Einführung in die Perl-Programmierung

Oliver Corff, corff@zedat.fu-berlin.de November 1999

Inhaltsverzeichnis

1 Die Sprache Perl

Unter den vielen Programmiersprachen, die es gibt, ist Perl ein ausgesprochenes Unikum, das dem Wunschtraum von der eierlegenden Wollmilchsau recht nahe kommt.[1] Während viele Programmiersprachen einen hohen Grad von Spezialisierung für bestimmte Zwecke aufweisen, ist Perl mit dem ausdrücklichen Ziel entworfen worden, viele häufig anfallende Aufgaben aus den unterschiedlichsten Bereichen in einfacher, unkomplizierter, zwangloser und eleganter Weise zu lösen.

Perl wird gerne für komplexe Anwendungen eingesetzt, bei denen große Textdatenmengen verarbeitet werden müssen, etwa die automatierte Erzeugung von HTML-Dateien oder die Verwaltung von Benutzerdaten in einem großen Rechnernetzwerk. Aufgrund seiner Laufzeiteigenschaften wird man Perl wohl weniger für zeitkritische Anwendungen wie das Management eines Raketenstarts einsetzen; auch ist es nicht unbedingt für die 3D-Programmierung der neuesten Rennwagensimulation geeignet. Eine nahezu prototypische Anwendung ist die Verbindung von Datenbanken mit einer WWW-Schnittstelle mittels CGI, die überproportional häufig in Perl realisiert wird.[2] Perl ist daher möglicherweise unter den Programmiersprachen mit der größten Benutzerexposition die unbekannteste und steht damit in unmittelbarem Gegensatz zur vielleicht bekanntesten Programmiersprache mit allerdings relativ kleiner Benutzerexposition, nämlich BASIC.

Perl erlaubt und unterstützt eine Gliederung der Programme in wiederverwendbare Module, die von Programmierern weltweit ausgetauscht werden und aus einem öffentlich zugänglichen Archiv bezogen werden können.

1.1 Was ist Perl?

Perl[3] ist eine interpretierte Sprache, die in vielen Aspekten an C angelehnt ist, aber auch Elemente von sed, awk, sh, Pascal und anderen Sprachen integriert. Perl stellt leistungsstarke Funktionen für die Manipulation textähnlicher Daten dar, ist aber auch in der Lage, mit Binärdaten umzugehen. Aufgrund der Anlehnung an bekannte Sprachen ist Perl in seinen Grundzügen schnell erlernbar. Programme lassen sich mit Perl leicht

[1] Zumindest bei Schafen macht die Natur übertriebene Optimierungen nicht mit, oder, wie man in einem fernen Lande sagt: Ein Schaf kann sich entweder auf die Wollproduktion oder das Fleischwachstum konzentrieren, wenn es frißt, aber nicht auf beides gleichzeitig.

[2] Beispielprogramme für Datenbankzugang und WWW-basierende Benutzeroberflächen finden sich in Kapitel 7.1 (Benutzeroberflaechen).

[3] Die folgenden Ausführungen dieses Absatzes lehnen sich stellenweise an Tom Christiansens Perl-Kurs an (`CPAN:/doc/perl_slides.tex`. Siehe auch Kapitel 1.6.1 (über das Perl-Archiv)).

entwickeln, da Perl keinen aufwendigen Compilerlauf kennt, sondern einfach das Skript einliest und ausführt bzw. bei Fehlern im Skript *vor* Ausführung zurückweist.[4]

Perl bietet sich für Aufgaben an, die bisher mit der shell, awk oder sed gelöst wurden, und arbeitet schneller und effizienter als diese, da viele willkürliche Begrenzungen (Längen von Zeichenketten etc.) in Perl nicht existieren. Perl paßt sich dynamisch dem Datenvolumen an und ist in der Lage, ganze Dateien als einen String zu speichern, wenn der vorhandene Arbeitsspeicher dies zuläßt. Darüberhinaus sind viele mit Perl entwickelte Problemlösungen hochportabel, da es Perl für eine Reihe von Hardware-Plattformen und Betriebssystemen gibt. Ein Beispiel sind die `sgml-tools`, mit denen auch dieser Text verfaßt wurde.

1.2 Perl als Zwitter aus Interpreter und Compiler

Perl ist zwar eine interpretierende Sprache, überrascht aber trotzdem durch die außerordentliche Schnelligkeit des Programmablaufs. Dies resultiert aus dem Konzept Perls, ein Skript einzulesen und vor der Ausführung in Bytecode umzuwandeln, der erst dann ausgeführt wird, wenn das ganze Skript fehlerfrei verarbeitet worden ist. Dieses Verfahren hat mehrere Vorteile. Im Gegensatz zu shell-Programmen, mit deren Abarbeitung auch dann begonnen wird, wenn sie fehlerhaft sind, kommt ein syntaktisch falsches Perl-Programm nicht zur Ausführung. Das bewahrt allerdings nicht vor Schäden durch semantische Fehler! Darüberhinaus kann Perl ausführliche Fehlermeldungen ausgeben, die eine schnelle Einkreisung des Fehlers gestatten. Der compilierte Bytecode läuft mit einer Geschwindigkeit ab, die auch bei großen Datenmengen keinen Anlaß zur Kaffeepause gibt, da zwei der Hauptbremsen von klassischen Interpretern wegfallen: weder muß jede Zeile Quelltext vor jedem (eventuell wiederholten) Programmschritt neu eingelesen werden (Dateizugriffe brauchen immer Zeit, da hier das Betriebssystem mit seiner nicht bekannten Auslastung beansprucht wird), noch muß jede solchermaßen erneut eingelesene Zeile wiederholt in Bytecode umgewandelt werden.

Der von Perl erzeugte Bytecode eines Programms ist normalerweise nicht zugänglich, es existieren aber Programme und seit Version 5.005 Perl-Module, mit denen er abgespeichert und zwecks Ausführung wieder geladen werden kann.

1.3 Verfügbarkeit und Modularität

Da Perl frei für eine große Zahl von Betriebssystemen verfügbar ist, gibt es eine Hunderte von Musterprogrammen und Modulen, die von Programmierern weltweit beigetragen wurden und im Perl-Archiv (siehe 1.6.1 (CPAN)) verfügbar gehalten werden. Es lohnt sich daher immer, vor der Inangriffnahme größerer Projekte das Archiv zu durchsuchen. Mit hoher Wahrscheinlichkeit findet sich dort ein Skript oder ein Modul, das entweder das gestellte Problem genau löst oder aber die nötigen Werkzeuge für eine einfache und elegante Lösung bereitstellt. Die für Perl verfügbaren Module werden im Kapitel 4 (Module) vorgestellt.

1.4 Aktueller Stand

Die jeweils aktuelle Perl-Version wird im Archiv `stable.tar.gz` des Quellcode-Verzeichnisses von CPAN abgelegt. Ende November 1999 war der *stabile* Versionsstand 5.005_03; Entwicklerversionen, die aber noch

[4]Die Unterscheidung zwischen Programmier- und Skriptsprache ist im Falle Perls nicht einfach zu treffen, zumal die Grenze ohnehin nicht klar definiert ist. Rainer Fischbach vergleicht in seinem Artikel *Schwierige Abgrenzung. Von den Job Control Languages bis Perl und Python* (iX 12/1999, p. 60 ff.) ca. 17 verschiedene Skriptsprachen (zu denen er auch Perl zählt) und geht auch auf die Schwierigkeiten der Zuordnung ein.

Tobias Himstedt, Kristian Köhntopp, Frank Pilhofer, Holger Schwichtenberg, Henning Behme und Christian Kirsch benutzen in ihrem Artikel *Schnell, nicht schmutzig. Formulareintrag in verschiedenen Scriptsprachen überprüfen* (iX 12/1999, p. 72 ff.) die Sprachen JavaScript, Perl, PHP, Python, Tcl und VBScript, um ein WWW-Programm mit gleicher Funktionalität zu schreiben. Sie resümieren, daß die Entscheidung für eine Sprache oft eher auf Grund indivueller Vorkenntnisse als wegen des Designs falle.

nicht stabil sind, liegen bereits bei oder kurz vor 5.006.

1.5 Portierungen

Nach dem aktuellen Stand von CPAN und seiner Spiegel (siehe auch 1.6.1 (CPAN) und 1.6 (Bezugsquellen)) auf CD-ROM zu urteilen, gibt es Perl zumindest für die folgenden Architekturen und Betriebssysteme:

1. *UNIX-Varianten*: AIX, Altos, Apollo, A/UX, BSD/OS, ConvexOS, CX/UX, DC/OS, SINIX, DEC OS/F, DGUX, DYNIX, EP/IX, ESIX, FreeBSD, HP-UX 9, IRIX, Interactive Unix, Linux, LynxOS, MPE/IX, netbsd, NeXT, SCO, Solaris, SunOS, Ultrix, Unicos, etc.
2. *Portierungen auf andere Plattformen*: Acorn Archimedes (RISCOS), Amiga, AOS, AS/400, Atari, BeOS, Guardian, LynxOS, Mac, HP MPE/ix, MSDOS, IBM MVS (=OS/390), Netware, Plan 9, QNX, VMS, Stratus VOS, Windows 3.1, Windows NT und Windows 95.

1.5.1 Einschränkungen dieser Portierungen

Da es zwischen einzelnen Betriebssystemen große Unterschiede bei Bit-Ordnung, Dateisystemen, Funktionsaufrufen, Netzwerkfähigkeit, Prozeßkommunikation, Prozeßerzeugung, Speicherverwaltung und voreingestellten Variablengrößen gibt, sind nicht alle Funktionsmerkmale des Originals auf allen Plattformen anzutreffen.

Ohne jede Einschränkung kann Perl auf folgenden Plattformen kompiliert werden: Amiga, Plan 9, QNX, VMS und selbstverständlich modernen UNIX-Varianten. Der volle Funktionsumfang von Perl ist auf den nicht-UNIX-Plattformen nicht unbedingt verfügbar, und manche Perl-Portierungen werden von Bibliotheken mit plattform-spezifischer Unterstützung begleitet, so die Win32-Portierung ActivePerl mit der dazugehörigen Bibliothek.

Die Ausführungen und Beispiele in diesem Text beziehen sich immer stillschweigend auf Perl 5.005 auf Linux. Im November 1999 war auch die Perl-Version 5.004 noch weit verbreitet; noch ältere Versionen sollten allerdings gemieden werden.

1.6 Bezugsquellen

Perl steht unter dem Schutz der GNU General Public License sowie der GNU Artistic License und ist daher frei erhältlich. Die Weitergabe von modifiziertem Quellcode und die Weitergabe von Binärcode sind nur gestattet, wenn die Originalquellen mitgeliefert werden. Die genauen Regelungen sind in den Dateien `README` und `Artistic` im Wurzelverzeichnis des Perl-Quellcodes zu lesen.

1.6.1 Zentrales Perl-Archiv CPAN

Das zentrale Perl-Archiv im Internet heißt CPAN (Central Perl Archive Network, in Analogie zu CTAN, Central TeX Archive Network) und besteht aus einem Netzwerk von gespiegelten Servern mit einheitlicher Verzeichnisstruktur. Erreichbar ist es unter *http://www.perl.com/CPAN*. Beim Einloggen in das CPAN-Archiv wird automatisch ermittelt, welcher Server der nächstgelegene ist, und der Benutzer dorthin umgeleitet. Eine Liste verfügbarer Archiv-Server in geographischer Darstellung enthält die Datei `CPAN:/misc/xearth.names`, die in Verbindung mit dem Programm `xearth` benutzt werden kann:[5]

```
$ xearth -markerfile xearth.names &
```

Mit Stand vom 16. Juli 1998 befinden sich deutsche CPAN-Spiegel in:

[5] Where do you want to *perl* today?

- Sankt Augustin `ftp.gmd.de`
- Göttingen `ftp.gwdg.de`
- München `ftp.leo.org`
- München `ftp.mpi-sb.mpg.de`
- Bochum `ftp.rz.ruhr-uni-bochum.de`
- Erlangen `ftp.uni-erlangen.de`
- Hamburg `ftp.uni-hamburg.de`

Diese Liste ist bei ihrer Veröffentlichung mit hoher Wahrscheinlichkeit nicht mehr vollständig. Große Perl-Archive gibt es nämlich auf den ftp-Servern vieler Universitäten, so auch der FU Berlin[6], ohne daß dies nominell CPAN-Spiegel sind. Da jedoch sämtliche Module in aktuellen Versionen vorliegen, bestehen zumindest für die praktische Arbeit keine großen Einschränkungen.

Eine Übersicht über weltweite CPAN-Spiegel ist auch in der manpage `perlmodlib` enthalten.

1.6.2 CPAN auf CD-ROM

Ähnlich wie von CTAN gibt es auch von CPAN regelmäßige Abzüge auf CD-ROM. Ein bekanntes Produkt ist die im Buchhandel erhältliche CD-ROM *Perl*, die jährlich von Walnut Creek CDROM herausgegeben wird. Nachteilig an diesem Publikationsrhythmus ist natürlich, daß bestimmte, neueste Module möglicherweise nicht darauf enthalten sind.

1.6.3 Perl im Bündel mit anderer Software

Neben der quasi-kanonischen CD-ROM-Fassung mit dem CPAN-Inhalt wird Perl als fertig kompiliertes und eingerichtetes System mit einer großen Auswahl von Modulen allen Linux-Distributionen beigelegt. Wer sich Linux auf seinem Rechner einrichtet, kann davon ausgehen, daß Perl installiert ist. Ein einfacher Test besteht darin, am Kommandoprompt `perl -v` zu tippen. Die folgende Ausgabe sollte erscheinen:

```
$ perl -v

This is perl, version 5.005_03 built for i686-linux

Copyright 1987-1999, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5.0 source kit.
```

Es kann im Regelfall davon ausgegangen werden, daß die diesen Distributionen beiliegende Perl-Installation vollständig im Sinne des Perl-Quellfiles auf CPAN ist. Dem Autor dieser Zeilen ist allerdings bekannt, daß die Perl-Version, die dem Microsoft NT Servicepack beiliegt, nur aus wenigen elementaren Komponenten besteht.

1.7 Informationsquellen zu Perl

Das Informationsangebot zu Perl ist sehr vielfältig und kann in die Systemdokumentation (on-line) und die systembegleitende Dokumentation eingeteilt werden. Sowohl die elektronische als auch die Dokumentation in Buchform sind von außerordentlich hoher Qualität, so daß im Regelfall keine Frage unbeantwortet bleibt.

