwender kennen dies von dem Betriebssystembefehl `chmod`, mit dessen Hilfe man die Zugriffsrechte für Dateien und Verzeichnisse ändern kann. Linux unterscheidet zwischen Lese-, Schreib- und Ausführungsrechten. Diese kann man bequem durch eine Folge von 3 Bit darstellen:

```
011      # nur Schreib- und Ausführungsrecht
100      # nur Leserecht
```

Diese drei Zugriffsrechte kann man für den Besitzer, die Gruppe und für alle anderen getrennt vergeben.

```
111101001  # Besitzer hat alle Rechte,
           # die Gruppe hat Lese- und Ausführungsrecht
           # alle anderen können die Datei nur ausführen
```

Bitfolgen haben allerdings die Eigenschaft, dass sie meist ziemlich lang sind. Um sich Tipparbeit zu ersparen, werden sie häufig als Zahlen interpretiert und in ein höheres Zahlensystem umgewandelt. Das Dezimalsystem ist dafür jedoch vollkommen ungeeignet, da die Umrechnung von Binärzahlen in Dezimalzahlen im Kopf praktisch nicht zu leisten ist. Anders verhält es sich mit Zahlensystemen, deren Basis eine Potenz von 2 ist – beispielsweise dem Oktalsystem (8 = 2^3). Der Exponent (3 für das Oktalsystem) gibt dabei an, wie viele Stellen im Binärsystem einer Stelle im Zielsystem entsprechen.

```
111 101 001  # Binär
 7   5   1   # Oktal
```

Aus diesem Grunde wird chmod mit Oktalzahlen aufgerufen:

```
chmod 777 demo.pl  # macht das perl-Skript für alle lesbar,
                   # überschreibbar und ausführbar
```

Passend zum Linux-Befehl chmod gibt es in Perl eine gleichnamige Funktion, chmod, mit der man aus einem Perl-Skript heraus die Zugriffsrechte einer Datei ändern kann.

```
chmod 0644 'test.pl';
```

Beachten Sie dabei die vorangestellte 0, die Perl anzeigt, dass es sich um die Oktalzahl 644 handelt und nicht um die dezimale Zahl 644 (was oktal 1204 entspricht). Beachten Sie des Weiteren, dass die führende 0 nur in Zahlen, nicht aber in Strings ("0644") als Kennzeichen einer Oktalzahl erkannt wird.

Direkte Bitprogrammierung

Perl erlaubt es Ihnen, die Bits Ihrer Variablen direkt zu manipulieren. Ich werde dieses Thema nicht weiter vertiefen, aber ich möchte Sie zumindest auf die verschiedenen Möglichkeiten zur Bitmanipulation hinweisen:

- ✘ Binäre Zahlenangaben: $var = 0b0001;
- ✘ Bitoperatoren: <<, >>, |, &, ^, ~
- ✘ Spezielle Funktionen: pack, unpack

Programm, das Text in Morsecode umwandelt

Nach so viel Theorie möchte ich Ihnen zum Abschluss noch ein kleines Perl-Skript vorstellen, das sich ebenfalls mit dem Thema Codierung beschäftigt – allerdings unter einem ganz anderen Aspekt. Das folgende Programm wandelt Texte in Morsecode um:

Listing 2.3: morse.pl

```
#!/usr/bin/perl -w

sub morsen {
  $_ = $_[0];
  / / && print '|| ';      # neues Wort
  /a/ && print '.- ';
  /b/ && print '-... ';
  /c/ && print '-.-. ';
  /d/ && print '-.. ';
  /e/ && print '. ';
  /f/ && print '..-. ';
```

```
  /g/ && print '--. ';
  /h/ && print '.... ';
  /i/ && print '.. ';
  /j/ && print '.--- ';
  /k/ && print '-.- ';
  /l/ && print '.-.. ';
  /m/ && print '-- ';
  /n/ && print '-. ';
  /o/ && print '--- ';
  /p/ && print '.--. ';
  /q/ && print '--.- ';
  /r/ && print '.-. ';
  /s/ && print '... ';
  /t/ && print '- ';
  /u/ && print '..- ';
  /v/ && print '...- ';
  /w/ && print '.-- ';
  /x/ && print '-..- ';
  /y/ && print '-.-- ';
  /z/ && print '--.. ';
  }


$text = 0;
print "Geben Sie den zu morsenden Text ein: ";
chomp ($text = <STDIN>);

while ($text =~ m/(.)/g) {
  morsen($1);
  }
print "\n";
```

Abb. 2.1: Ausführung des Morse-Skripts.

Analyse

Für den Perl-Neuling, der gerade seine ersten Schritte in der Perl-Programmierung unternimmt, dürfte der Quelltext dieses Skripts wahrscheinlich ebenso unverständlich sein wie der Morse-Code, den es erzeugt. Wir wollen daher auch gar nicht versuchen, das Skript bis ins Detail zu ergründen, sondern uns darauf beschränken, den groben Aufbau und den Ablauf des Skripts zu verstehen. Da in dem Skript etliche Techniken und Konstrukte verwendet werden, die erst in späteren Kapitel dieses Buches behandelt

werden (Funktionen, Schleifen, Standardvariablen, Reguläre Ausdrücke), gibt Ihnen das Skript nebenbei einen guten Überblick darüber, was Perl-Programmierung ist und was Sie noch im Laufe dieses Buches erwartet.

Die Ausführung des Skripts beginnt mit der Zeile:

```
$text = 0;
```

Danach wird der Anwender aufgefordert einen Text einzugeben. Dieser Text wird mit Hilfe des <>-Operators von der Tastatur (im Programm repräsentiert durch `STDIN`) eingelesen und in der Variablen `$text` gespeichert.

Danach soll der eingegebene Text Zeichen für Zeichen durchgegangen werden und die einzelnen Zeichen durch Morse-Code ersetzt werden. Dies geschieht in einer sogenannten Schleife, die durch das Schlüsselwort `while` eingeleitet wird. Im Kopf der Schleife wird `$text` mit Hilfe eines »regulären Ausdrucks« durchsucht:

```
while ($text =~ m/(.)/g) {
```

In diesem Ausdruck ist `m//` der Such-Operator zwischen dessen Schrägstrichen der zu suchende Text steht. Da wir nach beliebigen Zeichen suchen, verwenden wir als Suchbegriff den Punkt, der in regulären Ausdrücken für ein beliebiges Zeichen steht. Den Punkt setzen wir in runde Klammern, damit der Such-Operator jedes Mal, wenn er in `$text` ein Zeichen findet, dieses in der vordefinierten Variablen `$1` zwischenspeichert. Die Option `g` schließlich weist den Suchoperator an, den gesamten String in `$text` zu durchsuchen und nicht automatisch aufzuhören, wenn er das erste Zeichen gefunden hat. Die gesamte `while`-Schleife führt demnach dazu, dass der String in `$text` Zeichen für Zeichen durchgegangen. Immer wenn der Suchoperator das nächste Zeichen gefunden hat, kopiert er es in die Variable `$1` und diese wird im Rumpf der Schleife an die Funktion `morsen` übergeben.

```
while ($text =~ m/(.)/g) {
  morsen($1);
  }
```

Die Funktion `morsen` ist keine vordefinierte Perl-Funktion, sondern eine Funktion, die explizit am Anfang des Skripts definiert wurde. Sie beginnt damit, dass Sie das Zeichen, das ihr beim Aufruf übergeben wurde, aus dem vordefinierten Parameter `$_[0]` in die Standardvariable `$_` kopiert.

```
sub morsen {
  $_ = $_[0];
```

Danach folgt eine Reihe von Vergleichen, in denen geprüft wird, ob es sich bei dem Zeichen, das der Funktion übergeben wurde, um ein Leerzeichen

oder um einen Buchstaben handelt. Wenn ja, wird für das Zeichen der entsprechende Morse-Code ausgegeben.

```
/ / && print '|| ';
/a/ && print '.- ';
/b/ && print '-... ';
...
```

Die Vergleiche basieren wiederum auf regulären Ausdrücken: `/ /`, `/a/`, `/b/`. Diese regulären Ausdrücke sind einfach verkürzte Schreibweisen für:

```
$_ =~ m/ /;
$_ =~ m/a/;
$_ =~ m/b/; ...
```

2.8 Fragen und Übungen

1. Setzen Sie ein Perl-Skript auf, das das folgende Voltaire-Zitat ausgibt:

   ```
   »Wollen Sie ein Autor sein, wollen Sie ein Buch schreiben,
   dann denken Sie daran, dass es neu und nützlich oder
   zumindest sehr vergnüglich sein muss.«
   ```

2. Sind die folgenden Bezeichner gültige Variablennamen?

   ```
   $hugo
   $HUGO
   $_h_u_g_o
   $gesetzt?
   $3fach
   $dos2unix
   ```

3. Wie gibt man die Werte von Variablen aus? Wie kann man die Ausgabe formatieren?

4. Sie haben gestern einen Scheck über folgenden Betrag gefunden: 1.000.000,01 DM. Bevor Sie den Scheck im Fundbüro abgeben, überlegen Sie sich, wie man diesen Zahlenwert in einem Perl-Skript angeben könnte.

5. Trainieren wir ein wenig die Exponentialschreibweise. Seit Einstein weiß man, dass zwischen Energie und Masse die folgende Beziehung besteht: $E = m * c^2$. Schreiben Sie ein Perl-Skript, das berechnet, wie viel Energie frei wird, wenn Sie sich von einem Augenblick zum nächsten in reine Energie verwandeln würden. Die Lichtgeschwindigkeit c können Sie dabei mit 300.000.000 m/s ansetzen. Nutzen Sie in Ihrem Skript die Exponentialschreibweise.

6. Was geben die folgenden Zeilen aus?

```
$wert = 2000.01;
print $wert + "1.01e3", "\n";
printf "%f \n", $wert + "1.01e3";
printf "%d \n", $wert + "1.01e3";
```

7. Und noch einmal: Was geben die folgenden Zeilen aus?

```
$meineVar = "1001 Nacht";
printf "%s \n", $meineVar;
printf "%d \n", $meineVar;
```

8. Kann auf der linken Seite einer Zuweisung ein Ausdruck stehen?

Kapitel 3

Programmieren mit Zahlen und Strings

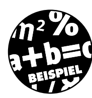
Ab hier geht´s richtig los! Im letzten Kapitel ging es hauptsächlich darum, wie Daten in Skripten repräsentiert werden. In diesem Kapitel lernen Sie die Operatoren und Funktionen kennen, mit denen Sie die Daten nun endlich auch vernünftig verarbeiten können.

Folgende Operatoren und Funktionen lernen Sie kennen:

- ✘ Den Zuweisungsoperator
- ✘ Die arithmetischen Operatoren
- ✘ Die mathematischen Funktionen
- ✘ Inkrement und Dekrement
- ✘ Die Definition von HERE-Texten
- ✘ Die Konkatenation und Wiederholung von Strings
- ✘ Die String-Funktionen

Im Vertiefungsteil erfahren Sie etwas mehr über:

- ✘ Die Auswertung von Ausdrücken mit mehreren Operatoren
- ✘ Bestimmte Fallstricke, die beim Aufsetzen von komplexen Ausdrücken zu beachten sind



Folgende Elemente lernen Sie kennen

Die meisten Operatoren, die mathematischen Funktionen und die String-Funktionen.

3.1 Der Zuweisungsoperator

Den Zuweisungsoperator = kennen Sie bereits aus dem vorangehenden Kapitel, denn ohne ihn geht kaum etwas.

Mit seiner Hilfe kann man Variablen

- konstante Werte
- Werte anderer Variablen
- Werte von Ausdrücken
- Ergebnisse von Funktionsaufrufen

zuweisen.

```
$var1 = 11;
$var2 = $var1;
$var2 = (14 * $var1) - 12;
$var2 = cos(0);
```

3.2 Arithmetischen Operatoren und mathematische Funktionen

Zahlen werden in Programmen in vielfältiger Weise missbraucht: als Schalter (0 und 1 gleich An und Aus), als Zählwerk, zur Codierung irgendwelcher anderen Daten. Das soll aber nicht darüber hinwegtäuschen, dass man mit Zahlen auch rechnen kann.

Zu diesem Zweck stellt uns Perl die arithmetischen Operatoren und eine Reihe von vordefinierten mathematischen Funktionen zur Verfügung.

Neben den hier vorgestellten Operatoren gibt es noch eine Reihe weiterer Operatoren, mit denen man Zahlen vergleichen kann. Diese werden in Kapitel 4.2 vorgestellt.

3.2.1 Die arithmetischen Operatoren

Die ersten vier arithmetischen Operatoren braucht man eigentlich niemandem zu erklären, da sie schlichtweg die mathematischen Grundrechenarten unterstützen.

Tabelle 3.1: Die Grundrechenarten

Operator	Bedeutung	Verwendung	
op1 + op2	Addition	`$var = 10 + 4;`	# $var = 14
op1 - op2	Subtraktion	`$var = 10 - 4;`	# $var = 6
op1 * op2	Multiplikation	`$var = 10 * 4;`	# $var = 40
op1 / op2	Division	`$var = 10 / 4;`	# $var = 2.5

Alle diese Operatoren bezeichnet man als binär, da sie immer zwei Operanden erfordern. Daneben gibt es noch zwei unäre Operatoren:

Tabelle 3.2: Die Vorzeichen

Operator	Bedeutung	Verwendung	
+op1	positives Vorzeichen	`$var = 2;`	# $var = 2
-op1	negatives Vorzeichen	`$var = -2;`	# $var = -2

Neben den Grundrechenarten gibt es noch weitere Rechenoperationen, für die in Perl eigene Operatoren definiert sind.

Tabelle 3.3: Exponent und Modulo

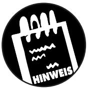
Operator	Bedeutung	Verwendung	
op1 ** op2	Exponent	`$var = 10 ** 2;`	# $var = 100
		Berechnet op1 hoch op2.	
op1 % op2	Modulo	`$var = 10 % 3;`	# $var = 1
		Teilt op1 durch op2 und gibt den dabei verbleibenden Rest als Ergebnis zurück.	
		op1 und op2 müssen Ganzzahlen sein!	

Mit Hilfe dieser Operatoren können wir endlich richtig nützliche Programme schreiben. Nehmen wir beispielsweise an, Sie haben 5000 DM zur Verfügung und überlegen, wie Sie diese am sinnvollsten anlegen. Mit Hilfe der Formel:

$$\text{Endkapital} = \text{Startkapital} * (1 + \text{Zinsen}/100)^{\text{Laufzeit}}$$

und einem kleinen Perl-Skript können Sie ausrechnen, wie sich Ihr Kapital auf einem Sparbuch in Abhängigkeit von Laufzeit und ausgehandelten Zin-

sen vermehrt (vorausgesetzt die Zinserträge werden dem Sparbuch gutgeschrieben).



Im nachfolgenden Skript wird der Eingabeoperator einfach als `<>` aufgerufen, was in diesem Fall eine verkürzte Schreibweise für `<stdin>` darstellt (siehe Kapitel 9).

Listing 3.1: zinsen1.pl

```
#!/usr/bin/perl

print "\n Programm zur Zinsberechnung\n\n";

print " Wie hoch ist das Startkapital:          ";
chomp($kapital = <>);

print " Wie hoch sind die Zinsen (in Prozent):  ";
chomp($zsatz = <>);

print " Wie viele Jahre beträgt die Laufzeit:   ";
chomp($laufzeit = <>);


$endkapital = $kapital * ( 1 + $zsatz/100.0) ** $laufzeit;
printf("\n\n Endkapital: %.2f DM\n", $endkapital);
```

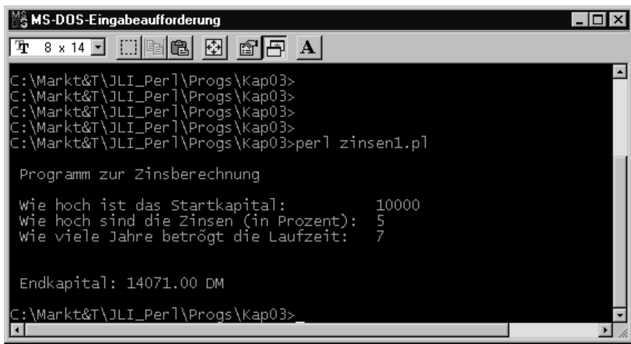


Abb. 3.1: Ausführung des Skripts zinsen1.pl

3.2.2 Die mathematischen Funktionen

Bestimmte häufig benötigte Rechenoperationen, wie zum Beispiel das Ziehen einer Wurzel oder das Berechnen des Sinus eines Winkels, sind mit Hilfe der Operationen für die Grundrechenarten nur sehr schwer zu realisieren. Für die wichtigsten dieser Rechenoperationen stellt Perl daher passende Funktionen zur Verfügung.

Tabelle 3.4: Eingebaute mathematische Funktionen

Funktion	Beschreibung
`abs(x)`	Absoluter Betrag
`atan2(x,y)`	Arcustangens von x/y
`cos(x)`	Kosinus (x in Bogenmaß)
`exp(x)`	Exponentialfunktion (e^x)
`hex(x)`	Umwandlung in Hexadezimalzahl
`int(x)`	Umwandlung in Ganzzahl (Integer)
`log(x)`	Natürlicher Logarithmus (ln x)
`oct(x)`	Umwandlung in Oktalzahl
`sin(x)`	Sinus (x in Bogenmaß)
`sqrt(x)`	Wurzel (x positiv)

Die Programmierung mit diesen Funktionen ist die Einfachheit selbst: Sie übergeben der Funktion die benötigten Parameter und erhalten das Ergebnis zurück. Um beispielsweise die Wurzel von 102355 zu berechnen, schreiben Sie:

```
$var = sqrt(102355);
```

Trigonometrische Funktionen

Wenn Sie mit den trigonometrischen Funktionen arbeiten, müssen Sie allerdings beachten, dass diese Funktionen als Parameter stets Werte in Bogenmaß (Radiant) erwarten. (Beim Bogenmaß wird der Winkel nicht in Grad, sondern als Länge des Bogens angegeben, den der Winkel aus dem Einheitskreis (Gesamtumfang 2) ausschneidet: `1 rad = 360°/2π; 1° = 2π/360 rad`.)

Nehmen wir an, Sie wollen die Höhe eines Bungalows ermitteln. Sie stellen sich in einiger Entfernung vor das Haus und markieren diesen Punkt am Boden. Von diesem Punkt aus peilen Sie die Dachkante an und messen den Winkel zwischen der Peillinie zur Dachkante und dem Boden, sagen wir 25 Grad. Jetzt müssen Sie noch die Entfernung zum Haus messen, sagen wir 18 Meter. Nach der Formel

`h = distanz * tan α`[1]

könnten wir jetzt die Höhe berechnen, wenn uns nicht die Funktion `tan` fehlen würde.

1 Zur Erinnerung: Für rechtwinklige Dreiecke ist der Tangens gleich dem Verhältnis aus Gegenkathete zu Ankathete.

Weitere mathematische Funktionen

Wem die Auswahl der eingebauten mathematischen Funktionen nicht ausreicht, der kann auf das CPAN-Modul `Math::Trig` zurückgreifen. Dort finden Sie neben Funktionen wie `asin`, `cosh` etc. auch die vordefinierte Konstante `pi` sowie eine Funktion zur Umrechnung von Winkelgrad in Bogenmaß: `deg2rad`.

Jetzt können wir das Programm zur Berechnung der Häuserhöhen aufsetzen:

Listing 3.2: haushoehe.pl

```
#!/usr/bin/perl

use Math::Trig;          # wegen tan und deg2rad


print "\n Programm zur Berechnung der Höhe eines Hauses\n\n";

print " Wie gross ist der gemessene Winkel:     ";
chomp($winkel = <>);
print " Wie gross ist die Entfernung zum Haus:  ";
chomp($distanz = <>);


$hoehe = $distanz * tan(deg2rad($winkel));

printf("\n\n Hoehe des Hauses: %.2f Meter\n", $hoehe);
```

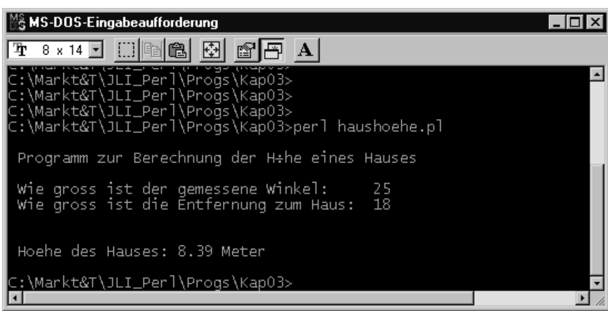


Abb. 3.2: Aufruf des Skripts haushoehe.pl

3.2.3 Die Gleitkommaproblematik

Bezüglich der Verarbeitung von Zahlen gibt es in Perl drei Eigentümlichkeiten, die man kennen sollte.

Perl rechnet grundsätzlich mit Gleitkommawerten

Mit Ausnahme des Modulo-Operators % arbeiten alle arithmetischen Operatoren und Funktionen mit Gleitkommazahlen. Wenn Sie also in einem Programm zwei Ganzzahlen addieren:

```
$var = 3 + 4;
```

wandelt der Perl-Interpreter die Zahlen in Gleitkommazahlen um und berechnet:

```
$var = 3.0 + 4.0;
```

Integer-Division

Wenn Sie mit dem /-Operator zwei Zahlen dividieren, die sich nicht ohne Rest teilen lassen, liefert Ihnen der Operator das Ergebnis als Gleitkommazahl zurück:

```
$var = 10 / 4;   # ergibt $var = 2.5
```

Dies ist zwar korrekt, nicht aber immer erwünscht. Manchmal ist man nur an dem ganzzahligen Ergebnis interessiert, sprich der Frage, wie oft der Divisor in den Dividend passt. In der Programmierung bezeichnet man dies als Integer-Division. Um unter Perl eine Integer-Division durchzuführen, gibt es zwei Möglichkeiten:

Sie können eine normale Gleitkommadivision durchführen und das Ergebnis in eine Ganzzahl (Integer) umwandeln:

```
$var = int 10 / 4;    # ergibt  $var = 2
```

Oder Sie verwenden die `use integer`-Anweisung[2]:

```
use integer;
$var = 10 / 4;        # ergibt  $var = 2
no integer;
$var = 10 / 4;        # ergibt  $var = 2.5
```

Genauigkeit von Gleitkommazahlen

Gleitkommazahlen werden – wie Sie aus dem vorangehenden Kapitel bereits wissen – intern als Kombination aus binärem Vorzeichen, Mantisse und Exponent dargestellt. Die Zahl 2001 wird beispielsweise als +0,2001 * 10^4 dargestellt (allerdings binärcodiert und zur Basis 2).

2 `use`-Anweisungen steuern die Arbeit des Interpreters. Man bezeichnet sie auch als Pragmas.

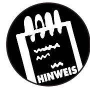
Dies hat Vor- und Nachteile.

Einerseits können Sie auf diese Weise Zahlen darstellen, die vom Betrag her sehr groß oder sehr klein sind:

```
6.022045e23      # Anzahl Teilchen in einem Mol
1.6021892e-19;   # Ladung eines Elektrons
```

Auf den meisten Plattformen sind für den Exponenten Werte von bis zu +/-300 möglich!

Andererseits ist die Genauigkeit der Zahlen begrenzt. Auf den meisten Rechnerarchitekturen stehen zur Abspeicherung der Mantisse 53 Bit zur Verfügung. Dies reicht im Durchschnitt zur Codierung von 16 dezimalen Ziffern. Enthält eine Zahl mehr signifikante Ziffern, gehen die letzten Ziffern verloren.



Die signifikanten Ziffern ermittelt man, indem man die Nullen am Anfang und Ende einer Zahl wegstreicht. Die Zahl 3045000 enthält demnach vier, die Zahl 300,450010 acht signifikante Ziffern.

Eine weitere Eigentümlichkeit entsteht durch die Umwandlung der dezimalen Zahlen in Binärzahlen. Bestimmte Gleitkommazahlen werden bei der Umwandlung ins Binärsystem nämlich zu reellen Zahlen mit unendlicher Bitfolge. Beispielsweise ist 0.9 binär gleich 0.1110011001100...

Bei der Darstellung dieser Zahlen im Computer muss die unendliche Bitfolge natürlich irgendwann abgeschnitten werden (sonst würde man ja den gesamten Arbeitsspeicher mit einer einzigen Zahl voll schreiben). Die Folge ist, dass der Computer nur mit einer Näherung der eigentlichen Zahl rechnet. Meist merkt man als Programmierer nichts davon. Es kann aber vorkommen, dass – insbesondere bei Zahlenvergleichen – unerwartete Ergebnisse auftreten.

3.3 Inkrement, Dekrement und kombinierte Zuweisung

Programmierer tippen nicht gerne. Folglich gibt es für eine Reihe von häufig benötigten Rechenoperationen eigene Operatoren.

Kombinierte Zuweisungen

Mit Hilfe des Zuweisungsoperators kann man einer Variablen einen beliebigen Wert zuweisen:

```
$var1 = 3.4;
$var1 = 2 * $var2;
```

In vielen Fällen baut der neue Wert dabei auf dem aktuellen Wert der Variablen auf, beispielsweise wenn Sie den Wert der Variablen um einen bestimmten Betrag erhöhen oder vermindern wollen oder den Wert mit einem Faktor multiplizieren oder dividieren möchten. Zu diesem Zweck müssen Sie den Variablenwert in den Ausdruck auf der rechten Seite einbauen:

```
$var = 2;
$var = $var + 3;   # $var gleich 5
```

Mit Hilfe des +=-Operators kann man dies vereinfacht schreiben als

```
$var += 3;
```

oder

```
$var = $var + 3*pi*$var;
```

als

```
$var += 3*pi*$var;
```

Tabelle 3.5: Die kombinierten Zuweisungsoperatoren

Operator	**Bedeutung**	**Entspricht**
op1 += op2	Addition	op1 = op1 + op2
op1 -= op2	Subtraktion	op1 = op1 - op2
op1 *= op2	Multiplikation	op1 = op1 * op2
op1 /= op2	Division	op1 = op1 / op2
op1 %= op2	Modulo	op1 = op1 % op2
op1 **= op2	Exponent	op1 = op1 ** op2

Inkrement und Dekrement

Skalare Variablen werden häufig als Zähler eingesetzt (zum Zählen der Vorkommen eines bestimmten Wortes in einer Textpassage, zum Kontrollieren von Schleifen). Dabei wird der Wert der Zählervariablen wiederholt um 1 erhöht oder vermindert – je nachdem, ob der Zähler rauf oder runter zählt. Die Erhöhung eines Zählers bezeichnet man als Inkrement, seine Verminderung als Dekrement.

Das Inkrementieren des Zählers lässt sich wie folgt implementieren:

```
$var = 1;
$var = $var + 1;
```

Einfacher geht es mit Hilfe des kombinierten Zuweisungsoperators +=:

```
$var += 1;
```

Noch einfacher geht es mit Hilfe des Inkrementoperators:

```
++$var;
```

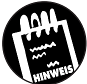
Tabelle 3.6: Inkrement und Dekrement

Operator	Bedeutung	Entspricht
++op1	Inkrement	op1 = op1 + 1
--op1	Dekrement	op1 = op1 - 1

Präfix- und Postfixnotation

Inkrement- und Dekrementoperatoren kann man dem Operanden voranstellen (Präfixnotation: ++op) oder nachstellen (Postfixnotation: op++).

Wenn man die Operatoren alleine in einer Anweisung benutzt (`++$var`, oder `$var++;`), spielt es keine große Rolle, welche Version man verwendet.



Geschwindigkeitsfanatiker wird es interessieren, dass die Präfixnotation schneller ist, da der Perl-Interpreter für ihre Implementierung weniger Maschinenbefehle benötigt.

Wenn Sie die Operatoren aber in einem Ausdruck verwenden, stellt sich die Frage, welcher Wert des Operanden für die Berechnung des Ausdrucks verwendet wird. Schauen Sie sich dazu die folgende Anweisung an:

```
$var1 = 12;
$var2 = 4 * ++$var1;
```

Welchen Wert hat `$var2` nach Ausführung dieses Codes? Um diese Frage zu beantworten, muss man wissen, welchen Wert `++$var1` in dem Ausdruck repräsentiert: den Wert, den `$var1` vor der Inkrementierung hat, oder den Wert, den die Variable nach der Inkrementierung hat?

- Die Präfixnotation erhöht den Wert der Variablen und gibt danach den neuen Wert zurück.
- Die Postfixnotation erhöht den Wert der Variablen, liefert aber den alten Wert zurück.

```
$var1 = 12;
$var2 = 4 * ++$var1;    # $var2 gleich 4*13 = 52
$var1 = 12;
$var2 = 4 * $var1++;    # $var2 gleich 4*12 = 48
```

3.4 String-Operatoren und String-Funktionen

Wie für die Zahlen so gibt es auch für Strings spezielle Operationen und Funktionen, mit denen man Strings bearbeiten kann.

3.4.1 Strings definieren

Beginnen wir mit einer kurzen Rekapitulierung dessen, was Sie bereits aus Kapitel 2.2 kennen.

- String-Konstanten werden in einfache oder doppelte Anführungszeichen eingeschlossen.
- Innerhalb von doppelten Anführungszeichen werden Escape-Zeichen und Variablennamen expandiert (durch die entsprechenden Werte ersetzt).
- In skalaren Variablen können Strings zwischengespeichert werden.

```
$str1 = 'Hallo Welt!';
$str2 = "Der Wert von Var ist: $Var \n";
```

Darüber hinaus gibt es noch weitere Möglichkeiten, Strings zu definieren.

Quoting mit q

In Kapitel 2.2 sind wir bereits auf die Problematik eingegangen, dass man das Zeichen, mit dem man den String umschließt (einfache oder doppelte Anführungszeichen), nur durch Voranstellung des Escape-Operators in den String selbst einbauen kann.

```
$hyperlink = "<A HREF= \"...\">";
```

Die Voranstellung des Escape-Operators ist allerdings recht unbequem. Einfacher wäre es, wenn man je nach Bedarf verschiedene Zeichen zur Kennzeichnung des Stringanfangs und -endes definieren könnte. Tatsächlich ist dies in Perl möglich.

Mit q und nachfolgendem Symbol können Sie einen Ersatz für das einfache Anführungszeichen definieren:

```
$str = q%'quote' ist das engl. Wort für 'zitat'%;
```

Mit qq und nachfolgendem Symbol können Sie einen Ersatz für das doppelte Anführungszeichen definieren:

```
$str = qq§Der Wert von Var ist: $Var \n§;
```

HERE-Texte

Schließlich gibt es die Möglichkeit, den String statt zwischen Symbolen zwischen zwei identischen, frei wählbaren Bezeichnern einzuschließen.

Listing 3.3: douglas.pl

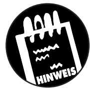
```
#!/usr/bin/perl -w

$gedicht = <<HERE_DOUGLAS;


            Archibald Douglas

                 von
            Theodor Fontane

  Ich habe es getragen sieben Jahr,
  Und ich kann es nicht tragen mehr!
  Wo immer die Welt am schönsten war,
  Da war sie öd und leer.

HERE_DOUGLAS

print $gedicht;
```

- ✘ Eingeleitet wird ein HERE-Text mit dem <<-Operator gefolgt von dem HERE-Bezeichner, den Sie selbst wählen. Zwischen dem Operator und dem Bezeichner darf kein Leerzeichen stehen. Wenn Sie den HERE-Bezeichner in Großbuchstaben schreiben und mit dem Präfix HERE beginnen (was beides nicht erforderlich ist), sind Anfang und Ende des HERE-Textes gut zu erkennen.
- ✘ Der eigentliche Text beginnt in der nachfolgenden Zeile und wird so ausgegeben, wie er im Quelltext steht. Variablennamen werden expandiert (durch ihren Wert ersetzt).
- ✘ Das Ende des HERE-Textes wird durch die Wiederholung des HERE-Bezeichners angezeigt. Der HERE-Bezeichner muss dabei allein am Anfang der Zeile stehen.

Wenn Sie nicht möchten, dass Variablennamen im HERE-Text expandiert werden, setzen Sie den ersten HERE-Bezeichner in einfache Anführungszeichen.

3.4.2 String-Operatoren

Für Strings gibt es nur wenige Operatoren (dafür aber umso mehr Funktionen.)



Neben den hier vorgestellten Operatoren gibt es noch Operatoren zum Vergleichen von Strings. Diese werden in Kapitel 4.2 vorgestellt.

Konkatenation und Wiederholung

Die wichtigste Operation zur Verarbeitung von Strings ist die Aneinanderreihung von verschiedenen Strings (fachmännisch auch als Konkatenation bezeichnet).

```
$str = "Hallo " . "Welt!";
```

Mit dem .-Operator können Sie aber nicht nur Strings, sondern auch Zahlen aneinander reihen.

```
$str = "Der Wert von var ist " . $var . "\n";
```

Dabei wird der Wert der Zahl in einen String umgewandelt (so wird beispielsweise 333 zu '333') und mit . konkateniert.

In anderen Sprachen lautet der Konkatenierungsoperator + (beispielsweise in C++, wo der +-Operator zur Konkatenierung von Objekten der Klasse `string` entsprechend überladen wurde). In Perl ist der +-Operator allerdings rein für Zahlen vorgesehen. Wenn Sie versuchen, Strings mit + aneinander zu hängen, wandelt der Perl-Interpreter die beteiligen Strings in Zahlen um (in 0, wenn der String nicht mit einer Ziffer beginnt).

Um einen Text oder eine Zeichenfolge mehrfach aneinander zu reihen, verwendet man den x-Operator:

```
$trennlinie = '*' x 20;  # erzeugt ********************
```

Inkrement

Der Vollständigkeit halber sei noch erwähnt, dass man auch den Inkrementoperator ++ (nicht aber den Dekrementierungsoperator --) auf Strings anwenden kann. Der Inkrementierungsoperator wird auf Buchstabenfolgen so angewendet, als handele es sich um Zahlen zur Basis a–z bzw. A–Z.

```
$str = 'aaa';
++$str;              # $str ist jetzt gleich 'aab'
$str = 'aaz';
++$str;              # $str ist jetzt gleich 'aba'
```

Die Inkrementierung funktioniert nur für den ++-Operator (nicht für + 1) und auch nur dann, wenn die String-Variable nicht zuvor schon irgendwo als Zahl ausgewertet wurde.

3.4.3 String-Funktionen

Eine Reihe von vordefinierten Funktionen unterstützen Sie bei der Programmierung mit Strings.

print, printf, sprintf

Die Funktionen `print` und `printf` habe ich Ihnen bereits in Kapitel 2.5 vorgestellt. Die Funktion `sprintf` arbeitet im Wesentlichen genauso wie die Funktion `printf`. Im Unterschied zu `printf` gibt `sprintf` den formatierten String jedoch nicht aus, sondern liefert ihn als Ergebnis zurück, so dass man ihn einer skalaren Variablen zuordnen kann. Auf diese Weise kann man beispielsweise reelle Zahlen mit vielen Nachkommastellen auf eine bestimmte Zahl von Nachkommastellen kürzen.

```
$var = sin 0.5;                  # $var = 0.479425538604203
$var = sprintf("%.2f", $var);    # $var = "0.48"
```

length

Liefert die Länge eines Strings, d.h. die Anzahl der Zeichen im String zurück (ohne abschließendes Terminierungszeichen).

```
$str = 'Hallo';
print length $str;      # 5
```

Die Funktion `length` liefert ab Perl 5.6 tatsächlich die Anzahl Zeichen im String und nicht – wie in älteren Perl-Versionen – die Anzahl Byte, die der String belegt. Dies ist wichtig, wenn man Text verarbeitet, der in UNICODE codiert ist, da in UNICODE jedes Zeichen durch 2 Byte codiert wird.

lc, uc und lcfirst, ucfirst

Mit Hilfe dieser Funktionen können Sie einen ganzen String oder nur das erste Zeichen im String in Klein- oder Großbuchstaben umwandeln. Sie entsprechen damit den Escape-Sequenzen `\L`, `\U`, `\l`, `\u` aus Tabelle 3.1.

```
$passwort = "krxplw";
$passwort = uc $passwort;      # "KRXPLW"
```

reverse

Diese Funktion dreht die Zeichenfolge in einem String um.

```
$str = reverse "Hallo";      # str = "ollaH"
```

Wozu ist die Funktion `reverse` gut? Nun, beispielsweise um Palindrome aufzuspüren. Palindrome sind Wörter, oder allgemeiner ausgedrückt: Zeichenfolgen, die man vorwärts wie rückwärts lesen kann: beispielsweise neger – regen. Solche Palindrome kann ein simples Programm allerdings nicht erkennen. Im strengeren Sinne ist ein Palindrom eine Zeichenfolge, die vorwärts wie rückwärts gelesen identisch ist: otto – otto. Solche Zeichenfolgen kann man leicht mit einem Perl-Skript identifizieren. Wir müssen dazu allerdings dem Stoff aus dem nachfolgenden Kapitel ein wenig vorgreifen und eine `if-else`-Konstruktion und den Vergleichsoperator `eq` einbauen:

Listing 3.4: palindrom.pl

```
#!/usr/bin/perl -w

$eingabe = 0;

print "Geben Sie eine Zeichenfolge ein: ";
chomp ($eingabe = <STDIN>);

if ($eingabe eq reverse $eingabe) {
  print $eingabe, " ist ein Palindrom\n";
  }
else {
  print $eingabe, " ist kein Palindrom\n";
  }
```

Wenn Sie `reverse` zusammen mit einer `print`-Anweisung verwenden, müssen Sie vor `reverse` das Schlüsselwort `scalar` setzen. Mehr dazu in Kapitel 8:

```
print scalar reverse "Hallo";
```

chr und ord

Zeichen werden intern durch Zahlen codiert. Mit Hilfe der Funktionen `chr` und `ord` können Sie zu einer Zahl das zugehörige Zeichen und zu einem Zeichen den zugehörigen Zahlencode ermitteln.

```
$zeichen = chr(65);          # $zeichen = A
$code_nl = ord("\n");        # $code_nl = 10
```

Ab Perl-Version 5.6 unterstützen die Funktionen neben ASCII auch UNICODE.

index und rindex

Mit der Funktion `index` können Sie feststellen, ob in einem String A ein zweiter String B enthalten ist. Die Funktion durchsucht den String vom Anfang an und liefert als Ergebnis die Position des ersten Vorkommens des gesuchten Teilstrings in String A zurück oder -1, wenn kein entsprechender Teilstring gefunden wurde.

```
#!/usr/bin/perl -w

$str = <<HERE_TEXT;
Es ist unleugbar, dass Krieg der natürliche
Zustand der Menschen war, bevor die Gesell-
schaft gebildet wurde, und zwar nicht einfach
der Krieg, sondern der Krieg aller gegen alle.

Thomas Hobbes
HERE_TEXT

print index ($str, "Krieg");         # liefert 23
```

Wenn Sie der Funktion als drittes Argument eine Position übergeben, sucht die Funktion ab dieser Position.

```
print index ($str, "Krieg", 24);     # liefert 138
```

Die Funktion `rindex` durchsucht den String von hinten nach vorne.

substr

Mit Hilfe dieser Funktion können Sie in einem String nach Teilstrings suchen und diese zurückliefern lassen oder ersetzen. Gleiches kann man allerdings auch mit regulären Ausdrücken erreichen, die kürzeren Code produzieren und wesentlich mächtiger sind (siehe Kapitel 10). Wer sich jedoch gar nicht mit den regulären Ausdrücken anfreunden kann, der kann in der PERLFUNC-Dokumentation nachschlagen, wie man `substr` aufruft.

chop und chomp

Die Funktion chop entfernt aus einem String das letzte Zeichen. Die Funktion chomp entfernt das letzte Zeichen dagegen nur dann, wenn es mit dem Wert der internen Variablen $/ übereinstimmt. Standardmäßig steht in dieser Variablen das Zeichen bzw. die Zeichenkombination für den Zeilenumbruch.

chomp ist in Perl nahezu untrennbar mit dem Einlesen von Eingaben über die Konsole verbunden. Da Eingaben über die Konsole stets durch Drücken der ↵-Taste abgeschickt werden, enden diese Eingaben immer mit einem Zeilenumbruch. Mit chomp lässt es sich mühelos entfernen:

```
$eingabe = 0;

print "Geben Sie einen Text ein: ";
chomp ($eingabe = <STDIN>);
```

Wenn Sie der Zeilenumbruch nicht stört (aus Zahlenwerten wird er bei der Weiterverarbeitung im numerischen Kontext automatisch entfernt) oder der Umbruch gar erwünscht ist, können Sie natürlich auch auf chomp verzichten.

pack und unpack

Die Funktionen pack und unpack dienen dazu, Daten in ein bestimmtes String-Format zu packen oder wieder zu entpacken. In den Händen fortgeschrittener Programmierer können pack und unpack mächtige Tools sein. Da wir noch nicht ganz zu den fortgeschrittenen Programmierern zählen, muss ich Sie allerdings mit der Vorstellung dieser Funktionen noch bis Kapitel 9.2.3 vertrösten.

3.4.4 Reguläre Ausdrücke

Die letzte – und mächtigste – Option zur Bearbeitung von Texten ist die Verwendung regulärer Ausdrücke, mit deren Hilfe man Texte analysieren, durchsuchen und passagenweise ersetzen kann.

Den regulären Ausdrücken ist im Fortgeschrittenen-Teil des Buches ein eigenes Kapitel gewidmet.

3.5 Vertiefung: Auswertung von Ausdrücken

Ein Ausdruck ist eine Kombination von Operatoren und Werten (in Form von Konstanten, Variablen und/oder Rückgabewerten von Funktionen), die vom Perl-Interpreter zu einem Ergebniswert ausgewertet werden. Entscheidend ist dabei, dass Programmierer und Interpreter bezüglich der korrekten Auswertung eines Ausdrucks gleicher Meinung sind. Aus diesem Grund gibt es für die Auswertung von Ausdrücken bestimmte Regeln.

3.5.1 Operatorenrangfolge

Gibt es in einem Ausdruck aber mehrere Operatoren, stellt sich die Frage, in welcher Reihenfolge die Operatoren ausgewertet werden.

```
$var1 = 12;
$var2 = $var1 * 3 + 4;        # 40 oder 84 ?
```

Hier gibt es eine festgelegte Rangfolge der Operatoren. Tabelle 3.7 listet die wichtigsten Operatoren gemäß ihrer Rangordnung auf. Operatoren, die in der Tabelle weiter oben stehen, haben eine höhere Präzedenz und binden ihre Operanden stärker als die weiter unten folgenden Operatoren. Eine vollständige Liste finden Sie in der PERLOP-Dokumentation.

Tabelle 3.7: Rangfolge und Assoziativität der wichtigsten Operatoren

<table>
<tr><th>Operator</th><th>Assoz.</th><th>Bedeutung</th></tr>
<tr><td>++ --</td><td>–</td><td>Inkrement und Dekrement</td></tr>
<tr><td>**</td><td>R-L</td><td>Exponent</td></tr>
<tr><td>! \ + -</td><td>R-L</td><td>Logisches NOT (siehe Kapitel 4.2), Referenz (siehe Kapitel 7), positives und negatives Vorzeichen</td></tr>
<tr><td>=~ !+</td><td>L-R</td><td>Pattern Matching (siehe Kapitel 10)</td></tr>
<tr><td>* / % x</td><td>L-R</td><td>Multiplikation, Division, Modulo, String-Wiederholung</td></tr>
<tr><td>+ - .</td><td>L-R</td><td>Addition, Subtraktion, Konkatenation</td></tr>
<tr><td>< > <= >= lt gt le ge</td><td>–</td><td>Vergleiche (siehe Kapitel 4.2)</td></tr>
<tr><td>== != <=> eq ne cmp</td><td>–</td><td>Weitere Vergleiche (siehe Kapitel 4.2)</td></tr>
<tr><td>&&</td><td>L-R</td><td>Logisches UND (siehe Kapitel 4.2)</td></tr>
<tr><td>||</td><td>L-R</td><td>Logisches ODER (siehe Kapitel 4.2)</td></tr>
<tr><td>..</td><td>–</td><td>Bereichsoperator (siehe Kapitel 5)</td></tr>
<tr><td>not</td><td>R-L</td><td>Logisches NOT (siehe Kapitel 4.2)</td></tr>
</table>

Operator	Assoz.	Bedeutung
and	L-R	Logisches UND (siehe Kapitel 4.2)
or xor	L-R	Logisches ODER und EXKLUSIVES ODER (siehe Kapitel 4.2)

Tabelle 3.7: Rangfolge und Assoziativität der wichtigsten Operatoren (Fortsetzung)

Mit Hilfe dieser Tabelle kann man die Frage, zu welchem Wert der Ausdruck `$var1 * 3 + 4` ausgewertet wird, leicht beantworten. Der Multiplikationsoperator * hat einen höheren Rang als der +-Operator und bindet die Operanden stärker. Es wird also zuerst `$var1 * 3` berechnet und dann zu diesem Zwischenergebnis `4` addiert.

```
$var1 = 12;
$var2 = $var1 * 3 + 4;        # 40
```

Möchte man eine andere Auswertungsreihenfolge erzwingen, muss man Klammern setzen:

```
$var1 = 12;
$var2 = $var1 * (3 + 4);      # 84
```

3.5.2 Assoziativität

In welcher Reihenfolge Operatoren unterschiedlicher Rangordnung ausgewertet werden, wissen Sie nun. Wie aber sieht es aus, wenn in einem Ausdruck mehrere Operatoren gleicher Rangordnung verwendet werden.

Dann werden die Operatoren entweder von links nach rechts oder von rechts nach links ausgewertet – oder die Rangfolge ist nicht festgelegt. Die Assoziativität der wichtigsten Operatoren können Sie Tabelle 4.7 entnehmen. Unter Berücksichtigung der Assoziativität kann man leicht ausrechnen, dass die folgende Anweisung

```
$var = 2**2**3;
```

der Variablen `$var` den Wert 256 und nicht den Wert 81 zuweist, weil der Interpreter zuerst 2**3 berechnet (ergibt 8) und mit dem Ergebnis noch einmal 2 potenziert (2 ** 8).

Dieses Beispiel sollte auch deutlich machen, dass Ausdrücke nicht immer leicht zu lesen und zu interpretieren sind. Es lohnt sich daher, auch in Ausdrücken, die eigentlich keiner Klammern bedürfen, selbige zu setzen, um die Auswertungsreihenfolge zu verdeutlichen: `2** (2**3)`. Ist das nicht gleich viel besser?

3.5.3 Seiteneffekte

Von Seiteneffekten spricht man, wenn eine Anweisung untergeordnete Anweisungen enthält:

```
$var2 = 3 * $var1++;     # Zuweisung eines neuen Wertes an
                         # $var2 sowie an $var1
$var2 = funk(2);         # Zuweisung eines Wertes an $var2
                         # und Aufruf der Funktion funk
```

Problematisch wird es meist dann, wenn Anweisung und untergeordnete Anweisung auf den gleichen Variablen operieren.

```
$var2 = 3;
$var2 = 3 * $var2++;
```

Wie wird dieser Ausdruck ausgewertet? Zuerst wird der alte Wert von `$var2` mit 3 multipliziert. Dann wird `$var2` inkrementiert (in `$var2` steht jetzt der Wert 4), und schließlich wird das Ergebnis der Multiplikation an `$var2` zugewiesen. In `$var2` steht also nach Ausführung der Anweisung der Wert 9 und der Effekt der Inkrementierung ist ganz verloren gegangen. Man sieht: hier hat sich der Programmierer selbst ausgetrickst.

Schauen wir uns noch ein weiteres Beispiel an, bei dem wir ein wenig vorgreifen und eine selbst definierte Funktion verwenden:

```
$var1 = 0;
sub funk {
  $var1 = $_[0];
  $result = 1;
  }
```

Diese Funktion übernimmt, wenn sie aufgerufen wird, einen Parameter und weist dessen Wert (der in `$_[0]` steht) der globalen Variablen `$var` zu. Schließlich liefert sie den Wert 1 als Ergebnis zurück. Diese Funktion wird nun wie folgt aufgerufen:

```
$var1 = 3;
$var2 = funk(2) + $var1 * 3;
```

Was hier passiert, ist kaum noch zu überblicken. Eine Möglichkeit wäre, dass zuerst die Multiplikation (ergibt 9), dann der Funktionsaufruf (liefert 1) und schließlich die Addition ausgeführt wird. Als Ergebnis würde dann der Variablen `$var2` der Wert 1+9 zugewiesen. Aber ist dies korrekt?

Korrekt ist, dass der Perl-Interpreter vor der Addition die Multiplikation ausführen muss. Mit Klammern ausgedrückt würde der auszuwertende Ausdruck also wie folgt lauten:

```
$var2 = funk(2) + ($var1 * 3);
```

Jetzt stellt sich aber die Frage, ob zuerst der linke Operand (`funk(2)`) oder der rechte Operand (`$var1 * 3`) der Addition berechnet wird. Im ersten Fall ist das Endergebnis 10, im zweiten Fall 7, und in beiden Fällen kann die Multiplikation vor der Addition ausgeführt werden. Tatsächlich gibt es für die Auswertungsreihenfolge der Operanden eines Operators keine feste Regel. Meist aber werden die Operanden von links nach rechts ausgewertet. Der Perl-Interpreter weist der Variablen `$var2` daher den Wert 7 zu.

Was lehrt uns das?

Es ist nicht schwer, in Perl Code aufzusetzen, den kein Mensch mehr versteht.

3.6 Fragen und Übungen

1. Welches Ergebnis liefert der folgende Ausdruck?

   ```
   8/5
   ```

2. Wie müssten Sie den Ausdruck formulieren, damit das Ergebnis 1 lautet?
3. Was ist der Unterschied zwischen

   ```
   $var = 33;
   $str = "Der Wert von var ist " . $var . "\n";
   ```

   und

   ```
   $var = 33;
   $str = "Der Wert von var ist " + $var + "\n";
   ```

4. Nennen Sie drei Möglichkeiten, einer skalaren Variablen einen String zuzuweisen!

5. Was ist falsch an folgendem Code?

```
$gedicht = << HERE_DOUGLAS;


            Archibald Douglas

                 von
            Theodor Fontane

  Ich habe es getragen sieben Jahr,
  Und ich kann es nicht tragen mehr!
  Wo immer die Welt am schönsten war,
  Da war sie öd und leer.

HERE_DOUGLAS
```

6. Was ist falsch an folgendem Code?

```
$gedicht = <<HERE_DOUGLAS


            Archibald Douglas

                 von
            Theodor Fontane

  Ich habe es getragen sieben Jahr,
  Und ich kann es nicht tragen mehr!
  Wo immer die Welt am schönsten war,
  Da war sie öd und leer.

  HERE_DOUGLAS
```

7. Wodurch wird die Reihenfolge festgelegt, in der Ausdrücke mit mehreren Operatoren ausgewertet werden?

8. Welchen Wert haben die folgenden Ausdrücke?

```
$var1 = 12;
$var2 = $var1++ * (3 + 4);
$var1 = 12;
$var2 = ++$var1 * (3 + 4);
```

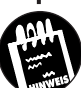
Kapitel 4

Steuerung des Programmflusses

Bis jetzt sahen unsere Perl-Skripten fast immer so aus, dass sie Anweisung für Anweisung von oben nach unten ausgeführt wurden. Ich sagte »fast«, denn manchmal (beispielsweise in Listing 2.3) haben wir auch ein wenig vorgegriffen und uns spezieller Konstrollstrukturen bedient, die es uns erlauben, auf die Programmausführung Einfluss zu nehmen.

Dabei geht es grundsätzlich um zwei typische Problemstellungen:

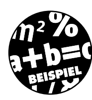
- ✘ Die bedingte Ausführung (eine oder mehrere Anweisungen sollen nur dann ausgeführt werden, wenn eine bestimmte Bedingung erfüllt ist)
- ✘ Die wiederholte Ausführung (eine oder mehrere Anweisungen sollen solange wiederholt ausgeführt werden, wie eine bestimmte Bedingung erfüllt ist)



Für beide Aufgaben gibt es in Perl spezielle Konstrukte, die wir nun kennen lernen werden.

Im Einzelnen lernen Sie,

- ✘ wie man Code in Abhängigkeit von einer Bedingung ausführen lässt
- ✘ wie man einfache und komplexe Bedingungen formuliert
- ✘ wie man Codeblöcke wiederholt ausführen lässt
- ✘ wie man Schleifeniterationen vorzeitig abbricht



In den Vertiefungsabschnitten erfahren Sie,

- ✘ wie man Verzweigungen und Schleifen noch kürzer formulieren kann
- ✘ welche Fallstricke beim Aufsetzen von Schleifen und Bedingungen zu beachten sind

Folgende Elemente lernen Sie kennen

Die Schlüsselwörter `if`, `else`, `while`, `do`, `for` und `foreach` plus den Abbruchbefehlen `redo`, `next` und `last`. Die Liste der Ihnen bereits bekannten Operatoren wird um die Vergleichsoperatoren für Zahlen und Strings sowie den Bedingungsoperator ergänzt. Schließlich stelle ich Ihnen noch die Standardvariable `$_` vor.

4.1 Die if-Bedingung

Die `if`-Verzweigung dient dazu, anhand einer Bedingung – beispielsweise des Wertes einer Variablen, des Rückgabewertes einer Funktion oder des Wahrheitswertes eines Booleschen Ausdrucks – zu entscheiden, ob der nachfolgende Anweisungsblock ausgeführt werden soll oder nicht.

```
if (Bedingung)
  {
  # Bedingung ist wahr
  Anweisungen;
  }
```

Schauen wir uns gleich einmal an, wie die Verwendung der `if`-Bedingung in der Praxis aussieht:

```
#!/usr/bin/perl -w

$eingabe = 0;

print "\n Geben Sie eine Zahl ein: ";
chomp ($eingabe = <STDIN>);

if ($eingabe >= 0)
  {
  printf (" Wurzel von %d: %.4f\n", $eingabe, sqrt $eingabe);
  }
print " Programm wird beendet\n";
```

Das obige Skript liest eine Zahl ein und berechnet die Wurzel dieser Zahl. Da man die Wurzel allerdings nur für positive Zahlen berechnen kann, stellt das Programm mit Hilfe einer `if`-Bedingung sicher, dass auch wirklich eine positive Zahl eingegeben wurde. Gibt der Anwender eine negative Zahl ein, wird die von der `if`-Bedingung kontrollierte Anweisung nicht ausgeführt – was gut ist, da sonst der Aufruf von `sqrt` mit einer negativen Zahl zum Programmabbruch führt und die nachfolgenden Anweisungen (im Beispiel die abschließende `print`-Anweisung) nicht mehr zur Ausführung kommen.

Anweisungen und Anweisungsblöcke

Wie Sie obigem Beispiel entnehmen können, werden die Anweisungen, die von der `if`-Bedingung kontrolliert werden, in geschweifte Klammern gefasst. Eine solche Folge von Anweisungen in geschweiften Klammern bezeichnet man als Anweisungsblock.

Alle Kontrollstrukturen arbeiten mit Anweisungsblöcken. Der typische Aufbau ist:

```
Schlüsselwort  Bedingung
    Anweisungsblock
```

if-else-Konstruktionen

Die einfache `if`-Bedingung entspricht der Anweisung: »Wenn die folgende Bedingung erfüllt ist, dann tue dies. Danach fahre normal mit der Ausführung des Programms fort.« Diese Anweisung kann man erweitern, indem man einen `else`-Teil anhängt. Die Aussage der Anweisung lautet dann: »Wenn die folgende Bedingung erfüllt ist (`if`), dann tue dies, ansonsten (`else`) tue das. Danach fahre normal mit der Ausführung des Programms fort.«

```
if (Bedingung)
  {
  //Bedingung ist wahr
  Anweisungen;
  }
 else
  {
  //Bedingung ist falsch
  Anweisungen;
  }
```

Mit Hilfe einer if-else-Konstruktion können wir das obige Beispielprogramm dahingehend erweitern, dass im Falle der Eingabe einer negativen Zahl der Anwender auf seinen Fehler hingewiesen wird.

Listing 4.1: wurzel.pl

```
#!/usr/bin/perl -w

$eingabe = 0;

print "\n Geben Sie eine Zahl ein: ";
chomp ($eingabe = <STDIN>);

if ($eingabe >= 0)
  {
  printf (" Wurzel von %d: %.4f\n", $eingabe, sqrt $eingabe);
  }
  else
  {
  print " Wurzel von negativen Zahlen " .
         "kann nicht berechnet werde\n";
  }
```

Mit Hilfe des Schlüsselwortes `elseif` kann man `if`-Bedingungen verschachteln:

```
if (Bed1) {Anweisungen }
  elseif (Bed2) { Anweisungen }
  elseif (Bed3) { Anweisungen }
  ...
  else {Anweisungen }
```

4.2 Wie formuliert man Bedingungen?

Bevor wir uns weitere Kontrollstrukturen anschauen, wollen wir uns ein wenig ausführlicher mit der Konstruktion von Bedingungen beschäftigen.

Bedingungen sind Ausdrücke, sprich eine Kombination von Daten und Operatoren, die zu einem eindeutigen Ergebniswert ausgewertet werden können (vergleiche Kapitel 2.4 und 3). Somit kennen wir jetzt zwei Stellen, an denen Ausdrücke eingesetzt werden:

- ✘ Auf der rechten Seite von Zuweisungen
- ✘ In den Bedingungen von Kontrollstrukturen

In einer Beziehung unterscheiden sich die Ausdrücke von Bedingungen allerdings von anderen Ausdrücken: Sie repräsentieren keine beliebigen Werte, sondern die Wahrheitswerte »wahr« und »falsch«.

Die Bedingungen von Kontrollstrukturen stellen eine besondere Umgebung, einen sogenannten Kontext, dar. Innerhalb dieses Kontextes interpretiert der Perl-Interpreter Ausdrücke automatisch als Wahrhweitswerte. In Kapitel 8 werden wir uns ausführlicher mit den Kontexten von Perl beschäftigen.

4.2.1 Arithmetische Ausdrücke in Bedingungen

Wenn Sie in einer Bedingung einen arithmetischen Ausdruck verwenden, beispielsweise:

```
if ($var)
```

oder

```
if ($var * 2)
```

geschieht Folgendes: Der Perl-Interpreter wertet den Ausdruck aus und erhält einen Ergebniswert. Diesen Ergebniswert interpretiert er dann als Booleschen Wahrheitswert: Ist der Ergebniswert gleich 0, so wird der Ausdruck als falsch angesehen, ansonsten als wahr.

Als »falsch« werden grundsätzlich folgende Ausdrücke interpretiert:

- ✘ der Wert 0
- ✘ der String "0"
- ✘ ein leerer String
- ✘ eine undefinierte Variable

Alle anderen Ergebniswerte werden als »wahr« interpretiert.

Arithmetische Ausdrücke werden in Bedingungen allerdings eher selten eingesetzt. Meist formuliert man die Ausdrücke in den Bedingungen mit Hilfe spezieller Vergleichsoperatoren.

4.2.2 Die Vergleichsoperatoren

Mit Hilfe der Vergleichsoperatoren können Sie Bedingungen formulieren wie:

»wenn der Wert von `$var` gleich 0 ist«

»wenn der Wert von `$var` größer als 0 ist«

»wenn der Wert von `$var1` größer als der Wert von `$var2` ist«

»wenn der Wert von `$var1 ** 3` größer als der Wert von `$var2` ist«

etc.

Tabelle 4.1: Die Vergleichsoperatoren

Vergleich	Zahlenoperator	String-Operator
Gleichheit	==	eq (equal)
Ungleichheit	!=	ne (not equal)
kleiner	<	lt (less than)
größer	>	gt (greater than)
kleiner gleich	<=	le (less than)
größer gleich	>=	ge (greater or equal)

```
#!/usr/bin/perl -w

$alter = 0;

print "\n Geben Sie bitte Ihr Alter ein: ";
chomp ($alter = <STDIN>);

if ($alter < 18)
  {
  printf ("\n Sorry, Du bist zu jung, um schon Perl " .
          "zu lernen\n");
  }
  else
  {
  printf ("\n Sorry, Sie sind zu alt, um noch Perl " .
          "zu lernen\n");
  }
```

String-Vergleiche

String-Vergleiche werden in der gleichen Weise durchgeführt, wie man Wörter im Wörterbuch nachschlägt. Man beginnt am Anfang der Strings und vergleicht diese Zeichen für Zeichen. Trifft man an einer Position auf zwei unterschiedliche Zeichen, prüft man, welches Zeichen im Alphabet (bzw. im ASCII-Code, siehe Anhang E) zuerst kommt. Der String mit diesem Zeichen ist kleiner als der andere String. Aus diesem Grunde sind folgende Vergleiche wahr:

```
"aaaz" lt "aaba"   # ASCII-Code von a ist kleiner als b
"Zaaz" lt "aaaz"   # ASCII-Code der Großbuchstaben ist
                   # kleiner als der Code der Kleinbuchstaben
"1aaz" lt "aaaz"   # ASCII-Code der Ziffern ist kleiner als
                   # der Code der Buchstaben
```

Das folgende Programm nutzt den Vergleichsoperator ne (Test auf Ungleichheit) für eine kleine, harmlose Passwortabfrage:

```
#!/usr/bin/perl -w

$passwort = "krxbdf";         # Passwort

print "\n Geben Sie bitte das Passwort ein: ";
chomp ($passwort = <STDIN>);

$passwort = lc($passwort);    # Eingabe in Kleinbuchstaben
                              # umwandeln

if ($passwort ne "krxbdf")    # Eingabe ungleich Passwort?
  {
  printf ("\n Sorry, falsches Passwort\n");
  }
  else
  {
  printf ("\n Gut geraten!\n");
  }
```

Zahlen versus Strings

Warum gibt es für Zahlen und Strings unterschiedliche Operatoren für die gleiche Art von Vergleich?

Bedenken Sie, dass Perl automatisch Zahlen in Strings umwandelt und umgekehrt. Ob in einem Ausdruck eine Zahl in einen String umgewandelt oder ein String in eine Zahl umgewandelt wird, hängt dabei von dem jeweiligen Operator ab. Das ist nicht ganz neu für uns. Schon bei der Addition/Konkatenation gab es zwei unterschiedliche Operatoren für Zahlen (+) und Strings (.). Ein paar Beispiele sollen die Bedeutung der unterschiedlichen Operatoren verdeutlichen:

```
$var1 = 12.5;
$var2 = -3;
$str1 = "aaaz";
$str2 = "aaba";

# Zahlenvergleiche
if ($var1 == 12.5)       # wahr
if ($var1 != 12.5)       # falsch
if ($var1 >= $var2)      # wahr
```

```
# Stringvergleiche
if ($str1 eq "aaaz")    # wahr
if ($str1 ne "aaaz")    # falsch
if ($str1 ge $str2)     # falsch1

# Vorsicht: gemischte Vergleiche
if ($var1 >= $str1)     # wahr, $str1 wird in einen
                        # Zahlenwert umgewandelt. Da in $str1
                        # nur "aaaz" steht, wird der String zu
                        # 0 und 12.5 ist größer als 0

if ($var1 ge $str1)     # falsch, $var1 wird in den String
                        # "12.5" umgewandelt. Da die Ziffern
                        # im ASCII-Code vor den Buchstaben
                        # kommen, ist der String "12.5"
                        # kleiner als "aaaz". Die Aussage er
                        # wäre größer, ist folglich falsch

if ($var1 eq "12.5")    # wahr
```

4.2.3 Die logischen Verknüpfungen

Bisher hatten wir es nur mit Bedingungen zu tun, die einen einfachen Vergleich durchführen:

```
if ($var1 < $var2)     # $var1 kleiner als $var2
if ($str1 lt $str2)    # $str1 kleiner als $str2
if ($var1 != 0)        # $var1 ungleich 0
if ($var1)             # Kurform für ($var1 != 0)
```

Ab und an kommt es jedoch vor, dass man in einer Bedingung mehrere Vergleiche durchführen möchte. Das folgende Skript zur Berechnung der Vielfachen von 2 stellt beispielsweise sicher, dass die vom Anwender eingegebenen Zahlen nicht nur positiv, sondern auch kleiner als 31 sind, da die Ergebnisse durch die Potenzierung sehr schnell ansteigen.

---

1 String-Vergleiche werden in der gleichen Weise durchgeführt, wie man Wörter im Wörterbuch nachschlägt. Man beginnt am Anfang der Strings und vergleicht diese Zeichen für Zeichen. Trifft man an einer Position auf zwei unterschiedliche Zeichen prüft man, welches Zeichen im Alphabet (bzw. im ASCII-Code) zuerst kommt. Der String mit diesem Zeichen ist kleiner als der andere String. Aus diesem Grunde ist "aaaz" kleiner als "aaba".

```
#!/usr/bin/perl -w

$basis    = 2;
$exponent = 0;

print "\n Geben Sie den Exponent ein: ";
chomp ($exponent = <STDIN>);

if ($exponent >= 0  && $exponent < 31)
  {
  printf (" %d ^ %d: %d\n", $basis, $exponent, $basis **
$exponent);
  }
  else
  {
  print " Ausdruck kann nicht berechnet werden\n";
  }
```

In der `if`-Bedingung werden die beiden Vergleiche mit Hilfe des `&&`-Operators verknüpft. Der `&&`-Operator entspricht einer logischen UND-Verknüpfung, d.h. die gesamte Bedingung ist nur dann wahr, wenn beide Vergleiche wahr sind.

Tabelle 4.2: Logische Operatoren

Operator	**Bedeutung**	**Auswertung**
! A1	Logische Verneinung	wahr, wenn A1 falsch
A1 && A2	Logisches UND	wahr, wenn A1 und A2 beide wahr
A1 || A2	Logisches ODER	wahr, wenn A1 oder A2 oder beide wahr
A1 ^ A2	Exklusives ODER	wahr, wenn entweder A1 oder A2 wahr

Es gibt noch einen zweiten Satz logischer Operatoren: `not`, `and`, `or` und `xor`. Diese unterscheiden sich von !, &&, || und ^ dadurch, dass sie einen wesentlich niedrigerer Rang haben (vergleiche Tabelle 3.7).

4.3 Verzweigungen

Oft ergibt sich die Situation, dass eine Variable auf mehrere Werte getestet werden soll, damit dann entsprechende Anweisungen abgearbeitet werden. Ein typisches Beispiel ist die Auswertung eines Menüs. Zuerst gibt das Programm ein Menü aus, das verschiedene mit Kennzahlen versehene Menübefehle zur Auswahl anbietet. Dann wird die vom Anwender eingegebene Kennzahl eingelesen und mit Hilfe einer Reihe von `if`-Anweisungen ausgewertet.

```
#!/usr/bin/perl -w

print <<HERE_MENUE;

Adresse eingeben        <1>
Adresse suchen          <2>
Adressliste ausgeben    <3>
Beenden                 <4>

HERE_MENUE

$eingabe = 0;
chomp ($eingabe = <STDIN>);

if ($eingabe == 1) {
  print "Adresse eingeben \n";
  }
if ($eingabe == 2) {
  print "Adresse suchen \n";
  }
if ($eingabe == 3) {
  print "Adressliste ausgeben\n";
  }
if ($eingabe == 4) {
  print "Beenden \n";
  }
```

Die Formulierung mit den aufeinander folgenden `if`-Anweisungen ist allerdings nicht sonderlich elegant. Übersichtlicher ist die folgende Konstruktion, die in der PERLSYN-Manpage beschrieben ist:

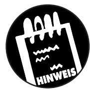
```
SWITCH: {
  $eingabe == 1   && do { print "Adresse eingeben \n";
                          last SWITCH; };
  $eingabe == 2   && do { print "Adresse suchen \n";
                          last SWITCH; };
  $eingabe == 3   && do { print "Adressliste ausgeben \n";
                          last SWITCH; };
  $eingabe == 4   && do { print "Beenden \n";
                          last SWITCH; };
  }
```

Ohne diese Konstruktion bis ins Detail erklären zu wollen, möchte ich Ihnen doch den grundlegenden Aufbau verdeutlichen, damit Sie eigene Konstruktionen dieser Art in Ihre Programme einbauen können.

Alles beginnt damit, dass die gesamte Konstruktion einen eindeutigen Namen erhält. In diesem Beispiel habe ich als Namen SWITCH gewählt. Welchen Namen Sie vergeben, ist an sich egal, allerdings passt SWITCH in diesem Fall recht gut, da diese Art von Konstruktion der `switch`-Verzweigung von C entspricht.

Auf den Bezeichner folgen die einzelnen, alternativen Verzweigungen. Jeder dieser Zweige besteht aus:

- ✘ der Bedingung `($eingabe == 1)`
- ✘ einer Verknüpfung `(&& do)`
- ✘ und dem auszuführenden Anweisungsblock `{..}`. Damit die Konstruktion nach der Ausführung eines ihrer Anweisungsblöcke direkt verlassen wird, endet jeder Anweisungsblock mit der Anweisung `last SWITCH;`.



In Abschnitt 5.4.2 werden wir die Menükonstruktion noch weiter verbessern.

4.4 Die Schleifen

Wenn Sie bestimmte Anweisungen mehrmals hintereinander ausführen müssen, wobei sich lediglich einzelne Parameter nach einem bestimmten Muster verändern, bietet sich der Einsatz einer Schleife an.

Obige Beschreibung klingt furchtbar, dabei ist es im Grunde ganz einfach. Schauen wir uns gleich die erste Schleifenkonstruktion an.

4.4.1 Die while-Schleife

Die `while`-Schleife ist die allgemeinste Schleife, an deren Aufbau man sich die Grundprinzipien der Funktionsweise von Schleifen gut verdeutlichen kann:

```
$i = 0;                                     # Initialisierung
while ($i <= 10)                            # Bedingung
  {
  print "2 hoch $i ist \t", 2**$i, "\n";   # Anweisungen
  ++$i;                                     # Inkrement
  }
```

Jede Schleife besteht aus einem Schlüsselwort und einem Anweisungsblock, der wiederholt ausgeführt werden soll:

```
while
  {
  Anweisungen
  }
```

Darüber hinaus muss man kontrollieren, wie oft die Schleife wiederholt werden soll. Dazu richtet man eine Schleifenvariable ein (die meist einen kurzen Namen hat: beispielsweise `$i` oder `$loop`). Diese Schleifenvariable wird

- ✘ vor der Schleife initialisiert (`$i = 0`)
- ✘ zu Anfang der Schleife in einer Bedingung ausgewertet (`$i <= 10`)
- ✘ in der Schleife verändert (meist wird die Variable inkrementiert: `++$i`, man kann sie aber auch dekrementieren oder ihren Wert in beliebiger anderer Weise ändern)

Durch die Kombination dieser drei Anweisungen kann man festlegen, wie oft die Schleife wiederholt wird. Dies wird deutlich, wenn man nachvollzieht, wie eine solche Schleife ausgeführt wird.

Ausführung einer Schleife

In obigem Skript beispielsweise wird die Schleifenvariable `$i` zu Anfang auf 0 gesetzt. Dann beginnt die Schleife. Zu Beginn der Schleife steht die Schleifenbedingung. Solange diese Bedingung erfüllt ist (wahr ist), solange wird die Schleife wiederholt ausgeführt. Da `$i` anfangs wie gesagt gleich 0 ist und die Schleifenbedingung prüft, ob `$i` kleiner gleich 10 ist, ist die Bedingung erfüllt, und der Anweisungsblock der Schleife wird ausgeführt (man spricht hier von Iteration). Innerhalb des Anweisungsblocks wird die Schleifenvariable inkrementiert, so dass sie nun den Wert 1 hat. Schließlich ist der Anweisungsblock komplett abgearbeitet. Die Programmausführung springt jetzt wieder zurück zur Schleifenbedingung, die erneut überprüft wird. Da 1 immer noch kleiner gleich 10 ist, wird die Schleife erneut ausgeführt und so weiter, bis in der 11. Abarbeitung der Schleife die Schleifenvariable `$i` auf 11 gesetzt wird. Danach ist die Schleifenbedingung nicht mehr erfüllt, und das Programm wird mit der ersten Anweisung hinter der Schleife fortgesetzt.

4.4.2 Die do-while-Schleife

Die `do-while`-Schleife ist eng mit der `while`-Schleife verwandt. Der einzige wirkliche Unterschied zwischen einer `while`- und einer `do-while`-Schleife besteht darin, dass bei der `while`-Schleife die Schleifenbedingung bereits

vor dem ersten Ausführen des Anweisungsblocks getestet wird, während die `do-while`-Schleife den Anweisungsblock auf jeden Fall einmal ausführt und erst danach die Bedingung überprüft.