[6] Die Adresse ist *ftp.fu-berlin.de:unix/languages/perl.*

1.7.1 On-line-Dokumentation I (manpages)

Bestandteil der Perl-Distribution 5.005 sind etliche Manualseiten (manpages), die insgesamt (als manpage formatiert) rund 700 bis 800 Buchseiten Text enthalten:

- `perl`: Übersicht
- `perldelta`: Unterschiede zu früher
- `perl5004delta`: (neu in 5.005) Unterschiede zur letzten Version
- `perlfaq[1-9]`: Häufig gestellte Fragen; Gliederung:
  - `perlfaq1`: Einführung zu Perl
  - `perlfaq2`: Plattformen und Bezugsquellen für Perl
  - `perlfaq3`: Wie kann ich ...?
  - `perlfaq4`: Datenmanipulation
  - `perlfaq5`: Dateien und Ausgabeformate
  - `perlfaq6`: Reguläre Ausdrücke
  - `perlfaq7`: Perl-idiomatischer Sprachgebrauch
  - `perlfaq8`: Perl und das Betriebssystem
  - `perlfaq9`: Perl und Netzwerke
- `perltoc`: (neu in 5.005) Globales Stichwortverzeichnis, allerdings stark unvollständig
- `perldata`: Datenstrukturen
- `perlsyn`: Syntax
- `perlop`: Operatoren und Vorrang
- `perlre`: Reguläre Ausdrücke
- `perlrun`: Ausführung, Kommandozeilen-Optionen
- `perlfunc`: Eingebaute Funktionen
- `perlopentut`: (neu in 5.005) `open()`-Tutorium
- `perlvar`: Vordefinierte Variablen
- `perlsub`: Subroutinen
- `perlmod`: Module, Einführung
- `perlmodlib`: Module, Übersicht, Anwendung und Selbsterstellung
- `perlmodlibinstall`: Module, Installation von CPAN
- `perlform`: Formate
- `perllocale`: Sprachanpassung
- `perlref`: Bezüge auf Daten
- `perlreftut`: (neu in 5.005) Tutorium für Bezüge auf Daten
- `perldsc`: Einführung zu Datenstrukturen

- `perllol`: Listen von Listen
- `perltoot`: OO-Tutorium
- `perlobj`: Objekte
- `perltie`: Versteckte Objekte
- `perlbot`: OO-Tricks und Beispiele
- `perlipc`: Prozeßkommunikation
- `perlthrtut`: (neu in 5.005) Tutorium über Threads
- `perldebug`: Perl-Debugger
- `perldiag`: Fehlermeldungen
- `perlsec`: Sicherheitsaspekte
- `perltrap`: Fallstricke
- `perlport`: (neu in 5.005) Portierungsanleitung
- `perlstyle`: Stilistische Anleitung
- `perlpod`: "Plain Old Documentation"
- `perlbook`: Bücher zu Perl
- `perlembed`: Perl in C-Anwendungen
- `perlapio`: Perls interne abstrahierte IO-Schnittstelle
- `perlxs`: Perls XS-Schnittstelle
- `perlxstut`: Perls XS-Tutorium
- `perlguts`: Perl interne Funktionen
- `perlcall`: C-Aufrufe von Perl
- `perlhist`: (neu in 5.005) Perls Geschichte, mit Tabellen über Versionsstände, Dateigrößen, etc.

Die Übersicht (`man perl`) empfiehlt, sich durch diesen Berg Information in angegebener Reihenfolge durchzuarbeiten. Das klingt, als ob zum erfolgreichen Arbeiten mit Perl die Lektüre all dieser manpages nötig sei. Dem ist nicht so; wenn man ein grundlegendes Verständnis für die Syntax von Perl erworben hat (z. B. nach Studium der `perlsyn`-manpage und einigen Beispielprogrammen), dann wird vermutlich `perlfunc` die wichtigste manpage für die tägliche Arbeit sein. In ihr sind alle eingebauten Funktionen von Perl dokumentiert; neben der allgemeinen Syntax gibt es nahezu immer ausführliche Beispiele und Verweise auf andere Funktionen mit verwandten Zwecken. Andere, auch häufig konsultierte manpages dürften `perlre` (Reguläre Ausdrücke), `perlrun` (Übersicht über die Kommandozeilenoptionen) und `perlform` (Deklaration von Ausgabeformaten) sein. Der Inhalt der anderen manpages überschreitet die Grenzen einer Einführung zu Perl deutlich und setzt teilweise längere Einarbeitung voraus.

Alle manpages gibt es auch bei CPAN als riesige, zusammenhängende Textdatei mit ca. 111.012 Zeilen Länge: `CPAN:/doc/manual/text/PerlDoc-5.005_02.txt.gz`, darüberhinaus auch als HTML-, PostScript-, gnuinfo- und texinfo-Datei.

1.7.2 On-line-Dokumentation II (pod)

Die Perl-Dokumentation liegt auch im pod-Format vor. Die Abkürzung steht für "Plain Old Documentation" und bezeichnet eine einfache Markup-Sprache, die die Speicherung von Programmcode und Dokumentation in einer Datei ermöglicht. Die manpage `perlpod` enthält weitergehende Informationen. Viele Module werden mit pod-Dokumentation ausgeliefert; auf jeder funktionierenden Perl-Installation finden sich die Kommandos `pod2html`, `pod2latex` und `pod2man`, mit der sich pod-Dokumente in HTML-Texte, LaTeX-Dateien oder manpages umwandeln lassen. `CPAN:/doc/pod2x/` bietet außerdem noch `pod2fm`, `pod2texinfo` und `pod2text` an, mit denen sich FrameMaker-, texinfo- und ASCII-Versionen der pod-Dokumente erstellen lassen. Alle erwähnten Programme sind selbstverständlich in Perl geschrieben.

1.7.3 Perls FAQ-Familie

Das eigentliche Perl-FAQ (Frequently A(sk|nswer)ed Questions), das zuletzt 1997 überarbeitet wurde, gibt es im USENET (siehe unten), bei CPAN (doc/FAQs/FAQ/PerlFAQ.html), oder aber in neun Teilen als `perlfaq[1-9]`-manpage. Darüberhinaus gibt es zu einer Reihe von bekannten Fragestellungen weitere themenbezogene FAQs, etwa zur CGI-Programmierung, zum verschiedenen Portierungen von Perl (DOS, Mac, Windows in verschiedenen Variationen); alle diese Texte finden sich in `CPAN:/doc/FAQs/`. Die manpage `perlfaq2` gibt weitere Hinweise auf Dokumentation in der CPAN-Hierarchie.

1.7.4 USENET: comp.lang.perl

Die klassische USENET-Nachrichtengruppe für Perl ist `comp.lang.perl`. `c.l.p.` ist seit Juli 1998 moderiert (also `c.l.p.moderated`) und wurde (siehe die Charta der Gruppe und Begründung des Moderationsantrags) mit dem Ziel geschaffen,

> [...], dem unerfahrenen Perl-Programmierer eine nützliche Informationsquelle zu beschaffen und gleichzeitig erfahrene Programmierer zu gewinnen und zu halten, die wertvolle Beiträge zu dieser Gruppe leisten können.

Darüberhinaus gibt es die Gruppen

- `comp.lang.perl.announce` für Ankündigungen,
- `comp.lang.perl.misc` für „Verschiedenes", also auch Fragen der Art „Wie mache ich ... in Perl". Wer zum ersten Mal hier eine Nachricht absendet, bekommt automatisch eine Übersicht mit aktueller Perl-Dokumentation zugeschickt;
- `comp.lang.perl.modules` für Module und
- `comp.lang.perl.tk` für GUIs auf der Basis von Tk/Tcl (siehe auch Kapitel 7.1 (Perl und Tk)); außerdem noch
- `comp.infosystems.www.authoring.cgi` für CGI-spezifische Fragen (siehe dazu auch Kapitel 6.6 (CGI-Programme)), jedoch mit deutlichem Schwerpunkt auf Perl.

In anderen Sprachen als Englisch erscheinen außerdem folgende Gruppen:

- `de.comp.lang.perl` deutsch,
- `fj.comp.lang.perl` japanisch,
- `fr.comp.lang.perl` französisch,

- `han.comp.lang.perl` koreanisch,
- `it.comp.lang.perl` italienisch,
- `japan.comp.lang.perl` ebenfalls japanisch,
- `dk.edb.programmering.lang.perl` dänisch,
- `tw.bbs.comp.lang.perl` chinesisch (in Langzeichen),
- `pl.comp.lang.perl` polnisch und
- `relcom.comp.lang.perl` russisch;

möglicherweise gibt noch weitere Gruppen zum Thema Perl.

1.7.5 Kanonische Perl-Bücher

Es gibt mittlerweile etliche Dutzend Bücher zur Perl-Programmierung (siehe Kapitel 8 (Literaturverzeichnis)), aber nur drei Texte, die in der Perl-Gemeinschaft als kanonische Texte anerkannt sind. Dies sind "Programming Perl" (von Larry Wall, dem Schöpfer von Perl, sowie Tom Christiansen und Randal L. Schwartz), "Learning Perl" von Randal L. Schwartz, und "The Perl Cookbook", von Tom Christiansen und Nathan Torkington.

Programming Perl, Second Edition Mit dem Erscheinen von Perl 5.000 wurde eine neue Ausgabe von "Programming Perl" aufgelegt. Das Buch stellt die umfassendste und vollständigste Referenz zur Sprache dar und wurde von Larry Wall mitverfaßt. Die `perlfunc`-manpage ist weitgehend deckungsgleich mit dem Buchabschnitt, der die Perl-Funktionen enthält. Das Buch ist für die Arbeit unentbehrlich, denn es enthält nicht nur die vollständige Befehlsreferenz und eine Liste der wichtigsten Perl-Module, sondern auch zu bestimmten Themen (etwa geschachtelten Datenstrukturen) detaillierte Einführungen.

Wegen des Kamels auf dem Buchumschlag ist es auch als Kamel-Buch bekannt. Um es von seinem Vorgänger (für Perl 4) abzuheben, wird es gelegentlich als Blaues Kamel-Buch bezeichnet. Der Vorgänger war rot.

Learning Perl, Second Edition Das Einführungsbuch zu Perl stellt die wichtigsten Konstrukte von Perl vor, ohne den Anfänger mit der Detailfülle der Standardreferenz zu verschrecken. Dieses Buch wird in den Nachrichtengruppen zu Perl oft einfach als Lama-Buch bezeichnet.

The Perl Cookbook Das Kochbuch für Perl ist für die praktische Arbeit hervorragend geeignet, da es für eine große Zahl wiederkehrender Probleme die möglichen Lösungsansätze diskutiert und Musterlösungen in Perl vorstellt. So enthält es zahlreiche Beispiele zur Zeichenkettenbehandlung, zum Umgang mit Regulären Ausdrücken, zur Datums- und Uhrzeitverarbeitung, darüberhinaus eigene Kapitel, die den wichtigsten Datentypen und Anwendungsgebieten von Perl gewidmet sind.

1.7.6 Andere Perl-Bücher

Die Unterscheidung in kanonische und nicht-kanonische Texte bezieht sich nicht auf die Qualität weiterführender Literatur, sondern auf die enge Verbindung zwischen der Entstehung von Perl und der dazugehörigen kanonischen Dokumentation, die teilweise einer einzigen Quelle entspringen. Andere Bücher müssen deshalb nicht qualitativ schlechter sein.

An dieser Stelle werden noch ein paar weitere Titel vorgestellt:

Perl in a Nutshell Von Ellen Siever, Stephan Spainhour und Nathan Patwardhan. O'Reilly, 1999. Ca. 656 Seiten.

Das Nußschalenbuch ist als Einführung zu einer ganzen Reihe Perl-typischer Anwendungen gut geeignet, da es ganze Abschnitte der Win32-Unterstützung von Perl, der CGI-Programmierung oder der graphischen Benutzerschnittstelle widmet.

Wer Perl lernt, weil er eigentlich nur ein CGI-Programm oder ein Demonstrationsprogramm mit graphischer Benutzeroberfläche schreiben möchte, ohne in die Tiefen von X oder dem Windows-API vorzudringen, kann durchaus mit diesem Buch alleine glücklich werden, da es auch eine kompakte Einführung von Bezug über Installation von Perl sowie den Grundlagen der Sprache enthält.

Advanced Perl Programming Von Sriram Srinivasan. O'Reilly, 1997. Ca. 404 Seiten.

Dieses Buch ist definitiv nicht für Anfänger geeignet, die sich eine erste Orientierung über Perl verschaffen wollen. Vielmehr führen mehrere seiner Kapitel Themen fort, die in den anderen Büchern nur einführend behandelt werden oder gar nicht erwähnt werden. Programmierer, die Perl selbst erweitern möchten, kommen um dieses Buch nicht herum.

Algorithms with Perl Von Jon Orwant, Jarkko Hietaniemi und John Macdonald. O'Reilly, 1999. Ca. 684 Seiten.

Dieses Buch ist kein Perl-Buch im strengen Sinne. Es ist eher ein kurzgefaßtes Informatik-Kompendium mit Perl als symbolischer Sprache. Viele Bücher über Informatik und Programmierung verwenden eine programmiersprachenähnliche Metasprache, die allerdings in wirklichen Computern nicht lauffähig ist. Ähnlich der pädagogischen Absicht, mit der N. Wirth PASCAL schrieb, verwenden die Autoren Perl als Grundlage, mit dem großen Vorteil, daß alle Beispiele sofort ohne weitere Umsetzung (und die damit verbundenen Probleme) ausführbar sind.

Dieses Buch wird trotz seines von Perl abweichenden Themenschwerpunktes allen empfohlen, die Perl als leistungsfähige Sprache für viele Anwendungen entdeckt haben, da die Beispiele helfen, Perl-idiomatische Beispiele von teilweise verblüffender Einfachheit zu finden.

1.7.7 Zeitschriftenartikel

Die deutsche Fachzeitschrift iX veröffentlich regelmäßig Beiträge zur Perl-Programmierung, die oft von namhaften Autoren verfaßt werden, die teilweise auch als Buchautoren zur Perl-Programmierung bekannt sind.

2 Grundlegende Elemente der Sprache

Perl ist eine Sprache, deren einzelne Funktionselemente oft anderen Sprachen entlehnt sind, sofern eine bestimmte Funktionalität in jener Sprache in vorbildlicher Weise bereitgestellt wird. Daher werden viele Programmierer in Perl vertraute Konstrukte vorfinden. Außerdem hat Perl auch seine eigenen, höchst leistungsfähigen Konstrukte und Datenstrukturen. In diesem Kapitel wird daher eine Übersicht über die syntaktischen Konventionen, die Datentypen, die wichtigsten Funktionen und Operatoren sowie eingebaute Variablen gegeben.

2.1 Syntaktische Konventionen von Perl-Programmen

Im Gegensatz zu Pascal-Programmen, die einen `BEGIN/END.`-Block benötigen, oder C-Programmen, die immer eine Funktion namens `main()` aufweisen müssen, werden Perl-Programme in der Regel in der Reihenfolge ihres Quelltextes abgearbeitet. Das einfachste Perl-„Programm” ist leer und bereitet dennoch der Laufzeitumgebung keine Probleme. Das kleinste Perl-Programm, das wirklich etwas bewirkt, sieht so aus:

```
print "Hallo, Welt!\n"; # Gibt einen bekannten Gruß aus
```

Hieran können wir bereits sehen, daß die Argumente einer Funktion nachgestellt werden, daß wörtlicher Text in doppelte Hochkommata eingeschlossen ist (was aber nicht zwingend ist, wie vieles in Perl nicht zwingend ist. Siehe Kapitel 2.8 (Wörtlicher Text und Begrenzer). Zwingend scheinen nur Kommentare wie diese zu sein.), daß Statements mit Semikola abgeschlossen werden[7] , und daß Kommentare zum Quelltext mit # eingeleitet werden. Kommentare dürfen sich übrigens nicht über mehrere Zeilen erstrecken; sollte ein mehrzeiliger Kommentar benötigt werden, so ist jede Kommentarzeile neu mit # einzuleiten. Außerdem muß dem Ausgabebefehl ausdrücklich mitgeteilt werden, daß die Textzeile mit einem Zeilenumbruch (\n) endet.

2.2 Ausführung von Perl-Programmen

Je nach Betriebssystem unterscheiden sich die Ausführungsmöglichkeiten für Perl-Skripte. In UNIX-ähnlichen Betriebssystemen gibt es fast immer die Möglichkeit, die Skripte mit einer Kommentarzeile zu beginnen, die den Pfad zum Perl-Interpreter enthält. Wenn das Perl-Skript als ausführbar gekennzeichnet ist, kann es direkt aufgerufen werden. Sollte dies nicht möglich sein, so kann auch Perl aufgerufen und das Skript als Argument übergeben werden.

2.2.1 Ausführung auf UNIX-ähnlichen Systemen

Unser erstes Beispiel 2.1 (Hallo, Welt!) läßt sich auf UNIX-typische Weise so schreiben und ausführen:

```
$ chmod 755 hallo
$ cat hallo
#!/usr/bin/perl
print "Hallo, Welt!\n"; # Gibt einen bekannten Gruß aus
$./hallo
Hallo, Welt!
$
```

In den meisten der folgenden Beispiele wird die erste Skriptzeile mit dem Interpreter-Aufruf in der Regel weggelassen.

2.2.2 Ausführung auf DOS-basierenden Systemen

Für DOS- und OS/2-basierende Systeme gibt es die Möglichkeit, Batch-Dateien zu schreiben (etwa namens **HALLO.BAT**), die für den Kommando-Interpreter (etwa **COMMAND.COM**) den Aufruf des Perl-Interpreters und den Skripttext enthalten können. Genaue Angaben zum Ablauf von Perl-Programmen finden sich in der `perlrun`-manpage.

2.2.3 Ausführung auf Windows-basierenden Systemen

Genaueres über Installation und Einsatz von Perl auf Windows-basierenden Systemen findet sich unter *http://www.activestate.com/ActivePerl*.

[7]Zumindest *fast* immer; nur das letzte Statement eines Blocks (s. 2.5 (BLOCK)) ist von dieser Regel ausgenommen.

2.2.4 Ausführung auf Mac-basierenden Systemen

Genaueres über Installation und Einsatz von Perl auf Mac-basierenden Systemen findet sich unter *http://www.macperl.com/*.

2.3 Variablentypen

Eines der herausragenden Merkmale von Perl ist die schwache Typisierung der Sprache. Es gibt in der praktischen Arbeit eigentlich nur zwei wirklich voneinander verschiedene Variablen- und Datentypen, die mit einfachen Operationen ineinander überführt werden können. Perl beschwert sich auch nur dann beim Benutzer über einen falschen Datentyp, wenn im gegebenen Kontext wirklich keinerlei Möglichkeit besteht, die Variable mit dem fraglichen Datum zu evaluieren.

Nach Konvention und Namensraum werden allerdings nicht zwei, sondern drei wichtige Daten- und damit Variablentypen unterschieden, nämlich Skalare, Arrays oder Listen, und namentlich referenzierte Arrays, auch Hashes genannt. Neben diesen klassischen Variablen, die mit Operatoren zu Ausdrücken verknüpft die Argumente von Funktionen bilden, gibt es auch Bezeichner für Datei-Handles, Formatvorlagen und andere Dinge, aber diese Bezeichner werden im Regelfall nicht als Operanden, sondern als Argumente eingesetzt.

Grundsätzlich gilt für Variablen in Perl, daß sie nicht *deklariert* werden müssen. Ihr erstmaliges Erscheinen im Programmtext führt lediglich dazu, daß ihr Wert auf die spezielle Größe `undef` gesetzt wird. Wenn versucht wird, den Wert einer solchermaßen nicht definierten Variablen auszulesen, wird nach außen hin die passende Leergröße, also beispielsweise eine leere Zeichenkette, ausgegeben.

2.3.1 Skalare

In Perl sind die sogenannten Skalare Platzhalter, die sowohl Zeichenketten als auch numerische Werte enthalten können. Es entfällt damit die Unterscheidung „klassischer" Programmiersprachen nach INT, LONGINT, STRING, CHAR, REAL und wie die Typen in C und Pascal benannt werden.

Skalare fangen immer mit dem Typbezeichner `$` an, durch den ein eigener Namensraum definiert wird: `$kalar` ist eine skalare Variable mit dem Namen `kalar`. Da Variable in Perl weder deklariert noch initialisiert werden müssen und Skalare jederzeit Zeichenketten und numerische Werte enthalten können, ist es somit möglich, folgende Zeilen ohne Laufzeitfehler abzuarbeiten:

```
$a=4;         # $a ist numerisch
print $a;     # '4' wird ausgegeben
$a="vier";    # $a ist jetzt eine Zeichenkette
print $a;     # 'vier' wird ausgegeben
```

Variable werden immer im *Kontext* evaluiert (oder *interpoliert*, um im Perl-Jargon zu bleiben). Daher stellt sich die Frage, ob '4' als Zeichenkette oder als numerischer Wert behandelt wird, eigentlich nicht, da dies durch die Operatoren bestimmt werden kann, denen die skalare Variable als Operand vorgelegt werden kann. Ausführliche Beispiel folgen im Kapitel 2.6 (Operatoren).

2.3.2 Listen und Arrays

Die nächste Daten- und Variablengruppe, die Perl zu bieten hat, sind Listen, als Variable auch Array genannt.

Listen sind Sammlungen von Skalaren. Wenn diese Sammlungen in einer Variable gebündelt werden sollen, wird dazu ein Array benutzt. Arrays können also in ihren einzelnen Listenelementen Zeichenketten und numerische Werte in beliebiger Verteilung enthalten.

Arrays werden durch ein vorangestelltes `@` gekennzeichnet, etwa so: `@rray` ist die Listenvariable `rray`. Der Namensraum der Arrays ist von dem der Listen abgegrenzt, so daß zwischen `$worte` und `@worte` keine Konflikte auftreten.

Einzelne Elemente eines Arrays werden mit Subskriptnotation bezeichnet, deren Startwert Null ist. Es handelt sich hierbei um den *Versatz* in die Elementenliste hinein, nicht um die *Nummer* des Elements. `$liste[3]` bezeichnet daher das vierte Element von `@liste`. Es erscheint vielleicht die Dollarnotation merkwürdig, aber per Definition ist das einzelne Element eines Arrays ein Skalar, der daher entsprechend gekennzeichnet sein muß.

```
@a=(4,5,6,'a','b','c'); # @a entält drei numerische Werte und drei Zeichenketten
print @a;               # `456abc' wird ausgegeben, in Listenkontext
print @a[1,2];          # `56' wird ausgegeben, in Listenkontext
print $a[1];            # `5' wird ausgegeben, nun ein Skalar
```

2.3.3 Namentlich referenzierte Arrays (Hashes)

Einer der mächtigsten Datentypen Perls sind die namentlich referenzierten Arrays oder Hashes (im Perl-Jargon). Ähnlich wie Arrays enthalten Hashes Sammlungen von Skalaren. Anders als bei Arrays erfolgt der Zugriff auf den einzelnen Wert jedoch nicht über eine Indexnummer, sondern über einen namentlich vergebenen Schlüssel. Perls Hashes sind daher so etwas wie rudimentäre Datenbanken in Variablenform.

Perls namentlich referenzierte Array-Bezeichner werden durch ein vorangestelltes `%` markiert. Auch hierdurch wird ein eigenständiger Namensraum erschlossen, so daß es keine Konflikte mit gleich buchstabierten Variablenbezeichnern anderen Typs gibt.

```
%telno=('Otto'          => 4711,
        'Schorschi'     => 123456,
        'Jupp'          => 654321
        );

print $telno{'Otto'};           # Druckt '4711'

print $telno{'Heinz'};          # Druckt nichts, da 'Heinz' nicht definiert

$telno{'Heinz'}='0800-HEINZ';   # Zufügen eines
                                # neuen Elementes

print $telno{'Heinz'};          # Druckt '0800-HEINZ'
```

2.3.4 Datei-Handles

Die nächste, häufig gebrauchte Variablenart sind Datei-Handles. Aus Konvention werden sie immer mit Großbuchstaben geschrieben, also z. B. `MYLISTING`, aber dies ist nicht zwingend notwendig.

Datei-Handles werden mit der Funktion `open` erzeugt. Einfache Beispiele der häufigsten Anwendungen folgen hier, interessehalber sei jedoch auf den vollen Funktionsumfang verwiesen, der in der manpage `perlfunc` unter dem Eintrag `open` zu finden ist.

```
open(MYLISTING,'mylist.txt');   # Bindet Datei mylist.txt an Handle MYLISTING
                                # und öffnet die Datei nur zum Lesen
```

```
$line=<MYLISTING>;              # Liest aus mylist.txt eine Zeile ein

open(MYRESULT,'>myresult.txt'); # Bindet Datei myresult.txt an Handle MYRESULT
                                # und öffnet die Datei neu zum Schreiben.
print MYRESULT $line;           # Eine eventuell vorhandene Datei gleichen
                                # Namens wird mit dem Öffnen überschrieben.

open(MYLOG,'>>mylog.txt');      # Bindet Datei mylog.txt an Handle MYLOG
print MYLOG $line;              # und öffnet die Datei zum Schreiben, ohne
                                # sie auf Null zurückzusetzen.

open(DATABASE,'+<datenbank.db');# Öffnet eine Datei zum Lesen *und* Schreiben

open(MYLIST,'|sort >list.txt'); # Übergibt den Inhalt von MYLIST an
                                # Pipe mit Weiterleitung auf den
                                # sort-Befehl des Betriebssystems;
                                # die Ausgabe wird in der Datei
                                # 'list.txt' abgelegt.
```

Da Datei-Handles `STDIN`, `STDOUT` und `STDERR` sind vordefiniert und bezeichnen die Standardeingabe, die Standardausgabe und die Standardfehlerausgabe. Wird im Programmtext keine andere Datei angegeben, so beziehen sich Lese- und Schreibvorgänge immer auf die Standardein- und ausgabekanäle. Daher druckt das allererste Beispiel in diesem Text sein *Hallo, Welt* auch ohne weitere Befehle auf den Bildschirm.

2.4 Vordefinierte Variable

Perl hält einen großen Vorrat vordefinierter Variabler bereit, die teilweise explizit, teilweise auch implizit angesprochen werden können. Die wichtigste Variable ist `$_`, die den aktuellen Skalar enthält, der bei der letzten Leseoperation eingelesen wurde. Normalerweise wird aus einer Datei (nehmen wir hier `STDIN` an) mit dem Befehl `$skalar=<STDIN>` gelesen; der Verweis auf `$skalar` kann auch wegfallen, so daß die `while`-Schleife vieler Perl-Programme, mit der die Daten eingelesen werden (`while (<STDIN>) {}`), in Wirklichkeit `while ($_=<STDIN>) {}` heißt.

Alle Funktionsaufrufe, die normalerweise einen Skalar als Argument erwarten, weichen auf die Standardvariable `$_` aus, wenn nicht explizit ein anderer Skalar angegeben wird. Die impliziten Variablen entsprechen den elliptischen Bezugskonstruktionen, denen man auch in natürlichen Sprachen begegnet:

- Hast Du gestern den Film gesehen? - Nein, habe ich nicht.

Der Fragende weiß genau, daß sich die gegebene Antwort auf das Sehen des Films bezieht und nicht auf irgendwelche anderen Vorgänge und Tatsachen, etwa Pizza zu essen oder ein Buch zu kaufen.

Die vordefinierten Variablen erschweren am Anfang das Verständnis von Perl-Texten, wenn man rigide Definitionen erwartet. Perl ist in manchen Aspekten aber an natürliche Sprachen angelehnt und zeichnet sich durch ähnliche Abkürzungsmechanismen aus.

Es gibt nicht nur vordefinierte Skalare, sondern auch vordefinierte Arrays; mit dem einfachen Aufruf von `split ;` wird eine Zeile (in `$_` enthalten) an den Leerstellen getrennt und die vorgefundenen Elemente einem Array namens `@_` übergeben, mit dem man weiter operieren kann.

Wer sich einen genauen Überblick über die stattliche Familie der vordefinierten Variablen verschaffen möchte, kann Einzelheiten auf der manpage `perlvar` nachlesen.

Wer mit den vordefinierten und impliziten Variablen Probleme hat, kann an ihrer Stelle auch die Entsprechungen dieser Variablen in englischer Sprache verwenden. Dies muß allerdings Perl mitgeteilt werden, bevor dies möglich ist. Dies geschieht mit folgendem Aufruf in der Nähe des Programmbeginns:

```
use English;
#
while (<>) {
        print $ARG      # $ARG steht jetzt fuer $_
        }
```

Eine Liste aller in Klartext verfügbaren Variablen findet sich im Nußschalenbuch auf Seite 200 und in der manpage `perlvar`.

Tatsächlich haben wir soeben das erste Mal ein *Modul* aufgerufen. Der Erweiterbarkeit von Perl durch Module ist das Kapitel 4 (Module) gewidmet.

2.5 Flußkontrolle

Während zunächst die einfachste Flußkontrolle darin besteht, alle Statements eines Programmtextes hintereinander zu schreiben, um sie dann in dieser Reihenfolge ausführen zu lassen, ist dies natürlich für viele Fälle unzureichend, da weder Wiederholungen von Programmteilen noch bedingte Ausführung möglich wären. Perl bietet nicht nur klassische Mittel wie `if`-Konstrukte und `while`-Konstrukte für bedingte und wiederholte Ausführung, sondern gestattet darüberhinaus eine erstaunliche syntaktische Freiheit beim Schreiben des Programmtextes.

Übrigens kann bereits die einfachste Reihung aller Statements ohne *sichtbare* Flußkontrolle in Form etwa einer `while`-Schleife in Perl ein sinnvolles Programm darstellen, das auf den gesamten Eingabetext angewandt wird. Zu diesem Zweck muß Perl allerdings mit bestimmten Einstellungen (siehe Kapitel 3.1 (Kommandozeilen-Optionen)) aufgerufen werden.

Die wichtigsten Elemente für die Ablaufkontrolle werden nun angegeben, wobei zu bemerken ist, daß ein `BLOCK` unabhängig von der Zahl der enthaltenen Statements immer in geschweifte Klammern eingeschlossen sein muß. Die von C bekannte Methode, einzelne Statements als Block zuzulassen, gilt in Perl nicht. Auch einzelne Statements müssen immer in geschweifte Klammern gefaßt werden, wenn sie als Block auftreten sollen.