Dieses Verhalten ist günstig, wenn man einen bestimmten Code mindestens einmal und gegebenenfalls mehrfach ausführen lassen möchte. Eine praktische Anwendung hierfür ist die Implementierung eines Menüs für ein menügesteuertes Programm.

Listing 4.2: menue.pl

```
#!/usr/bin/perl -w

$menue = <<HERE_MENUE;

  Adresse eingeben        <1>
  Adresse suchen          <2>
  Adressliste ausgeben    <3>
  Beenden                 <4>

HERE_MENUE


$eingabe = 0;
do {
   print $menue;
   chomp ($eingabe = <STDIN>);

   SWITCH: {
   $eingabe == 1   && do { print "Adresse eingeben \n";
                           last SWITCH; };
   $eingabe == 2   && do { print "Adresse suchen \n";
                           last SWITCH; };
   $eingabe == 3   && do { print "Adressliste ausgeben \n";
                           last SWITCH; };
   $eingabe == 4   && do { print "Beenden \n";
                           last SWITCH; };
   }

} while ($eingabe != 4);
```

Zu Beginn des Programms wird das Menü angezeigt, die Eingabe des Anwenders eingelesen und gegebenenfalls der zugehörige Menübefehl ausgeführt. Danach wird die Bedingung der `do-while`-Schleife überprüft. Hat der Anwender Option 4 ausgewählt, wird das Programm beendet, ansonsten wird die Schleife erneut ausgeführt und dem Anwender wieder das Menü repräsentiert.

In der `do-while`-Schleife nicht das Semikolon hinter der `while`-Bedingung vergessen!

do-until Die `do-while`-Schleife wird so lange ausgeführt, wie die Schleifenbedingung erfüllt ist. Die `do-until`-Schleife, bei der gegenüber der `do-while`-Schleife einfach das Schlüsselwort `while` durch `until` ersetzt wird, wird beendet, wenn die Schleifenbedingung erfüllt ist.

4.4.3 Die for-Schleife

```
for (Initialisierung, Bedingung, Inkrement)
  {
  Anweisungen;
  }
```

Die `for`-Schleife ist eine weitere Schleifenvariante, bei der die drei Anweisungen zur Bearbeitung der Schleifenvariable (Initialisierung, Bedingung und Inkrement) im Schleifenkopf zusammengezogen sind – was die `for`-Schleife im Vergleich zur `while`-Schleife sehr übersichtlich macht. Eingesetzt wird die `for`-Schleife vor allem dann, wenn die Anzahl der Schleifeniterationen bereits bei Aufsetzen des Codes feststeht. Dies war beispielsweise in unserem einführenden `while`-Schleifenbeispiel der Fall, weswegen wir den Code dieser Schleife jetzt als `for`-Schleife schreiben werden.

```
#!/usr/bin/perl -w
# Ausgabe der ersten 10 Potenzen von 2

for ($i = 0; $i <= 10; ++$i)
  {
  print "2 hoch $i ist \t", 2**$i, "\n";
  }
```

Ausgabe

```
2 hoch 0 ist    1
2 hoch 1 ist    2
2 hoch 2 ist    4
2 hoch 3 ist    8
2 hoch 4 ist    16
2 hoch 5 ist    32
2 hoch 6 ist    64
2 hoch 7 ist    128
2 hoch 8 ist    256
2 hoch 9 ist    512
2 hoch 10 ist   1024
```

Zauber mit Schleifen, Zahlen und Kaninchen

Kürzlich haben mich meine Kinder darauf angesprochen, ob sie nicht zwei Kaninchen haben könnten. Nun wäre es natürlich nett, ein Kaninchenpärchen zu halten, doch ist die Vermehrung dieser Tierchen ja geradezu sprichwörtlich. Vielleicht sollte man daher vor dem Kauf abschätzen, mit wie vielen Kaninchen man nach 10 Jahren zu rechnen hat.

Ein Kaninchenweibchen kann direkt nach einer Geburt wieder trächtig werden. Bei einer Tragzeit von ziemlich genau einem Monat bestehen daher gute Chancen, dass ein Kaninchenweibchen im Jahr sieben Würfe zu durchschnittlich (hier schwanken die Voraussagen etwas) vier Jungen austrägt. Die in einem Jahr geborenen Jungen erzeugen meist erst im darauffolgenden Jahr eigenen Nachwuchs. Ein Kaninchenpärchen erzeugt nach dieser Rechnung also jedes Jahr 28 Nachkommen. Wie sieht die Population dann nach 20 Jahren aus?

Solch ungehinderte Wachstumsvorgänge folgen meist der Formel:

```
N(t) = N(0) * e^(k*t)
```

Da wir wissen, dass im ersten Jahr aus zwei Kaninchen insgesamt 30 Kaninchen werden, lässt sich die Wachstumsrate k (pro Jahr) leicht bestimmen:

```
k = ln(30/2) = ln15
```

Das folgende Programm errechnet uns daraus die Populationsgröße für die ersten 10 Jahre:

Listing 4.3: kaninchen.pl

```
#!/usr/bin/perl -w

$population = 2;
$jahr = 0;

for($jahr = 1; $jahr <= 10; $jahr++)
     {
     $population = 2*exp(log(15)*$jahr);
     printf("\n Nach dem %d. Jahr sind es %.0f Tiere.",
            $jahr, $population);
     }

print "\n";
```

Ich überlassen es Ihnen, dieses kleine Programm nachzuprogrammieren und die wundersame Kaninchenvermehrung zu bestaunen.

4.4.4 Die foreach-Schleife

Die `foreach`-Schleife ist so eine Art Spezialisierung und Verkürzung der `for`-Schleife. Ihre Eigenart ist, dass sie die Elemente einer Liste durchläuft. Damit fallen die Formulierungen einer Schleifenbedingung weg: der Anweisungsblock der Schleife wird einmal für jedes Element in der Liste ausgeführt. Danach wird die Schleife automatisch beendet. Auch der Schleifenvariablen kommt eine andere Bedeutung zu: Ihr wird in jedem Schleifendurchgang der Wert des gerade bearbeiteten Listenelements zugewiesen.

```
foreach $i LISTE {
  Anweisungen;
  }
```

Listen kann man in Perl auf zwei Arten formulieren:

- ✘ als Aufzählung der Elemente in der Liste

  ```
  (2, 3, 5, 7, 11, 13, 17, 19, 23)
  ('weiss', 'rot', 'blau', 'gruen')
  ```

- ✘ als Bereichsangabe (wobei nur Listen mit aufsteigenden Werten erzeugt werden können)

  ```
  (1..10)
  ('a'..'z')
  ```

Ein kleines Beispiel soll den Einsatz der `foreach`-Schleife verdeutlichen. Weitere Beispiele gibt es in Kapitel 5 zu den Arrays und Hashes.

```
#!/usr/bin/perl -w

print "Die ersten Primzahlen \n";

foreach $i (2, 3, 5, 7, 11, 13, 17, 19, 23) {
   print $i, "\n";
   }
```

4.5 Abbruchbefehle für Schleifen

Standardmäßig wird eine Schleife beendet, wenn die Schleifenbedingung nicht mehr erfüllt ist. Es gibt aber auch spezielle Befehle, mit denen man in die Ausführung der Schleife eingreifen kann.

Tabelle 4.3: Abbruchbefehle für Schleifen

Befehl	Beschreibung
redo	Mach´s noch einmal Sam! Mit `redo` können Sie veranlassen, dass die aktuelle Schleifeniteration an der Stelle des `redo`-Aufrufs abgebrochen und von neuem begonnen wird. Die Schleifenbedingung wird dabei nicht erneut ausgewertet (und in `for`-Schleifen wird auch nicht die Inkrement-Anweisung ausgeführt).
next	Mit Hilfe des Befehls `next` kann die aktuelle Schleifeniteration abgebrochen werden. Nach Ausführung der `next`-Anweisung werden die restlichen Anweisungen des Schleifenblocks übersprungen und die nächste Schleifeniteration wird begonnen. Das heißt, die Schleifenbedingung wird erneut ausgewertet (und in `for`-Schleifen wird auch die Inkrement-Anweisung ausgeführt).
last	Mit Hilfe des Befehls `last` kann die ganze Schleife abgebrochen werden. Die Programmausführung wird danach mit der ersten Anweisung nach der Schleife fortgesetzt.

```
#!/usr/bin/perl -w

$text = <<HERE_TEXT;

Geben Sie eine Zahl zwischen 0 und 9 ein.
(0 zum Beenden des Programms)

HERE_TEXT

$eingabe = 1;

while ($eingabe != 0)
  {
  print $text;

  chomp ($eingabe = <STDIN>);
  if ($eingabe < 0 || $eingabe > 9) {
     redo;
     }

  print "Korrekte Eingabe\n";
  }
```

Die Befehle `redo`, `next` und `last` versagen, wenn Sie versuchen, aus einer `do`-Schleife zu springen. Dies liegt daran, dass `do`-Konstruktionen in Perl keine wirklichen Kontrollstrukturen, sondern Funktionsaufrufe darstellen.

Benannte Schleifen

Wenn Sie Schleifen verschachteln, beziehen sich die Befehle `redo`, `next` und `last` stets auf die innerste Schleife, in der sie aufgerufen werden. Wenn Sie die Befehle auf eine äußere Schleife beziehen wollen, müssen Sie die Schleifenkonstrukte mit Bezeichnern versehen:

```
#!/usr/bin/perl -w

AUSSEN: foreach $i (1..6) {
  print "\n";
  INNEN: foreach $j (6, 5, 4, 3, 2, 1) {
    if ($j == 3) {
       next AUSSEN;
       }
    print "($i, $j) ";
    }
  }
```

4.6 Vertiefung: Abkürzungen

Perl ist die Sprache der unzähligen Wege. Kaum ein Konstrukt, das man nicht auch anders, kürzer formulieren könnte. Die wichtigsten Abkürzungen für Bedingungen und Schleifen möchte ich Ihnen kurz vorstellen.

Der Bedingungsoperator ?:

Die `if`-Konstruktion

```
if Bedingung {
  $var = Ausdruck1;
  }
  else {
  $var = Ausdruck2;
  }
```

kann man auch verkürzt schreiben als:

```
$var = Bedingung ? Ausdruck1 : Ausdruck2;
```

Bedingte Anweisungen

Um die Ausführung einer einzelnen Anweisung von einer Bedingung abhängig zu machen, kann man auch folgende Konstruktion verwenden:

```
Anweisung if² Bedingung;
```

Bei dieser Konstruktion wird zuerst die Bedingung ausgewertet. Ist diese erfüllt, wird die Anweisung ausgeführt, sonst nicht. Nutzen Sie diese Konstruktion, um einfache `if`-Bedingungen zu formulieren, die in eine Zeile passen:

```
$wurzel = sqrt($eingabe) if ($eingabe > 0);
```

Oder um `print`-Anweisungen, die Ihnen bei der Fehlersuche im Programm helfen, in Abhängigkeit vom der Definition einer Variable ein- und auszuschalten.

```
$debug = 1;
$var = 1;

print $var if defined $debug;
```

Analog gibt es eine Konstruktion zur bedingten Ausführung eines Anweisungsblocks:

```
do {Anweisungen} if Bedingung
```

Die Standardvariable $_

`$_` ist eine von Perl vordefinierte skalare Variable, die im Zusammenhang mit etlichen Perl-Funktionen und -Konstrukten verwendet werden kann. Ein Beispiel für den möglichen Einsatz der `$_`-Variablen ist die `foreach`-Schleife.

```
print "Die ersten Primzahlen \n";

foreach $i (2, 3, 5, 7, 11, 13, 17, 19, 23) {
   print $i, "\n";
   }
```

Hier werden die einzelnen Elemente der Liste nacheinander in der Variablen `$i` abgelegt. Wenn wir keine Variable übergeben, kopiert Perl die Listenelemente automatisch in die Standardvariable `$_`. Obige Schleife kann daher auch folgendermaßen formuliert werden:

```
print "Die ersten Primzahlen \n";

foreach (2, 3, 5, 7, 11, 13, 17, 19, 23) {
   print $_, "\n";
   }
```

---

2 Statt `if` können Sie auch `unless` verwenden.

4.7 Vertiefung: Fallstricke

In diesem Kapitel haben wir einen gewaltigen Sprung nach vorne gemacht. Bedingungen und Schleifen sind wichtige Konstruktionselemente, ohne die kaum ein anständiges Programm auskommt. Und das Schönste dabei ist: diese Konstrukte sind gar nicht so schwer zu verstehen.

Doch Vorsicht! So einfach Bedingungen und Schleifen im Gebrauch auch sein mögen, so schnell können sie auch zu hässlichen Fehlern führen. Bevor wir zum nächsten Kapitel schreiten, möchte ich Sie daher noch auf zwei besonders hässliche Fehler hinweisen, die erst zur Laufzeit auftreten.

Endlosschleifen

Bei jeder Schleifenkonstruktion ist penibel darauf zu achten, dass die Schleife irgendwann korrekt verlassen wird (sei es, dass die Schleifenbedingung zum Abbruch führt oder dass die Schleife vermittels `last` verlassen wird.

Betrachten Sie beispielsweise folgende Schleife:

```
for ($i = 99; $i > 0; $i+=2)
  {
  # von 99 bis 1 alle ungeraden Zahlen ausgeben
  print "$i hat den Wert \t", $i, "\n";
  }
```

Hier bleibt die Bedingung stets erfüllt, weil die Schleifenvariable `$i` inkrementiert, statt dekrementiert wird.

Auch der Urheber der folgenden Schleife wurde ein Opfer seiner Schusseligkeit:

```
for ($i = 99; $i != 0; $i-=2)
  {
  print "$i hat den Wert \t", $i, "\n";
  }
```

Diese Schleife wird endlos fortgesetzt, da die Schleifenvariable `$i` nie den Wert 0 annimmt (sie springt von 1 auf -1). Stattdessen zählt Sie bis zum Sankt-Nimmerleins-Tag (oder zumindest bis zum Programmabbruch durch Bereichsüberschreitung) die negativen Zahlen runter.

Noch teuflischer wird es freilich, wenn Sie den Wert der Schleifenvariablen auch noch innerhalb der Schleife verändern. Dies ist zwar durchaus erlaubt und kann auch sinnvoll eingesetzt werden, doch achten Sie dann ganz besonders darauf, dass die Abbruchbedingung irgendwann erfüllt wird.

Ein ganz besonderes Fehlerpotential steckt auch in der Kombination von `while`-Schleifen mit `next`.

Da in einer `while`-Schleife die Schleifenvariable innerhalb der Schleife verändert wird, kann es schnell passieren, dass Sie die `next`-Anweisung vor die Bearbeitung der Schleifenvariable setzen. Unter Umständen führt dies dann dazu, dass sich die Schleifenvariable nicht mehr ändert und die Schleife endlos fortgesetzt wird.

```
$i = 0;
while ($i < 20)
  {
  next unless $i % 5;
  print "2 hoch $i ist \t", 2**$i, "\n";
  ++$i;
  }
```

Hier sollen die ersten Potenzen von 2 ausgegeben werden. Lediglich die Potenzen mit Vielfachen von 5 sollen ausgeschlossen werden. Dafür sorgt die Anweisung `next unless $i % 5;`. Erinnern Sie sich daran, dass der Modulo-Operator den Rest einer Integer-Division liefert. Dies bedeutet umgekehrt, dass er nur dann 0 (im Booleschen Sinne also »falsch«) zurückliefert, wenn die Division ohne Rest aufgeht. In unserem Code ist dies der Fall, wenn in `$i` ein Vielfaches von 5 steckt.

Die `next`-Anweisung funktioniert in obigem Beispiel übrigens ganz wunderbar. Das Problem ist, dass die Inkrementierung `++$i` unter der `next`-Anweisung steht. Beim Eintritt in die Schleife enthält `$i` den Wert 0. Da 0 % 5 gleich 0 ist, ist die Bedingung »unless 0« gleich »if not 0« gleich »if 1« erfüllt, und die `next`-Anweisung wird ausgeführt. Es startet also direkt die nächste Schleifeniteration. Da sich aber der Wert von `$i` nicht geändert hat, endet auch diese Iteration bei der `next`-Anweisung und so weiter und so fort.

Verkürzte Auswertung der logischen Operatoren

Logisch verknüpfte Ausdrücke werden nicht immer vollständig ausgewertet

Wenn Sie mit Hilfe der logischen Operatoren (siehe 5.2.3) komplexere Bedingungen aufbauen, müssen Sie daran denken, dass diese nicht in jedem Fall vollständig berechnet werden. Dies liegt daran, dass der Perl-Interpreter oft schon bereits nach Auswertung eines der ersten Teilausdrücke feststellen kann, ob der Gesamtausdruck wahr oder falsch sein wird. In solchen Fällen wertet der Interpreter die weiteren Teilausdrücke nicht mehr aus.

```
if (($x < $y) && ($x++ < 5))
```

Obiger Code prüft, ob `$x` kleiner als `$y` und kleiner als 5 ist und sorgt dafür, dass `$x` um eins erhöht wird. Ob dieser Code nicht nur syntaktisch, sondern auch semantisch korrekt arbeitet, hängt davon ab, was der Programmierer

beabsichtigt hat. Soll $x in jedem Fall inkrementiert werden, unabhängig davon, ob die Bedingung erfüllt ist oder nicht, so ist dem Programmierer ein Fehler unterlaufen.

Dies liegt daran, dass eine logische Aussage, die aus zwei Teilaussagen besteht, die mit einem logischen UND verbunden sind, nur dann »wahr« sein kann, wenn beide Teilausdrücke »wahr« sind. Ist in obigem Beispiel also $x größer oder gleich $y, ist der erste Teilausdruck »falsch«, und die if-Bedingung kann nicht mehr erfüllt werden. Der zweite Teilausdruck wird daher gar nicht mehr überprüft, d.h. er wird überhaupt nicht mehr ausgeführt. Folglich wird $x auch nicht mehr inkrementiert.

4.8 Fragen und Übungen

1. Ist die folgende if-Bedingung korrekt?

```
$i = 3;

if($i)
  {
  # tue irgendetwas
  }
```

2. Ist die folgende while-Bedingung korrekt? Wenn ja, wie könnte Sie beendet werden?

```
$i = 3;

while($i)
  {
  ...
  }
```

3. Wann werden die folgenden Bedingungen als »wahr« ausgewertet? (Hilfe: Betrachten Sie $x und $y als Koordinaten in der Ebene.)

```
if (($x < 35 && $x > 20) && ($y < 100 && $y > 80)
    || ($x < 150 && $x > 120) && ($y < 95 && $y > 75))
```

und

```
if (sqrt($x**2+$y**2) == 1)
```

4. Worin besteht der Fehler in der folgenden Schleife?

```
$i = 1;
$j = 0;
$k = 0;

print "i \t j \t k";
while ($i < 10)
    {
    $j = $i*$i - 1;
    $k = $j*$j - 1;
    print "$i \t $j \t $k\n";
    }
```

5. Die folgende Schleife definiert mit Hilfe des Komma-Operators zwei Schleifenvariablen. Wie sieht die Ausgabe der Schleife aus?

```
#!/usr/bin/perl -w

for($n = 0, $m = 0; $n < 10 && $m <3; $n++, $m += 2)
  {
  print "n * m = ", $n*$m, "\n";
  }
```

6. Setzen Sie eine `for`-Schleife zur Berechnung der ersten hundert Quadratzahlen auf.

7. Wandeln Sie die `for`-Schleife aus Übung 6 in eine `while`-Schleife um.

8. Wandeln Sie die `for`-Schleife aus Übung 6 in eine `foreach`-Schleife um und verwenden Sie die Standardvariable `$_`.

9. Setzen Sie ein Perl-Skript auf, das für Winkel von 0 bis 360 Grad den Sinus berechnet. Für die Berechnung des Sinus können Sie die vordefinierte Perl-Funktion `sin` verwenden, der Sie den Winkel in Bogenmaß (Radiant)[3] übergeben. Formatieren Sie die Ausgabe in einer Tabelle mit zwei Spalten für den Winkel und den Sinus. Es genügt, wenn Sie die Vielfache von 30 Grad ausgeben.

10. Benutzen Sie den Modulo-Operator, um zu überprüfen, ob eine Zahl gerade ist.

11. Nutzen Sie die Binärcodierung des Computers, um zu entscheiden, ob eine Zahl gerade ist.

---

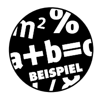
3 1 rad = $360^\circ/2\pi$; $1^\circ = 2\pi/360$ rad.

Kapitel 5

Listen, Arrays und Hashes

Immer nur einzelne Daten zu bearbeiten, wird auf die Dauer langweilig. Irgendwann wünscht man sich, Daten auch in größeren Verbunden bearbeiten zu können. Wie kann man beispielsweise schnell Variablen für 100 Zahlen erzeugen? Wie kann man einen Text in einzelne Wörter zerlegen? Wie kann man Adressen oder andere strukturierte Daten sinnvoll in einem Programm repräsentieren, wie bearbeiten? Fragen, auf die wir in diesem Kapitel Antworten finden wollen.

Zu diesem Zweck werden wir uns mit dem Datenkonstrukt der Liste sowie den Variablentypen Array und Hash auseinandersetzen. Ich werde ihnen zeigen, wie man mit diesen Variablen programmiert (d.h. wie man sie definiert, wie man sie ausgibt, wie man auf einzelne Elemente in einem Array oder Hash zugreift), und Ihnen erste Beispiele dafür geben, welche Möglichkeiten in diesen Konstrukten stecken. Weiterführende Anwendungen finden Sie im Fortgeschrittenen- und Praxis-Teil (siehe auch Index unter Arrays und Hashes).



Im Einzelnen lernen Sie,

- ✘ was Listen sind und wie man sie definiert
- ✘ was Arrays sind und wie man sie definiert
- ✘ wie man auf einzelne Elemente in Arrays zugreift
- ✘ wie man Zufallszahlen erzeugt
- ✘ wie man Arrays sortiert
- ✘ wie man in Arrays sucht

- wie man Strings und Arrays ineinander umwandelt
- was Hashes sind und wie man sie definiert
- wie man auf einzelne Elemente in Hashes zugreift
- wie man Hashes sortiert
- wie man sich die Schlüssel oder Werte eines Hash als Array zurückliefern lassen kann

Folgende Elemente lernen Sie kennen

Listen, Hashes und Arrays. Die Operatoren `..` und `qw` zur Definition von Listen; eine Reihe von Listenfunktionen: `map`, `print`, `pop`, `push`, `shift`, `unshift`, `splice`, `sort`, `grep`; zwei spezielle Hash-Funktionen: `keys`, `values` und vier Funktionen zum Löschen und Verifizieren von Variablen: `delete`, `undef`, `exists`, `defined`; die besonderen Vergleichsoperatoren `cmp` und `<=>` und last but not least: einen Furby.

5.1 Listen

Eine Liste ist – aus Sicht von Perl – eine beliebige Zusammenstellung skalarer Daten.

5.1.1 Listendefinition

In der Listendefinition werden die einzelnen Daten, die Elemente der Liste, durch Kommata voneinander getrennt. Die Liste als Ganzes wird in runde Klammern gesetzt.

In Listen können Sie beliebige Zahlenzusammenstellungen ablegen.

```
(5, 133, -12.5)
(5, 4, 3, 2, 1)            # Countdown
(125, 127, 129, 135, 155)  # systolische Blutdruckwerte in
                           # den letzten 5 Tagen vor einer
                           # wichtigen Klausur
```

Außer Zahlenwerten kann man selbstverständlich auch Strings in Listen verwalten – etwa die Namen Ihrer besten Freunde oder auch ganze Sätze.

```
("Werner", "Otto", "Gaston", "Calvin")
("Na, wie geht\'s", "Lass mich schlafen!", "Schon acht Uhr?")
```

Schließlich können Sie in einer Liste skalare Daten der verschiedenen elementaren Datentypen beliebig mischen – obwohl sich die Frage stellt, wofür dies gut sein soll[1]:

```
(5, "Hallo", -0.344, "bzz", 1)
```

Der Bereichsoperator ..

Buchstabenfolgen sowie Folgen natürlicher Zahlen kann man verkürzt mit Hilfe des Bereichsoperators formulieren:

```
(-2..2, 3..6)   # gleich (-2,-1,0,1,2,3,4,5,6)
('aa'..'ad')    # gleich (aa,ab,ac,ad)
```

Der qw-Operator

Damit Ihnen bei längeren Listen das Eintippen der Kommata nicht zu lästig wird, stellt Ihnen Perl den `qw`-Operator zur Verfügung. Der `qw`-Operator interpretiert Leerzeichen automatisch als Elementgrenzen. Und wenn Sie den `qw`-Operator für String-Listen verwenden, erspart er Ihnen zusätzlich noch das Tippen der Anführungszeichen.

```
qw(125 127 129 135 155)
qw(Werner Otto Gaston Calvin)
```

Verwenden Sie den `qw`-Operator nicht, wenn Ihre Liste Strings enthält, die aus mehreren Wörtern bestehen.

5.1.2 Grenzen des Machbaren

Diese Überschrift ist ein wenig provokant formuliert; Perl-Kenner wissen, dass in Perl selbstverständlich alles geht – nur manchmal eben nicht so, wie sich der Programmierer es wünscht.

Zuweisung an Skalare

Sie können eine Liste nicht in einem Skalar speichern (schließlich ist ein Skalar ja eine Variable für einen einzelnen Wert und nicht eine Liste von Werten). Stattdessen brauchen Sie einen neuen Typ von Variablen: ein Array (siehe nachfolgenden Abschnitt). Wenn Sie dennoch versuchen, einer skalaren Variablen eine Liste zuzuweisen, steht in der skalaren Variablen danach der Wert des letzten Listenelements.

```
$var = ("Werner", "Otto", "Gaston", "Calvin");
print $var;   # Ausgabe: Calvin
```

1 Der Abschnitt zu den Hashes wird uns darauf eine Antwort geben.

Listen als Listenelemente

Beispielsweise können Sie in eine Liste keine andere Liste als Element einfügen. Das heißt, Sie können zwar schon bei der Listendefinition eine Liste als Element angeben, doch wird diese untergeordnete Liste in ihre einzelnen Elemente aufgebrochen. Die folgende Listendefinition

```
(1, 2, 3, ("str1", "str2"), 4, 5)
```

entspricht also

```
(1, 2, 3, "str1", "str2", 4, 5);
```

Dies hat durchaus praxisrelevante Folgen, beispielsweise wenn Sie die Länge einer Liste ermitteln. Die obige Liste enthält nämlich nicht sechs sondern sieben Elemente!

5.1.3 Furby

Im vorangehenden Abschnitt habe ich Ihnen erklärt, was man mit Listen nicht machen kann. In diesem Abschnitt will ich Ihnen zeigen, was man mit Listen machen kann. Man kann mit Hilfe von Listen nämlich Furbys programmieren.

Sie wissen nicht, was Furbys sind? Furbys sind kleine, knuddelige Plüschtiere, die ungefragt Kommentare abgeben, verdammt teuer sind und derzeit hoch in Kurs stehen (bei Erscheinen dieses Buches vielleicht auch schon nicht mehr). Zur Einstimmung auf die Listen und Arrays will ich Ihnen zeigen, wie Sie Ihren Rechner in einen Furby verwandeln können.

Listing 5.1: furby1.pl

```
#!/usr/bin/perl -w

@sprich = ("Na, wie geht\'s?", "Lass mich schlafen!",
           "Ist es schon acht Uhr?", "Ich mag dich.",
           "Ein Pokemon.. den mach ich kalt.");

print "\n $sprich[rand @sprich] \n";
```

Abb. 5.1: Ich mag dich auch

Wenn Sie dieses Skript ausführen, wählt es aus der Liste ein zufälliges Element aus und gibt es auf die Konsole aus – als Furby-Prototyp schon ganz nett. Wie aber funktioniert dieses Skript? Um diese Frage beantworten zu können, müssen wir noch einiges über Listen und Arrays lernen. Ich verschiebe die Beantwortung dieser Frage daher auf das Ende von Abschnitt 5.2.2.

5.2 Arrays

Technisch gesehen ist ein Array eine spezielle Datenstruktur, in der man beliebig viele skalare Daten verwalten kann. Aus Sicht des Programmierers ist ein Array eine Variable, der er Listen zuweisen kann.

5.2.1 Arrays definieren

Arrays sind Variablen. Wie Sie bereits wissen, beginnt in Perl jede Variable mit einem speziellen Präfix, das den Typ der Variable anzeigt. Wer unter den Lesern ein besonders gutes Gedächtnis hat, der wird sich auch noch daran erinnern, dass das Präfix für Array-Variablen der Klammeraffe @ ist.

```
@blutdruckwerte = (125, 127, 129, 135, 155);
```

Wie Sie sehen, werden Array-Variablen in der gleichen Weise definiert wie Skalare, nur dass man statt `$` das Präfix `@` und statt eines einzelnen Wertes eine Liste zur Initialisierung verwendet.

Doch nicht immer ist es sinnvoll oder möglich, alle Elemente bei der Einrichtung des Arrays anzugeben. Oder hätten Sie Lust, ein Array mit den ersten hundert geraden Zahlen zu initialisieren?

```
@gerade = qw(2 4 6 8 10 12 14 16 18 20 22 24 26 ... );
```

In so einem Fall kann man sich behelfen, in dem man das Array durch Angabe des letzten Elements definiert:

```
$gerade[99] = 200;
```

Diese Anweisung weist dem 100-ten Element im Array einen Wert zu (hier 200). Ist das Array noch nicht definiert oder hat es bisher weniger als 100 Elemente, wird es entsprechend angelegt oder erweitert. Achtung! Das Array hat zwar Platz für 100 Elemente, diese Elemente haben aber noch keine definierten Werte. Um den Elementen eines Arrays Werte zuzuweisen, nutzt man meist Schleifen oder die Funktion `map` (siehe unten).

5.2.2 Mit Arrays arbeiten

Die Programmierung mit Arrays ist nicht mit der Programmierung mit skalaren Variablen zu vergleichen. Skalare sind einzelne Daten, die man auswertet und bearbeitet. Arrays dagegen sind »Container«, in denen man skalare Daten aufbewahrt. Folglich stehen bei der Programmierung mit Arrays ganz andere Fragen im Vordergrund:

Wie viele Elemente befinden sich gerade in dem Array?

Wie kann ich auf die einzelnen Elemente im Array zugreifen?

Wie kann ich Elemente hinzufügen, wie löschen?

Anzahl der Elemente im Array

Die Anzahl der Elemente in einem Array kann man bestimmen, indem man das Array einer skalaren Variablen zuweist:

```
@meinArray = (1, 5, 7);
$anzahl = @meinArray;
print $anzahl, "\n";     # Ausgabe 3
```

Beachten Sie, dass Perl die Zuweisung eines Arrays und die Zuweisung einer Liste an eine skalare Variable unterschiedlich behandelt! Die Zuweisung eines Arrays liefert die Anzahl der Elemente im Array, die Zuweisung einer Liste liefert das letzte Element in der Liste.

Auf einzelne Elemente zugreifen

Die Elemente eines Arrays stehen in einer festen Reihenfolge und können daher über ihre Position angesprochen werden. Die Position hängt man als Index in eckigen Klammern an den Array-Namen an. Doch Vorsicht! Da man auf ein einzelnes Element, sprich einen Skalar, zugreift, muss man vor den Array-Namen das Präfix `$` setzen:

```
@meinArray = (1, 5, 7);
print $meinArray[2];          # Ausgabe 7
```

Oops, ist uns da ein Tippfehler unterlaufen? Das zweite Element im Array müsste doch den Wert 5 und nicht 7 haben? »Das ist schon richtig so«, werden die C-Hasen unter Ihnen schmunzelnd denken. Tatsächlich beginnen die Indizes zu den Array-Elementen bei 0 und nicht bei 1. Das Array `@meinArray` besteht daher aus den Elementen `$meinArray[0]`, `$meinArray[1]` und `$meinArray[2]`.

Was die alten C-Hasen jedoch vermutlich nicht wissen, ist, dass man die Indizierung in Perl ändern kann. In Wirklichkeit beginnt die Indizierung mit dem Wert, der in der Standardvariablen `$[` steht. Es wird jedoch allgemein davon abgeraten, dieser Variablen einen anderen Wert (beispielsweise 1) zuzuweisen, da einfach zu viele Programmierer an die Indizierung ab 0 gewöhnt sind.

Wo auch immer die Indizierung beginnen mag, auf eines können Sie sich verlassen: der höchste Index des Arrays `@meinArray` wird von Perl für Sie in der Standardvariablen `$#meinArray` festgehalten.

Arrays durchlaufen

Mit Hilfe des indizierten Zugriffs, einer `for`-Schleife und der Anzahl der Elemente im Array ist es kein Problem, ein Array Element für Element zu durchlaufen und zu bearbeiten.

```
@meinArray = (0..20);

for($i = 0; $i < @meinArray; $i += 2) {
  print $meinArray[$i], " ";
  }
```

Hier wird allerdings wegen der Inkrementanweisung `$i +=2` nur jedes zweite Element ausgegeben. Das Beispiel ist durchaus mit Bedacht so gewählt, denn wenn Sie wirklich alle Elemente eines Arrays durchlaufen wollen, gibt es bessere Möglichkeiten als eine `for`-Schleife.

foreach

Am einfachsten und sichersten durchläuft man ein Array mit Hilfe der `foreach`-Schleife, da in diesem Fall die Indizierung komplett wegfällt:

```
$zaehler = 0;
foreach $elem (@meinArray) {
  $elem = $zaehler;
  ++$zaehler;
  }
```

Oder kürzer:

```
$zaehler = 0;
foreach (@meinArray) {
  $_ = $zaehler++;
  }
```

map

Der Funktion `map` kommt eine ganz besondere Bedeutung zu. Mit ihrer Hilfe kann man nicht nur alle Elemente eines Arrays durchlaufen und bear-

beiten, sondern auch das Ergebnis der Bearbeitung als neue Liste zurückliefern lassen (während die Elemente im ursprünglichen Array unverändert bleiben).

Wie kann man mit `map` die Elemente eines Arrays bearbeiten? Der Trick dabei ist, dass die Funktion `map` die Elemente gar nicht selbst bearbeitet. Stattdessen übernimmt `map` als erstes Argument eine Funktion (oder einen in geschweiften Klammern stehenden Ausdruck) und wendet diese Funktion (oder den Ausdruck) auf die einzelnen Elemente des Arrays an. Dies wiederum geht aber nur, wenn die Funktion (bzw. der Ausdruck) weiß, wie sie auf das jeweils zu bearbeitende Element zugreifen kann. Deshalb legt `map` die Elemente in der Standardvariablen `$_` ab, und jede Funktion (jeder Ausdruck), der auf `$_` operiert, kann mit `map` aufgerufen werden.

Genug der Theorie. Was muss man tun, um alle Elemente in einem Array mit 2 zu multiplizieren und das Ergebnis als neues Array zurückzuliefern? Zuerst setzt man einen Ausdruck auf, der das jeweils aktuelle Element (steht in `$_`) mit 2 multipliziert.

```
$_ * 2
```

Danach setzt man diesen Ausdruck in geschweifte Klammern und übergibt ihn zusammen mit dem Array der Elemente an `map`.

```
@meinArray = (0..10);
@neu = map {$_ * 2 } @meinArray;
```

Um für alle Elemente im Array die Wurzel zu berechnen, übergibt man einfach die Funktion `sqrt` als erstes Argument. (Die vordefinierten mathematischen Funktionen operieren alle auf `$_`, wenn man ihnen keine Parameter übergibt.)

```
@meinArray = (0..10);
@neu = map (sqrt, @meinArray);
```

Tja, und wenn man jetzt kein neues Array zurückliefern will, sondern die Elemente des übergebenen Arrays selbst ändern möchte? Dann weist man das Ergebnis der Berechnung einfach `$_` zu (oder nutzt gleich eine `foreach`-Schleife).

```
@meinArray = (0..10);
map {$_ = sqrt} @meinArray;
```

Arrays ausgeben

Bevor Sie jetzt weiterlesen, überlegen Sie sich bitte selbst einen Weg, wie Sie die einzelnen Elemente eines Arrays auf die Konsole ausgeben können.

- ✘ Sie haben eine `for`-Schleife erzeugt. Nicht schlecht!
- ✘ Sie haben eine `foreach`-Schleife verwendet. Noch besser!
- ✘ Sie haben einfach `print "@ihrArray"` aufgerufen. Na, da haben Sie doch wohl gespitzt?

Tatsächlich besteht der einfachste Weg zur Ausgabe eines Arrays darin, die Array-Variable an `print` zu übergeben. `print` durchläuft dann automatisch die einzelnen Elemente im Array und gibt sie aus. Ein wenig nachteilig ist, dass dabei die einzelnen Elemente nicht getrennt werden. Dies kann man jedoch beheben, indem man das Array in doppelte Anführungszeichen setzt.

Furby

Mittlerweile haben wir eine Menge über Listen und Arrays gelernt. Mit diesem Vorwissen sollten wir in der Lage sein, unseren Furby besser zu verstehen.

Listing 5.2: furby1.pl

```
#!/usr/bin/perl -w

@sprich = ("Na, wie geht\'s?", "Lass mich schlafen!",
           "Ist es schon acht Uhr?", "Ich mag dich.",
           "Ein Pokemon.. den mach ich kalt.");

print "\n $sprich[rand @sprich] \n";
```

Wirklich neu ist in diesem Programm nur die Funktion `rand`, die zur Erzeugung von Zufallszahlen dient. Übergibt man `rand` einen Wert (oder Ausdruck), liefert die Funktion eine zufällige Gleitkommazahl zurück, die zwischen 0 und dem übergebenen Wert liegt.

In Listing 5.2 wird `rand` mit dem Ausdruck `@sprich` aufgerufen. Dabei liefert `@sprich` die Anzahl der Elemente im Array zurück (siehe Abschnitt »Anzahl der Elemente im Array«). `rand` erzeugt eine Zufallszahl zwischen 0 und `@sprich`. Dieser Wert wird als Index in dem Array verwendet (wobei die Zufallszahl zu einer Ganzzahl abgerundet wird), `$sprich[index]` liefert dann das zugehörige Element im Array zurück.

Wie zufällig sind Zufallszahlen?

Vom Computer generierte Zahlen sind nie wirklich zufällig (andererseits: was ist schon wirklich zufällig?). Der Computer versucht die Ziehung von Zufallszahlen dadurch zu simulieren, dass er beim ersten Aufruf von `rand` eine Folge von Zahlen generiert, in der niemand ein auch nur irgendwie geartetes Muster erkennen kann. Bei jedem Aufruf von `rand` liefert er aus dieser Zahlenfolge eine Zahl zurück.

Die Erzeugung der Zufallszahlen wird noch durch eine zweite Funktion gesteuert: `srand`. Wenn Sie am Anfang Ihres Programms `srand` mit einem konstanten Wert aufrufen (beispielsweise `srand 1;`), so wird bei jedem Aufruf des Programms die gleiche Folge von Zufallszahlen erzeugt. Dies kann insbesondere beim Aufspüren von Programmfehlern sehr hilfreich sein.

5.2.3 Elemente hinzufügen oder entfernen

Um Elemente an den Anfang oder das Ende eines Arrays anzuhängen oder Elemente vom Anfang oder Ende des Arrays zu entfernen, verwendet man die Funktionen `pop`, `push` und `unshift`, `shift`.

Funktion	Beschreibung

push	Hängt eines oder mehrere Elemente an das **Ende** eines Arrays an.
	`@array = (1, 2, 3);`
	`push(@array, 4);         # (1, 2, 3, 4)`
	`@array = (1, 2, 3);`
	`push(@array, 4..5);      # (1, 2, 3, 4, 5)`
	`@demo = (4, 5);`
	`@array = (1, 2, 3);`
	`push(@array, @demo);     # (1, 2, 3, 4, 5)`

unshift	Hängt eines oder mehrere Elemente an den **Anfang** eines Arrays an.
	`@array = (1, 2, 3);`
	`unshift(@array, 4..5);   # (4, 5, 1, 2, 3)`

pop	Entfernt das **letzte** Element eines Arrays und liefert es zurück.
	`@array = (1, 2, 3);`
	`$elem = pop @array;      # (1, 2)`
	`print $elem;             # Ausgabe: 3`

shift	Entfernt das **erste** Element eines Arrays und liefert es zurück.
	`@array = (1, 2, 3);`
	`$elem = shift @array;    # (2, 3)`
	`print $elem;             # Ausgabe: 1`

$#arrayName

Es wurde bereits kurz erwähnt, dass Perl den höchsten Index eines Arrays `meinArray` in einer Standardvariablen namens `$#meinArray` speichert. Über diesen Index können Sie auch die Größe des Array ändern. Wenn Sie den Index herabsetzen, schrumpft das Array entsprechen, wenn Sie den Index heraufsetzen, wächst das Array. Die neu hinzugekommenen Array-Elemente haben allerdings keinen Wert, sie sind undefiniert. Dies führt zu hässlichen Warnungen beim Ausführen des Skripts (vorausgesetzt der Schalter -w ist gesetzt). Sie können diese Meldungen verhindern, indem Sie mit Hilfe von `defined` prüfen, ob ein Array-Element definiert ist oder nicht.

```
@array = (1, 2);
$#array = 5;        # (1, 2, undefined, undefined, undefined)

foreach (@array) {
  if (defined $_) {
    $_ *= 2;
    print $_, " ";
    }
```

Ausgabe:

2 4

splice

Mit Hilfe der Funktion `splice` kann man Teile eines Arrays löschen oder durch andere Elemente ersetzen.

Um einen Teilbereich aus einem Array zu löschen, übergeben Sie `splice` das Array, den Index des ersten zu löschenden Elements und die Anzahl der zu löschenden Elemente.

```
@array = (0..9);
@ersetzt = splice(@array, 3, 2);
print "@array\n";                # (0, 1, 2, 5, 6, 7, 8, 9)
print "@ersetzt\n";              # (3, 4)
```

Wenn Sie den gelöschten Bereich durch eine beliebige Zahl anderer Elemente ersetzen wollen, übergeben Sie diese als letzten Parameter – beispielsweise:

```
@neu = (213, 78);
@array = (0..9);
@ersetzt = splice(@array, 3, 6, @neu);
print "@array\n";                # (0, 1, 2, 'a', 'b', 9)
print "@ersetzt\n";              # (3, 4, 5, 6, 7, 8)
```

5.2.4 Arrays sortieren und durchsuchen

Das Sortieren und Durchsuchen von Listen oder Arrays ist ein ganz typisches Programmierproblem, das sich nicht nur Perl-Programmierern stellt. Es gibt daher auch eine große Anzahl theoretischer, programmiersprachenunabhängiger Abhandlungen, die sich allein mit der Frage befassen, wie man eine gegebene Datenmenge effizient sortiert (oder durchsucht). Es wäre eine schändliche Missachtung dieser verdienstvollen Arbeiten, wenn ich Ihnen einfach nur mitteilen würde, dass man in Perl zum Sortieren die Funktion `sort` und zum Durchsuchen meist die Funktion `grep` oder einen regulären Ausdruck verwendet. Lassen Sie mich also ein wenig ausholen ...

Das Sortierproblem

Alles begann mit der Erfindung des Kartenspiels. Seitdem sieht man überall auf der Welt Menschen, die dabei sind, ihre Karten zu sortieren. Wenn man diesen Menschen über die Schulter blickt, fallen einem zwei Dinge auf:

- Es gibt verschiedene Arten der Sortierung: Manche sortieren aufsteigend (von der Sieben bis zum As), andere absteigend. Manche sortieren die Zehn zwischen Neun und Bube ein, andere zwischen König und As.
- Es gibt verschiedene Techniken, die unterschiedlich lange dauern. Ein gewiefter Skatspieler sortiert sein Handblatt in weniger als einer Minute, manche Patience-Spieler benötigen über eine Stunde, um gerade einmal 52 Karten in die gewünschte Reihenfolge zu bringen.

Eine dieser Techniken wollen wir jetzt genauer untersuchen. Dazu möchte ich Sie bitten, aus einem 52-blättrigen Kartenspiel[2] die Karten Kreuz 4 bis Kreuz 10 zu entnehmen, diese zu mischen oder – besser noch – in der folgenden Reihenfolge vor sich auszulegen.

♣5 ♣6 ♣9 ♣7 ♣4 ♣10 ♣8

Diese Karten sollen aufsteigend sortiert werden. Dazu gehen wir die Karten von vorne durch und vergleichen jede Karte mit der vorangehenden Karte, bis wir auf eine Karte treffen, deren Wert kleiner ist als der Wert der vorangehenden Karte. Dies ist das erste Mal bei der ♣7 der Fall. Wir könnten die ♣7 jetzt herausnehmen und weiter vorne einsortieren. Doch es geht uns ja nicht nur um das Sortieren von Karten. Wir wollen auch ein Sortierverfahren finden, das man nachprogrammieren kann. Sicherlich, mit Hilfe von `splice` kann man Elemente aus Arrays entfernen und an anderer Stelle

---

2 Übrigens: Vor einigen Jahren gab es auf der Royal Mile in Edinburgh einen kleinen Laden, in dem es fast ausschließlich Spielkarten – allerdings in allen erdenklichen Varianten und Ausstattungen – zu kaufen gab. Weiß jemand, ob es diesen Laden noch gibt?

wieder einfügen, doch ist das Löschen und Einfügen einzelner Elemente ein sehr zeitaufwendiges Verfahren. Die meisten Sortierverfahren verschieben die Elemente daher durch Tauschen. Statt Elemente zu löschen und einzufügen, weist man dabei einfach Element A den Wert von Element B und Element B den Wert von Element A zu:

```
@array = (5, 6, 9, 7, 4, 10, 8);

$temp = $array[2];          # 9 zwischenspeichern
$array[2] = $array[3];
$array[3] = $temp;
print "@array\n";           # (5, 6, 7, 9, 4, 10, 8)
```

Tauschen Sie jetzt also die Karten 9 und 7. Dann fahren wir mit dem Vergleichen der Karten bei der 9 fort. Das nächste Mal, dass die Karten nicht in der richtigen Folge stehen, ist bei der 4 und der 9 sowie der 10 und der 8 der Fall, die wir jeweils austauschen:

♣5 ♣6 ♣7 ♣4 ♣9 ♣8 ♣10

Unser Kartenspiel ist damit allerdings noch nicht korrekt sortiert. Also beginnen wir von vorne und tauschen wieder alle Pärchen, die nicht aufsteigend sortiert sind. (Im zweiten Durchgang sind dies die 4 und die 7 sowie 8 und 9, in den nachfolgenden Durchgängen wird nur noch die 4 langsam an ihre Position gerückt)

Im ungünstigsten Fall müssen Sie die Zahlenfolge sechsmal durchgehen, bis sie sortiert ist. Anders ausgedrückt: Nach diesem Sortierverfahren, das unter dem Namen BubbleSort bekannt ist, müssen Sie eine Folge von n Zahlen n–1 Male durchgehen, um zu garantieren, dass die Folge sortiert ist.

Effizienter ist ein Verfahren, das man Quicksort nennt (was nicht zu unrecht »schnelles Sortieren« bedeutet). Ausgangspunkt ist wieder folgende Kartenfolge:

♣5 ♣6 ♣9 ♣7 ♣4 ♣10 ♣8

Quicksort arbeitet ebenfalls mit dem Tauschen von Elementen, allerdings werden die zu tauschenden Elemente auf intelligentere Weise ausgewählt. Quicksort beginnt damit, die zu sortierende Folge in zwei Hälften zu teilen. Dazu wird das mittlere Element als Referenz ausgewählt (in unserem Falle die ♣7)[3]. Jetzt wird die Folge von vorne durchsucht, bis ein Element gefunden wird, das größer ist als das Referenzelement (9). Gleichzeitig wird die

3 Gibt es kein mittleres Element (Folgen mit gerader Zahl von Elementen), wählt man einfach ein Element, das ungefähr in der Mitte liegt.

Folge von hinten durchsucht, bis ein Element gefunden wird, das kleiner ist als das Referenzelement (4). Diese beiden Elemente werden ausgetauscht:

♣5 ♣6 ♣4 ♣7 ♣9 ♣10 ♣8

Danach werden die Folgen von vorne und hinten weiter durchsucht und die gefundenen Elemente paarweise ausgetauscht, bis man sich irgendwo trifft (was in unserem Fall zufällig direkt bei der 7 der Fall ist). Danach werden die beiden Hälften links und rechts des Treffpunkts für sich nach dem gleichen Verfahren sortiert.

♣5 ♣4 ♣6 ♣7 ♣9 ♣8 ♣10

Und so weiter, bis die Elemente sortiert sind.

♣4 ♣5 ♣6 ♣7 ♣9 ♣8 ♣10

Fazit

Sie haben jetzt zwei typische Sortierverfahren kennen gelernt und erfahren, dass die meisten gängigen Sortierverfahren darauf beruhen, dass Elemente paarweise getauscht werden.

Ob zwei Elemente getauscht werden, hängt von dem Ergebnis des Vergleichs der Elemente ab. Durch die Art des Vergleichs kann man steuern, in welcher Reihenfolge die Elemente sortiert werden. (In unserem BubbleSort-Beispiel haben wir getauscht, wenn Element A größer war als Element B. Auf diese Weise wurde das Array aufsteigend sortiert. Tauscht man, wenn A kleiner als B ist, wird das Array absteigend sortiert.)

Wie effizient ein Sortierverfahren arbeitet, hängt nicht davon ab, wie die Elemente paarweise verglichen werden, sondern davon, welche Elemente verglichen und getauscht werden.

Sortieren mit sort

Die vordefinierte Perl-Funktion `sort` sortiert nach dem oben vorgestellten Quicksort-Verfahren. Im einfachsten Fall muss man der Funktion dazu nur das zu sortierende Array übergeben und erhält ein neues Array mit den sortierten Elementen zurück:

```
@array = ('aa', 'FE', 'caa', 'ca', 'ab');
@sortiert = sort (@array);
print "@sortiert\n";
```

Ausgabe:

```
FE aa ab ca caa
```

Versucht man allerdings, auf diese Weise Zahlen zu sortieren, erhält man überraschende Ergebnisse.

```
@array = (34, 35, 36, 1100, 2, 5);
@sortiert = sort (@array);
print "@sortiert\n";
```

Ausgabe:

```
1100 2 34 35 36 5
```

Der Grund liegt nicht in dem Sortierverfahren, sondern in der Vergleichsfunktion. Standardmäßig sortiert `sort` nämlich lexikografisch, d.h. die Zahlen werden nicht als Zahlen, sondern als Ziffernfolgen interpretiert und wie Wörter verglichen. Daher ist jede Zahl, die mit einer 1 beginnt, kleiner als jede Zahl die mit Ziffern zwischen 2 bis 9 anfängt – unabhängig davon, wie viele Zifferstellen nachfolgen.

Um dieses Manko zu beheben, muss man der Funktion `sort` eine eigene Vergleichsfunktion übergeben. Diese Vergleichsfunktion muss zwei Elemente, A und B, vergleichen und

-1 zurückliefern, wenn Element A vor Element B einsortiert werden soll,

0 zurückliefern, wenn beide Elemente gleich sind

-1 zurückliefern, wenn Element A hinter Element B einsortiert werden soll.

Zudem muss die Funktion wissen, dass die `sort`-Funktion die zu vergleichenden Elemente in den Standardvariablen `$a` und `$b` ablegt. Die Vergleichsfunktion muss also `$a` und `$b` vergleichen und sollte sich hüten, diesen Variablen Werte zuzuweisen.

Für einfache Vergleiche muss man keine echte Vergleichsfunktion aufsetzen. Es genügt ein Ausdruck, der einen der speziellen Vergleichsoperatoren, `cmp` (für Strings) oder <=> (für Zahlen), verwendet, die extra zur Verwendung mit `sort` die Ergebniswerte -1, 0 und 1 zurückliefern.

Tabelle 5.1: Operatoren für Sortierungen

Operator	**Beschreibung**
str1 cmp str2	Lexikografischer Vergleich von Strings. Liefert -1, wenn str1 < str2 0, wenn str1 == str2 1, wenn str1 > str2

Tabelle 5.1: Operatoren für Sortierungen (Fortsetzung)

Operator	Beschreibung
z1 <=> z2	Numerischer Vergleich von Zahlen. Liefert -1, wenn z1 < z2 0, wenn z1 == z2 1, wenn z1 > z2

Wie im Falle der Funktion `map` (siehe Abschnitt »Arrays durchlaufen«) kann man die Vergleichsoperation als Ausdruck in geschweiften Klammern übergeben.

```
@sortiert = sort {$a cmp $b} @array;
```

Dieser Befehl sortiert das Array in der gleichen Weise wie `sort (@array)`, d.h. die `sort`-Funktion verwendet standardmäßig den Vergleich: `$a cmp $b`.

Will man Zahlen sortieren, braucht man nur `cmp` durch `<=>` zu ersetzen.

```
@array = (34, 35, 36, 1100, 2, 5);
@sortiert = sort {$a <=> $b} @array;
print "@sortiert\n";
```

Ausgabe:

```
2 5 34 35 36 1100
```

Na, das sieht doch schon besser aus! Und wenn man die Zahlen in absteigender Folge sortieren möchte? Dann braucht man nur die Vergleichsfunktion zu ändern. In unserem kleinen Beispiel genügt es, wenn man die Variablen `$a` und `$b` austauscht.

```
@sortiert = sort {$b <=> $a} @array;
print "@sortiert\n";
```

Ausgabe:

```
1100 36 35 34 5 2
```

Für komplexere Sortierungen muss man spezielle Funktionen aufsetzen. Das Aufsetzen von Funktionen ist zwar erst Thema des nachfolgenden Kapitels, dennoch möchte ich Ihnen an dieser Stelle schon einmal vorab zeigen, wie man Vergleichsfunktionen aufsetzt. Als einfaches Beispiel erzeugen wir eine Vergleichsfunktion, die Zahlen aufsteigend sortiert (also mit `$a <=> $b` äquivalent ist).

```
sub zahlen_vgl {
  return -1 if ($a < $b);
  return  0 if ($a == $b);
  return  1 if ($a > $b);
  }

@array = (34, 35, 36, 1100, 2, 5); print "@array\n";
@sortiert = sort zahlen_vgl @array;
```

Suchen

Im Vergleich zum Sortieren könnte man das Suchen direkt als einfach bezeichnen. Der einfachste Suchalgorithmus überhaupt besteht darin, die Elemente im Array von vorne nach hinten zu durchlaufen und mit dem gewünschten Suchkriterium zu vergleichen.

In Perl brauchen Sie dafür nicht einmal eine `for`- oder `foreach`-Schleife aufzusetzen. Sie können direkt auf die vordefinierte Funktion `grep` zurückgreifen.

```
@array = (0..100);
@ungerade = grep {$_ % 2} @array;
print "@ungerade\n";
```

Als erstes Argument übergeben Sie `grep` einen Ausdruck (oder eine Funktion) für das Suchkriterium. Dieser Ausdruck sollte »wahr« ergeben, wenn das überprüfte Element (das `grep` in der Standardvariablen `$_` zwischenspeichert) das Kriterium erfüllt. In obigem Beispiel liefert der Modulo-Operator den Rest der Division durch 2. Für alle ungeraden Zahlen beträgt dieser Rest 1, und da der numerische Wert 1 als »wahr« interpretiert wird, erfüllen alle ungeraden Zahlen das Kriterium. `grep` sammelt alle Elemente, auf die das Kriterium zutrifft, und gibt sie als neue Liste zurück. Obiges Beispiel gibt also alle ungeraden Zahlen zwischen 0 und 100 aus.

Komplexere Suchkriterien kann man mit Hilfe regulärer Ausdrücke (siehe Kapitel 10) oder durch Aufsetzen eigener Funktionen definieren. Das folgende Skript zum Beispiel durchsucht ein Array nach Primzahlen:

Listing 5.3: prim.pl

```
#!/usr/bin/perl -w

sub primzahlen_berechnen {
  $prim = 1;
  for ($i = $_-1; $i > 1; --$i)
    {
    if($_ % $i == 0) {
       $prim = 0;
       }
    }
  return $prim;
  }

@array = (0..10, 12, 13, 54, 77, 100..150);

@primzahlen = grep(primzahlen_berechnen, @array);

print "\nGefundene Primzahlen: \n@primzahlen\n";
```

Das Skript verwendet eine simple Brute-Force-Methode zur Ermittlung der Primzahlen (es versucht einfach, die Array-Elemente durch alle kleineren Zahlen zu teilen). Bestünde die Aufgabenstellung darin, alle Primzahlen bis zu einer bestimmten Zahl zu finden, würde man dazu einen intelligenteren Algorithmus verwenden (zumindest das Sieb des Eratosthenes).

Binäre Suche

Wenn man in großen sortierten Arrays gezielt nach dem einen oder anderen Wert sucht, ist das stupide Durchlaufen des Arrays vom ersten zum letzten Element unnötig zeitraubend.

Schneller geht es, wenn man den Suchraum bei jedem Vergleich halbiert. Dazu vergleicht man im ersten Schritt den gesuchten Wert mit dem mittigen Element des Arrays. Vermutlich werden beide nicht übereinstimmen, aber das macht nichts. Entscheidend ist, ob der gesuchte Wert kleiner oder größer als der mittlere Wert ist. Ist der gesuchte Wert kleiner und weiß man, dass das Array aufsteigend sortiert ist, kann man daraus folgern, dass das gesuchte Element in der ersten Hälfte des Arrays liegen muss. Man kann den Suchraum also halbieren. Im nächsten Schritt wählt man die Mitte aus dem halbierten Suchraum und prüft wieder, ob das gesuchte Element in der oberen oder unteren Hälfte des aktuellen Suchraums liegt. Auf diese Weise wird der Suchraum bei jedem Vergleich halbiert.

Listing 5.4: binsuche.pl

```
#!/usr/bin/perl -w

@array = (0..100);

$gesucht = 34;

$anf  = 0;
$ende = $#array;
while ($anf <= $ende)
  {
  $mitte = int (($anf+$ende)/2);

  if ($array[$mitte] == $gesucht) {
    print "gefunden: $array[$mitte]\n";
    last;
    }
    elsif ($array[$mitte] < $gesucht) {
      $anf = $mitte + 1;
      }
      else {
      $ende = $mitte - 1;
      }
  }
```

5.2.5 Arrays und Strings

Wissen Sie noch, wofür das Akronym »PERL« steht? Richtig, für »Practical Extraction and Report Language« – eine Sprache zum Extrahieren und Aufbereiten von Daten. Die in diesem Abschnitt vorgestellten Funktionen `split` und `join` werden Sie davon überzeugen, dass Perl seinen Namen zu Recht trägt.

Text in Arrays verwandeln

Mit Hilfe der Funktion `split` kann man einen String in Einzelteile aufbrechen und diese in einem Array ablegen. Voraussetzung ist, dass die Einzelteile im String durch ein geeignetes Trennzeichen auseinander gehalten werden, beispielsweise durch Leerzeichen oder Kommata.

So löst die folgende Anweisung einen String mit Zahlenwerten in ein Array von Zahlen auf:

```
$str = "1 2 3 4 5";
@array = split(" ", $str);
```

Sie können die Zahlen auch direkt aus Variablen entnehmen oder eine Kombination aus Zeichen als »Trennzeichen« verwenden:

```
$var1 = 10;
$var2 = 20;
$var3 = 30;
@array = split(", ", "$var1, $var2, $var3");
```

Und natürlich können Sie auch Textzeilen in einzelne Wörter auftrennen:

```
$str = "Practical Extraction and Report Language";
@woerter = split(" ", $str);
foreach (@woerter) {
  print "$_ \n";
  }
```

Ausgabe:

```
Practical
Extraction
and
Report
Language
```

»Nicht schlecht«, werden Sie sagen, »aber was kann ich damit anfangen?« Lesen Sie den nächsten Abschnitt.

CSV-Dateien verarbeiten

CSV-Dateien sind Dateien, in denen Tabellendaten gespeichert werden. Jede Zeile in einer CSV-Datei enthält eine Zeile der Tabelle. Innerhalb jeder Zeile sind die Daten der einzelnen Felder (Spalten) durch Kommata (gelegentlich auch Semikola) getrennt. CSV-Dateien wie die folgende kann man in einem einfachen Texteditor erstellen oder mit Hilfe eines speziellen Tabellenkalkulationsprogramms, das die Abspeicherung in einer CSV-Datei unterstützt (beispielsweise MS Excel).

Listing 5.5: Die Datei aktien.cvs

```
"Fondsname","WKN","Ausgabepreis","Rücknahmepreis",
"Rücknahmepreis Vortag"
"EM FernostFonds",973820,371.68,353.98,356.59
"EM LateinamerikaFonds",973819,317.49,302.37,302.43
"EM OsteuropaFonds",973821,559.2,532.57,533.71
"UniAsia",971267,28.85,27.48,27.57
"UniDynamicFonds: Europa",987194,69.14,66.48,66.65
```

Stellen Sie sich vor, Sie erhalten per E-Mail jeden Tag Aktiendaten in diesem Format. Jeden Tag kontrollieren Sie, wie sich der Rücknahmepreis Ihrer Fonds geändert hat, und jeden Tag ärgern Sie sich über das schwer lesbare Format. Da lohnt es sich, ein kleines Perl-Skript von gerade einmal sieben Zeilen aufzusetzen, das die Sie interessierenden Daten extrahiert und besser lesbar ausgibt.

Listing 5.6: aktien.pl

```
#!/usr/bin/perl -w

print "\n";
@daten = <STDIN>;

foreach (@daten) {
  @zeile = split(",",$_);
  printf(" %30s: %s\n",$zeile[0], $zeile[3]);
  }
```

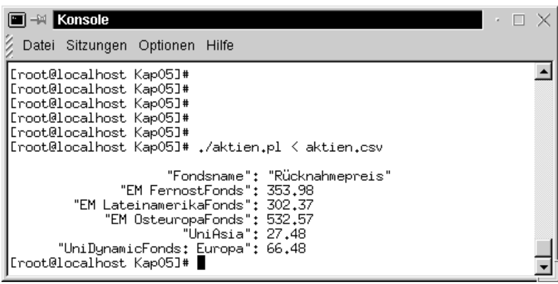


Abb. 5.2: Ausführung des Skripts aktien.pl

Analyse

Beginnen wir mit der Zeile `@daten = <STDIN>;`. Bisher haben wir die Daten, die über die Standardeingabe kommen, immer in eine skalare Variable eingelesen. Hier wird jedoch in ein Array eingelesen. In diesem Falle liest das Skript so lange eine Zeile nach der anderen ein, bis es auf ein Dateiendezeichen trifft (Strg D unter Linux, Strg Z unter Windows). Diese Technik wird häufig verwendet, wenn man Dateien über die Kommandozeile vermittels des Umleitsymbols an ein Skript übergeben will.

Wenn Sie das Skript also wie folgt aufrufen:

```
> perl aktien.pl < aktien.cvs             # unter Windows
# ./aktien.pl < aktien.cvs                # unter Linux
```

liest es den Inhalt der Datei AKTIEN.CVS zeilenweise ein und legt die einzelnen Zeilen im Array `@daten` ab.

Danach übergibt die `foreach`-Schleife die einzelnen Zeilen an die Funktion `split`. Diese trennt die jeweils aktuelle Zeile an den Kommata und speichert die einzelnen Felder im Array `@zeile`.

Die abschließende `printf`-Anweisung gibt für jede Zeile das erste und das vierte Element (Fondsname und Rücknahmepreis) in ansprechender Formatierung aus.

```
                "Fondsname": "Rücknahmepreis"
          "EM FernostFonds": 353.98
    "EM LateinamerikaFonds": 302.37
        "EM OsteuropaFonds": 532.57
                  "UniAsia": 27.48
  "UniDynamicFonds: Europa": 66.48
```

Falls Sie Dateien verarbeiten wollen, in denen die Felder nicht durch Kommata getrennt sind, ist dies selbstverständlich auch kein Problem. Sie brauchen nur das Trennzeichen aus dem `split`-Aufruf anzupassen.

Statt die von `split` zurückgelieferten Daten in einem Array zu speichern, können Sie die Daten auch in einer Liste von Skalaren ablegen. Falls Sie dabei bestimmte Felder überspringen möchten, verwenden Sie die Funktion `undef`.

```
foreach (@daten) {
  ($fonds,undef,undef,$ruecknahmepreis) = split(",",$_);
  printf("%30s: %s\n",$fonds, $ruecknahmepreis);
  }
```

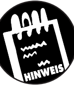
Listen in Text verwandeln

Das Gegenstück zu `split` ist die Funktion `join`, mit deren Hilfe Sie eine Liste oder ein Array in einen String umwandeln können. Dabei können Sie angeben, durch welches Zeichen (welche Zeichenkombination) die einzelnen Array-Elemente im String getrennt werden sollen.

```
@daten = (1..5, 10..15);
print join(" - ", @daten);
```

Ausgabe:

```
1 - 2 - 3 - 4 - 5 - 10 - 11 - 12 - 13 - 14 - 15
```



Mit `join` und `split` können Sie bequem formatierte Daten (wie sie z.B. in CVS-Dateien, Log-Dateien von Webservern oder Passwortdateien vorliegen) einlesen, bearbeiten und wieder im gleichen Format zurückschreiben.

5.3 Hashes

Hashes sind eine spezielle Form von Arrays, die in der Literatur häufig auch als »assoziative Arrays« bezeichnet werden. Was aber sind »assoziative Arrays«?

Bei einem normalen Array können Sie auf die einzelnen Elemente im Array über einen Zahlenindex zugreifen. Bei einem assoziativen Array greifen Sie über frei definierbare Strings auf die Array-Elemente zu, d.h. Sie assoziieren jedes Array-Element mit einem String. Diese Kombination von String und eigentlichem Element bezeichnet man auch als Schlüssel/Wert-Paar (im Englischen: key/value).

5.3.1 Hashes definieren

Ein kleines Beispiel wird Ihnen schnell klar machen, wozu man Hashes verwendet und wie man sie in Perl-Programmen erzeugt.

Nehmen wir an, Sie wollen in einem Perl-Skript Adressen verarbeiten. Eine Adresse ist im Grunde nichts anderes als eine Liste von bestimmten Daten: Vorname, Nachname, Straße, Hausnummer, PLZ, Stadt. Es spricht also erst einmal nichts dagegen, eine Adresse wie folgt zu definieren:

```
@adresse = ("Rip", "van Winkle", "Friedhofstrasse", 23,
            50000, "Gravesville");
```

Zwei Dinge sollten Sie an einer derartigen Konstruktion stören:

- ✘ Hier werden in einem Array Elemente unterschiedlicher Bedeutung gespeichert, die zusammen ein Element höherer Organisationsebene (die Adresse) bilden. Die meisten Programmierer erwarten aber, dass in einem Array nur gleichberechtigte Elemente (beispielsweise Messwerte) abgelegt sind.
- ✘ Der Zugriff über Zahlenindizes macht das Programm schwerer lesbar. Oder können Sie auf die Schnelle noch sagen, ob `$adresse[4]` die Hausnummer oder die Postleitzahl ist.

Hash-Variablen beginnen mit einem %-Zeichen

Als Ausweg bietet sich der Einsatz eines Hash an. Dabei schreibt man die Schlüssel und die Werte hintereinander in die Liste, mit der der Hash initialisiert wird. Als Schlüssel bietet es sich an, die Bezeichnungen der einzelnen Adressfelder zu verwenden. Außerdem ist zu beachten, dass Hash-Variablen mit einem %-Zeichen beginnen.

```
%adresse = ("Vorname", "Rip", "Nachname", "van Winkle",
            "Straße", "Friedhofstrasse", "Hausnummer", 23,
            "PLZ", 50000, "Stadt", "Gravesville");
```

So wie oben ist die Strukturierung der Liste allerdings nur schwer zu erkennen. Perl kennt daher noch eine alternative Syntax, in der die Schlüssel-Wert-Paare deutlicher hervortreten.

```
%adresse = ("Vorname"    => "Rip",
            "Nachname"   => "van Winkle",
            "Straße"     => "Friedhofstrasse",
            "Hausnummer" => 23,
            "PLZ"        => 50000,
            "Stadt"      => "Gravesville");
```

Als Belohnung für die zusätzliche Tipparbeit beim Aufsetzen der Initialisierungsliste können wir dann über die Schlüssel auf die Werte zugreifen.

```
print $adresse{Hausnummer};     # Hausnummer ausgeben
```

Als Werte sind beliebige Skalare erlaubt, die Schlüssel eines Hash müssen dagegen Strings sein, wobei man auf die Anführungszeichen verzichten kann – wenn man möchte.

5.3.2 Mit Hashes arbeiten

Auf einzelne Elemente zugreifen

Auf die einzelnen Elemente eines Arrays greift man über eine Indexangabe in eckigen Klammern zu.

```
@meinArray = (1, 5, 7);
print $meinArray[2];
```

Auf ein einzelnes Feld eines Hash greift man mit einer analogen Syntax zu, nur dass man statt eines Index in eckigen Klammern einen Schlüssel in geschweiften Klammern verwendet.

```
%meinHash = ("Vorname" => "Rip", "Nachname" => "van Winkle");
print $meinHash{Vorname};
```

Zur Erinnerung: Wie bei den Arrays ist zu beachten, dass der Zugriff über einen Schlüssel einen einzelnen skalaren Wert zurückliefert, weswegen dem Namen der Hash-Variablen ein $ vorangestellt werden muss.

Schlüssel/Wert-Paare hinzufügen

Nehmen wir an, Sie hätten folgendes Hash vorliegen:

```
%person = ("Vorname" => "Rip", "Nachname" => "van Winkle");
```

Dieses Hash möchten Sie jetzt um ein weiteres Feld zum Abspeichern des Alters der Person erweitern. Kein Problem! Denken Sie sich einfach einen passenden Schlüssel aus (beispielsweise `"Alter"`) und weisen Sie diesem einen neuen Wert zu:

```
$person{Alter} = 99;
```

Die Einrichtung eines neues Schlüssel/Wert-Paares sieht von der Syntax her also genauso aus wie das Zuweisen eines neuen Wertes an einen bestehenden Schlüssel. Wenn es den angegebenen Schlüssel noch nicht gibt, wird das Schlüssel/Wert-Paar neu eingerichtet, existiert der Schlüssel bereits, wird sein Wert überschrieben.

Ein Schlüssel – mehrere Werte

Da die Neudefinition eines Schlüssels automatisch als einfache Wertzuweisung interpretiert wird, wenn der Schlüssel bereits existiert, ist es nicht möglich, ein Hash mit mehreren, gleichnamigen Schlüsseln einzurichten.

Um dennoch unter einem Schlüssel mehrere Werte speichern zu können, muss man ein wenig tricksen. Eine Möglichkeit ist, den Schlüssel mit einer Referenz (siehe Kapitel 7) auf eine Liste zu assoziieren und die verschiedenen Werte zu dem Schlüssel in die Liste einzutragen. Eine andere Möglichkeit besteht darin, die oben angesprochene Liste durch einen String zu simulieren, in dem die einzelnen Werte durch ein bestimmtes Trennzeichen (beispielsweise »:«) auseinander gehalten werden.

Schlüssel/Wert-Paare löschen

Mit Hilfe der Funktion `delete` kann man einzelne Schlüssel/Wert-Paare löschen:

```
delete $person{Alter};
```

Wenn Sie den Schlüssel im Hash belassen und nur den Wert des Schlüssels löschen wollen, verwenden Sie die Funktion `undef`.

```
undef $person{Nachname};
```

Nicht vorhandene Schlüssel und undefinierte Werte können beim Zugriff Ärger und gegebenenfalls unschöne Fehlermeldungen hervorrufen. Mit Hilfe der Funktionen `exists` und `defined` kann man sich vor solchen Missgriffen schützen.

```
if (exists($person{Nachname}))      # existiert Schlüssel?
  {
  if (defined($person{Nachname}))   # Wert definiert
     {
     print $person{Nachname};
     }
  }
```

Keine Angst. In der Praxis kann man auf solche Überprüfungen häufig verzichten.

Ganzen Hash löschen

Und was macht man, wenn man einen kompletten Hash löschen will? Nun man geht die Felder im Hash durch und löscht sie einzeln. Nein, das war nur ein Scherz. Schreiben Sie einfach:

```
%hash = ();
```

oder:

```
undef %hash;
```

Felder eines Hash durchlaufen und ausgeben

Ein typisches Problem bei der Programmierung mit Hashes ist das Durchlaufen des Hash. Wie so häufig in Perl gibt es auch zu diesem Problem mehrere mögliche Lösungen. Einige davon werde ich Ihnen jetzt vorstellen.

Nehmen wir an, Sie haben folgendes Hash definiert, das Sie ausgeben wollen:

```
%person = ("Vorname" => "Rip", "Nachname" => "van Winkle",
           "Alter" => 99);
```

Der erste Gedanke ist, das Hash in einer Schleife Feld für Feld zu durchlaufen und die Werte auszugeben. Die Idee ist auch zweifelsohne richtig, doch wenn Sie jetzt darangehen, eine entsprechende Schleife aufzusetzen, stoßen Sie auf ein Problem: Die Schlüssel eines Hash kann man nicht durch Inkrementierung wie die Indizes eines Arrays durchlaufen. Unter Umständen, beispielsweise wenn Sie ein Hash aus den Daten einer Datei aufbauen (siehe Beispiel aus Kapitel 14), wissen Sie vermutlich weder wie die Schlüssel heißen noch wie viele Schlüssel es insgesamt sind.

Ohne Hilfe kommen wir hier nicht weiter. Was wir benötigen ist eine vordefinierte Perl-Funktion, die für uns die Schlüssel des Hash ermittelt. Diese Funktion heißt `keys`.

keys

Die Funktion `keys` liefert die Schlüssel eines Hash in Form einer Liste zurück. Die Liste kann man dann wie gewohnt in einer `foreach`-Schleife durchlaufen.

```
%person = ("Vorname" => "Rip", "Nachname" => "van Winkle",
           "Alter" => 99);

foreach $schluessel[4] (keys %person) {
   printf("%15s:  %s\n", $schluessel, $person{$schluessel});
   }
```

Der Aufruf `keys %person` erzeugt eine Liste der Schlüssel aus dem `%person`-Hash. Diese Liste wird von der `foreach`-Schleife durchlaufen, wobei der jeweils aktuelle Schlüssel in der Variablen `$schluessel` abgelegt wird. In der `printf`-Anweisung werden der Schlüssel und der zugehörige Wert ausgegeben.

Ausgabe:

```
       Nachname:  van Winkle
          Alter:  99
        Vorname:  Rip
```

Was an dieser Ausgabe auffällt, ist, dass die Schlüssel/Wert-Paare nicht in der gleichen Reihenfolge ausgegeben werden, in der sie definiert wurden. Dies hat mit der Art und Weise zu tun, in der Hashes intern im Speicher abgelegt werden. Wenn Sie die Erstellungsreihenfolge beibehalten wollen, müssen Sie auf das CPAN-Modul `Tie::IxHash` zurückgreifen.

Wenn Sie möchten, können Sie die Ausgabe auch nach den Schlüsseln oder Werten sortieren:

```
# Sortierung nach Schlüsseln
foreach $schluessel (sort keys %person) {
   printf("%15s:  %s\n", $schluessel, $person{$schluessel});
   }
```

Ausgabe:

```
          Alter:  99
       Nachname:  van Winkle
        Vorname:  Rip
```

4 Die meisten Programmierer nennen diese Variable `$keys` – was den Code wegen der Ähnlichkeit zur Funktion `keys` für Anfänger allerdings unnötig kompliziert.

Die Sortierung nach Werten erfordert allerdings, dass man eine eigene Vergleichsfunktion aufsetzt.

```
# Sortierung nach Werten
foreach $schluessel
       (sort { $person{$a} cmp $person{$b} } keys %person)
   {
   printf("%15s:  %s\n", $schluessel, $person{$schluessel});
   }
```

Ausgabe:

```
          Alter:  99
        Vorname:  Rip
       Nachname:  van Winkle
```

values Wenn Sie nur an den Werten eines Hash interessiert sind, beispielsweise nur die Werte des Hash ausgeben wollen, brauchen Sie nicht den Umweg über die Schlüssel zu gehen. Mit Hilfe der Funktion `values` können Sie sich direkt eine Liste der Werte zurückliefern lassen.

```
foreach $wert (values %person) {
   print "$wert, ";
   }
```

Oder einfacher:

```
print join(' - ', values %person);
```

Neben `keys` und `values` kann man zum Durchlaufen von Hashes auch die Funktion `each` verwenden, die bei jedem Aufruf das jeweils nächste Schlüssel/Wert-Paar zurückliefert:

```
while (($schluessel, $wert) = each %person) {...}
```

5.3.3 Wörter zählen

Hashes werden meist zur Repräsentation strukturierter Daten – Adressen, Mitarbeiterdaten etc. – benutzt. Sie eignen sich aber auch hervorragend für Häufigkeitsbestimmungen.

Listing 5.7: woerter.pl

```
#!/usr/bin/perl -w

%woerterHash = ();

print "\n";
@zeilen = <STDIN>;
```

```
foreach (@zeilen) {
  chomp($_);
  @woerter = split(' ',$_);

  foreach $wort (@woerter) {
     $woerterHash{lc($wort)}++;
     }
  }

foreach $key (sort keys %woerterHash) {
  printf("%15s : %s\n", $key, $woerterHash{$key});
  }
```

Wenn Sie dieses Skript aufrufen, übergeben Sie ihm mit Hilfe des Umleitsymbols den Inhalt einer Textdatei.

```
> perl woerter.pl < john_maynard.txt        # unter Windows
% ./woerter.pl < john_maynard.txt           # unter Linux
```

Wenn Sie möchten, können Sie die Ausgabe des Skripts in eine Datei umleiten.

```
> perl woerter.pl < john_maynard.txt  > ausgabe.txt
% ./woerter.pl < john_maynard.txt  > ausgabe.txt
```

Das Skript liest die Datei zeilenweise in das Array `@zeilen` ein. Dann werden die einzelnen Zeilen anhand der Leerzeichen in Wörter aufgetrennt. Für jedes Wort wird ein Schlüssel/Wert-Paar angelegt. Die zugehörige Anweisung

```
$woerterHash{lc($wort)}++;
```

ist im wahrsten Sinne des Wortes multifunktional. Wenn das Wort in `$wort` das erste Mal auftaucht, wird das Hash `%woerterHash` um einen entsprechenden Schlüssel erweitert. Der Wert des Schlüssels wird durch die Inkrementanweisung heraufgesetzt und hat danach den Wert 1. Trifft das Skript bei seiner Analyse auf ein Wort, das schon einmal vorgekommen ist, gibt es bereits einen entsprechenden Schlüssel, und es wird lediglich der Wert zu diesem Schlüssel erhöht. Sie sehen: die Anweisung zählt die Vorkommen der einzelnen Wörter und baut dabei ganz automatisch das zugehörige Hash auf. (Der Funktionsaufruf `lc($wort)` sorgt übrigens dafür, dass das Skript nicht zwischen Klein- und Großschreibung unterscheidet.)

Eine etwas verbesserte Version dieses Skripts werden Sie in Übung 7 aus Kapitel 10 vorgestellt bekommen, wenn Sie wissen, wie man mit regulären Ausdrücken arbeitet.

5.4 Vertiefung: kombinierte Datenstrukturen

Listen können lediglich skalare Elemente enthalten. Folglich können auch höhere Datenstrukturen wie Arrays und Hashes nur aus skalaren Elementen aufgebaut werden – eine herbe Einschränkung!

Stellen Sie sich vor, Sie möchten eine kleine Datenbankanwendung schreiben, mit der Sie Ihre CDs verwalten. Zu jeder CD möchten Sie den Komponisten (bzw. die Gruppe), den Titel und die Kategorie (Klassik, Rock, Jazz) speichern. Was liegt also näher, als die einzelnen CDs in Form von Hashes zu repräsentieren:

```
%cd = ("composer"  => "Guiseppe Verdi",
       "titel"     => "La Traviata",
       "kategorie" => "Klassik");
```

In der Datenbank sollen aber Dutzende von CDs verwaltet werden. Was liegt da näher, als die CD-Hashes in einer Liste zu organisieren – also eine Liste mit Hashes als Elementen. Tja, nur ist dies ja nicht möglich, denn Hashes sind definitiv keine skalaren Daten. Da kann man wohl nichts machen.

Kann man doch! Seit Perl 5 gibt es in Perl einen zusätzlichen skalaren Datentyp: die Referenzen (auch Zeiger genannt). Im Detail werden wir uns diesem Datentyp erst in Kapitel 7 zuwenden, doch für den Aufbau komplexer Datenstrukturen sind die Referenzen einfach zu wichtig, als dass man sie in diesem Kontext unterschlagen könnte.

Referenzen sind letzten Endes nichts anderes als Skalare, die als Werte Speicheradressen beinhalten. Erzeugt man eine Referenz auf ein Hash:

```
$cd_ref = \%cd;
```

legt man im Skalar `$cd_ref` die Speicheradresse des Hash `%cd` ab. Über diese Speicheradresse, sprich über die Referenz, kann man später jederzeit wieder auf das Hash und seine Daten zugreifen. Wenn man eine Referenz auf ein Hash (oder ein Array oder einen Skalar) hat, ist das also genauso gut, wie wenn man die Hash-Variable selbst zur Verfügung hätte – man muss nur beachten, dass eine etwas andere Syntax für den Zugriff auf die Daten im Hash erforderlich ist.

Da eine Referenz selbst ein skalarer Wert ist, kann man sie in ein Array eintragen:

```
@cds = ();
push(@cds, \%cd);
```

Auf diesem Wege kann man die CD-Sammlung als Liste von Hash-Referenzen erzeugen. Wie dies im Detail aussieht, werden Sie in Kapitel 14 sehen.

5.5 Fragen und Übungen

1. Werden Listen mit runden oder eckigen Klammern definiert?
2. Definieren Sie möglichst effizient eine Liste für folgende Zahlen: -3, -2, -1, 0, 1, 2, 3, 33, 34, 35, 36, 37, 38, 39, 40.
3. Mit welchem Präfix beginnen Array-Variablen?
4. Wie lauten die Indizes des ersten und des letzten Elements des folgenden Arrays?

   ```
   @meinArray = (1..100);
   ```
5. Wie kann man sich den höchsten Index eines Arrays und wie die Anzahl der Elemente in einem Array zurückliefern lassen?
6. Simulieren Sie die Funktionen `shift`, `unshift`, `pop`, `push` durch `splice`-Aufrufe.
7. Was passiert, wenn Sie den Wert eines Array-Elements aus Versehen mit `@meinArray[5]` statt mit `$meinArray[5]` zurückliefern lassen?
8. Wie kann man zwei Arrays aneinander hängen?
9. Setzen Sie eine Vergleichsfunktion zum Sortieren von Spielkarten auf.
10. Mit welchem Präfix beginnen Hash-Variablen?
11. Schreiben Sie ein Programm, das eine Adresse von Tastatur einliest, in einem Hash speichert und dann ausgibt.
12. Schreiben Sie ein Programm, das Wochentage mit Hilfe eines Hash in Zahlen umwandelt.
13. Wozu dienen die Funktionen `keys` und `values`?

Kapitel 6

Funktionen

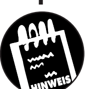
Die ganze Zeit schon haben wir Funktionen wie `print`, `chomp` u.a. verwendet, ohne uns wirklich Rechenschaft darüber abzulegen, was Funktionen eigentlich sind und wie Funktionen »funktionieren«. Dies werden wir nun nachholen. Und da man am meisten lernt, wenn man etwas selber macht, werden wir uns in diesem Kapitel anschauen, wie man eigene Funktionen erstellt.

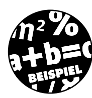
Im Einzelnen lernen Sie,

- ✘ was Funktionen sind und wofür man sie braucht
- ✘ wie man Funktionen definiert
- ✘ wie man Funktionen aufruft
- ✘ wie man Funktionen Werte übergeben kann
- ✘ wie man allgemein nützliche Funktionen implementiert
- ✘ wie Funktionen Werte zurückliefern können

Im Vertiefungsabschnitt erfahren Sie,

- ✘ wie man mit Hilfe von `my`, `vars` und `strict` Fehler durch automatisch definierte Variablen unterbindet
- ✘ wie man mit Packages arbeitet und was qualifizierte Bezeichner sind
- ✘ wie man eigene Module implementieren kann





Folgende Elemente lernen Sie kennen

Funktionen, die Schlüsselwörter `sub`, `return`, `my`, `vars` und `local`, die Standardvariable @_.

6.1 Wofür braucht man Funktionen?

Klären wir zuerst einmal, was Funktionen sind. Im Grunde genommen sind Funktionen nichts anderes als mit Namen versehene Anweisungsblöcke.

```
#!/usr/bin/perl -w

sub MeineFunktion
  {
  print "  Hallo aus meiner Funktion \n\n";
  }
...
```

Obiger Codeauszug definiert eine Funktion namens `MeineFunktion`. Dank dieser Definition verbindet der Perl-Interpreter den Namen `MeineFunktion` untrennbar mit dem darunter angegebenen Anweisungsblock.

Will man jetzt den Anweisungsblock der Funktion irgendwo im Skript ausführen lassen, braucht man nur den Funktionsnamen anzugeben. Jedes Vorkommen des Funktionsnamens interpretiert der Perl-Interpreter als einen Funktionsaufruf, d.h. er führt den Anweisungsblock der Funktion aus.

```
...
print "Rufe MeineFunktion auf: \n";
MeineFunktion();                            # Funktionsaufruf
```

Findige Leser (oder sollte ich sagen faule, aber meist sind ja gerade die Faulen besonders findig) werden den Wert dieser Benennung von Anweisungsblöcken sofort erfassen: Wenn ein Anweisungsblock mehrfach im Skript auftaucht, kann man sich Tipparbeit ersparen, indem man den Anweisungsblock nur einmal in einer Funktionsdefinition aufsetzt und danach überall dort, wo man den Code ausführen lassen möchte, nur noch den Funktionsnamen setzt. Das erspart einem Tipparbeit und schont Fingerkuppen und Sehnen.

Vorteile von Funktionen

Neben der reinen Tipperspa rnis gibt es aber noch eine Reihe weiterer positiver Effekte:

Wiederverwendung bestehenden Codes. Dass dies ein Vorteil ist, haben wir gerade gehört. Doch welche Anweisungen eignen sich für die Auslagerung in eine Funktion, welche Anweisungen wiederholen sich? Natürlich

kann man in einem Skript von 120 Zeilen nicht hingehen und die Zeilen 2 bis 22 einfach willkürlich herausnehmen und in eine Funktion umwandeln. Das wäre Blö..., äh ... sinnlos. Es gibt in größeren Skripten aber fast immer Anweisungen, die zwanglos zusammengehören, weil sie zusammen eine konkrete Aufgabe lösen – beispielsweise Daten aus einer Datenbank einlesen, eine komplizierte Berechnung durchführen (wie zum Beispiel die vordefinierte Perl-Funktion `sin` oder die Funktion `fakultaet`, die wir in Abschnitt 7.5 erstellen) oder die Elemente eines Arrays oder Hash ausgeben (siehe Abschnitt 7.3). Kurz gesagt: eine Funktion sollte eine konkrete, nicht übermäßig komplexe Aufgabe erfüllen.

Hat man nun ein Teilproblem identifiziert und in Form einer Funktion implementiert, ergeben sich daraus gleich mehrere Vorteile.

- ✘ Man erspart sich unnötige Tipparbeit, weil man überall, wo die betreffende Aufgabe zu erledigen ist, nur noch die Funktion aufrufen muss.
- ✘ Das Skript ist leichter zu debuggen. Stellen Sie sich vor, Sie verwenden in Ihrem Skript ein Array, dessen Inhalt Sie an mehreren Stellen im Skript ausgeben. Da der Code zur Ausgabe des Inhalts immer der gleiche ist, haben Sie die Anweisungen nur einmal aufgesetzt und dann kopiert (was einem ebenfalls Tipparbeit erspart). Beim ersten Ausführen des Skripts entdecken Sie dann, dass Sie bei der Ausgabe einen Fehler gemacht haben. Jetzt müssen Sie alle Stellen, an die Sie den fehlerhaften Code zur Ausgabe des Array kopiert haben, aufsuchen und einzeln korrigieren. Hätten Sie den Code als Funktion implementiert, müssten Sie den Code nur ein einziges Mal (in der Definition der Funktion) korrigieren.
- ✘ Man kann die Funktion in anderen Perl-Programmen wiederverwenden. Wenn Sie eine Funktion aufgesetzt haben, die ein Problem löst, das auch in anderen Perl-Skripten auftaucht, brauchen Sie das Rad nicht ständig neu zu erfinden (sprich das Problem nicht ständig neu lösen). Kopieren Sie die Funktion einfach in die Skripten, in denen Sie die Funktion benötigen, oder erstellen Sie ein eigenes Perl-Modul, in dem Sie Ihre Perl-Funktionen ablegen. Dann brauchen Sie die Funktionen nicht einmal von Skript zu Skript zu kopieren, sondern nur mit `use` das Modul einzubinden – so wie Sie auch die Funktionen der CPAN-Module einbinden (siehe Anhang B).

Organisation des Quelltextes. Ein anderer positiver Effekt der Funktionen ist, dass der Quelltext durch Auslagerung von Codeblöcken in Funktionen übersichtlicher organisiert werden kann – ein Effekt, der sich selbst dann einstellt, wenn die Funktionen nicht mehrfach aufgerufen werden müssen.

Stellen Sie sich nur einmal vor, Sie sollen ein bestehendes Perl-Skript überarbeiten. Das Skript liest eine Reihe von Daten aus einer Datei ein, bearbeitet diese und gibt die geänderten Daten aus. Der Code zum Einlesen der Daten aus der Datei muss angepasst werden, da sich das Format der Eingabedatei geändert hat. Leider ist das Skript schlecht kommentiert und besteht aus einer schwer zu durchblickenden Aneinanderreihung von ca. 100 Zeilen. Würden Sie sich da nicht auch wünschen, dass das Skript wie folgt aufgebaut wäre?

```
#!/usr/bin/perl -w

# Beginn des Programms

daten_einlesen();
daten_bearbeiten();
daten_ausgeben();

# Ende des Programms

sub daten_einlesen
  {
  ...
  }

sub daten_bearbeiten
  {
  ...
  }

sub daten_ausgeben
  {
  ...
  }
```

In einem solchermaßen aufgebauten Skript findet man sich schnell zurecht. Und wenn man den Code zum Einlesen der Daten sucht, braucht man nur in den Code der Funktion `daten_einlesen` zu springen. (Ein Beispiel für die Codeorganisation mit Hilfe von Funktionen sehen Sie in Kapitel 14).

Anpassung bestehender Funktionen. Schließlich kann man die Arbeitsweise bestimmter vordefinierter Perl-Funktionen (beispielsweise `sort` und `map`) durch Übergabe eigener Funktionen als Parameter steuern und anpassen (siehe Abschnitt 7.2.1).

Sie sehen, es spricht einiges dafür, dass wir uns einmal anschauen, wie man eigene Funktionen erstellen kann.

6.2 Funktionen definieren und aufrufen

6.2.1 Funktionen definieren

Funktionen kann man praktisch an jeder beliebigen Stelle eines Skripts definieren (natürlich nicht innerhalb von Kommentaren). Der Interpreter erkennt die Funktionsdefinition automatisch an dem Schlüsselwort sub (für Subroutine), mit der jede Funktionsdefinition einzuleiten ist.

Manche Programmierer sprechen in Anlehnung an das Schlüsselwort sub auch von Subroutinen statt von Funktionen, andere verstehen unter Funktionen nur die vordefinierten Perl-Funktionen und bezeichnen selbst definierte Funktionen als Subroutinen.

Eine typische Funktionsdefinition besteht also aus dem Schlüsselwort sub, dem Funktionsnamen und dem zugehörigen Anweisungsblock:

```
sub Funktionsname {
   # Anweisungen
   }
```

Ein simples Beispiel für Definition und Aufruf einfacher Funktionen haben Sie ja bereits in Abschnitt 7.1 gesehen. Ein weiteres Beispiel soll noch einmal die Definition von Funktionen zur Steuerung von Perl-Funktionen wie map und sort (siehe Kapitel 5.2.2 und 5.2.4) demonstrieren.

```
#!/usr/bin/perl -w

@liste = (100, 3000, 89.90, 125.40);

sub mwst { $_ * 0.16; }              # Funktionsdefinition
@mwst_liste = map(mwst, @liste);     # Übergabe als Argument

print "@mwst_liste\n";
```

Zur Erinnerung: Die Funktion map geht die Elemente des übergebenen Arrays einzeln durch und speichert sie in der Standardvariablen $_. Übergibt man map als erstes Argument eine Funktion, ruft map diese Funktion für jedes Element im Array auf. Indem man eigene Funktionen aufsetzt, die mit der Standardvariable $_ arbeiten, kann man selbst vorgeben, welche Operationen auf den Elementen des Arrays ausgeführt werden sollen. In obigem

Beispiel wird dies genutzt, um für jedes Element (wir gehen einmal davon aus, dass es sich um Preise handelt) die Mehrwertsteuer zu berechnen. Zum Schluss liefert `map` die berechneten Werte als neue Liste zurück:

```
16 480 14.384 20.064
```

Wenn Sie in der Funktion `mwst` den Wert von `$_` ändern (statt ihn nur zu verwenden), ändern Sie auch die Werte im Array `@liste`!

6.2.2 Funktionen aufrufen

Der Aufruf einer Funktion geschieht immer über den Funktionsnamen. Diesen kann man aber je nach Gusto verschieden ausschmücken:

```
meineFunktion();                # zu empfehlen
&meineFunktion;
&meineFunktion();
meineFunktion;
```

Grundsätzlich sind alle vier Aufrufformen gleichwertig. Lediglich bei der letzten Variante ist zu beachten, dass der Aufruf nur dann erfolgreich ist, wenn die Funktionsdefinition im Skript vor dem Aufruf steht (bei den anderen Aufrufformen können die Funktionen auch am Ende des Skripts definiert werden).

Die Varianten mit dem Präfix & stellen die traditionelle Art des Funktionsaufrufs dar, die heute aber mehr und mehr durch die moderne, stärker an C angelehnte, erste Aufrufform abgelöst wird.

6.3 Funktionen mit Parametern

Als ich oben in Abschnitt 7.1 etwas forsch postuliert habe, dass Funktionen nicht anderes als benannte Anweisungsblöcke seien, war das zugegebenermaßen stark vereinfacht. Funktionen sind zwar benannte Anweisungsblöcke, das ist nach wie vor wahr, doch verbindet sich mit einer Funktionsdefinition weit mehr als nur die Benennung eines Anweisungsblocks. Funktionen können nämlich mit dem Code, der sie aufruft, kommunizieren, indem Sie Daten aus dem Aufruf übernehmen und selbst Daten zurückliefern. Sonderlich verwundern dürfte Sie dies nicht, schließlich haben wir schon die ganze Zeit Gebrauch davon gemacht. Beispielsweise wenn wir der vordefinierten Funktion `print` den auszugebenden String übergeben haben (`print`

"Mein Text\n";) oder wenn die Funktion map uns ein neues, bearbeitetes Array zurückgeliefert hat (@neuesArray = map {$_ * 2 } @meinArray;).

Wie wichtig es ist, dass Funktionen Daten aus dem Aufruf übernehmen können, wollen wir uns an einem kurzen Beispiel klar machen.

6.3.1 Allgemein verwendbare Funktionen

Was früher die Götter in Weiß waren, sind heute die Turnschuh-Pioniere im E-Commerce. Nicht mehr die Macht über Leben und Tod ist es, was in der Gesellschaft Anerkennung und Ehrfurcht findet, sondern die erste Million mit 24 und die Macht über das Wohl und Wehe der Aktionäre. Stellen Sie sich also vor, sie wären ein junger, dynamischer Firmengründer. Ihre Firma entwickelt sich gut, und Sie haben immer mehr feste Kunden. Um den Überblick zu behalten, legen Sie eine Kundendatei an und schreiben ein paar kleine Perl-Skripten zur Bearbeitung und Abfrage der Kundendaten.

Eines dieser Skripten haben Sie so konzipiert, dass die Daten des aktuell bearbeiteten Kunden in ein Hash namens %kunde eingelesen werden:

```
%kunde = ("Vorname" => "Rudolf",
          "Nachname" => "Ramge",
          "Strasse" => "Pflaumenweg",
          "Hausnummer" => 10,
          "PLZ" => 85123,
          "Stadt" => "Ottobrunn",
          "KundenNr" => 10023,
         );
```

Beim Aufsetzen des Skripts merken Sie, dass Sie relativ häufig den Namen des Kunden in Kombination mit der Kundennummer ausgeben[1]. Also schreiben Sie eine Funktion, die genau dies tut:

```
sub kundenname {
  print "$kunde{Vorname} $kunde{Nachname} " ,
        "(KundenNr.: $kunde{KundenNr})";
  }
```

1 Ich muss gestehen, dass ich über wenig Erfahrung mit Software zur Kundenverwaltung verfüge. Ob eine Funktion zur Ausgabe des Namens und der Kundennummer in einem solchen Programm Sinn hat oder nicht, ist aber für das Beispiel vollkommen unwichtig. Es geht uns hier ja nicht darum, was die Funktion macht, sondern wie sie es macht!

Diese Funktion arbeitet ganz wunderbar in Skripten, in denen die Kundendaten in einem Hash namens %kunde abgespeichert sind. Wenn die Kundendaten aber zufällig einmal in einem zweiten Hash namens %meinKunde stehen, versagt die Funktion. Das Problem liegt darin, dass die Funktion voraussetzt, dass die Daten, die sie bearbeitet, in einer bestimmten Variablen stehen. Eine gute Funktion sollte so etwas nicht tun, denn es erschwert die allgemeine Einssetzbarkeit der Funktion. Wesentlich besser wäre es, wenn die Funktion auf beliebigen Hashes mit Kundendaten operieren könnte. Dies erreicht man, indem man die Funktion so implementiert, dass sie das Hash mit den Kundendaten beim Aufruf übernimmt:

```
# Aufruf mit Übergabe eines Arguments
kundenname(%kunde);
kundenname(%meinKunde);
```

Die Frage ist jetzt allerdings, wie die Funktion das übergebene Argument entgegennehmen kann?

6.3.2 Argumente und Parameter

Funktionen können über @_ (bzw. $_[0], _[1], etc.) auf die ihnen übergebenen Werte zugreifen

Perl verwendet für die Übergabe von Argumenten an Funktionen eine weitere Standardvariable: @_. Wird eine Funktion aufgerufen, übernimmt Perl die übergebenen Argumente und stellt sie via @_ der aufgerufenen Funktion zur Verfügung. In der Implementierung der Funktion kann man via $_[0], $_[1][2] etc. auf die einzelnen Elemente des Arrays @_, sprich auf die übergebenen Argumente, zugreifen.

$_[0], $_[1] etc. werde ich im weiteren Verlauf als Parameter der Funktion bezeichnen und von den Argumenten unterscheiden. Beim Aufsetzen des Funktionscodes arbeiten wir mit den Parametern. Wir wissen beim Schreiben der Funktion nicht, welche Werte die Parameter später beinhalten werden, aber wir wissen, was wir mit diesen Werten machen wollen. Innerhalb der Funktion arbeiten wir also nur mit den Parametern. Wenn die Funktion aufgerufen und ausgeführt wird, werden den Parametern konkrete Werte zugewiesen – die Argumente. Bei Ausführung der Funktion stehen in den Parametern die übergebenen Argumente.

---

2 Zur Erinnerung: Die einzelnen Elemente eines Array @array werden über das Skalarpräfix angesprochen: $array[0], $array[1] etc.

Viele Programmierer machen sich nicht die Mühe, zwischen Argumenten und Parametern zu unterscheiden. Sie sprechen entweder nur von Argumenten oder nur von Parametern oder verwenden beide Begriffe synonym. Solange allen Beteiligen klar ist, worüber man spricht, ist diese Verwischung der Begriffe sicherlich nicht schlimm. Programmieranfänger tun allerdings gut daran, sich den Unterschied zwischen Parametern und Argumenten erst einmal im Kopf ganz klar zu machen. Die begriffliche Unterscheidung kann dazu beitragen.

Programmierer, die bereits in C oder C++ programmiert haben, werden sich vermutlich fragen, ob es nicht in Perl, wie auch in C, eine Möglichkeit gibt, die Parameter der Funktion als eigene Variablen im Funktionskopf zu definieren. Nein, diese Möglichkeit gibt es nicht; Perl verwendet immer @_.

Argumente in der Funktion entgegennehmen

Schauen wir uns an, wie eine Funktion Argumente aus ihrem Aufruf entgegennimmt:

```
sub demoFunk {
  print "Erster Parameter : $_[0]\n";
  print "Zweiter Parameter: $_[1]\n";
  }
```

Diese Funktion erwartet, dass man ihr zwei skalare Werte als Argumente übergibt. Wenn dies geschieht, sollten die Werte in den Parametern `$_[0]` und `$_[1]` stehen. Um uns zu vergewissern, dass dem so ist, gibt die Funktion die Inhalte der Parameter `$_[0]` und `$_[1]` aus.

Wenn die Funktion danach wie folgt aufgerufen wird:

```
$var = 222;
demoFunk(111, $var);[3]
```

erzeugt sie folgende Ausgabe:

```
Erster Parameter : 111
Zweiter Parameter: 222
```

3 Wenn es eindeutig ist, welche nachfolgend aufgelisteten Werte Argumente für den Funktionsaufruf sind, und die Funktionsdefinition im Skript vor dem Funktionsaufruf steht, kann man beim Aufruf die Klammern weglassen: `demoFunk 111, $var;`

Prima, so haben wir uns das gewünscht! Jetzt sollte es doch auch möglich sein, das Kunden-Hash aus unserem einführenden Beispiel als Argument an die Funktion zu übergeben.

6.3.3 Listen, Arrays und Hashes als Argumente

Selbstverständlich können Sie auch Listen oder Listenvariablen (Arrays und Hashes) als Argumente an Funktionen übergeben – womit wir wieder bei unserer Beispielfunktion `kundenname` wären, die wir ja auch so implementieren möchten, dass sie das zu verarbeitende Kunden-Hash als Argument übernimmt.

Doch Vorsicht, es gibt da noch einen Haken. Perl bricht nämlich Listen, Arrays und Hashes, die man als Argumente an Funktionen übergibt, in ihre einzelnen Listenelemente auf. Wenn Sie also ein Array übergeben, steht danach in `$_[0]` nicht das Array, sondern das erste Element im Array, in `$_[1]` steht das zweite Array-Element und so fort. Wenn Sie ein Hash übergeben, steht in `$_[0]` der erste Schlüssel, in `$_[1]` der Wert zu dem Schlüssel, in `$_[2]` der zweite Schlüssel und so fort.

Wenn die Funktion nur ein einzelnes Array oder ein Hash als Argument übernimmt, ist das nicht so schlimm. Wurde ein Array übergeben, steht das Array innerhalb der Funktion `halt` in @_. Wurde ein Hash übergeben, kann man dieses aus der Liste der Werte in @_ rekonstruieren. Auf diese Weise können wir dann endlich eine abschließende Version von `kundenname` aufsetzen:

```
sub kundenname {
  %meinHash = @_;
  print "$meinHash{Vorname} $meinHash{Nachname} " ,
        "(KundenNr.: $meinHash{KundenNr})";
  }
...
kundenname(%kunde);
```

Das funktioniert allerdings nur so lange, wie der Funktion nur ein einziges Array oder ein Hash übergeben wird. Werden zusätzlich noch Skalare oder mehrere Arrays und Hashes übergeben, löst Perl alle diese Argumente in eine einzige große Liste auf. In so einem Fall zu versuchen, die einzelnen Argumente in der Parameterliste @_ ausfindig zu machen und zu rekonstruieren, dürfte in den meisten Fällen recht mühselig, wenn nicht gar unmöglich sein.

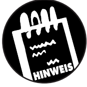
Glücklicherweise gibt es in Perl auch zu diesem Problem eine Lösung, und zwar – wie schon beim Problem der komplexen Datenstrukturen (siehe Abschnitt 5.4) – in Form von Referenzen.

Arrays und Hashes werden meist als Referenzen an Funktionen übergeben

Wenn Sie Arrays oder Hashes im Funktionsaufruf als Referenzen übergeben (einfach einen \ vor den Variablennamen stellen), speichert Perl die Referenzen, die ja einfache Skalare sind, wie gehabt in den Parametern `$_[0]` und `$_[1]`. In der Funktion kann man dann über diese Referenzen wie über normale Array- oder Hash-Variablen, allerdings mit leicht veränderter Syntax, auf die Elemente im Array oder im Hash zugreifen. Das folgende Skript soll dies verdeutlichen:

```
#!/usr/bin/perl -w

%kunde = ("Vorname" => "Rudolf",
          "Nachname" => "Ramge",
          "Strasse" => "Pflaumenweg",
          "Hausnummer" => 10,
          "PLZ" => 85123,
          "Stadt" => "Ottobrunn",
          "KundenNr" => 10023,
         );

@array1 = (1, 2, 3);

demoFunk(\@array1, \%kunde1); # Funktionsaufruf mit einer
                              # Array- u. einer Hash-Referenz
                              # als Argument

sub demoFunk {
  print "Array ausgeben: ";
  my $arrayRef = $_[0];
  foreach $elem (@$arrayRef) {
    print "$elem ";
    }
  print "\n\n";

  print "Hash ausgeben: ";
  my $hashRef = $_[1];
  foreach $key (keys %$hashRef) {
    print "$hashRef->{$key} ";
    }
  }
```

Analyse In der Funktion demoFunk werden die Referenzen aus $_[0] und $_[1] zuerst in eigene Referenzskalare kopiert ($arrayRef und $hashRef). Mit Hilfe dieser Referenzen werden dann das übergebene Array und das Hash in foreach-Schleifen durchlaufen, wobei zu beachten ist, dass den Referenznamen im Kopf der foreach-Schleife jeweils der @- oder der %-Operator vorangestellt wird, um anzuzeigen, dass es sich um eine Referenz auf ein Array bzw. ein Hash handelt.



In Kapitel 7 werden wir uns ausführlicher mit Referenzen beschäftigen.

6.4 Funktionen mit lokalen Variablen

Wenn Sie wollen, können Sie innerhalb einer Funktion auch Variablen deklarieren. Doch Vorsicht! Variablen, die Sie in Funktionen deklarieren, werden nicht von gleichnamigen Variablen außerhalb der Funktion unterschieden.

```
#!/usr/bin/perl -w

sub demoFunk {
  $dummy = 0;
  # weiterer Code
  }


$dummy = 12;

print "Variable vor  Funktionsaufruf: $dummy\n";   # 12

demoFunk();

print "Variable nach Funktionsaufruf: $dummy\n";   #  0
```

Hier wird eine Variable $dummy deklariert und mit dem Wert 12 initialisiert. Die nachfolgende print-Anweisung bestätigt dies. Danach wird die Funktion demoFunk aufgerufen, die $dummy den Wert 0 zuweist. Die Funktion demoFunk ändert also den Wert einer Variablen, die auch außerhalb der Funktion existiert.

Was auf den ersten Blick wie ein praktisches Mittel zum Datenaustausch zwischen Funktion und umliegendem Skriptcode aussieht, ist in Wirklichkeit eine teuflische Fehlerquelle. Betrachten wir dazu folgendes Beispiel:

```
#!/usr/bin/perl -w

sub tausche {
  $temp = $wert1;
  $wert1 = $wert2;
  $wert2 = $temp;
  }

$wert1 = 7;
$wert2 = 12;

print "$wert1  $wert2\n";
tausche();
print "$wert1  $wert2\n";
```

Hier tauscht die Funktion `tausche` die Werte der Variablen `$wert1` und `$wert2` aus. Wenn man das obige Skript ausführt, erhält man folgende Ausgabe:

```
7  12
12  7
```

Also alles wie gewünscht?!

Hören wir was Arthur Kochovsky, der Reich-Ranicki unter den Perl-Trainern (leicht untersetzte Statur, scharfe Zunge und irgendwie doch knuddelig), zu diesem Skript sagen würde.

»Grauenhaft. Die Simplizität dieser Wortschnipsel, die grobe Strickart des Textes ... das ist weit entfernt von der genialen Sparsamkeit eines Handkes, das zeugt von Belang- und Einfallslosigkeit. Ja, zugegeben, die Bedeutung der Worte tritt klar und für jeden erkennbar zu Tage, aber von einem guten Programmierer erwarten wir doch etwas mehr! Doch das Allerschlimmste (hier hebt Kochovsky mahnend den Finger), und da dürfen wir als Trainer nicht drüber wegsehen, sind die eklatanten Verstöße gegen die Grundregeln der Programmierkunst.«

Worin bestehen diese Verstöße gegen die Programmierkunst?

✘ Zum einem operiert die Funktion `tausche` auf globalen Variablen (`wert1` und `wert2`). Dies bedeutet, dass man a) die Namen dieser Variablen kennen muss, um die Funktion ordnungsgemäß aufrufen zu können, und b) Gefahr läuft, sich bei Aufruf der Funktion ungewollt die Werte gleichnamiger Variablen zu überschreiben. Besser und sauberer wäre es, wenn die Funktion die zu bearbeitenden Werte über Parameter entgegennehmen würde.

- Zum anderen deklariert die Funktion eine Hilfsvariable `temp` (»temp« wie temporär). Obwohl diese Variable eigentlich nur intern für die Implementierung der Funktion benötigt wird, ist sie global gültig und wirkt sich auf den umliegenden Code aus. Gibt es im umliegenden Code eine gleichnamige Variable, wird deren Wert bei Aufruf der Funktion überschrieben.

Stellen Sie sich vor, Sie würden diese Funktion in ein Perl-Modul packen und anderen Programmierern zur Verfügung stellen. Im schlimmsten Fall ruft einer der Programmierer die Funktion wie folgt auf:

```
($wert1, $wert2, $wert3, $wert4) = @array;
if ($wert4 > $temp)
  {
  tausche($wert3, $wert4);
  }
```

Der Programmierer liest die ersten vier Werte aus einem Array mit Temperaturwerten in Skalare. Dann prüft er, ob der letzte der Temperaturwerte höher ist als der Temperaturwert in einer Variablen `$temp` (»temp« ist hier eine Abkürzung für Temperatur), die er bereits weiter oben definiert und initialisiert hat. Ist der Wert in `$wert4` größer, ruft er die Funktion `tausche` auf, um die Werte in `$wert3` und `$wert4` zu tauschen.

Doch alles läuft schief. Der Perl-Interpreter stellt zwar die Werte von `$wert3` und `$wert4` in das Array `@_`, doch dieses wird von der Funktion `tausche` ja gar nicht verwendet. Also werden die Werte von `$wert3` und `$wert4` auch nicht getauscht. Dafür tauscht die Funktion die Werte von `$wert1` und `$wert2`. Zu guter Letzt wird noch der Wert der Temperatur in `$temp` auf den Wert von `$zahl1` gesetzt, weil der Programmierer seine Temperaturvariable zufällig genauso genannt hat wie die temporäre Variable in der Funktion `tausche`.

Was lernen wir daraus?

- Zu bearbeitende Daten sollte man immer als Parameter an Funktionen übergeben!
- Variablen, die nur innerhalb einer Funktion benötigt werden, sollten als lokale Variablen deklariert werden, die nur innerhalb der Funktion gültig sind!

Wie aber erzeugt man lokale Variablen? Mit Hilfe des Schlüsselwortes `my`.

6.4.1 Das Schlüsselwort my

Wenn Sie einer Variablendeklaration in einer Funktion das Schlüsselwort `my` voranstellen, erzeugen Sie eine Variable, die nur innerhalb der Funktion gültig ist. Gibt es im umliegenden Skriptcode eine gleichnamige Variable, verdeckt die lokale `my`-Variable in der Funktion die globale Variable aus dem umliegenden Code. Die Funktion greift daher immer auf den Wert der lokalen `my`-Variablen zu und der Wert der globalen Variablen bleibt unbehelligt.

```
#!/usr/bin/perl -w

sub demoFunk {
  my $dummy = 0;
  print "$dummy in Funktion: $dummy\n";
  }


$dummy = 12;
print "$dummy vor  Funktionsaufruf: $dummy\n";
demoFunk();
print "$dummy nach Funktionsaufruf: $dummy\n";
```

Ausgabe

```
Variable vor  Funktionsaufruf: 12
Variable in Funktion: 0
Variable nach Funktionsaufruf: 12
```

Wenn Sie in einer Funktion doch einmal auf eine globale Variable zugreifen wollen, die durch eine gleichnamige lokale Variable verdeckt ist, stellen Sie dem Variablennamen den Package-Namen voran. Alle globalen Variablen sind automatisch Bestandteil des Standard-Packages `main`. In obigem Skript könnte die Funktion `demoFunk` vermittels `$main::dummy` auf die globale Variable `$dummy` mit dem Wert 12 zugreifen.

Mit Hilfe von `my` können wir nun auch die Funktion `tausche` ordnungsgemäß implementieren:

```
sub tausche {
  my $temp = $_[0];
  $_[0] = $_[1];
  $_[1] = $temp;
  }

$wert1 = 7;
$wert2 = 12;

print "$wert1  $wert2\n";         #  7  12
tausche($wert1, $wert2);
print "$wert1  $wert2\n";         #  12  7
```

6.4.2 Unveränderliche Parameter

Dem Schlüsselwort `my` kommt bei der Implementierung von Funktionen noch eine weitere wichtige Aufgabe zu. Wie Sie am Beispiel der Funktion `tausche` gesehen haben, kann man über die Parameter in @_ nicht nur den Wert der Argumente abfragen, man kann die Argumente auch direkt ändern (außer natürlich es werden Konstanten statt Variablen als Argumente übergeben).

Manchmal – wie im Falle der `tausche`-Funktion – ist die direkte Änderung der übergebenen Argumente sehr praktisch. In anderen Fällen ist sie eher unerwünscht. Dann ist es angebracht, eine Kopie der Parameter anzulegen.

Dafür gibt es mehrere Möglichkeiten.

```
my $var = $_[0];
```

Sind es mehrere Parameter, die kopiert werden sollen, kann man diese auch in eine Skalarliste einlesen.

```
my ($var1, $var2) = @_;
my ($var1, undef, $var2) = @_;  # überspringt 2. Parameter
```

Oder Sie ziehen die einzelnen Parameter mit `shift` (siehe Kapitel 5.2.2) aus dem Array @_ heraus.

```
my $var1 = shift;
my $var2 = shift;
```

Wenn eine Funktion Argumente übernimmt, deren Inhalt sie nicht ändern soll, sollte man den Wert des zugehörigen Parameters in eine lokale `my`-Variable kopieren!

6.5 Funktionen mit Rückgabewerten

Am Beispiel der Funktion `tausche` haben Sie bereits gesehen, dass eine Funktion über ihre Parameter nicht nur Daten entgegennehmen, sondern auch zurückliefern kann (indem sie die zurückzuliefernden Daten in die Elemente des Arrays @_ schreibt).

Das Zurückliefern von Daten über die Parameter ist allerdings eher ungewöhnlich und setzt voraus, dass als Argumente zu den betroffenen Para-

metern keine Konstanten übergeben werden. Der übliche Weg zum Zurückliefern von Daten aus einer Funktion führt über das Schlüsselwort `return`.

Ein nettes Beispiel für eine Funktion, die einen Ergebniswert zurückliefert, ist die Implementierung einer Funktion zur Berechnung der Fakultät.

```
n! = n * (n-1) * (n-2) * ... * 1
0! = 1
```

Die Fakultät erhält man – wie obiger Formel zu entnehmen – durch wiederholt ausgeführte Multiplikationen, wobei der Faktor, mit dem multipliziert wird, jedes Mal um 1 verringert wird.

Listing 6.1: fakultaet.pl

```
#!/usr/bin/perl -w

sub fakultaet {
  my $zahl = shift;
  my $loop;
  my $fakul = 1;

  for ($loop=$zahl; $loop > 1; $loop--) {
     $fakul *= $loop;
     }

  return $fakul;        # Ergebnis zurückliefern
  }


$eingabe = 0;

print "\n Geben Sie eine Zahl ein, deren Fakultaet ",
      "berechnet werden soll:  ";

chomp ($eingabe = <STDIN>);


$fakul = fakultaet($eingabe);
print "\n$eingabe! = $fakul\n";
```

Die Fakultät ist eine extrem schnell ansteigende Funktion. Bereits für relativ kleine Eingaben wie die Zahl 10 ergeben sich sehr hohe Werte (10! = 3 628 800).

Abb. 6.1: Ausführung des Skripts fakultaet.pl

```
C:\Markt&T\JLI_Perl\Progs\Kap06>perl fakultaet.pl
 Geben Sie eine Zahl ein, deren Fakultaet berechnet werden soll:  3
3! = 6
C:\Markt&T\JLI_Perl\Progs\Kap06>perl fakultaet.pl
 Geben Sie eine Zahl ein, deren Fakultaet berechnet werden soll:  4
4! = 24
C:\Markt&T\JLI_Perl\Progs\Kap06>perl fakultaet.pl
 Geben Sie eine Zahl ein, deren Fakultaet berechnet werden soll:  5
5! = 120
C:\Markt&T\JLI_Perl\Progs\Kap06>
```

Das Zurückliefern von Ergebniswerten über `return` ist nicht nur meist eine saubere Lösung als das Manipulieren der Parameter, es hat auch den Vorteil, dass man die Funktion in Anweisungen als Platzhalter für den Ergebniswert der Funktion verwenden kann.

Die letzten Zeilen aus dem Skript FAKULTAET.PL:

```
$fakul = fakultaet($eingabe);
print "\n$eingabe! = $fakul\n";
```

hätte man also auch einfacher formulieren können:

```
print "\n$eingabe! = ", fakultaet($eingabe), "\n";
```

Funktionen mit mehreren return-Anweisungen

Im Grunde genommen hätten Sie in obiger Implementierung der Fakultätsfunktion das Schlüsselwort `return` sogar weglassen können, so dass in der letzten Anweisung der Funktion einfach die Variable `$fakul` stehen geblieben wäre. Perl liefert nämlich automatisch den Wert der letzten Anweisung einer Funktion als Rückgabewert zurück.

Der Vorteil des Schlüsselworts `return` liegt darin, dass es

- zum einem den zurückgelieferten Wert deutlich kenntlich macht und
- zum anderen die Funktion beendet und daher automatisch immer die letzte Anweisung der Funktion darstellt.

Den zweiten Effekt von `return` kann man dazu nutzen, eine Funktion an mehreren Stellen zu verlassen.

```
sub fakultaet {
  my $zahl = shift;
  my $loop;
  my $fakul = 1;

  if ($zahl == 0 || $zahl == 1)
     {
     print "Das war einfach!\n";
     return 1;
     }
    else
     {
     print "Das ist schon schwerer!\n";
     for ($loop=$zahl; $loop > 1; $loop--) {
        $fakul *= $loop;
        }
     return $fakul;
     }

  print "Dies ist nicht das Ende\n";
  }
```

Die obige Implementierung der Fakultät erspart sich für die Eingaben 0 und 1 die unnötig aufwendige Berechnung in der Schleife und liefert stattdessen direkt den Wert 1 zurück.

Die abschließende `print`-Anweisung kommt nie zur Ausführung, da die Funktion für jede Eingabe schon zuvor mit einer der `return`-Anweisungen verlassen wird.

Mehrere Ergebniswerte zurückliefern

Leser, die bereits mit C programmiert haben, wird es verwundern, Perl-Anhänger werden es als selbstverständlich ansehen: Sie können mit `return` auch Listen zurückliefern.

```
return @array;
```

oder

```
return ($erg1, $erg2, $erg3, 45.5);
```

6.6 Vertiefung: Packages, my und strict

Hinter den zwei Formen der Deklaration, die wir gesehen haben:

- der Deklaration durch erste Zuweisung und
- der Deklaration mit `my`

stehen zwei unterschiedliche Konzepte zur Verwaltung von Bezeichnern und zur Verhinderung von Namenskonflikten.

Globale Variablen

Die »Deklaration durch erste Zuweisung« ist eigentlich eine Deklaration durch Verwendung, d.h. wann immer Sie eine Variable mit einem neuen Namen verwenden (sei es in einer Zuweisung, in einem Ausdruck oder als Argument für eine Funktion), erkennt der Perl-Interpreter, dass es noch keine Variable mit diesem Namen gibt, und erzeugt sie. Der Perl-Interpreter fasst also automatisch die erste Verwendung einer Variablen als Deklaration auf. Da Variablen nach der Deklaration aber keinen Wert haben, sollte man als verantwortungsbewusster Programmierer darauf achten, dass die erste Verwendung eine Zuweisung ist, so dass die Variable danach einen definierten Wert besitzt.

```
$wert = 7;             # Deklaration und Zuweisung
$wert = 2 * $zaehler;  # Dekaration von $zaehler
                       # $zaehler ist undefiniert, liefert 0
```

Undefinierte Variablen

Variablen, die bei der Deklaration keinen Wert zugewiesen bekommen, sind undefiniert. Werden sie in einem numerischen Kontext verwendet (beispielsweise in einem arithmetischen Ausdruck), werden sie so behandelt, als enthielten sie den Wert 1, in Stringkontexten werden sie als leerer String '' interpretiert.

Mit Hilfe der Funktion `defined` kann man prüfen, ob eine Variable definiert ist oder nicht. Mit Hilfe der Funktion `undef` kann man den Wert einer Variablen löschen, so dass diese wieder undefiniert ist.

Variablen, die einfach dadurch deklariert werden, dass man ihren Namen verwendet, sind globale Variablen, d.h. sie können überall im Skript verwendet werden – jedenfalls kommt es dem Programmierer so vor, solange er nicht mit Packages arbeitet.

Packages

Packages sind Namensräume oder -bereiche, die den globalen Namensbereich aufteilen. Wenn Sie also eine globale Variable einführen, gehört diese automatisch dem aktuellen Namensbereich an. Wenn Sie selbst keinen anderen Namensbereich vorgeben, ist dies der Namensbereich des Standard-Package »`main`«. Eine global deklarierte Variable `$var` heißt also in Wirklichkeit `$main::var` (oder `$packageName::var`, falls Sie zuvor mit der Anweisung »`package packageName;`« einen anderen Namensbereich eingeschaltet haben).

Qualifizierte Bezeichner

Bezeichner, die aus dem Package-Namen und dem Variablennamen (oder Funktionsnamen) bestehen, nennt man »voll qualifizierte« Bezeichner. Mit ihrer Hilfe kann man auf Variablen (oder Funktionen) aus anderen Packages zugreifen oder auf Variablen in Funktionen, die durch lokale Variablen gleichen Namens verdeckt sind (siehe Abschnitt 7.4).

Doch welchen Sinn hat es überhaupt, globale Variablen auf Packages zu verteilen?

In einfachen Programmen ergibt es eigentlich keinen Sinn, mit Packages zu arbeiten. Dies ist auch der Grund, warum es das »unsichtbare« Standard-Package `main` gibt. Dank seiner impliziten Verwendung braucht sich der Programmierer beim Aufsetzen einfacher Skripten überhaupt keine Gedanken um die Verwendung von Namensbereichen/Packages zu machen und sein Skript ist dennoch automatisch Package-konform.

Interessant werden die Packages erst bei größeren Programmen oder wenn Code von anderen Perl-Modulen per `use`-Anweisung eingebunden werden (siehe Anhang B). Wenn Sie ein Skript mit globalen Variablen aufsetzen und ein Perl-Modul einbinden, das ebenfalls globale Variablen deklariert, kann es schnell passieren, dass zufällig eine Ihrer globalen Variablen genauso heißt wie eine globale Variable aus dem Modul. Die Folge ist, dass Perl beide Variablen als eine einzige Variable ansieht – ein unangenehmer Fehler.

Hier setzt nun das Konzept der Packages an. Indem der Entwickler des Moduls alle im Modul deklarierten Bezeichner in ein von ihm selbst definiertes Package stellt (das üblicherweise den gleichen Namen trägt wie das Modul), werden Namenskonflikte zwischen globalen Package-Elementen und globalen Elementen des umliegenden Skripts vermieden. Im Abschnitt 7.7 werden wir ein Beispiel für die Erstellung eines eigenen Moduls und eines zugehörigen Package sehen.

Lokale Variablen

Durch die Einrichtung von Packages kann man den globalen Namensbereich aufteilen. Auf diese Weise kann man sehr effektiv Namenskonflikte verhindern, die durch ansonsten gleich lautende globale Variablen (und Funktionen) in verschiedenen Modulen entstehen würden.

Innerhalb des Codes eines Skripts (oder Moduls) kann man Namenskonflikten aus dem Weg gehen, indem man für Variablen, die nur für die Ausführung einer Funktion oder eines Anweisungsblocks benötigt werden, innerhalb des Blocks mit Hilfe eines der Schlüsselwörter `my` oder `local` deklariert.

```
sub demo {
  my  $wert;                # nur in Funktion gültig

  ...

  for(my $loop=1; $loop < 2; $loop++)  # $loop und
    {                                  # $wert sind nur in
    my $wert = 100;                    # for-Schleife
                                       # gueltig.
                                       # $wert verdeckt $wert
    }                                  # aus Funktion demo
  }
```

Grundsätzlich werden Variablen, die lokal zu einem Block sein sollen, mit `my` deklariert. Das Schlüsselwort `local`, mit dessen Hilfe man ebenfalls lokale Variablen deklarieren kann, wird nur benötigt, wenn eine Variable, die lokal zu einer Funktion ist, in einer untergeordneten Funktion sichtbar sein soll – aber wie gesagt, das Schlüsselwort `local` wird nur selten wirklich gebraucht.

```
sub demo {
  my    $wert_my    = 10;
  local $wert_local = 20;

  print "$wert_my\t";
  print "$wert_local\n";

  demo2();
  }

sub demo2 {
  print "$wert_my\t";         # $wert_my undefiniert
  print "$wert_local\n";      # $wert_local aus demo
  }
```

Schließlich können Sie my-Variablen auch außerhalb von Funktionen und Blöcken deklarieren. In diesem Fall ist der »Block«, zu dem die Variablen lokal sind, die Quelltextdatei des Skripts. Man spricht in diesem Fall vom Dateibereich oder »file scope«.

Der Unterschied zwischen einer my-Variablen im Dateibereich und einer globalen Variablen ist der, dass my-Variablen im Dateibereich nicht zum aktuellen Package gehören.

```
#!/usr/bin/perl -w

   $wert_global = 1;       # globale Variable
my $wert_my     = 2;       # file scope-Variable


$main::wert_global = 100;  # $main::wert_global und
                           # $wert_global sind identisch

$main::wert_my     = 200;  # erzeugt neue globale Variable

print "$wert_global\n";         # Ausgabe 100
print "$main::wert_global\n";   # Ausgabe 100
print "$wert_my\n";             # Ausgabe 2
print "$main::wert_my\n";       # Ausgabe 200
```

strict

Dass Perl die erste Verwendung einer Variablen als Deklaration auffasst, bedeutet, dass es in Perl keine nichtdeklarierten Variablen gibt. Dies begünstigt zwei Arten von Fehlern:

- ✘ Die Verwendung einer Variablen, bevor dieser ein Wert zugewiesen wurde
- ✘ Neudeklaration ungewollter Variablen durch Tippfehler

```
$wert = 7;
$wert = 2 * $zaehler;    # Verwendung vor Zuweisung

...
$zaehler = 1;
...
```

Hier hat der Programmierer mitten in seinem Skript die Variable `$zaehler` deklariert und mit dem Anfangswert 1 initialisiert. Später hat er dann den Code weiter oben überarbeitet, sich seiner bereits deklarierten Variablen `$zaehler` erinnert und diese in den Ausdruck `2 * $zaehler` eingebaut. Dabei hat er übersehen, dass `$zaehler` erst weiter unten initialisiert wird.

```
$zaehler = 1;
...
$zaheler++;                # Tippfehler
```

Hier hat sich der Programmierer vertippt. Seine Absicht war wohl, den Wert von `$zaehler` zu inkrementieren, aber stattdessen hat er eine neue Variable `$zaheler` deklariert und dieser den Wert 1 zugewiesen.

Um sich gegen solche Fehler zu wappnen, gibt es zwei Möglichkeiten:

- ✘ `-w`. Sie rufen Perl mit der Option `-w` auf, so dass der Perl-Interpreter neben Fehlermeldungen auch Warnungen ausgibt. Ist die Option `-w` gesetzt, gibt der Perl-Interpreter für jede Verwendung einer undefinierten Variablen eine Warnung der folgenden Form aus:

  ```
  Use of uninitialized value in multiplication (*) at
  test07_12.pl line 5.
  ```

  Trifft der Perl-Interpreter dagegen auf eine Variable, die nur ein einziges Mal verwendet wird, nimmt er an, dass es sich um einen Tippfehler handeln könnte, und gibt folgende Warnung aus:

  ```
  Name "main::zaheler" used only once: possible typo at
  test07_12.pl line 10.
  ```

- ✘ `strict`. Sicherer als die Ausgabe von Warnungen ist die Verwendung des Pragmas `use strict;`. Wenn Sie dieses Pragma an den Anfang Ihrer Skripten stellen (was wir ab jetzt in unseren Beispielen tun werden), erlaubt der Perl-Interpreter keine impliziten Deklarationen durch erste Verwendung mehr, sondern fordert, dass alle verwendeten Variablen zuvor mit `my` oder `use vars` deklariert werden.

```
#!/usr/bin/perl -w
use strict;

use vars qw($wert);      # global deklarierte Variable

my $zaehler;             # lokale Variable (file scope)


$wert    = 7;
$zaehler = 2;
%wert    = 2 * $loop;    # Fehler! $loop wurde nicht vorab
                         # deklariert
```

```
sub demo {
  my $wert = 12;
  print "$wert\n";          # 12
  print "$main::wert\n";    #  7, Zugriff auf verdeckte
                            #     Package-Variable
  }
```

6.7 Vertiefung: eigene Module erstellen

Wenn Sie erst einmal die ersten, allgemein nützlichen Funktionen erstellt haben, wird es nicht lange dauern, bis Sie es leid sind, den Code dieser Funktion per Cut&Paste in Ihre aktuellen Skripten zu kopieren. Wäre es nicht wesentlich angenehmer, wenn Sie die Funktionen in ein Modul packen und dieses bei Bedarf über eine `use`-Anweisung einbinden könnten?

Modul erstellen

Als Demonstrationsobjekt verwenden wir die Fakultätsfunktion aus Abschnitt 7.5. Um diese in einem Modul abzulegen, gehen Sie wie folgt vor:

1. Legen Sie mit einem einfachen ASCII-Editor eine neue Datei für das Modul an.
2. Erzeugen Sie in der ersten Zeile des Moduls mit Hilfe des Schlüsselworts `package` einen neuen Namensbereich, der den gleichen Namen trägt, unter dem Sie später das Modul speichern werden.

   ```
   package MathExt;
   ```

3. Kopieren Sie den Code der Fakultätsfunktion in die Datei.
4. Schließen Sie den Code des Moduls mit der Anweisung `1;` ab. (Dies ist erforderlich, da die `use`-Anweisung vom Modul einen Rückgabewert erwartet.)

   Listing 6.2: MathExt.pm

   ```
   package MathExt;

   sub fakultaet {
     my $zahl = shift;
     my $loop;
     my $fakul = 1;
   ```

```
  for ($loop=$zahl; $loop > 1; $loop--) {
     $fakul *= $loop;
     }

  return $fakul;        # Ergebnis zurückliefern
  }

1;
```

5. Speichern Sie die Datei im obersten LIB-Verzeichnis Ihres Perl-Compilers. (Wenn in dem Verzeichnis bereits die pm-Dateien LIB.PM oder OPEN.PM stehen, haben Sie das richtige Verzeichnis gewählt.)

Modul einbinden und Elemente verwenden

Jetzt möchten wir das neu erzeugte Modul in einem Skript verwenden.

1. Legen Sie mit einem einfachen ASCII-Editor eine neue Datei für das Skript an.
2. Binden Sie mit Hilfe einer `use`-Anweisung das Modul ein.

```
#!/usr/bin/perl -w

use MathExt;
```

3. Setzen Sie den Code des Skripts auf.
4. Verwenden Sie die Funktion `fakultaet` aus dem Modul. Denken Sie daran, dass Sie dem Funktionsnamen den Package-Namen des Moduls voranstellen müssen.

Listing 6.3: fakultaet2.pl

```
#!/usr/bin/perl -w

use MathExt;


$eingabe = 0;
print "\n Geben Sie eine Zahl ein, deren Fakultaet ",
      "berechnet werden soll:  ";

chomp ($eingabe = <STDIN>);

$fakul = MathExt::fakultaet($eingabe);    # Aufruf der
                                          # Modul-Funktion

print "\n$eingabe! = $fakul\n";
```

Zugriff auf Elemente vereinfachen

Da wir für das Modul ein eigenes Package definiert haben, können Zugriffe auf die im Modul definierten Elemente nur über voll qualifizierte Bezeichner erfolgen. Dies ist sehr sinnvoll, da es Namenskonflikte effektiv verhindert. Es kann aber auch lästig sein. Um die qualifizierten Bezeichner zu umgehen, gibt es mehrere Möglichkeiten:

- ✘ Wenn das Skript, welches das Modul einbindet, nur dieses oder hauptsächlich dieses Modul verwendet, könnte man im Skript das Package des Moduls einschalten.

```
#!/usr/bin/perl -w

use MathExt;
package MathExt;
...
```

Sinnvoller sind Abwandlungen im Modul selbst.

- ✘ Deklarieren Sie im Modul keinen eigenen Namensbereich, so dass die im Modul deklarierten Elemente automatisch dem Standard-Namensbereich `main` angehören.
- ✘ Exportieren Sie die Elemente des Moduls in den Namensbereich des aufrufenden Skripts.

```
# in Modul
package MathExt;
require Exporter;
@ISA    = qw(Exporter);
@EXPORT = qw(fakultaet);
```

Statt alle in der Exportliste aufgeführten Elemente bedingungslos mit `@EXPORT` in den Namensbereich des aufrufenden Skripts zu exportieren, können Sie die Elemente mit `@EXPORT_OK` exportieren. In der `use`-Anweisung müssen die Elemente, die importiert werden sollen, dann explizit angegeben werden: `use MathExt qw(fakultaet);`.

6.8 Fragen und Übungen

1. Durch welches Schlüsselwort werden Funktionsdefinitionen eingeleitet?
2. Wie können Sie mit einem einzigen zusätzlichen Zeichen das Listing aus Abschnitt 7.2 (Berechnung der Mehrwertsteuer) so ändern, dass die Preise im Array `@liste` durch die berechnete Mehrwertsteuer ersetzt werden?

```
#!/usr/bin/perl -w

@liste = (100, 3000, 89.90, 125.40);

sub mwst { $_ * 0.16; }              # Funktionsdefinition
@mwst_liste = map(mwst, @liste);     # Übergabe als Argument

print "@mwst_liste\n";
```

3. Wie deklariert man Variablen, die zu einer Funktion lokal sind?
4. Wie kann eine Funktion Werte aus dem Aufruf entgegennehmen?
5. Wie kann man verhindern, dass die Funktion die Variablen, die als Argumente übergeben werden, verändert?
6. Wie kann eine Funktion Ergebniswerte zurückliefern?
7. Setzen Sie eine Funktion auf, die ein Array als Argument übernimmt und die Summe der Elemente im Array zurückliefert.
8. Wann muss man die Aufrufargumente zu einer Funktion in runde Klammern setzen?
9. Wie deklariert man eine Schleifenvariable, die zu der Schleife lokal ist?
10. Was ist der Unterschied zwischen einer `my`-Variablen im Dateibereich und einer globalen Variablen?
11. Welcher Unterschied besteht zwischen `my` und `local`?
12. Schreiben Sie das Modul MATHEXT.PM so um, dass es den Bezeichner der Funktion `fakultaet` optional exportiert, und formulieren Sie das Skript FAKULTAET2.PL so um, dass es den Bezeichner importiert.

Für Fortgeschrittene

Kapitel 7: Referenzen
Kapitel 8: Kontext und Standardvariablen
Kapitel 9: Ein- und Ausgabe
Kapitel 10: Suchen mit regulären Ausdrücken
Kapitel 11: Objektorientierte Programmierung in Perl
Kapitel 12: Grafische Oberflächen

Gratulation! Den Grundlagenteil haben Sie erfolgreich hinter sich gebracht. Jetzt wäre ein guter Zeitpunkt, das Buch erst einmal zur Seite zu legen und das Gelernte ein wenig zu verinnerlichen. Schreiben Sie kleine Testprogramme, in denen Sie das Gelernte überprüfen. Entwickeln Sie eigene Programmideen und versuchen Sie, diese umzusetzen. Gehen Sie in Gedanken den Stoff noch einmal durch und stellen Sie sich selbst Fragen: Was sind Variablen? Wie implementiert man eine `for`-Schleife? Wie fügt man neue Elemente in ein Array ein? Setzen Sie einen schriftlichen Aufsatz über die Vorteile des Pragmas `use strict;` auf. Lernen Sie den Grundlagenteil auswendig. Gründen Sie eine Perl-Selbsthilfegruppe, in der Sie einmal wöchentlich über interessante Perl-Probleme diskutieren. Träumen Sie von Perl. Träumen Sie **in** Perl!

Nein, ganz im Ernst, wir sind hier ja nicht in einem Trainingskurs, und Sie müssen auch keine Abschlussklausur ablegen. Sicher, das Wissen, das Sie aus dem Grundlagenteil mitbringen, ist wichtig – zumal der Stoff im Fortgeschrittenenteil vorausgesetzt wird. Aber wer kann bei einer solchen Fülle an Informationen schon alles im Kopf behalten. Grämen Sie sich also nicht, wenn Sie sich an bestimmte Konzepte und Konstrukte nicht mehr genau erinnern können. Das Buch geht Ihnen ja nicht verloren. Schlagen Sie einfach die betreffende Stelle im Buch nach und frischen Sie Ihre Erinnerung auf. Und wenn Sie eine spezielle Syntax vergessen haben (Perl ist ja reich an Symbolen und kryptisch anmutenden Konstruktionen), dann schauen Sie doch erst einmal im Anhang A nach, wo ich für Sie die wichtigsten Symbole, Syntaxformen und Konstrukte zusammengestellt habe.

Was den nun folgenden Fortgeschrittenenteil angeht, so möchte ich Sie noch einmal darauf hinweisen, dass wir in diesem etwas schneller voranschreiten werden als im Grundlagenteil. Etliche der Kapitel in diesem Teil behandeln Themengebiete, die meiner Einschätzung nach für die Perl-Programmierung nicht so wichtig sind, die aber den einen oder anderen Leser sicher interessieren werden (beispielsweise die Kapitel zur objektorientierten Programmierung in Perl oder die Erstellung von Perl-Skripten mit grafischen Benutzeroberflächen). Diese Kapitel sind dazu gedacht, den Leser in die betreffenden Themengebiete einzuführen und ihm – falls er in dieser Richtung programmieren will – einen guten Start zu verschaffen. Andere Kapitel behandeln Themen, die, obwohl es sich um fortgeschrittene Techniken handelt, von fundamentaler Bedeutung für die Programmierung mit Perl sind (beispielsweise die Kapitel zur Ein- und Ausgabe oder zu den regulären Ausdrücken). Diese Themen werden selbstverständlich ausführlicher besprochen.

Kapitel 7

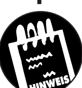
Referenzen

Referenzen gehören erst seit Version 5 zu Perl, haben aber bereits ihren festen Platz unter den Perl-Konzepten gefunden. Mit ihrer Hilfe kann man auf elegante Weise komplexe Datenstrukturen aufbauen und als Argumente an Funktionen übergeben.

Im Einzelnen lernen Sie,

- ✘ was Referenzen sind
- ✘ wie man Referenzen definiert
- ✘ wie man die Adresse einer Variablen bestimmt
- ✘ wie man Referenzen dereferenziert

Im letzten Abschnitt erfahren Sie, wie man Referenzen sinnvoll einsetzt.



Folgende Elemente lernen Sie kennen

Die Referenzen, den Adressoperator \, die verschiedenen Syntaxformen zur Dereferenzierung von Referenzen auf Skalare, Arrays und Hashes.

7.1 Was sind Referenzen?

Referenzen sind Verweise auf andere Variablen (Skalare, Arrays oder Hashes). Wenn man zu einer Variablen eine Referenz erzeugt, kann man danach sowohl über den Variablennamen als auch über die Referenz auf den Wert der Variablen zugreifen.

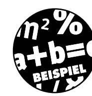
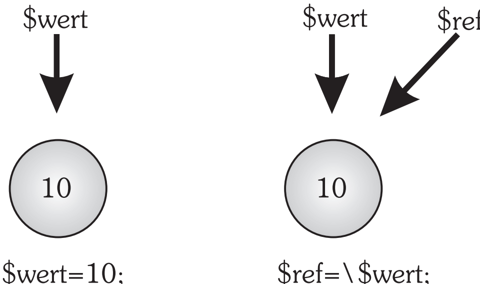


Abb. 7.1: Skalar und Referenz verweisen auf das gleiche Speicherobjekt



Wenn Sie eine Variable deklarieren und ihr einen Wert zuweisen, reserviert der Perl-Interpreter im Arbeitsspeicher des Computers einen bestimmten Bereich, in den er den Wert der Variablen hineinschreibt. Die Speicheradresse, an der dieser Speicherbereich beginnt, verbindet er mit dem Variablennamen. Die Variablennamen, mit denen wir arbeiten, stehen also letztendlich für die Adressen von Speicherbereichen.

Wenn Sie eine Referenz auf eine Variable deklarieren, reserviert der Perl-Interpreter für die Referenz einen zugehörigen Speicherbereich und legt darin die Adresse der Variablen ab. Da die Referenz die Adresse des Speicherbereichs der Variablen kennt, kann man über die Referenz auf den Speicherbereich der Variablen zugreifen.

Abb. 7.2: Adressen eines Skalars ($wert) und einer Referenz ($ref) sowie der Speicherinhalt dieser Adressen nach den Zuweisungen $wert = 10; und $ref = \$wert;.

Symboltabelle

$wert	0x1F00
$ref	0x1F04

Speicher

Inhalt	Adresse
10	0x1F00
	0x1F02
0x1F00	0x1F04
	0x1F06

Perl-Referenzen versus C-/C++-Referenzen

Perls Referenzen unterscheiden sich sowohl von den Referenzen als auch von den Zeigern, die man von C/C++ kennt.

Perls Referenzen können wie die C++-Referenzen nur auf definierte Objekte (Variablen) verweisen. Sie können nicht wie C-/C++-Zeiger auf beliebige Speicheradressen verweisen.

Perls Referenzen können nacheinander auf unterschiedliche Speicherobjekte verweisen.

Perls Referenzen verändern die Referenzzähler der Speicherobjekte. Wenn Sie eine Referenz auf ein Speicherobjekt einrichten, wird der Referenzzähler des Speicherobjekts um 1 erhöht. Wird die Referenz auf ein anderes Speicherobjekt gesetzt, wird der Referenzzähler des alten Speicherobjekts um 1 vermindert. Im Referenzzähler hält Perl fest, wie viele Variablen und Referenzen auf ein Speicherobjekt weisen, und sorgt dafür, dass das Speicherobjekt so lange bestehen bleibt, wie es noch mindestens eine Variable oder eine Referenz gibt, die auf das Objekt weist.

7.2 Wie deklariert man Referenzen?

Referenzen sind einfache skalare Variablen, deren Werte Adressen anderer Variablen sind. Eine Referenz wird daher wie ein normaler Skalar deklariert. Wie aber setzt man eine Referenz auf eine Variable? Man kann schließlich nicht einfach die Variable zuweisen:

```
$var = 3;
$ref = $var;
```

Obiger Code erzeugt natürlich keine Referenz, sondern eine ganz normale skalare Variable, die zufällig `$ref` heißt und den Wert von `$var` zugewiesen bekommt. Um aus `$ref` eine echte Referenz auf `$var` zu machen, müssen wir `$ref` nicht den Wert, sondern die Adresse von `$var` zuweisen. Dazu stellt man dem Variablennamen den Adressoperator \ voran.

```
$ref = \$var;        # Referenz auf Skalar
$ref = \@array;      # Referenz auf Array
$ref = \%hash;       # Referenz auf Hash
```

Wenn Sie sich darüber vergewissern wollen, dass in `$ref` wirklich die Adresse der angegebenen Variablen steht, geben Sie `$ref` einfach mit `print` aus.