Hier nun die wichtigsten Schlüsselwörter:

if

prüft eine Bedingung und veranlasst gegebenenfalls die Ausführung eines Befehlsblocks.

Die allgemeine Form lautet:

```
if (Bedingung) BLOCK
```

Daher ist folgendes Beispiel korrekt:

```
if ($Alter>18) {print "Sie dürfen wählen!\n";}
```

Es ist allerdings auch möglich, die Reihenfolge umzustellen und die folgende Form zu benutzen:

```
Statement if (Bedingung)
```

Daher ist folgendes Beispiel funktionsfähig:

```
Abfragen if ($query->param('Knopf') eq 'Query');
```

Der `if`-Konstrukt kann um eine Alternative ergänzt werden. Dazu dient das Schlüsselwort `else`:

```
if (Bedingung) BLOCK
else BLOCK
```

Daher ist folgendes Beispiel korrekt:

```
if ($Alter>18) {print "Sie dürfen wählen!\n";}
else {print "Sie müssen Ihre Eltern fragen!\";}
```

while

bewirkt die wiederholte Ausführung eines Blocks bei erfolgreicher Bedingungsprüfung.

Die allgemeine Form lautet:

```
while (Bedingung) BLOCK
```

In typischer Weise werden mit `while` Schleifen gebildet, mit denen über ganze Dateien iteriert wird:

```
while (<STDIN>) {
        # Mache etwas mit den von STDIN nach $_ eingelesenen Variablen
        }
```

unless

bewirkt das Gegenteil von `if`. Sobald eine Bedingung nicht erfüllt ist, wird ein Block ausgeführt.

```
unless (Bedingung) BLOCK
```

```
unless ($Alter>18) {print "Sie dürfen nicht wählen."}
```

for

bewirkt eine beschränkte Wiederholung eines Blocks. Die allgemeine Form ist:

```
for ([Anweisungen und] Bedingung) BLOCK
```

Ein funktionierendes Beispiel wäre:

```
for ($i=0; $i<=10; $i++) {print "$i;"}
```

Hier ist besonders hervorzuheben, daß alle drei Anweisungen in den runden Klammern nach `for` optional (!) sind. Daher ist auch der folgende Ausdruck gültig:

```
for (;;) { Actions }
```

Mit `(;;)` wird eine endlose Schleife initiiert, aus der kontrolliert ausgestiegen werden muß.

foreach

arbeitet für jedes Element der angegebenen Liste einen Block ab (eine Liste ist eine Sammlung von Skalaren). Die allgemeine Form ist:

```
foreach SKALAR (LISTE) BLOCK
```

Als LISTE kann sowohl eine Liste stehen, die von einem Operator im Listenkontext erzeugt wird, als auch ein Array, das eine Liste enthält, als auch eine wörtliche Liste. Daher sind die folgenden Beispiele funktionsfähig:

```
@worte=(Am, Anfang, war, das, Wort);
#
# @worte ist ein Array von Skalaren und daher eine LISTE..
#
# $wort ist eine Laufvariable.
foreach $wort (@worte) {print "-- $wort --\n";}
#
# Die LISTE ergibt sich nun aus dem Kontext: sort gibt eine Liste aus.
#
foreach $wort (sort @worte) {print "-- $wort --\n";}
```

2.6 Operatoren

Perl unterscheidet in vertrauter Weise zwischen Operatoren und Funktionen. Im Grenzfall ist ein und dasselbe Objekt je nach Aufrufart ein Operator oder eine Funktion; die Entscheidung fällt mit der Anwendung und der Schreibweise. Wenn das Argument eines Befehls in Klammern steht, ist es eine Funktion. Außerdem akzeptiert eine Funktion als Argument sowohl Ausdrücke als auch Literale, wohingegen für die Verküpfung von wörtlichen Operanden und Operatoren bestimmte Einschränkungen gelten (s.u., unäre Operatoren). Operatoren werden unterschieden nach:

- Zahl der Operanden:
  1. Unäre Operatoren
  2. Binäre Operatoren
  3. Ternäre Operatoren
- Art des Operanden:
  - Operand ist eine Zahl
  - Operand ist eine Zeichenkette
- Art der Operation:
  - Arithmetik
  - Vergleiche
  - Zuweisungen
  - Mustervergleiche
  - Dateiprüfungen
  - Logische Operationen
  - Operationen auf Bit-Ebene
  - Operationen für bedingte Ausführung

wobei diese Aufzählung in der Darstellung des Nußschalenbuchs folgt. In der Praxis überschneiden sich diese Kategorien: so muß ein Vergleichsoperator muß explizit für Zahlen oder Zeichenketten ausgewiesen werden. Bestimmte Verknüpfungsoperatoren erzeugen mit Operanden verschiedener Art automatisch korrekte Ergebnisse, da implizit die Auswahl der richtigen Operation erfolgt. Gelegentlich werden allerdings scheinbar kryptische Ergebnisse erzeugt, die sich aber mit etwas Verständnis der Perl zugrundeliegenden Logik leicht entschlüsseln lassen:

```
$baer="Bär ";           # eine Zeichenkette
$wolf="Wolf";           # eine Zeichenkette
print $baer.$wolf;      # "Addition" von Zeichenketten,
                        # Resultat ist 'Bär Wolf'
print $baer+$wolf;      # Erzwungene numerische Addition, beide
                        # Variablen hatten den numerischen Wert
                        # "0", daher ist das Resultat "0" !
```

Für eine genaue Übersicht über alle Operatoren einschließlich Präzedenz und Assoziativität wird auf die manpage `perlop` sowie Kamel- und Nußschalenbuch verwiesen. Hier folgen einige ausgewählte Beispiele.

2.6.1 Operatoren für Zahlen und Zeichenketten

Perl unterscheidet nach Datentyp zwischen Operatoren für Textwerte und Operatoren für numerische Werte sowie nach Syntax zwischen unären, binären und ternären Operatoren.

Die Unterscheidung zwischen Zeichenketten und numerischen Werten ist besonders für Vergleiche von höchster Bedeutung, wie sich aus dem Vergleich der Skalare *2* und *11* einleuchtend ergibt. Als numerischer Wert ist *2* kleiner als *11*, aber als Zeichenkette sortiert hätte es einen höheren Rang.

Daher muß man bei Vergleichsoperationen immer den Datentyp angeben, was durch die Wahl des passenden Operators geschieht:

```
# a gleich b?
if ($a == $b) {}                # numerisch
if ($a eq $b) {}                # Zeichenkette ("equal")

# a ungleich b?
if ($a != $b) {}                # numerisch
if ($a ne $b) {}                # Zeichenkette ("not equal")

# a kleiner als b?
if ($a <  $b) {}                # numerisch
if ($a lt $b) {}                # Zeichenkette ("less than")

# a größer als b?
if ($a >  $b) {}                # numerisch
if ($a gt $b) {}                # Zeichenkette ("greater than")

# a kleiner als oder gleich b?
if ($a <= $b) {}                # numerisch
if ($a le $b) {}                # Zeichenkette ("less than or equal")

# a größer als oder gleich b?
if ($a >= $b) {}                # numerisch
```

```
if ($a ge $b) {}                # Zeichenkette ("greater than or equal")
```

Ein weiterer nach Inhalt verschieden wirkender Operator ist beispielweise der Zufügungs-Operator (Additionsoperator für Zahlen, Konkatenationsoperator für Zeichenketten).

```
# Hinzufuegungen
$a=4;           # $a wird '4'
$b=5;           # $b wird '5'
print $a+$b;    # Die Zahlensumme wird ausgegeben
print $a.$b;    # Die 'Summe' der Zeichenketten wird ausgegeben
```

Interessanterweise gibt es auch Inkrementoperatoren für Zeichenketten, die das lateinische Alphabet als Zahlensystem mit der Basis 26+26 betrachten, das durch die Zeichen 'A' bis 'z' ausgedrückt wird, und einen Multiplikationsoperator für Zeichenketten.

```
# Inkrementoperationen von Zeichenketten
$n=x            # $n wird 'x'
print ++$n;     # Ausgegeben wird 'y'
print ++$n;     # Ausgegeben wird 'z'
print ++$n;     # Ausgegeben wird 'aa'

# Multiplikation von Zeichenketten
print '*' x 16; # Ausgegeben wird '****************'
```

2.6.2 Unäre Operatoren

Die von C bekannten unären Operatoren ++ und - gibt es im numerischen Kontext auch in Perl:

```
$n=100;         # $n wird 100
print $n++;     # Gibt '100' aus; *anschließend* wird $n zu 101
print ++$n;     # Erhöht $n auf 102 und gibt diese Zahl aus
```

Ihre Anwendung im Zeichenketten-Kontext wurde eben schon vorgeführt.

2.6.3 Dateiprüfungs-Operatoren

Manchmal ist es wichtig, den konkreten Ablauf eines Perl-Programmes vom Vorhandensein und der Art einer Datei abhängig zu machen. In diesen Fällen greift man zu Dateiprüfungsoperatoren. Angenommen in `$a` sei ein Dateiname enthalten, so liefern die folgenden, beispielhaften Tests Boolesche Ergebnisse:

```
-e $a           # Wahr wenn Datei $a existiert
-r $a           # Wahr wenn Datei $a lesbar ist
-w $a           # Wahr wenn Datei $a schreibbar ist
-d $a           # Wahr wenn Datei $a ein Verzeichnis ist
-f $a           # Wahr wenn Datei $a eine gewöhnliche Datei ist
-T $a           # Wahr wenn Datei $a eine reine Text-Datei ist
```

2.7 Funktionen

Wie schon in der Einleitung zu Kapitel 2.6 (Operatoren) gesagt, hängt die Klassifizierung eines bestimmten Befehls oft von der Aufrufsyntax und dem übergegebenen Datentyp ab. Grundsätzlich kann man sich aber an die Einteilung der Perl-manpages halten und alle Befehle, die in `perlfunc` erklärt sind, als Funktionen verstehen.

Es gibt bei Perl einen großen Schatz von Funktionen für die Verarbeitung von Zeichenketten und numerischen Werten und die Umwandlung zwischen diesen. Diese Funktionen, aus denen übrigens das Gros der meisten Perl-Programme aufgebaut ist, sind in allen Portierungen von Perl gleichermaßen verfügbar, weswegen es keine Probleme mit der Ausführung auf verschiedenen Rechnern geben sollte.

Darüberhinaus gibt es bei Perl noch eine Gruppe von Funktionen, die auf bestimmte Schnittstellen des Betriebssystems aufsetzt und daher nicht auf allen Rechnertypen gleich implementiert sein kann. Dies betrifft insbesondere Funktionen für Netzwerkdienste und die Verwaltung von Dateien, Gruppen und Benutzern.

Die manpage `perlfunc` stellt alle Funktionen nach Kategorie und außerdem in alphabetischer Ordnung vor, so daß bereits aus der Zuordnung zu einer bestimmten Kategorie ersehen werden kann, ob die entsprechende Funktion für das gewünschte Betriebssystem überhaupt verfügbar ist.

2.7.1 Häufig gebrauchte Funktionen

An dieser Stelle kann keine Übersicht über den gesamten Funktionsumfang von Perl gegeben werden. Stattdessen wird auf die Aufzählung nach Kategorie zu Beginn der manpage `perlfunc` verwiesen (auch enthalten im Nußschalenbuch auf Seite 86). Die manpage `perlfunc` enthält außerdem eine vollständige alphabetische Darstellung aller Funktionen mit umfangreichen Beispielen.

Hier werden nur einige wichtige Funktionen erwähnt, die entweder besonders häufig verwendet werden oder besonders idiomatisch für Perl sind, da sie für Perls eigene Datenstrukturen verwendet werden.

Manche Funktionen können sowohl in Listenkontext als auch in skalarem oder Booleschem Kontext angewandt werden, liefern dann aber möglicherweise völlig unterschiedliche Ergebnisse. In solchen Fällen hilft ein Blick in die manpage weiter!

`chomp`

Sichere Variante von `chop`. Während `chop` das letzte Zeichen entfernt, entfernt `chomp` nur Zeilenumbrüche. Die Vorgabe für den Zeilenumbruch findet sich in der Sondervariablen `$/`; s. 2.4 (Vordefinierte Variable).

`chop`

Entfernt das letzte Zeichen einer Zeichenkette. Wird gebraucht, um den Zeilenumbruch bei Eingabezeilen zu entfernen.

```
while (<>) {
        chop;   # Zeilenumbruch wird von $_ entfernt
        print;  # Zeile wird gedruckt.
        }
```

`die`

Ermöglicht den definierten Ausstieg aus einem Programm, wenn ein Fehler vorhersehbar ist. Im folgenden Beispiel wird das Programm beendet, wenn eine Datei (deren Name in `$meinfile` enthalten ist) nicht gefunden wird.

```
open(MYHANDLE,$meinfile) or die "$meinfile nicht gefunden.";
```

`eval`

Gibt Perl-Code erst zur Laufzeit des Hauptprogramms zur Ausführung, so daß ohne Neustart von Perl Code auf Ausführbarkeit getestet werden kann. Eine praktische Anwendung zeigt das Programm `testbed.pl` in Kapitel 6.1 (Test-Programm).

`exec`

Beendet das gerade laufende Perl-Programm und startet an seiner Statt ein anderes. Im Gegensatz hierzu läßt `system` das gerade laufende Programm weiterleben.

`format`

Erzeugt eine formatierte Ausgabe. Siehe die detailliertere Erläuterung in Kapitel 2.10 (Formate).

`grep`

Kann einen Wahrheitswert oder eine Liste mit gefundenen Textfragmenten ausgeben.

```
@alle_zeilen=<STDIN>;                   # Alle Zeilen der Eingabe
                                        # werden eingelesen
@Kommentare = grep /^#/, @alle_zeilen;  # Gibt nur die Kommentare aus
```

`join`

Kombiniert Elemente eines Arrays zu einem Skalar. Erwartet ein Trennsymbol:

```
@jahreszeiten=(Fruehling,Sommer,Herbst,Winter);
$allevier=join(":",@jahreszeiten);
print $allevier;                # 'Fruehling:Sommer:Herbst:Winter'
```

`join` ist der Antagonist von `split`.

`pack`

Packt eine Anzahl von Daten in eine binär kodierte Form. Kann beispielsweise für die Umwandlung von Binärzahlen in andere Zahlen- und Zeichenformate verwendet werden:

```
$zeichen=pack("B8","11110000"); # Grundform: pack TEMPLATE DATEN
print $zeichen;                 # Gibt Zeichen aus, dessen
                                # ASCII-Wert in binärer Darstellung
                                # "11110000" ist.
```

`pop`

Gibt den letzten Eintrag eines Arrays (bzw. einer Liste) aus und entfernt diesen gleichzeitig.

```
@jahreszeiten=(Fruehling,Sommer,Herbst,Winter);
$Xmas=pop @jahreszeiten;
print "$Xmas\n";                                # 'Winter'
print @jahreszeiten;                            # 'FruehlingSommerHerbst'
```

Das Gegenstück zu `pop` ist `push`. Siehe auch `shift` und `unshift`, die auf den Anfang eines Arrays wirken.

`print`

Allgemeine Druckfunktion. Erwartet ein optionales Datei-Handle und eine Liste mit Daten. Liefert als Funktion die Zahl der gedruckten Daten zurück.

`printf`

Druckt eine formatierte Zeichenkette, wobei eine Formatbeschreibung, gefolgt von einer Datenliste, erwartet wird. Bietet vielfältige Darstellungsoptionen. Die verwandte Funktion `sprintf` gibt nur Zeichenketten zurück, verwendet aber die gleichen Regeln zur Formatbeschreibung.

`push`

Hängt Liste an das Ende eines Arrays an:

```
@jahreszeiten=(Fruehling,Sommer,Herbst);
push @jahreszeiten, "Winter";
print @jahreszeiten;                    # 'FruehlingSommerHerbstWinter'
```

Gegenteil von `pop`. Siehe auch `shift` und `unshift`, welche auf den Anfang eines Arrays wirken.

`reverse`

Kehrt die Reihenfolge einer Liste um:

```
@song=("do","re"","mi");
print @song;                    # 'doremi'
print reverse @song;            # 'miredo'
```

Um eine Zeichenkette rückwärts auszugeben, muß diese erst in eine Liste konvertiert werden:

```
print reverse split(//,"Wort"); # 'troW'
                                #
$Wort="Wort";                   # oder, *viel* einfacher:
$troW=reverse($Wort);           # (Man beachte die Klammern!)
print $troW;                    # 'troW'
```

`shift`

Gibt den ersten Wert eines Arrays aus und entfernt ihn gleichzeitig aus dem Array. Das ganze Array wird um ein Element kürzer.

```
@a=(a,b,c,d);
print shift @a;         # 'a'
print @a;               # 'bcd'
```

Die Umkehroperation ist `unshift`.

Siehe auch `pop` unf `push`, welche auf das Ende eines Arrays wirken.

`sort`

Sortiert eine Liste. Kann auf Anforderung numerische und Zeichenketten-Daten sortieren; außerdem ist die Definition eigener Sortierroutinen möglich.

`splice`

Schneidet Stück aus Array aus und fügt optional eine Ersetzung ein. Im folgenden Beispiel wird `$kunde` durch `@vollername` ersetzt:

```
@kundensatz=(Herd,Meier,Kredit);
@vollername=(Gerhard,Hubert,Meier);
splice @kundensatz,1,1,@vollername;
print join(":",@kundensatz);    # 'Herd:Gerhard:Hubert:Meier:Kredit'
```

`split`

Erwartet die Angabe einer Trennstelle. Wird benutzt, um einen Skalar (eine Zeichenkette) in mehrere Elemente aufzubrechen, die als einzelne Felder einem Array bzw. einer Liste übergeben werden:

```
$zeile="Anfang:Mitte:Ende";
@worte=split(/:/,$zeile);
print $worte[1];                # 'Mitte'
```

Die Ziel-Liste kann auch explizit konstruiert werden:

```
$nutzer="Mueller:Hans:m";
($nachname,$vorname,$geschlecht)=split(/:/,$nutzer);
print $vorname;                 # 'Hans'
```

Kehrt die Wirkung von `join` um.

`sprintf`

Erzeugt eine formatierte Zeichenkette aus einer Formatangabe und den anschließenden Datenfeldern. Die Definition der Felder ist die gleiche wie bei `printf`.

`system`

Betriebssystem-Aufruf. Nachdem `system` ein Kommando ausgeführt hat, läuft das kommandogebende Programm weiter. im Vergleich dazu ersetzt `exec` das aufrufende Programm. Vergleiche auch `eval`.

`tr`

Übersetzungsfunktion zum Umsetzen von Zeichenmengen:

```
$name="Bernie";
$name=~tr/A-Z/a-z/;      # Übersetzt alle Großbuchstaben in Kleinbuchstaben
print $name;             # 'bernie'
```

`unshift`

Schiebt neues Element an den Anfang einer Liste und gibt die Zahl der Elemente in der neuen Liste zurück:

```
@Essen=(Milch,Eier);
print unshift @Essen, Butter;   # '3'
print @Essen;                   # 'ButterMilchEier'
```

Kehrt die Wirkung von `shift` um. Siehe auch `pop` unf `push`, welche auf das Ende einer Liste wirken.

2.7.2 Funktionsdeklarationen

Wie in jeder echten Programmiersprache, so gibt es auch in Perl die Möglichkeit, Funktionen selbst zu definieren. Dies wird mit dem Schlüsselwort `sub` bewirkt, auf das der Funktionsname und ein Block mit der Definition folgen.

```
sub MeineFunktion BLOCK
```

Funktionsdefinitionen sind in mehreren Beispielprogrammen enthalten. Hier wird das Skelett einer Funktion noch einmal vorgeführt:

```
sub MeineFunktion {
        my $lokale_Variable=$_[0];      # liest den ersten übergebenen
                                        # Parameter aus
        # Aktionen
        $MeineFunktion="Rückgabewerte"; # so kann ein Skalar zurück-
                                        # gegeben werden
}
```

Anders als Pascal und viele andere Programmiersprachen, die bereits bei der Deklaration einer Funktion der genaue Zahl der Parameter und der zu übergebenden Argumente erwarten und im Sonderfall *keines* Parameters die Deklaration beispielsweise statt mit `FUNCTION` mit `PROCEDURE` einleiten, benötigt Perl keine Angaben über die Zahl der Parameter und Argumente. Diese werden in dem für die Funktion lokalen Array `@_` abgelegt, von wo sie ausgelesen werden können. Damit entfällt auch die C-typische Angabe von Prototypen.

2.8 Wörtlicher Text und Begrenzer

Die verschiedenen Anführungszeichen, die bei Perl erlaubt sind, bewirken, ob Variable interpoliert werden oder nicht, oder ob der Text in Anführungszeichen als Kommando aufgefaßt wird.

```
' '     q//     wörtlich, es wird nicht interpoliert
" "     qq//    wörtlich, es wird interpoliert
` `     qx//    Kommando, es wird interpoliert
( )     qw//    Wortliste, es wird nicht interpoliert
/ /     m//     Mustervergleich mit Interpolation
s///    s///    Ersetzungsoperation mit Interpolation
y///    tr///   Übersetzungsoperation zwischen
                Zeichenmengen ohne Interpolation
```

Grundsätzlich werden wörtliche Werte, die an Variablen übergeben werden, in einfache oder doppelte Anführungszeichen eingeschlossen. Davon hängt dann ab, ob eventuelle Variable interpretiert werden oder wörtlich eingesetzt werden:

```
$Wert="DM $betrag";     # $betrag wird mit seinem Wert eingesetzt
$Wert='DM $betrag';     # $betrag wird als '$betrag' eingesetzt; nicht gewollt.
```

Unter bestimmten Umständen können wörtliche Werte auch ohne Anführungszeichen übergeben werden; Perl bezeichnet dies als *bare words* (nackte Wörter):

```
$Wort=Silberling;       # Nur erlaubt, wenn keine Sonderzeichen
                        # enthalten sind; gilt auch für Arrays
@Pronomen=(er, sie, es) #
```

Zahlen bilden eine Sonderform der nackten Wörter, sie können immer ohne Begrenzer übergeben werden:

```
$betrag=4000;   # Kein Begrenzer nötig
```

Wenn die auf `q` lautenden Formen als Begrenzer genommen werden, kann sich Text auch über mehrere Zeilen erstrecken, was besonders für CGI-Programme sehr praktisch ist. Die Formen mit `q` akzeptieren alle Klammerarten, die der ASCII-Zeichensatz bietet, als Begrenzer.

2.9 Reguläre Ausdrücke

Ein Regulärer Ausdruck ist ein nach bestimmten Regeln geformtes Gebilde, dessen Auswertung einen oder mehrere Zeichenketten ergibt, die dann mit existenten Zeichenketten verglichen werden können.[8]

[8] Exakter gesprochen, handelt es sich bei Regulären Ausdrücken um Ausdrücke, mit denen sich Wörter nach Regeln von Sprachen des Typs III bilden lassen.

Alle Regulären Ausdrücke müssen von ihrer Umgebung abgegrenzt werden. Dazu wird in Perl meistens der Schrägstrich benutzt, obwohl dies nicht zwingend vorgeschrieben ist.

Reguläre Ausdrücke genügen daher zumindest der folgenden Form:

```
# /RA/                           # Ein Regulärer Ausdruck wird im
                                 # Regelfall in Schrägstriche
                                 # eingeschlossen.

if (/RA/) { print ; }            # Druckt die in $_ gespeicherte
                                 # Zeichenkette dann, wenn sie RA
                                 # enthält.
```

Reguläre Ausdrücke werden aber auch häufig für bedingte Such- und Ersetzungsoperationen eingesetzt. Die Notation `s/alt/neu/` ersetzt jedes Vorkommen des Ausdrucks `alt` gegen den Ausdruck `neu`.

Der einfachste RA (englisch RE, Regular Expression) wird durch Zeichen gebildet, die nicht anders interpretiert werden können als in ihrer normalen Bedeutung.

Daher ist `/RA/` ein Regulärer Ausdruck, mit dem sich die Buchstabenfolge `RA` in einem gegebenen Text finden läßt. Dies kann die Textzeile `RA` ebenso sein wie die Zeile `Wir trinken alle heiße RAMA mit Marshmallows`.

Hingegen wird die Buchstabenfolge `ra` in `Brahma` nicht gefunden, da diese ja nicht mit `RA` identisch ist.

Es muß daher Wege geben, Reguläre Ausdrücke zu erweitern. Dazu bedient man sich sogenannter Metazeichen, die nicht für sich stehen, sondern etwas anderes bedeuten. Die wichtigsten Metazeichen werden nun vorgestellt:

`.`

Der Punkt steht für jedes beliebige Zeichen außer dem Zeilenende.

`$`

Das Dollarzeichen verankert den Suchausdruck am Zeilenende.

`^`

Der Circonflex verankert den Suchausdruck am Zeilenanfang.

`[ ]`

In eckigen Klammern wird eine Zeichenmenge erfaßt, die an *einer* Stelle des RAs erwartet wird. `[abc]` bedeutet, daß an einer Stelle ein `a`, ein `b` oder ein `c` stehen darf.

In eckigen Klammern negiert der Circonflex als erstes Zeichen die Menge. `[^abc]` bedeutet also, daß an einer Stelle weder ein `a`, ein `b` oder ein `c` stehen darf, dafür aber alle anderen Buchstaben.

In eckigen Klammern erzeugt der Bindestrich - eine Liste. `[a-e]` erfaßt also alle Buchstaben von `a` bis `e`. Analog gilt auch hier die Negierungsregel mit `^`. Der Ausdruck `[^a-e]` findet also alle Zeichen außer den Buchstaben von `a` bis `e`.

`?`

Das Fragezeichen gibt an, daß das vorausgegangene Atom mindestens keinmal, aber auch einmal erscheinen kann.

`+`

Das Pluszeichen gibt an, daß das vorausgegangene Atom mindestens einmal, aber auch mehrfach erscheinen kann.

`*`

Der Stern gibt an, daß das vorausgegangene Atom beliebig oft erscheinen kann. Beliebig oft schließt auch keinmal ein. Daher wird mit dem Ausdruck `/ab*c/` neben `abc` und `abbbbbbc` auch `ac` gefunden.

`{n,m}`

Alternativ kann in geschweiften Klammern auch ein Bereich angegeben werden, wobei `n` für die kleinste Zahl, `m` hingegen für die größte erlaubte Zahl des Vorkommens steht.

`|`

Der Vertikalstrich reiht mehrere gleichberechtigte Suchausdrücke aneinander. Der Ausdruck `/auf|oder|ab/` findet die Zeichenketten `auf`, `oder` und `ab`. Das heißt, daß natürlich nicht nur die gleichlautenden Wörter, sondern auch `Lauf`, `Schabefleisch` und `Modernes Leben` gefunden werden, um nur ein paar Beispiele zu nennen.

`()`

Einfache Klammern dienen zur Zwischenspeicherung eines Ausdrucks, der später weiterverwendet werden kann. Auf den solchermaßen zwischengespeicherten Ausdruck kann anschließend bei einer Ersetzungsoperation zurückgegriffen werden.

`\`

Der rückwärts geneigte Schrägstrich kehrt die Wirkung von Zeichen um. Metazeichen werden nun als einfache Zeichen interpretiert, während bisherige einfache Zeichen nun neue Bedeutungen annehmen.

`$1..9`

Mit Dollar-Zahl oder rückwärts geneigtem Schrägstrich wird ein in Klammern eingeschlossener Teilausdruck referenziert. Mit „Dollar-Zahl" läßt sich der Ausdruck auch außerhalb der Operation weiter referenzieren.

```
$wort="Wort";
$wort =~ s/(o)/\1-\1/;
print $wort;            # Gibt "Wo-ort" aus.
$wort="Wort";
$wort =~ s/(o)/$1-$1/;
print $1;               # Gibt "o" aus.
```

`\s`

Mit `\s` kann man Wortzwischenräume finden, da `\s` als `[ \t\n]+` definiert ist.

`\w`

Mit `\w` kann man Wörter finden, da `\w` als `[A-Za-z]+` definiert ist.

Es gibt darüberhinaus noch eine Reihe weiterer Buchstaben mit Sonderfunktion, die in *Programming Perl* auf Seite 64 ff. und in der manpage `perlre` näher erläutert werden.

Substitutionsoperationen der Art `s/alt/neu/` erlauben die Angabe von Optionen, deren wichtigste hier aufgeführt werden. Mehrere Optionen dürfen miteinander kombiniert werden.

`g`

Diese Option gibt an, daß eine Substitution für alle Fundstellen innerhalb der Zeichenkette erfolgen soll:

```
$wort="Schlammschlacht";
$wort=~s/la/ru/;
print $wort;           # "Schrummschlacht" erscheint
$wort="Schlammschlacht";
$wort=~s/la/ru/g;
print $wort;           # "Schrummschrucht" erscheint
```

`e`

Diese Option gibt an, daß die rechte Seite des Substitutionsausdrucks evaluiert werden soll. Es ist daher möglich, auf der rechten Seite eine Funktion oder eine Variablenabfrage in den Ersetzungsstring zu schreiben:

```
s/(.)/$meinhash{$1}/eg; # Ersetzt jedes Zeichen durch den Wert, der
                        # im Hash %meinhash gefunden wird, wobei
                        # das Zeichen als Schlüssel fungiert
```

`x`

Diese Option gibt an, daß der Substitutionsausdruck zwecks leichterer Lesbarkeit durch Leerstellen, Zeilenumbrüche und Kommentare unterbrochen werden darf:

```
s/([0-9]{5})[ \t]?      # Postleitzahl, dann Leerraum
  ([A-ZÄÖÜ][a-zäöü]+    # Städtename
  [ A-Zäöüa-zäöü]*)     # Zusatz (z.B. "im Breisgau")
  /PLZ: \1Ort: \2/x;    # Ersetzungsausdruck
```

Die eben angegebene Form ist inhaltlich identisch mit der folgenden, aber mit Sicherheit einfacher zu verstehen:

```
s/([0-9]{5})[ \t]?([A-ZÄÖÜ][a-zäöü]+[ A-Zäöüa-zäöü]*)/PLZ: \1Ort: \2/;
```

2.10 Formate (Ausgabeschablonen)

Perl ist auch eine Sprache zur Erzeugung strukturierter Datenübersichten (*reports* auf neudeutsch), wie es im Namen Perl (Practical Extraction and Report Language) ja schon gesagt wird.