Da Referenzen wie alle Skalare mit `$` deklariert werden, kann man sie nicht von anderen skalaren Variablen unterscheiden. Es empfiehlt sich daher, Referenzen Namen zu geben, die darauf hindeuten, dass es sich um eine Referenz handelt (beispielsweise durch Anhängen eines passenden Suffix: `_ptr` oder `_ref`). Dies verhindert allerdings nicht, dass Sie einer Referenz auch einen normalen skalaren Wert zuweisen können. Wenn Sie sich vergewissern wollen, dass in einer Referenzvariablen auch tatsächlich eine Referenz enthalten ist, rufen Sie die Funktion `ref` auf und übergeben Sie der Funktion die zu prüfende Variable. Handelt es sich tatsächlich um eine Referenz, liefert die Funktion den Wahrheitswert »wahr« zurück.

Referenzen können auf beliebige Speicherobjekte weisen, neben Skalaren, Arrays und Hashes also auch auf Funktionen, Datei-Handles oder andere Referenzen.

7.3 Wie greift man auf Referenzen zu?

Wenn Sie eine Referenz wie eine normale Variable verwenden, greifen Sie lediglich auf die in der Referenz gespeicherte Adresse zu. Auf diese Weise können Sie eine Referenz initialisieren oder auf eine andere Variable umlenken.

```
$var1 = 3;
$var2 = 4;
$ref = \$var1;
$ref = \$var2;
```

Dereferenzierung

Das Einrichten einer Referenz würde allerdings nicht viel Sinn ergeben, wenn es nicht auch möglich wäre, über die Referenz auf den Wert der Variablen zuzugreifen, auf die die Referenz verweist. Für den Zugriff auf die referenzierte Variable gibt es übrigens einen eigenen Fachausdruck: Man nennt dies »Dereferenzierung«.

Um eine Referenz zu dereferenzieren, bedarf es einer besonderen Syntax. Diese sieht so aus, dass man den Namen der Referenz in geschweifte Klammern einschließt und diesem »Block« das Präfix für den Datentyp des zurückgelieferten Wertes voransetzt. Die Dereferenzierung eines Skalars sieht dementsprechend wie folgt aus:

```
$var = 3;
$ref = \$var;
${$ref} = 2222;            # Dereferenzierung
```

Hier wird `$ref` dereferenziert. Das Ergebnis der Dereferenzierung liefert einen Skalar (`$ref` wurde als Referenz auf `$var2` eingerichtet). Diesem Skalar wird der Wert 2222 zugewiesen. Auch der Wert von `$var` ist jetzt gleich 2222, denn schließlich beziehen sich `$var` und `${$ref}` ja auf das gleiche Speicherobjekt!

Nach dem gleichen Muster geht man vor, wenn man auf ein Array zugreifen möchte. Dabei muss man allerdings aufpassen, ob man auf das ganze Array oder nur ein einzelnes Array-Element zugreifen möchte. Im ersten Fall muss man dem Blockausdruck das Präfix @ voranstellen, im zweiten Fall verwendet man das Präfix $ und hängt an den Blockausdruck den Index an.

```
@array  = (1, 2, 3, 4, 5);

$array_ref = \@array;       # Array-Referenz deklarieren
push( @{$array_ref}, 7 );   # Zugriff auf ganzes Array
${$array_ref}[2] = -3;      # Zugriff auf Array-Element
print @{$array_ref}, "\n";
```

Vereinfachungen

Die Syntax zur Dereferenzierung lässt sich – wie fast alles in Perl – ein wenig vereinfachen.

Wenn Sie auf das vollständige Speicherobjekt zugreifen wollen, lassen Sie einfach die geschweiften Klammern weg:

`@$array_ref` statt `@{$array_ref}`

Wenn Sie auf ein Array- oder Hash-Element zugreifen wollen, schreiben Sie einfach den Referenznamen und hängen dann über den Operator -> den Index an.

`$array_ref->[2]` statt `${$array_ref}[2]`
`$hash_ref->{'Name'}` statt `${$hash_ref}{'Name'}`

Übersicht

Tabelle 7.1 gibt Ihnen eine Übersicht über die verschiedenen Formen der Deklaration und Dereferenzierung.

Tabelle 7.1: Syntax für Referenzen

Operation	Skalar	Array	Hash
Deklaration	$ref_s = \$skalar	$ref_a = \@array	$ref_h = \%hash
Dereferenzierung	$$ref_s	@$ref_a	%$ref_h
	${$ref_s}	@{$ref_a}	%{$ref_h}
Dereferenzierung	–	${$ref_a}[0]	${$ref_h}{'key'}
(auf Element)		$ref_a->[0]	$ref_h->{'key'}

7.4 Wofür braucht man Referenzen?

Wenn es über Referenzen nicht mehr zu berichten gäbe, als dass man sie auf bestehende Variablen richten und dann durch Dereferenzierung auf den Inhalt dieser Variablen zugreifen kann, wären die Referenzen für die Perl-Programmierung nicht sonderlich interessant. Der besondere Vorteil der Referenzen liegt darin, dass man mit ihrer Hilfe Dinge machen kann, die sich mit anderen Variablen nicht so einfach lösen lassen.

7.4.1 Komplexe Datenstrukturen

Wie Sie wissen, sind als Elemente für Listen (und damit auch für Arrays und Hashes) nur Skalare erlaubt. Konstruktionen wie Listen von Listen oder Arrays von Hashes sind somit nicht möglich.

Nun haben wir aber einen weiteren Skalartyp kennen gelernt: die Referenzen. Da Referenzen Skalare sind, kann man sie in Listen einbauen. Da Referenzen aber auch auf Arrays und Hashes verweisen können, stellt es kein Problem mehr dar, komplexe Datenstrukturen wie Arrays von Hashes oder Hashes von Hashes aufzubauen.

Arrays von Arrays

Um ein Array von Arrays zu erzeugen, nehmen Sie einfach einzelne Array-Referenzen in das Gesamtarray auf.

```
@array1  = (1, 2);
@array2  = (33, 44, 55);
@array   = (\@array1, \@array2);
```

Sie können auch Referenzen auf Listen erzeugen, indem Sie die Listen in eckige Klammern setzen:

```
@array   = ([1, 2], [33, 44, 55]);
```

Wenn Sie die Elemente im Array durchgehen wollen, setzen Sie eine äußere Schleife auf, die die Array-Referenzen im Gesamtarray durchläuft. In einer zweiten, inneren Schleife greifen Sie über die aktuelle Referenz auf das zugehörige untergeordnete Array zu und gehen dessen Elemente durch.

```
foreach $array_ref (@array) {
  foreach $elem (@{$array_ref}) {
     print "$elem ";
     }
  print "\n";
  }
```

Arrays von Hashes

Um ein Array von Hashes zu erzeugen, nehmen Sie einfach einzelne Hash-Referenzen in das Array auf.

```
%hash1  = ("Vorname" => "Rip", "Nachname" => "van Winkle");
%hash2  = ("Vorname" => "Peter", "Nachname" => "Kreuter");
@array   = (\%hash1, \%hash2);
```

Sie können auch Referenzen auf Hashes erzeugen, indem Sie die Hashes in geschweifte Klammern setzen.

```
@array   = ({"Vorname" => "Rip", "Nachname" => "van Winkle"},
            {"Vorname" => "Peter", "Nachname" => "Kreuter"});
```

Wenn Sie die Elemente im Array durchgehen wollen, setzen Sie eine äußere Schleife auf, die die Array-Referenzen im Gesamtarray durchläuft. In einer zweiten, inneren Schleife greifen Sie über die aktuelle Referenz auf das zugehörige untergeordnete Hash zu und gehen dessen Elemente durch.

```
foreach $hash_ref (@array) {
  foreach $key (keys %{$hash_ref}) {
     print "$key: ", "$hash_ref->{$key} / " ;
     }
  print "\n";
  }
```

Nach dem gleichen Muster kann man auch noch Hashes von Hashes aufbauen, doch das überlasse ich Ihnen (siehe Übung 5).

Hashes mit Listen

Bei Vorstellung der Hashes in Kapitel 5.3 wurde bereits darauf hingewiesen, dass es grundsätzlich zu jedem Hash-Schlüssel nur einen Wert geben darf. Man kann diese Einschränkung aber umgehen, indem man zu einem Hash-Schlüssel eine Array-Referenz als Wert vorsieht. In diesem Array kann man dann beliebig viele Werte speichern.

```
%hash  = ("Vorname" => "Rip",
          "Nachname" => "van Winkle",
          "Soehne" => ["Peter", "Ralf", "Joseph"]);


foreach $key (keys %hash) {
  print "$key: ";
  if( ref( $hash{$key} ))
    {
```

```
    foreach $elem (@{$hash{$key}}) {
      print "$elem ";
      }
    }
    else
    {
    print "$hash{$key} ";
    }
  print "\n";
  }
```

In der Schleife wird – wie für Hashes gewohnt – eine Liste der Schlüssel (`keys`) erstellt und diese dann durchlaufen. Für jeden Schlüssel wird dann in einer `if`-Bedingung und mit Hilfe der Funktion `ref` überprüft, ob es sich bei dem Wert des Schlüssels (`$hash{$key}`) um eine Referenz handelt. Wenn ja, wird der Wert des Schlüssels dereferenziert, wobei wir davon ausgehen, dass die Referenz auf eine Liste verweist: `@{$hash{$key}}`. Das resultierende Array wird dann Element für Element ausgegeben.

7.4.2 Übergabe von Referenzargumenten an Funktionen

Wie Sie bereits aus Kapitel 6.3 wissen, löst Perl Hashes und Arrays, die an Funktionen übergeben werden, in ihre Elemente auf. Um dies zu verhindern, kann man die Arrays/Hashes als Referenzen übergeben.

```
#!/usr/bin/perl -w

%kunde = ("Vorname" => "Rudolf",
          "Nachname" => "Ramge",
          "KundenNr" => 10023,
         );

@array1 = (1, 2, 3);

demoFunk(\@array1, \%kunde1); # Funktionsaufruf mit einer
                              # Array- u. einer Hash-Referenz
                              # als Argument

sub demoFunk {
  print "Array ausgeben: ";
  my $arrayRef = $_[0];
  foreach $elem (@$arrayRef) {
    print "$elem ";
    }
  print "\n\n";
```

```
  print "Hash ausgeben: ";
  my $hashRef = $_[1];
  foreach $key (keys %$hashRef) {
    print "$hashRef->{$key} ";
    }
  }
```

In der Funktion `demoFunk` werden die Referenzen aus `$_[0]` und `$_[1]` zuerst in eigene Referenzskalare kopiert (`$arrayRef` und `$hashRef`). Mit Hilfe dieser Referenzen werden dann das übergebene Array und das Hash in `foreach`-Schleifen durchlaufen.

7.4.3 Dynamischer Aufbau von Arrays

Referenzen kann man auf die Speicherobjekte anderer Variablen setzen. Das ist uns bereits bekannt. Referenzen erhöhen den Referenzzähler der Objekte, auf die sie gesetzt werden. Das ist uns ebenfalls bekannt (siehe Abschnitt 7.1), aber es ist doch wert, dass man noch einmal darüber nachdenkt.

Wenn Sie in einem Block eine lokale `my`-Variable deklarieren, existiert diese nur so lange, wie der Block ausgeführt wird. Wird der Block beendet, werden die Variable und das dazugehörige Speicherobjekt automatisch von Perl aufgelöst. (Wird der gleiche Block später noch einmal ausgeführt, wird die lokale Variable neu erzeugt.) Indem man aber eine Referenz auf die lokale Variable einrichtet, erreicht man, dass beim Verlassen des Blocks nur die Variable aufgelöst wird, während das Speicherobjekt bestehen bleibt.

Man kann dies beispielsweise nutzen, um in einer Einleseroutine ein Array von Hashes aufzubauen. Der folgende Code liest aus einer Datei namens CDs.cvs Informationen über die CDs einer CD-Sammlung ein. Jede Zeile der Datei enthält die Daten zu einer CD: Komponist oder Interpret, Titel, Musikkategorie.

```
my $CD_db = "CDs.cvs";
my @cds = ();                          # Liste der CDs

open(CD_FILE, $CD_db) or
   die "\nDatei $CD_db konnte nicht geoeffnet werden ($!)\n";
```

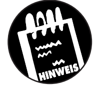
```
while(<CD_FILE>) {                    # Zeile für Zeile einlesen
  my %cd = ();                        # lokales Hash

  ($t, $c, $k) = split(",", $_);      # eingelesene Zeile in
                                      # Felder aufsplitten

  $cd{composer}  = $c;                # Hash erzeugen
  $cd{titel}     = $t;
  $cd{kategorie} = $k;

  push(@cds, \%cd);                   # Referenz auf Hash in
                                      # Array eintragen
}
```

In diesem Code wird innerhalb der `while`-Schleife eine lokale Hash-Variable namens `%cd` erzeugt. Die nächsten Zeilen dienen dann dazu, die einzelnen Daten aus der eingelesenen Zeile aufzutrennen und den Schlüsseln des Hash zuzuweisen. Die letzte Codezeile der `while`-Schleife trägt eine Referenz auf die lokale Hash-Variable in das Array `@cds` ein.

Der Trick besteht nun darin, dass bei Beenden des Schleifenblocks die lokale Variable `%cd` aufgelöst wird. Das Speicherobjekt, das zuvor noch mit der Variablen verbunden war, bleibt aber erhalten, weil es noch eine Referenz gibt, die auf das Objekt verweist: die Referenz, die in das Array `@cds` eingetragen wurde. Beim nächsten Schleifendurchlauf (für die nächste Eingabezeile) wird die Variable `%cd` neu erzeugt und es wird ein neues Speicherobjekt für die Variable eingerichtet. Nachdem dem Speicherobjekt die Werte aus der Eingabezeile zugewiesen wurden, wird auch für dieses Speicherobjekt eine Referenz in das Array `@cds` eingetragen.

Auf diese Weise wird das Array `@cds` Hash für Hash aufgebaut, ohne dass man für jedes Hash eine eigene Hash-Variable deklarieren müsste.

Wenn Sie die CD-Daten in eine globale Variable `%cd` einlesen würden, gäbe es nur ein Speicherobjekt und bei jedem neuerlichen Schleifendurchlauf würden die alten CD-Daten mit den neuen Daten überschrieben. Im Array `@cds` stünden dann lauter Referenzen auf ein und dasselbe Objekt.



Das vollständige Programm zur CD-Verwaltung finden Sie in Kapitel 14.

7.5 Fragen und Übungen

1. Was ist eine Referenz?
2. Wie deklariert man Referenzen auf Skalare, Arrays und Hashes?
3. Kann man auch Referenzen auf Funktionen einrichten? Wenn ja, welchen Sinn könnte das haben?
4. Was liefern die folgenden Dereferenzierungen zurück? In den Beispielen ist absichtlich nicht angegeben, worauf die Referenz `$ref` verweist, da Sie dies aus der Syntax der Derefenzierung ablesen sollen.

```
a)  ${$ref}[0]
b)  $$ref
c)  @{$ref}
d)  @$ref
e)  %$ref
f)  $ref->[0]
g)  $ref->{'key'}
h)  %{$ref}
i)  ${$ref}{'key'}
```

5. Bauen Sie ein Hash von Hashes auf.

Kapitel 8

Kontext und Standardvariablen

Dieses Kapitel ist in gewisser Weise ein erläuternder Nachtrag und eine zusammenfassende Darstellung zu zwei wichtigen Perl-spezifischen Konzepten, mit denen wir schon die ganze Zeit zu tun hatten: der Bedeutung des Kontextes und der Verwendung von Standardvariablen.

Im Einzelnen erfahren Sie,

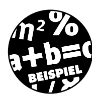
- ✘ was Perl unter Kontext versteht
- ✘ welche Kontexttypen es gibt und wie sie entstehen
- ✘ welchen Effekt der Kontext haben kann
- ✘ wie man den Kontext selbst festlegen kann

sowie

- ✘ was Standardvariablen sind und
- ✘ welche Standardvariablen es gibt

Folgende Elemente lernen Sie kennen

Die Funktion `scalar` und den Operator `()` zur Umwandlung von Kontexten sowie folgende Standardvariablen: `$_`, `@_`, `$a`, `$b`, `$/`, `$\`, `$#`, `$^L`, `$ARGV`, `%ENV`.



8.1 Kontext

Laut Linguistik stellt ein Kontext eine »sprachliche Umgebung« dar, die die Bedeutung von Wörtern und Satzteilen beeinflussen kann. Der sinngebende oder sinnwandelnde Einfluss des Kontextes ist in unserer eigenen Sprache so alltäglich, dass er uns kaum noch bewusst wird.

Beispielsweise ist die Bedeutung von Pronomen (z.B. das »er« im vorangehenden Satz) stets nur aus dem Kontext zu schließen. Manche Wörter (etwa die »Teekesselchen« Ball und Atlas) haben unterschiedliche Bedeutungen. In welcher Bedeutung ein solches Wort gebraucht wird, hängt dann vom Kontext ab (der Ball, auf dem getanzt wird, und der Ball, mit dem man spielt; der Atlas, der Länder und Meere abbildet, und der Hüne Atlas, der das Himmelsgewölbe trägt).

Während unser Gehirn Kontextinformationen offenbar spielend verarbeiten kann, tun sich Compiler und Interpreter äußerst schwer damit. Die meisten Programmiersprachen sind daher weitgehend kontextfrei. Nicht jedoch Perl!

8.1.1 Kontext in Perl

Das Kontextkonzept von Perl gleicht am ehesten der Verwendung von Wörtern mit mehreren Bedeutungen. Es gibt jedoch einen kleinen Unterschied in der Verwendung des Kontextes. In der natürlichen Sprache wird der Kontext meist unbewusst verarbeitet. Erst wenn man auf ein Wort trifft, dessen Bedeutung aus sich selbst nicht zu erschließen ist, wird der Kontext mit einbezogen. Ein solches Verfahren ist in einer Programmiersprache nicht umsetzbar. Perl geht daher umgekehrt vor.

Die verschiedenen Perl-Konstrukte, Operatoren und Funktionen definieren einen bestimmten Kontext (insgesamt gibt es fünf Kontexttypen). Dieser Kontext legt dann fest, wie die darin auftretenden Variablen, Operatoren und Funktionen interpretiert oder ausgeführt werden.

Die beiden wichtigsten Kontexte sind:

- der skalare Kontext und
- der Listenkontext

Den skalaren Kontext kann man weiter unterteilen in einen numerischen, einen Booleschen und einen String-Kontext.

Kontextsensitive Elemente

Variablen (und Ausdrücke) sind in dem Sinne kontextsensitiv, als ihr Wert in unterschiedlichen Kontexten unterschiedlich interpretiert wird. Ein Skalar in einem skalaren Kontext ist dabei nicht weiter aufregend, ebenso wenig wie die Verwendung einer Liste (eines Arrays oder Hash) in einem Listenkontext. Interessant wird es erst, wenn eine Variable in einem Kontext verwendet wird, der nicht mit ihrem eigentlichen Typ übereinstimmt.

- ✘ Listen und Arrays werden im skalaren Kontext als die Anzahl ihrer Elemente ausgewertet.
- ✘ Hashes werden im skalaren Kontext als String interpretiert, der Aufschluss über die interne Realisierung des Hash gibt (für uns ist dies nicht weiter interessant).
- ✘ Skalare werden im Listenkontext als einelementige Listen ausgewertet.

Skalare Werte können ja nach Kontext in Zahlen (numerischer Kontext), Strings (String-Kontext) oder Wahrheitswerte (Boolescher Kontext) umgewandelt werden.

Kontextsensitiv sind auch **Funktionen**, die in Skalar- und Listenkontext unterschiedliche Ergebnisse zurückliefern. Verwechseln Sie dieses Verhalten nicht mit dem Aufruf einer nicht kontextsensitiven Funktion, deren Rückgabewert je nach Kontext unterschiedlich interpretiert wird. Die kontextsensitiven Funktionen liefen wirklich in verschiedenen Kontexten unterschiedliche Werte und führen manchmal auch ganz andere Anweisungen aus. Ein Beispiel für eine kontextsensitive Funktion ist `localtime`. Wird diese Funktion in einem skalaren Kontext aufgerufen, liefert sie einen formatierten Datums-Uhrzeit-String zurück, wird sie in einem Listenkontext aufgerufen, liefert sie eine Liste ihrer internen Werte (Sekunde, Minute etc.).

Schließlich gibt es auch **Operatoren**, die kontextsensitiv sind. So liest der Eingabeoperator `<>` im skalaren Kontext nur eine einzelne Zeile, während er im Listenkontext eine Zeile nach der anderen bis zum »Dateiende« einliest.

Kontexterzeugende Elemente

Eine Zuweisung an einen Skalar erzeugt einen skalaren Kontext. Eine Zuweisung an ein Array erzeugt einen Listenkontext.

```
@liste = (-1, 0, 1, 2, 3);

$skalar = @liste;   # 5
@array  = $skalar;  # (5)
```

Viele der gängigen Operatoren definieren für ihre Operanden einen bestimmten Kontext. Beispielsweise erwarten die arithmetischen Operatoren (+, -, *, /) und die numerischen Vergleichsoperatoren (==, !=, <, <=> etc.) Zahlen als Operanden (numerischer Kontext), während der Konkatenationsoperator . und die Stringvergleichsoperatoren (`eq`, `ne`, `lt`, `cmp` etc.) nur Strings als Operanden akzeptieren.

```
print "Geben Sie eine Zahl ein: ";
$eingabe = <>;
chomp($eingabe);

$eingabe *= 3;                # der arithmetischer Operator
                              # wandelt den String in $eingabe
                              # in eine Zahl um.

$str = "Neuer Wert: " . $eingabe;   # der .-Operator wandelt
                                    # die Zahl in $eingabe in
                                    # einen String um
```

Verzweigungen und Schleifen erzeugen in ihren Bedingungen einen Booleschen Kontext

```
while (1) {          # Werte ungleich 0 werden in Bedingung
 ...                 # (Boolescher Kontext) als »wahr«
                     # interpretiert
```

Etliche der vordefinierten Perl-Funktionen erwarten Listen als Argumente (siehe PERLFUNC-Dokumentation).

```
print localtime();  # gibt unformatierte
                    # Datum/zeitinformationen aus
                    # bspw.: 19391022510041731
```

8.1.2 Wie kann man einen Kontext ändern?

In den meisten Fällen werden Sie feststellen, dass die verschiedenen Perl-Elemente immer genau den Kontext erzeugen, der im Grunde auch vom Programmierer gewünscht wird. Und sollte ein Perl-Konstrukt doch einmal den »falschen« Kontext erzeugen, dann liegt es oftmals daran, dass der Programmierer die falsche Syntax gewählt hat.

```
if ($eingabe != "bsuzf") {     # != statt ne verwendet
   ...
@zeile = <>;                   # @ statt $ erzeugt hier
                               # Listenkontext -> der
                               # Operator <> liest
                               # bis zum Dateiende
```

Sollte man wirklich einmal mit dem vorgegebenen Kontext nicht einverstanden sein, kann man den Kontext explizit ändern.

scalar

Mit Hilfe der Funktion `scalar` ist es möglich, innerhalb eines Listenkontextes einen skalaren Kontext zu erzwingen. So kann man beispielsweise erzwingen, dass kontextsensitive Funktionen wie `localtime` in einem Listenkontext (wie im Aufruf von `print`) skalar ausgewertet werden.

```
print scalar localtime();    # liefert ein formatierte
                             # Datum/Zeitangabe
                             # Thu Jun 22 11:21:53 2000
```

Oder man kann einem Array die Anzahl der Elemente einer Liste als neues Element zuweisen.

```
@liste = (-1, 0, 1, 2, 3);
@array  = scalar @liste;
```

Hier bewirkt `scalar`, dass `@liste` – trotz der Zuweisung an ein Array – im skalaren Kontext ausgewertet wird und die Anzahl der Elemente in `@liste` zurückliefert. Dieser skalare Wert wird dann – wegen der Ausführung im Listenkontext – zu einer einelementigen Liste umgewandelt und `@array` zugewiesen.

()

Umgekehrt ist es auch möglich, in einem skalaren Kontext einen Listenkontext zu erzwingen. So kann man beispielsweise in einem skalaren Kontext eine kontextsensitive Funktion im Listenkontext ausführen lassen.

```
$sek = (localtime())[0];     # liefert die Sekunden der
                             # aktuellen Uhrzeit
```

8.1.3 Wie implementiert man kontextsensitive Funktionen?

Mit Hilfe der vordefinierten Funktion `wantarray` können Sie in einer selbst definierten Funktion prüfen, ob diese in einem skalaren Kontext oder einem Listenkontext aufgerufen wird. So können Sie je nach dem Kontext des Aufrufs die Funktion entweder einen skalaren Wert oder eine Liste von Werten zurückgeben lassen.

```
sub meineFunktion {
  if (wantarray) {
    # wantarray hat »wahr« ergeben -> Listenkontext
    return @array;
    }
```

```
  else {
    # wantarray hat »falsch« ergeben -> skalarer Kontext
    return $skalar;
    }
  }
```

Manchmal kann man das zurückzuliefernde Array und den Skalar in einem Zug berechnen. Dann kann man `wantarray` zusammen mit dem Bedingungsoperator `?:` in der `return`-Anweisung der Funktion aufrufen. Die folgende Funktion gibt hierfür ein Beispiel: Sie addiert die Zahlen in der ihr übergebenen Liste und liefert je nach Wunsch die Gesamtsumme oder ein Array mit den Teilsummen zurück.

Listing 8.1: summe.pl

```
#!/usr/bin/perl -w

sub summe {
  my @werte = @_;
  my $loop;
  my $summe = 0;
  my @summen = ();

  push (@summen, $summe);
  for ($loop=0; $loop <= $#werte; $loop++) {
     $summe += $werte[$loop];
     push (@summen, $summe);
     }

  return wantarray ? @summen : $summen[$#summen];
  }


$summe = summe(1, 2, 3, 4, 5, 6, 7);
print "\nGesamtsumme: $summe\n";

@summe = summe(1, 2, 3, 4, 5, 6, 7);
print "Teilsummen:  " . join(", ", @summe), "\n";
```

```
MS-DOS-Eingabeaufforderung
C:\Markt&T\JLI_Perl\Progs\Kap08>
C:\Markt&T\JLI_Perl\Progs\Kap08>
C:\Markt&T\JLI_Perl\Progs\Kap08>
C:\Markt&T\JLI_Perl\Progs\Kap08>
C:\Markt&T\JLI_Perl\Progs\Kap08>
C:\Markt&T\JLI_Perl\Progs\Kap08>perl summe.pl

Gesamtsumme: 28
Teilsummen:  0, 1, 3, 6, 10, 15, 21, 28

C:\Markt&T\JLI_Perl\Progs\Kap08>_
```

Abb. 8.1: Ausführung des Skripts summe.pl

Die `return`-Anweisung dieser Funktion lautet: »Wenn `wantarray` wahr ergibt, liefere das Array `@summen` zurück, ansonsten das letzte Element im Array `@summen`.

8.2 Standardvariablen

Eine zweite Eigentümlichkeit von Perl sind die zahlreichen globalen Standardvariablen, die in vielen Situationen zum Einsatz kommen oder – falls gewünscht – zum Einsatz kommen können. So zum Beispiel die Übergabe von Funktionsargumenten im globalen Array `@_` oder der Einsatz der Standardvariablen `$_` in `foreach`-Schleifen.

Kurz- und Langform

Die meisten Standardvariablen gibt es in Kurzform und in Langform. Die Kurzform besteht aus zwei bis drei Symbolen, die Langform aus einem richtigen, aussagekräftigen Namen in Großbuchstaben. Für manche Standardvariablen gibt es keine Langform oder keine Kurzform oder es gibt gleich zwei Langformen.

Tabelle 8.1: Kurz- und Langform

Kurzform	Langform	Beschreibung
$_	$ARG	Die Standardvariable
$\	$OFS $OUTPUT_RECORD_SEPARATOR	Feldtrenner bei der Ausgabe mit `print`
$$	$PID $PROCESS_ID	ID des aktuellen Prozesses
@_		Parameterliste von Funktionen
	@ARGV	Argumente aus der Kommandozeile

Wenn Sie die Langformen verwenden wollen, müssen Sie diese mit Hilfe des Pragmas `use English;` aktivieren. Ich würde Ihnen aber dringend empfehlen, sich gleich mit den Kurzformen vertraut zu machen, weil Ihnen diese in bestehendem Perl-Code weitaus häufiger begegnen werden und weil Sie Ihnen beim Aufsetzen eigener Skripten viel Tipparbeit ersparen.

Übersicht

Perl kennt wirklich eine ganze Reihe von Standardvariablen! Sie alle in diesem Kapitel vorzustellen, wäre äußerst aufwendig und ziemlich sinnlos, da Sie etliche dieser Standardvariablen vermutlich nie benötigen werden. Ich werde mich daher auf eine kleine repräsentative Auswahl beschränken und Sie ansonsten auf die PERLVAR-Dokumentation verweisen.

$_ Die wichtigste Standardvariable ist zweifelsohne `$_`. Etliche Perl-Konstrukte und Funktionsaufrufe können verkürzt werden, indem man mit `$_` arbeitet.

- `foreach`-Schleifen kopieren das aktuell zu bearbeitende Array-Element standardmäßig nach `$_`.

  ```
  foreach (@liste) {...}
  ```

  entspricht

  ```
  foreach $elem (@liste) {...}
  ```

- Viele vordefinierte Perl-Funktionen verwenden `$_` als Standardargument (siehe PERLFUNC-Dokumentation).

  ```
  print;
  sin;
  @liste = split(}',}');
  ```

  entsprechen

  ```
  print $_;
  sin $_;
  @liste = split(}',}' $_);
  ```

- Die Funktionen `grep` und `map` verwenden `$_` zum Auswählen/Manipulieren der Array-Elemente.

  ```
  @gefunden = grep {$_ < 5} @werte;
  @quadrat = map {$_ * $_} @werte;
  ```

- Die Operatoren `m//` und `s//` für das Pattern Matching mit regulären Ausdrücken (siehe Kapitel 10) verwenden automatisch `$_`, wenn man den =~-Operator weglässt.

  ```
  s/alterText/neuerText/;
  ```

  entspricht

  ```
  $_ =~ s/alterText/neuerText/;
  ```

- ✘ Der Eingabeoperator <> verwendet standardmäßig $_.

  ```
  while (<>) {...}
  ```

  entspricht

  ```
  while ($_=<>) {...}
  ```

Das Listenpendant zu $_ ist @_.

@_

- ✘ Perl verwendet @_ als Parameterliste für Funktionen

  ```
  sub funk { print "Parameter: @_"; }
  ```

- ✘ Verschiedene Listfunktionen verwenden @_ als Standardargument (siehe PERLFUNC-Dokumentation).

  ```
  shift;
  pop;
  ```

  entsprechen

  ```
  shift @_;    # wenn in Funktion aufgerufen
  pop @_;
  ```

Die Sortierfunktion sort legt die zu vergleichenden Aray-Elemente in den Standardvariablen $a und $b ab (siehe Kapitel 5.2.4).

$a, $b

```
@sortiert = sort {$b <=> $a} @array;
```

Verschiedene Standardvariablen wie $/ (Zeilentrenner) oder $\ (Feldertrenner) beeinflussen die Arbeitsweise der vordefinierten Funktion print (siehe Kapitel 9.1.2).

print

Wie für print gibt es auch für den Report-Generator write spezielle Standardvariablen wie $# (Ausgabeformat für Zahlen) oder $^L (Seitentrenner) (siehe Kapitel 9.1.3).

write

Über die Standardvariable $ARGV kann man die Kommandozeilenargumente abfragen, die dem Programm beim Aufruf von der Konsole übergeben wurden (siehe Kapitel 9.3).

$ARGV

Über die Standardvariable %ENV kann man die Werte der gesetzten Umgebungsvariablen abfragen (siehe Kapitel 9.4).

%ENV

Des Weiteren gibt es Standardvariablen für das Pattern Matching, für Statusmeldungen ($0 = Name des laufenden Programms, $^O = Name des Betriebssystems, @INC = Suchpfad für Perl-Module etc.), Prozessinformationen ($$ = ID des aktuellen Prozesses, $< = User-ID) und anderes.

8.3 Fragen und Übungen

1. Welche Arten von Kontext unterscheidet Perl?
2. Welcher Kontext liegt in den folgenden Konstrukten vor?

```
a)  print @array
b)  print $var;
c)  if($var == 1)
d)  $var = 3 + @array;
```

3. Wie kann man einen skalaren Kontext erzwingen?
4. Nennen Sie vier Fälle, in denen die Standardvariable `$_` verwendet werden kann.

Kapitel 9

Ein- und Ausgabe

Programme haben im Grunde nur eine Aufgabe: Daten zu verarbeiten. Bevor ein Programm jedoch Daten verarbeiten kann, muss es irgendwie an die Daten herankommen. Manche Programme können sich ihre Daten selbst erzeugen. Beispielsweise wird ein Programm, das die Quadratzahlen der ersten 100 natürlichen Zahlen berechnen soll, sich seine Ausgangsdaten (die 100 natürlichen Zahlen) einfach mit Hilfe einer `for`-Schleife erzeugen. Doch dies sind Ausnahmen! Die weitaus überwiegende Zahl der Programme nimmt seine Daten, oder zumindest einen Teil der benötigten Daten, von außen auf. Das Einlesen von Daten ist daher ein ganz wichtiger Bestandteil praktisch jedes Programms. Mindestens ebenso wichtig ist die Ausgabe der Ergebnisse. Ein Programm, das Daten verarbeitet, ohne die Resultate abzuspeichern, auszugeben oder zumindest an andere Programme weiterzuleiten, wäre schließlich ziemlich nutzlos.

Im Einzelnen lernen Sie,

- ✘ wie man mit Hilfe des Eingabeoperators Daten von der Tastatur oder aus einer Datei einliest
- ✘ wie man die Funktion `print` durch Setzen verschiedener Standardvariablen konfigurieren kann
- ✘ wie man mit Hilfe des Report Generator (der Funktion `write`) anspruchsvolle tabellarische Ausgaben erzeugen kann
- ✘ wie man Dateien öffnet und schließt
- ✘ wie man in Dateien schreibt und aus Dateien liest

- ✘ was man unter Binärdateien und dem Binärmodus versteht
- ✘ wie man Argumente aus dem Skriptaufruf übernimmt
- ✘ wie man die Werte von Umgebungsvariablen abfragt

Folgende Elemente lernen Sie kennen

Den Eingabeoperator <> und die vordefinierten Datei-Handles `STDIN`, `STDOUT` und `STDERR`, die Standardvariablen `$.`, `$\`, `$,`, `$"`, `$|` und `$=`, `$%`, `$.`

Die Dateifunktionen `open`, `close` und `binmode`, die Funktionen `die`, `pack` und `unpack`, die Standardvariablen `@ARGV` und `%ENV`.

9.1 Ein- und Ausgabe

In diesem Abschnitt werden wir uns näher mit der Ein- und Ausgabe über die Konsole beschäftigen. Konsolenprogramme können Eingaben über die Tastatur mit dem Eingabeoperator <> einlesen und ihre Ergebnisse mit `print`, `printf` oder `write` auf den Konsolenbildschirm ausgeben.

Programme mit grafischen Benutzeroberflächen (siehe Kapitel 12) verwenden Fenster zum Einlesen und Ausgeben von Daten. Da dem Informationsaustausch über Fenster gänzlich andere Mechanismen als dem Datenaustausch über die Konsole zugrunde liegen, kann man die hier vorgestellten Techniken in diesen Programmen nicht verwenden.

9.1.1 Der Eingabeoperator <>

Den Eingabeoperator <> kennen Sie bereits aus den Beispielen der vorangehenden Kapitel. Die wichtigsten Fakten zum Eingabeoperator wollen wir an dieser Stelle noch einmal zusammentragen und uns einige Anwendungsbeispiele anschauen.

Der Eingabeoperator liest Textzeilen aus »Dateien«

Der Eingabeoperator liest Textzeilen ein. Von wo er diese Textzeilen einliest, hängt von dem »Datei-Handle« ab, das ihm in seinen spitzen Klammern übergeben wird. Dieses Datei-Handle kann eine wirkliche Datei (siehe Abschnitt 10.2) oder ein Eingabegerät repräsentieren.

Für das Standardeingabegerät, üblicherweise die Tastatur, gibt es einen bereits vordefinierten Datei-Handle, `STDIN`. Mit Hilfe dieses Handles kann man Daten über die Tastatur einlesen.

```
$eingabe = <STDIN>;
```

Daten über die Kommandozeile einlesen

Wenn man dem Eingabeoperator keinen Datei-Handle übergibt, prüft er, ob in der Kommandozeile Dateien angegeben wurden. Wenn nein, liest der Operator von `STDIN` ein (`<>` und `<STDIN>` sind für diesen Fall also äquivalent). Wurden Dateien in der Kommandozeile angegeben, öffnet der Operator diese Dateien, hängt sie aneinander und liest sie zeilenweise ein.

Wer gerne Schach spielt, der findet sicherlich auch Gefallen daran, berühmte Schachpartien nachzuspielen. Unter den Beispielen zu diesem Buch[1] befindet sich eine Datei SKHAN.CHS, die in Kurznotation die Partie des indischen Schachgenies Sulthan Khan[2] gegen Capablanca enthält. Mit Hilfe des folgenden Skripts können Sie sich diese Partie zum Nachspielen zeilenweise ausgeben lassen.

Listing 9.1: skhan1.pl

```
#!/usr/bin/perl -w

while ($zeile = <>) {
  chomp ($zeile);
  print $zeile;
  <STDIN>;
  }
```

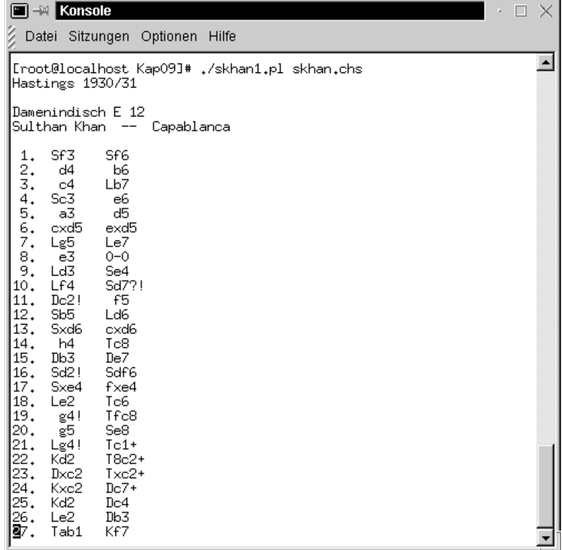
Aufruf: **> perl skhan1.pl skhan.chs** (Windows)

Aufruf: **# ./skhan1.pl skhan.chs** (Linux)

1 Die Beispielskripten zu diesem Buch können vom Markt&Technik-Server *www.mut.de* oder von meiner Website *www.civilserve.com/rdlouis* heruntergeladen werden.

2 Der Inder Mir Malik Sulthan Khan (1905-1966) war einer der wenigen Menschen, die es je schafften, den Groß- und Schachweltmeister Capablanca zu besiegen (Capablanca hat in seinem ganzen Leben offiziell nur 35 Partien verloren). Dabei war Sulthan Khan alles andere als ein Schachprofi. Er kam 1929 als Diener seines Herrn Sir Hoyat Khan nach England und machte dort durch sein geniales, intuitives Spiel am Schachbrett schnell auf sich aufmerksam. In nur wenigen Jahren begeisterte er die gesamte europäische Schachwelt, um schließlich 1933 mit seinem Herrn nach Pakistan zu gehen und damit genauso geheimnisvoll zu verschwinden, wie er aufgetaucht war.

Abb. 9.1: Die ersten Zeilen der Schachpartie



Analyse Wird das Skript zusammen mit einer Textdatei aufgerufen, liest die `while`-Schleife die Zeilen der Datei nacheinander ein. `chomp()` schneidet das mit eingelesene Zeilentrennzeichen ab (standardmäßig ist dies das Neue-Zeile-Zeichen), `print` gibt die aktuelle Zeile aus.

Mit <STDIN> kann man ein Programm anhalten

Danach wird die Programmausführung mit Hilfe der Anweisung `<STDIN>` angehalten. Erst wenn der Anwender die ↵-Taste drückt, wird das Programm fortgesetzt und die nächste Zeile ausgegeben. Das Drücken der ↵-Taste ist übrigens auch der Grund dafür, dass wir das Neue-Zeile-Zeichen aus der eingelesenen Zeile entfernt haben. Ansonsten würden zwischen zwei Zeilen nämlich immer zwei Zeilenumbrüche erfolgen (einmal durch das Neue-Zeile-Zeichen der eingelesenen Zeile und einmal durch Drücken der ↵-Taste).

Der Eingabeoperator kann zusammen mit $_ verwendet werden

Wenn Sie den Eingabeoperator in einer `while`-Bedingung aufrufen, brauchen Sie nicht unbedingt eine Variable für die Aufnahme der eingelesenen Zeilen anzugeben. In `while`-Bedingungen liest der Eingabeoperator nach `$_` ein, wenn keine explizite Zuweisung erfolgt. Das Perl-Skript SKHAN1.PL könnte man also auch wie folgt schreiben:

Listing 9.2: skhan2.pl

```
#!/usr/bin/perl -w

while (<>) {          # liest nach $_
  chomp();
  print ;
  <STDIN>;
  }
```

Der Eingabeoperator ist kontextsensitiv

Der Eingabeoperator unterscheidet, ob er in einem skalaren oder einem Listenkontext aufgerufen wird.

- ✘ Wenn Sie das Ergebnis des Operators an einen Skalar zuweisen, rufen Sie den Operator in einem skalaren Kontext auf. In diesem Fall liest der Operator genau eine Zeile ein.

```
$skalar = <>;       # liest einzelne Zeile
```

- ✘ Wenn Sie das Ergebnis des Operators einem Array zuweisen, rufen Sie den Operator in einem Listenkontext auf. In diesem Fall liest der Operator so lange Zeilen ein, bis er auf das Dateiende stößt.

```
@array = <>;       # liest Zeilen bis Dateiende
```

Bei Eingaben über die Konsole kann man das Dateiendezeichen als [Strg][D] (unter Unix/Linux) oder [Strg][Z] (unter Windows) eingeben.

Eigene Zeilentrennzeichen definieren

Dass der Eingabeoperator zeilenweise einliest, liegt übrigens an dem Wert der Standardvariablen `$/`. Der Eingabeoperator ist nämlich so implementiert, dass er so lange Zeichen einliest, bis er auf ein Zeichen trifft, das mit dem Zeichen in `$/` übereinstimmt. Da in `$/` standardmäßig das Neue-Zeile-Zeichen (`"\\n"`) abgelegt ist, liest der Eingabeoperator per Voreinstellung ganze Zeilen.

Es ist kein großes Problem, ein eigenes Trennzeichen anzugeben, beispielsweise:

```
$/ = "\t";       # Tabulator
$/ = '4';        # Die Ziffer 4
$/ = "";         # Absatzmarke
```

Allerdings sollten Sie dies nur tun, wenn Sie »Zeilen« aus Dateien einlesen. Wenn Sie Daten über die Tastatur einlesen und das Zeilentrennzeichen um-

definieren, bedeutet dies nämlich, dass der Anwender seine Eingaben über die Tastatur mit dem von Ihnen definierten Zeilentrennzeichen abschließen muss. Ansonsten werden die Eingaben trotz Drückens der ↵-Taste nicht akzeptiert.

Mehrere Daten auf einmal über die Tastatur einlesen

Wenn Ihnen daran gelegen ist, mehrere Werte auf einmal über die Eingabe einzulesen, sollten Sie nicht versuchen, mit eigenen Zeilentrennzeichen zu operieren.

Lesen Sie die Werte nacheinander ein:

```
%kunde = ();

print "\n";
print "Geben Sie den Vornamen  des Kunden ein: ";
chomp($eingabe = <STDIN>);
$kunde{'Vorname'} = $eingabe;

print "Geben Sie den Nachnamen des Kunden ein: ";
chomp($eingabe = <STDIN>);
$kunde{'Nachname'} = $eingabe;

print "Geben Sie die Kundennr. des Kunden ein: ";
chomp($eingabe = <STDIN>);
$kunde{'Kundennr'} = $eingabe;

print %kunde, "\n";
```

Oder lesen Sie die Werte auf einmal ein und splitten Sie die eingelesene Zeile dann mit Hilfe der Funktion `split` auf:

```
%kunde = ();

print "\n";
print "Geben Sie Vornamen, Nachname und Kundennummer ein: ";
chomp($eingabe = <STDIN>);

($vn, $nn, $kn) = split(' ', $eingabe);

$kunde{'Vorname'} = $vn;
$kunde{'Nachname'} = $nn;
$kunde{'Kundennr'} = $kn;

print %kunde, "\n";
```

9.1.2 Ausgabe mit print

Die Funktion `print` (sowie ihre Schwesterfunktion `printf` zur formatierten Ausgabe) kennen Sie bereits aus etlichen Beispielen und aus Kapitel 2.5. Wovon ich Ihnen bisher aber noch nichts erzählt habe, sind die Standardvariablen, die Sie zusammen mit `print` verwenden können.

$. Zeilennummer

Wenn Sie den Inhalt einer Datei zeilenweise in einer Schleife einlesen und bearbeiten, steht in `$.` die aktuelle Zeilennummer. Diese können Sie mit `print` (oder auch mit `printf`) ausgeben.

Das folgende Skript liest eine Datei über die Kommandozeile ein und gibt den Inhalt der Datei mit Zeilennummern aus.

Listing 9.3: zeilen.pl

```
#!/usr/bin/perl -w

while (<>) {
  printf("%-5s %s", $., $_);
  }
```

Wenn Sie diesem Programm in der Kommandozeile die Datei eines Perl-Skripts übergeben, erhalten Sie eine durchnummerierte Ausgabe des Skripts – was beispielsweise interessant sein kann, um Warnungen oder Fehlermeldungen des Perl-Interpreters nachzuspüren.

Wenn Sie wollen, können Sie die Ausgabe auch in eine Datei umleiten:

```
> perl zeilen.pl skhan.pl > nummeriert.txt    (Windows)
> ./zeilen.pl skhan.pl > nummeriert.txt    (Linux)
```

Abb. 9.2: Nummerierung von skhan1.pl

$\ Abschluss einer Ausgabe

Die meisten Programmiersprachen kennen Ausgabefunktionen, die jede Ausgabe automatisch mit einem Zeilenumbruch abschließen. `print` tut dies allerdings nicht, weswegen wir das Neue-Zeile-Zeichen häufig explizit an die Ausgaben angehängt haben:

```
print "$var \n";
```

oder

```
print $var,  "\n";
```

Es gibt aber auch die Möglichkeit, `print` so einzustellen, dass Ausgaben automatisch mit einem Zeilenumbruch abgeschlossen werden. `print` hängt nämlich an jede Ausgabe das Zeichen aus der Standardvariablen `$\` an. Da diese per Voreinstellung leer ist, merkt man davon nichts. Wenn man ihr aber das Neue-Zeile-Zeichen zuweist, werden alle nachfolgenden Ausgaben mit einem Zeilenumbruch abgeschlossen.

```
$\ = "\n";
print $var;       # Ausgaben stehen jeweils in
print $var;       # neuen Zeilen
```

$, und $" Elementtrenner für Arrays

Worin besteht der Unterschied zwischen den beiden folgenden Ausgaben?

```
print @array;
print "@array";
```

Im ersten Fall werden die Elemente des Arrays direkt hintereinander ausgegeben. Im zweiten Fall werden die Elemente durch Leerzeichen getrennt. Dieses Verhalten beruht auf den Standardvariablen `$,` und `$"`.

Die Variable `$,` legt das Trennzeichen für die Elemente aus Arrays (und Hashes) fest, die ohne oder in einfachen Anführungszeichen ausgegeben werden:

```
$, = ":";
$kunde{'Vorname'}  = "Rip";
$kunde{'Nachname'} = "van Winkle";
$kunde{'Nr'} = 12345;

print %kunde;     # Nr:12345:Nachname:van Winkle:Vorname:Rip

@array = ("Rip", "van Winkle", 12345);

print @array;     # Rip:van Winkle:12345
```

Die Variable `$"` legt das Trennzeichen für die Elemente aus Arrays fest, die in doppelten Anführungszeichen ausgegeben werden (per Voreinstellung enthalt `$"` das Leerzeichen).

```
$" = ":";
@array = ("Rip", "van Winkle", 12345);

print @array;     # Rip:van Winkle:12345
```

$| Pufferung

Ausgaben, die mit `print` oder `printf` getätigt werden, sind automatisch an den Datei-Handle `STDOUT` gerichtet, der mit dem Konsolenbildschirm verbunden ist. Zu beachten ist dabei, dass diese Ausgabe unter Unix/Linux gepuffert ist, d.h. die Ausgabe wird in einem internen Puffer zwischengespeichert und erst ausgegeben, wenn ein Neue-Zeile-Zeichen ausgegeben, eine Eingabe erwartet (Koppelung von Ausgabe und Eingabe) oder das Programm beendet wird.

Um dieses Verhalten zu deaktivieren, genügt es, der Standardvariablen `$|` den Wert 1 zuzuweisen.

Fehlermeldungen, die grundsätzlich nicht gepuffert sein sollten, gibt man allerdings üblicherweise gar nicht erst auf `STDOUT`, sondern auf `STDERR` aus. `STDERR` ist ein weiterer vordefinierter Datei-Handle, der nie gepuffert ist.

```
print STDERR "\nHier ist etwas falsch gelaufen\n";
```

Die konsequente Fehlerausgabe auf `STDERR` hat zudem den Vorteil, dass man mit dem Suchbefehl des Editors schnell zu den verschiedenen Fehlerausgaben in einem Skript springen kann.

9.1.3 Der Report-Generator write

Die Erzeugung tabellarischer Ausgaben ist in den meisten Programmiersprachen ein Greuel, weil man dem Code, der die tabellarische Ausgabe erzeugt, nie ansehen kann, wie die Ausgabe später aussehen wird. Also heißt es: Code aufsetzen, Ausprobieren, Code ändern, Ausprobieren, Code noch mal ändern, Ausprobieren. Nicht so in Perl! Perl erlaubt Ihnen, die Ausgabe im Code so zu erzeugen, wie sie später aussehen wird.

Seitenaufbau der Report-Ausgabe

Der Report-Generator erzeugt eine seitenorientierte Ausgabe. Jede Seite besteht aus einem Kopfteil, der auf jeder Seite wiederholt wird, und einem Rumpfteil, der die fortlaufenden Daten enthält.

Wie kann man diesen Aufbau nutzen, um Daten aus einer Textdatenbank – beispielsweise einer CVS-Datei zur Verwaltung von CDs – in einer gut lesbaren Tabelle auszugeben?

Listing 9.4: Inhalt der CVS-Datei

```
"Guiseppe Verdi","La Traviata","Klassik"
"Sergei Rachmaninov","Klavierkonzert Nr. 2","Klassik"
"Peter I. Tchaikovsky","Klavierkonzert Nr. 1","Klassik"
```

Jeder Eintrag in der CVS-Datei steht für eine CD und enthält folgende Informationen: Komponist, Titel der CD und Kategorie.

Es bietet sich an, diese Begriffe als Spaltenüberschriften der Tabelle in den Kopfteil zu packen, so dass sie auf jeder neuen Seite oben angezeigt werden. Auch eine Überschrift und die Angabe der aktuellen Seite würden sich im Kopfteil gut machen.

Im Rumpfteil sollen dagegen untereinander die Daten aus den einzelnen Datensätzen (Zeilen der CVS-Datei) erscheinen.

Wie lässt sich dies realisieren?

Codierung des Kopf- und Rumpfteils

Kopf- wie Rumpfteil beginnen mit dem Befehl `format` und enden mit einem Punkt, der allein in einer Zeile steht. Kopf- und Rumpfteil müssen zudem einen Namen erhalten, der sich nach dem Datei-Handle richtet, in das die Ausgabe erfolgen soll. Der Rumpfteil heißt genauso wie das Datei-Handle (also beispielsweise `STDOUT` für Ausgaben auf die Konsole), der Kopfteil trägt den gleichen Namen wie der Rumpfteil plus dem Suffix `_TOP`.

```
format STDOUT_TOP =
.

format STDOUT =
.
```

Als Nächstes setzt man den auszugebenden Text auf, und zwar so, wie er bei der Ausgabe erscheinen soll. Wo aber bei der Ausgabe der Wert einer Variablen eingefügt werden soll, da verwendet man einen der Platzhalter aus Tabelle 9.1.

Tabelle 9.1: Platzhalter für write

Platzhalter	**Ausgabe**
@<<<<<<<<<<<	Linksbündige Ausgabe. Die Anzahl der <-Zeichen plus dem @-Zeichen geben die Länge des Ausgabefeldes an.
@>>>>>>>>>>>	Rechtsbündige Ausgabe. Die Anzahl der >-Zeichen plus dem @-Zeichen geben die Länge des Ausgabefeldes an.
@|||||||||||	Zentrierte Ausgabe. Die Anzahl der |-Zeichen plus dem @-Zeichen geben die Länge des Ausgabefeldes an.
@##.#######	Format für Zahlen.
@*	Mehrzeiliges Feld (siehe PERLFORM-Dokumentation)

```
format STDOUT_TOP =

Eingetragene CDs in @<<<<<<<<<<<                   Seite @>>

-------------------------------------------------------------
Komponist           Titel                           Kategorie
-------------------------------------------------------------
.

format STDOUT =
@<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<   @<<<<<<<<<<<
.
```

Zuletzt schreibt man unter die Zeilen, die Platzhalter enthalten (und zwar direkt in die darunter liegende Zeile!), die Variablen, die die einzufügenden Werte enthalten.

Für die aktuelle Seite kann man dabei die Standardvariable `$%` verwenden.

```
format STDOUT_TOP =

Eingetragene CDs in @<<<<<<<<<<<                   Seite @>>
                     $CD_db,                             $%

-------------------------------------------------------------
Komponist           Titel                           Kategorie
-------------------------------------------------------------
.

format STDOUT =
@<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<   @<<<<<<<<<<<
$comp, $titel, $kat
.
```

Ausgabe der Tabelle

Zuletzt muss man die Ausgabe noch erzeugen. Dazu implementiert man eine Schleife, in der die einzelnen auszugebenden Datensätze durchlaufen werden und die Funktion `write` aufgerufen wird.

```
sub liste_ausgeben {
  foreach $cd (@cds) {
    $comp = $cd->{'composer'};
    $titel = $cd->{'titel'};
    $kat = $cd->{'kategorie'};
    write STDOUT;;
    }
}
```

Obiger Code geht davon aus, dass der Inhalt der CVS-Datei zuvor von Hashreferenzen in ein Array `@cds` eingelesen wurde. In der `foreach`-Schleife wird das Array durchlaufen und für jedes Hash werden die einzelnen Schlüssel in die globalen Variablen `$comp`, `$titel` und `$kat` kopiert, die wir im Rumpfteil verwendet haben.

Man hätte im Rumpfteil auch direkt `$cd->{'composer'}`, `$cd->{'titel'}` und `$cd->{'kategorie'}` verwenden können.

Zum guten Schluss können Sie noch festlegen, wie viele Zeilen eine Seite der Ausgabe enthalten soll (`$=`), mit welcher Seitenzahl begonnen werden soll (`$%`) und wie viele Zeilen für diese Seite noch übrig sein sollen (`$-`).

Insgesamt sieht der Code dann wie folgt aus:[3]

```
format STDOUT_TOP =

Eingetragene CDs in @<<<<<<<<<<<                  Seite @>>
                    $CD_db,                             $%

-----------------------------------------------------------
Komponist           Titel                          Kategorie
-----------------------------------------------------------
.
```

3 Die Länge der Platzhalter ist zum Teil etwas gekürzt, um das Listing an die Seitenbreite des Buches anzupassen. Das vollständige Programm zu diesem Beispiel finden Sie in Kapitel 14.

```
format STDOUT =
@<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<   @<<<<<<<<<<<
$comp, $titel, $kat
.

sub liste_ausgeben {
  $- = 0;
  $% = 0;
  $= = 25;

  foreach $cd (@cds) {
    $comp = $cd->{'composer'};
    $titel = $cd->{'titel'};
    $kat = $cd->{'kategorie'};
    write STDOUT;;
    }
}
```

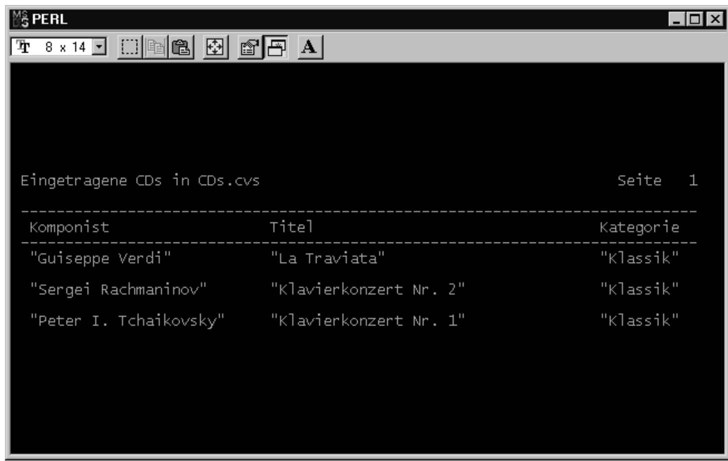
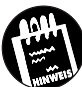


Abb. 9.3: Ausgabe des Report-Generators



Außer tabellarischen Ausgaben kann man mit `write` auch Serienbriefe oder Ähnliches erzeugen.

9.2 Dateien

Mindestens ebenso wichtig wie die Ein- und Ausgabe über die Konsole ist das Verarbeiten von Daten aus Dateien. Zur Programmierung mit Dateien gehört, dass man weiß, wie man Dateien öffnet, wie man Daten aus Dateien einlesen kann, wie man Daten in Dateien abspeichert und wie man die Dateien zum Schluss wieder schließt.

Eine äußerst einfache Methode zum Einlesen des Inhalts von Textdateien haben Sie bereits kennen gelernt.

Listing 9.5: skhan1.pl

```
#!/usr/bin/perl -w

while ($zeile = <>) {
  chomp ($zeile);
  print $zeile;
  <STDIN>;
  }
```

Aufruf: > **perl skhan1.pl skhan.chs** (Windows)

Aufruf: > **./skhan1.pl skhan.chs** (Linux)

Wenn Sie den Eingabeoperator ohne Angabe eines Datei-Handles aufrufen, können Sie dem Programm in der Kommandozeile eine Datei übergeben (oder auch mehrere), deren Inhalt dann vom Eingabeoperator zeilenweise eingelesen wird.

Diese praktische Methode hilft uns aber nicht weiter, wenn wir Daten in Dateien schreiben wollen, die Daten binär codiert sind oder das Skript immer mit ein und derselben Datei arbeit und die Angabe der Datei in der Kommandozeile somit überflüssig ist. In solchen Fällen muss man für die Datei ein eigenes Datei-Handle erzeugen.

9.2.1 Daten in Dateien schreiben

Die meisten Programmierbücher beginnen mit dem Lesen von Dateien und erläutern danach erst das Schreiben von Dateien. Das Lesen von Dateien ist weder schwerer noch leichter als das Schreiben von Dateien, setzt aber voraus, dass man eine Datei zur Verfügung hat, deren Inhalt man einlesen kann und deren inneren Aufbau man kennt. Wir könnten uns zu diesem Zweck auf irgendeine Datei einigen (beispielsweise die Logdatei eines Webservers oder eine Konfigurationsdatei des Betriebssystems), doch dann

wird es mit Sicherheit den einen oder anderen Leser geben, auf dessen Rechner die betreffende Datei nicht vorhanden ist. Also gehen wir umgekehrt vor und erzeugen zuerst einmal die Datei, die wir im nächsten Abschnitt lesen werden.

Das Schreiben einer Datei erfolgt in drei Schritten:

1. Datei öffnen und mit Datei-Handle verbinden.

Zum Öffnen von Dateien verwendet man die Funktion `open`.

```
my $dateiname = "daten.txt";
open(DATEI, ">> $dateiname");
```

Als Argumente übergibt man `open` einen Namen für den einzurichtenden Datei-Handle und den Namen der zu öffnenden Datei. Die Funktion richtet den gewünschten Datei-Handle ein (Datei-Handles stellen unter Perl einen eigenen Datentyp dar), öffnet die angegebene Datei und verbindet diese mit dem Datei-Handle. In unserem Skript wird die Datei jetzt durch den Datei-Handle repräsentiert; alle nachfolgenden Zugriffe auf die Datei erfolgen über den Datei-Handle.

Der Dateiname kann direkt als String oder als Variable angegeben werden. Übersichtlicher ist es, wenn man den Dateinamen am Anfang des Skripts einer Variablen zuweist. (Dies hat auch den Vorteil, dass man das Skript schnell zum Öffnen einer anderen Datei umschreiben kann.)

Wenn die zu öffnende Datei nicht im selben Verzeichnis wie das Perl-Skript steht, müssen Sie im Dateinamen auch den Pfad zur Datei angeben. Windows-Anwender, die es gewohnt sind, Verzeichnisse mit einem Backslash \ zu trennen, müssen beachten, dass der Backslash in doppelten Anführungszeichen ein Sonderzeichen ist. Um einen Backslash in einen Dateinamen einzubauen, muss man daher zwei Backslashes hintereinander schreiben (oder einfache Anführungszeichen verwenden): `"C:\\datei.txt"`. Die meisten Windows-Versionen erlauben aber auch die Verwendung eines einfachen Slash, `"C:/datei.txt"`, wie es auch unter Unix/Linux üblich ist.

Soweit ist alles ganz einfach. Etwas störend ist nur die Zeichenfolge `>>` vor dem Dateinamen (bzw. der Variablen, die den Dateinamen enthält). Diese Zeichenfolge gibt an, zu welchem Zweck die Datei geöffnet werden soll.

Tabelle 9.2: Öffnen-Modi

Öffnen-Modus	Beschreibung
`open(FH, "datei");`	Öffnet die Datei zum Lesen.
`open(FH, "< datei");`	Ist die Datei nicht vorhanden, scheitert die Funktion.
`open(FH, "> datei");`	Öffnet die Datei zum Schreiben.
	Ist die Datei nicht vorhanden, wird sie neu angelegt. Ist die Datei bereits vorhanden, wird ihr alter Inhalt gelöscht.
`open(FH, ">> datei");`	Öffnet die Datei zum Anhängen.
	Ist die Datei nicht vorhanden, wird sie neu angelegt. Ist die Datei bereits vorhanden, wird der neue Inhalt an den alten Inhalt angehängt.
`open(FH, "+< datei");`	Öffnet die Datei zum Lesen und Schreiben.
	(Erfordert meist fortgeschrittene Funktionen zur Dateibehandlung).

Fehlerbehandlung

Zum Schluss müssen wir noch klären, was geschehen soll, wenn die `open`-Funktion scheitert und die Datei nicht mit dem Datei-Handle verbunden wird. Nachfolgende Zugriffe auf den Datei-Handle sollten dann unbedingt unterbunden werden. Zu diesem Zweck könnte man den `open`-Aufruf in die Bedingung einer `if-else`-Konstruktion einbauen (die Funktion `open` liefert `undef` (entspricht dem Wahrheitswert »falsch«) zurück, wenn sie scheitert). Wenn das Programm ohne die Datei aber gar nicht sinnvoll arbeiten kann und daher nach dem Scheitern von `open` beendet werden sollte, empfiehlt es sich, den `open`-Aufruf an den Anfang des Skripts zu stellen und mit einem `die`-Aufruf zu kombinieren:

```
open(DATEI, ">> $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";
```

Diese Konstruktion sieht man in Perl-Skripten, die mit Dateien arbeiten, sehr häufig. Sie funktioniert wie folgt:

- ✘ Wenn `open` die Datei öffnen konnte, liefert sie einen Wert zurück, der dem Wahrheitswert »wahr« entspricht. Da eine OR-Konstruktion bereits »wahr« ist, wenn nur einer der Teilausdrücke »wahr« ist, wird der nachfolgende Ausdruck mit `die` überhaupt nicht mehr ausgewertet (`die` wird nicht aufgerufen) und das Skript wird normal fortgesetzt.
- ✘ Wenn `open` scheitert, liefert `open` den Wahrheitswert »false«, so dass der Interpreter auch noch den zweiten Ausdruck der OR-Verknüpfung aus-

wertet. Er ruft also die Funktion `die` auf. Die Funktion `die` gibt noch den als Argument übergebenen String aus und beendet dann sofort das Programm.

2. Daten in Datei schreiben.

Für die Ausgabe von Daten in Dateien verwendet man die gleichen Funktionen wie für die Ausgabe von Daten auf die Konsole: `print`, `printf` oder `write`. Der einzige Unterschied ist, dass man den Funktionen als erstes Argument den Datei-Handle übergibt.

```
print DATEI $var1, $var2;
```

Wenn Sie die Argumente zu einer Dateifunktion nicht in Klammern einfassen, wird der Datei-Handle nicht durch Komma von den anderen Argumenten getrennt.

3. Datei schließen.

Zum Schluss wird die Datei geschlossen.

```
close(DATEI);
```

Das Schließen der Datei führt dazu, dass eventuell noch gepufferte Daten in die Datei geschrieben werden und die Datei mit dem Dateiendezeichen abgeschlossen wird. Wenn Sie das Programm mit geöffnetem Datei-Handle beenden, kann es passieren, dass Daten verloren gehen oder die Datei beschädigt wird.

Beispiel

Listing 9.6: schreiben.pl

```
#!/usr/bin/perl -w
use strict;

my @persondaten = ();
my $dateiname = "daten.txt";   # Name der zu öffnenden Datei


# Datei öffnen und mit Datei-Handle DATEI verbinden

  open(DATEI, ">> $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";
```

```
# Daten von Konsole einlesen

  my $eingabe = "";
  print "\n";
  print "Geben Sie Ihren Vornamen ein: ";
  chomp($eingabe = <STDIN>);
  push (@persondaten, $eingabe);

  print "Geben Sie Ihren Nachnamen ein: ";
  chomp($eingabe = <STDIN>);
  push (@persondaten, $eingabe);

  print "Geben Sie Ihr Alter ein: ";
  chomp($eingabe = <STDIN>);
  push (@persondaten, $eingabe);


# Daten in Datei schreiben

  $, = "\t";                # Felder durch Tabs trennen
  $\ = "\n";                # Ausgabe mit \n beenden
  print DATEI @persondaten; # Daten in Datei schreiben


# Datei schliessen

  close(DATEI);
```

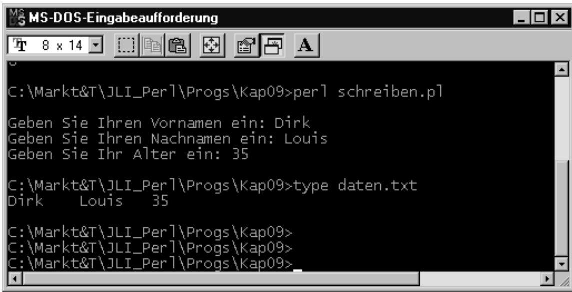


Abb. 9.4: Ausführung und Ausgabe des Skripts schreiben.pl

Überschreiben verhindern

Wenn Sie Dateien mit > zum Schreiben öffnen, kann es schnell passieren, dass der Inhalt bestehender Dateien ungewollt gelöscht wird. Sie können dies verhindern, indem Sie im Programm abfragen, ob die Datei, in die geschrieben werden soll, bereits existiert. Perl definiert hierfür (und für eine Reihe weiterer Fragen zu Dateien, siehe PERLFUNC-Dokumentation unter

Eintrag -X) einen eigenen Test, den man in einer `if`-Bedingung verwenden kann: `(-e "Dateiname")`.

```
if (-e $dateiname) {
     print "Datei existiert bereits. Ueberschreiben (j/n)? ";
     chomp($_ = <STDIN>);
     $_ = lc;
     if ($_ eq 'n') { die; }
     }

open(DATEI, "> $dateiname")
   ...
```

9.2.2 Daten aus Dateien einlesen

Das Lesen einer Datei erfolgt in drei Schritten:

1. Datei öffnen und mit Datei-Handle verbinden.

Zum Öffnen von Dateien verwendet man wie beim Schreiben die Funktion `open`. Zu beachten ist nur, dass man zum Öffnen im Lese-Modus das <-Zeichen vor den Dateinamen stellt (oder nur den Dateinamen angibt, siehe Tabelle 10.2).

```
my $dateiname = "daten.txt";
open(DATEI, "< $dateiname");
```

2. Daten aus Datei lesen.

Wie man die Daten einliest, hängt davon ab, wie der Inhalt der Datei aufgebaut ist.

Handelt es sich um eine Textdatenbank oder allgemein um eine Datei, deren einzelne Zeilen durch bestimmte Trennzeichen in Felder aufgeteilt sind, können Sie den Dateiinhalt mit dem <>-Operator zeilenweise einlesen, die einzelnen Zeilen mit Hilfe der `split`-Funktion in Felder auftrennen und deren Inhalte Variablen (oder Array-Elementen) zuweisen.

```
while(my $zeile = <DATEI>)
    {
    chomp($zeile);
    @persondaten = (split("\t", $zeile));
...
```

Enthält die Datei einen Fließtext, den Sie durchsuchen wollen (beispielsweise mit Hilfe regulärer Ausdrücke, siehe Kapitel 10), haben Sie die Möglichkeit, den Text

- ✘ zeilenweise (`while($zeile = <HANDLE>)`)
- ✘ absatzweise (`$/=''; while($absatz = <HANDLE>)`)
- ✘ oder vollständig (`undef $/; $text = <HANDLE>;`)

einzulesen und zu durchsuchen.

Neben dem <>-Operator kann man auch die Funktionen `readline` oder `read` zum Einlesen der Daten verwenden.

3. Datei schließen.

Zum Schluss wird die Datei geschlossen.

```
close(DATEI);
```

Beispiel

Listing 9.7: lesen.pl

```
#!/usr/bin/perl -w
use strict;

my @persondaten = ();
my $dateiname = "daten.txt";   # Name der zu öffnenden Datei


# Datei öffnen und mit Datei-Handle DATEI verbinden

  open(DATEI, "<$dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";

# Daten aus Datei einlesen

  while(my $zeile = <DATEI>)
    {
    chomp($zeile);
    @persondaten = (split("\t", $zeile));

    # Daten auf Konsole ausgeben
    $, = "\t";                 # Felder durch Tabs trennen
    $\ = "\n";                 # Ausgabe mit \n beenden
    print @persondaten;
    }


# Datei schliessen

  close(DATEI);
```

9.2.3 Binärmodus und Binärdateien

Bisher haben wir uns nur mit dem Lesen und Schreiben von Textdateien beschäftigt. Textdateien sind Dateien, deren Inhalt aus einer Folge von Zeichen besteht. Wenn Sie in eine solche Datei den Inhalt einer Variablen mit einem numerischen Wert ausgeben:

```
$var = 26;
print <DATEI> $var;
```

wird in der Datei nicht die Zahl 26 gespeichert, sondern die Ziffernfolge "26". Statt einer Binärzahl, deren Wert der Dezimalzahl 26 entspricht (11010), stehen in der Datei die ASCII-Codes (oder UNICODES) der Zeichen.

Der Inhalt einer solchen Datei kann nicht nur mit einem Perl-Skript, sondern mit jedem Texteditor gelesen werden.

Binärdateien

Möchte man verhindern, dass der Inhalt einer Datei mit einem beliebigen Texteditor eingesehen werden kann, besteht die Möglichkeit, Daten nicht als Zeichen, sondern in Binärcode auszugeben. Dazu wandelt man die auszugebenden Daten mit Hilfe der Perl-Funktion `pack` in einen String um, der die Bitfolge der zugehörigen Binärcodierung enthält – beispielsweise:

```
$binstr = pack("N", $var);    # wandelt Ganzzahlwert in $var
                              # in Binärformat und gibt die
                              # Bitfolge als String zurück.
```

Will man binär geschriebene Daten wieder einlesen, muss man wissen, wie viele Byte die einzelnen Daten belegen (wenn Sie Ganzzahlen mit `pack("N"..)` codieren, können Sie auf den meisten Systemen von 4 Byte ausgehen). Mit Hilfe der Funktion `read` können Sie die Bytes einlesen und in einer Variablen speichern. Mit `unpack` können Sie die Bitfolge zurückverwandeln.

```
read(DATEI, $var, 4);
$str = unpack("N", $var);
```

Binärmodus

Vielleicht ist Ihnen beim Durchsuchen der PERLFUNC-Dokumentation bereits die Funktion `binmode` aufgefallen und Sie haben sich gefragt, ob man mit dieser Funktion nicht vielleicht Daten automatisch binär codiert ausgeben könnte? Tja, da muss ich Sie enttäuschen. Die Funktion `binmode` ist zwar für Binärdateien gedacht, erzeugt aber keine binären Daten.

Bisher haben Sie den Zeilenumbruch nur als das Neue-Zeile-Zeichen `\n` kennen gelernt. Leider wird der Zeilenumbruch auf unterschiedlichen Betriebssystemen unterschiedlich repräsentiert. Unter Unix/Linux ist es tatsächlich das Zeichen `\n`, unter MAC das Zeichen `\r` und unter VMS oder MSDOS/WINDOWS die Kombination aus `\r\n`.

Um den Programmierer von der lästigen Unterscheidung zwischen diesen verschiedenen Zeilenumbruchdarstellungen zu befreien, sorgen Ihr Perl-Interpreter und die Perl-Funktionen zur Dateiein- und -ausgabe dafür, dass bei der Ausgabe `\n` automatisch in die betriebssysteminterne Repräsentation konvertiert und bei der Eingabe die betriebssysteminterne Repräsentation automatisch in `\n` umgewandelt wird.[4]

Diese automatische Konvertierung, die für Textdateien wunderbar funktioniert, kann zu groben Fehlern führen, wenn man versucht, Binärdateien zu schreiben oder zu lesen. Daher sollte man beim Lesen oder Schreiben von Binärdateien nach dem `open`-Aufruf stets `binmode(<HANDLE>)` aufrufen.

Listing 9.8: lesen2.pl

```
#!/usr/bin/perl -w
use strict;

my $dateiname = "progr.exe";

open(DATEI, "< $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";

binmode(DATEI);

while(read(DATEI,my $zeile, 4) )
    {
    chomp($zeile);
    print $zeile;
    }

close(DATEI);
```

Was passiert eigentlich, wenn man Binärdaten einliest und in eine Textdatei oder auf Konsole schreibt? Dann interpretieren der Texteditor oder die Konsole die binären Daten als ASCII-Codes von Zeichen. Die Folge ist, dass man einen unentwirrbaren Zeichensalat sieht – gelegentlich unterbrochen von lesbarem Text, wenn die Binärdatei auch echten Text enthielt.

4 Gleiches gilt für das Dateiendezeichen, das auf manchen Betriebssystemen ebenfalls abweichend behandelt wird.

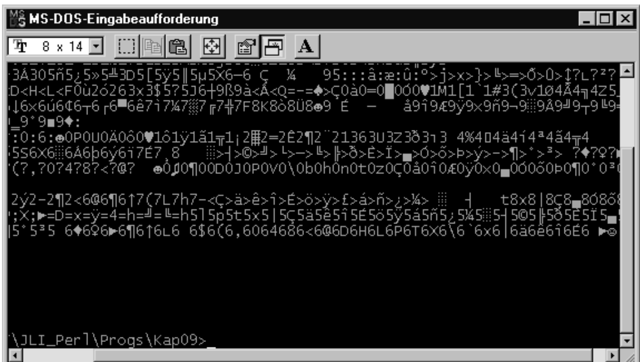


Abb. 9.5: Binäre Daten, die als Text interpretiert werden

9.3 Kommandozeilenargumente

Wenn Sie viel mit der Konsole und Konsolenprogrammen arbeiten, werden Sie aus eigener Erfahrung wissen, dass Konsolenprogramme oftmals mit Argumenten (Dateinamen, Schalter etc.) aufgerufen werden, die über die Kommandozeile übergeben werden, und Sie werden sich fragen, ob dies nicht auch mit Perl-Skripten möglich ist. Nun, es ist möglich.

Wenn einem Perl-Programm beim Aufruf von der Konsole Kommandozeilenargumente übergeben werden, speichert der Perl-Interpreter diese für Sie in der Standardvariablen `@ARGV`.

```
#!/usr/bin/perl -w
use strict;

foreach my $arg (@ARGV) {
  print "$arg\n";
  }
```

9.4 Umgebungsvariablen

Jedes Betriebssystem unterhält eine Reihe von Umgebungsvariablen, die zum größten Teil Informationen über die aktuelle Konfiguration des Systems enthalten (und von Programmen oder dem Anwender angepasst oder um weitere Umgebungsvariablen erweitert werden können). Da die einzelnen Umgebungsvariablen über einen Namen (beispielsweise PATH) und einen Wert verfügen, bietet es sich an, sie in Perl in Form eines Hash zu repräsentieren. Dieses Hash heißt `%ENV`.

Wenn Sie den Wert einer Umgebungsvariablen abfragen wollen und den Namen der Umgebungsvariablen kennen, brauchen Sie den Namen nur als Schlüssel zum Hash zu verwenden:

```
$pfad =  "$ENV{'PATH'}\n";
```

Wenn Sie sich erst einmal über die auf Ihrem System vorhandenen Umgebungsvariablen informieren wollen, tippen Sie auf Ihrer Konsole PRINTENV (Unix/Linux) oder SET (MSDOS-Eingabeauffoderung) ein oder führen das folgende Skript aus:

```
#!/usr/bin/perl -w
use strict;

foreach my $key (keys %ENV) {
  print $key, ": ", $ENV{$key}, "\n";
  }
```

9.5 Fragen und Übungen

1. Mit dem Eingabeoperator kann man einzelne Daten von der Konsole einlesen, man kann Dateien aus der Kommandozeile und Dateien über ein Datei-Handle einlesen. Wie wird der Eingabeoperator dazu im Skript aufgerufen?
2. In Übung 9 aus Kapitel 4 haben Sie ein Skript aufgesetzt, dass den Sinus zu verschiedenen Winkeln ausgibt. Schreiben Sie dieses Programm um, so dass es die Ausgabe mit Hilfe des Report-Generators `write` erzeugt.
3. Wie öffnet man eine Datei »demo.txt« zum Lesen, Schreiben, Anhängen?
4. Ändern Sie das Skript SCHREIBEN.PL so ab, dass der Name der zu öffnenden Datei über die Konsole abgefragt wird
5. Welches ist der kürzeste Weg, den Inhalt einer Datei auf die Konsole auszugeben?
6. Wie kann man den gesamten Inhalt einer Datei in einen String einlesen?
7. Wie kann man den Inhalt mehreren Dateien in einen String einlesen?

Kapitel 10

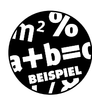
Suchen mit regulären Ausdrücken

Die regulären Ausdrücke oder, genauer gesagt, das Suchen nach Textstellen mit Hilfe regulärer Ausdrücke (Pattern Matching) ist eine der großen Stärken von Perl.

Mit Hilfe regulärer Ausdrücke können Sie Texte nach Wörtern, Textpassagen oder Zeichenmustern durchsuchen. Sie können prüfen, ob es ein Wort oder ein Muster in einem Text überhaupt gibt, Sie können das Wort oder Muster durch einen anderen Text ersetzen und Sie können die gefundenen Vorkommen in speziellen Variablen zurückliefern. Kurz und gut, mit Perls regulären Ausdrücken können Sie weit mehr machen, als die Suchen&Ersetzen-Befehle der meisten Textverarbeitungsprogramme erlauben.

Im Einzelnen lernen Sie,

- ✘ wie man nach Wörtern oder Textpassagen sucht
- ✘ wie man mit Mustern (regulären Ausdrücken) nach alternativen oder variablen Zeichenfolgen sucht
- ✘ welche Optionen es zur Einstellung der Pattern-Matching-Operatoren gibt
- ✘ wie man Textpassagen ersetzt
- ✘ wie man gefundene Textpassagen weiter verarbeiten kann



Folgende Elemente lernen Sie kennen

Die Pattern-Matching-Operatoren `m//` und `s//`, die Standardvariablen `$1`, `$2`...

10.1 Nach Wörtern suchen

Beginnen wir mit einem ganz einfachen Beispiel, um ein Gefühl für die Arbeit mit regulären Ausdrücken zu bekommen. Nehmen wir an, Sie haben auf Ihrem Rechner einen Text vorliegen, in dem die berühmtesten Giftmorde beschrieben sind. Nun haben Sie kürzlich von dem Mordfall »Madeleine Smith« gehört und wollen nachschauen, ob in Ihrem Text auch etwas über diese vermutliche Giftmörderin zu finden ist. Nehmen wir weiter an, dass es in dem Text tatsächlich einen Eintrag zu Madeleine Smith gibt, der wie folgt beginnt und von dessen Existenz Sie ja eigentlich gar nichts wissen.

```
...
Madeleine Smith

Der Prozess gegen Madeleine Smith wegen der Ermordung ihres
Liebhabers Pierre Emile L´Angelier durch Verabreichung von
Arsenik endete am Donnerstag, dem 9. Juli 1857, mit dem
Richterspruch "Schuldbeweis nicht erbracht", und sie verließ
den High Court of Justiciary in Edinburgh durch einen
Nebeneingang als freier Mensch.
...
```

Die Frage ist nun, wie man ein Perl-Skript aufsetzt, dass den Suchbegriff »Madeleine Smith« in dem Text findet.

Zuerst einmal lesen wir den Text ein. Wir übernehmen die Textdatei über die Kommandozeile entgegen, so dass wir ihren Inhalt in unserem Skript einfach mit `<>` einlesen können (siehe Kapitel 9.1.1). Des Weiteren aktivieren wir den Schlürfmodus des Eingabeoperators, indem wir die Standardvariable `$/` auf einen undefinierten Wert setzen.

```
#!/usr/bin/perl -w
use strict;

undef $/;
my $text;

$text = <>;
```

Jetzt ist der gesamte Inhalt der über die Kommandozeile angegebenen Datei in der Variablen `$text` zwischengespeichert. Das nächste Problem besteht darin, in diesem Text die Zeichenfolge »Madeleine Smith« zu finden. Ohne reguläre Ausdrücke und den mit Ihnen verbunden Suchmechanismus müssten wir den String jetzt Zeichen für Zeichen durchgehen und bei jedem »M« prüfen, ob diesem die Zeichenfolge »adeleine Smith« nachfolgt. Dank der Perl-eigenen Unterstützung für reguläre Ausdrücke und Mustererkennung (Pattern Matching) können wir stattdessen auf den Pattern-Matching-Operator `m//` zurückgreifen.

```
$text =~ m/Madeleine Smith/;
```

Analyse

Der String, der durchsucht werden soll, wird dem Pattern-Matching-Operator mit dem =~-Operator vorangestellt. Die Zeichenfolge, nach der gesucht wird, steht zwischen den /-Strichen des Operators. Der gesamte Ausdruck liefert den Wert »wahr« zurück, wenn der Suchbegriff im String gefunden wurde, und kann daher in Bedingungen verwendet werden.

Die /-Striche sind übrigens nicht verbindlich. Wenn der Suchbegriff ebenfalls Schrägstriche enthält, ist es meist einfacher, die Schrägstriche des Operators auszutauschen (`m{/verzeichnis/datei}`) als den Schrägstrichen im Suchbegriff den Escape-Operator voranzustellen (`m/\/verzeichnis\/datei}`).

Außer dem Schrägstrich / gibt es noch eine ganze Reihe weiterer Zeichen, denen in regulären Ausdrücken eine besondere Bedeutung zukommt und die wir im Laufe dieses Kapitels noch kennen lernen werden. Wenn diese Zeichen als normale Textzeichen interpretiert werden sollen, muss man Ihnen den Backslash voranstellen: `^ . ? { } [ ] ( ) / + * $ |`.

Das vollständige Skript könnte also wie folgt aussehen:

Listing 10.1: gifte1.pl

```
#!/usr/bin/perl -w
use strict;

undef $/;
my $text;

$text = <>;
```

```
if ($text =~ m/Madeleine Smith/) {
    print "Madeleine gefunden\n";
    }
```

Aufruf: > **perl gifte1.pl gifte.txt** (Windows)

Aufruf: # **./gifte1.pl gifte.txt** (Linux)

Ausgabe:

```
Madeleine gefunden
```

Pattern-Matching-Optionen

Etwas unbefriedigend an dem obigen Skript ist, dass es den Suchbegriff offensichtlich nur ein einziges Mal findet, obwohl der Begriff ja mindestens zweimal im Text enthalten ist. Dies liegt daran, dass der Pattern-Matching-Operator so eingestellt ist, dass er gleich beim ersten Treffer die Suche abbricht und zurückkehrt. Man kann dieses Verhalten aber ändern, indem man die Option `g` (für globale Suche) an den abschließenden Schrägstrich des Operators anhängt.

Listing 10.2: gifte2.pl

```
#!/usr/bin/perl -w
use strict;

undef $/;
my $text;

$text = <>;

while ($text =~ m/Madeleine Smith/g) {
    print "Madeleine gefunden\n";
    }
```

Aufruf: > **perl gifte2.pl gifte.txt**

```
Madeleine gefunden
Madeleine gefunden
```

Dank der angehängten Option `g` kann man den Pattern-Matching-Ausdruck in einer `while`-Bedingung aufrufen. Jedes Mal, wenn der Pattern-Matching-Operator ein Vorkommen des Suchbegriffs findet, liefert er den Wahrheitswert »wahr« zurück, und die `while`-Schleife wird ausgeführt. Beim nachfolgenden Überprüfen der Schleifenbedingung setzt der Operator seine Suche dann an der aktuellen Position im String fort. Trifft der Operator auf das Ende des zu durchsuchenden Strings, liefert er »falsch« zurück, und die Schleife wird beendet.

Wenn Sie den Pattern-Matching-Ausdruck ohne die Option g in der Schleife aufrufen, liefert der Ausdruck stets »wahr« zurück, weil der Operator den Suchstring bei jeder Überprüfung der Schleifenbedingung von Beginn an neu durchsucht. Die Folge ist, dass die Schleife endlos ausgeführt wird.

Neben der Option g gibt es noch weitere Optionen, die man an den Operatoraufruf anhängen und mit denen man die Arbeit des Pattern-Matching-Operator anpassen kann:

Tabelle 10.1: Die wichtigsten m//-Optionen

Option	**Bedeutung**
g	globale Suche, die nacheinander alle Vorkommen findet
i	keine Unterscheidung zwischen Groß- und Kleinschreibung
m	Suche erkennt \n-Zeichen im String als Zeilenende (betrifft die Metazeichen ^ und $, die den Zeilenanfang und das Zeilenende repräsentieren)
s	Suche über Zeilenende hinweg (betrifft das Metazeichen ».«, das ansonsten alle Zeichen außer dem Neue-Zeile-Zeichen \n repräsentiert; ist die Option s gesetzt, umschließt . auch das \n-Zeichen (siehe Listing links.pl weiter unten in diesem Kapitel)).

Zeilenweise durchsuchen

Jetzt wissen wir, dass im Text auch der Fall der Madeleine Smith angesprochen wird (oder zumindest der Name »Madeleine Smith« erwähnt wird). Das Skript wäre allerdings noch hilfreicher, wenn es auch gleich die Zeilennummern der gefundenen Vorkommen anzeigen würde.

Dazu müssen wir den Text zeilenweise einlesen und die Zeilen in einer while-Schleife verarbeiten. Die g-Option können wir uns sparen, da der Pattern-Matching-Operator jetzt auf einzelne Zeilen angewendet wird und es für uns nicht von Interesse ist, ob es in einer Zeile mehrere Vorkommen des Suchbegriffs gibt. Das resultierende Skript sieht wie folgt aus:

```
#!/usr/bin/perl -w
use strict;

my $zeilennr = 0;
my $zeile    = "";

while ($zeile = <>) {
  $zeilennr++;
  if ($zeile =~ m/Madeleine Smith/) {
    print "Madeleine gefunden in Zeile $zeilennr\n";
    }
  }
```

Aufruf: > **perl gifte3.pl gifte.txt**

Madeleine gefunden in Zeile 3

Madeleine gefunden in Zeile 5

Kürzungen

Das gesamte Skript kann man allerdings auch kürzer formulieren, damit es etwas von dem typischen Perl-Flair vermittelt. Als Erstes verzichten wir auf die Variable `$zeile` und lassen die Zeilen stattdessen in `$_` einlesen. Gleichzeitig verzichten wir auf `strict` – das Skript ist übersichtlich genug, dass wir auch ohne `strict` keine Fehler bei der Verwendung der Variablen machen.

```
#!/usr/bin/perl -w

my $zeilennr = 0;

while (<>) {
  $zeilennr++;
  if ($_ =~ m/Madeleine Smith/) {
    print "Madeleine gefunden in Zeile $zeilennr\n";
    }
  }
```

Auch der Pattern-Matching-Ausdruck kann vereinfacht werden. Gibt man keine zu durchsuchende Variable an, wird automatisch `$_` durchsucht. Den Teil `$_ =~` kann man sich also sparen. Zudem ist das führende m optional. Es wird nur dann wirklich benötigt, wenn man den Suchbegriff statt in Schrägstriche in irgendwelche anderen Zeichen kleiden will. Das gesamte Skript vereinfacht sich somit zu:

```
#!/usr/bin/perl -w

my $zeilennr = 0;

while (<>) {
  $zeilennr++;
  if (/Madeleine Smith/) {
    print "Madeleine gefunden in Zeile $zeilennr\n";
    }
  }
```

Schließlich können wir noch die `if`-Bedingung umformulieren (siehe Kapitel 4.6).[1]

1 Den Variablennamen und die Ausgabe habe ich gekürzt, damit die print-Zeile auch im Buch in eine Zeile passt.

Listing 10.3: gifte3.pl

```
#!/usr/bin/perl -w

my $z_nr = 0;

while (<>) {
  $z_nr++;
  print "Madeleine gefunden ($z_nr)\n" if /Madeleine Smith/
  }
```

Na, das schaut doch gleich viel perliger aus!

10.2 Mit Mustern suchen

Wenn Sie bisher nur mit Programmiersprachen wie C/C++ oder Java programmiert haben, wird Sie der Pattern-Matching-Operator sicherlich schon mächtig beeindruckt haben (mir ging es jedenfalls bei der ersten Begegnung mit diesem Operator so). Wenn Perl jedoch Ihre erste Programmiersprache ist, werden Sie vielleicht denken »Na ja, alles schön und gut, aber der Suchen-Befehl meines Textverarbeitungsprogramms kann mehr.« Wenn dem so ist, dann liegt dies vermutlich daran, dass Sie mit Ihrem Textverarbeitungsprogramm nicht nur nach konkreten Wörtern oder Textpassagen suchen können, sondern auch nach regulären Ausdrücken.

Reguläre Ausdrücke sind Suchmuster, die neben normalen Zeichen auch Sonderzeichen und Platzhalter für bestimmte Zeichengruppen oder Zeichenwiederholungen enthalten können.

Vielleicht interpretiert Ihr Editor das Zeichen `*` in Suchbegriffen als eine beliebige Zeichenfolge und das `<`-Zeichen als einen Wortanfang. Dann würde die Suche nach `<ein*` alle Wörter finden, die mit »Zeichen« anfangen (beispielsweise »ein« und »einerlei«, aber nicht »mein«). Gleiches ist auch mit den regulären Ausdrücken von Perl möglich, auch wenn die Sonderzeichen und Platzhalter etwas anders lauten.

Alternative Zeichen und Zeichenfolgen

Kommen wir noch einmal auf das Beispiel mit der mutmaßlichen Giftmörderin zurück. Nehmen wir an, einer Ihrer Bekannten hat Ihnen von dem Fall der Madeleine Smith erzählt. Vertieft in das Gespräch haben Sie natürlich nicht daran gedacht, dass Sie später ein Perl-Skript zur Suche nach »Madeleine Smith« aufsetzen würden, und folglich haben Sie Ihren Bekannten auch nicht gefragt, wie man »Madeleine« eigentlich schreibt, ob französisch »Madeleine« oder vielleicht doch englisch »Madeline«? Wenn die Dame aber Madeline geschrieben wird, kann man mit dem Suchbegriff »Madeleine« nicht fündig werden. Man muss also beide alternativen Schreibweisen

abfangen – beispielsweise durch OR-Verknüpfung zweiter Pattern-Matching-Ausdrücke:

```
if( ($_ =~ m/Madeleine Smith/) || ($_ =~ m/Madeline Smith/))
```

Besser und einfacher ist es allerdings, beide alternativen Schreibweisen in einem Ausdruck anzugeben.

Alternativen werden mit dem Metazeichen | angegeben. Im einfachsten Fall gibt man die Alternativen vollständig an.

```
if( $_ =~ m/Madeleine Smith|Madeline Smith/ )
```

Wenn die alternativen Suchbegriffe größtenteils übereinstimmen, kann man sich Tipparbeit ersparen, indem man nur die wirklich unterschiedlichen Zeichenfolgen angibt – diese dann aber in runde Klammern setzt.

```
if( $_ =~ m/Madel(ei|i)ne/ )
```

Das i kann man sogar auch noch streichen; Perl fasst eine leere Alternative automatisch als »kein Zeichen« auf.

```
if( $_ =~ m/Madel(e|)ine Smith/ )
```

Selbstverständlich erlaubt Perl auch die Angabe mehrerer Alternativen. So könnte man im Beispiel neben der französischen und englischen auch noch die amerikanische Schreibweise (Madelain) abfangen:

```
if( $_ =~ m/Madel(eine|ine|ain) Smith/ )
```

Schließlich gibt es noch die Möglichkeit, für ein einzelnes Zeichen eine Reihe von Alternativen anzugeben. Diese werden dann in eckige Klammern gesetzt. Wenn Sie beispielsweise nach 1., 2., 3. und 4. suchen, können Sie schreiben:

```
m/(1|2|3|4)\./  # dem Punkt muss ein Backslash vorangehen,
                # da er sonst ein Metazeichen darstellt
```

oder

```
m/[1234]\./
```

oder

```
m/[1-4]\./      # findet: 1., 2., 3. und 4.
```

Wenn Sie an allen Ziffern von 0 bis 9 interessiert sind, brauchen Sie nicht einmal mit `[0-9]` eine eigene Zeichengruppe zu definieren, da diese bereits definiert ist: `\d`.

```
m/\d\./  # entspricht m/[0-9]\./
```

Tabelle 10.2: Zeichengruppen

Zeichengruppe	Bedeutung
`(al|el|il)`	alternative Zeichenfolgen
`(al|el|)`	alternative Zeichenfolgen oder keine Zeichen
`[abc%]`	alternative Zeichen: a, b, c, %
`[a-f]`	alternative Zeichen: a, b, c, d, e oder f
`[a-zA-Z]`	alternative Zeichen: jeder Buchstabe
`[^a-zA-Z]`	alternative Zeichen: jedes Zeichen, das kein Buchstabe ist
	Das ^-Symbol negiert die Zeichengruppe. Normalerweise stehen die eckigen Klammern als Platzhalter für eines der Zeichen aus der Zeichengruppe. Das ^-Symbol in den eckigen Klammern bedeutet, dass die eckigen Klammern als Platzhalter für ein Zeichen stehen, dass gerade nicht in der Zeichengruppe aufgeführt sein darf.
`\d`	entspricht [0-9] (eine Ziffer)
`\D`	entspricht [^0-9] (keine Ziffer)
`\w`	entspricht [0-9a-zA-Z_] (ein Wortzeichen)
	Wortzeichen sind also sämtliche Buchstaben, sämtliche Ziffern und _
`\W`	entspricht [^0-9a-zA-Z_] (kein Wortzeichen)
`\s`	entspricht [`\t\n\r\f`] (ein Whitespace-Zeichen)
	Whitespace-Zeichen sind Zeichen, die Leerräume erzeugen: Leerzeichen, Tabulator, Zeilenumbruch, Wagenrücklauf, Seitenumbruch
`\S`	entspricht [^ `\t\n\r\f`] (kein Whitespace-Zeichen)
`.`	ein beliebiges Zeichen außer `\n` (wenn die Option s angehängt ist, schließt . auch das `\n`-Zeichen mit ein)

Zeichenwiederholungen

Alternative Zeichen sind ein wichtiger Bestandteil vieler regulärer Ausdrücke. Mindestens ebenso interessant sind die Wiederholungen von einem oder mehreren Zeichen.

Nehmen wir an, Sie suchen in einem Text nach dem Wort »Nummerierung«, wobei Sie auch die alte Schreibweise »Numerierung« abfangen wollen. Statt nach alternativen Zeichen zu suchen:

```
m/Nummerierung|Numerierung/ oder m/(Nu(mm|m)erierung)/
```

können Sie im regulären Ausdruck angeben, dass das »m« ein- oder zweimal im Suchbegriff vorkommen darf:

```
m/(Num{1,2}erierung)/
```

`{n, m}` ist einer von mehreren Quantifizierern, die angeben, wie oft das vorangehende Zeichen (das auch als Zeichengruppe angegeben werden kann) wiederholt werden soll oder kann (siehe Tabelle 10.3).

Tabelle 10.3: Quantifizierer

Quantifizierer	Bedeutung
`{n,m}`	mindestens n, höchstens m Wiederholungen
`{n,}`	mindestens n Wiederholungen
`{n}`	genau n Wiederholungen
`*`	keinmal oder mehrmals (entspricht {0,}
`+`	einmal oder mehrmals (entspricht {1,}
`?`	einmal oder keinmal (entspricht {0,1}

Gierige und nicht gierige Quantifizierer

Das Problem mit den Quantifizierern ist, dass sie gierig alle Zeichen schlucken, die sie finden können. Wenn Sie beispielsweise den folgenden Text:

```
$text = "Rantanplan";
```

mit folgendem Muster durchsuchen:

```
if ($text =~ m/^(R\w*n)/)
```

finden Sie »Rantanplan« und nicht etwa »Ran«.

Der Grund hierfür ist, dass der `\w*`-Ausdruck so lange Zeichen liest, wie es geht. Da der gesamte String nur aus Wortzeichen besteht, liest der Ausdruck den gesamten String ein. Als Abschluss sieht das Muster ein »n« vor. Jetzt erkennt der Pattern-Matching-Operator, dass er vermutlich zu viele Zeichen eingelesen hat. Da er auch weniger Zeichen für `\w*` hätte einlesen können, geht er jetzt Zeichen für Zeichen zurück und sucht nach einem »n«. Lange zu suchen braucht er nicht; gleich das erste Zeichen, um das er zurückgeht, ist ein »n«. Jetzt hat er das Muster gefunden: »Rantanplan«.

Wenn man möchte, dass die Quantifizierer nicht so viele Zeichen wie möglich einlesen, sondern aufhören, wenn der nächste Teil des Musters beginnt, müssen Sie die »nicht gierigen« Versionen der Quantifizierer aufrufen. Diese verwenden die gleichen Symbole wie ihre gierigen Geschwister, allerdings mit angehängtem Fragezeichen:

```
{n, m}?, {n,}?, {n}?, *?, +?, ??
```

Anker

Haben Sie schon Feuer gefangen? Sind Sie im Pattern-Matching-Fieber? Dann haben Sie vielleicht auch schon probiert, mit

```
m/\d{3}/)  # oder m/\d\d\d/)
```

alle dreistelligen Zahlen aufzuspüren. Wenn ja, werden Sie wahrscheinlich auch schon bemerkt haben, dass das Suchmuster »drei Ziffern« ja nicht nur auf dreistellige Zahlen wie 123 oder 453 zutrifft, sondern auch als Bestandteil größerer Wörter vorkommt und auch gefunden wird: 1234, abc345sd etc.

Was uns hier fehlt, ist ein »Anker«, der eine Wortgrenze angibt. Dieser Anker lautet `\b`.

```
m/\b\d{3}\b/
```

Dieser reguläre Ausdruck sucht nach allen dreistelligen Zahlen.

Neben `\b` gibt es noch weitere Anker, die man verwenden kann (siehe Tabelle 10.4).

Tabelle 10.4: Anker

Anker	Bedeutung
`\b`	Wortgrenze (Übergang von `\w` zu `\W`)
`\B`	Nichtwortgrenze
`^`	Anfang des Strings (wenn die Option `/m` gesetzt ist, auch jede Position direkt hinter einem Neue-Zeile-Zeichen (`\n`))
`$`	Ende des Strings (wenn die Option `/m` gesetzt ist, auch jede Position vor einem Neue-Zeile-Zeichen (`\n`))
`\A`	Anfang des Strings
`\Z`	Ende des Strings

10.3 Suchen und Ersetzen

Es liegt auf der Hand, dass es bei so vielen Möglichkeiten zum Suchen von Mustern in Texten auch eine Möglichkeit zum Ersetzen der gefundenen Vorkommen geben muss.

Wenn Sie ein Muster suchen und ersetzen wollen, beginnen Sie den Pattern-Matching-Ausdruck mit s/ statt mit m/ und geben direkt hinter dem zu suchenden Muster den Ersetzungsstring an:

```
$string =~ s/Suchmuster/Ersetzung/;
```

Wenn Sie den Operator wie oben aufrufen, wird nur das erste Vorkommen ersetzt. Wenn Sie alle Vorkommen in `$string` ersetzen wollen, müssen Sie die Option `g` anhängen.

```
$string =~ s/Suchmuster/Ersetzung/g;
```

Das folgende Beispielskript hilft dabei, einen Text von alter auf neue Rechtschreibung umzustellen.

Listing 10.4: rechtschreibung.pl

```
#!/usr/bin/perl -w
use strict;

my %woerter = ('heute abend' => 'heute Abend',
                'achtgeben' => 'Acht geben',
                'im allgemeinen' => 'im Allgemeinen',
                'Adreßbuch'=> 'Adressbuch',
                'daß' => 'dass',
                'als erstes' => 'als Erstes',
                'in bezug auf' => 'in Bezug auf',
                'auf deutsch' => 'auf Deutsch',
                'fallenlassen' => 'fallen lassen',
                'fertigstellen' => 'fertig stellen',
             'im großen und ganzen' => 'im Großen und Ganzen',
                'muß' => 'muss',
                'Numerierung' => 'Nummerierung',
                'Tip' => 'Tipp',
                );

open(EIN, "< $ARGV[0]")
    or
    die "\nDatei $ARGV[0] konnte nicht geoeffnet werden\n";

open(AUS, "> $ARGV[1]")
    or
    die "\nDatei $ARGV[1] konnte nicht geoeffnet werden\n";

undef $/;
my $text = <EIN>;

use locale;

foreach my $key (keys %woerter)
  {
  $text =~ s/\b$key\b/$woerter{$key}/g;
  }

print AUS $text;


close(EIN);
close(AUS);
```

Aufruf: **> perl rechtschreibung.pl alterText.txt neuerText.txt** (Windows)

Aufruf: **# ./rechtschreibung.pl alterText.txt neuerText.txt** (Linux)

Das Skript entnimmt der Kommandozeile die Namen zweier Dateien. Die erste Datei ist die Quelldatei, die den Text enthält, der in die neue deutsche Rechtschreibung verwandelt werden soll. Die zweite Datei wird neu erzeugt. In ihr speichert das Skript den überarbeiteten Text.

Die Wörter in ihren unterschiedlichen Schreibweisen entnimmt das Programm dem Hash `%woerter`. Die Schlüssel des Hash enthalten die alte und die Werte die neue Schreibweise. In einer Schleife werden die Schlüssel des Hash durchlaufen und jeweils alle Vorkommen des aktuellen Schlüssels durch seinen Wert ersetzt. Schließlich wird der überarbeitete Text ausgegeben und die Dateien werden geschlossen.

Beachten Sie bitte auch die etwas unscheinbare Anweisung `use locale;`. Dieses Pragma sorgt dafür, dass die Perl-Funktionen und -Operatoren nationale Eigentümlichkeiten (wie sie auf dem Rechner eingestellt sind) berücksichtigt. Speziell im Programm RECHTSCHREIBUNG.PL sorgt es dafür, dass das ß-Zeichen beim Pattern Matching korrekt behandelt wird.

Rückverweise

Was macht man, wenn man beim Ersetzen die gefundenen Vorkommen erweitern will?

Perl erlaubt Ihnen, beliebige Teile eines Musters in skalaren Variablen abzuspeichern. Sie brauchen den betreffenden Musterausschnitt dazu nur in runde Klammern zu setzen.

```
#!/usr/bin/perl -w

$text = "Deutschland spielte gegen Portugal 0:3";

$text =~ s/(\D*)(\d):(\d)/$1$3:$2/;
print "$text\n";
```

Das obige Skript ist unter dem Eindruck der Fußball-Europameisterschaft 2000 entstanden und hätte, wenn ich es früher fertig bekommen hätte, der deutschen Mannschaft vielleicht den Einzug ins Viertelfinale ermöglicht.

Betrachten wir die Zeile mit dem Ersetzen-Operator und konzentrieren wir uns erst einmal auf das zu suchende Muster:

```
s/(\D*)(\d):(\d)/
```

Wenn man einmal die runden Klammern ignoriert, die hier offensichtlich keine alternativen Zeichen einfassen, so erkennt man, dass das Muster wie folgt aufgebaut wird:

`\D*` liest eine beliebige Zahl von Zeichen, die keine Ziffern sein dürfen. Angewendet auf `$text` schluckt dieser Teil des Musters also den Teilstring "`Deutschland spielte gegen Portugal `".

`\d` schluckt eine einzelne Ziffer (`0`).

`:` schluckt `:`.

`\d` schluckt eine einzelne Ziffer (`3`).

Die runden Klammern um die Musterteile bewirken, dass die eingefassten Musterteile mit den Variablen `$1`, `$2` etc. verbunden werden. Bei Anwendung des Musters auf einen String werden die Stringteile, die von den eingeklammerten Musterteilen geschluckt werden, in die zugehörigen Variablen kopiert und stehen sofort danach zur Weiterverarbeitung zur Verfügung.

Im Ersetzungsstring wird dies ausgenutzt: `/$1$3:$2/` rekonstruiert den durchsuchten String fast vollständig – lediglich der Inhalt der Variablen `$2` und `$3` wird vertauscht. Die Ausgabe des Skripts lautet danach:

```
Deutschland spielte gegen Portugal 3:0
```

Wenn Sie innerhalb eines Suchmusters Rückverweise auf vorangehende Stellen des Suchmusters verwenden wollen, lauten die Rückverweise `\1`, `\2` etc.

Wenn Sie Rückverweise verschachteln, richtet sich die Nummerierung der Rückverweise nach den öffnenden runden Klammern.

10.4 Gefundenen Text weiter verarbeiten

Nicht selten werden reguläre Ausdrücke dazu verwendet, bestimmte Inhalte aus einem Text zu extrahieren und anderweitig zu verarbeiten. Dazu ist es erforderlich, dass die vom Pattern-Matching-Operator gefundenen Vorkommen in Variablen gespeichert werden.

Eine Möglichkeit zur Abspeicherung der gefundenen Vorkommen (bzw. Teile der Vorkommen) haben Sie bereits im vorangehenden Abschnitt gesehen: Man klammert die Musterteile, deren Inhalt man speichern lassen möchte, und erhält das Ergebnis in den Variablen `$1`, `$2` etc. zurück.

Eine weitere Möglichkeit besteht darin, den Operator im Listenkontext aufzurufen:

- ✘ Wenn Sie den `m//`-Operator im Listenkontext ohne die Option `g` aufrufen, werden die Inhalte der geklammerten Teilausdrücke in die angegebenen Listenelemente kopiert.

```
$text = "Deutschland spielte gegen Portugal 0:3";
($partie, $ergebnis) = $text =~ m/(\D*)(\d:\d)/;
print "$partie\n";
print "$ergebnis\n";
```

- ✘ Wenn Sie den `m//`-Operator im Listenkontext mit der Option `g` aufrufen, werden die gefundenen Vorkommen in das angegebene Array kopiert.

```
$text = "Wadde hadde dudde da";
@de = $text =~ m/\b\w*?de/g;
print "@de\n";
```

Das folgende Beispiel sucht in einem HTML-Dokument nach enthaltenen Hyperlinks und kopiert diese in eine neue HTML-Datei. Der Name der zu durchsuchenden und der neu anzulegenden HTML-Datei wird über die Kommandozeile übergeben.

Listing 10.5: links.pl

```
#!/usr/bin/perl -w
use strict;

open(EIN, "< $ARGV[0]")
    or
    die "\nDatei $ARGV[0] konnte nicht geoeffnet werden\n";
```

```
open(AUS, "> $ARGV[1]")
    or
    die "\nDatei $ARGV[1] konnte nicht geoeffnet werden\n";

undef $/;
my $text = <EIN>;
my @links = ();


@links = $text =~ m{<A\s+?href.*?/A>}sg;
map( {$_ = "<p>$_</p>\n"} @links);

print AUS @links;


close(EIN);
close(AUS);
```

Aufruf: > **perl links.pl webseite.html neu.html** (Windows)

Aufruf: # **./links.pl webseite.html neu.html** (Linux)

Interessant ist vor allem wieder der Pattern-Matching-Ausdruck:

```
m{<A\s+href.*?/A>}sg
```

Dieser Ausdruck soll nach Hyperlinks suchen. Dazu sucht er zuerst nach Textstellen, die mit `<A` beginnen. `\s+?` bedeutet, dass danach mindestens ein Whitespace-Zeichen folgen muss. Es kann aber auch sein, dass mehrere Whitespace-Zeichen folgen, daher der +-Quantifizierer. Danach kommt wieder normaler Text: `href`. Alles was dann zwischen `href` und dem abschließenden `/A>` steht, wird von dem Ausdruck `.*?` eingelesen. Dabei ist zu beachten, dass in dem Text zwischen `href` und `/A>` auch Zeilenumbrüche (`\n`) enthalten sein können. Der Punkt steht aber standardmäßig für alle Zeichen außer `\n`. Um zu erreichen, dass eventuell eingeschlossene `\n`-Zeichen mitgeschluckt werden, setzen wir am Ende des Pattern-Matching-Ausdrucks die Option `s`. Die Option `g` sorgt schließlich dafür, dass alle Vorkommen gesucht und in dem Array `@links` abgelegt werden.

Der nachfolgende `map`-Aufruf dient lediglich der Formatierung der HTML-Ausgabe.

```
map( {$_ = "<p>$_</p>\n"} @links);
```

Er schließt die gefundenen Hyperlinks in `<p>`-Tags ein, damit sie in der Ausgabe untereinander in eigenen Absätzen stehen.

10.5 Fragen und Übungen

1. Wie lauten die Pattern-Matching-Operatoren zum Suchen sowie zum Suchen und Ersetzen?
2. Wie kann man erreichen, dass eine Suche alle Vorkommen findet?
3. Wie kann man die gefundenen Vorkommen in Variablen zurückliefern lassen?
4. Wie kann man erreichen, dass bei der Suche nicht zwischen Groß- und Kleinschreibung unterschieden wird?
5. Was suchen die folgenden regulären Ausdrücke?

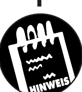
```
/\s*;\s*/
s/,/\./g
s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words
/(\b\d+\.\d*\b)/g
```

6. Setzen Sie reguläre Ausdrücke auf, die
   a) das erste Wort im Suchstring finden
   b) das erste Wort in jeder Zeile finden
   c) Stabreime entdecken können (zwei aufeinanderfolgende Wörter mit gleichem Anfangsbuchstaben)
   d) Folgen von drei gleichen Buchstaben entdecken
7. Verbessern Sie das Programm WOERTER.PL aus Kapitel 5.3.3, so dass vor der Zählung sämtliche Satzzeichen aus dem Text verbannt werden.
8. Lesen Sie eine HTML-Datei ein und färben Sie alle Überschriften rot.

Kapitel 11

Objektorientierte Programmierung in Perl

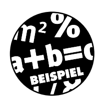
»Objektorientierte Programmierung« ist eines der großen Schlagwörter, das die Programmierszene der letzten Jahre weltweit geprägt hat. Dabei ist die Idee der objektorientierten Programmierung gar nicht so neu. Bereits unter den Programmiersprachen der ersten Tage – ich denke da zurück bis in die Bronzezeit des Software Engineering, also etwa bis in die Siebziger – gab es objektorientierte Sprachen (Smalltalk, Modula). Schlagzeilen machte die objektorientierte Programmierung aber erst als Bjarne Stroustrup Anfang bis Mitte der Achtziger die Programmiersprache C um objektorientierte Konzepte erweiterte und aus C die Sprache C++ wurde. Der vorerst letzte Höhepunkt des Siegeszuges der objektorientierten Programmierung ist ... nein, nicht die Einführung objektorientierter Konzepte in Perl 5, sondern die Entwicklung der derzeit sehr erfolgreichen und rein objektorientierten Sprache Java. Warum Java und nicht Perl?



In Perl spielen die objektorientierten Konzepte derzeit aber noch nicht die gleiche Rolle wie in Programmiersprachen wie C++ oder Java. Warum dies so ist, was objektorientierte Programmierung überhaupt bedeutet und wie man in Perl objektorientiert programmiert, das soll Gegenstand dieses Kapitels sein.

Im Einzelnen beschäftigen wir und in diesem Kapitel mit

- ✘ der Zielsetzung der objektorientierten Programmierung
- ✘ den Grundkonzepten der objektorientierten Programmierung
- ✘ der Beziehung zwischen Klassen und Objekten



- der Definition von Methoden
- der Bedeutung des Konstruktors
- dem Schutz der internen Daten der Klasse

Folgende Elemente lernen Sie kennen

Die Funktion `bless`.

11.1 Perl und OOP

Warum spielt die objektorientierte Programmierung in Perl nicht die gleiche Rolle wie in anderen Programmiersprachen? Zum einem liegt dies natürlich an der Verankerung der objektorientierten Konzepte in der Sprache. In der aktuellen Version ist Perl keine echte objektorientierte Sprache, sondern mehr eine nicht objektorientierte Sprache, mit der man objektorientierte Programmierung simulieren kann. Es liegt aber auch daran, dass der Nutzeffekt der objektorientierten Konzepte in Perl-Skripten nicht mit dem Effekt in C++- oder Java-Programmen vergleichbar ist.

Die Techniken der objektorientierten Programmierung zielen vor allem darauf ab, Quellcode übersichtlicher, modularer und dadurch auch sicherer und besser wiederverwertbar zu machen. Der »Trick«, den die objektorientierte Programmierung dazu anwendet, besteht darin, zusammengehörende Daten und Funktionen, die auf diesen Daten operieren, in Objekten zu kapseln.

Nehmen wir an, Sie schreiben ein Programm zur Verwaltung eines Bankkontos. Zu dem Bankkonto gehören verschiedene Daten:

- Name des Inhabers
- Kontostand
- Kreditvolumen

Des Weiteren gibt es verschiedene Operationen, die auf dem Bankkonto ausgeführt werden:

- Abfragen des Kontostands
- Einzahlen
- Abheben

Ohne objektorientierte Konzepte wird das Bankkonto in Ihrem Programm durch eine Sammlung von einzelnen Variablen (für die Daten) und Funktionen (für die Operationen auf den Daten) repräsentiert. Die Beziehung zwi-

schen diesen Daten und Funktionen wird vor allem in Ihrem Kopf hergestellt. Okay, natürlich könnte man die Daten in einem Hash zusammenfassen (was sehr zu empfehlen wäre), und die Beziehung zwischen den Funktionen und den Daten kann und wird über die Parameter der Funktion hergestellt. Es bleibt aber das Problem, dass Sie die Übersicht darüber behalten müssen, welche Funktionen auf welche Daten angewendet werden dürfen. In einem Programm, das lediglich zur Verwaltung eines Bankkontos geschrieben wurde, mag das dem Programmierer keine wirkliche Schwierigkeiten bereiten. Nehmen wir aber einmal an, es handele sich um ein größeres Programm, das für eine Bank erstellt wurde und das nicht nur ein einzelnes Konto verwaltet, sondern alle Konten der Kunden plus die Kundendaten, die vorgenommenen Buchungen, die vergebenen Kredite etc. Zum Schluss enden Sie mit einem Wust an Daten und Funktionen, und Ihre Aufgabe ist es, im Kopf zu behalten, wann Sie welche Funktion für welche Daten aufrufen können.

OOP kapselt zusammengehörende Daten und Funktionen in Klassen

Hier setzt die objektorientierte Programmierung an. Kunden, Kontos, Buchungen und Kredite betrachtet sie als Objekte. Jedes dieser Objekte verfügt über eigene Variablen und Funktionen. Objekte, die die gleichen Variablen und Funktionen beinhalten (beispielsweise alle Konten), gehören einer gemeinsamen Klasse an (der Klasse der Konten). Für den Programmierer hat dies den Vorteil, dass er statt in einzelnen Daten und Funktionen in Objekten denken kann. Wenn er den Kontostand eines bestimmten Kontos erhöhen will, muss er nicht überlegen, wie die Variable heißt, in der der Kontostand abgelegt ist, oder welche Funktion die richtige zum Einzahlen ist. Er ruft einfach die Einzahlen-Funktion des Konto-Objekts auf.

Die Kapselung von Daten und Funktionen in Klassen ist die Basis der objektorientierten Programmierung, auf der weitere Konzepte (Schutzmechanismen für Klassen, Vererbung, Polymorphismus, Templates etc.) aufbauen. Alle diese Konzepte zielen darauf ab, den Quellcode besser zu organisieren, um die Erstellung des Codes sowie seine Wartung und eventuelle Wiederverwertung zu erleichtern. Es liegt auf der Hand, und das obige Beispiel dürfte dies auch deutlich gemacht haben, dass diese Konzepte sich als umso segensreicher erweisen, je umfangreicher der Code ist und je mehr verschiedenartige Daten in dem Code verarbeitet werden. Dies ist aber auch der Grund, warum die objektorientierte Programmierung in Perl nicht den gleichen Stellenwert einnimmt wie in C++ oder Java. Perl-Skripten sind meist kleine Programme, die eine einzelne, klar definierte Aufgabe erledigen. Niemand würde auf die Idee kommen, eine Bank-Software, das Buchungssystem einer Reisegesellschaft oder sonstige Programmungetüme in Perl zu schreiben. In kleinen Skripten aber kann man gut auf objektorientierte Konzepte verzichten.

Trotzdem wollen wir uns in den nachfolgenden Abschnitten ein wenig intensiver in die objektorientierte Programmierung einarbeiten. Zum einem weil es im CPAN mittlerweile etliche Module gibt, die objektorientiert programmiert sind, zum anderen weil Grundkenntnisse der objektorientierten Programmierung heute einfach zum Repertoire eines guten Programmierers gehören.

11.2 Grundlagen der objektorientierten Programmierung

Da Perl die objektorientierten Konzepte derzeit nicht wirklich in der Sprache verankert hat, sondern eher simuliert, werde ich zur Einführung dieser Konzepte auf eine andere Sprache, auf C++, ausweichen. Schauen wir uns also einmal an, was objektorientierte Programmierung in C++ bedeutet.

Klassen und Objekte

Im objektorientierten Sinne sind Klassen abstrakte Beschreibungen einer Gruppe von gleichartigen Objekten. So sind beispielsweise der Peugeot und der Mercedes, die bei Ihnen vor dem Haus stehen, Objekte der Klasse `Auto`. Die Zahlen 1, 2, -4 etc. sind Objekte der Klasse `Ganzzahlen`. Alle Objekte einer Klasse haben gemeinsame Eigenschaften und Verhaltensweisen. Autos haben eine PS-Zahl, einen Verbrauch, einen Preis, eine Farbe (die Eigenschaften) und können starten, beschleunigen, bremsen, hupen (die Verhaltensweisen). Ganzzahlen haben als einzige Eigenschaft einen Wert und als Verhaltensweisen die auf ihnen erlaubten Operationen: Addition, Subtraktion, Negation etc.

Das Beispiel mit den Zahlen bringt uns der Grundidee der objektorientierten Programmierung schon sehr nahe. Was passiert, wenn wir mit Zahlen programmieren?

```
$var1 = 3;
$var2 = $var1 * 2;
```

Wir wissen, dass Ganzzahlen in Perl zum Datentyp der Skalare gehören. Also deklarieren wir eine Variable `$var` dieses Datentyps. Der Datentyp `Skalar` stellt im objektorientierten Sinne eine Klasse dar, jede Variable dieses Datentyps ist ein Objekt der Klasse. (Um ganz korrekt zu sein: das eigentliche Objekt wird intern im Speicher angelegt, die Variable ist nur ein Name, der im Programm dieses Objekt repräsentiert.)

Wenn wir der Variablen einen Wert zuweisen, entspricht dies der Zuweisung eines Wertes an die Eigenschaft des Objekts. Die Operationen, die auf den

Zahlen erlaubt sind (Addition, Subtraktion etc.), stellen die Verhaltensweisen des Objekts dar.

Solange wir mit Skalaren programmieren, programmieren wir also in gewisser Weise objektorientiert. Leider hört dies sofort auf, wenn wir mit anderen Objekten, beispielsweise zweidimensionalen Vektoren (x,y), programmieren wollen. Da es in Perl keinen Datentyp gibt, der der Klasse der Vektoren entspricht, definieren wir als Ersatz zwei Skalare für x und y (oder ein Hash mit den Schlüsseln x und y) und setzen Funktionen zur Bearbeitung der Vektoren (Addition, Subtraktion etc.) auf. Die Objektorientiertheit geht dabei allerdings verloren. Wir können keine Variable vekt deklarieren, die ein Objekt der Klasse Vektor darstellt und genau über zwei Eigenschaften (x und y) verfügt. Bestenfalls haben wir ein Hash %vekt mit den zwei Schlüsseln x und y, das uns als Ersatz dient. Doch niemand hindert uns, dieses Hash um weitere Schlüssel zu erweitern, niemand hindert uns, die Schlüssel x und y zu löschen. Ein Hash ist eben kein Datentyp, sondern lediglich eine komplexe Datenstruktur. Schließlich gibt es auch keine Operationen, die wir auf Vektor-Objekte anwenden könnten.

An diesem Punkt kommen wir zur Programmiersprache C++. C++ erlaubt dem Programmierer nämlich, eigene Datentypen zu definieren, die Klassen im objektorientierten Sinne repräsentieren.

```
// Klassendeklaration
class Vektor
  {
  private:
    int x;                          // Instanzvariablen
    int y;

  public:
    Vektor();                       // Methoden
    void addieren(Vektor v);
    void subtrahieren(Vektor v);
    int skalarprodukt(Vektor v);
    vektor vektorprodukt(Vektor v);
  };
```

Hier wird eine Klasse Vektor definiert, die über zwei Datenelemente, x und y, und mehrere Methoden verfügt.

Funktionen, die Teil einer Klasse sind, nennt man Methoden. Variablen, die die Datenelemente der Klasse darstellen, nennt man Instanzvariablen.

Zum besseren Verständnis dieses Codes sei darauf hingewiesen, dass in C++ Variablendeklarationen immer aus der Angabe des Datentyps und des Variablennamens bestehen; der Variablentyp wird also nicht wie bei Perl durch Voranstellung eines Symbols angezeigt, sondern ist nur in der Deklaration erkennbar. Der Datentyp `int` steht beispielsweise für Ganzzahlen. Methodendefinitionen, oder allgemein Funktionsdefinitionen, beginnen nicht mit einem eigenen Schlüsselwort (wie `sub` in Perl), sondern mit dem Datentyp des Rückgabewerts. Darauf folgen der Funktionsname und in runden Klammern die Parameter der Funktion. In C++ werden keine Standardparameter verwendet (vergleiche `@_` in Perl). Der Programmierer muss für jeden Parameter in der Funktionsdefinition eine eigene Variable deklarieren.

Instanzbildung und Zugriff auf Klassenelemente

Nach der Definition des Klassentyps kann man Objekte der Klasse erzeugen. Dies geschieht einfach durch Deklaration einer Variablen vom Datentyp der Klasse.

```
Vektor v1;                     // Erzeugung von Objekten
Vektor v2;
```

Variablen von Klassentypen nennt man Instanzen (nicht zu verwechseln mit den Instanzvariablen, die die untergeordneten Variablen jeder Instanz der Klasse bilden).

Über die Instanz kann man auf die Instanzvariablen des Objekts zugreifen, um deren Werte abzufragen oder zu setzen, und man kann die Methoden der Klasse aufrufen.

```
v1.addieren(v2);               // Aufruf einer Methode
```

Aber Achtung! In C++ kann eine Klasse selbst entscheiden, auf welche ihrer Elemente man über ein Objekt der Klasse zugreifen kann. Zu diesem Zweck werden in der Klassendeklaration Zugriffsspezifizierer vergeben, die für alle darunter deklarierten Klassenelemente gelten.

```
// Klassendeklaration
class Vektor
  {
  private:                     // Zugriffsspezifizierer
    int x;
    ...
```

```
  public:                       // Zugriffsspezifizierer
    Vektor();
    void addieren(Vektor v);
    ...
  };
```

Der Zugriffsspezifizierer `public` macht die nachfolgenden Elemente öffentlich, d.h. man kann über ein Objekt auf das Element zugreifen:

```
v1.addieren(v2);         // ruft über das Objekt v1 die
                         // Methode addieren auf
```

Der Zugriffsspezifizierer `private` schützt die nachfolgenden Elemente vor dem Zugriff über ein Objekt:

```
v1.x = 3;                // wäre syntaktisch korrekt, ist
                         // aber nicht erlaubt, da x
                         // private ist
```

Elemente, die `private` sind, können daher nur über `public`-Methoden der Klasse manipuliert werden. So kann der Programmierer, der die Klasse definiert, sicherstellen, dass die Datenelemente der Objekte immer korrekte Werte enthalten. Nehmen wir an, der Programmierer setzt eine Klasse für Bruchzahlen auf.

```
class Bruch
  {
  public:
    int zaehler;
    int nenner;
    ...

    double get_bruch() {return (double) zaehler / nenner;}
    ...
  };
```

Obige Implementierung setzt voraus, dass der Programmierer, der mit der Klasse `Bruch` arbeitet, weiß, dass er die Instanzvariable `nenner` nicht auf 0 setzen darf (sonst führt ein nachfolgender Aufruf von `get_bruch` zu einer Division durch Null!).

```
Bruch b;
b.zaehler = 12;
b.nenner = 0;   // Holla!
b.get_bruch();  // jetzt geht alles in die Brüche
```

Dank der objektorientierten Schutzmechanismen in C++ kann der Programmierer der Klasse solche Fehler aber problemlos verhindern, indem er die Instanzvariable `nenner` `private` macht und zum Setzen der Instanzvariablen eine `public`-Methode vorsieht, die prüft, dass nur Werte ungleich 0 zugewiesen werden:

```
class Bruch
  {
  public:
    int zaehler;
  private:
    int nenner;

  public:
    Bruch() {zaehler = 1; nenner = 1;}
    double get_bruch() {return (double) zaehler / nenner;}
    void set_nenner(int wert) { if (wert != 0) {
                                    nenner = wert;
                                    }
                                else {
                                    nenner = 1;
                                    }
                              }

  };
...
Bruch b;
b.zaehler = 12;
b.set_nenner(0);   // wird auf 1 gesetzt
b.get_bruch();
```

Methodendefinition

Die (`public`-) Methoden einer Klasse legen fest, was man mit den Objekten der Klasse machen kann. Da man Vektoren addieren und subtrahieren kann, ist es sinnvoll in der Klasse entsprechende Methoden zu definieren, die diese Operationen ausführen.

```
void Vektor::addieren(vektor v)
  {
  x = x + v.x;
  y = y + v.y;
  }
```

Diese Funktion addiert zu den Werten der Instanzvariablen des aktuellen Objekts (`x` und `y`) die Werte der Instanzvariablen des übergebenen Parameters (`v`).

Aufgerufen werden die Methoden wie alle Klassenelemente über ein konkretes Objekt:

```
Vektor v1, v2, v3;
...
v1.addieren(v2);
v2.addieren(v3);
```

Woher aber weiß die Methode `addieren`, wenn Sie für das Objekt `v1` aufgerufen wird, dass mit den Instanzvariablen `x` und `y` aus ihrer Methodendefinition `v1.x` und `v1.y` gemeint sind, während bei einem Aufruf über das Objekt `v2` die Instanzvariablen `v2.x` und `v2.y` gemeint sind?

Wenn man ein Objekt einer Klasse deklariert:

```
Vektor v1;
```

wird im RAM Speicherplatz für die Instanzvariablen des Objekts reserviert. Jedes Objekt erhält also einen kompletten Satz von Kopien der in der Klasse definierten Instanzvariablen. Zugriffe auf die Instanzvariablen eines Objekts (`v1.x = 1;`) führen daher direkt zum Speicher des Objekts.

Die Methodendefinitionen einer Klasse stehen allerdings nur einmal im Speicher und alle Objekte der Klasse verwenden den gleichen Code. Trotzdem manipulieren die Methoden immer nur die Instanzvariablen des Objekts, über dessen Instanznamen sie aufgerufen wurden. Wie ist das möglich?

Der Trick dabei ist, dass die Methodendefinition intern vom C++-Compiler (C++ verwendet einen Compiler statt eines Interpreters) um einen Parameter erweitert wird. Dieser Parameter, der immer als erster Parameter definiert ist, enthält eine Referenz auf das Objekt, für das die Methode aufgerufen wurde. Gleichzeitig stellt der Compiler in der Definition der Methode allen Zugriffen auf Klassenelemente diese Referenz voran.

Intern sieht die Definition von `addieren` demnach wie folgt aus:

```
void Vektor::addieren(Vektor* objref, Vektor v)
  {
  objref->x = objref->x + v.x;
  objref->y = objref->y + v.y;
  }
```

C++ kennt zwar auch Referenzen, doch funktionieren diese anders als die Perl-Referenzen. Die »Referenz« auf das aufrufende Objekt ist in C++ daher in Wirklichkeit ein Zeiger, der im übrigen den Namen `this` trägt.

Der Konstruktor

Eine besondere Methode ist der sogenannte Konstruktor (der in C++ den gleichen Namen trägt wie die Klasse und keinen Rückgabetyp hat).

Der Konstruktor wird automatisch bei der Instanzbildung, sprich der Deklaration einer Variablen vom Typ der Klasse, aufgerufen. Dies wird üblicherweise dazu genutzt, um den Instanzvariablen der Klasse Anfangswerte zuzuweisen. In C++ kann der Konstruktor nicht explizit wie andere Methoden aufgerufen werden.

```
void Vektor::Vektor()  // Konstruktor der Klasse Vektor
  {
  x = 1;
  y = 1;
  }
```

Soviel zu C++ und den grundlegenden objektorientierten Konzepten. Es gibt in der objektorientierten Programmierung zwar noch eine Reihe weiterer wichtiger Konzepte (statische Klassenelemente, Vererbung, Polymorphismus, Überschreibung von Methoden, Bindung etc.), doch als Einführung in OOP lassen wir es bei dem Grundkonzept der Kapselung von Daten und Methoden in Klassen bewenden (zumal etliche der fortgeschrittenen Konzepte in Perl nicht realisiert werden können).

11.3 Objektorientierte Programmierung in Perl

In Perl kann man keine Datentypen deklarieren. Wie also deklariert man eine Klasse? Indem man ein Package erzeugt. Nicht jedes Package muss eine Klasse enthalten, aber jede Klasse muss in einem Package deklariert sein.

Das Package für die Klasse muss auch nicht zwangsweise als eigenes Modul eingerichtet und in den Include-Pfad des Perl-Interpreters gestellt werden. Insbesondere zum Aufsetzen und Testen einer Klasse empfiehlt es sich, das Package mit der Klassendefinition erst einmal in einem normalen Skript unterzubringen.

Die Klassendeklaration

Wir beginnen die Erstellung unserer Perl-Klasse daher mit einer Package-Deklaration.

```
#!/usr/bin/perl -w

#***************** Definition der Klasse Vektor ********
package Vektor;
```

Ähnlich wie in einer C++-Klassendefinition führt man in dem Package der Klasse die einzelnen Elemente der Klasse auf. Doch Vorsicht! Durch die normale Deklaration von Variablen und Funktionen kann man keine Instanzvariablen oder Methoden definieren. Dazu bedarf es verschiedener Tricks.

Der Konstruktor und die Instanzvariablen

In C++ trägt der Konstruktor den gleichen Namen wie die Klasse und wird automatisch bei der Instanzbildung aufgerufen. In Perl muss der Konstruktor explizit aufgerufen werden und kann einen beliebigen Namen tragen (per Konvention wird er aber meist `new` genannt). Der Konstruktor ist im Grunde also eine Methode wie jede andere, der allerdings eine besondere Aufgabe zufällt. Im Konstruktor werden die Instanzvariablen eingerichtet.

```
sub new {
  my $classname = shift;
  my $self = {};
  $self->{x} = 0;
  $self->{y} = 0;
  bless ($self, $classname);
  return $self;
  }
```

Der Konstruktor beginnt damit, dass er die ihm übergebenen Argumente einliest. Als erstes Argument erwartet er stets den Namen der Klasse (sprich den Namen des Package der Klasse).

```
  my $classname = shift;
```

Dann deklariert er die Instanzvariablen, das heißt, er deklariert einen lokalen Hash und erzeugt die einzelnen Instanzvariablen der Klasse als Schlüssel/Wert-Paare des Hash.

```
  my $self = {};
  $self->{x} = 0;
  $self->{y} = 0;
```

Jetzt kommt der wichtigste Schritt der ganzen Konstruktordefinition. Mit Hilfe der Methode `bless` wird aus dem lokalen Hash (wir haben es `$self` genannt) eine Referenz auf ein Objekt. Zu diesem Zweck übergibt man `bless` den Namen des Hash und den Namen der Klasse. Zum Schluss wird die neu entstandene Objektreferenz zurückgeliefert.

```
  bless ($self, $classname);
  return $self;
  }
```

Weiter unten im Skript kann man nun bereits Instanzen der Klasse erzeugen.

```
#**************** Eigentliches Skript ********
package main;

$vektobj1 = new Vektor;
```

Hier wird der Konstruktor der Klasse `Vektor` aufgerufen. Die vom Konstruktor zurückgelieferte Referenz wird in dem Skalar `$vektobj1` gespeichert.

Konstruktoren mit mehreren Argumenten

Wenn man will, kann man den Konstruktor auch so implementieren, dass er zur Initialisierung der Instanzvariablen Argumente aus seinem Aufruf übernimmt:

```
package Vektor;
sub new {
  my $classname = shift;
  my $self = {};
  $self->{x} = shift;
  $self->{y} = shift;
  bless ($self, $classname);
  return $self;
  }

package main;
$vektobj = new Vektor (8, 67);
```

Die Methoden

Ähnlich wie der C++-Compiler sorgt auch der Perl-Interpreter dafür, dass bei Methodenaufrufen intern als erstes Argument eine Referenz auf das aufrufende Objekt übergeben wird. Aus Aufrufen wie

```
$vektobj1 = new Vektor;
$vektobj1->set(8,67);
```

macht der Perl-Interpeter:

```
$vektobj1->set($vektobj1, 8,67);
```

Mit Hilfe der übergebenen Objektreferenz kann man dann in der Implementierung der Methode auf die Instanzvariablen des Objekts zugreifen.

```
sub get {
  my $self = shift;                    # Objektreferenz abfragen
  return ($self->{x}, $self->{y});  # auf Instanzvariablen
                                       # zugreifen
  }
sub set {
  my $self = shift;
  $self->{x} = shift;
  $self->{y} = shift;
  }

sub addiere {
  my $self = shift;
  $self->{x} += shift;
  $self->{y} += shift;
  }
```

Klassendefinition abschließen

Steht die Klassendefinition allein in einer eigenen Package-Datei, braucht man als letzte Anweisung nur noch einen »wahren« Wert zurückliefern:

1;

Steht die Klassendefinition mit mehreren Klassen oder sonstigem Code zusammen in einer Datei, schließt man die Klassendefinition ab, indem man einfach ein neues Package einschaltet.

Schutzmechanismen

Perl kennt keine Zugriffsspezifizierer, mit denen man Zugriffe auf Klassenelemente, die über eine Objektreferenz erfolgen, unterbinden könnte (vergleiche `private` und `public` von C++).

Für Programmierer, die bereits definierte Klassen verwenden, empfiehlt es sich daher, möglichst nur Methoden der Klasse aufzurufen und den direkten Zugriff auf Instanzvariablen zu vermeiden.

Umgekehrt sollte man beim Implementieren einer Klasse darauf achten, dass für alle wichtigen Operationen entsprechende Methoden vorhanden sind. Will man dem Anwender Lese- oder Schreibzugriff auf Instanzvariablen gewähren, sollte man dazu `get/set`-Methoden vorsehen, mit denen

der Anwender die Werte der betreffenden Instanzvariablen abfragen oder verändern kann (siehe Abschnitt »Die Methoden«).

Der vollständige Code

Listing 11.1: vektor.pl

```
#!/usr/bin/perl -w

#***************** Definition der Klasse Vektor ********
package Vektor;

sub new {
  my $classname = shift;
  my $self = {};
  $self->{x} = 0;
  $self->{y} = 0;
  bless ($self, $classname);
  return $self;
  }

sub get {
  my $self = shift;
  return ($self->{x}, $self->{y});
  }

sub set {
  my $self = shift;
  $self->{x} = shift;
  $self->{y} = shift;
  }

sub addiere {
  my $self = shift;
  $self->{x} += shift;
  $self->{y} += shift;
  }

#***************** Eigentliches Skript ********
package main;


$vektobj1 = new Vektor;
print "Vektor = (",join(':',$vektobj1->get),")\n";

$vektobj1->set(22, 33);
print "Vektor = (",join(':',$vektobj1->get),")\n";
```

```
$vektobj2 = new Vektor;
$vektobj2->set(8,67);
$vektobj1->addiere($vektobj2->get);
print "Vektor1 = (",join(':',$vektobj1->get),")\n";
print "Vektor2 = (",join(':',$vektobj2->get),")\n";
```

Ausgabe:

```
Vektor = (0:0)
Vektor = (22:33)
Vektor1 = (30:100)
Vektor2 = (8:67)
```

Klassen in eigene Module auslagern

Wenn Sie die Implementierung der Klasse abgeschlossen und die Verwendung der Klasse getestet haben, können Sie die Klasse als eigenes Modul einrichten.

1. Löschen Sie den `main`-Code im Skript der Klasse.
2. Schließen Sie die Klassendefinition mit der Anweisung `1;` ab.
3. Speichern Sie die Datei unter dem Namen der Klasse (sprich des Package) und mit der Extension `pl` in Lib-Verzeichnis Ihres Perl-Interpreters (siehe Ausgabe von `perl -V`).

Wenn Sie die Klasse danach in einem Ihrer Skripten verwenden wollen, müssen Sie die Klasse mit dem `use`-Pragma einbinden.

11.4 Fragen und Übungen

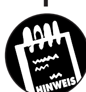
1. In welcher Beziehung stehen Klassen und Objekte zueinander?
2. Was ist eine Instanz?
3. Gibt es einen Unterschied zwischen einer Instanz und einem Objekt?
4. Welche Rolle spielen Packages für die Klassendefinition in Perl?
5. Setzen Sie die Minimalversion eines Konstruktors auf!
6. Was unterscheidet eine Methode von einer Funktion?
7. Wandeln Sie VEKTOR.PL in ein pm-Modul um.

Kapitel 12

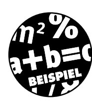
Grafische Oberflächen

Bisher haben wir ausschließlich Konsolenprogramme erstellt. Unter Windowing-Systemen wie XWindows, KDE oder Microsoft Windows werden diese Programme im Kontext eines Konsolenfensters ausgeführt (je nach Windowing-System spricht man statt von einem Konsolenfenster auch von einem Terminal oder der MDOS-Eingabeaufforderung). Ein solches Konsolenfenster vermittelt zwischen dem Windowing-System, das von allen unter ihm ausgeführten Programme erwartet, dass sie fensterbasiert sind, und den Konsolenprogrammen, denen es die Umgebung einer Konsole vorgaukelt.

Es gibt unzählige Anwendungsmöglichkeiten für Konsolenprogramme, so dass man ohne Mühe ein 1000-seitiges Buch nur zum Thema »Perl-Konsolenprogramme« aufsetzen könnte. Wenn man allerdings als Anwender (und welcher Programmierer spielt nicht gerne auch einmal mit den Programmen seiner Kollegen) gewohnt ist, vorwiegend mit einem Windowing-System und mit fensterbasierten Anwendungen zu arbeiten, entsteht über kurz oder lang das drängende Bedürfnis, die eigenen Perl-Programme doch auch mit einer schönen grafischen Oberfläche zu versehen und statt in der Konsole in einem eigenen Fenster auszuführen.



Wie man mit Perl fensterbasierte Programme schreibt, ist Thema dieses Kapitels.



Im Einzelnen lernen Sie,

- welche Anforderungen Windowing-Systeme an fensterbasierte Programme stellen
- wie man mit dem `Tk`-Modul arbeitet
- wie man Hauptfenster und Steuerelemente erzeugt und konfiguriert
- wie man auf Ereignisse (beispielsweise das Anklicken eines Schalters) reagiert
- wie man Bilddateien lädt und im Fenster des Programms anzeigt
- wie man Code in regelmäßigen Abständen ausführt
- wie man Eingabemasken für Programme aufsetzt

Folgende Elemente lernen Sie kennen

Das `Tk`-Modul, verschiedene seiner Objekte (`MainWindow`, `Button`, `Label`, `Entry`, `ScrlListbox`, `Frame`), die Funktion MainLoop und eine Auswahl der wichtigsten Methoden: `configure`, `pack`, `after`.

12.1 Grundlagen und ein erstes Tk-Programm

Was unterscheidet ein fensterbasiertes Programm eigentlich von einem Konsolenprogramm?

Der auffälligste, und manchmal auch der einzige Unterschied, ist natürlich, dass fensterbasierte Programme über eine grafische Benutzeroberfläche verfügen und auf dem Desktop in Form eines Fensters erscheinen. Die grafische Benutzeroberfläche eröffnet dem Programm (bzw. dem Programmierer) neue Möglichkeiten, bedeutet aber auch Verpflichtungen.

Zu den neuen Möglichkeiten, die sich eröffnen, gehören unter anderem die Programmierung mit Grafiken und die Unterstützung der Maus als Eingabegerät. Die Verpflichtungen betreffen die Kommunikation mit dem Windowing-System (XWindows unter Unix/Linux, Windows unter Microsoft Windows). Das Windowing-System verwaltet alle auf dem System laufenden Anwendungen und deren Fenster. Will ein Programm unter einem Windowing-System ausgeführt werden, muss es sich beim Windowing-System korrekt anmelden. Es muss wissen, wie es auf Befehle und Nachrichten des Windowing-Systems zu reagieren hat, und es sollte sich Windows-konform

verhalten (d.h. in etwa so funktionieren, wie es Windows-Anwender von anderen fensterbasierten Programmen gewohnt sind).

Das klingt, als käme da eine ganze Menge Arbeit auf uns zu. Nun, ja und nein. Tatsächlich ist der Aufbau einer korrekten grafischen Benutzeroberfläche mit einem nicht zu unterschätzenden Programmieraufwand verbunden. Wenn man will, kann man ohne Probleme zwei Stunden an der grafischen Oberfläche eines Programms basteln, das nicht mehr macht, als »Hallo!« in seinem Fenster auszugeben. Man kann es sich aber auch einfacher machen, indem man eine leistungsfähige Bibliothek verwendet, die die Windows-Programmierung erleichtert und dem Programmierer einen großen Teil der Arbeit abnimmt. Für Perl steht eine solche Bibliothek zur Verfügung: `Tk`.

Sie können `Tk` sowohl für die Windows-Programierung unter Unix/Linux als auch unter Microsoft Windows verwenden.

Ist Tk installiert?

Um zu testen, ob `Tk` installiert ist, brauchen Sie lediglich ein leeres Skript aufzusetzen, in das Sie das `Tk`-Modul einbinden:

```
#!/usr/bin/perl
use strict;

use Tk;
```

Wenn das `Tk`-Modul korrekt installiert ist, wird das Skript anstandslos ausgeführt. Sollte `Tk` auf Ihrem Rechner nicht installiert sein, erhalten Sie eine Fehlermeldung der Form:

```
Can't locate Tk.pm in @INC...
```

In diesem Fall müssen Sie `Tk` nachinstallieren (siehe Anhang B).

Ein Hauptfenster erzeugen

Wenn `Tk` verfügbar ist, können wir damit beginnen, die grafische Oberfläche aufzubauen.

Aber Achtung! `Tk` ist objektorientiert implementiert. Wenn Sie Kapitel 11 übersprungen haben, wäre jetzt vielleicht ein günstiger Zeitpunkt, das Versäumte nachzuholen. Falls Sie jedoch absolut kein Interesse haben, sich intensiver in die Konzepte der objektorientierten Programmierung einzuarbeiten, müssen Sie deshalb nicht auf die Windows-Programmierung verzichten. Die Verwendung der `Tk`-Objekte und deren Methoden ist so einfach, dass man sich schnell daran gewöhnt.