Es ist natürlich möglich, mit `sprintf` und möglichen Hilfskonstruktionen ansprechend formatierte Datenbankauszüge zu erstellen, noch einfacher ist aber die Verwendung des Perlschen Formatgenerators.

2.10.1 Formatdeklarationen

Ein Format wird ähnlich wie eine benutzerdefinierte Funktion deklariert. Auf das Schlüsselwort `format` folgen der optionale Name des Formats und ein Gleichheitszeichen. In der nächsten Zeile kommt die erste Zeilenschablone, darauf folgen die Variablen, die nach dieser Schablone ausgegeben werden sollen. Die Formatdeklaration wird durch einen Punkt zu Zeilenanfang abgeschlossen.

Formate können überall im Quelltext stehen und sind an keine bestimmte Ordnung in Bezug auf die Hauptroutinen des Programms gebunden.

Aufgerufen wird ein Format durch das Befehlswort `write`, dem optional der Formatname nachgestellt wird. Wenn der Formatname in der Deklaration und im Aufruf weggelassen wird, so wird die Standardausgabe als Voreinstellung angenommen.

Das folgende Codefragment zeigt eine Iteration über das Array `@worte`, dessen einzelne Elemente rechtsbündig ausgegeben werden. Eine praktische Anwendung ist in Kapitel 6.4 (Rückläufige Sortierung) beschrieben.

```
foreach $wort (@worte) {
        write FORMATNAME
}

format FORMATNAME =
@>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
$wort
.
```

Text in Formaten kann rechtsbündig, linksbündig, zentriert, und über mehrere Zeilen umgebrochen ausgegeben werden. Die dafür nötigen Deklarationen und Befehle sind alle in der manpage `perlform` erklärt.

3 Laufzeiteigenschaften

3.1 Kommandozeilenparameter und Argumente

Perl verfügt über abschreckend viele Kommandozeilenoptionen, wie ein erster Blick auf die manpage `perlrun` schnell offenbart. Das heißt jedoch nicht, daß man Perl nicht ohne Kommandozeilenparameter betreiben könnte. Die wichtigsten Parameter, die für die eine oder andere Aufgabe nützlich sind, seien hier vorgestellt.

`-0`

Die Null (kein `O`!) erwartet als unmittelbar folgenden Parameter den Wert des Datensatztrenners. Der Datensatztrenner teilt Perl mit, was als Datensatz zu interpretieren ist; im Normalfall ist dies der Zeilenumbruch. So fängt in Listen ja jeder neue Eintrag (=Datensatz) mit einer neuen Zeile an.

Sollte dieses Verhalten nicht gewünscht sein, weil man Texte bearbeiten möchte, deren Daten innerhalb eines „Datensatzes" über mehrere Zeilen verteilt sind, so muß man sie absatzweise einlesen. Dies wird erreicht, indem als Datensatztrenner eine Null übergeben wird, so daß dann folgendes Beispiel gilt:

```
#!/usr/bin/perl -00
while (<>) {
        # $_ enthält immer einen ganzen Absatz mit Zeilenumbrüchen!
}
```

`-d`

aktiviert den Debugger. Es wird damit möglich, einen Programmtext Anweisung für Anweisung abzuarbeiten, Variable zu inspizieren und mit neuen Werten zu versehen, etc.

`-i`***Suffix***

Mit dieser Option läßt sich angeben, ob und unter welchen Namen Sicherheitskopien von einer Datei angelegt werden.

`-n`

schließt den ganzen Programmtext in einer `while`-Schleife ein, um ihn auf alle Zeilen eines Quelltextes anzuwenden, und simuliert damit die implizite Schleife von awk und sed.

`-p`

geht noch einen Schritt weiter als `-n` und druckt automatisch den Wert der Standardvariablen aus, so daß auch kein `print`-Befehl mehr nötig ist.

`-w`

gibt Warnungen aus. Diese, vielleicht praktischste aller Kommandozeilenoptionen sollte zumindest während der Entwicklungsphase eines Perl-Programms immer aktiviert sein, da sie auf nicht deklarierte und nicht initialisierte Variable und viele andere Dinge hinweist.

4 Module

Perl ist modular konzipiert, so daß wiederkehrende Probleme mit Hilfe fertiger Programmbibliotheken, der Module, gelöst werden können. Es ist daher möglich, den Perl-Interpreter kompakt und schlank (und damit schnell) zu halten.

Bei Bedarf können Module für die Lösung bestimmter Aufgaben herangezogen werden. Diese Module können selbst geschrieben werden, es können aber auch die Module verwendet werden, die Teil der Standardbibliothek jeder vollständigen Perl-Installation sind. Darüberhinaus gibt es Module für die Lösung der unterschiedlichsten Spezialprobleme, deren vollständige Einbindung den Rahmen einer kompakten Perl-Distribution sprengen würde und die auch nicht von jedem Benutzer immer unbedingt bereitgehalten werden müssen. Diese Module finden sich in der großen, rasch wachsenden und ständig aktualisierten Modulbibliothek des Perl-Archivs CPAN.

4.1 Einbindung von Modulen

Module sind eigentlich nichts anderes als Perl-Programme, die in der Regel nicht eigenständig ausgeführt werden (obwohl dies bei einigen Modulen auch möglich und sinnvoll ist). Alle in einem Modul deklarierten Variablen und Funktionen stehen auch dem Programm zur Verfügung, das dieses Modul aufruft. Daher braucht man beispielsweise in einem Perl-Programm, das als CGI-Skript ausgeführt werden soll, nur am Anfang die Zeile

```
use CGI;
```

zu schreiben, und schon steht der volle Umfang der CGI-Bibliothek bereit. Welche Funktionen bereitgestellt werden, ist der Dokumentation des jeweiligen Moduls zu entnehmen. Wichtige Module haben ihre eigene manpage, so daß in unserem Beispiel als manpage `CGI` aufgerufen werden kann.

4.1.1 Die Standard-Modulbibliothek

Jede Perl-Installation wird mit einer Standardbibliothek von Modulen ausgeliefert, die bei ordnungsgemäßer Installation automatisch benutzbar ist. Dem Anfänger sei empfohlen, zur Übersicht zunächst *nicht* die manpage `perlmod` zu lesen, da diese eine theoretische Einführung in das Modulkonzept von Perl enthält, zu deren Verständnis auch das Objektmodell von Perl vorausgesetzt wird. Brauchbarer ist dagegen die manpage `perlmodlib`, wo bis zum Kapitel *Standard Modules* vorzublättern ist. Die Module sind so zahlreich, daß sie hier ausnahmsweise nicht alle genannt werden; es wird daher auf die manpage und *Programming Perl* verwiesen, die beide vollständige, aber knappe Beschreibungen der Bibliothek enthalten. Einzelne Modulgruppen sind im Nußschalen-Buch allerdings sehr viel ausführlicher erklärt, so daß sich das Nußschalen-Buch besser für Anfänger eignet, während *Programming Perl* fortgeschrittene Kenntnisse voraussetzt.

4.2 Übersicht über die bei CPAN vorhandenen Module

Die im CPAN gespeicherten Module sind in folgende Kategorien unterteilt:

Perl Core Modules

Diese Module sollten eigentlich bei jeder Perl-Installation dabei sein.

Development Support

Entwicklungsunterstützung

Operating System Interfaces

Schnittstellen zu Betriebssystemen

Networking Devices IPC

Netzwerkfähigkeiten

Data Type Utilities

Routinen zur Datentypwandlung

Database Interfaces

Schnittstellen zu Datenbanken

User Interfaces

Hier findet sich u. a. Tkperl, eine Perl-Version mit eingebauten Tk/Tcl-Routinen.

Language Interfaces

Schnittstellen zu anderen Programmiersprachen

File Names Systems Locking

Dateinamenverwaltung

String Lang Text Proc

Zeichenkettenverarbeitung

Opt Arg Param Proc

Erweiterungen zur Verwaltung von Kommandozeile, Umgebungen etc.

Internationalization Locale

Anpassung an Fremdsprachen

Security and Encryption

Datensicherheit und Verschlüsselung

World Wide Web HTML HTTP CGI

Alles zur Bearbeitung von HTML-Daten

Server and Daemon Utilities

(Der Name sagt es)

Archiving and Compression

Archivierung und Datenkompression

Images Pixmaps Bitmaps

Alles zur pixelweisen Bildbearbeitung

Mail and Usenet News

Alles zum Thema Post und Nachrichten

Control Flow Utilities

Routinen für die Ablaufkontrolle

File Handle Input Output

Datei-Handels, Ein- und Ausgabe

Microsoft Windows Modules

Spezialisierte Module für Windows

Miscellaneous Modules

Verschiedenes

Commercial Software Interfaces

Schnittstellen zu kommerzieller Software

Not In Modulelist

Alles andere

4.3 Installation neuer Module

Die Installation neuer Module kann automatisiert oder manuell erfolgen. Wenn der Computer Netzanbindung hat, das gewünschte Modul bei CPAN vorliegt und am Modul keine Änderungen vorgenommen werden sollen, empfiehlt sich die automatisierte Methode, ansonsten kann immer noch manuell verfahren werden.

4.3.1 Automatisches Installationsverfahren

Perl ist in der Lage, bei CPAN vorliegende Module automatisch zu holen, zu entpacken und zu installieren. Das Perl-Modul CPAN (in der Standardinstallation enthalten) kann mit folgendem Aufruf beim Start von Perl eingebunden werden:

```
$ perl -MCPAN -e shell
```

Bei diesem Aufruf stellt Perl normalerweise eine interaktive Eingabeaufforderung (`-e shell`) bereit. Erfolgt der Aufruf zum ersten Mal, werden verschiedene Parameter abgefragt und nach Vorgabe bzw. Benutzervorschlag gesetzt. Anschließend kann man dann einen CPAN-Spiegel angeben, von dem das Programm in Zukunft Module anfordern soll.

Bei jedem späteren Aufruf steht sofort ein Kommandoprompt bereit, an dem man z.B. `i Tk` eintippen kann, um das Modul Tk zu installieren. Es wird dann auch automatisch die passende Versionsnummer ausgesucht. Wenn das Modul gefunden ist, wird es in den Computer übertragen, dort entpackt und kompiliert. Anschließend kann es sofort benutzt werden. Sollte festgestellt werden, daß Abhängigkeiten mit weiteren Modulen bestehen, so werden auch diese automatisch installiert.

Ausführliche Beschreibungen finden sich in der manpage `CPAN` sowie im Nußschalenbuch auf Seite 188.

4.3.2 Manuelles Installationsverfahren

Zunächst wird das Archiv des Moduls in ein frei wählbares Verzeichnis kopiert. Das kann `/tmp` ebenso sein wie `/usr/local/src`, oder `/home/goofy/test`. Wenn das Archiv mit dem Modul (das immer als komprimiertes `tar`-Archiv vorliegt) mit dem Befehl

```
$ tar fxvz MyModule-0.93.tar.gz
```

entpackt worden ist, entsteht ein neues Unterverzeichnis, das regelmäßig eine `README`-Datei und eine Datei namens `Makefile.PL` enthält. Zunächst ist in das neue Unterverzeichnis zu wechseln. Es empfiehlt sich immer, zuerst die `README`-Datei zu konsultieren. Sie enthält oft Hinweise auf Mindestanforderungen der benötigten Software (Perl-Versionsnummern, auf jeden Fall benötigte andere Perl-Module und ihre Versionsnummern, externe Entwicklungsbibliotheken etc.). Bei größeren Modulen finden sich hier auch maschinenspezifische Hinweise für Compiler, Pfade etc.

Wenn nach der Lektüre der `README`-Datei dem Installationsbeginn nichts mehr im Wege steht, kann die Installation mit folgenden Schritten vollzogen werden:

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

4.4 Versionsprobleme

Das große Tempo, mit dem die Entwicklung von Perl und seinen Modulen vorangetrieben wird, hat natürlich zunächst den Vorteil, daß dem Anwender ein ständig wachsender Funktionsumfang zur Verfügung steht. Auf der anderen Seite ist nicht immer ganz einleuchtend, daß auch große Gewinne an Funktionalität nur in einer kleinen Veränderung der Versionsnummer resultieren. So gering der Übergang von 5.003 nach 5.004 erscheinen mag, so sehr verändert sich im Einzelfall die Behandlung einzelner Datenstrukturen.

Während im Normalfall die hinzugekommene Funktionalität keine Probleme verursacht, kann es bei der Weitergabe von Perl-Texten zu Schwierigkeiten kommen, wenn stillschweigend die Leistungsmerkmale einer bestimmten Version angenommen werden.

Kritischer kann es schon bei Modulen werden, deren Entwicklung noch nicht abgeschlossen ist. Hier kann es gut sein, daß sich interne Schnittstellen ohne Ankündigung ändern. Module, die aufeinander aufbauen, sind daher an die Einhaltung der geforderten Versionsnummern gebunden. Es kann nämlich durchaus sein, daß MeinModul-2.63 ganz wunderbar mit MeinAnderesModul-0.74 zusammenarbeitet, aber die neuere Version MeinModul-2.81 das andere Modul MeinAnderesModul-0.77 voraussetzt.

Leider kommt es in wenigen Fällen auch vor, daß bestimmte Mechanismen gelegentlich umgeschrieben werden und bei gleicher Funktionsweise zwar gleich erscheinen, aber gegenüber der Vorgängerversion optimiert sind, wobei dann neue Nebenwirkungen und Wechselwirkungen auftreten. In Einzelfällen verlangen daher bestimmte Anwendungen einen früheren als den aktuellen Versionsstand. Ein Beispiel hierfür ist das auf gnuplot basierende Plot-Programm.

Glücklicherweise gibt es in Perl einen Mechanismus zur Abfrage und Weitergabe von Versionsnummern, so daß im Zweifelsfall der Benutzer auf entstehende Probleme hingewiesen wird.

Für das Gros der täglichen Arbeit mit Perl stellen Versionskonflikte des Hauptprogramms und der Module kein Problem dar.

5 Programmierstil, oder TIMTOWTDI

Eine der auf Neulinge gelegentlich abschreckend wirkenden Eigenschaften von Perl ist neben den kryptischen eingebauten Variablen die große formale und syntaktische Freiheit in der Quellcodegestaltung. Diese Freiheit ermöglicht es fortgeschrittenen Programmierern, erstaunlich kompakten Code zu schreiben. Außerdem führen in Perl viele Wege zum Ziel. Dies ist in der Abkürzung TIMTOWTDI zusammengefaßt: There Is More Than One Way To Do It. Wie viele Wege es wirklich gibt, auch einfache Dinge in Perl auf überzeugende Weise zu komplizieren, wird in der JAPH-Sammlung dokumentiert. Die dortige Aufgabe ist, den Text *Just*

Another Perl Hacker auszugeben. Die bizarrsten Ideen werden in einer eigenen Datei namens JAPH (Just Another Perl Hacker, zu finden auf CPAN:/misc/) gesammelt. Diese Sammlung eignet sich nicht nur zum Studium von Verfahren, die man besser nicht anwendet, sondern enthält auch ein paar wirklich überzeugende Beispiele bestimmter Funktionen und Programmiertechniken.