Der erste Schritt besteht darin, ein Hauptfenster zu erzeugen, das aus Sicht des Windowing-Systems und des Anwenders das Programm repräsentiert.

```
#!/usr/bin/perl
use strict;

use Tk;

my $hauptfenster = new MainWindow;
```

Für Hauptfenster ist in `Tk` die Klasse `MainWindow` definiert. Mit Hilfe von `new`, dem Konstruktor der Klasse, erzeugt man ein `MainWindow`-Objekt. (Intern erzeugt das `Tk`-Modul nicht nur ein `MainWindow`-Objekt, sondern auch ein echtes Fenster. Das Fenster wird beim Windowing-System als Hauptfenster der Anwendung angemeldet und mit dem `MainWindow`-Objekt verbunden).

Damit man im Programm mit dem erzeugten Hauptfenster arbeiten kann, liefert der `new`-Konstruktor eine Referenz auf das `MainWindow`-Objekt zurück. Diese Referenz speichern wir in einer skalaren Variablen (die im Beispiel `$hauptfenster` heißt). Fortan können wir über die Referenz `$hauptfenster` auf das `MainWindow`-Objekt, sprich das Hauptfenster der Anwendung, zugreifen.

Fenster, Objekte, Referenzen

Grundsätzlich spricht nichts dagegen, die Referenz `$hauptfenster` als das Hauptfenster der Anwendung zu betrachten. In der Folge werde ich auch nicht mehr groß zwischen den Fenstern und den Referenzen, die diese Fenster im Skript repräsentieren, unterscheiden. Zuvor möchte ich aber nicht versäumen, alle Klarheiten zu beseitigen und Sie darauf aufmerksam zu machen, dass man zwischen drei Dingen unterscheiden muss, die letztlich alle das Gleiche meinen, aber dennoch vollkommen verschieden sind.

Dem Fenster. Ein Fenster ist ein Windowing-Objekt. Es wird beim Windowing-System angemeldet und kann mit diesem kommunizieren (insbesondere kann es Benachrichtigungen vom Windowing-System entgegennehmen).

Dem Tk-Objekt. Dies ist ein Objekt im objektorientierten Sinn. Es ist ein Teil Ihres Programms, das mit einem Fenster verbunden ist. Über die Eigenschaften und Methoden des Objekts können Sie das Fenster manipulieren, das mit dem Objekt verbunden ist.

Der Objektreferenz. Das `Tk`-Objekt existiert – ebenso wie Zahlen, Strings etc. – nur im Arbeitsspeicher. Um aus unserem Skript heraus auf das Objekt zugreifen zu können, benötigen wir eine Referenz, die auf das Objekt verweist (im Beispiel `$hauptfenster`).

Steuerelemente aufnehmen

Als Anwender versteht man unter einem Fenster etwas, das eine Titelleiste und einen Rahmen besitzt und mit der Maus auf dem Desktop verschoben werden kann. Das Windowing-System definiert den Begriff des »Fensters« jedoch ganz anders. Für das Windowing-System ist ein Fenster, ein rechteckiger Bereich der Benutzeroberfläche, mit dem der Anwender interagieren kann. Diese Definition umschließt:

- ✘ **TopLevel-Fenster** (Hauptfenster), die über Titelleiste und Rahmen verfügen
- ✘ **Container-Fenster** (Frames), die unsichtbar sind und dazu dienen, untergeordnete Fenster zu gruppieren und anzuordnen
- ✘ **Steuerelemente** (Schalter, Eingabefelder, Optionsfelder, Bildlaufleisten etc.) die in Container- und TopLevel-Fenster eingebettet werden können

Unser nächster Schritt besteht nun darin, ein Steuerelement in das Hauptfenster unserer Anwendung aufzunehmen.

```
#!/usr/bin/perl
use strict;

use Tk;

my $hauptfenster = new MainWindow;

my $schalter = $hauptfenster->Button(
                               "-text" => "Drück mich",
                               );

   $schalter->pack();
```

Zur Erzeugung eines Steuerelements, das als eingebettetes Fenster im Hauptfenster angezeigt wird, stellt die Klasse `MainWindow` eine Reihe von speziellen Methoden zur Verfügung.

Tabelle 12.1: MainWindow-Methoden zur Erstellung der wichtigsten Steuerelemente

Steuerelement	MainWindow-Methode
Schalter	`Button`
Textfeld	`Label`
Eingabefeld	`Entry`
Kontrollkästchen	`Checkbutton`
Optionsfeld	`Radiobutton`
Listenfeld	`Listbox`
Menü	`Menu`

Beim Aufruf der Methode kann man das einzurichtende Steuerelement auch gleich konfigurieren, indem man bestimmten Eigenschaften des Steuerelements Werte zuweist. Die Eigenschaften werden in Form eines Hash verwaltet. Welche Eigenschaften zur Verfügung stehen, hängt zum Teil von dem jeweiligen Steuerelement ab und ist in der Dokumentation zum `Tk`-Modul beschrieben (eine Reihe von Eigenschaften werden Sie auch im Verlauf dieses Kapitels kennen lernen).

Um beispielsweise einen Schalter (englisch: Button) mit dem Titel »Drück mich« in das Hauptfenster aufzunehmen, schreibt man:

```
my $schalter = $hauptfenster->Button(
                                "-text" => "Drück mich",
                                );
```

Der Schalter ist jetzt als untergeordnetes Steuerelement des Hauptfensters eingerichtet. Er wird aber noch nicht im Hauptfenster angezeigt. Dazu muss man zuerst noch die Methode `pack` aufrufen.

```
$schalter->pack();
```

Ereignisse abfangen und bearbeiten

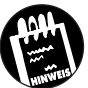
Schalter wirken auf Anwender wie Sirenen, die mit süßem Gesang dazu verführen, doch einmal auf den Schalter zu klicken und abzuwarten, was dann passiert. Unser Schalter macht, so wie er bis jetzt implementiert ist, allerdings gar nichts, was beim Anwender bestenfalls auf Verwunderung stoßen wird. Unser nächster Schritt soll daher darin bestehen, das Anklicken des Schalters mit einer Aktion zu verbinden.

Stellen Sie sich vor, das Programm läuft schon und der Anwender hat gerade den Schalter angeklickt. Jetzt schaltet sich das Windowing-System ein. Es registriert den Mausklick und schaut in seiner internen Fensterveraltung nach, in welchem Fenster der Mausklick erfolgte. Dann ermittelt das Windowing-System die Anwendung, zu der das Fenster gehört, und schickt diesem eine Benachrichtigung, dass sein Schalter angeklickt wurde. Doch dies ist nicht die einzige Nachricht, die das Windowing-System an die Anwendung schickt. Mittlerweile hat der Anwender die Maustaste losgelassen (das Windowing-System schickt auch dafür eine Nachricht) und die Maus ein wenig bewegt (das Windowing-System bombardiert die Anwendung mit Nachrichten zur aktuellen Position der Maus).

Aufgabe der Anwendung ist es, alle diese Nachrichten der Reihe nach entgegenzunehmen und einer geeigneten Verarbeitung zuzuführen. Dies geschieht durch Aufruf der `Tk`-Methode `MainLoop`.

```
MainLoop;
```

MainLoop nimmt die Nachrichten entgegen und verteilt sie weiter. Die Nachricht über das Anklicken des Schalters schickt sie an den Schalter. Jetzt kommt wieder Tk zum Zuge. Alle Tk-Objekte, die Fenster repräsentieren, sind so eingerichtet, dass sie die einkommenden Nachrichten entgegennehmen und verarbeiten. Meist sieht diese Verarbeitung so aus, dass die Nachricht einfach nur entgegengenommen und ansonsten nicht weiter beantwortet wird. Darüber hinaus bieten Ihnen die Objekte aber auch die Möglichkeit, in die Nachrichtenverarbeitung einzugreifen, indem Sie für bestimmte Ereignisse sogenannte Callback-Funktionen einrichten. Wenn das betreffende Ereignis eintritt (beispielsweise das Anklicken eines Schalters) und die Nachricht von dem zugehörigen Fenster empfangen wird, ruft die interne Tk-Nachrichtenverarbeitung dann Ihre Funktion auf und führt sie aus.



Weil nicht Sie, sondern der interne Tk-Code die Funktion aufrufen, spricht man von Callback-Funktionen. Die Einrichtung einer Callback-Funktion ist, als würden Sie zum Tk-Modul sagen: »Hier ist meine Funktion. Ruf sie auf, wenn es soweit ist!«

Die Callback-Funktion für das Anklicken des Schalters gibt man bei Einrichtung des Schalters an. Das Button-Objekt sieht dafür die Option -command vor:

```
my $schalter = $hauptfenster->Button(
                                "-text" => "Drück mich",
                                "-command" => sub {exit 0} );
```

Hier wird die Callback-Funktion quasi ad hoc definiert. Man hätte aber genauso gut eine Referenz auf eine nachfolgend definierte Funktion übergeben können (siehe nächsten Abschnitt).

Das vollständige Programm

Damit ist das Programm fertig und lauffähig. Sie können es von der Konsole oder auch aus einem Dateimanager aufrufen (vorausgesetzt, der Perl-Interpreter ist als Standardbearbeiter für .pl-Dateien eingerichtet).

Listing 12.1: hallotk.pl

```
#!/usr/bin/perl 1
use strict;

use Tk;

my $hauptfenster = new MainWindow;

my $schalter = $hauptfenster->Button(
                                "-text" => "Drück mich",
                                "-command" => sub {exit 0} );

   $schalter->pack();


MainLoop;
```

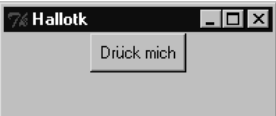


Abb. 12.1: Das Fenster von hallotk.pl

12.2 Furby

Mit Hilfe der grafischen Möglichkeiten, die uns das Windowing-System und Tk bieten, wollen wir jetzt unser Furby-Programm aus Kapitel 5.1.3 ein wenig aufpeppen. Im ersten Schritt soll Furby ein Gesicht erhalten.

Furby1 – Anzeigen von Bitmaps

Zuerst brauchen wir ein Bild eines Furbys. Sie können dazu einen Furby einscannen, das Bild aus der ZIP-Datei mit den Beispielen zum Buch nehmen oder sich mit Hilfe Ihres Browsers eine Furby-Abbildung von der offiziellen Furby-Site, *www.furby.com*, herunterladen (und als bmp-Datei speichern).

---

1 Ich habe den Warnmodus des Interpreters ausgeschaltet, um störende Warnungen aus dem Tk-Modul zu unterdrücken.

Um das Furby-Bild anzuzeigen, gehen wir in zwei Schritten vor:

1. Wir laden das Bild aus seiner Datei in ein `Photo`-Objekt. Die Option zur Angabe des Bilddateinamens lautet `-file`. Statt in der `Photo`-Methode kann man die Option auch nachträglich als Argument der `configure`-Methode angeben:

```
my $furby_bild = $hauptfenster->Photo();
   $furby_bild->configure( "-file" => "furby.bmp");
```

2. Wir kopieren das Bild aus dem `Photo`-Objekt in den Schalter. Dazu benutzen wir die `-image`-Option des Schalters. Zudem passen wir die Breite und Höhe des Schalters an die Abmaße des Bildes an. (Die Größe des Fensters wird automatisch an die Abmaße der im Fenster enthaltenen untergeordneten Fenster angepasst.)

```
my $schalter = $hauptfenster->Button(
                         "-image" => $furby_bild,
                         "-width" => $furby_bild->width,
                         "-height" => $furby_bild->height,
                         "-command" => sub {exit 0} );
```

Das vollständige Programm sieht wie folgt aus:

Listing 12.2: furby1.pl

```
#!/usr/bin/perl
use strict;

use Tk;

my $hauptfenster = new MainWindow;

my $furby_bild = $hauptfenster->Photo();
   $furby_bild->configure( "-file" => "furby.bmp");


my $schalter = $hauptfenster->Button(
                         "-image" => $furby_bild,
                         "-width" => $furby_bild->width,
                         "-height" => $furby_bild->height,
                         "-command" => sub {exit 0} );
   $schalter->pack( );


MainLoop;
```

Abb. 12.2: Furby1.pl

Furby2 – Bilder wechseln

Um den Furby etwas interessanter zu machen, kopieren Sie das Furby-Bild mehrere Male und bearbeiten die Kopien in einem geeigneten Grafikprogramm. Zeichnen Sie Sprechblasen in die Kopien und geben Sie verschiedene kurze Texte in die Sprechblasen ein.

Die einzelnen Bilder werden im Programm in ein Array, `@furbys`, geladen.

```
my @furbys = ();

for (my $loop=0; $loop <= 4; $loop++) {
   my $bild = $hauptfenster->Photo();
   my $name = "furby" . $loop . ".bmp";
   $bild->configure( "-file" => $name);
   push (@furbys, \$bild);
   }
```

Wenn der Anwender jetzt den Furby-Schalter anklickt, soll nicht mehr das Programm beendet, sondern stattdessen aus dem Array der Furby-Bilder ein Bild zufällig ausgewählt und angezeigt werden.

```
my $schalter = $hauptfenster->Button(...
                                    "-command" => \&klick);
   $schalter->pack( );

...
sub klick {
  my $index = rand(5);
  $schalter->configure("-image" => ${$furbys[$index]} );
  }
```

Das vollständige Programm sieht jetzt so aus:

Listing 12.3: furby2.pl

```
#!/usr/bin/perl
use strict;

use Tk;

my $hauptfenster = new MainWindow;

my $furby_bild = $hauptfenster->Photo();
   $furby_bild->configure( "-file" => "furby.bmp", );

my @furbys = ();

for (my $loop=0; $loop <= 4; $loop++) {
   my $bild = $hauptfenster->Photo();
   my $name = "furby" . $loop . ".bmp";
   $bild->configure( "-file" => $name);
   push (@furbys, \$bild);
   }


my $schalter = $hauptfenster->Button("-image" => $furby_bild,
                           "-width" => $furby_bild->width,
                           "-height" => $furby_bild->height,
                           "-command" => \&klick);
   $schalter->pack( );


MainLoop;


sub klick {
  my $index = rand(5);
  $schalter->configure("-image" => ${$furbys[$index]} );
  }
```

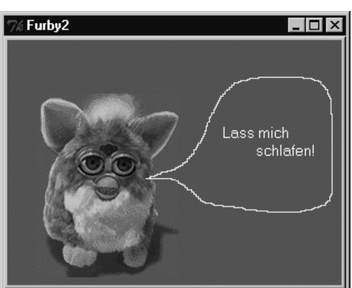


Abb. 12.3: Furby2.pl

Furby3 – Periodische Ausführung von Code

Zum Schluss wollen wir das Programm dahingehend abändern, dass unser Furby sich automatisch in bestimmten zeitlichen Abständen zu Wort meldet. Hierbei hilft uns und die Tk-Methode after:

```
Tk::after(5 * 1000, \&timer);
```

Die Methode after erwartet als Argumente eine Zeitangabe (in Millisekunden) und eine Funktionsreferenz. Wenn after aufgerufen wird, wartet die Funktion so lange, wie im ersten Argument angegeben, und ruft dann die übergebene Funktion auf. Wenn man after in der Funktion aufruft, die man auch als Referenz an after übergibt, erreicht man, dass die Funktion in periodischen Zeitabständen aufgerufen wird.

```
sub timer {
     my $index = rand(5);
     $schalter->configure("-image" => ${$furbys[$index]} );
     print "\a";
     Tk::after(5 * 1000, \&timer);
  }
```

Die Funktion timer (für Zeitgeber) wählt aus dem Array @furbys ein Bild aus, kopiert es in den Schalter, gibt einen Signalton aus und ruft sich dann auf dem Umweg über after selbst wieder auf.

An sich müssten wir after jetzt nur noch irgendwo im Skript einmal aufrufen, um den Timer-Mechanismus in Gang zu setzen, doch da dies unsere endgültige Furby-Version sein wird, gestatten wir uns gleich etwas mehr Komfort. Wir wollen den Mechanismus so einrichten, dass er ein- und ausgeschaltet werden kann.

Dazu definieren wir eine Variable $an, die nur die Werte 0 und 1 annehmen soll. Der Wert 0 bedeutet, dass der Timer ausgeschaltet ist, der Wert 1 bedeutet, dass der Timer angeschaltet ist. Um den Timer ein- und auszuschalten, definieren wir die Funktion klick.

```
my $an = 0;
...

sub klick {
  if (!$an) {
     $an = 1;
     Tk::after(5 * 1000, \&timer);
     }
    else {
     $an = 0;
     }
  }
```

Wenn `klick` aufgerufen wird und der Timer aus ist (`!an`), wird `$an` auf 1 und der Timer in Gang gesetzt. Wenn `klick` aufgerufen wird, wenn der Timer an ist, wird `$an` auf 0 gesetzt. Damit dies auch zum Ausschalten des Timers führt, prüfen wir in der Timer-Funktion den Wert von `$an`:

```
sub timer {
  if ($an) {
     my $index = rand(5);
     $schalter->configure("-image" => ${$furbys[$index]} );
     print "\a";
     # $hauptfenster->focusForce();
     Tk::after(5 * 1000, \&timer);
     }
  }
```

Jetzt müssen wir die `klick`-Funktion nur noch als Callback-Funktion des Furby-Schalters einrichten, und der Anwender kann den Mechanismus über den Furby-Schalter ein- und ausschalten.

```
my $schalter = $hauptfenster->Button(
                                ...
                                "-command" => \&klick);
```

Hier das vollständige und endgültige Furby-Programm:

Listing 12.4: furby3.pl

```
#!/usr/bin/perl
use strict;

use Tk;

my $an = 0;

my $hauptfenster = new MainWindow;

my $furby_bild = $hauptfenster->Photo();
   $furby_bild->configure( "-file" => "furby.bmp", );

my @furbys = ();

for (my $loop=0; $loop <= 4; $loop++) {
   my $bild = $hauptfenster->Photo();
   my $name = "furby" . $loop . ".bmp";
   $bild->configure( "-file" => $name);
   push (@furbys, \$bild);
   }
```

```
my $schalter = $hauptfenster->Button(
                              "-image" => $furby_bild,
                              "-width" => $furby_bild->width,
                              "-height" => $furby_bild->height,
                              "-command" => \&klick);
   $schalter->pack( );


MainLoop;


sub klick {
  if (!$an) {
     $an = 1;
     Tk::after(5 * 1000, \&timer);
     }
    else {
     $an = 0;
     }
  }

sub timer {
  if ($an) {
     my $index = rand(5);
     $schalter->configure("-image" => ${$furbys[$index]} );
     print "\a";
     # $hauptfenster->focusForce();
     Tk::after(5 * 1000, \&timer);
     }
  }
```

Abb. 12.4: Den Timer-Mechanismus kann man hier zwar nicht sehen, aber ich wollte Ihnen dieses Bild nicht vorenthalten

12.3 Eingabemasken für Programme

Furbys sind klein, knuddelig und ziemlich unnütz – genauso wie Furby-Programme. Sinnvoller sind dagegen Perl-Programme, die grafische Oberflächen zur Ein- und Ausgabe nutzen. Wie dies geht, demonstriert das folgende Beispiel.

Ich habe das Listing mit einer Zeilennummerierung versehen, um in der nachfolgenden Analyse einen eindeutigen Bezug zu den Zeilen des Programms herstellen zu können.

Listing 12.5: zinsen2.pl

```
1:  #!/usr/bin/perl
2:  use strict;
3:
4:  use Tk;
5:
6:  my $hauptfenster = new MainWindow;
7:
8:  my $linkerTeil = $hauptfenster->Frame();
9:     $linkerTeil->Label("-text" =>
                         "Startkapital:          ")->pack();
10:    my $kapital = $linkerTeil->Entry();
11:       $kapital->pack();
12:
13:    $linkerTeil->Label("-text" =>
                        "Zinssatz (in Prozent): ")->pack();
14:    my $zsatz = $linkerTeil->Entry();
15:       $zsatz->pack();
16:
17:    $linkerTeil->Label("-text" =>
                         "Laufzeit (in Jahren):  ")->pack();
18:    my $laufzeit = $linkerTeil->Entry();
19:       $laufzeit->pack();
20:
21: my $rechterTeil = $hauptfenster->Frame();
22:    my $daten = $rechterTeil->ScrlListbox("-label" =>
                                     "Kapitalentwicklung");
23:       $daten->pack();
24:
25:
26: my $schalter = $hauptfenster->Button(
                          -text => "Drück mich",
                          -command => \&zinsen_berechnen );
27:
```

```
28:
29: $linkerTeil->pack(-side => "left");
30: $rechterTeil->pack(-side => "right");
31: $schalter->pack(-expand => "yes", -padx => "5m",
                                     -pady => "5m");
32:
33:
34: MainLoop;
35:
36:
37: sub zinsen_berechnen {
38:   my $kap = $kapital->get;
39:   my $zs  = $zsatz->get;
40:   my $j   = $laufzeit->get;
41:   my $endkapital = 0;
42:
43:  for(my $loop = 0; $loop <= $j; $loop++) {
44:
45:     my $endkapital = $kap * ( 1 + $zs/100.0 * $loop);
46:
47:     $daten->insert("end", " " . $endkapital);
48:     }
49:  }
```

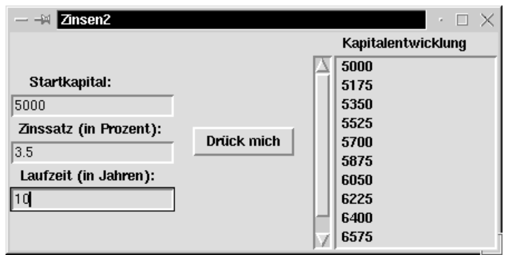


Abb. 12.5: Ein- und Ausgabe des Zinsen-programms

Analyse Das Auffälligste an diesem Programm ist, dass es über eine Vielzahl von Steuerelementen verfügt. Ein Großteil des Codes dieses Programms besteht daher darin, diese Steuerelemente zu erzeugen und korrekt anzuordnen.

Zur Anordnung von Steuerelementen bedient man sich

- ✘ zum einem sogenannter Container-Fenster (die über die Methode `Frame` erzeugt werden),
- ✘ zum anderen der `pack`-Methode der einzelnen Fenster, in der man angeben kann, wie das Fenster im übergeordneten Fenster auszurichten ist.

Das Hauptfenster des Zinsenprogramms ist in drei Teile aufgeteilt: einen linken Frame, einen rechten Frame und dazwischen ein Schalter:

```
8:  my $linkerTeil  = $hauptfenster->Frame();
21: my $rechterTeil = $hauptfenster->Frame();
26: my $schalter    = $hauptfenster->Button(...)

29: $linkerTeil->pack(-side => "left");
30: $rechterTeil->pack(-side => "right");
31: $schalter->pack(-expand => "yes", -padx => "5m",
                                      -pady => "5m");
```

Die Frames werden mit Hilfe der Option `-side` auf die linke und rechte Seite verbannt.

Damit der Schalter nicht am oberen Rand des Hauptfensters klebt, sondern zentriert erscheint, wird er mit Hilfe der Option `-expand` so weit ausgedehnt, dass er den gesamten ihm zur Verfügung stehenden Platz einnimmt. Die Optionen `-padx` und `-pady` sorgen dafür, dass der Schalter etwas Abstand zu den umliegenden Frames hat.

In die Frames werden die eigentlichen Steuerelemente eingefügt.

```
8:  my $linkerTeil = $hauptfenster->Frame();
9:     $linkerTeil->Label("-text" =>
                         "Startkapital:          ")->pack();
10:    my $kapital = $linkerTeil->Entry();
11:       $kapital->pack():
12:    ...
```

In den linken Frame werden drei Labels (`Label`) und drei Eingabefelder (`Entry`) eingefügt. Die Labels dienen als Beschriftungen zu den Eingabefeldern. Da wir im weiteren Verlauf nicht mehr auf die Labels zugreifen müssen, lohnt es sich nicht, eine Referenz auf das erzeugte Label-Objekt abzuspeichern (wie wir es für die Eingabefelder machen, denn schließlich müssen wir später noch deren Wert abfragen). Den `pack`-Aufruf hängen wir daher direkt an den Aufruf der Label-Methode an (die in so einem Fall das Objekt repräsentiert, das sie erzeugt).

In den rechten Frame nehmen wir ein Listenfeld auf. In diesem Listenfeld soll später die berechnete Kapitalvermehrung angezeigt werden.

```
21: my $rechterTeil = $hauptfenster->Frame();
22:    my $daten = $rechterTeil->ScrlListbox("-label" =>
                                       "Kapitalentwicklung");
23:       $daten->pack();
```

Zum Schluss wird eine Funktion zur Berechnung der Kapitalentwicklung aufgesetzt und als Callback-Funktion des Schalters eingerichtet:

```
26: my $schalter = $hauptfenster->Button(
                           -text => "Drück mich",
                           -command => \&zinsen_berechnen );
27:
28: ...

36:
37: sub zinsen_berechnen {
38:   my $kap = $kapital->get;
39:   my $zs  = $zsatz->get;
40:   my $j   = $laufzeit->get;
41:   my $endkapital = 0;
42:
43:  for(my $loop = 0; $loop <= $j; $loop++) {
44:
45:     my $endkapital = $kap * ( 1 + $zs/100.0 * $loop);
46:
47:     $daten->insert("end", " " . $endkapital);
48:     }
49:  }
```

In der Funktion `zinsen_berechnen` werden zuerst die Werte aus den Eingabefeldern abgefragt (mit Hilfe der `Entry`-Methode `get`, die den Inhalt des Eingabefeldes zurückliefert).

In einer `for`-Schleife wird dann berechnet, wie sich das Kapital Jahr für Jahr bis zum Ende der Laufzeit vermehrt (im Gegensatz zum Zinsprogramm aus Kapitel 3 gehen wir hier davon aus, dass die Zinserträge auf ein separates Konto gehen und nicht mit verzinst werden). Die berechneten Beträge werden mit Hilfe der `insert`-Methode in das Listenfeld kopiert.

12.4 Fragen und Übungen

1. Welches Modul benötigt man für die Erstellung von fensterbasierten Programmen mit Perl?
2. Wie erzeugt man das Hauptfenster einer Anwendung?
3. Was bewirkt der Aufruf `MainLoop`?
4. Wie lauten die Methoden zur Erzeugung von Eingabefeldern, Labels, Bildern, Schaltern, Container-Fenstern?
5. Wie kann man ein Steuerelement konfigurieren?
6. Wie kann man Steuerelemente in einem Fenster arrangieren?

Teil III

Praxis

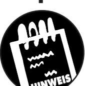
Kapitel 13: Grafik: Diagramme erstellen
Kapitel 14: Datenbank: CDs verwalten
Kapitel 15: Internet: Informationen aus Webseiten zusammenstellen
Kapitel 16: CGI: Perl-Programme für Websites
Kapitel 17: CGI: Ein Gästebuch

Den offiziellen Teil dieses Lehrbuchs erkläre ich hiermit für abgeschlossen. Wenn Sie möchten, können Sie das Buch jetzt beiseite legen und Ihrer eigenen Perl-Wege gehen. Sie können aber auch noch ein Weilchen bei diesem Buch verweilen und sich mit mir zusammen ein paar Beispiele für den praktischen Einsatz von Perl-Programmen anschauen.

Kapitel 13

Grafik: Diagramme erstellen

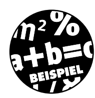
Grafikprogrammierung in Perl ist nicht notwendigerweise an das Vorhandensein eines Windowing-Systems und die Programmierung mit dem `Tk`-Modul gebunden. Mit Hilfe des Moduls `GD` können auch Konsolenprogramme Grafiken erzeugen und bearbeiten. Das einzige Manko dabei ist, dass das Konsolenprogramm die Grafik in Ermangelung einer grafischen Anzeigefläche blind bearbeiten muss. Mit `GD` kann man also keine Grafikprogramme schreiben, in denen der Anwender mit Hilfe der Maus interaktiv eigene Zeichnungen erstellen kann (dazu bedarf es des `Tk`-Moduls und eines Canvas-Objekts). Wohl aber kann man Perl-Programme schreiben, die selbständig Grafiken aufbauen und als Bilddatei, genauer gesagt als PNG-Datei, abspeichern.



Es gibt verschiedene Versionen des `GD`-Moduls, die sich darin unterscheiden, dass sie die erzeugten Grafiken als PNG- oder als GIF-Bilder speichern. Wenn Sie mit einer Version arbeiten, die GIF-Bilder erzeugt, müssen Sie in den nachfolgend beschriebenen Skripten statt der Methode `png` die Methode `gif` verwenden.

Daten in Diagramme umwandeln

Ein typisches Einsatzgebiet für Programme, die selbständig Grafiken erzeugen, ist die grafische Visualisierung von Daten, spricht die Erstellung von Diagrammen auf der Grundlage tabellarischer Zahlenkolonnen. Hierfür gibt es ein eigenes Perl-Modul, `Chart`, das auf dem `GD`-Modul aufsetzt und das Sie höchstwahrscheinlich erst einmal aus dem CPAN (oder über ActiveState) herunterladen müssen (siehe Anhang B).



Zur Belohnung können wir dann in fünf Minuten aus nüchternen Zahlen ansprechende Grafiken erstellen. Für das erste Beispiel habe ich eine kleine Tabelle ausgewählt, die anzeigt, wie sich der Gesamtverbrauch (in GWh) an regenerativen Energieträgern (Wasserkraft, Windkraft, Sonnenenergie etc.) in den Jahren von 1990 bis 1997 entwickelt hat.

```
Jahr               1990   1992   1994   1996   1997
Regen. Energien   58000  55000  71000  78000  79000
```

Das folgende Perl-Skript zaubert mit Hilfe des `Chart`-Moduls aus diesen Daten ein Diagramm.

Die Zeilennummerierung dient lediglich der besseren Orientierung bei der anschließenden Analyse.

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  use Chart::Bars;
5:
6:  # x-Achsenbeschriftung
7:  my @jahre     = qw(  1990  1992  1994  1996  1997);
8:  # Datensatz
9.  my @verbrauch = qw( 58000 55000 71000 78000 79000);
10:
11: my $diagr = Chart::Bars->new(400,200);
12:
13: $diagr->add_dataset(@jahre);
14: $diagr->add_dataset(@verbrauch);
15:
16: $diagr->png("Verbrauch.png");
```

Analyse Beginnen wir mit den Zeilen 7 und 9. In diesen Zeilen werden die Daten zusammengestellt, auf deren Grundlage das Diagramm erstellt wird. Man hätte die Daten auch aus einer Datei auslesen können, aber für ein erstes einführendes Beispiel, in dem es um das `Chart`-Modul und nicht um irgendwelche Einleseroutinen geht, ist es so sicherlich übersichtlicher. Wichtig ist, dass die Datenreihen in Arrays gespeichert werden – und zwar sowohl die Texte für die Beschriftung der x-Achsenticks als auch die Zahlen des Datensatzes, der grafisch dargestellt werden soll.

Kommen wir nun zur eigentlichen Erstellung des Diagramms. Zuerst wird in Zeile 4 das Modul `Chart::Bars` eingebunden. Tatsächlich könnten wir mit einem Modul `Chart` gar nichts anfangen. Wir müssen uns für eines der zu `Chart` gehörenden untergeordneten Module entscheiden, die die verschiedenen Diagrammtypen repräsentieren.

Chart-Modul	Diagrammtyp
`Points`	Datensätze werden als freie Punkte dargestellt.
`Lines`	Datensätze werden als Linienzug dargestellt.
`LinesPoints`	Datensätze werden als Linienzug mit Punkten dargestellt.
`Bars`	Datensätze werden als Balken dargestellt.
`StackedBars`	Datensätze werden als aufeinander gestapelte Balken dargestellt.

Tabelle 13.1: Chart-Module

In Zeile 11 wird dann ein Diagramm-Objekt erzeugt (siehe Kapitel 11 zur objektorientierten Programmierung). Dem `new`-Konstruktor übergeben wir gleich die gewünschte Breite und Höhe des Diagramms (in Pixel).

Jetzt werden dem Diagramm-Objekt die zu verarbeitenden Daten übergeben. Dazu verwendet man die Methode `add_dataset`. Dem ersten Datensatz, den `add_dataset` als Argument entgegennimmt (Zeile 13), kommt dabei eine besondere Bedeutung zu:

- ✘ Er enthält die Beschriftungen für die x-Achsenticks.
- ✘ Er gibt vor, wie viele Daten in den weiteren Datensätzen vorhanden sein müssen.

Die nachfolgend übergebenen Datensätze (Zeile 14) werden dann – je nach Diagrammtyp – zu Linien, Punktfolgen, Balkenfolgen etc. verarbeitet.

Zum guten Schluss speichert man das Diagramm in einer Grafikdatei. In Zeile 16 wird dazu die Methode `png` aufgerufen, die das Diagramm im PNG-Format speichert. (Linux-Versionen des Chart-Moduls verwenden meist `gif`, siehe Hinweis am Anfang des Kapitels.)

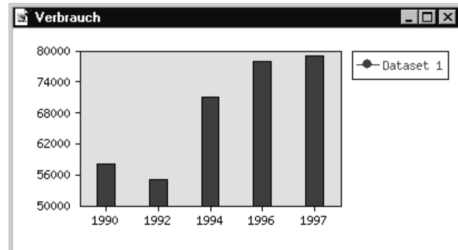


Abb. 13.1: Erste Version des Diagramms

Na, sieht das nicht viel ansprechender aus? Die PNG-Datei können Sie mit jedem Grafikprogramm betrachten, das das PNG-Format unterstützt. Sie können die Grafik aber auch in gedruckten Publikationen verwenden, als Slide oder Folie für einen Vortrag aufbereiten oder in Ihre Website einbin-

den. Allerdings sollten Sie dann etwas mehr Wert auf die Beschriftung des Diagramms legen.

Diagramme beschriften

Das folgende Perl-Skript bereitet den Verbrauch an Primärenergieträgern in Deutschland als gestapelte Balken auf. Angesichts des beschlossenen Atomausstiegs ist es ja nicht uninteressant, welche Mengen die Kernenergie im Vergleich zu anderen Energieträgern Jahr für Jahr produziert.

Ausgangspunkt ist die folgende Tabelle, die, so wie sie hier abgebildet ist, in der Datei ENERGIEN.TXT[1] gespeichert ist und dem Skript beim Aufruf über die Kommandozeile übergeben wird.

Listing 13.1: energien.txt

```
Primärenergieverbrauch in Deutschland absolut (in GWh)
Quelle: Arbeitsgemeinschaft Energiebilanzen

 ;                   1990;    1992;    1994;    1996;    1997;
Steinkohle;        641000;  610000;  595000;  578000;  568000;
Braunkohle;        890000;  605000;  517000;  468000;  442000;
Kernenergie;       463000;  481000;  459000;  490000;  517000;
Mineralöl;        1456000; 1564000; 1579000; 1612000; 1592000;
Erdgas;            637000;  662000;  714000;  878000;  830000;
Regen. Energien;    58000;   55000;   71000;   78000;   79000;
```

Das Skript liest diese Daten ein und erzeugt daraus Arrays für die x-Achsenticks, die einzelnen Datensätze und die Legende (Beschriftung der Datensätze). Der Code zum Einlesen der Daten ist so angelegt, dass das Skript keinerlei Annahmen über die Anzahl der Datensätze oder die Anzahl der Daten pro Datensatz macht. Wenn beispielsweise irgendwann der Datensatz »Kernenergie« wegfallen oder die Sparte »Regen. Energien« weiter aufgeteilt werden sollte oder einfach nur die Daten für das Jahr 1998 angehängt werden, braucht das Skript nicht überarbeitet zu werden.

Listing 13.2: diagramm.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  use Chart::StackedBars;
5:
6:  my $diagr = Chart::StackedBars->new(600,400);
7:
8:  my @legende;
9:  my %farben;
10: my $titel;
```

1 Dass die Daten nur bis zum Jahre 1997 reichen, liegt daran, dass ich im Internet keine aktuelleren Daten gefunden habe.

```
11:
12: # Titel
13:  chomp($titel = <>);
14:  $_ = <>; $_ = <>;
15:
16: # x-Achsenticks
17:  $_ = <>;
18:  chomp;
19:  my @achse = split(/\s*;\s*/);
20:  shift(@achse);
21:  $diagr->add_dataset(@achse);
22:
23: # Datensätze
24:  my $zaehler = 0;
25:  while (<>) {
26:   chomp;
27:   next if $_ eq '';
28:   my @felder = split(/\s*;\s*/);
29:   push(@legende, shift(@felder));
30:   $farben{"dataset$zaehler"} = [int(rand(255)),
31:                                 int(rand(255)),
32:                                 int(rand(255))];
33:   $diagr->add_dataset(@felder);
34:   $zaehler++;
35:   }
36:
37: $diagr->set('title' => $titel);
38: $diagr->set('x_label' => "Jahre");
39: $diagr->set('y_label' => "Energieverbrauch (GWh)");
40: $diagr->set('colors' => \%farben);
41: $diagr->set('legend_labels' => \@legende);
42: $diagr->png("Verbrauch.png");
```

Aufruf: **>perl diagramm.pl energien.txt** (Windows)

Aufruf: **# ./diagramm.pl energien.txt** (Linux)

Analyse

Statt die Balken der einzelnen Energieträger für jedes Jahr nebeneinander anzeigen zu lassen, sollen sie übereinander gestapelt werden. Aus diesem Grund wird in Zeile 4 das Modul `Chart::StackedBars` eingebunden. Nach der Erzeugung des Diagrammobjekts (Zeile 6) beginnt das Einlesen der Daten.

Als Erstes wird der Titel der Tabelle eingelesen (Zeilen 13) und in einer Variablen gespeichert. Später (in Zeile 37) werden wir den Tabellentitel als Diagrammtitel verwenden. Die beiden nächsten Zeilen werden überlesen.

Jetzt stehen wir in der Zeile mit den Jahreszahlen. Die Jahreszahlen benötigen wir für die x-Achsenticks. Als Erstes wird die Zeile eingelesen (Zeile 17), dann vom Zeilenumbruch befreit (Zeile 18) und in Zeile 19 in die einzelnen Felder aufgeteilt. Als Trennzeichen verwenden wir nicht einfach nur das Semikolon, sondern den regulären Ausdruck `/\s*;\s*/`, der ein Semikolon mit beliebig vielen Whitespaces vor und nach dem Komma darstellt. Dabei ist zu beachten, dass die Zeile mit einem leeren Feld beginnt. Dieses Feld wird mit Hilfe der `shift`-Funktion aus dem Array entfernt (Zeile 20). Danach wird das Array mit den Beschriftungen für die x-Achsenticks an das Diagrammobjekt übergeben.

Die noch verbliebenen Zeilen enthalten die eigentlichen Datensätze, die in einer `while`-Schleife eingelesen werden (Zeilen 23-35). Abgesehen davon, dass in Zeile 27 Leerzeilen übersprungen werden, werden die Zeilen mit den Datensätzen ähnlich verarbeitet wie die Zeile mit den Jahreszahlen: Die Datensätze werden an den Semikola aufgespalten, in ein Array geschrieben, um das erste Element verkleinert und an das Diagrammobjekt übergeben.

An sich wären wir damit schon fertig und könnten zu Zeile 37 weitergehen, wo das Diagramm mit Hilfe der Methode `set` konfiguriert und formatiert wird. Dann gibt es aber Probleme, wenn wir den Datensätzen Farben und Legendentexte zuweisen wollen (Zeilen 40 und 41). Prinzipiell könnte man das Hash für die Farben und das Array für die Legende zwar in den Zeilen 40 und 41 aufbauen, doch müsste man sich dann wohl oder übel bezüglich der Anzahl der Datensätze und deren Beschriftung festlegen. Besser und flexibler ist es, das Farben-Hash und das Legenden-Array beim Einlesen der Datensätze aufzubauen. Soweit es die Legende betrifft, können wir davon profitieren, dass die Beschriftungen der einzelnen Datensätze ja bereits in der ersten Spalte der Tabelle steht. Wir brauchen also nur das erste Feld des aktuellen Datensatzes, das wir mit `shift` aus dem Datensatz entfernen, in das Legenden-Array einzutragen (Zeile 29). Etwas komplizierter gestaltet sich der Aufbau des Farben-Hash. Das `Chart`-Modul schreibt nämlich vor, dass die Farben für die Datensätze die Schlüssel `dataset0` bis maximal `dataset15` tragen. Wir müssen die Schlüssel daher dynamisch in der Schleife erzeugen – und zwar als Kombination aus dem String `"dataset"` und dem Inhalt einer Zählervariablen: `"dataset$zaehler"`. Die Farbe selbst wird als RGB-Wert definiert, das heißt, man mischt die Farbe aus den drei Lichtfarben Rot, Grün und Blau. Für jeden Farbanteil kann man einen Wert zwischen 0 und 255 wählen – wobei im Listing die Wahl der Einfachheit halber dem Zufallsgenerator `rand` überlassen wird.

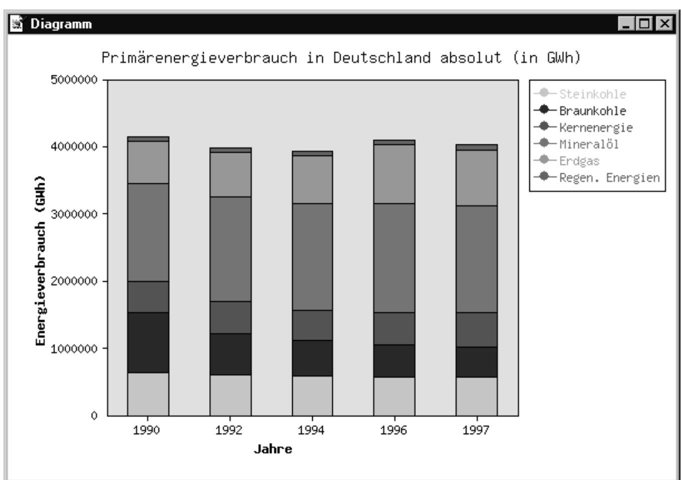


Abb. 13.2: Das formatierte Diagramm

Messergebnisse visualisieren

Als Naturwissenschaftler steht man häufig vor dem Problem, dass man in langwierigen Messreihen eine Menge von Daten gesammelt hat, die man hernach auf vernünftige Weise auswerten und – meist in Hinblick auf eine anvisierte Publikation – aufbereiten muss.

Stellen Sie sich vor, Sie wären Pharmakologe und hätten eine Messreihe durchgeführt, die Aufschluss darüber geben soll, wie sich die gleichzeitige Nahrungsaufnahme auf die Aufnahme eines bestimmten Medikaments auswirkt (sprich, ob nachher auf dem Beipackzettel stehen soll, dass die Tablette nüchtern oder beim Essen eingenommen werden soll).

Zuerst haben Sie Ihrem menschlichen Versuchskaninchen das Medikament mit dem Essen verabreicht. Ab dem Zeitpunkt der Einnahme haben Sie alle 15 Minuten die Konzentration des Medikaments im Blut des Probanden bestimmt. Eine analoge Messreihe haben Sie für die Einnahme des Medikaments auf nüchternen Magen durchgeführt. Die Messdaten haben Sie folgendermaßen in einer Datei notiert:

Listing 13.3: messdaten.txt

```
Ohne Nahrungsaufnahme
0,0
1,8
10,0
16,3
16,9
16,3
10,4
7,9
```

```
5,7
4,8
3,5
2,8
2,3
1,9
1,5
1,3
1,2

Mit Nahrungsaufnahme
0,0
0,05
0,4
1,1
3,2
5,5
7,8
8,3
8,6
8,3
7,9
7,2
6,0
5,2
4,6
3,8
3,2
```

Ein Perl-Skript soll nun aus diesen Daten eine Grafik erzeugen, mit der man beide Konzentrationsverläufe vergleichen kann.

Listing 13.4: messkurve.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  use Chart::LinesPoints;
5:
6:  my $diagr = Chart::LinesPoints->new(400,400);
7:
8:  my @achse;
9:  my @data1;
10: my @data2;
11:
12: undef $/;
13: my $text = <>;
14: $text =~ s/,/\./g;
```

```
15:
16: # Datensätze einlesen
17: $text =~ m/Ohne Nahrungsaufnahme(.*?)Mit
                                             Nahrungsaufnahme/s;
18: @data1 = $1 =~ /(\d+.\d*)/g;
19:
20: $text =~ m/Mit Nahrungsaufnahme(.*?)$/s;
21: @data2 = $1 =~ /(\d+.\d*)/g;
22:
23: # x-Achsenticks berechnen
24: for (my $loop=0; $loop <= $#data1; $loop++) {
25:    push(@achse, ($loop * 15));
26:    }
27:
28:
29: $diagr->set('title' => 'Resorption von Medikament XXX');
30: $diagr->set('y_label'=> 'Konzentration im Serum [mg/l]');
31: $diagr->set('x_label' => 'Zeit [min]');
32: $diagr->set('legend_labels' => ['Ohne Nahrungsaufnahme',
                                    'Mit Nahrungsaufnahme']);
33: $diagr->add_dataset(@achse);
34: $diagr->add_dataset(@data1);
35. $diagr->add_dataset(@data2);
36: $diagr->png("MedikamentXXX.png");
```

Aufruf: **>perl messkurve.pl messdaten.txt** (Windows)

Aufruf: **# ./messkurve.pl messdaten.txt** (Linux)

Analyse

Das Skript beginnt mit der Einbindung des Moduls `Chart::LinesPoints` und der Erzeugung eines entsprechenden Diagrammobjekts (Zeilen 4 und 6).

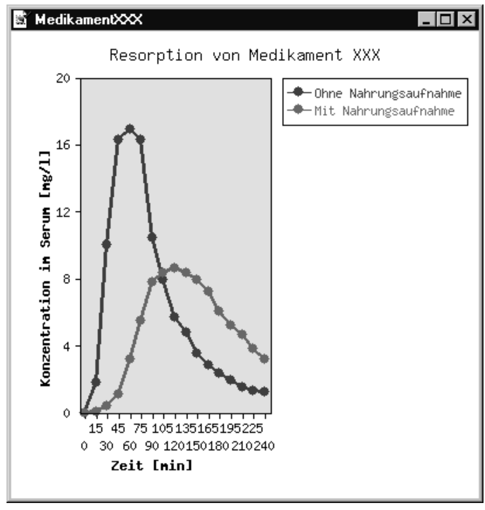
Dann werden die Daten der Messreihen eingelesen. Dazu wird der Text im Schlürfmodus (`undef $/;`) in einem Rutsch eingelesen und in der String-Variablen `$text` abgelegt. Für den Fall, dass der Autor der Datei nicht beachtet hat, dass Nachkommastellen in Perl nicht mit einem Komma, sondern einem Punkt abgetrennt werden, werden alle Kommas in Punkte umgewandelt (Zeile 14). (Beachten Sie den Escape-Operator \ vor dem Punkt. Ohne den Backslash würde Perl den Punkt als Metazeichen interpretieren.)

Als Nächstes werden die Daten der ersten Reihe in die Standardvariable `$1` ausgelesen (Zeile 17). Mit Hilfe des regulären Ausdrucks `/(\d+.\d*)/g` werden die einzelnen Zahlen identifiziert und – da der reguläre Ausdruck in einem Listenkontext ausgewertet wird – in das Array `@data1` kopiert (Zeile 18). In gleicher Weise werden die Messwerte der zweiten Messreihe extrahiert.

Schließlich muss noch das Array für die Achsenticks erzeugt werden. Aus Faulheit wurden die Zeitpunkte der Messungen in der Datei MESSDATEN.TXT nicht festgehalten. Dies ist nicht weiter schlimm, da wir ja wissen, dass alle 15 Minuten gemessen wurde. Wir können das Array für die Achsenticks daher ohne Probleme nachkonstruieren (Zeilen 23–26).

Zu guter Letzt wird das Diagrammobjekt konfiguriert (Zeilen 29–32), die Achsenticks und die Datensätze werden übergeben (Zeilen 33–35) und das Diagramm wird als PNG-Datei gespeichert (Zeile 36).

Abb. 13.3: Grafische Darstellung von Messkurven[2]



2 Im obigen Beispiel kann man sehen, dass die gleichzeitige Nahrungsaufnahme die Resorption des Medikaments verzögert und vermindert hat. Während die Einnahme eines Medikaments mit den Mahlzeiten meist dazu führt, dass die Resorption des Medikaments zeitlich verzögert erfolgt, kann das Ausmaß der Resorption je nach Art des Medikaments reduziert oder auch vergrößert werden. Daher sollte man stets die Hinweise des Beipackzettels beachten, da es sonst zu stark unterschiedlichen Dosen kommen kann.

Noch mehr Möglichkeiten

Wenn Sie mit den Möglichkeiten zur grafischen Aufbereitung, die Ihnen das `Chart`-Modul bietet, nicht zufrieden sind, stehen Ihnen folgende Wege offen:

- ✘ Sie können den Code des `Chart`-Moduls überarbeiten.
- ✘ Sie können auf die Funktionen des `GD`-Moduls zurückgreifen, mit deren Hilfe auch das Chart-Modul implementiert wurde.
- ✘ Sie können ein fensterbasiertes Programm schreiben (siehe Kapitel 12), in dem Programm ein `Canvas`-Objekt erzeugen und dessen Möglichkeiten ausschöpfen (siehe Dokumentation zum `Tk`-Modul).

Kapitel 14

Datenbank: CDs verwalten

Praktisch jede Anwendung manipuliert Daten – und wenn es nur zwei Integer-Werte sind, die multipliziert werden. Wenn die Datenmengen, die ein Programm zu verarbeiten hat, über ein bestimmtes Maß anwachsen, stellt sich irgendwann die Frage, wie man die Daten am besten verwaltet.

Daten im RAM verwalten

Im einfachsten Fall werden alle Daten während der Ausführung des Programms im Arbeitsspeicher gehalten. Aufbewahrt und organisiert werden die Daten dabei

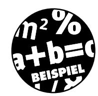
- ✘ in einzelnen Variablen (Skalaren)
- ✘ in Listen (Arrays und Hashes)
- ✘ oder in höheren Datenstrukturen (Array von Hashes etc.)

Daten extern speichern

Sollen Daten zwischen den Aufrufen des Programms persistent abgespeichert werden, sichert man sie üblicherweise in eine Datei. Dies kann eine einfache Binärdatei sein, in der Daten gleichen Typs lückenlos hintereinander abgelegt sind, oder eine Textdatei, in der die einzelnen Daten durch besondere Zeichen (beispielsweise Kommata) getrennt werden.

Mit Hilfe von Textdateien kann man auch schon kleine Datenbanken anlegen, indem man jeweils einen Datensatz in eine Zeile schreibt und die einzelnen Felder eines Datensatzes durch Kommata (oder ein anderes Zeichen) trennt (die meisten professionellen Datenbankprogramme können solche Dateien einlesen und in ihr eigenes Format konvertieren).

Diesen Weg geht auch das folgende Programm, mit dem man eine CD-Sammlung aufbauen kann. Grundlage des Programms ist die Textdatei CDS.CVS, in der jede Zeile einen Datensatz darstellt und die Daten zu einer



CD enthält (Name des Komponisten oder Interpreten, Titel der CD und Kategorie).

Listing 14.1: cds.cvs

```
Guiseppe Verdi,La Traviata,Klassik
Sergei Rachmaninov,Klavierkonzert Nr. 2,Klassik
Peter I. Tchaikovsky,Klavierkonzert Nr. 1,Klassik
Ludwig van Beethoven,Symphony Nr. 7,Klassik
George Winston,Ballads and Blues,Jazz
Dvorak,Symphony Nr. 9 (Aus der neuen Welt),Klassik
```

Mit dem zugehörigen Datenbankprogramm CDS.PL kann man:

- ✘ neue Datensätze eingeben
- ✘ sich eine CD vorschlagen lassen, wenn man einmal unentschlossen ist, was man eigentlich hören möchte
- ✘ sich die eingetragenen CDs in einer Liste ausgeben lassen
- ✘ das Programm beenden

Da das Programm rund 180 Zeilen enthält, werde ich mir nicht die Mühe machen, das Programm Zeile für Zeile mit Ihnen durchzugehen und zu analysieren, zumal die eingesetzten Techniken alle aus den vorangehenden Kapiteln bekannt sind (oder dort nachgelesen werden können). Ich werde daher nur den grundsätzlichen Aufbau des Programms erläutern und die Deutung der einzelnen Zeilen dem Leser als Wiederholungsübung überlassen.

Das Programm hat eine recht einfache Struktur. Am Anfang steht der Hauptcode des Skripts der vor allem aus der Implementierung eines Konsolenmenüs besteht. Über das Menü werden vier Funktionen aufgerufen, die im Anschluss an den Hauptcode des Skripts definiert sind. Beginnen wir mit dem Hauptcode:

```
1:   #!/usr/bin/perl -w
2:   use strict;
3:
4:   use vars qw($c $t $k $cd);
5:
6:   my $CD_db = "cds.cvs";    # Textdatenbank
7:   my @cds = ();             # Liste der CDs
8:
9:
10:  # CDs aus Textdatenbank einlesen
11:
12:  open(CD_FILE, $CD_db) or
13:         die "\nDatei $CD_db konnte nicht geoeffnet
                werden ($!)\n";
14:
```

```
15:  while(<CD_FILE>) {          # Zeile für Zeile einlesen
16:      my %cd = ();            # einzelne CD
17:
18:      chomp;
19:     ($c, $t, $k) = split(",", $_);
20:      $cd{composer}  = $c;              # Hash erzeugen
21:      $cd{titel}     = $t;
22:      $cd{kategorie} = $k;
23:
24:      push(@cds, \%cd);   # Referenz in Array eintragen
25:    }
26:
27:  close(CD_FILE);
28:
29:
30:  # Menue ausgeben und Menuebefehle verarbeiten
31:
32:  my $menue = <<HERE_MENUE;
33:
34:    CD eingeben               <1>
35:    CD vorschlagen            <2>
36:    Liste der CDs ausgeben    <3>
37:    Beenden                   <4>
38:
39:  HERE_MENUE
40:
41:
42:  my $eingabe = 0;
43:  do {
44:    print $menue;
45:    print "  Ihre Eingabe: ";
46:    chomp ($eingabe = <STDIN>);
47:
48:   SWITCH: {
49:   $eingabe == 1   && do { einlesen();
50:                           last SWITCH; };
51:   $eingabe == 2   && do { vorschlagen();
52:                           last SWITCH; };
53:   $eingabe == 3   && do { &liste_ausgeben();
54:                           last SWITCH; };
55:   $eingabe == 4   && do { &speichern();
56:                     print "\n  Programm wird beendet \n";
57:                     last SWITCH; };
58:    }
59:
60:   } while ($eingabe != 4);
61:
62:
```

Nach der Deklaration einiger globaler und dateiglobaler Variablen (Zeilen 4–7) öffnet das Skript die zugrunde liegende Textdatenbank (Zeile 12) und liest den Inhalt der Datei zeilenweise ein. Die eingelesenen Zeilen werden in ihre Felder zerlegt und zum Aufbau eines lokalen Hash verwendet (`my %cd`). Zum Schluss wird eine Referenz auf das Hash-Objekt in das dateiglobale Array `@cds` eingetragen (Zeile 24).

Das Skript liest also zuerst den Inhalt der Datenbank aus und baut daraus ein Array von Hash-Referenzen auf. Alle weiteren Funktionen des Programms operieren auf diesem Array (und nicht mehr auf dem Inhalt der Datenbankdatei). Erst beim Beenden des Programms schreibt das Skript den Inhalt des Arrays `@cds` in die Datei zurück. Dieses Verfahren hat seine Vor- und Nachteile. Ein Vorteil ist, dass die Operationen auf den Daten sehr schnell ablaufen, da die Daten alle im Arbeitsspeicher zur Verfügung stehen und nicht erst nach Bedarf eingelesen und zurückgeschrieben werden müssen. Nachteilig ist der erhöhte Speicherbedarf (der aber unkritisch sein sollte, solange Sie mit dem Programm nur Ihre persönliche Datenbank verwalten), so wie die Gefahr von Datenverlusten, wenn das Programm nicht korrekt beendet wird. Letzteres betrifft vor allem die Funktion zum Einlesen neuer Datensätze. Die Daten für die neu eingetragenen CDs stehen nämlich bis zum vorschriftsmäßigen Beenden des Programms nur im Arbeitsspeicher. Stürzt das Programm ab, gehen die eingetragenen Daten verloren.

Als nächster Schritt wird das Menü des Programms ausgegeben. Das Menü ist als HERE-Dokument definiert (Zeilen 32–39). In einer `do-while`-Schleife (Zeilen 43–60) wird das Menü angezeigt, die Eingabe des Anwenders eingelesen und die passende Funktion zur Bearbeitung des ausgewählten Menübefehls aufgerufen – so lange, bis der Anwender mit der Option zum Beenden (4) die Schleife verlässt und das Programm beendet.

Unter dem Hauptcode des Skripts sind die Funktionen zu den Menübefehlen definiert. Die erste dieser Funktionen dient dem Einlesen neuer Datensätze.

```
63:
64:  #****** Funktionen zu den Menuebefehlen ******
65:
66:  # einlesen
67:
68:  sub einlesen {
69:    my $eingabe;
70:    my %neueCD;
71:
72:    print "\n";
73:    print "  Geben Sie den Komponisten an: ";
```

```
74:    chomp ($eingabe = <STDIN>);
75:    $neueCD{composer} = $eingabe;
76:    print "\n";
77:    print "  Geben Sie den Titel an: ";
78:    chomp ($eingabe = <STDIN>);
79:    $neueCD{titel} = $eingabe;
80:    print "\n";
81:    print "  Geben Sie die Kategorie an (Klassik, Rock,
                                              Jazz): ";
82:    chomp ($eingabe = <STDIN>);
83:    $neueCD{kategorie} = $eingabe;

84:    push(@cds, \%neueCD);    # Referenz in Array eintragen
85:  }
86:
```

Die Funktion `einlesen` liest über die Tastatur die Werte für die Felder des neuen Datensatzes ein, speichert diese in einem lokalen Hash und trägt zum Schluss eine Referenz auf dieses Hash in das dateiglobale Array `@cds` ein.

Die zweite Funktion schlägt dem unentschlossenen Musikfreund eine CD aus der Sammlung vor.

```
87:
88:  # vorschlagen
89:
90:  sub vorschlagen {
91:
92:    # Kategorie zur Auswahl anbieten
93:
94:    print "\n  Waehlen Sie eine Kategorie aus: \n\n";
95:    print "\t Klassik   <1>\n";
96:    print "\t Rock      <2>\n";
97:    print "\t Jazz      <3>\n\n";
98:
99:    print "  Ihre Eingabe: ";
100:   chomp (my $eingabe = <STDIN>);
101:
102:   SWITCH: {
103:   $eingabe == 1   && do { $eingabe = "Klassik";
104:                           last SWITCH; };
105:   $eingabe == 2   && do { $eingabe = "Rock";
106:                           last SWITCH; };
107:   $eingabe == 3   && do { $eingabe = "Jazz";
1ß8:                           last SWITCH; };
109:   }
```

```
110:
111:  # Alle CDs der Kategorie in eigenes Array kopieren
112:
113:  my @kategorie;
114:  foreach my $cd (@cds) {
115:    if (%$cd->{kategorie} eq $eingabe) {
116:       push(@kategorie, $cd);
117:       }
118:    }
119:
120:  # Eine CD aus dem lokalen Array auswählen und
121:  # vorschlagen
122:
123:  if ($#kategorie < 0) {
124:    print "\n  Keine CDs in Kategorie gefunden\n\n";
125:    }
126:    else
127:    {
128:    my $ausgewaehlt = int(rand($#kategorie+1));
129:    print "\n  Wie waere es mit: ";
130:    print "'", %{$kategorie[$ausgewaehlt]}->{titel},
               "' von ";
131:    print %{$kategorie[$ausgewaehlt]}->{composer},
               "\n\n";
132:    }
133:}
134:
```

Die Funktion beginnt mit dem Anzeigen und Auswerten eines kleinen Menüs (Zeilen 92–109), das es dem Anwender erlaubt, eine Kategorie für die Auswahl der CD anzugeben. So kann sich der Anwender gezielt eine Klassik-, Rock- oder Jazz-CD empfehlen lassen.

Hat sich der Anwender für eine Kategorie entschlossen, geht das Skript das Array der CDs durch, sucht alle CDs heraus, die zu der angegebenen Kategorie gehören, und kopiert diese in ein eigenes Array `@kategorie` (Zeilen 113–118).

Zum guten Schluss wird – sofern sich das lokale Array nicht als leer erweist (Zeile 123) – mit Hilfe der `rand`-Funktion eine CD aus dem lokalen, kategoriespezifischen Array ausgewählt und dem Anwender vorgeschlagen (Zeilen 126–133).

```
135:
136: # Liste ausgeben
137:
138: format STDOUT_TOP =
```

```
139:
140:  Eingetragene CDs in @<<<<<<<<<<              Seite @>>
141:                     $CD_db,                       $%
142:
143: --------------------------------------------------------
144:  Komponist             Titel                  Kategorie
145: --------------------------------------------------------
146: .
147:
148: format STDOUT =
149:  @<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<   @<<<<<<<<<<<
150: $cd->{'composer'}, $cd->{'titel'}, $cd->{'kategorie'}
151:
152: .
153:
154: sub liste_ausgeben {
155:   $- = 0;
156:   $% = 0;
157:  $= = 25;
158:
159:  foreach $cd (@cds) {
160:    write STDOUT;
161:    }
162: }
163:
```

Die vollständige Ausgabe der eingetragenen CDs erfolgt mit Hilfe des Regport-Generators. Die verwendeten Formate und die Implementierung der Funktion wurden bereits in Kapitel 9.1.3 beschrieben.

Listing 14.2: cds.pl

```
164:
165: # speichern
166:
167: sub speichern {
168:
169:   open(CD_FILE, ">$CD_db")
170:      or die "\nDatei $CD_db konnte nicht geoeffnet
               werden ($!)\n";
171:
172:  foreach my $cd (@cds) {
173:    print CD_FILE  %$cd->{composer}, ",";
174:    print CD_FILE  %$cd->{titel}, ",";
175:    print CD_FILE  %$cd->{kategorie}, "\n";
176:    }
177:
178:  close(CD_FILE);
179:  }
```

Wenn der Anwender das Programm durch Auswahl der Option 4 beendet, ruft das Programm intern zuerst noch die Funktion `speichern` auf, die den aktuellen Inhalt des dateiglobalen Arrays `@cds` in die zugrunde liegende Textdatenbank schreibt. Der alte Inhalt wird dabei komplett überschrieben (`>$CD_db` statt `>>$CD_db`).

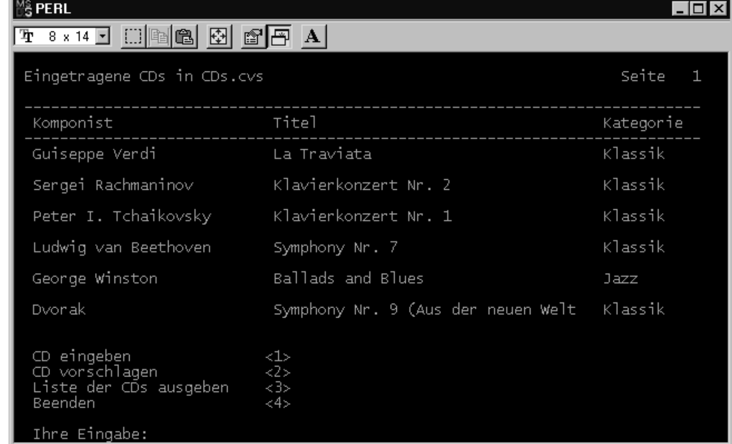


Abb. 14.1: Liste der aktuell enthaltenen Einträge und das Menü des Programms

Zugriff auf »echte« Datenbanken

Sind größere Datenmengen zu verwalten und steigen die Ansprüche an die Organisation der Daten in der Datenbank, empfiehlt es sich für die Verwaltung der Datenbank, eine spezielle Datenbank-Software (etwa Access oder Oracle) zu verwenden und mit dem eigenen Skript dann nur noch die Daten bei Bedarf aus der Datenbank abzufragen und eventuell neue oder geänderte Daten in die Datenbank zurückzuschreiben.

Hierzu benötigt man (neben der Datenbank) einen Treiber für die Datenbank sowie eine API, eine Sammlung von Funktionen, mit denen das Skript die Datenbank-Software steuern und auf die Daten in der Datenbank zugreifen kann. Eine solche API finden Sie beispielsweise im Modul `DBI`. Wenn Sie von Ihren Perl-Skripten auf echte Datenbanken zugreifen wollen, sollten Sie unbedingt mit dem Lesen der Dokumentation zum `DBI`-Modul beginnen.

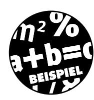
Kapitel 15

Internet: Informationen aus Webseiten zusammenstellen

Surfen Sie auch regelmäßig im Internet? Ja? Nein? Ach so, Sie gehören zu den Leuten, die ernsthaft mit dem Internet arbeiten. Dann gibt es bestimmt eine Reihe von Informationen, die Sie immer wieder in mehr oder weniger regelmäßigen Abständen abrufen, und jedes Mal ärgern Sie sich, dass diese Informationen auf mehrere Webseiten verteilt sind, die Sie alle nacheinander ansteuern müssen. Nun, in diesem Fall kann vielleicht ein kleines Perl-Skript abhelfen.

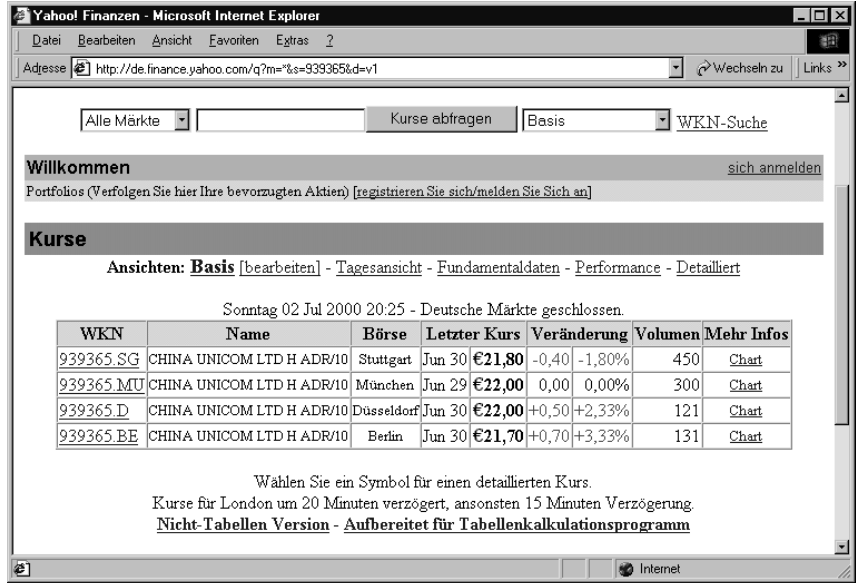
URLs zusammenstellen

Das Zusammentragen der URLs der Webseiten, die für Sie interessant sind, dürfte wohl keine Schwierigkeit darstellen. Steuern Sie die Webseiten einfach mit Ihrem Browser an und kopieren Sie die URLs aus dem Adressfeld des Browsers in eine Textdatei (bzw. gleich in Ihr Skript, siehe unten).



Das funktioniert übrigens nicht nur für einfache URLs mit Verzeichnisangaben, sondern auch mit URLs, die auf Skripten zugreifen. Yahoo-Deutschland bietet beispielsweise über den Link »Finanzen« einen Suchdienst zum Abfragen aktueller Aktienkurse an. Wenn Sie in das Suchfeld eine WKN eingeben und die Suche starten, erscheint eine spezielle Webseite mit Kursdaten und sonstigen aktuellen Informationen zu der betreffenden Aktie. Im Adressfeld des Browsers wird die komplette URL inklusive der Angaben für die Suchmaschine angezeigt. Für das nachfolgende Skript habe ich drei dieser URLs kopiert (China Mobile (WKN 909622), ChinaDotCom (WKN 924123) und Siemens (WKN 723610) und in einer Liste zusammengefasst.

Abb. 15.1: URLs bestimmen




```
my @urls =
       ("http://de.finance.yahoo.com/q?m=*&s=909622&d=v1",
        "http://de.finance.yahoo.com/q?m=*&s=924123&d=v1",
        "http://de.finance.yahoo.com/q?m=*&s=723610&d=v1");
```

Das folgende Perl-Skript soll diese drei Webseiten nacheinander aufrufen und den Inhalt der Seiten in einer gemeinsamen HTML-Seite speichern. Wenn man die Kurse der Aktien kontrollieren will, braucht man sich dann nur noch ins Internet einzuloggen und das Perl-Skript aufzurufen. Dieses trägt die gewünschten Informationen zusammen und speichert sie in einer Datei. Ist das Skript fertig, kann man die Internetsitzung beenden und sich die Inhalte der Webseiten gemütlich von der lokalen Festplatte aus ansehen.

Das Skript

Das komplette Skript ist erstaunlich kurz.

Listing 15.1: webinhalte.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4: use LWP::Simple;
5:
6: my @urls =
7:        ("http://de.finance.yahoo.com/q?m=*&s=909622&d=v1",
8:         "http://de.finance.yahoo.com/q?m=*&s=924123&d=v1",
9:         "http://de.finance.yahoo.com/q?m=*&s=723610&d=v1");
```

```
10: my $html;
11:
12: $html = "<html><title>Aktienkurse</title><body>";
13:
14: foreach my $url (@urls) {
15:  my $doc;
16:
17:  $doc = get($url);
18:
19:  $doc =~ m/<BODY>(.*)<\/BODY>/si;
20:  $html .= "$1";
21:  }
22:
23: $html .= "</body></html>";
24:
25: print $html;
```

In Zeile 4 wird das Modul `LWP::Simple` eingebunden. Dies stellt uns die Funktion `get` zur Verfügung, mit der man sich den Inhalt einer Webseite zurückliefern lassen kann (Zeile 17).

In den Zeilen 6–9 wird das Array mit den URLs definiert (die Liste kann selbstverständlich nach Belieben erweitert werden).

In Zeile 10 wird die Variable `$html` deklariert. Darin werden die Webinhalte gespeichert. Da von den verschiedenen Webseiten stets nur der Text zwischen den `<body>`-Tags gespeichert wird, wird der Rest des HTML-Codegerüsts mit den `print`-Anweisungen in den Zeilen 12 und 24 in `$html` geschrieben.

Danach beginnt eine `foreach`-Schleife, in der die einzelnen URLs aus der Liste abgearbeitet werden. Zuerst wird die aktuelle URL in Zeile 17 an die Funktion `get` übergeben. Die Funktion ruft die Webseite auf (vorausgesetzt, es besteht eine Verbindung zum Internet und die URL ist korrekt), liest den Inhalt ein und liefert ihn als Ergebniswert zurück. Diesen Text speichern wir in der lokalen Variablen `$doc`. Der nächste Schritt besteht jetzt darin, aus dem HTML-Code der Seite den Text zwischen den `<body>`-Tags herauszutrennen. Klar, dass man in Perl dafür einen regulären Ausdruck verwendet (Zeile 19).

Im regulären Ausdruck aus Zeile 19 repräsentiert `.*` den Text zwischen den Body-Tags. Die Option `s` sorgt dafür, dass das Suchmuster .* auch über Zeilenumbrüche hinwegliest, die Option `i` sorgt dafür, dass nicht zwischen Groß- und Kleinschreibung unterschieden wird (in HTML hat die Groß- und Kleinschreibung keine Bedeutung: `<body>`, `<Body>` und `<BODY>` sind alles korrekte Schreibweisen). Da `.*` in runde Klammern gefasst ist, speichert der

reguläre Ausdruck den zugehörigen Text in der Standardvariablen `$1`. Dieser Text wird in Zeile 20 an den aktuellen Inhalt der Variablen `$hmtl` angehängt.

Zum Schluss gibt das Skript die gesammelten Webinhalte auf die Konsole aus (Zeile 25). Diese Ausgabe muss man dann nur noch bei Aufruf des Skripts in eine neue HTML-Datei umleiten.

Aufruf: **> perl webinhalte.pl > aktien.html** (Windows)

Aufruf: **# ./webinhalte.pl > aktien.html** (Linux)

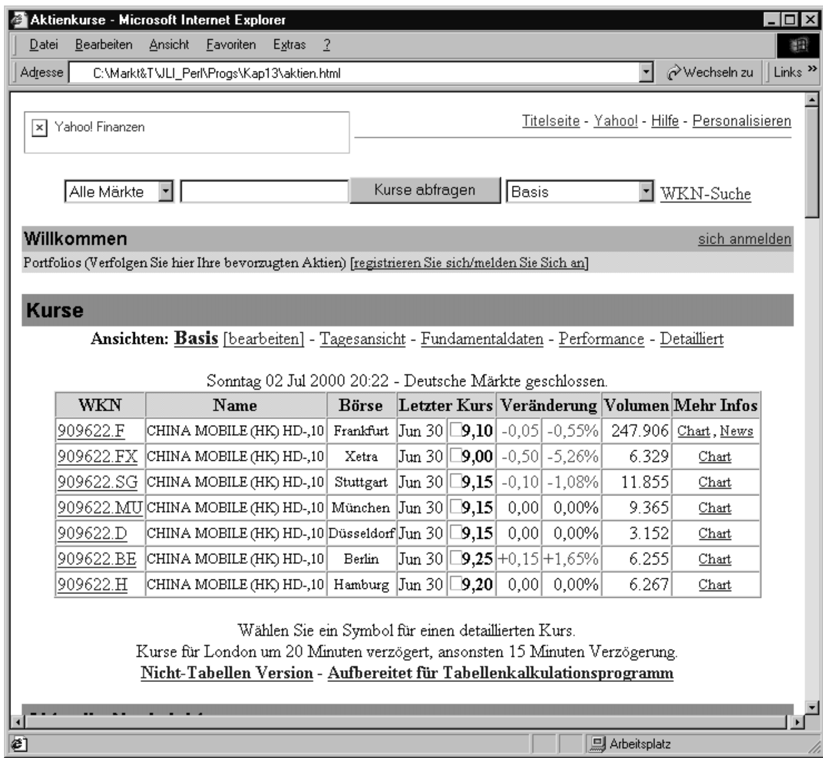


Abb. 15.2: Das erzeugte HTML-Dokument mit den Informationen der ersten angeforderten Webseite

Kapitel 16

CGI: Perl-Programme für Websites

Eines der Hauteinsatzgebiete für Perl ist mittlerweile die CGI-Programmierung geworden, d.h. die Erstellung von kleinen Programmen, die auf Webservern installiert und ausgeführt werden – meist um dynamische Webinhalte anzubieten.

Konfiguration des Webservers

Die vermutlich schwierigste Aufgabe bei der CGI-Programmierung ist die Einrichtung des Webservers. Dieser muss so konfiguriert werden, dass er die Ausführung von Programmen aus bestimmten Verzeichnissen gestattet und dass er zur Ausführung von Perl-Programmen den Perl-Interpreter heranzieht.

Die in diesem Abschnitt angegebenen Pfadangaben sind größtenteils Standardwerte, die bei der Installation verändert werden können. Es ist also durchaus möglich, dass auf Ihrem Rechner abweichende Pfade und Verzeichnisse verwendet werden.

Der erste Schritt besteht natürlich darin, den Server – soweit nicht schon vorhanden – zu installieren.

- ✘ Den Apache-Server für Linux (oder auch für Windows) können Sie beispielsweise von *http://www.apache.org* herunterladen. Vermutlich findet er sich aber auch auf Ihrer Linux-CD.

- ✘ Wenn Sie mit Windows arbeiten, werden Sie den PWS (Personal Web Server) oder den IIS (Internet Information Server) verwenden. Sofern

sich die entsprechenden Server nicht auf Ihrer Betriebssystem-CD finden, können Sie die Server-Software von der Microsoft-Website herunterladen (http://www.microsoft.de).

- ✘ Alternativ können Sie auch den Webserver OmniHttpd von der Website *http://www.omnicron.ab.ca* herunterladen und installieren (Doppelklick auf die heruntergeladene EXE-Datei).

Vergewissern Sie sich, dass der Webserver läuft.

- ✘ Für den Apache-Server rufen Sie je nach Version /ETC/RC.D/INIT.D /HTTPD START oder /USR/LOCAL/APACHE/BIN/APACHECTL START auf.
- ✘ Der PWS und der IIS werden üblicherweise automatisch gestartet und erscheinen beim Programmstart als Symbol in der Taskleiste. Über das Kontextmenü können sie angehalten und weiter ausgeführt werden. (Unter Windows NT können Sie über EINSTELLUNGEN/SYSTEMSTEUERUNG/DIENSTE nachsehen, ob der Server läuft.)
- ✘ Der OmniHTTPd kann bei der Installation so eingerichtet werden, dass er automatisch gestartet wird. Läuft der Server, erscheint in der Windows-Taskleiste ein entsprechendes Symbol für den Server. Läuft der Server nicht, kann er über seine Programmgruppe (Aufruf über START/ PROGRAMME) gestartet werden.

Testen Sie den Webserver. Erstellen Sie eine einfache HTML-Datei, beispielsweise:

Listing 16.1: test.html

```
<HTML>
<TITLE>Hallo vom Server</TITLE>

<BODY> Hallo! Test erfolgreich bestanden.</BODY>
</HTML>
```

Speichern Sie diese im Dokumenten-Verzeichnis des Servers und versuchen Sie, die Webseite über den Server in Ihren Browser zu laden.

- ✘ Die Konfiguration des Apache-Servers erfolgt größtenteils über die Datei HTTPD.CONF. Diese Datei befindet sich je nach Version in einem der Verzeichnisse /ETC/HTTPD/CONF oder /USR/LOCAL/APACHE/CONF. Suchen Sie in der Datei nach einem Eintrag für DocumentRoot. Gibt es keinen entsprechenden Eintrag, verwendet der Server höchstwahrscheinlich /HOME/HTTPD/HTML oder /USR/LOCAL/APACHE/HTDOCS als Dokumentenverzeichnis. Nachdem Sie die HTML-Seite abgespeichert haben, laden Sie sie über einen Webbrowser (LYNX oder NETSCAPE). Geben Sie dabei nicht das Dokumentenverzeichnis an, sondern den Namen des Webservers, gefolgt vom Namen der Datei. Der Name des Servers kann

in der Konfigurationsdatei HTTPD.CONF unter dem Stichwort SERVER-NAME angegeben werden. Gibt es in der Konfigurationsdatei keinen entsprechenden Eintrag, heißt der Webserver wie Ihr Rechner, beispielsweise *http://localhost*.

- ✘ Das Dokumentenverzeichnis des Personal Web Server wird im Personal Web Manager (Doppelklick auf Symbol des Webservers in Taskleiste) angezeigt. Standardmäßig lautet es C:/INETPUB/WWWROOT. Nachdem Sie die HTML-Seite abgespeichert haben, laden Sie sie über einen Webbrowser (Internet Explorer oder Netscape Navigator). Geben Sie dabei nicht das Dokumentenverzeichnis an, sondern den Namen des Web-Servers, gefolgt vom Namen der Datei. Der Name des Web Servers kann ebenfalls über den Personal Web Manager abgefragt werden. Standardmäßig ist es der Name des lokalen Rechners, beispielsweise: *http://meinRechner* oder einfach *http://localhost*.
- ✘ Das Dokumentenverzeichnis des OmniHttpd lautet C:/HTTPD/HTDOCS, sofern Sie den Server in das Verzeichnis C:/HTTPD installiert haben.

Richten Sie den Server für die Ausführung von Perl-CGI-Skripten ein.

- ✘ Für Ihren Apache-Server wurde bereits ein Verzeichnis cgi-bin eingerichtet (je nach Version unter /HOME/HTTPD/CGI-BIN oder /USR/LOCAL/APACHE/CGI-BIN). Meist ist der Apache-Server so eingerichtet, dass Sie Ihre Skripten nur noch in dieses Verzeichnis kopieren müssen. Unter Umständen müssen Sie aber noch einen SkriptAlias einrichten, das Zugriffe auf *http://servername/cgi-bin* zu dem CGI-BIN-Verzeichnis umleitet (das ja kein Unterverzeichnis des Dokumentenverzeichnisses ist). Loggen Sie sich dann als Root ein, laden Sie die Konfigurationsdatei HTTPD.CONF (siehe oben) und erweitern Sie diese um folgenden Eintrag:

```
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/[1]
<Directory /home/httpd/cgi-bin>
   AllowOverride None
   Options None
</Directory>
```

- ✘ Starten Sie den Server danach neu: je nach Version /ETC/RC.D/INIT.D /HTTPD RESTART oder /USR/LOCAL/APACHE/BIN/APACHECTL RESTART.
- ✘ Wie Sie den Personal Web Server oder den IIS einrichten, hängt von der Server-Version und dem verwendeten Betriebssystem ab. In der HTML-

1 Je nach Apache-Version kann das Verzeichnis auch `/usr/local/apache/cgi-bin/` lauten. Sie können aber auch ein eigenes Verzeichnis für Ihre CGI-Programme festlegen.

Dokumentation zu ActivePerl (C:/PERL-VERZEICHNIS/HTML/INDEX.HTML) finden Sie hierzu ausführliche Informationen.

- ✘ Wenn Sie mit dem OmniHttpd arbeiten, klicken Sie in der Windows-Taskleiste mit der rechten Maustaste auf das Symbol des Servers und rufen Sie den Befehl PROPERTIES ein. In dem erscheinenden Dialogfenster wechseln Sie zum Register »ADVANCED«. Dort aktivieren Sie die Option ENABLE PERL CGI SUPPORT und tragen im Eingabefeld PERL CGI COMMAND LINE den Pfad zu Ihrem Perl-Interpreter ein, beispielsweise C:\PERL\BIN\PERL.EXE. Starten Sie den Server danach neu.

Testen Sie die Ausführung von Perl-CGI-Skripten auf dem Server. Setzen Sie dazu das folgende Perl-Skript auf und speichern Sie die Datei im CGI-BIN-Verzeichnis Ihres Servers. (Führen Sie das Skript zur Probe einmal aus, um sich zu vergewissern, dass es syntaktisch korrekt ist.)

Listing 16.2: testcgi.pl

```
#!/usr/bin/perl -w

print "Content-type: text/html\n\n";
print "<html>\n";
print "<head>\n";
print "<title>CGI-Testprogramm</title>\n";
print "</head>\n";
print "<body>\n";

foreach (keys %ENV)  {
  print "$_ = $ENV{$_}<br>\n";
  }

print "</body></html>\n";
```

Achten Sie darauf, dass die Datei vom Webserver ausgeführt werden kann.

- ✘ Wenn Sie unter Linux mit Apache arbeiten, ändern Sie die Zugriffsrechte für die Datei mit `chmod 755 testcgi.pl`.
- ✘ Wenn Sie mit dem PWS oder IIS arbeiten, müssen Sie bereits bei der Einrichtung des cgi-bin-Verzeichnisses darauf achten, dass diese das Ausführen der darin enthaltenen Dateien erlaubt.
- ✘ Wenn Sie mit dem OmniHttpd arbeiten, können Sie alternativ auch die Webseite `default.htm` aufrufen und über den Link »Test Perl, CGI and SSI« die Perl-Installation mit den Perl-Skripten testen, die mit dem Server ausgeliefert wurden.

Rufen Sie dann das Skript über Ihren Browser auf.

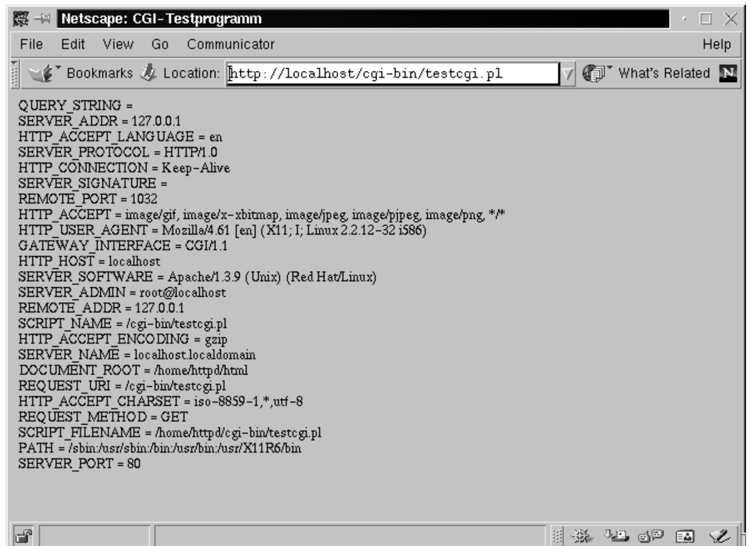


Abb. 16.1: Ausgabe des Testskripts im Browser

Ausführung von CGI-Programmen

CGI steht für Common Gateway Interface – eine Schnittstellenspezifikation, die festlegt, wie der Server CGI-Programme aufruft, Eingaben vom Browser an das CGI-Programm weiterleitet und die Ausgabe des CGI-Programms an den Browser zurückliefert.

Im einfachsten Fall läuft die CGI-Kommunikation wie folgt ab:

1. Der Browser schickt statt der Anforderung einer Webseite die URL eines CGI-Programms.

   ```
   http://webserver/cgi-bin/meinskript.pl
   ```

2. Der Webserver nimmt die Anforderung entgegen, sucht auf seiner Festplatte das CGI-Programm MEINSKRIPT.PL und ruft es auf.
3. Das CGI-Programm liefert eine HTML-Datei zurück. Dazu gibt das CGI-Programm einfach den HTML-Code der Datei Zeile für Zeile auf die Konsole aus.
4. Der Webserver schickt die Ausgabe des Programms an den Browser.

Für den Perl-Programmierer ist dabei vor allem Schritt 3 interessant. CGI-Programme sind nicht darauf beschränkt, HTML-Dateien zurückzuliefern. Sie können ebenso gut Grafiken oder andere Webinhalte zurückliefern. Wichtig ist, dass die Ausgabe auf die Konsole (STDOUT) erfolgt und dass das Programm die Daten im richtigen Format und mit einem korrekten

Header ausgibt. Schauen wir, inwieweit unser Beispielskript aus dem obigen Abschnitt diese Voraussetzungen erfüllt.

Listing 16.3: testcgi.pl

```
#!/usr/bin/perl -w

print "Content-type: text/html\n\n";
print "<html>\n";
print "<head>\n";
print "<title>CGI-Testprogramm</title>\n";
print "</head>\n";
print "<body>\n";

foreach (keys %ENV)  {
  print "$_ = $ENV{$_}<br>\n";
  }

print "</body></html>\n";
```

Analyse Als Erstes muss das Programm einen Header ausgeben. Der Header ist unerlässlich für die korrekte Verarbeitung der Daten durch den Browser und kann verschiedene Informationen enthalten. Er muss aber mindestens eine Angabe über die Art der zu empfangenden Daten enthalten, damit sich der Browser darauf einstellen kann, wie er die Daten zu verarbeiten hat.

In obigem Skript besteht der Header nur aus der Information über die Art der gelieferten Daten:

```
print "Content-type: text/html\n\n";
```

Wichtig ist dabei auch, dass der Header mit einem doppelten `\n` abgeschlossen wird, damit zwischen dem Header und den eigentlichen Daten eine Leerzeile steht (dies ist für den Browser das Zeichen, dass der Header hier endet).

Danach wird mit Hilfe mehrerer `print`-Anweisungen der HTML-Code einer Webseite ausgegeben.

```
<html>
<head>
<title>CGI-Testprogramm</title>
</head>
<body>
...
</body></html>
```

Im Körper der Webseite (zwischen den `<body>`-Tags) werden die Werte der verschiedenen Umgebungsvariablen ausgegeben. Den zugehörigen HTML-Code erzeugt die `foreach`-Schleife des Skripts, die die Einträge im Hash `%ENV` durchgeht (siehe Kapitel 9.4). Schauen Sie sich die Umgebungsvariablen ruhig einmal genauer an, etliche der Variablen sind für die CGI-Programmierung interessant und geben Auskunft über die Art der CGI-Kommunikation und über den beteiligten Browser und Server (siehe Abbildung 17.1).

Grafiken zurückliefern

Sie erinnern sich noch an das Skript DIAGRAMM.PL aus Kapitel 13? Wir benutzten das Skript dazu, auf der Grundlage tabellarischer Daten zum Primärenergieverbrauch eine Balkengrafik zu erstellen. Diese Grafik, die von dem Skript im PNG-Format abgespeichert wurde, konnte man dann mit Hilfe eines passenden Grafikprogramms betrachten. Es wurde auch schon erwähnt, dass man diese Grafiken in Webseiten einbinden kann, beispielsweise indem man als Quelle des Bildes die PNG-Datei angibt.

```
<IMG src="diagramm.png">
```

Dies hat jedoch den Nachteil, dass die Daten nicht automatisch aktualisiert werden. Die Daten waren ja in einer Datei ENERGIEN.TXT gespeichert.

Listing 16.4: energien.txt

```
Primärenergieverbrauch in Deutschland absolut (in GWh)
Quelle: Arbeitsgemeinschaft Energiebilanzen

 ;                   1990;    1992;    1994;    1996;    1997;
Steinkohle;        641000;  610000;  595000;  578000;  568000;
Braunkohle;        890000;  605000;  517000;  468000;  442000;
Kernenergie;       463000;  481000;  459000;  490000;  517000;
Mineralöl;        1456000; 1564000; 1579000; 1612000; 1592000;
Erdgas;            637000;  662000;  714000;  878000;  830000;
Regen. Energien;    58000;   55000;   71000;   78000;   79000;
```

Angenommen Sie hätten nun die Daten für 1998 vorliegen und tragen diese in die Datei ENERGIEN.TXT ein. Die Datei ENERGIEN.TXT wäre dann aktualisiert, nicht aber die Grafik VERBRAUCH.PNG, die in Ihre Webseite eingebunden ist. Sie müssen also noch explizit das Skript `diagramm.pl` aufrufen, die die Grafikdatei aktualisiert. Diesen Schritt – sowie ein wenig Festplattenspeicher auf dem Server – könnte man sich ersparen, wenn man von der Webseite aus direkt das Perl-Skript aufruft und dieses die Grafik nicht auf die Festplatte schreibt, sondern – ausgestattet mit einem passenden Header – direkt an den Browser schickt.

Um dies zu erreichen, bedarf es nur einiger weniger Änderungen am Skript.

Listing 16.5: cgidiagr.pl

```
#!/usr/bin/perl -w
use strict;

use Chart::StackedBars;

my $dateiname = "energien.txt";
my $diagr = Chart::StackedBars->new(600,400);

my @legende;
my %farben;
my $titel;

open(DATEI, "< $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";

# Titel
 chomp($titel = <DATEI>);
 $_ = <DATEI>; $_ = <DATEI>;

# x-Achsenticks
 $_ = <DATEI>;
 chomp;
 my @achse = split(/\s*;\s*/);
 shift(@achse);
 $diagr->add_dataset(@achse);

# Datensätze
 my $zaehler = 0;
 while (<DATEI>) {
  chomp;
  next if $_ eq '';
  my @felder = split(/\s*;\s*/);
  push(@legende, shift(@felder));
  $farben{"dataset$zaehler"} = [int(rand(255)),
                                int(rand(255)),
                                int(rand(255))];
  $diagr->add_dataset(@felder);
  $zaehler++;
  }

$diagr->set('title' => $titel);
$diagr->set('x_label' => "Jahre");
$diagr->set('y_label' => "Energieverbrauch (GWh)");
$diagr->set('colors' => \%farben);
$diagr->set('legend_labels' => \@legende);
print $diagr->cgi_png("Verbrauch.png");
```

Analyse

Eigentlich gibt es nur zwei Änderungen! Die erste Änderung ist, dass das Skript die Datei nicht über die Kommandozeile entgegennimmt, sondern mit `open` öffnet.

```
my $dateiname = "energien.txt";
...
open(DATEI, "< $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";
```

Da der Dateiname ohne Pfad angegeben wurde, geht das Skript davon aus, dass sich die Datei ENERGIEN.TXT in demselben Verzeichnis wie das Perl-Skript befindet.

In den nachfolgenden Zeilen wurden alle Lesezugriffe mit dem Eingabeoperator <> um die Angabe des Datei-Handles `DATEI` erweitert.

```
# Titel
 chomp($titel = <DATEI>);
 $_ = <DATEI>; $_ = <DATEI>;

# x-Achsenticks
 $_ = <DATEI>;
 ...
```

Die zweite Änderung betrifft die letzte Zeile des Skripts. Statt die Datei mit Hilfe der Funktion `png` als Datei auf der Festplatte zu speichern, wird die Grafik mit Hilfe der Funktion `cgi_png` aufgebaut und mit einem passenden Header ausgestattet. (Wenn Sie sich dafür interessieren, wie der Header – der dem Browser den Typ der Daten verrät – lautet, führen Sie das Skript doch einfach einmal von der Konsole aus (leiten Sie die Ausgabe gegebenenfalls in eine Textdatei um)).

Die `print`-Anweisung schickt die Daten dann direkt an die Standardausgabe, sprich an den Browser, wenn das Skript vom Webserver ausgeführt wird.

Linux-Versionen des Moduls `Chart` verwenden statt `cgi_png` die Funktion `cgi_gif`.

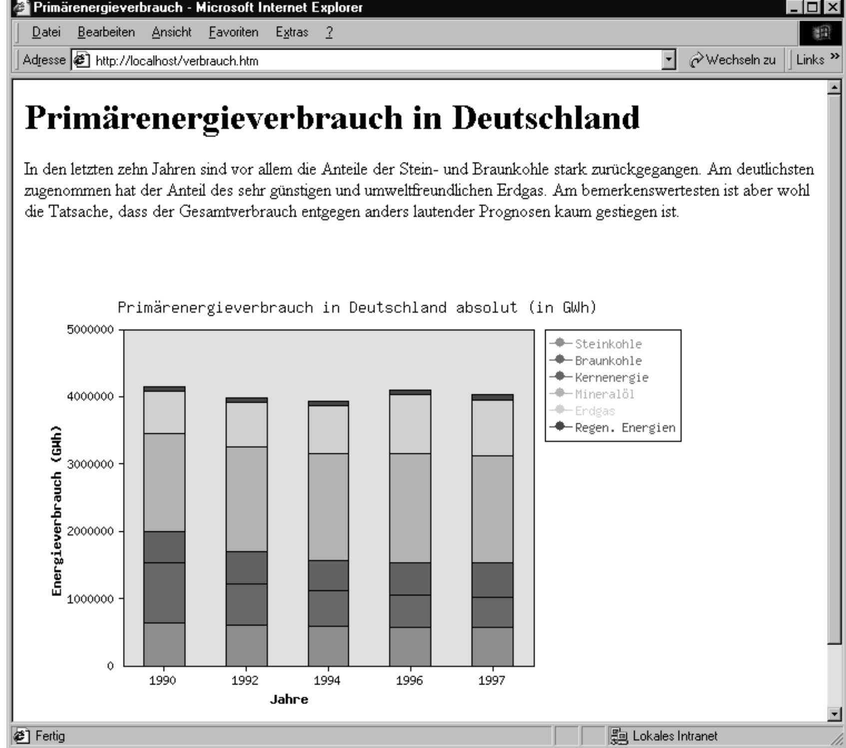


Abb. 16.2: Von Perl-Skript zurückgelieferte Grafik

CGI: Ein Gästebuch

CGI-Programme werden häufig dazu verwendet, Formulareingaben auf Seiten des Servers zu verarbeiten. Um Programme zur Verarbeitung von Formulareingaben schreiben zu können, muss man wissen, wie die Daten vom Browser an den Server und vom Server an das Programm weitergereicht werden.

GET und POST

Die CGI-Spezifikation sieht zwei Wege vor, wie ein CGI-Programm Daten von einem Browser entgegennehmen kann.

Tabelle 17.1: CGI-Methoden zur Datenübergabe

Methode	Beschreibung
GET	Bei dieser Methode hängt der Browser die Zeichenkette einfach an den URL des CGI-Programms an, das aufgerufen werden soll, und schickt den URL des Programms mit den angehängten Daten an den Server. Der Server trennt die Zeichenkette mit den Eingaben wieder von dem URL und speichert sie in der Umgebungsvariablen `QUERY_STRING` ab. Danach ruft der Server das CGI-Programm ab. Dieses muss so programmiert sein, dass es die Eingabe aus der Umgebungsvariablen `QUERY_STRING` einliest und auswertet. Dieses Verfahren ist an sich recht unkompliziert, hat aber den Nachteil, dass die Länge der übergebenen Zeichenkette beschnitten wird, wenn der Speicherplatz für die Umgebungsvariable nicht ausreicht (üblicherweise 1 Kbyte = 1024 Zeichen).

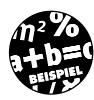


Tabelle 17.1: CGI-Methoden zur Datenübergabe (Fortsetzung)

Methode	Beschreibung
POST	Bei dieser Methode speichert der Server die Eingabe-Zeichenkette nicht in einer Umgebungsvariablen, sondern übergibt sie über die Standardeingabe an das CGI-Programm. Lediglich die Länge der codierten Eingabe-Zeichenkette wird in einer Umgebungsvariablen (`CONTENT_LENGTH`) abgelegt.

CGI-Datencodierung

Unabhängig davon, welchen Weg der Browser wählt, muss er die Daten für die Übertragung an den Server noch codieren (Leerzeichen und Sonderzeichen werden durch einfache Zeichenfolgen codiert, Eingaben aus Formularfeldern werden als Name=Wert-Paare codiert).

Angenommen der Besucher einer Webseite tippt in ein Eingabefeld seinen Namen ein:

Ihr Name : Dirk Louis

Wenn das Eingabefeld im HTML-Code die Name-ID `Feld1` hat und das Formular so konfiguriert ist, dass es seine Eingaben zur Auswertung per GET an ein CGI-Programm namens CGISKRIPT.PL schickt, so würde der fertige URL, der vom Browser an den Server gesendet wird, wie folgt aussehen:

```
http://server/cgi-bin/cgiskript.pl?Feld1=Dirk+Louis
```

Formulareingaben entgegennehmen

Wird ein CGI-Programm vom Server zur Verarbeitung von Formulardaten aufgerufen, muss es als Erstes feststellen, auf welchem Weg (GET oder POST) ihm die Daten übergeben wurden, dann muss es die Daten einlesen und dekodieren. Jetzt erst kann es darangehen, die Daten zu verarbeiten.

Grundsätzlich kann man all dies mit Hilfe der Umgebungsvariablen `REQUEST_METHOD`, `QUERY_STRING` und `CONTENT_LENGTH` selbst implementieren. Man kann es sich aber auch einfacher machen und das Perl-Modul `CGI` und dessen Funktionen verwenden. Dann braucht man nämlich nur mit Hilfe des Import-Tags `:standard` die Standardfunktionalität des Moduls einzubinden und kann danach die Eingaben problemlos mit Hilfe der Funktion `param` einlesen:

```
use CGI qw(:standard);
...
my $name = param('name');
```

Doch eines nach dem anderen! Beginnen wir damit, eine Webseite mit einem passenden Testformular aufzusetzen.

Abb. 17.1: Das Formular im Browser

Das Formular

Der HTML-Code des Formulars aus Abbildung 18.1 sieht wie folgt aus:

Listing 17.1: Formular.html

```
<html>
<head>
<title>Formular</title>
</head>

<body>
<h1>Formular</h1>

<form action="/cgi-bin/formular.pl">

  <p>Geben Sie bitte Ihren Namen ein:
    <input name="name" size="50"></p>

    <input type="submit" value="Abschicken">

</form>

</body></html>
```

Formulare werden im HTML-Code in die Tags `<form>` und `</form>` eingefasst. Zwischen diesen Tags stehen der Text und die Formularelemente. Das obige Formular enthält zwei Formularelemente: ein Eingabefeld `<input name="name" size="50">` und einen Schalter `<input type="submit" value="Abschicken">`.

Zwei Dinge am HTML-Code des Formulars sind aus Sicht des CGI-Programmierers besonders wichtig:

- ✘ Jedem Formularfeld kann mit Hilfe des Attributs `name=` ein Name zugewiesen werden. Beim Abschicken der Formulareingaben sendet der Browser die Eingaben zusammen mit dem Namen des dazugehörigen Formularelements (also beispielsweise `name="Peter"`). So kann das Perl-

Skript, das die Daten entgegennimmt, anhand der Namen feststellen, welche Eingaben zu welchen Formularelementen gehören.

- ✘ Über das `action`-Attribut des Formulars kann man festlegen, von welchem Skript die Formulareingaben verarbeitet werden sollen und wo dieses Programm zu finden ist (im Listing FORMULAR.HTML beispielsweise wird als Bearbeiter das Skript FORMULAR.PL aus dem `cgi-bin`-Verzeichnis des Webservers angegeben.

Damit steht fest: das verarbeitende Skript muss FORMULAR.PL heißen, im Verzeichnis CGI-BIN stehen und die Eingabe aus dem Feld mit dem Namen »`name`« verarbeiten.

Das CGI-Skript

Listing 17.2: formular.pl

```
1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:standard);
4:  use strict;
5:
6:  my $name = param('name');
7:
8:
9:  my $antwort = <<HERE_ANTWORT;
10: <html>
11: <head>
12: <title>Hallo</title>
13: </head>
14:
15: <body>
16: <h1>Hallo $name!</h1>
17:
18: <p>Es freut mich, Dich auf meiner Homepage willkommen
       hei&szlig;en zu d&uuml;rfen</p>
19:
20: </body>
21: </html>
22: HERE_ANTWORT
23:
24: print header, $antwort;
```

In Zeile 3 wird das CGI-Modul eingebunden. Dieses stellt uns die Methode `param` zur Verfügung, die das Entgegennehmen der Formulareingaben vereinfacht. Wir brauchen der Funktion nur den Namen des Formularfeldes zu übergeben, an dessen Inhalt wir interessiert sind, und die Funktion liefert uns diesen zurück (Zeile 6).

Der Rest des Skripts besteht darin, in HTML-Code eine Antwortseite aufsetzen. Für diese Aufgabe gibt es im CGI-Modul eine Reihe von vordefinier-

ten Funktionen (`start_html`, `start_form`, `end_form`, `table`, `h1`, `end_html` etc.). Ich persönlich ziehe es allerdings vor, den HTML-Code der zurückzuliefernden Seite in Form eines `HERE`-Dokuments aufzusetzen, da der typische Aufbau der HTML-Seite dann sichtbar bleibt und meines Erachtens leichter zu korrigieren ist. Zum HTML-Code selbst ist an sich nicht viel zu sagen. Beachten Sie aber, dass in Zeile 16 der Inhalt des `name`-Eingabefelds in die Begrüßung eingearbeitet wurde.

Zum Schluss wird der HTML-Code mit `print` an die Standardausgabe geschickt – nicht jedoch, ohne vorher die CGI-Funktion `header` aufzurufen, die den Header für den Browser zurückliefert.

Speichern Sie jetzt das Skript unter dem Namen FORMULAR.PL im CGI-BIN-Verzeichnis Ihres Webservers (wenn das CGI-Verzeichnis Ihres Servers anders lautet, müssen Sie die Pfadangabe in FORMULAR.HTML ändern). Wenn Sie unter Unix/Linux arbeiten, müssen Sie die Zugriffsrechte ändern, damit das Skript vom Webserver ausgeführt werden kann (`chmod 755 formular.pl`). Rufen Sie die Datei FORMULAR.HTML über Ihren Browser auf, geben Sie Ihren Namen in das Eingabefeld des Formulars ein und klicken Sie auf den Abschicken-Schalter. Wenn alles korrekt eingerichtet ist, sollte kurz darauf die Antwortseite im Browser erscheinen.

Abb. 17.2: Antwortseite im Browser

CGI-Skripten von der Konsole aus testen

Falls Sie keine Antwortseite sehen, könnte dies daran liegen, dass sich beim Aufsetzen des Skripts ein Fehler eingeschlichen hat. Dann empfiehlt es sich, die Korrektheit des Skripts erst einmal von der Konsole aus zu testen. Syntaxfehler können so schnell festgestellt werden. Was macht man aber, wenn der Fehler im logischen Aufbau des Skripts liegt oder die Ausgabe des Skripts fehlerhaft ist.

Um solche Fehler aufspüren zu können, ist es erforderlich, dass das Skript über die Konsole die gleichen Daten entgegennimmt, die es auch vom

Browser erhalten würde. Dazu müssen Sie die Eingaben in der Kommandozeile als `name=wert`-Paare übergeben und Leerzeichen im `wert`-Teil durch +-Zeichen ersetzen.

Abb. 17.3: CGI-Skripten von der Konsole aus testen

```
MS-DOS-Eingabeaufforderung
C:\httpd\Cgi-Bin>
C:\httpd\Cgi-Bin>
C:\httpd\Cgi-Bin>perl formular.pl name=Dirk+Louis
Content-Type: text/html

<html>
<head>
<title>Hallo</title>
</head>

<body>
<h1>Hallo Dirk Louis!</h1>

<p>Es freut mich, Dich auf meiner Homepage willkommen hei&szligen zu d&uumlrfen<
/p>

</body>
</html>

C:\httpd\Cgi-Bin>
```

Ein Gästebuch implementieren

Was ein Gästebuch ist, wissen Sie sicher. Zahllose private Homepages verfügen über ein Gästebuch, in das sich die Besucher der Website eintragen und dass sie sich natürlich auch ansehen können.

Im Folgenden wollen wir uns anschauen, wie man ein solches Gästebuch mit Hilfe von CGI und Perl realisieren kann. Die hier gezeigte Implementierung besteht aus drei Elementen:

- ✘ einem Formular, über das sich die Besucher der Website in das Gästebuch eintragen können
- ✘ dem Perl-Skript, das diese Formulareingaben verarbeitet und an das Gästebuch anhängt
- ✘ einer HTML-Datei, die das eigentliche Gästebuch darstellt

Beginnen wir mit dem Gästebuch.

1. Legen Sie unter dem Dokumentenverzeichnis Ihres Webservers ein neues Unterverzeichnis namens MEINWEB an.
2. Setzen Sie das Grundgerüst des Gästebuchs auf. Achten Sie vor allem auf die letzte Zeile, die genauso wie hier abgedruckt aussehen muss.

```
<html>
<head>
<title>G&auml;stebuch</title>
</head>
```

```
<body>
<h1> Mein G&auml;stebuch </h1>


</body></html>
```

3. Speichern Sie die Datei des Gästebuchs als GAESTEBUCH.HTML im Verzeichnis MEINWEB (und achten Sie besonders unter Unix/Linux darauf, dass das Perl-Skript Schreibrechte hat).

4. Setzen Sie den HTML-Code für das Formular zum Eintragen in das Gästebuch auf.

   Das folgende Formular verfügt über drei Eingabefelder und ein mehrzeiliges Textfeld mit den Namen: `name`, `email`, `website` und `kommentar`. Als Bearbeiter für die Formulardaten wurde das Skript `/cgi-bin/gaestebuch.pl` angegeben (wenn das CGI-Verzeichnis Ihres Servers anders lautet, müssen Sie die Pfadangabe entsprechend anpassen).

Listing 17.3: Eintragen.html

```
<html>
<head>
<title>G&auml;stebucheintrag</title>
</head>

<body>
<h1>Tragen Sie sich bitte in mein G&auml;stebuch ein:</h1>
<hr>
<form action="/cgi-bin/gaestebuch.pl">
  <p>Geben Sie bitte Ihren Namen an      :
                        <input name="name" size="50"></p>
  <p>Möchten Sie Ihre EMail hinterlassen :
                      <input name="email" size="50"> </p>
  <p>Haben Sie auch eine eigene Website  :
                     <input name="website" size="50"></p>
  <p>Ihr Kommentar :  </p>
  <p>
  <textarea name="kommentar" rows="9" cols="59"
  </textarea></p>
 <input type="submit" value="In G&auml;stebuch eintragen">
</form>
<hr>

</body></html>
```

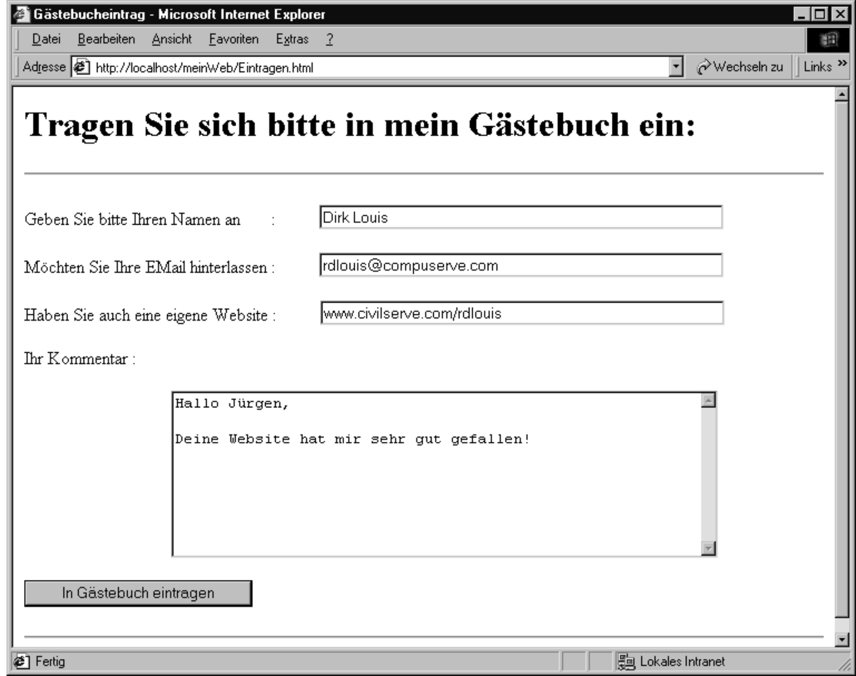


Abb. 17.4: Das Formular zum Eintragen in das Gästebuch

5. Speichern Sie die Datei des Eingabeformulars unter einem beliebigen Namen im Verzeichnis MEINWEB.

6. Setzen Sie das Perl-Skript auf.

Listing 17.4: gaestebuch.pl

```
1:  #!/usr/bin/perl -w
2:
3:  use CGI qw(:standard);
4:  use strict;
5:
6:
7:  # ********  Gästebuch öffnen ********
8:
9:  my $gaestebuch = "../htDocs/MeinWeb/gaestebuch.html";
10:
11: open(BUCH, "+< $gaestebuch")
12:    or
13:    die "\nDatei $gaestebuch konnte nicht geoeffnet
          werden\n";
14:
15:
16: # ********  Eingaben aus Formular lesen ********
17:
```

```
18: my ($name, $email, $website, $kommentar) =
19:        (param('name'), param('email'), param('website'),
20:         param('kommentar'));
21:
22:
23: # ********  Eintrag an Gästebuch anhängen ********
24:
25: my $neuereintrag = <<HERE_EINTRAG;
26: <p>$name schrieb: </p>
27: <p><i>$kommentar</i></p>
28: <p><font size="1">$name ist &uuml;ber $email erreichbar
29:  und unterh&auml;lt eine eigene Website ($website).
30: </font size="1"></p>
31: <hr>
32: HERE_EINTRAG
33:
34: seek(BUCH,-14,2);
35: print BUCH "$neuereintrag\n</body></html>";
36: close(BUCH);
37:
38:
39: # ********  Dankseite zurücksenden ********
40:
41: my $danke = <<HERE_DANKE;
42: <html>
43: <head>
44: <title>Danke!</title>
45: </head>
46:
47: <body>
48: <h1>Danke!</h1>
49:
50: <p>Danke, dass Sie sich in mein G&auml;stebuch
51: eingetragen haben!</p>
52: <p>Haben Sie denn auch schon einmal in das
53: <a href="/MeinWeb/gaestebuch.html">G&auml;stebuch.</a>
54: hineingeschaut?</p>
55:
56: <p> </p>
57: <p><i>So long,   Dirk</i></p>
58:
59: </body></html>
60: HERE_DANKE
61:
62: print header, $danke;
```

Der Pfad zur Gästebuchdatei muss für Ihren Server angepasst werden!

Analyse Als Erstes öffnet das Skript die Gästebuchdatei (Zeilen 7 bis 14). Gegen den Code an sich ist nichts zu sagen, er könnte jedoch noch ein wenig verbessert werden. Bedenken Sie, dass das Skript unter Umständen von zwei verschiedenen Besuchern Ihrer Website mehr oder weniger gleichzeitig aufgerufen werden könnte. Wenn die beiden Instanzen dann in die Datei schreiben, kann es passieren, dass Daten verloren gehen oder gar die ganze Datei korrumpiert wird. Sie können dies verhindern, indem Sie die Datei nach dem Öffnen mit Hilfe der Funktion `flock` aus dem `Fcntl`-Modul sperren:

```
use Fcntl qw(:flock);
...
flock(BUCH, LOCK_EX);
```

In den Zeilen 16–20 werden die Inhalte der Formularfelder ausgelesen. Hierzu ist anzumerken, dass das Skript nicht prüft, ob für alle Felder sinnvolle Eingaben vorliegen. Ich habe im Beispiel darauf verzichtet, um den Code nicht unnötig zu komplizieren. Wenn Sie CGI-Skripten schreiben, die Eingaben verarbeiten, sollten Sie die Eingaben aber unbedingt prüfen (mit `if(!defined param('name'))`)

Achten Sie auch darauf, was Sie mit den Formulareingaben machen. Böswillige Hacker könnten statt des erwarteten Namens oder Kommentars einen Betriebssystembefehl, eine Perl-Befehlsfolge oder ähnlichen Code eingeben, der je nachdem wie die Eingabe weiterverarbeitet wird, ein riesiges Loch in das Sicherheitsnetz Ihres Servers reißen kann.

Unter Verwendung der Formulareingaben wird – in Form eines HERE-Dokuments – ein neuer Eintrag für das Gästebuch aufgesetzt (Zeilen 25–32). Dieser wird in Zeile 35 an das Gästebuch angehängt. Allerdings wollen wir nicht, dass der neue Eintrag an die abschließende `</body></html>`-Zeile angehängt wird, sondern er soll diese ersetzen. Wie kann man dies bewerkstelligen? Jeder Datei-Handle verfügt über einen Positionsmarker, der angibt, an welche Stelle der Datei als Nächstes geschrieben bzw. von wo gelesen wird. Diese Positionsmarke kann man mit Hilfe der Perl-Funktion `seek` verschieben. Dazu übergibt man `seek` den Datei-Handle, die Anzahl der Zeichen, um die verschoben werden soll, und die Position, ab der verschoben werden soll (siehe PERLFUNC-Dokumentation). Der Aufruf

```
seek(BUCH,-14,2);
```

verschiebt die Positionsmarke vom Dateiende an (Argument 2) um 14 Zeichen nach vorne (Argument -14).

Danach werden der neue Eintrag und eine neue abschließende Zeile in die Datei geschrieben (Zeile 35).

Zu guter Letzt wird eine Antwortseite zurückgeliefert, die dem Besucher der Webseite anzeigt, dass seine Eingaben verarbeitet wurden, und die ihm einen Link zum Anzeigen des Gästebuchs anbietet (Zeilen 39–62).

7. Speichern Sie das Skript unter dem Namen GAESTEBUCH.PL im CGI-BIN-Verzeichnis Ihres Webservers. Wenn Sie unter Unix/Linux arbeiten, müssen Sie die Zugriffsrechte ändern, damit das Skript vom Webserver ausgeführt werden kann (`chmod 755 formular.pl`).
8. Rufen Sie jetzt das HTML-Dokument mit dem Formular über Ihren Browser auf, füllen Sie die Felder des Formulars aus und klicken Sie auf den Schalter. Wenn alles korrekt eingerichtet ist, sollte kurz darauf die Antwortseite im Browser erscheinen.

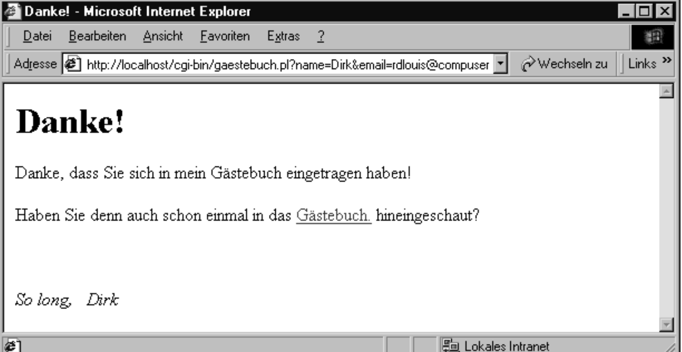


Abb. 17.5: Antwortseite des Perl-Skripts gaestebuch.pl

9. Über den Link können Sie das Gästebuch einsehen.

Wenn Sie das Gästebuch beim Testen des Skripts wiederholt aufrufen, kann es passieren, dass die zuletzt eingetragenen Formularangaben nicht angezeigt werden. Dies muss nicht am Skript liegen, sondern kann damit zu tun haben, dass Ihr Browser den Inhalt der Gästebuchdatei aus seinem Cache rekonstruiert. Aktualisieren Sie dann die Seite oder löschen Sie den Cache.

Abb. 17.6: Das Gästebuch

Teil IV

Anhang

Anhang A: Synopsis
Anhang B: Module
Anhang C: Online-Hilfe
Anhang D: Der Debugger
Anhang E: Lösungen zu den Übungen
Anhang F: ASCII-Tabelle
Anhang G: Webadressen

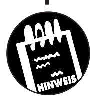
Anhang A

Synopsis

Konstanten

Stringkonstanten

```
'Dies ist ein Text'          # String
"Wert von var1 ist $var1"    # String mit Variablenexpansion

q%Dies ist ein Text%;        # Änderung des Quoting-Zeichens
qq%Wert von var1 ist $var1%

$text = <<HERE_DOK;          # HERE-Dokument
...
HERE_DOK
```



Zahlenkonstanten

```
333                          # Ganzzahl
333.33                       # Gleitkommazahl
333_333_333.333              # Trennung der Tausender
1.6021892e-19                # Exponentialschreibweise
0b0011                       # Binärdarstellung
0123                         # Oktaldarstellung
0x12E                        # Hexadezimaldarstellung
```

Listenkonstanten

```
()                             # Leere Liste
(5, 133, -12.5)                # Zahlen
("Na, wie geht\'s", "Geh!")    # Strings
("Hallo", -0.344, "bzz", 1)    # gemischt

(-2..2, 3..6)                  # Bereiche
('aa'..'ad')

("Vorname" => "Rip",           # Hash-Syntax
 "Nachname" => "Winkle");
qw(125 127 129 135 155)        # Änderung des
qw(Werner Otto Gaston)         # Quoting-Zeichnes
```

Symbolische Konstanten

```
use constant PI => 3.1415;     # konstante Variablen
```

Variablen

```
$var = 123;                    # Skalar
$str = "Hallo";                # Skalar
@array = ();                   # Array
%hash = ();                    # Hash (assoziatives Array)

$ref = \$var;                  # Referenz

my $var;                       # Deklaration einer lokalen
                               # Variablen
local $var                     # Deklaration einer lokalen
                               # Variablen
use vars                       # Deklaration einer
                               # Package-Variablen
```

Zahlen

Konstanten

```
333                            # Ganzzahl
333.33                         # Gleitkommazahl
333_333_333.333                # Trennung der Tausender
1.6021892e-19                  # Exponentialschreibweise
0b0011                         # Binärdarstellung
0123                           # Oktaldarstellung
0x12E                          # Hexadezimaldarstellung
```

Variablen

```
$var = 123;                 # Skalar
@array = (5, 133, -12.5)    # Array
%hash = ("wert1" => 1,      # Hash
         "wert2" => 2);
```

Operatoren

```
+op1                        # positives Vorzeichen
-op1                        # negatives Vorzeichen
op1 + op2                   # Addition
op1 - op2                   # Subtraktion
op1 * op2                   # Multiplikation
op1 / op2                   # Division
op1 ** op2                  # Exponent
op1 % op2                   # Modulo
==                          # Gleichheit
!=                          # Ungleichheit
<                           # kleiner
>                           # größer
<=                          # kleiner gleich
>=                          # größer gleich
<=>                         # -1 0 1
<<                          # Bitweiser Linksshift
>>                          # Bitweiser Rechtsshift
|                           # Bitweises ODER
&                           # Bitweises UND
^                           # Bitweises XOR
~                           # Bitweises Komplement
```

Funktionen

```
abs                         # Absoluter Betrag
atan2                       # Arcustangens von x/y
cos                         # Kosinus (x in Bogenmaß)
exp                         # Exponentialfunktion (ex)
hex                         # Umwandlung in Hexadezimalzahl
int                         # Umwandlung in Ganzzahl
log                         # Natürlich. Logarithmus (ln x)
oct                         # Umwandlung in Oktalzahl
sin                         # Sinus (x in Bogenmaß)
sqrt                        # Wurzel (x positiv)

int                         # Gleitkommazahl -> Ganzzahl
ceil                        # Aufrunden
floor                       # Abrunden
sprintf                     # in String konvertieren
```

```
hex                            # in Hexadezimalcode
oct                            # in Oktalcode

rand                           # Zufallszahl zurückliefern
srand                          # Zufallsgenerator einstellen
```

Module

```
use Math::Trig;
use Math::Complex;
use Math::BigInt;
use Math::BigFloat;
```

Strings

Stringkonstanten

```
'Dies ist ein Text'            # String
"Wert von var1 ist $var1"      # String mit Variablenexpansion

q%Dies ist ein Text%;          # Änderung des Quoting-Zeichens
qq%Wert von var1 ist $var1%

$text = <<HERE_DOK;            # HERE-Dokument
...
HERE_DOK
```

Variablen

```
$var = "Hallo";                       # Skalar
@array = ("Hallo", "eins und");       # Array
%hash = ("wert1" => "Hallo",          # Hash
         "wert2" => "eins und ");
```

Operatoren

```
str1 . str2                    # Konkatenation
str x Zahl;                    # Wiederholung
$str = 'aaa'; ++$str;          # Inkrement
eq                             # Gleichheit
ne                             # Ungleichheit
lt                             # kleiner
gt                             # größer
le                             # kleiner gleich
ge                             # größer gleich
cmp                            # -1 0 1
```

Funktionen

```
print                         # Ausgabe
printf                        # formatierte Ausgabe
sprintf                       # format. Ausgabe in String
length                        # Länge
lc                            # Zeichen klein
uc                            # Zeichen groß
lcfirst                       # erstes Zeichen klein
ucfirst                       # erstes Zeichen groß
reverse                       # Zeichenfolge umkehren
chr                           # ASCII-Code -> Zeichen
ord                           # Zeichen -> ASCII-Code
index                         # Suche von vorne
rindex                        # Suche von hinten
substr                        # Teilstrings
chop                          # letztes Zeichen entfernen
chomp                         # letztes Zeichen entfernen,
                              # wenn gleich $/
pack                          # Daten in String packen
unpack                        # String -> Daten
split                         # String aufspalten
```

Escape-Sequenzen

```
\a                            # Signalton
\b                            # Backspace
\e                            # Escape
\f                            # Neue Seite
\n                            # Neue Zeile
\r                            # Wagenrücklauf
\t                            # Tabulator
\\                            # \
\"                            # "
\'                            # '
\nnn                      # Zeichen, dessen ASCII-Code der
                          # oktalen Zahl nnn entspricht
\xnn                      # Zeichen, dessen ASCII-Code der
                          # hexadezimalen Zahl nn entspricht
\cX                       # Control-Darstellung
\u                        # nächstes Zeichen groß
\U                        # alle nachfolgenden Zeichen groß
\l                        # nächstes Zeichen klein
\L                        # alle nachfolgenden Zeichen klein
```

Module

```
use locale;        # 8-Bit-ASCII mit Umlauten
no locale        # 7-Bit-ASCII
```

Operatoren

Operator (nach Rangordnung)	Assoz.	Bedeutung
++ --	–	Inkrement und Dekrement
**	R-L	Exponent
! \ + -	R-L	Logisches NOT, Referenz, positives und negatives Vorzeichen
=~ !+	L-R	Pattern Matching
* / % x	L-R	Multiplikation, Division, Modulo, String-Wiederholung
+ - .	L-R	Addition, Subtraktion, Konkatenation
<< >>	–	Bitweise Shift-Operatoren
< > <= >= lt gt le ge	–	Vergleiche
== != <=> eq ne cmp	–	Weitere Vergleiche
&	L-R	Bitweises UND
| ^	L-R	Bitweises ODER und EXKLUSIVES ODER
&&	L-R	Logisches UND
||	L-R	Logisches ODER
..	–	Bereichsoperator
= += -= *= /= %=	R-L	Zuweisung
not	R-L	Logisches NOT
and	L-R	Logisches UND
or xor	L-R	Logisches ODER und EXKLUSIVES ODER

Kontrollstrukturen

```
if (Bedingung) {...}
if (Bedingung) {...} else {...}
if(Bedingung) {...} elseif (Bedingung) {...} else {...}

while (Bedingung) {...}
until(Bedingung) {...}

do {...} while (Bedingung);
do {...} until (Bedingung);

for (Init;Beding;Inkrement) {...}
foreach $key (@liste){...}

SWITCH: {
  $eingabe == 1   && do { ... last SWITCH; };
  $eingabe == 2   && do { ... last SWITCH; };
  $eingabe == 3   && do { ... last SWITCH; };
  $eingabe == 4   && do { ... last SWITCH; };
  ...  }
```

Kurzschreibweisen

```
$var = Bedingung ? Ausdruck1 : Ausdruck2;
Anweisung if Bedingung;
Anweisung unless Bedingung;
do {...} if Bedingung;
do {...} unless Bedingung;
```

Vergleichsoperatoren

Vergleich	Zahlen	Strings
Gleichheit	==	eq
Ungleichheit	!=	ne
kleiner	<	lt
größer		gt
kleiner gleich	<=	le
größer gleich		ge
-1 0 1	<=>	cmp

Logische Operatoren

! A1	**Logische Verneinung**	**wahr, wenn A1 falsch**
A1 **&&** A2	Logisches UND	wahr, wenn A1 und A2 beide wahr
A1 || A2	Logisches ODER	wahr, wenn A1 oder A2 oder beide wahr
A1 ^ A2	Exklusives ODER	wahr, wenn entweder A1 oder A2 wahr

Abbruchbefehle

```
redo                # Aktuelle Schleifeniteration wiederholen
next                # Aktuelle Schleifeniteration abbrechen
last                # Schleife beenden
```

Benannte Schleifen

```
AUSSEN: foreach $i (@liste1) {
  ...
  INNEN: foreach $j (@liste2) {
    if (Bedingung) {
       next AUSSEN;
       }
    }
  }
```

Listen, Array, Hashes

Listendefinitionen

```
()                            # Leere Liste
(5, 133, -12.5)               # Zahlen
("Na, wie geht\'s", "Geh!")   # Strings
("Hallo", -0.344, "bzz", 1)   # gemischt

(-2..2, 3..6)                 # Bereiche
('aa'..'ad')

("Vorname" => "Rip",          # Hash-Syntax
 "Nachname" => "Winkle");
qw(125 127 129 135 155)       # Änderung des
qw(Werner Otto Gaston)        # Quoting-Zeichnes
```

Arrays

```
@array = ();                    # Array
$array[3][4] = 12               # Zugriff auf Array-Element
$anzahl = @array                # Anzahl der Elemente in Array
$#array                         # größter Index
@teilarray = @array[2-4];       # Teilarray zurückliefern
foreach $key (@array) {...}     # Array durchlaufen
```

Array-Funktionen

```
sort (@array);                  # Array sortieren
sort {$a cmp $b} @array;
@neu = map {$_ * 2 } @array;    # Array-Elemente durchlaufen
                                # und bearbeiten
grep grep {$_ % 2} @array;      # Array-Elemente suchen
print "@array\n";               # Array ausgeben
print join(', ', @array);
push(@array, 4);                # Element anhängen
unshift(@array, 4..5);          # Element vorne einfügen
pop @array;                     # Letztes Element löschen
shift @array;                   # Erstes Element löschen
@ersetzt = splice(@array, 3, 2);   # Elemente ersetzen
@array = split(" ", $str);      # Strings aufteilen
```

Hashes

```
%hash = ();                           # Hash-Variable
%person = ("Vorname", "Rip",          # Hash-Definitonen
           "Nachname", "Winkle");
%person = ("Vorname" => "Rip",
           "Nachname" => "Winkle");

$person{Vorname} = "neuerName";       # Schlüssel einrichten
                                      # Schlüssel Wert zuweisen
delete $person{Vorname};              # Schlüssel/Wert-Paar
                                      # löschen
undef $person{Vorname};               # Wert des Schlüssels
                                      # löschen
if (exists($person{Nachname}))        # existiert Schlüssel?
if (defined($person{Nachname}))       # Wert definiert

foreach $schluessel (keys %person) {...}      # Schlüssel
                                              # durchlaufen
foreach $schluessel (sort keys %person) {...} # Schlüssel
                                              # sortiert
                                              # durchlaufen
```

```
foreach $werte (values %person) {...}          # Werte
                                               # durchlaufen
foreach $werte (sort values %person) {...}     # Werte
                                               # sortiert
                                               # durchlaufen
```

Funktionen

Aufrufe

```
meineFunktion();                # zu empfehlen
&meineFunktion;
&meineFunktion();
meineFunktion;
```

Funktionsdefinition

```
sub funk { }                    # Funktionsdefinition
```

Parameter

```
sub demoFunk {                  # Parameter abfragen
  print "Param 1: $_[0]\n";
  print "Param 2: $_[1]\n";
  }
sub demoFunk {                  # Parameter abfragen
  my $param1 = shift;
  my $param2 = shift;


demoFunk($var1, $var2);         # Argumente an Funktion
                                # übergeben
```

Lokale Variablen

```
sub demoFunk {
  my $dummy = 0;                # lokale Variable
  local $tmp = 0;               # lokale Variable
   ...
```

Werte zurückliefern

```
sub demoFunk {
  ...
  return $ergebnis;             # Skalar zurückliefern
  }
```

```
sub demoFunk {
  ...
  return @werte;              # Array zurückliefern
  }

sub demoFunk {
  ...
  $ergebnis = 1;              # Letzten Wert zurückliefern
  }
```

Kontextsensitive Funktionen

```
sub meineFunktion {
  if (wantarray) {
    # wantarray hat »wahr« ergeben -> Listenkontext
    return @array;
    }
  else {
    # wantarray hat »falsch« ergeben -> skalarer Kontext
    return $skalar;
    }
  }
```

Referenzen

```
$ref = \$var;                 # Referenz auf Skalar
$ref = \@array;               # Referenz auf Array
$ref = \%hash;                # Referenz auf Hash
$ref = \&funktion;            # Referenz auf Funktion

$ref = [1, 2, 3, 8, 9];       # Referenz auf anonymes Array
$ref = {"Name" => "Rip",      # Referenz auf anonymes Hash
        "Vorname" => "van Winkle"};
```

Dereferenzierung

```
$$ref_s                       # Skalar
${$ref_s}                     # Skalar

@$ref_a                       # Array
@{$ref_a}                     # Array
${$ref_a}[0]                  # Array-Element
$ref_a->[0]                   # Array-Element

%$ref_h                       # Hash
%{$ref_h}                     # Hash
```

```
${$ref_h}{'key'}              # Hash-Element
$ref_h->{'key'}               # Hash-Element
```

Funktionen

```
ref($var)                     # prüft, ob $var eine
                              # Referenz ist
```

Array von Arrays

```
@array   = (\@array1, \@array2);
@array   = ([1, 2], [33, 44, 55]);
foreach $array_ref (@array) {
  foreach $elem (@{$array_ref}) {
     print "$elem ";
     }
  print "\n";
  }
```

Array von Hashes

```
@array   = (\%hash1, \%hash2);
@array   = ({Vorname => "Rip", Nachname => "van Winkle"},
            {Vorname => "Peter", Nachname => "Kreuter"});
foreach $hash_ref (@array) {
  foreach $key (keys %{$hash_ref}) {
     print "$key: ", "$hash_ref->{$key} / " ;
     }
  print "\n";
  }
```

Hash von Hashes

```
%hash1  = ("Vorname" => "Rip",
           "Nachname" => "van Winkle");
%hash2  = ("Vorname" => "Peter", "Nachname" => "Kreuter");
%bighash   = ("h1" => \%hash1, "h2" => \%hash2);


foreach $bigkey (keys %bighash)
  {
  $subhash_ref = $bighash{$bigkey};

  foreach $key (keys %{$subhash_ref}) {
     print "$key: ", "$subhash_ref->{$key} / " ;
     }
  print "\n";
  }
```

Hash mit Liste

```
%hash  = (Vorname => "Rip",
          Nachname => "van Winkle",
          Soehne => [Peter, Ralf, Joseph]);


foreach $key (keys %hash) {
  print "$key: ";
  if( ref( $hash{$key} ))
    {
    foreach $elem (@{$hash{$key}}) {
      print "$elem ";
      }
    }
    else
    {
    print "$hash{$key} ";
    }
  print "\n";
  }
```

Die wichtigsten Standardvariablen

```
$_                        # Allround-Standardvariable
@_                        # Parameterübergabe
$a, $b                    # für sort
$/                        # Zeilentrennzeichen (<>, print)
$\                        # Abschluss einer Ausgabe (print)
$, und $"                 # Elementtrenner für Arrays (print)
$|                        # Pufferung (print
$.                        # Zeilennummer (write, print)
$#                        # Ausgabeformat für Zahlen (write)
$^L                       # Seitentrenner (write)
$0                        # Name des laufenden Programms
$^O                       # Name des Betriebssystems
@INC                      # Suchpfad für Perl-Module
$$                        # ID des aktuellen Prozesses
$ARGV                     # Kommandozeilenargumente
%ENV                      # Umgebungsvariablen
$1                        # Variable, in der der
                          # Pattern-Matching-Operator
                          # gefundene Teilmuster speichert
```

Ein- und Ausgabe

Eingabeoperator <>

```
$skalar = <>;                  # liest einzelne Zeile
@array = <>;                   # liest Zeilen bis Dateiende
$skalar = <DATEI-HANDLE>       # liest aus Datei
<STDIN>;                       # Programm anhalten
chomp($eingabe = <STDIN>);     # Wert von Tastatur einlesen
while ($zeile = <>) {          # Datei aus Kommandozeile
einlesen
  chomp ($zeile);
  print $zeile;
  <STDIN>;
  }
open(DATEI, ">> $dateiname")  # Datei aus Datei-Handle
     or die "\nFehler\n";      # einlesen
while(my $zeile = <DATEI>)
    {
    chomp($zeile);
    ...
```

Standardvariablen für print

```
$/                         # Zeilentrennzeichen
$\                         # Abschluss einer Ausgabe
$, und $"                  # Elementtrenner für Arrays
$|                         # Pufferung
$.                         # Zeilennummer
```

Formattypen für printf

```
%c                         # einzelnes Zeichen
%s                         # String
%d                         # Ganzzahl
%f                         # Gleitkommzahl
%e                         # Gleitkommazahl in
                           # Exponentialschreibweise
%o                         # oktale Ganzzahl
%x                         # hexadezimale Ganzzahl
%%                         # Ausgabe des %-Zeichens
```

Formatflags für printf

```
+                         # Zahlen immer mit Vorzeichen
Leerzeichen               # pos. Zahlen mit Leerzeichen statt
                          # +-Symbol
-                         # Linksbündige Ausrichtung
b                         # Feldbreite der Ausgabe
#                         # Präfix 0 bzw. 0x für Oktal- bzw.
                          # Hexadezimalzahlen
```

Reportgenerator write

```
format STDOUT_TOP =

Überschrift                                       Seite @>>
                                                     $%

------------------------------------------------------------
Spalte1          Spalte2                          Spalte3
------------------------------------------------------------
.

format STDOUT =
@<<<<<<<<<<<<<<<<  @<<<<<<<<<<<<<<<<<<<<<<<<<<   @<<<<<<<<<<<
$feld1, $feld2, $feld3
.
sub ausgeben {
  $- = 0;                 # Restzeilen für diese Seite
  $% = 0;                 # erste Seitenzahl
  $= = 25;                # Anzahl Zeilen pro Seite

  foreach $elem (@liste) {
    $feld1 =
    $feld2 =
    $feld3 =
    write STDOUT;;
    }
}
```

Dateien

```
open(FH, "datei");        # Öffnet die Datei zum Lesen
open(FH, "< datei");
open(FH, "> datei");      # Öffnet die Datei zum Schreiben.
open(FH, ">> datei");     # Öffnet die Datei zum Anhängen.
open(FH, "+< datei");     # Öffnet die Datei zum Lesen und
                          # Schreiben.
```

```
open(DATEI, ">> $dateiname")       # Datei öffnen
    or
    die "\nDatei $dateiname konnte nicht geoeffnet werden\n";

print DATEI $var1, $var2;          # in Datei schreiben

while(my $zeile = <DATEI>)         # aus Datei lesen
    {
    chomp($zeile);
    @daten = (split("\t", $zeile));

close(DATEI);                      # Datei schließen

binmode(DATEI);                    # Binärmodus einschalten
```

Kommandozeilenargumente

```
foreach my $arg (@ARGV) {
  print "$arg\n";
  }
```

Umgebungsvariablen

```
foreach my $key (keys %ENV) {
  print $key, ": ", $ENV{$key}, "\n";
  }
```

Die wichtigsten Symbole für reguläre Ausdrücke

Operatoraufrufe

```
$text =~ m/Suchmuster/
$text =~ m%Suchmuster%
$text =~ m/Suchmuster/g
$text =~ m/Suchmuster/s
$text =~ m/Suchmuster/i
$text =~ m/Suchmuster/m
$text =~ s/Suchen/Ersetzen/
$text =~ s/Suchen/Ersetzen/g    # globale Suche
$text =~ s/Suchen/Ersetzen/s    # Suche über Zeilenende
                                # hinweg
$text =~ s/Suchen/Ersetzen/i    # keine Unterscheidung
                                # zwischen Groß- und
                                # Kleinschreibung
$text =~ s/Suchen/Ersetzen/m    # Suche erkennt \n-Zeichen im
                                # String als Zeilenende
```

Metazeichen in regulären Ausdrücken

```
^ . ? { } [ ] ( ) / + * $ |
```

Zeichengruppen

```
(al|el|il)  # alternative Zeichenfolgen
(al|el|)    # alternative Zeichenfolgen oder keine Zeichen
[abc%?]     # alternative Zeichen: a, b, c, % oder ?
[a-f]       # alternative Zeichen: a, b, c, d, e oder f
[a-zA-Z]    # alternative Zeichen: jeder Buchstabe
[^a-zA-Z]   # alternative Zeichen: jedes Zeichen, das kein
            # Buchstabe ist
\d          # entspricht [0-9] (eine Ziffer)
\D          # entspricht [^0-9] (keine Ziffer)
\w          # entspricht [0-9a-zA-Z_] (ein Wortzeichen)
\W          # entspricht [^0-9a-zA-Z_] (kein Wortzeichen)
\s          # entspricht [ \t\n\r\f] (ein Whitespace-Zeichen)
\S          # entspricht [^ \t\n\r\f]
            # (kein Whitespace-Zeichen)
.           # ein beliebiges Zeichen außer \n (wenn die
            # Option s angehängt ist, schließt . auch das
            # \n-Zeichen mit ein).
```

Quantifizierer

```
{n,m}       # mindestens n, höchstens m Wiederholungen
{n,}        # mindestens n Wiederholungen
{n}         # genau n Wiederholungen
*           # keinmal oder mehrmals (entspricht {0,}
+           # einmal oder mehrmals (entspricht {1,}
?           # einmal oder keinmal (entspricht {0,1}
Durch Anhängen von ? erhält man die nichtgierigen Versionen
der Quantifizierer
```

Anker

```
\b          # Wortgrenze (Übergang von \w zu \W)
\B          # Nichtwortgrenze
^           # Anfang des Strings (wenn die Option /m gesetzt
            # ist, auch jede Position direkt hinter einem
            # Neue-Zeile-Zeichen (\n))
$           # Ende des Strings (wenn die Option /m gesetzt
            # ist, auch jede Position vor einem
            # Neue-Zeile-Zeichen (\n))
\A          # Anfang des Strings
\Z          # Ende des Strings
```

Rückverweise

```
()          # fasst Teilmuster ein, das zurückgeliefert wird
/1          # Rückverweis innerhalb des Suchmusters
$1          # Variable, in der zurückgeliefertes Teilmuster
            # gespeichert wird
```

Objektorientierte Programmierung

Klassendeklariation

```
#*** Definition der Klasse in eigenem Package ********
package MeineKlasse;
```

Konstruktor

```
sub new {
  my $classname = shift;
  my $self = {};
  $self->{'ivar1'} = 0;
  $self->{'ivar2'} = 0;
  bless ($self, $classname);
  return $self;
  }
```

Methodendefinition

```
sub set {
  my $self = shift;
  $self->{'ivar1'} = shift;
  $self->{'ivar2'} = shift;
  ...
  }
```

Instanzbildung

```
use MeineKlasse;  # Packages der Klasse

$obj1 = new MeineKlasse;
```

Zugriff auf Elemente

```
$obj->methodenname();
```

Anhang B

Module

Module einbinden

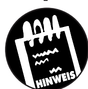
Um Elemente eines bestimmten Perl-Moduls in einem Skript verwenden zu können, müssen Sie das Modul mit Hilfe des `use`-Befehls einbinden:

```
use Tk;
```

oder

```
use Text::Wrap;
```

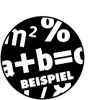
falls sich das Modul in einem Unterverzeichnis befindet.



Wie Sie danach auf die Elemente des Moduls zugreifen können, hängt davon ab, wie das Modul implementiert ist und wie es seine Elemente exportiert (siehe Kapitel 6.7). Meist stehen die Elemente in den Modulen in eigenen Namensbereichen, die genauso lauten wie das Modul. In diesem Fall kann man auf die Elemente zugreifen, indem man dem Elementnamen den Namensbereich voranstellt.

```
$Text::Wrap::columns = 12;
```

Was aber, wenn man feststellt, dass das betreffende Modul gar nicht installiert ist?



Module herunterladen und installieren

Perl verfügt über eine umfangreiche Online-Bibliothek: dem Comprehensive Perl Archive Network, kurz CPAN. Jeder Perl-Programmierer kann zu dieser Bibliothek beitragen und jeder Perl-Programmierer kann sich die Module des CPAN kostenlos herunterladen. Wenn Sie sich einen Überblick über das CPAN und die darin enthaltenen Module verschaffen wollen, starten Sie mit *http://www.perl.com/CPAN-local/README.html*. Über dieser Webseite finden Sie Informationen über die Inhalte der CPAN-Module sowie deren Installation.

Die Online-Dokumentation auf der Perl-Website beschreibt allerdings nur das manuelle Installieren von Modulen. Es geht aber auch anwenderfreundlicher: mit CPAN.PM oder PPM von ActiveState.

CPAN.pm

Mit dem CPAN.pm-Modul von Andreas König kann man das Herunterladen und Installieren von CPAN-Modulen automatisieren.

1. Stellen Sie über Modem oder ISDN-Karte eine Verbindung zum Internet her.
2. Rufen Sie das CPAN-Tool auf:

   ```
   perl -MCPAN -e shell
   ```

3. Wenn Sie das Tool das erste Mal aufrufen, werden Sie mit einer Reihe von Fragen zur Konfiguration und Anpassung an Ihr System konfrontiert. Danach erscheint ein neuer Prompt. Von hier aus können Sie nach Modulen suchen:

   ```
   cpan> i Modulname
   ```

4. und das gewünschte Modul installieren:

   ```
   cpan> install Modulname
   ```

ActiveState

Wenn Sie mit einer ActiveState-Version arbeiten (egal ob unter Windows oder Linux), können Sie neue CPAN-Module mit Hilfe des Programms PPM nachinstallieren.

1. Stellen Sie über Modem oder ISDN-Karte eine Verbindung zum Internet her.
2. Rufen Sie PPM über die Konsole auf (unter Windows die MSDOS-Eingabeaufforderung). Es erscheint ein eigener ppm-Prompt.

3. Lassen Sie sich eine Liste der verfügbaren Module anzeigen. (Eventuell sollten Sie zuvor einstellen, nach wie vielen Zeilen die Ausgabe der Liste unterbrochen werden soll.

```
ppm> set more 20
ppm> search
```

4. Wenn Sie das gewünschte Modul (ActiveState spricht von Packages) gefunden haben, lassen Sie es installieren.

```
ppm> install PackageName
```

5. Beenden Sie PPM nach erfolgreicher Dateiübertragung.

```
ppm > quit
```

Weitere Informationen zu PPM finden Sie in der Online-Dokumentation auf der ActiveState-Website (*www.activestate.com*).