In den folgenden Beispielprogrammen werden im Gegensatz zu JAPH nur praktikable und nachvollziehbare Lösungen angeboten. Es kann allerdings durchaus sein, daß mehrere Wege nach Rom führen, oder sich mehrere Lösungsansätze anbieten. Dies kann unerheblich sein, so lange nur Vorlieben und Abneigungen des Programmierers eine Rolle spielen, kann aber auch ernsthafte Konsequenzen im Laufzeitverhalten oder der grundsätzlichen Logik des Programmablaufs mit sich bringen.

5.1 Laufzeitverhalten von Stilvarianten

Es ist besonders hervorzuheben, daß Perl ein deutlich unterschiedliches Laufzeitverhalten für verschiedene Operatoren und Operationen aufweist. Es empfiehlt sich immer, nach einem für Perl idiomatischen Weg zu suchen, da dieser meistens signifikant schneller abgearbeitet wird. Das folgende einfache Beispiel benutzt entweder eine konventionelle Schreibung der Addition zweier Zeichenketten oder aber den Perl-typischen Operator.

```
#!/usr/bin/perl

# Ein Laufzeittest

for ($i=0; $i<=50000; $i++) {
        # Konventioneller Weg, zwei Strings zu verknüpfen
        # $zeichenkette=$zeichenkette."a"
        # Perl-idiomatischer Weg zur Lösung der gleichen Aufgabe
        $zeichenkette.="a"
}
```

Hier sehen wir Abarbeitungszeit und Systemauslastung des Programms, wenn der konventionelle Weg beschritten wird:

```
$ time ./run.pl
67.85user 0.07system 1:09.43elapsed 97%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (197major+59minor)pagefaults 0swaps
```

Hier sehen wir Abarbeitungszeit und Systemauslastung des gleichen Programms, wobei allerdings hier der idiomatischere Weg beschritten wird:

```
$ time ./run.pl
0.54user 0.01system 0:01.14elapsed 48%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (197major+37minor)pagefaults 0swaps
```

Der Unterschied (eine Sekunde statt einer Minute Laufzeit bei halber statt voller CPU-Auslastung) ist erstaunlich groß, wobei ja in der ganzen Laufzeit von einer Sekunde bereits der Aufruf des Perl-Interpreters und die Laufzeitkompilation enthalten sind. Es kann durchaus sein, daß der konventionelle Programmierstil bei kleinen Datenmengen, wie sie typischerweise zur Validation eines Programms benutzt werden, zu scheinbar vorzeigbaren Ergebnissen führt, da diese ja *korrekt* sind. Mit kleinen Datenmengen läßt sich aber keine Aussage zum Laufzeitverhalten machen. Erst mit „echten" Datenmengen stellt man dann fest, daß das Programm Antwortzeiten produziert, die es für den vorgesehenen Zweck völlig unbrauchbar machen können.

Im obigen Beispiel wächst die Laufzeitdifferenz exponentiell mit der Datenmenge; wenn für `$i` ein kleiner Maximalwert (z. B. 1000) eingesetzt wird, ist kaum ein Unterschied zu spüren. Wird aber statt der 50000 unseres Beispiels ein noch größerer Wert eingesetzt, so wachsen die Antwortzeiten ins Uferlose.

5.2 Logisches Programmgerüst

Wenn sich das Problem stellt, aus einer lateinischen Transliteration beispielsweise einen kyrillischen Text zu generieren, so kommt man nicht umhin, der Behandlung von Digraphen besondere Aufmerksamkeit zu widmen. Während für einfache 1:1-Ersetzungen die Funktion `tr` völlig ausreicht, kann man mit ihr keine Texte übersetzen, in denen mehrere Buchstaben des Transliterats für einen Buchstaben des Ziels stehen. Man kann aber mit einer Reihe von Substitutionen arbeiten. Wenn im lateinischen Text `s` und `sh` auftreten, die für distinkte kyrillische Buchstaben stehen, so könnte ein Ansatz so aussehen:

```
s/sh/<Kode für sh>/g;
s/s/<Kode für s>/g;
```

wobei sinngemäß die anderen Buchstaben auch zu substituieren sind. Es kommt allerdings auf die Reihenfolge an. Würden beide Zeilen miteinander vertauscht, würde das Skript kein `sh` mehr im Quelltext finden und der Zieltext wäre verfälscht.

Ein weiteres Problem dieses Ansatzes ist es, daß die Datenräume von Quelle und Ziel nicht voneinander getrennt sind. In dem Maße, wie die Substitutionen für jeden einzelnen Buchstaben des Alphabetes vollzogen werden, verschwindet der Quelltext aus der Variablen, bis er schließlich nicht mehr vorhanden ist. Er könnte zwar vor der Substitutionsoperation in einer anderen Variablen gesichert werden, aber prinzipiell wirft der hier vorgestellte Ansatz Probleme auf, die bei komplexen Programmen zu unübersehbaren Schwierigkeiten führen können.

Sinnvoller ist es daher, Funktionen zu schreiben, die den Quelltext unangetastet lassen und einen neuen Wert zurückgeben:

```
$neu=transliteriert($alt);

sub transliteriert {                            # Funktion wird definiert
        my $eingabe=$_[0];                      # Parameter liegen in @_
        # Kommandos                             # Hier wird gearbeitet...
$transliteriert=$irgendein_lokaler_wert;        # Rückgabe
}
```

Das angegebene Fragment ist natürlich nur als Gerüst zu verstehen, da die eigentliche Transliterationsroutine ja sehr komplex sein kann und von Fall zu Fall neu zu schreiben ist.

6 Musterprogramme

Die hier aufgeführten Beispielprogramme sind ohne Einschränkungen funktionsfähig und können für weitere Ausbauten leicht erweitert werden.

6.1 Ein Programm zum Probieren und Üben

Manchmal kann es praktisch sein, Perl-Kommandos einzeln einzugeben, um ihre Eigenschaften zu untersuchen; es ist jedoch lästig, dafür jedes Mal den Perl-Interpreter neu aufzurufen. Mit Hilfe des `eval()`-

Operators lassen sich Perl-Konstrukte ausführen, die auch fehlerhaft sein dürfen. Im Gegensatz zum normalen Perl-Skript wird die Ausführung dann nicht abgebrochen, sondern in der Laufzeitumgebung eine Fehlernummer und eine Kopie der vermuteten Fehlerstelle ausgegeben, die dann bei Bedarf weiterverarbeitet werden können. Das folgende Skript liest Zeilen von der Standardeingabe und versucht, sie als Perl-Kommandos auszuführen.

```
#!/usr/bin/perl
while (<STDIN>) {
        chomp;
        $command=$_;
        eval $command;
        if ($@) {
                print STDERR "Code: $command; eval: $@\n" ;
        }
}
```

6.2 Wortschatzstatistik

Mit dem nächsten Programm wird gezeigt, wie man mit einfachen Mitteln eine alphabetisch sortierte und gezählte Liste aller Wörter eines gegebenen Textes erstellen kann.

```
#!/usr/bin/perl
#
# Sehr einfaches Programm zur Zählung von Wörtern in einem Text
#

while (<>) {                            # Zeilenweises Einlesen nach $_
        split(/\W/);                    # \W trennt auf nicht-Wort-Zeichen
        foreach $wort (@_) {            # Einzelne Wörter befinden sich in @_
                $wortliste{$wort}++;    # Einklinken in Hash, eventuell
        }                               # Inkrementieren des Zählers
}

# Nachdem alle Wörter gelesen worden sind,
# wird das gefüllte Hash ausgegeben.
#
foreach $wort (sort keys %wortliste) {
        print "$wort: $wortliste{$wort}\n";
}
```

Obwohl in dem kleinen Beispielprogramm kanonische Verfahren zur sortierten Auflistung von Schlüsseln in Hashes verwendet werden (`sort keys %wortliste`), ist das Programm natürlich nicht besonders leistungsfähig. Verbesserungswürdig ist zunächst der Mechanismus des Aufbrechens in einzelne Wörter. Der Reguläre Ausdruck `\W` ist eine verkürzte Schreibweise für `[^A-Za-z]` und trennt die eingelesene Zeile in ein Array, das voreingestellt in `@_` vorliegt. Bedauerlicherweise erfaßt der Ausdruck `\W` keine deutschen Umlaute, diese müssen daher anders angegeben werden.

Auch die Ausgabe der Daten läßt in der Form noch zu wünschen übrig. Es wird als Übung für den Leser angegeben, die Ausgabe mit Hilfe von `printf` oder eines zu definierenden Formats in wohlgeordnete Spalten zu bringen und eventuell das Ergebnis nach Häufigkeit zu sortieren.

6.3 Buchstabenstatistik

Das Programm zur Wortschatzermittlung läßt sich mit wenigen Mitteln abändern, so daß sich nun Buchstaben zählen lassen. Buchstabenstatistiken eignen sich beispielsweise zur automatisierten Erkennung der Sprache eines Textes; darüberhinaus können mit ausgezählten Buchstabenmengen Parser gebaut und getestet werden, da jeder gefundene Buchstabe eines Eingabetextes explizit ausgegeben wird und nicht übersehen werden kann.

```
#!/usr/bin/perl -w
# at least this is where my perl should reside
#
#
# Read all data
#
while (<>) {
        s/sh/S/g;
        @letters=split(/[ \t\n]*/);
        foreach $letters (@letters) {
                if (ord $letters>0)
                        {
                        $hash{$letters}++};
                        }
        }
# Spit out all data
#
foreach $key (sort {$hash{$b} <=> $hash{$a} } keys %hash)
        {
        printf "%8d %s -> %3d\n", $hash{$key}, $key, ord $key;
        }
```

Um die gefundenen Daten nach Häufigkeit zu sortieren, muß der zu einem Schlüssel (`keys %hash`) gefundene Wert numerisch sortiert werden. Im optionalen Block der Funktion `sort`, der hier in geschweiften Klammern angegeben ist, wird ein Vergleich der gefundenen Werte (die temporär in die vordefinierten Variablen `$a` und `$b` eingelesen werden) ausgeführt. Wird die Vergleichsoperation mit <=> angegeben, so werden die Werte numerisch sortiert; würde stattdessen `cmp` stehen, so fände ein Zeichenkettenvergleich statt.

6.4 Rückläufig sortierte rechtsbündige Ausgabe einer Wortliste

Mit den folgenden Lösungsvarianten der Aufgabe, eine Wortliste rückläufig sortiert rechtsbündig auszugeben, läßt sich eindrucksvoll die Mächtigkeit von Perl demonstrieren. Rückläufig sortierte Wortlisten werden vielfach eingesetzt, um Wortbildung und grammatische Endungen zu untersuchen, aber auch, um verstümmelte Texte zu entziffern.

Die erste angegebene Lösung geht noch einen relativ wenig idiomatischen Weg zum Ziel.

```
#!/usr/bin/perl

@worte=();              # Initialisieren

@worte=<STDIN>;         # Worte einlesen
```

```
@etrow= sort {reverse($a) cmp reverse($b)} @worte;  # sortieren

foreach $wort (@etrow) { # Rückläufig sortiertes Array ausgeben
        write;
}
                              # Dabei wird folgendes Format benutzt
format =
@>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
$wort
.
```

Zuerst wird ein leeres Array namens `@worte` initialisiert, das dann die Eingabe von STDIN als Listenkontext liest. Anschließend wird das Array sortiert, wobei `sort` mitgeteilt wurde, daß ein Stringvergleich (`cmp` auf die gewendeten Zeichenketten (`reverse($a)`, `reverse($a)`) durchzuführen ist. Das solchermaßen sortierte Array wird in `@etrow` abgelegt, und dann Element für Element mit `write` dem unbenannten Format auf STDOUT übergeben, das für die rechtsbündige Ausgabe zuständig ist.

in idiomatischem Perl-Stil spart man sich den ganzen Vorlauf und das benannte Array `@worte`, ebenso das Array `@etrow` zur Zwischenablage der Ergebnisse. Stattdessen läßt sich die Aufgabe auch mit dem folgenden, wesentlich kompakteren Code lösen:

```
#!/usr/bin/perl

foreach $wort (sort {reverse($a) cmp reverse($b)} <STDIN>) {
        write;
}

format =
@>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
$wort
.
```

6.5 Akronymdatenbank, erster Anlauf

Das folgende (mehrteilige) Programmbeispiel beschreibt, wie man aus einem bestehenden LaTeX-Text eine über Schlüssel aufrufbare Datenbank machen kann. Das Textbeispiel zeigt ein Bruchstück eines Glossars mongolischer Abkürzungen, denen jeweils mehrere Felder zugeordnet sind. Das erste Feld nach der Abkürzung enthält eine laufende Nummer, falls es mehrere Bedeutungen einer Abkürzung gibt, das nächste Feld enthält die mongolische Auflösung der Abkürzung, anschließend folgen englische und deutsche Übersetzung.

```
\Item{A}{(1)}{antenn}{Aerial}{Antenne}

\Item{A}{(2)}{Armi}{Army}{Armee}

\Item{AA}{(1)}{Aj Axuï}{Economy}{Wirtschaft}

\Item{AAATÄSNMX}{}{Azi Afrikiïn Ard Tümniï Äw Sanaany Nägdliïn Mongolyn Xoroo}
        {Mongolian Committee for Afro-Asian Peoples' Solidarity}
        {Mongolisches Komitee für die Solidarität Afro-Asiatischer Völker}
```

```
[... viel weggelassen...]

\Item{YaÜÄEZ}{}{Yapony Üïldwärqnii̇ Äwläliïn Erönxiï Zöwlöl}
        {Japanese Central Council of Trade Unions}
        {Japanischer Zentralrat der Gewerkschaften}

\Item{YaX}{}{Yawgan Xoroo}{Infantery Regiment}{Infanterieregiment}

\Item{YaC}{}{Yawgan Cäräg}{}{Infanterist}
```

Es ist nun ein Perl-Programm zu schreiben, das unter Benutzung der Datenbankdienste des Betriebssystems eine Datenbank erstellt, in der mit der Abkürzung als Schlüssel der komplette Text aus mongolischer Glosse sowie den Übersetzungen aufgerufen werden kann.

Der Einfachheit halber spalten wir das Programm in zwei Teile, die als separate Skripte gespeichert werden. Das erste Skript erstellt die Datenbank, während das zweite Skript eine einfache Abfrage an der Konsole ermöglicht. Beide Skripte greifen auf die gleiche Datenbank zu.