Anhang C

Online-Hilfe

Wenn Sie einmal mit den Erklärungen in diesem Buch nicht weiterkommen, mehr Informationen zu einer Funktion brauchen oder Rat zu einem bestimmten Problem suchen, stehen Ihnen folgende weitere Informationsquellen zur Verfügung.

Die Manpages

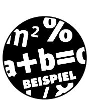
Die erste Anlaufstation bei Fragen zu Perl sollten stets die Manpages sein. Die Manpages gliedern sich in mehrere Teildokumente auf (siehe Tabelle 1.1). Um die einzelnen Dokumente aufzurufen, stehen Ihnen folgende Möglichkeiten zur Verfügung:

Unter UNIX/Linux können Sie die Manpages mit MAN DOKUMENTNAME aufrufen. Der folgende Befehl ruft beispielsweise die Dokumentation zu den vordefinierten Perl-Funktionen auf:

```
> man perlfunc
```

Die gesamte Perl-Dokumentation liegt üblicherweise auch im Perl-eigenen POD-Format vor. In diesem Format können Sie die Manpages über den Befehl PERLDOC aufrufen, der üblicherweise sowohl unter UNIX/Linux als auch unter Windows (Aufruf über die MSDOS-Eingabeaufforderung) verfügbar ist: :

```
> perldoc perldata
```



Mit Hilfe von PERLDOC können Sie zudem direkt zur Beschreibung einer bestimmten Funktion springen:

```
> perldoc -f split
```

Wenn Sie mit ActivePerl arbeiten (egal, ob unter Windows oder UNIX/Linux), können Sie die Perl-Dokumentation auch im HTML-Format lesen.

Laden Sie dazu die Datei INDEX.HTML aus dem HTML-Unterverzeichnis der ActivePerl-Installation.

Die einzelnen Manpages

Über die Aufrufe

```
> man perl
> perldoc perl
```

beziehungsweise den Link PERL unter »Core Perl Docs« in der HTML-Dokumentation von ActiveState können Sie sich eine Liste der einzelnen Perl-Dokumente mit kurzen Inhaltsangaben anzeigen lassen. Zu Ihrer Bequemlichkeit sind die wichtigsten Dokumente auch in der nachfolgenden Tabelle aufgeführt.

Tabelle C.1: Die wichtigsten Manpages

Dokument	Inhalt
perl	Übersicht über die einzelnen Dokumente
perl5005delta	Änderungen gegenüber Perl 5.005
perlboot	Einführung in OOP
perldata	Datentypen
perldebug	Debugger
perldelta	Änderungen gegenüber der letzten Version
perldos	Hinweise für Perl unter DOS
perldsc	Einführung in Datenstrukturen
perlfaq	FAQs
perlform	Report Generator
perlfunc	Beschreibung der vordefinierten Funktionen
perlguts	Interne Perl-Funktionen
perllexwarn	Warnungen des Interpreters
perllol	Arrays von Arrays
perlmod	Module
perlmodinstall	Installieren von CPAN-Modulen

Dokument	Inhalt
perlmodlib	Module schreiben und verwenden
perlnumber	Repräsentation von Zahlen
perlobj	Objekte
perlop	Operatoren
perlopentut	Tipps zu open()
perlpod	POD-Dokumentationen
perlre	Reguläre Ausdrücke
perlref	Referenzen
perlreftut	Einführung in Referenzen
perlrun	Aufruf des Interpreters
perlsub	Funktionen
perlstyle	Stil
perlsyn	Syntax
perltoc	Inhaltsverzeichnis
perltoot	OOP
perltootc	OOP
perlunicode	Unicode-Unterstützung
perlvar	Beschreibung der Standardvariablen
perlwin32	Hinweise für Perl unter DOS

Tabelle C.1: Die wichtigsten Manpages (Fortsetzung)

Die FAQs

Wenn Sie zu einem konkreten Programmierproblem einen fertigen Lösungsvorschlag suchen und ihr Problem von allgemeinem Interesse ist (so dass man davon ausgehen kann, dass andere Programmierer ebenfalls schon über dieses Problem gestolpert sind), stehen die Chancen gut, dass Sie in den FAQs fündig werden.

Die FAQ-Dokumente können Sie auf den gleichen Wegen wie die anderen Perl-Manpages aufrufen. Als Dokumentnamen geben Sie PERLFAQX an, wobei X für eine Zahl zwischen 1 und 9 steht:

PERLDOC PERLFAQ3

Eine Übersicht über die Inhalte der einzelnen FAQ-Dokumente finden Sie in PERLDOC PERLFAQ.

Die Newsgroups

Wenn Sie weder in den Manpages noch in den FAQs eine Antwort auf Ihre Frage oder eine Lösung zu Ihrem Problem gefunden haben, versuchen Sie es doch einmal in folgenden Newsgroups:

```
comp.lang.perl
de.comp.lang.perl
```

Weiterführende Bücher

»Programming Perl« von Wall, Christiansen und Schwartz, O´Reilly.

»Perl – Einführung, Anwendungen, Referenz« von Farid Hajii, Addison-Wesley.

»Perl Kochbuch« von Christiansen, Torkington, O´Reilly & Associates.

»Mastering Regular Expressions« von Jeffrey Friedl, O´Reilly & Associates.

»The Perl Journal«, Zeitschrift nur für Perl.

Anhang D

Der Debugger

Nur in den allerseltensten Fällen ist die Arbeit des Programmierers mit dem erfolgreichen Kompilieren und Linken der Anwendung abgeschlossen. Danach beginnt das Austesten des Programms, verbunden mit dem Ausmerzen auftretender Fehler (Bugs).

Der Begriff »Bug« (englisch für »Wanze«, »Insekt«) wurde übrigens an der Harvard University geprägt, wo eine in die Schaltungen eingedrungene Motte den Computer lahm legte.

Was ist ein Debugger?

Ein Debugger ist ein Programm, das ein anderes Programm schrittweise ausführen kann. Sofern in dem debuggten Programm spezielle Debug-Informationen vorhanden sind (im Programm verwendete Bezeichner, Zeilennummern etc.), kann der Debugger während der Ausführung des debuggten Programms anzeigen, welche aktuellen Werte in den Variablen des Programms gespeichert sind und welche Quelltextzeile gerade ausgeführt wird.

Wie debuggt man?

In der Praxis sieht eine Debug-Sitzung so aus, dass man ständig die folgenden drei Schritte wiederholt:

1. **Programm im Debugger ausführen**. Um ein Perl-Programm im Debugger auszuführen, rufen Sie den Perl-Interpreter mit der Option `-d` auf:

   ```
   > perl -d demo.pl
   ```

2. **Programm schrittweise ausführen**. Lassen Sie das Programm bis zu dem Quelltextabschnitt ausführen, der Ihnen verdächtig vorkommt.

   Setzen Sie dazu mit dem Debug-Befehl »`b Zeilennummer`« einen Haltepunkt (englisch: Breakpoint) in die erste Zeile des verdächtigen Codeabschnittes. Wenn Sie sich mit der Zeilennummer nicht sicher sind, lassen Sie sich mit Hilfe des Befehls `l` (l wie »line«) einen nummerierten Codeausschnitt anzeigen (beispielsweise `l 10-20`, um die 10. bis 20. Zeile anzuzeigen. Handelt es sich bei dem verdächtigen Code um eine Funktion, rufen Sie `b` mit dem Namen der Funktion auf. (Wenn Sie einen gesetzten Haltepunkt wieder löschen wollen, verwenden Sie den Befehl `d` mit der Zeilennummer des Haltepunktes.)

   Nachdem Sie den Haltepunkt gesetzt haben, können Sie das Skript mit dem Befehl c in einem Rutsch bis zu dieser Zeile ausführen lassen.

   Die verdächtigen Zeilen können Sie mit den Befehlen `s` und `n` Zeile für Zeile durchgehen. Im Unterschied zum Befehl `n`, der Funktionsaufrufe als eine Anweisung ausführt, verzweigt der Befehl `s` in Funktionen und führt auch diese Zeile für Zeile aus.

3. **Programmzustand prüfen**. Während man das Programm schrittweise ausführt, lässt man sich den Inhalt wichtiger Variablen anzeigen. So kann man gleichzeitig kontrollieren, ob die Progammanweisungen in der gewünschten Reihenfolge ausgeführt werden, und sich die Werte der Variablen wie geplant verändern.

   Den Inhalt globaler Package-Variablen können Sie mit dem Befehl `X` ausgeben. Den Inhalt lokaler Variablen geben Sie mit `x` aus:

   ```
   X $glob_var;
   x @lok_var;
   ```

   Wenn Sie `X` oder `x` ohne Variablennamen aufrufen, werden alle globalen bzw. aktuellen lokalen Variablen ausgegeben.

4. **Beenden Sie die Debug-Sitzung**. Mit dem Befehl `q` können Sie den Debugger beenden.

Hilfe zu den Debug-Befehle können Sie sich im Debugger durch Aufruf des Befehls `h h` anzeigen lassen. Ausführlichere Hilfe finden Sie in der Perl-Manpage PERLDEBUG.

Anhang E

Lösungen zu den Übungen

Kapitel 2

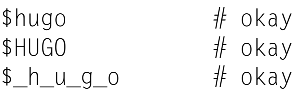
1. Setzen Sie ein Perl-Skript auf, das das folgende Voltaire-Zitat ausgibt:

```
 »Wollen Sie ein Autor sein, wollen Sie ein Buch schreiben,
dann denken Sie daran, dass es neu und nützlich oder
zumindest sehr vergnüglich sein muss.«
```

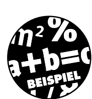
Lösung:

```
#!/usr/bin/perl -w

$text =  "Wollen Sie ein Autor sein, wollen Sie ein Buch
schreiben, dann denken Sie daran, dass es neu und nützlich
oder zumindest sehr vergnüglich sein muss.";

print "\n$text\n";
```

2. Sind die folgenden Bezeichner gültige Variablennamen?



```
$hugo           # okay
$HUGO           # okay
$_h_u_g_o       # okay
$gesetzt?       # falsch, da ungültiges Zeichen: ?
$3fach          # falsch, beginnt mit Ziffer
$dos2unix       # okay
```



3. Wie gibt man die Werte von Variablen aus? Wie kann man die Ausgabe formatieren?

   Zur Ausgabe verwendet man die Funktion `print`, der man den Variablennamen als Argument übergibt. Alternativ kann man auch einen String in doppelten Anführungszeichen übergeben, in dem der Variablenname enthalten ist. Will man die Ausgabe formatieren, muss man auf die Funktion `printf` zurückgreifen:

```
$var = 123.4567;

print 'Wert von $var: ', $var, "\n";
print "Wert von \$var: $var \n";
printf("Wert von \$var: %.2f\n", $var);
```

4. Sie haben gestern einen Scheck über folgenden Betrag gefunden: 1.000.000,01 DM. Bevor Sie den Scheck im Fundbüro abgeben, überlegen Sie sich, wie man diesen Zahlenwert in einem Perl-Skript angeben könnte.

   In Perl werden die Nachkommastellen nicht durch ein Komma, sondern durch einen Punkt abgetrennt. Tausenderstellen kann man, wenn man will, mit Unterstrichen gruppieren. In einem Perl-Skript würde die Zahl 1.000.000,01 also wie folgt aussehen:

```
$var = 1000000.01;
$var = 1_000_000.01;
```

5. Trainieren wir ein wenig die Exponentialschreibweise. Seit Einstein weiß man, dass zwischen Energie und Masse die folgende Beziehung besteht: $E = m * c^2$. Schreiben Sie ein Perl-Skript, das berechnet, wie viel Energie frei wird, wenn Sie sich von einem Augenblick zum nächsten in reine Energie verwandeln würden. Die Lichtgeschwindigkeit c können Sie dabei mit 300.000.000 m/s ansetzen. Nutzen Sie in Ihrem Skript die Exponentialschreibweise.

```
#!/usr/bin/perl -w

$gewicht = 0;
$energie = 0;

print "\n Geben Sie Ihr Gewicht in kg ein: ";
chomp($gewicht = <STDIN>);
```

```
$energie = $gewicht * 3e8 * 3e8;

printf(" Es werden %e Joule freigesetzt\n", $energie);
```

Wenn ich verpuffe, werden beispielsweise ca. 7*10^18 Joule frei. ich teile Ihnen das nicht mit, damit Sie daraus mein Gewicht zurückrechnen, sondern um Sie darauf aufmerksam zu machen, welche unglaublichen Energiemengen in uns stecken. Da 3.6*10^6 Joule genau einer Kilowattstunde entsprechen, ergibt sich daraus, dass ein Mann meines Gewichts ca. 2 * 10^12 kWh entspricht. Das sind 2*10^6, also rund 2 Millionen Gigawattstunden Energie (etwa das Vierfache des jährlichen Gesamtstromverbrauchs von Deutschland). Und wenn Sie wissen wollen, wie viel ein Mensch wert ist, dann multiplizieren Sie doch einfach die berechneten Kilowattstunden mit dem kWh-Preis Ihres Energieversorgers.

6. Was geben die folgenden Zeilen aus?

```
$wert = 2000.01;
print $wert + "1.01e3", "\n";        # 3010.01
printf "%f \n", $wert + "1.01e3";    # 3010.010000
printf "%d \n", $wert + "1.01e3";    # 3010
```

7. Und noch einmal: Was geben die folgenden Zeilen aus?

```
$meineVar = "1001 Nacht";
printf "%s \n", $meineVar;          # 1001 Nacht
printf "%d \n", $meineVar;          # 0
```

Bei der Übergabe an `printf` wird der Inhalt von `$meineVar` nicht in eine Zahl konvertiert (wie es beispielsweise in einer Addition: `$andereVar = $meineVar + 1`; der Fall wäre).

8. Kann auf der linken Seite einer Zuweisung ein Ausdruck stehen?

Nein, auf der linken Seite einer Zuweisung muss ein Bezeichner stehen, der mit einem Speicherbereich verbunden ist, in den der Wert der Zuweisung geschrieben werden kann.

Kapitel 3

1. Welches Ergebnis liefert der folgende Ausdruck?

```
8/5                   # 1.6
```

2. Wie müssten Sie den Ausdruck formulieren, damit das Ergebnis 1 lautet?

```
int(8/5)
```

3. Was ist der Unterschied zwischen

```
$var = 33;
$str = "Der Wert von var ist " . $var . "\n";
```

und

```
$var = 33;
$str = "Der Wert von var ist " + $var + "\n";
```

Der erste Ausdruck verwendet den Konkatenationsoperator. Der Wert von `$var` wird daher in einen String umgewandelt und an den vorangehenden String angehängt (»Der Wert von `var` ist 33«). Der zweite Ausdruck verwendet den numerischen Additionsoperator. Dieser versucht, den ersten String in eine Zahl umzuwandeln. Da dies nicht gelingt, interpretiert er den String als 0. Auch der zweite String wird zu 0 und der Ausdruck wird berechnet zu 0 + 33 + 0.

4. Nennen Sie drei Möglichkeiten, einer skalaren Variablen einen String zuzuweisen!

   Durch Zuweisung einer Stringkonstanten, durch Zuweisung einer skalaren Variablen, die einen String enthält, oder durch Einlesen einer Zeile von der Tastatur.

5. Was ist falsch an folgendem Code?

```
$gedicht = << HERE_DOUGLAS;


            Archibald Douglas

                 von
            Theodor Fontane

  Ich habe es getragen sieben Jahr,
  Und ich kann es nicht tragen mehr!
  Wo immer die Welt am schönsten war,
  Da war sie öd und leer.

HERE_DOUGLAS
```

Zwischen `<<` und `HERE_DOUGLAS` darf kein Leerzeichen stehen.

6. Was ist falsch an folgendem Code?

```
$gedicht = <<HERE_DOUGLAS


            Archibald Douglas

                 von
            Theodor Fontane

  Ich habe es getragen sieben Jahr,
  Und ich kann es nicht tragen mehr!
  Wo immer die Welt am schönsten war,
  Da war sie öd und leer.

  HERE_DOUGLAS
```

   Hinter dem ersten `HERE_DOUGLAS` fehlt das Semikolon und das abschließende `HERE_DOUGLAS` steht nicht am Anfang der Zeile.

7. Wodurch wird die Reihenfolge festgelegt, in der Ausdrücke mit mehreren Operatoren ausgewertet werden?

   Durch die Rangordnung der Operatoren, durch die Assoziativität (bei Operatoren gleicher Rangordnung) und durch das Setzen von Klammern.

8. Welchen Wert haben die folgenden Ausdrücke?

```
$var1 = 12;
$var2 = $var1++ * (3 + 4);    # $var2 = 84
$var1 = 12;
$var2 = ++$var1 * (3 + 4);    # $var2 = 91
```

Kapitel 4

1. Ist die folgende `if`-Bedingung korrekt?

```
$i = 3;

if($i)
  {
  # tue irgendetwas
  }
```

   Ja. Die Anweisungen unter der `if`-Bedingung werden ausgeführt, wenn der Wert der Variablen `$i` ungleich 0 ist.

2. Ist die folgende `while`-Bedingung korrekt? Wenn ja, wie könnte Sie beendet werden?

```
$i = 3;

while($i)
  {
  ...
  }
```

Ja und nein. Das hängt, von dem Anweisungsblock der `while`-Schleife ab. Dieser sollte Code enthalten, der irgendwann zum Verlassen der Schleife führt (`last`-Befehl).

3. Wann werden die folgenden Bedingungen als »wahr« ausgewertet? (Hilfe: Betrachten Sie `$x` und `$y` als Koordinaten in der Ebene.)

```
if (($x < 35 && $x > 20) && ($y < 100 && $y > 80)
    || ($x < 150 && $x > 120) && ($y < 95 && $y > 75))
```

und

```
if (sqrt($x**2+$y**2) == 1)
```

Die erste Bedingung ist erfüllt, wenn die Koordinate (`$x, $y`) in einem der beiden angegebenen Rechtecke liegt.

Die zweite Bedingung ist erfüllt, wenn die Koordinate auf einem Einheitskreis um den Ursprung liegt (Satz des Pythagoras).

4. Worin besteht der Fehler in der folgenden Schleife?

```
$i = 1;
$j = 0;
$k = 0;

print "i \t j \t k";
while ($i < 10)
    {
    $j = $i*$i - 1;
    $k = $j*$j - 1;
    print "$i \t $j \t $k\n";
    }
```

Der Fehler liegt darin, dass die Schleifenvariable `$i` nicht verändert und die Schleife folglich nicht beendet wird.

5. Die folgende Schleife definiert mit Hilfe des Komma-Operators zwei Schleifenvariablen. Wie sieht die Ausgabe der Schleife aus?

```
#!/usr/bin/perl -w

for($n = 0, $m = 0; $n < 10 && $m <3; $n++, $m += 2)
  {
  print "n * m = ", $n*$m, "\n";
  }
```

Ausgabe:

```
n * m = 0
n * m = 2
```

6. Setzen Sie eine `for`-Schleife zur Berechnung der ersten hundert Quadratzahlen auf.

```
#!/usr/bin/perl -w

for($n = 1; $n <= 100; $n++)
  {
  printf("Quadrat von %d = %d\n",$n,$n*$n);
  }
```

7. Wandeln Sie die `for`-Schleife aus Übung 6 in eine `while`-Schleife um.

```
$n = 1;
while($n <= 100)
  {
  printf("Quadrat von %d = %d\n",$n,$n*$n);
  $n++;
  }
```

8. Wandeln Sie die `for`-Schleife aus Übung 6 in eine `foreach`-Schleife um und verwenden Sie die Standardvariable `$_`.

```
foreach (1..100)
  {
  printf("Quadrat von %d = %d\n",$_,$_*$_);
  }
```

9. Setzen Sie ein Perl-Skript auf, das für Winkel von 0 bis 360 Grad den Sinus berechnet. Für die Berechnung des Sinus können Sie die vordefinierte Perl-Funktion `sin` verwenden, der Sie den Winkel in Bogenmaß (Radiant)[1] übergeben. Formatieren Sie die Ausgabe in einer Tabelle mit

1 1 rad = $360°/2\pi$; $1° = 2\pi/360$ rad.

zwei Spalten für den Winkel und den Sinus. Es genügt, wenn Sie die Vielfache von 30 Grad ausgeben.

```
#!/usr/bin/perl -w

$rad = 0;

print "\n";
print "  Winkel  \t Sinus\n";
print "______________________________\n";

for($w = 0; $w <= 360; $w += 30)
  {
  $rad = ($w * 2 * 3.1415) / 360;
  printf("  %d      \t %.2f\n",$w, sin($rad) );
  }
```

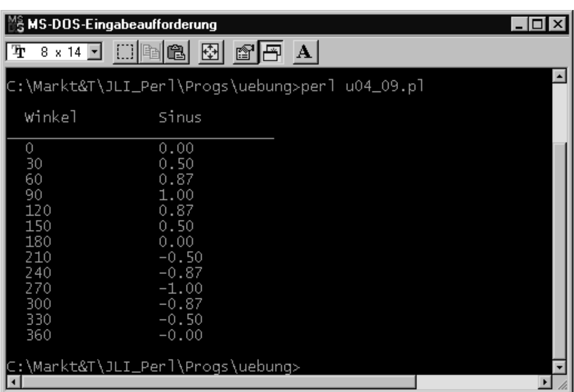


Abb. E.1: Ausgabe der Sinuswerte

10. Benutzen Sie den Modulo-Operator, um zu überprüfen, ob eine Zahl gerade ist.

    Wenn Sie zwei Zahlen mit dem Modulo-Operator % dividieren, errechnet der Compiler, wie oft der Divisor in den Dividenden passt. Die Differenz zwischen diesem Wert und dem Dividend wird zurückgeliefert.

Um zu prüfen, ob eine Zahl gerade ist, dividiert man sie als `modulo 2` und prüft, ob das Ergebnis gleich 0 ist (die Division ohne Rest aufging):

```
#!/usr/bin/perl -w

for($zahl = 0; $zahl <= 10; $zahl++)
  {
  if (($zahl % 2) == 0)
    {
    print "$zahl ist gerade\n";
    }
  }
```

11. Nutzen Sie die Binärcodierung des Computers, um zu entscheiden, ob eine Zahl gerade ist.

    In Binärdarstellung kann man gerade Zahlen einfach daran erkennen, dass das letzte Bit 0 ist. Wenn man nun eine Zahl mit der Eins bitweise UND-verknüpft, erzeugt der Computer als Ergebnis eine Zahl, für die nur die Bits auf 1 gesetzt sind, die in beiden verknüpften Zahlen bereits auf 1 stehen.

    0010 1100 // gerade Zahl

    0000 0001 // Eins

    0000 0000 // AND-Verknüpfung

    Ist das Ergebnis 0, muss die erste Zahl gerade gewesen sein. Die folgende Schleife gibt daher nur die Quadratzahlen für gerade Zahlen aus:

```
#!/usr/bin/perl -w

for($zahl = 0; $zahl <= 10; $zahl++)
  {
  if (($zahl & 1) == 0)
    {
    print "$zahl ist gerade\n";
    }
  }
```

Kapitel 5

1. Werden Listen mit runden oder eckigen Klammern definiert?

   Mit runden Klammern!

2. Definieren Sie möglichst effizient eine Liste für folgende Zahlen: -3, -2, -1, 0, 1, 2, 3, 33, 34, 35, 36, 37, 38, 39, 40.

   ```
   (-3..3, 33..40)
   ```

3. Mit welchem Präfix beginnen Array-Variablen?

   Mit `@`.

4. Wie lauten die Indizes des ersten und des letzten Elements des folgenden Arrays?

   ```
   @meinArray = (1..100);
   ```

   Der erste Index ist 0, der letzte 99.

5. Wie kann man sich den höchsten Index eines Arrays und wie die Anzahl der Elemente in einem Array zurückliefern lassen.

   `$#meinArray` liefert den Index des letzten Elements

   `scalar @meinArray` liefert die Anzahl Elemente im Array

6. Simulieren Sie die Funktionen `shift` und `unshift` durch `splice`-Aufrufe.

   ```
   $erstes = splice(@array, 0, 1);  # shift
   splice(@array, 0, 0, $element);  # unshift
   ```

7. Was passiert, wenn Sie den Wert eines Array-Elements aus Versehen mit `@meinArray[5]` statt mit `$meinArray[5]` zurückliefern lassen?

   Abgesehen davon, dass Sie sich eine Verwarnung des Interpreters einhandeln, liefert er das sechste Element zurück, aber nicht als einzelnes Element, sondern als einelementige Liste. Allgemein gilt, dass Sie durch das Anhängen von Indizes an ein Array ein Teilarray zurückliefern lassen:

   ```
   ($elem1, $elem2) = @meinArray[5,7];
   ```

8. Wie kann man zwei Arrays aneinander hängen?

   Übergeben Sie `splice` das erste Array, den größten Index des ersten Arrays + 1 (ans Ende anhängen), eine Null (nichts ersetzen) und das zweite Array:

```
@array1 = (1..10);
@array2 = (22, 23, 24);

splice(@array1, $#array1 +1, 0, @array2);
```

9. Setzen Sie eine Vergleichsfunktion zum Sortieren von Spielkarten auf.

Für die Verwirklichung einer solchen Vergleichsfunktion gibt es natürlich eine Vielzahl von Möglichkeiten. Ich habe mich dazu entschlossen, in der Vergleichsfunktion ein Array zu definieren, in dem alle möglichen Kartenwerte in der richtigen Reihenfolge abgelegt sind. (Das Schlüsselwort `my` sorgt übrigens dafür, dass die eingerichteten Variablen nur in der Funktion gültig sind. Dies geschieht aus Sicherheitsgründen, siehe Kapitel 6.)

Wird die Vergleichsfunktion aufgerufen, prüft sie, mit welchen Elementen im Vergleichsarray die beiden übergebenen Elemente `$a` und `$b` übereinstimmen. Die Indizes der Elemente im Vergleichsarray geben dann Aufschluss über den zurückzuliefernden Wert.

```
#!/usr/bin/perl -w

sub karten_vgl
  {
  my $index;
  my $ai;
  my $bi;
  my @reihenfolge = ("Sieben", "Acht", "Neun", "Zehn",
                     "Bube", "Dame", "Koenig", "As");

  for($index = 0; $index < $#reihenfolge + 1; $index++)
    {
    $ai = $index if ($a eq $reihenfolge[$index]);
    $bi = $index if ($b eq $reihenfolge[$index]);
    }
  return -1 if ($ai < $bi);
  return  0 if ($ai == $bi);
  return  1 if ($ai > $bi);
  }

@array = ("Acht", "Koenig", "Neun", "Dame");
@sortiert = sort karten_vgl @array;
print "@sortiert\n";
```

10. Mit welchem Präfix beginnen Hash-Variablen?

    Mit einem `%`.

11. Schreiben Sie ein Programm, das eine Adresse von Tastatur einliest, in einem Hash speichert und dann ausgibt.

```
#!/usr/bin/perl -w

%adresse = ();

print "Name: ";
chomp( $adresse{"Name"} = <> );

print "Strasse: ";
chomp( $adresse{"Strasse"} = <> );

print "Hausnummer: ";
chomp( $adresse{"Hausnummer"} = <> );

print "Stadt: ";
chomp( $adresse{"Stadt"} = <> );

print "\n\nEingebene Adresse: \n\n";
print " Name       : ", $adresse{"Name"}, "\n";
print " Strasse    : ", $adresse{"Strasse"}, "\n";
print " Hausnummer : ", $adresse{"Hausnummer"}, "\n";
print " Stadt      : ", $adresse{"Stadt"}, "\n";
```

12. Schreiben Sie ein Programm, das Wochentage mit Hilfe eines Hash in Zahlen umwandelt.

```
#!/usr/bin/perl -w

%tage = ("Montag" => 1, "Dienstag" => 2,
         "Mittwoch" => 3, "Donnerstag" => 4,
         "Freitag" => 5, "Samstag" => 6, "Sonntag" => 7);


$eingabe = "Mittwoch";
print "$eingabe: ", $tage{"$eingabe"}, "\n";
```

13. Wozu dienen die Funktionen `keys` und `values`?

    Die Funktion `keys` liefert die Schlüssel eines Hash als Liste zurück.

    Die Funktion `values` liefert die Werte eines Hash als Liste zurück.

Kapitel 6

1. Durch welches Schlüsselwort werden Funktionsdefinitionen eingeleitet?

   Durch das Schlüsselwort `sub`.

2. Wie können Sie mit einem einzigen zusätzlichen Zeichen das Listing aus Abschnitt 6.2.1 (Berechnung der Mehrwertsteuer) so ändern, dass die Preise im Array `@liste` durch die berechnete Mehrwertsteuer ersetzt werden?

```
#!/usr/bin/perl -w

@liste = (100, 3000, 89.90, 125.40);

sub mwst { $_ * 0.16; }              # Funktionsdefinition
@mwst_liste = map(mwst, @liste);     # Übergabe als Argument

print "@mwst_liste\n";
```

   Ersetzen Sie in der Funktion `mwst` den Operator * durch *=.

3. Wie deklariert man Variablen, die zu einer Funktion lokal sind?

   Mit dem Schlüsselwort `my` (oder `local`).

4. Wie kann eine Funktion Werte aus dem Aufruf entgegennehmen?

   Über das Array @_.

5. Wie kann man verhindern, dass die Funktion die Variablen, die als Argumente übergeben werden, verändert?

   Indem man die Werte der Argumente aus @_ einliest und in lokale Variablen kopiert. Um ganz sicherzugehen, kann man die Argumente mit `shift` auslesen, um so gleichzeitig das Array @_ zu leeren.

```
sub demoFunk {
  my $param1 = shift @_;
  my $param2 = shift @_;
```

6. Wie kann eine Funktion Ergebniswerte zurückliefern?

   Mit Hilfe des Schlüsselworts `return` (oder über die letzte Anweisung der Funktion).

7. Setzen Sie eine Funktion auf, die ein Array als Argument übernimmt und die Summe der Elemente im Array zurückliefert.

```
#!/usr/bin/perl -w

sub summe {
  my @werte = @_;
  my $loop;
  my $summe = 0;

  for ($loop=0; $loop <= $#werte; $loop++) {
     $summe += $werte[$loop];
     }

  return $summe;
  }


$ergebnis = summe(1, 2, 3, 4, 5, 6, 7);
print "$ergebnis \n";
```

8. Wann muss man die Aufrufargumente zu einer Funktion in runde Klammern setzen?

   Wenn die Funktion in einem Listenkontext aufgerufen wird und es nicht klar ist, ob die hinter dem Funktionsnamen aufgeführten Elemente zum Funktionsaufruf oder zur Liste gehören.

9. Wie deklariert man eine Schleifenvariable, die zu der Schleife lokal ist?

   Indem man die Schleifenvariable im Schleifenkopf oder Schleifenrumpf mit `my` deklariert.

```
for (my $loop=0; $loop <= $#werte; $loop++) {
 ...
   }
```

10. Was ist der Unterschied zwischen einer `my`-Variable im Dateibereich und einer globalen Variablen?

    Die globale Variable gehört ihrem Package (Namensbereich) an, die `my`-Variable gehört keinem Package an.

11. Welcher Unterschied besteht zwischen `my` und `local`?

    Mit `my` deklarierte Variablen sind im Gegensatz zu `local`-Variablen nicht in untergeordneten (aufgerufenen) Funktionen sichtbar.

12. Schreiben Sie das Modul MATHEXT.PM so um, dass es den Bezeichner der Funktion `fakultaet` optional exportiert, und formulieren Sie das Skript FAKULTAET2.PL so um, dass es den Bezeichner importiert.

    Das Modul:

```
package MathExt2;
require Exporter;
@ISA    = qw(Exporter);
@EXPORT_OK = qw(fakultaet);

sub fakultaet {
  my $zahl = shift;
  my $loop;
  my $fakul = 1;

  for ($loop=$zahl; $loop > 1; $loop--) {
     $fakul *= $loop;
     }

  return $fakul;        # Ergebnis zurückliefern
  }

1;
```

    Das Skript:

```
#!/usr/bin/perl -w

use MathExt2 qw(fakultaet);


$eingabe = 0;
print "\n Geben Sie eine Zahl ein, deren Fakultaet ",
      "berechnet werden soll:  ";

chomp ($eingabe = <STDIN>);


$fakul = fakultaet($eingabe);    # Aufruf der
                                 # Modul-Funktion

print "\n$eingabe! = $fakul\n";
```

Kapitel 7

1. Was ist eine Referenz?

   Eine Referenz ist eine skalare Variable, die als Wert die Adresse einer anderen Variablen (Skalar, Array, Hash) enthält.

2. Wie deklariert man Referenzen auf Skalare, Arrays und Hashes?

```
$ref = \$var;        # Referenz auf Skalar
$ref = \@array;      # Referenz auf Array
$ref = \%hash;       # Referenz auf Hash
```

3. Kann man auch Referenzen auf Funktionen einrichten? Wenn ja, welchen Sinn könnte das haben?

   Ja:

```
$ref_f = \&funktionsname;
```

   Auf dem Umweg über die Referenzen könnte man beispielsweise Funktionen in Arrays verwalten.

4. Was liefern die folgenden Dereferenzierungen zurück? In den Beispielen ist absichtlich nicht angegeben, worauf die Referenz `$ref` verweist, da Sie dies aus der Syntax der Derefenzierung ablesen sollen.

```
a)  ${$ref}[0]          # Array-Element
b)  $$ref               # Skalar
c)  @{$ref}             # Array
d)  @$ref               # Array (wie c)
e)  %$ref               # Hash
f)  $ref->[0]           # Array-Element (wie a)
g)  $ref->{'key'}       # Hash-Element
h)  %{$ref}             # Hash (wie e)
i)  ${$ref}{'key'}      # Hash-Element (wie g)
```

5. Bauen Sie ein Hash von Hashes auf.

```
#!/usr/bin/perl -w

%hash1  = ("Vorname" => "Rip",
           "Nachname" => "van Winkle");
%hash2  = ("Vorname" => "Peter", "Nachname" => "Kreuter");
%bighash   = ("h1" => \%hash1, "h2" => \%hash2);
```

```
foreach $bigkey (keys %bighash)
  {
  $subhash_ref = $bighash{$bigkey};

  foreach $key (keys %{$subhash_ref}) {
     print "$key: ", "$subhash_ref->{$key} / " ;
     }
  print "\n";
  }
```

Kapitel 8

1. Welche Arten von Kontext unterscheidet Perl?

   Skalaren Kontext und Listenkontext. Den skalaren Kontext kann man weiter unterteilen in einen numerischen, einen Booleschen und einen String-Kontext.

2. Welcher Kontext liegt in den folgenden Konstrukten vor?

```
a)  print @array          # print erzeugt Listenkontext
b)  print $var;           # print erzeugt Listenkontext
c)  if($var == 1)         # Boolescher Kontext
d)  3 + @array;           # Numerischer Kontext
```

3. Wie kann man einen skalaren Kontext erzwingen?

   Mit Hilfe des Schlüsselworts `scalar`.

4. Nennen Sie vier Fälle, in denen die Standardvariable `$_` verwendet werden kann.

- ✘ in `foreach`-Schleifen
- ✘ von der Funktion `print`
- ✘ von der Funktion `chomp`
- ✘ von der Funktion `map`
- ✘ von der Funktion `grep`
- ✘ von den Operatoren `m//` und `s//` für das Pattern Matching
- ✘ vom Eingabeoperator

Kapitel 9

1. Mit dem Eingabeoperator kann man einzelne Daten von der Konsole einlesen, man kann Dateien aus der Kommandozeile einlesen und man kann Dateien über ein Datei-Handle einlesen. Wie wird der Eingabeoperator dazu jeweils aufgerufen?

```
$eingabe = <STDIN>;            # Daten von Konsole einlesen

while ($zeile = <>) {          # Dateien aus Kommandozeile
  chomp ($zeile);
  ...

my $dateiname = "daten.txt";  # Datei über Datei-Handle
open(DATEI, "< $dateiname");
while(my $zeile = <DATEI>)
    {
    chomp($zeile);
    ...
```

2. In Übung 9 aus Kapitel 4 haben Sie ein Skript aufgesetzt, das den Sinus zu verschiedenen Winkel ausgibt. Schreiben Sie dieses Programm um, so dass es die Ausgabe mit Hilfe des Report-Generators `write` erzeugt.

```
#!/usr/bin/perl -w

use Math::Trig;

format STDOUT_TOP =
 --------------------------
 Winkel           Sinus
 --------------------------
.

format STDOUT =
@>>>>>          @#.####
  $w,           sin($rad)
.


$rad = 0;
print "\n";

for($w = 0; $w <= 360; $w += 30)
  {
  $rad = ($w * 2 * pi) / 360;
  write;
  }
```

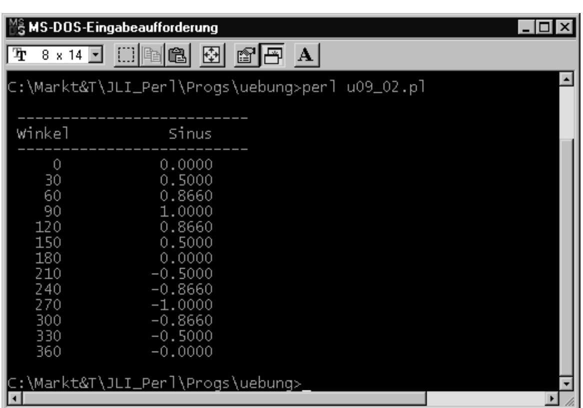


Abb. E.2: Ausführung des Skripts

3. Wie öffnet man eine Datei »demo.txt« zum Lesen, Schreiben, Anhängen?

```
open(FH, "demo.txt");      # Öffnet die Datei zum Lesen
open(FH, "< demo.txt");
open(FH, "> demo.txt");    # Öffnet die Datei zum Schreiben
open(FH, ">> demo.txt");   # Öffnet die Datei zum Anhängen
```

4. Ändern Sie das Skript SCHREIBEN.PL so ab, dass der Name der zu öffnenden Datei über die Konsole abgefragt wird

```
#!/usr/bin/perl -w
use strict;

my @persondaten = ();
my $dateiname = "";       # Name der zu öffnenden Datei

print "Name der Ausgabedatei: ";
chomp($dateiname = <>);

# Datei öffnen und mit Datei-Handle DATEI verbinden

  open(DATEI, ">> $dateiname")
    or
    die "\nDatei $dateiname konnte nicht geoeffnet
werden\n";

# Daten von Konsole einlesen
...
```

5. Welches ist der kürzeste Weg, den Inhalt einer Datei auf die Konsole auszugeben?

```
#!/usr/bin/perl -w

while (<>) {
  print;
  }
```

6. Wie kann man den gesamten Inhalt einer Datei in einen String einlesen?

   Indem man den Eingabeoperator in den Schlürfmodus versetzt:

```
undef $/;
my $text;

$text = <DATEI-Handle>;
```

7. Wie kann man den Inhalt mehreren Dateien in einen String einlesen?

   Indem man den Eingabeoperator in den Schlürfmodus versetzt und die Dateien in der Kommandozeile übergibt.

```
undef $/;
my $text;

$text = <>;
Aufruf: > perl skript.pl datei1 datei2 datei3
```

Kapitel 10

1. Wie lauten die Pattern-Matching-Operatoren zum Suchen sowie zum Suchen und Ersetzen?

```
m/Suchbegriff/
s/Suchen/Ersetzen/
```

2. Wie kann man erreichen, dass eine Suche alle Vorkommen findet?

   Indem man die Option `g` verwendet: `m/Suchbegriff/g`

3. Wie kann man die gefundenen Vorkommen in Variablen zurückliefern lassen?

   Indem man den Suchoperator im Listenkontext aufruft:

```
@array = $text =~ m/Suchbegriff/g;
```

oder die zu den Vorkommen gehörenden Musterteile in Klammern fasst und die Vorkommen in den Variablen `$1`, `$2` etc. abgreift.

```
$text =~ m/<BODY>(.*)<\/BODY>/;
$html .= "$1";
```

4. Wie kann man erreichen, dass bei der Suche nicht zwischen Groß- und Kleinschreibung unterschieden wird?

   Indem man die Option `g` verwendet: `m/Suchbegriff/i`

5. Was suchen die folgenden regulären Ausdrücke

```
/\s*;\s*/                      # Semikolon mit beliebig
                               # viel Whitespace drum herum
s/,/\./g                       # tauscht Komma gegen Punkt
s/^([^ ]*) *([^ ]*)/$2 $1/;    # tauscht die ersten
                               # beiden Wörter
/(\b\d+\.\d*\b)/g              # findet Bruchzahlen
```

6. Setzen Sie reguläre Ausdrücke auf, die

```
a) das erste Wort im Suchstring finden
b) das erste Wort in jeder Zeile finden
c) Stabreime entdecken können (zwei aufeinander folgende
   Wörter mit gleichem Anfangsbuchstaben)
d) Folgen von drei gleichen Buchstaben entdecken
```

   Lösungen:

```
a) m/^\w+/g;
b) m/^\w+/mg;
c) m/(\b(\w)\w+\s+\b\2\w+)/ig;
d) m/((.)\2\2)/g
```

7. Verbessern Sie das Programm WOERTER.PL aus Kapitel 5.3.3, so dass vor der Zählung sämtliche Satzzeichen aus dem Text verbannt werden.

```
#!/usr/bin/perl -w

use locale;                    # Deutsche Umlaute erhalten

print "\n";
@zeilen = <STDIN>;

foreach (@zeilen) {
  chomp($_);
  @woerter = split(' ',$_);
```

```
    foreach $wort (@woerter) {
       $wort =~ s/\W//g;        # Nichtwortzeichen löschen
       next if ($wort eq '');   # Leere Wörter löschen
       $woerterHash{lc($wort)}++;
       }
    }

  foreach $key (sort keys %woerterHash) {
    printf("%15s : %s\n", $key, $woerterHash{$key});
    }
```

8. Lesen Sie eine HTML-Datei ein und färben Sie alle Überschriften rot.

```
#!/usr/bin/perl -w
use strict;

open(EIN, "< $ARGV[0]")
    or
    die "\nDatei $ARGV[0] konnte nicht geoeffnet werden\n";

open(AUS, "> $ARGV[1]")
    or
    die "\nDatei $ARGV[1] konnte nicht geoeffnet werden\n";

undef $/;
my $text = <EIN>;

$text =~ s/<([Hh]\d)>/<$1 style="color: red">/g;

print AUS $text;

close(EIN);
close(AUS);
```

Kapitel 11

1. In welcher Beziehung stehen Klassen und Objekte zueinander?

   Klassen sind abstrakte Beschreibungen einer Gruppe gleichartiger Objekte. Im Programm stellt jede Klasse einen »Datentyp« dar und die Objekte sind die Elemente dieses Datentyps.

2. Was ist eine Instanz?

   Als Instanzen bezeichnet man die Variablen von Datentypen.

3. Gibt es einen Unterschied zwischen einer Instanz und einem Objekt?

   Wenn man beide Begriffe trennen möchte, so bezeichnet man das Speicherabbild als Objekt und den Variablennamen, der mit dem Objekt verbunden ist, als Instanz.

4. Welche Rolle spielen Packages für die Klassendefinition in Perl?

   Perl erlaubt keine Deklaration von selbstdefinierten Datentypen. Klassen werden in Perl daher ersatzweise in Packages gekapselt.

5. Setzen Sie die Minimalversion eines Konstruktors auf!

```
sub new {
  bless ();
  }
```

6. Was unterscheidet eine Methode von einer Funktion?

   Methoden übernehmen als erstes Argument eine Referenz auf das Objekt, für das sie aufgerufen wurden. Über diese Referenz können Sie auf die Klassenelemente des aufrufenden Objekts zugreifen.

7. Wandeln Sie VEKTOR.PL in ein pm-Modul um.

   Modul:

```
#***************** Definition der Klasse Vektor ********
package Vektor;

sub new {
  my $classname = shift;
  my $self = {};
  $self->{x} = 0;
  $self->{y} = 0;
  bless ($self, $classname);
  return $self;
  }

sub get {
  my $self = shift;
  return ($self->{x}, $self->{y});
  }

sub set {
  my $self = shift;
  $self->{x} = shift;
  $self->{y} = shift;
  }
```

```
sub addiere {
  my $self = shift;
  $self->{x} += shift;
  $self->{y} += shift;
  }

1;
```

Skript:

```
#!/usr/bin/perl -w

use Vektor;

$vektobj1 = new Vektor;
print "Vektor = (",join(':',$vektobj1->get),")\n";
...
```

Kapitel 12

1. Welches Modul benötigt man für die Erstellung von fensterbasierten Programmen mit Perl?

   Das Modul `Tk`.

2. Wie erzeugt man das Hauptfenster einer Anwendung?

   Durch Instantiierung eines `MainWindow`-Objekts.

   ```
   use Tk;

   my $hauptfenster = new MainWindow;
   ```

3. Was bewirkt der Aufruf `MainLoop`?

   Durch Aufruf von `MainLoop` wird die Ereignisbehandlung des Skripts in Gang gesetzt. `MainLoop` nimmt die Betriebssystem-Nachrichten entgegen und leitet sie an die jeweiligen Fenster weiter.

4. Wie lauten die Methoden zur Erzeugung von Eingabefeldern, Labels, Bildern, Schaltern, Container-Fenstern?

   ```
   $hauptfenster->Entry
   $hauptfenster->Label
   $hauptfenster->Photo
   $hauptfenster->Button
   $hauptfenster->Frame
   ```

5. Wie kann man ein Steuerelement konfigurieren?

   Jedes Steuerelement verfügt über bestimmte Optionen (siehe `Tk`-Dokumentation), denen man bei Erzeugung des Steuerelements oder über die `configure`-Methode Werte zuweisen kann.

```
my $schalter = $hauptfenster->Button(
                               -text => "Drück mich",
                               -command => \&zinsen_berechnen
                               );
$schalter->configure("-image" => $meinbild);
```

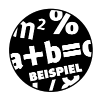
6. Wie kann man Steuerelemente in einem Fenster arrangieren?

   Durch die Aufteilung in Frames und mit Hilfe der Optionen zur `pack`-Methode.

Anhang F

ASCII-Tabelle

Dec	Hex	Zeichen	Dec	Hex	Zeichen	Dec	Hex	Zeichen	Dec	Hex	Zeichen
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	NL	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	NP	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	18	CAN	61	3D	=	93	5D	]	125	7D	}
30	19	EM	62	3E	>	94	5E	^	126	7E	~
31	1A	SUB	63	3F	?	95	5F	_	127	7F	DEL



Anhang G

Webadressen

Die offizielle Perl-Website

http://www.perl.com

Die ActiveState-Website

www.activestate.com

Der OmniHTTPd-Server

http://www.omnicron.ab.ca

Die Markt&Technik-Website

http://www.mut.de

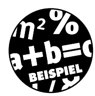
Unter *www.mut.de/books/382725841* finden Sie die Beispielskripten zu diesem Buch.

E-Mail-Adresse des Autors

dirklouis@cs.com

Stichwortverzeichnis

-- (Dekrement) 84
' (Strings) 44
- (Subtraktion) 77
- (Vorzeichen) 77
! (Logik) 105
!= (Vergleich) 102
" (Strings) 44
(Kommentar) 26
$ 212
$ (Reguläre Ausdrücke) 239
$ (Skalar) 49, 126, 145, 185
$" 203
$, 212, 213
$. 211
$/ 203, 209
$_ 115, 128, 202, 208
$| 213
$1 242
$a 135, 203
$ARG 203
$b 135, 203
% (Hash) 49, 144
% (Modulo) 77
%ENV 203, 227
& (Bitoperator) 70
&& (Logik) 105
( ) (Listen) 122
( ) (Listenkontext) 199
( ) (Reguläre Ausdrücke) 236, 242
* (Multiplikation) 77
* (Reguläre Ausdrücke) 238
** (Exponent) 77
. (Konkatenation) 87
. (Nachkommastellen) 47
. (Reguläre Ausdrücke) 237
.. (Bereiche) 123
/ (Division) 77
/ (Reguläre Ausdrücke) 231
: (Reguläre Ausdrücke) 242
? (Reguläre Ausdrücke) 238
?: (Bedingungsoperator) 114, 200
@ (Array) 49, 125
@_ 160, 168, 203
@ARGV 227
@EXPORT 179
@EXPORT_OK 179
@ISA 179
[ ] (Indizierung) 126
[ ] (Referenzen) 188
[ ] (Reguläre Ausdrücke) 236
\ (Escape-Operator) 45
\ (Referenz) 185
\n 226
^ (Bitoperator) 70
^ (Logik) 105
^ (Reguläre Ausdrücke) 239
_ (Tausenderstellen) 47
{ } (Blöcke) 99
{ } (Dereferenzierung) 186
{ } (Hashes) 144
{ } (Referenzen) 189
{ } (Reguläre Ausdrücke) 238
| (Bitoperator) 70
| (Reguläre Ausdrücke) 236
|| (Logik) 105
~ (Bitoperator) 70
+ (Addition) 77
+ (Reguläre Ausdrücke) 238
+ (Vorzeichen) 77
++ (Inkrement) 84
+< (Datei-Modus) 220
< (Datei-Modus) 220, 377
< (Vergleich) 102
<< (Bitoperator) 70
<< (Datei-Modus) 220, 377
<< (HERE-Text) 86
<= (Vergleich) 102
<=> (Vergleich) 135
<> (Eingabeoperator) 58, 197, 206, 223
= (Zuweisung) 51, 76
=~ (Regul. Ausdrücke) 231
== (Vergleich) 102
=> (Hash-Definition) 144
> (Datei-Modus) 220, 377
> (Vergleich) 102
>= (Vergleich) 102
>> (Bitoperator) 70



A
ActivePerl 17
– für Unix/Linux 22
– für Windows 23
– HTML-Dokumentation 24
ActiveState 17, 350
Addition 77
Adressoperator 185
and (Logik) 105
Anführungszeichen 46
Anweisungen 35, 55
– bedingte 114
– Blöcke 99
Anweisungsblöcke 99
– lokale Variablen 175
Apache-Server 307
Argumente 161
Arithmetische Operatoren 76
Arrays 125
– als Elemente von Hashes 189
– als Funktionsargumente 162
– Anzahl der Elemente 126
– Array-Elemente trennen 212, 213
– aus Strings erstellen 139
– ausgeben 129
– definieren 125
– durchlaufen 127
– Elemente ersetzen 131
– Elemente hinzufügen 130
– Elemente löschen 130, 131
– Elemente tauschen 133
– foreach-Schleifen 127
– höchster Index 127
– in Arrays suchen 137
– in Strings umwandeln 142
– Indizierung 126
– map-Funktion 127
– sortieren 132
– von Arrays 188
– von Hashes 189
– Zugriff auf Elemente 126
– Zuweisung an Skalare 126
Assoziativität 93
Ausdrücke 56
– arithmetische 101
– Auswertung 92
– in Bedingungen 101
– Klammern 93
– Reguläre Ausdrücke 229
– Seiteneffekte 94
– verkürzte Auswertung 117

B
Backslash 45
Bedingungen 100
– Auswertung von Ausdrücken 101
– Kontext 198
– logische Verknüpfungen 104
– Vergleichsoperatoren 101
Bedingungsoperator 114, 200
Beispielskripten
– aktien.pl 141
– binsuche.pl 139
– cds.pl 301
– cgidiagr.pl 314
– diagramm.pl 286
– douglas.pl 86
– euro1.pl 60
– euro2.pl 62
– fakultaet.pl 169
– fakultaet2.pm 178
– formular.pl 320
– furby1.pl 124, 271
– furby2.pl 273
– furby3.pl 275
– gaestebuch.pl 324
– gifte1.pl 231
– gifte2.pl 232
– gifte3.pl 235
– hallo.pl 26
– hallotk.pl 270
– haushoehe.pl 80
– kaninchen.pl 111
– lesen.pl 224
– lesen2.pl 226
– links.pl 243
– MathExt.pm 177
– menue.pl 109
– messkurve.pl 290
– palindrom.pl 89
– prim.pl 138
– rechtschreibung.pl 240
– schreiben.pl 221
– skhan1.pl 207
– skhan2.pl 209
– summe.pl 200
– testcgi.pl 312
– vektor.pl 260
– webinhalte.pl 304
– woerter.pl 148
– wurzel.pl 100
– zeilen.pl 211
– zinsen1.pl 78

– zinsen2.pl 277
Beispielskripten, herunterladen 387
Bereichsoperator 123
Bezeichner 35, 50
– aus Modulen 179
– qualifizierte 173
Bezugsquellen
– Beispielskripten 387
– Perl 16
– Perl-Module 350
Binärdarstellung, von Daten 66
Binärdateien 225
Binäre Suche 138
Binärmodus 225
Binärzahlen 48
binmode 225
Bitprogrammierung 70
bless 258
Blöcke 99
– lokale Variablen 175
Boolesche Werte 101
BubbleSort 133
Bücher 356
Bytecode 33

C
Callback-Funktionen 269
CGI 307
– Codierung von Browserdaten 318
– Formulareingaben auswerten 317
– Gästebuch 322
– GET 317
– Grafiken zurückliefern 313
– Header 312
– Kommunikation Browser-Server-Skript 311
– Perl-Skripten als Bearbeiter angeben 320
– Perl-Skripten testen 321
– POST 318
– Sicherheit 326
– Umgebungsvariablen 313
– Webseiten zurückliefern 311
– Webserver konfigurieren 307
chmod 69
chomp 58, 91
chop 91
chr 90
close 221
cmp (Vergleich) 135
Compiler 33
constant 48
CPAN 350
CPAN.pm 350
CSV-Dateien 140

D
Dateien 218
– absatzweise lesen 224
– anhängen 220, 377
– aus Dateien lesen 223
– Binärdateien 225
– Binärmodus 225
– binmode 225
– Dateiendezeichen 209
– Datei-Handle 219
– Dateiname 219
– Datei-Tests 222
– Eingabeoperator 223
– Fehlerbehandlung 220
– in Dateien schreiben 218, 221
– mit Datei-Handle verbinden 219
– Modi zum Öffnen 220
– neu erzeugen 220, 377
– öffnen 219
– Pfadangaben in Dateinamen 219
– Positionsmarker 326
– schließen 221
– sperren 326
– Textdateien 225
– über die Kommandozeile einlesen 207, 218
– überschreiben 220, 377
– vollständig lesen 224
– zeilenweise lesen 224
Dateiendezeichen 209
Datei-Handles 57, 206, 219
Datenbanken 295
Datentypen 42, 63
– Codierung im Speicher 67, 68
– Datei-Handles 206, 219
– Ganzzahlen 67
– Gleitkommazahlen 68
– in C++ 63
– Referenzen 183
– Strings 68
– Typkonvertierungen 81
Debugger 357
– Ablauf einer Debug-Sitzung 357
– beenden 358
– Haltepunkte 358

– Programm schrittweise ausführen 358
– starten 358
– Variablen auswerten 358
defined 54, 131, 146, 172
Dekrement 83, 88
delete 145
Dereferenzierung 186
Diagramme 283
die 220
Division 77, 81
Dokumentation 24, 353
– Activeperl 354
– Bücher 356
– FAQs 355
– Newsgroups 356
– perldoc 353
– perlfunc 353
do-until 110
do-while 108

E
Ein- und Ausgabe 57, 205
– $ 212
– $" 213
– $, 212
– $. 211
– $/ 209
– $| 213
– Arrays 129
– aus Dateien lesen 223
– Ausgabe auf die Konsole 58
– Ausgabe in Datei umlenken 149
– chomp 58
– Dateien 218
– Dateien über die Kommandozeile einlesen 207, 218
– Datei-Handles 206
– Dateiinhalt als Standardeingabe 141
– Daten über die Tastatur einlesen 207, 210
– Eingabe über die Tastatur 57
– Eingabeoperator 58, 197, 206, 223
– format 214
– Formatstrings 60
– in Dateien schreiben 218
– Kommandozeilenargumente 227
– Neue-Zeile-Zeichen entfernen 58
– print 59, 211
– print mit Zeilenumbruch 212
– printf 60
– Pufferung 213
– Report-Generator 213
– STDERR 213
– STDIN 57, 207
– STDOUT 57
– Strings ausgeben 88
– Trennung von Array-Elementen 212, 213
– Umgebungsvariablen 227
– Umleitung 141, 149
– write 213
– Zeilennummer 211
Eingabeoperator 58, 197, 206, 223
– $/ 209
– Dateien über die Kommandozeile einlesen 207, 218
– Datei-Handles 206
– Kontext 209
– Zeilentrennzeichen 209
else 99
elseif 100
Endlosschleifen 116
English 201
eq (Vergleich) 102
Escape-Operator 45, 231
Escape-Sequenzen 46
exists 146
Expansion 47
Exponent 77
Exponentialschreibweise 47

F
Fakultät 169
FAQs 355
Fenster 266
Fensterbasierte Programme 263
– after 274
– Bitmaps 270
– Eingabefelder 279
– Eingabemasken 277
– Ereignisbehandlung 268
– Fenster 266
– Frame-Fenster 278
– Hauptfenster 265
– Listenfelder 279
– MainLoop 268
– MainWindow 266
– pack 278
– periodische Ausführung von Code 274

– Schalter 268
– Steuerelemente 267
– Steuerelemente konfigurieren 268, 271
– Textfelder (Labels) 279
– Zeitgeber 274
for 110
foreach 112, 127
format 214
Formatstrings 60
Funktionen 153
– Argumente 161
– aufrufen 158
– bless 258
– Callback-Funktionen 269
– chomp 91
– chop 91
– chr 90
– close 221
– defined 146, 172
– definieren 157
– delete 145
– die 220
– exists 146
– format 214
– grep 137
– index 90
– join 142
– keys 147
– Kontext 197, 198
– kontextsensitive Funktionen implementieren 199
– lc 89
– lcfirst 89
– length 88
– Listen als Argumente 162
– localtime 197
– local-Variablen 174
– lokale Variablen 164
– map 127
– mathematische 78
– open 219
– ord 90
– pack 91, 225
– Parameter 158
– pop 130
– print 211, 212
– push 130
– rand 129
– read 224
– readline 224
– ref 185
– return 169
– reverse 89
– rindex 90
– Rückgabewerte 168
– scalar 199
– seek 326
– shift 130, 168
– sort 134
– splice 131
– split 139, 210, 223
– sprintf 88
– srand 130
– String-Funktionen 88
– substr 90
– trigonometrische 79
– uc 89
– ucfirst 89
– undef 145, 172
– unpack 91, 225
– unshift 130
– values 148
– Vergleichsfunktionen 135
– Vorteile 154
– vorzeitig beenden 170
– wantarray 199
– Werte entgegennehmen 158
– write 213
Furby 124

G

Ganzzahlen 67
Gästebucher 322
ge (Vergleich) 102
Gleitkommazahlen 68, 80
– Genauigkeit 81
– signifikante Ziffern 82
Grafik
– Bitmaps 270
– Canvas-Objekt 294
– Diagramme 283
– Messdaten grafisch aufbereiten 289
Grafische Oberflächen 263
grep 137
Groß- und Kleinschreibung 50
gt (Vergleich) 102

H

Hashes 143
– als Array-Elemente 189

– als Funktionsargumente 162
– Array der Schlüssel zurückliefern lassen 147
– Array der Werte zurückliefern lassen 148
– auf Elemente zugreifen 144
– definieren 143
– durchlaufen 146
– Elemente hinzufügen 145
– Elemente löschen 145
– Indizes 144
– leeren 146
– mit Listen 189
– Schlüssel/Wert-Paare 143
– Schlüsselnamen dynamisch erzeugen 288
HERE-Texte 86
Herunterladen
– Beispielskripte 387
– Perl 16
– Perl-Module 350
Hexadezimalzahlen 48
Hilfe 353

I
if 98, 114
index 90
Indizierung
– für Arrays 126
– Hashes 144
Inkrement 83, 88
Installation
– testen 23
– unter Unix/Linux 20
– unter Windows 23
Instanzbildung 252, 258
Instanzen 252
Instanzvariablen 251, 257
integer 81
Integer-Division 81
Internet 303
Interpreter 30
– Arbeitsweise 32
– Bytecode 33
– Optionen in Shebang-Zeile 26
– Syntaxüberprüfung 29
– Vor- und Nachteile 34
– Warnungen ausgeben 30

J
join 142

K
keys 147
Klammern 93
– eckige 188, 236
– geschweifte 99, 144, 186, 189, 238
– runde 122, 236, 242
Klassen 250, 256
Kommandozeilenargumente 227
Kommentare 26, 36
Konfiguration
– Webserver 307
Konkatenation 87
Konsole 24
Konstanten 44
– konstante Variablen 48
– String-Konstanten 44
– Zahlenkonstanten 47
Konstruktor 256, 257
Kontext 196
– ändern 198
– Funktionen 199
– kontexterzeugende Elemente 197
– kontextsensitive Elemente 197
– Kontexttypen 196
– reguläre Ausdrücke 243
Kontrollstrukturen 97
– Abbruchbefehle 112
– do-until-Schleife 110
– do-while-Schleife 108
– Endlosschleifen 116
– foreach-Schleife 112
– for-Schleife 110
– if-Bedingung 98
– if-else-Verzweigung 99
– Mehrfachverzweigungen 105
– while-Schleife 107
Konvertierung
– Arrays in Strings 142
– integer 81
– Klein- und Großschreibung 89
– Strings in Zahlen 65
– Text in Arrays 139
– Zahlen in Strings 65

L
last 113
lc 89
lcfirst 89
le (Vergleich) 102

length 88
Listen 122
– als Elemente von Hashes 189
– als Funktionsargumente 162
– als Listenelemente 124
– Bereichsoperator 123
– definieren 122
– qw-Operator 123
– Zuweisung an Skalare 123, 126
local 174
locale 241
localtime 197
Lokalisierung 241
Lösungen, zu den Übungen 359
lt (Vergleich) 102

M
m// 231
MainLoop 268
MainWindow 266
Manpages 353
map 127
Maschinenbefehle 56
Menüs 105
Messdaten, grafisch aufbereiten 289
Methoden 251, 254, 258
Module 177
– Bezeichner exportieren 179
– CGI 318
– Chart 283
– DBI 302
– eigene Module 177
– einbinden 178, 349
– Elemente verwenden 178
– Fcnt 326
– GD 283
– Tk 265
Modulo 77
Morse-Code 70
MSDOS-Eingabeaufforderung 24
Multiplikation 77
Muster 235
my 167, 174

N
Namensbereiche 173
ne (Vergleich) 102
Neue-Zeile-Zeichen 58, 209, 226
Newsgroups 356
next 113
not (Logik) 105

O
Objekte 250
Objektorientierte Programmierung 247
– bless 258
– in C++ 250
– in Perl 256
– Instanzbildung 252, 258
– Instanzen 252
– Instanzvariablen 251, 257
– Klassen 250, 256
– Konstruktor 256, 257
– Methoden 251, 254, 258
– Objekte 250
– Packages 256
– Schutzmechanismen 259
– Zugriff auf Klassenelemente 252
– Zugriffsspezifizierer 252
Oktalzahlen 48
OmniHttpd 308
open 219
Operatoren 35
– <=> 135
– Addition 77
– Adressoperator 185
– arithmetische 76
– Assoziativität 93
– Auswertungsreihenfolge 92
– Bedingungsoperator 114, 200
– Bereichsoperator 123
– cmp 135
– Dekrement 83, 88
– Division 77, 81
– Escape-Operator 231
– Exponent 77
– Inkrement 83, 88
– kombinierte Zuweisungsoperatoren 83
– Kontext 197, 198
– logische 105, 117
– Modulo 77
– Multiplikation 77
– Pattern-Matching-Operator 231, 239
– Strings 87
– Subtraktion 77
– Vergleichsoperatoren 101
– Vorzeichen 77
– Zuweisungsoperator 51, 76
or (Logik) 105
ord 90

P
pack 91, 225
Packages 173, 256
Palindrome 89
Parameter 158
– Arrays als Argumente 162
– Hashes als Argumente 162
– unveränderliche 168
Pattern Matching 229
– $1 242
– alternative Zeichen 235
– Anker 239
– globale Suche 232
– Groß- und Kleinschreibung 233
– Kontext 243
– Kürzungen 234
– m//-Operator 231
– Muster 235
– nach Wörtern suchen 230
– Optionen 232
– Quantifizierer 238
– Rückverweise 241
– s//-Operator 239
– Sonderzeichen 231
– suchen und ersetzen 239
– Textstellen weiter bearbeiten 243
– Zeichengruppen 237
– Zeichenwiederholungen 237
– zeilenorientierte Suche 233
Perl
– Bezugsquellen 16
– Binärversionen 16
– Bücher 356
– Datentypen 43
– Debugger 357
– Dokumentation 24, 353
– FAQs 355
– für Unix/Linux 17
– für Windows 17
– herunterladen 16
– Installation 20, 23
– Installationsverzeichnisse ermitteln 24
– Interpreter 30
– Manpages 353
– Module 349
– Newsgroups 356
– OOP 248, 256
– perldoc 353
– perlfunc 353
– Quellcodeversionen 18
– Version feststellen 24
– Webserver konfigurieren 307
perldoc 24, 353
perlfunc 353
Personal Web Server 307
POD 24
pop 130
Pragmas
– English 201
– locale 241
– strict 175
– vars 176
print 59, 211, 212
printf 60
Programme siehe Skripten
Programmfluss, steuern 97
push 130

Q
q (Quoting) 85
qq (Quoting) 86
Quicksort 133
Quoting 85
qw (Operator) 123

R
rand 129
read 224
readline 224
redo 113
ref 185
Referenzen 183
– als Funktionsargumente 163, 190
– Arrays dynamisch aufbauen 191
– Arrays von Arrays 188
– Arrays von Hashes 189
– auf Arrays 187
– auf Hashes 187
– auf referenzierte Variable zugreifen 186
– auf Skalare 187
– auf Variablen richten 185
– deklarieren 185
– dereferenzieren 186
– Einsatzgebiete 188
– Hashes mit Listen 189
– komplexe Datenstrukturen 188
– Referenzzähler 185
– umlenken 186

Reguläre Ausdrücke 229
– $1 242
– alternative Zeichen 235
– Anker 239
– globale Suche 232
– Groß- und Kleinschreibung 233
– Kontext 243
– Kürzungen 234
– m//-Operator 231
– Muster 235
– nach Wörtern suchen 230
– Optionen 232
– Quantifizierer 238
– Rückverweise 241
– s// 239
– Sonderzeichen 231
– suchen und ersetzen 239
– Textstellen weiter bearbeiten 243
– Zeichengruppen 237
– Zeichenwiederholungen 237
– zeilenorientierte Suche 233
Report-Generator 213
– Kopfteil 214
– Platzhalter 215
– Rumpfteil 214
– Seitenaufbau 214
– write 216
return 169
reverse 89
rindex 90
Rückgabewerte 168

S

s// 239
scalar 199
Schleifen
– Abbruchbefehle 112
– benannte Schleifen 114
– do-until 110
– do-while 108
– Endlosschleifen 116
– for 110
– foreach 112, 127
– while 107
Schlüsselwörter 35
seek 326
Seiteneffekte 94
Shebang-Zeile 27
shift 130, 168
Skalare 50, 64
Skripten 26
– anhalten 208
– Anweisungen 35, 55
– Ausdrücke 56
– ausführen 27
– Bezeichner 35
– CGI 307
– Ein- und Ausgabe 205
– Eingabemasken 277
– Kommandozeilenargumente 227
– Kommentare 26, 36
– Konstanten 44
– Kontrollstrukturen 97
– Lokalisierung 241
– mit grafischen Oberflächen 263
– Operatoren 35
– Schlüsselwörter 35
– Shebang-Zeile 26
– Stil 37
– Symbole 35
– Umgebungsvariablen 227
– Variablen 49
– Whitespace 36
– Zahlen 47
– Zeilen nummerieren 211
Skripterstellung 25
– Fehlerbehebung 28
– Optionen für den Interpreter 26
– Quelltext 26
– Skripte ausführen 27
– Syntaxüberprüfung 29
– Texteditor 25
sort 134
Sortieren 132
– BubbleSort 133
– Quicksort 133
– Vergleichsfunktionen 135
splice 131
split 139, 210, 223
sprintf 88
srand 130
Standardfehlerausgabe 213
Standardvariablen 201
– $ 203, 212
– $" 213
– $, 212
– $. 211
– $/ 203, 209
– $_ 115, 202, 208
– $| 213
– $1 242

– $a 203
– $ARG 203
– $b 203
– %ENV 203, 227
– @_ 160, 203
– @ARGV 227
– Kurz- und Langform 201
STDERR 213
STDIN 207
Steuerelemente 267
Stil 37
strict 175
Strings 68, 85
– Anführungszeichen 46
– ASCII-Code der Zeichen 90
– aus Arrays erstellen 142
– ausgeben 88
– Daten in Strings packen 91
– definieren 85
– durchsuchen 229
– Escape-Operator 45
– Escape-Sequenzen 46
– Funktionen 88
– HERE-Texte 86
– in Arrays verwandeln 139
– in Zahlen umwandeln 65
– Konkatenation 87
– Länge 88
– letztes Zeichen entfernen 91
– mit regulären Ausdrücken durchsuchen 229
– Operatoren 87
– Quoting 85
– String-Konstanten 44
– Teilstrings suchen 90
– umkehren 89, 91
– Umwandlung in Großbuchstaben 89
– Umwandlung in Kleinbuchstaben 89
– Variablenexpansion 47
– vergleichen 102
– Vergleichsoperatoren 102
sub 157
substr 90
Subtraktion 77
Suchen 137
– Binäre Suche 138
– in Arrays 137
– in Text 229
– reguläre Ausdrücke 229
Symbole 35
Syntaxüberprüfung 29

T
Tauschen 133
Text siehe Strings
Textdateien 225
Texteditor 25
Tk 265
– after 274
– Bitmaps 270
– Eingabefelder 279
– Eingabemasken 277
– Ereignisbehandlung 268
– Fenster 266
– Frame-Fenster 278
– Hauptfenster 265
– Listenfelder 279
– MainLoop 268
– MainWindow 266
– pack 278
– periodische Ausführung von Code 274
– Schalter 268
– Steuerelemente 267
– Steuerelemente konfigurieren 268, 271
– Textfelder (Labels) 279
– Zeitgeber 274

U
uc 89
ucfirst 89
Umgebungsvariablen 227, 313
Umlaute einfügen 59
Umleitung 141, 149
undef 54, 142, 145, 172
unpack 91, 225
unshift 130
until 110
use 173

V
values 148
Variablen 49
– $ 203, 212
– $" 213
– $, 212
– $. 211
– $/ 203, 209
– $_ 115, 128, 202, 208
– $| 213

– $1 242
– $a 135, 203
– $ARG 203
– $b 135, 203
– %ENV 203, 227
– @_ 160, 168, 203
– @ARGV 227
– Adresse zurückliefern 185
– Arrays 125
– Bezeichner 50
– einrichten 49
– globale 172
– Hashes 143
– initialisieren 50
– konstante 48
– Kontext 197
– local 174
– lokale 164, 174
– my 174
– Parameter 160
– Skalare 50
– Standardvariablen 201
– undefinierte 172
– Variablenexpansion in Strings 47
– Variablentypen 49
– Verdeckung 167
– Werte abfragen 53
– Werte löschen 54
– Werte zuweisen 51
– Zugriff auf verdeckte Variablen 167

Variablenexpansion 47
vars 176
Verdeckung 167
Vergleichsfunktionen 135
Vergleichsoperatoren 101
Verzweigungen
– if 98
– if-else 99
– mehrfache 105
Vorzeichen 77

W

wantarray 199
Warnungen 30
Webadressen 387
Webseiten, automatisch anfordern und speichern 303
Webserver
– Apache-Server 307
– OmniHttpd 308
– Perl-Unterstützung einrichten 307
– Personal Web Server 307
while 107
Whitespace 36
Windowing-Systeme 264
write 213

X

x (Wiederholung) 87
xor (Logik) 105

Z

Zahlen
– Binärzahlen 48
– Exponentialschreibweise 47
– Hexadezimalzahlen 48
– in Strings umwandeln 65
– Nachkommastellen 47
– Oktalzahlen 48
– Vergleichsoperatoren 102
– Zahlenkonstanten 47
– Zufallszahlen 129

Zeilentrennzeichen 209
Zeilenumbruch 226
Zufallszahlen 129
Zugriffsspezifizierer 252
Zuweisung
– kombinierte Zuweisungsoperatoren 83
– Zuweisungsoperator 51, 76

Zuweisungsoperator 51, 76

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.

Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks und zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Plazierung auf anderen Websites, der Veränderung und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.

Bei Fragen zu diesem Thema wenden Sie sich bitte an:

mailto:info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de) herunterladen