6.5.1 Skript zur Erzeugung der Datenbank

Das Skript *akuimp.pl* liest den Datentext von der Konsole ein, wozu der Absatzmodus aktiviert wird (`perl -00`). Auf diese Weise wird als Separator zwischen zwei Einträgen der Absatz (also mindestens zwei Zeilenwechsel) angenommen, und es wird unerheblich, ob die einzelnen in geschweiften Klammern stehenden Felder auf der selben Zeile stehen oder nicht. Ein einzelnes, in {} geklammertes Feld darf sich auch über mehrere Zeilen erstrecken.

Jede eingelesene Zeile wird in drei Teile geteilt: die Abkürzung, die (möglicherweise leere) laufende Nummer und den Block der Glossen und Übersetzungen. Getrennt wird mit der Folge *schließende geschweifte Klammer - möglicher Zeilenumbruch, mögliche Kommentare - öffnende geschweifte Klammer*. Damit Glosse und Übersetzungen tatsächlich in einem Feld verbleiben, wird `split()` (siehe dazu den Eintrag in der `perlfunc`-manpage) mit dem optionalen Argument der maximalen Feldzahl versehen:

```
($Abbrev,
 $Number,
 $Gloss)=split(/}[^}{]*{/,$_,3);
```

Da eine Abkürzung unter Umständen mehrere Bedeutungen haben kann, müssen gleiche Abkürzungen durch eine laufende Nummer unterschieden werden, damit sich verschiedene Schlüssel erzeugen lassen. Zu diesem Zweck wird einfach das Nummernfeld an das Abkürzungsfeld angehängt.

Mit dem so erzeugten (und vorher noch kosmetisch bereinigtem) Schlüssel wird nun ein namentlich referenziertes Array befragt und bei Nichtvorhandensen des Schlüssels aufgestockt:

```
if ($Acronyms{$Key} eq "") {    # Glosse einklinken
        $Acronyms{$Key}=$Gloss;
```

Fertig ist das ganze Import-Skript. Hier noch einmal der vollständige Text:[9]

[9] Die Musterdateien auf Disk stimmen nicht unbedingt absolut mit den hier angegebenen Beispieltexten überein. Bei den abgedruckten Versionen sind noch Kommentare ergänzt worden, die in den Dateiversionen nicht enthalten sind.

```
#!/usr/bin/perl -00
dbmopen(%Acronyms,"acronym",0666);      # Für Lesen und Schreiben öffnen!
while (<STDIN>) {
         s/{\\([Ss])h}/\|\\$1h\|/g;
        ($Abbrev,                       # Felder aufteilen
         $Number,
         $Gloss)=split(/}[^}{]*{/,$_,3);

         foreach $wort ($Abbrev,$Gloss) {       # Kosmetik, unerheblich
                $wort=~s/\|\\([Ss])h\|/\{\\$1h\}/g;
         }

        $Abbrev=~s/^\\Item{//;          # Abkürzungstext von
        $Abbrev=~s/[ -].*$//;           # Ballast wie Interpunktion
        $Abbrev=~s/\\?[,.\/]//g;        # und wechselnder Großschreibung
        $Abbrev=~s/./\U$&/g;    # befreien
        $Abbrev=~s/{\\SH}/SH/g;
        $Abbrev=~s/\\SH/SH/g;

        $Key=$Abbrev.$Number;           # Schlüssel bilden

        if ($Acronyms{$Key} eq "") {    # Glosse einklinken
                $Acronyms{$Key}=$Gloss;

dbmclose(%Acronyms);                    # Schotten dicht, gute Nacht.
```

Hinweis: Im obigen Skript werden die veralteten Perl-Aufrufe `dbmopen()` und `dbmclose()` verwendet, die ein Hash direkt an die DBM-Funktionen des Betriebssystems binden. Es wird empfohlen, sich für zukunftsfeste Applikationen an die `tie()`- und `untie()`-Funktionen zu halten, die die Anbindung an fest gespeicherte Dateien universeller gestalten, da der Benutzer in der Wahl der Datenbank nicht mehr eingeschränkt ist. Die Schnittstelle ist jetzt objektorientiert und in das Modul `DB_File` ausgelagert, welches Bestandteil der Standard-Installation von Perl ist (wenigstens auf UNIX-ähnlichen Systemen). Die `DB_File`-manpage enthält eine Reihe von Anwendungsbeispielen. Weitere ausführliche Information ist auch in der `perltie`-manpage enthalten.

6.5.2 Die Datenbankabfrage

Das Skript *akuproc.pl* ist nur geringfügig aufwendiger als das Skript zur Datenbankerzeugung. Es wird eigentlich nur durch Routinen aufgebläht, die das Programm benutzerfreundlicher machen. Von der Konsole wird zunächst eine Abkürzung eingelesen, wobei dem Benutzer freigestellt wird, Umlaute in `"[AOU]`-Notation einzugeben, falls die Tastatur keine Umlaute anbietet.

Im nächsten Schritt wird wieder ein namentlich referenziertes Array mit dem Schlüssel befragt. Findet sich kein Eintrag, so wird der Schlüssel automatisch um eine angehängte laufende Nummer ergänzt, die so lange inkrementiert wird, bis keine Einträge mehr gefunden werden.

```
$nummer=1;                      # Nummer bereitstellen
if ($Acronyms{$Abbreviation} eq "") {
        print "\nSorry, $Abbreviation not found...\n";
        print "Trying numbered abbreviation...\n";
        while ($Acronyms{$Abbreviation."($nummer)"}) {
```

```
                &Ausgabe($Abbreviation."($nummer)");
                $nummer++;
        }
}
```

Jeder gefundene Eintrag wird einer Ausgaberoutine übergeben, die hier recht primitiv gehalten ist; einerseits wird der Array-Aufruf wiederholt, andererseits könnte eine schönere (und einfacher zu schreibende) Ausgabe mit Hilfe des `format`-Befehls erreicht werden. Dies wird sicher dann nötig, wenn mehr als drei Felder in unterschiedlicher Anordnung auszugeben sind.

```
#!/usr/bin/perl                         # Pfad zu Perl

dbmopen(%Acronyms,"acronym",0444);      # Lesendes Öffnen der Datenbank

print "  Acronym: ";                    # Rudimentäres Prompt

while ($Abbreviation=<STDIN>) {
        chomp $Abbreviation;            # Leere Zeile bricht ab (s.  u.)
        ($Abbreviation gt "") || die "Program aborted.";
        $Abbreviation=~s/"A/Ä/g;        # Umlaute bereitstellen
        $Abbreviation=~s/"O/Ö/g;
        $Abbreviation=~s/"U/Ü/g;
        $Abbreviation=~s/"a/ä/g;
        $Abbreviation=~s/"o/ö/g;
        $Abbreviation=~s/"u/ü/g;
        $Abbreviation=~s/([Ss])h/\\|$1h|/g; #

        $nummer=1;                      # Nummer bereitstellen

        if ($Acronyms{$Abbreviation} eq "") {   # Nichts enthalten,
                                                # probiere es mit
                                                # angehängter Nummer
                                                # nochmal... also
                                                # AA(1) und AA(2)
                                                # kommen so zum
                                                # Vorschein...
                                                #
                print "\nSorry, $Abbreviation not found...\n";
                print "Trying numbered abbreviation...\n";
                while ($Acronyms{$Abbreviation."($nummer)"}) {
                        &Ausgabe($Abbreviation."($nummer)");
                        $nummer++;
                }
        }
        else {
                &Ausgabe($Abbreviation);
        }
        print "  Acronym: ";                    # Wieder das Prompt
}
#################################################################
sub Ausgabe {
```

```
        $Feld = shift @_;
        print "\n----------------------------------------\n";
        $Resultat=$Acronyms{$Feld};
        $Resultat=~s/{\\([Ss])h}/$1h/g;
        ($Mong,$Eng,$Deu)=split(/}[^{}]*{/,$Resultat);

        $Deu =~ s/}[^}]*$//;

        print "Mongolian: $Mong\n";
        print "  English: $Eng\n";
        print "   German: $Deu\n\n\n\n";
}
```

6.6 Akronymdatenbank, zweiter Anlauf (CGI)

Interessant wird eine Datenbank, wenn der Inhalt ohne großen Aufwand vielen Benutzern zur Verfügung gestellt werden kann. Perl bringt von Haus aus kein GUI (Graphical User Interface) mit, aber es gibt ja mit den WWW-Browsern einen betriebssystemübergreifenden Quasi-Standard für die Darstellung der verschiedensten Daten. Nahezu alle Browser sind in der Lage, *Formulare* darzustellen, die dem Benutzer auch Texteingaben ermöglichen. Für Perl gibt es das Modul CGI, das Objekte zur Gestaltung von WWW-Seiten anbietet. Die `CGI`-manpage ist recht umfangreich; außerdem gibt es noch eine Reihe von Beispielskripten, die bei der Installation automatisch in ein passendes html-Verzeichnis gestellt werden. Die Installation eines WWW-Servers wird hier nicht weiter erklärt, da sie an anderer Stelle ausführlich beschrieben ist.[10]

Im Gegensatz zur Kommandozeilenversion der Abkürzungsdatenbank verfügt die WWW-Version über ein Fenster, in das der Suchbegriff eingegeben wird; die Ausgaben erscheinen auf der gleichen Seite. Der folgende Code-Ausschnitt ist unvollständig; es werden nur die Erweiterungen gezeigt, die für die Verwendung als CGI-Skript notwendig sind.

```
#!/usr/bin/perl

use CGI;

dbmopen(%Acronyms,"acronym",0444);
################################################################
#
# Wir beginnen mit der HTML-Ausgabe
#
$query = new CGI;

print $query->header;
print $query->start_html('3. Access the database');
# [...Auslassung, nicht funktionsbeeinträchtigend...]
print $query->startform.
                q{<TABLE>
                <TR><TD ALIGN=RIGHT>Abbreviation:</TD>
                    <TD ALIGN=LEFT>}.$query->textfield(-name=>'Abbreviation',
                               -default=>'',
```

[10]Es gibt eine reichliche Literaturauswahl über die CGI-Programmierung; für Linux-Systeme bietet sich auch das WWW-mSQL-HOWTO an, das leider in einigen Punkten nicht den aktuellsten Versionsstand widerspiegelt.

```
                                -columns=>20).q{</TD><TD>};
print   $query->submit(-name=>'Knopf',
                        -value=>'Query').
        $query->defaults('Clear Form').q{</TD></TR></TABLE>}.
        $query->endform;
#       $query->dump.           # Wenn's Probleme gibt...

&Abfragen if ($query->param('Knopf') eq 'Query');

print $query->end_html;

sub Abfragen {
        $Abbreviation=$query->param('Abbreviation');
        $MailSubject=$Abbreviation;

# [... Auslassungen...]

        if ($Acronyms{$Abbreviation} eq "") {   # Nichts enthalten;
        }                                       # Fehlermeldung
        else {                                  # nicht vergessen!
                &Ausgabe($Abbreviation);
        }
}
################################################################
sub Ausgabe {
# [... Auslassungen...]
        ($Abbrev,$Mong,$Eng,$Deu)=split(/}[^{}]*{/,$Resultat);

        $Deu =~ s/}[^}]*$//;    # Letzten Wert von Begrenzern befreien

# Eigentliche Ausgabe als HTML-Tabelle
#
        print q{<HR><TABLE>};
        print "<TR><TD ALIGN=RIGHT>Acronym:</TD><TD ALIGN=LEFT>$Abbrev</TD></TR>";
        print "<TR><TD ALIGN=RIGHT>Mongolian:</TD><TD ALIGN=LEFT>$Mong</TD></TR>";
        print "<TR><TD ALIGN=RIGHT>English:</TD><TD ALIGN=LEFT>$Eng</TD></TR>";
        print "<TR><TD ALIGN=RIGHT>German:</TD><TD ALIGN=LEFT>$Deu</TD></TR>";
        print q{</TABLE>};

# Einbettung einer Benachrichtigung des Datenbankverantwortlichen
# bei unvollständiger Erklärung und Übersetzung
#
        if ($Mong eq "" || $Eng eq "" || $Deu eq "") {
        print qq{Please mail corrections and addenda to the
        <A HREF="mailto:infomong\@zedat.fu-berlin.de?subject=acrobase
         entry: [$Feld]?">editor</A>
                at <TT>infomong\@zedat.fu-berlin.de</TT>};
        }
}
```

6.7 Aufgabe: Entwicklung einer interaktiven Perl-Shell

Perl ist als mächtige Universalsprache bekannt, die theoretisch auch als Shell verwendet werden kann. Bereits das Übungsprogramm (siehe 6.1 ()) hat ja gezeigt, wie es mit wenigen Zeilen möglich ist, Perl-Texte zu schreiben, die sich selbst vor provozierten Abstürzen retten können. Das Übungsprogramm hat jedoch in der angegebenen Form einen entscheidenden Nachteil: Jeder am Prompt übergebene Konstrukt muß syntaktisch vollständig sein. Ein Ausdruck, der sich über mehrere Zeilen erstreckt, ist nicht zulässig.

Aufgabe: Testen Sie das Übungsprogramm, indem Sie nur einen Blockanfang (geöffnete geschweifte Klammer {) eingeben. Beobachten Sie die Reaktion. Erweitern Sie das Programm so, daß es auch in der Lage ist, mehrzeilige Konstrukte zu verarbeiten. Versetzen Sie sich dazu in die Lage eines Shell-Programmierers, der eine Schleife mit `for i in (*)` beginnt, auf der nächsten Zeile ein `do` eingibt, dann auf der nächsten Zeile die nötigen Kommandos eingibt und das Ganze mit `done` abschließt. Ergänzen Sie das Programm nach erfolgreicher Fertigstellung um eine Editier- und Aufrufmöglichkeit für bereits eingegebene Zeilen.

Die interessantesten Lösungen werden an dieser Stelle vorgestellt.

7 Perl und „die Welt da draußen”

Eine Einführung in Perl sollte nicht ohne einen Hinweis auf die außergewöhnlich sozialen und polyglotten Eigenschaften von Perl enden. Perl fühlt sich nicht nur auf vielen Systemen zu Hause, sondern kommuniziert auch problemlos mit anderen Programmiersprachen, ganz abgesehen von der Tatsache, daß es in sich bereits die Obermenge zahlreicher Konstrukte anderer Programmiersprachen darstellt. Neben der Möglichkeit, C-Programme in Perl einzubinden, gibt es auch die Möglichkeit, Perl von C aus aufzurufen (siehe die `perlcall`-manpage). Außerdem wurde Perl um komplette Sprachen erweitert bzw. andere Programme wurden um Perl erweitert; so gibt es Tkperl (aka Perl/Tk), das die GUI-Funktionen von Tk/Tcl anbietet, modperl für Webserver (ein Apache mit integriertem Perl) und nviperl, den klassischen Editor vi mit integriertem Perl. Hier folgt eine kurze Vorstellung der wichtigsten Merkmale von Perl/Tk und modperl.

7.1 Perl/Tk

Perl/Tk stellt Module zur Verfügung, die es gestatten, aus Perl heraus die leistungsfähige graphische Benutzerschnittstelle Tk/Tcl aufzurufen. Dazu ist erforderlich, daß Tk/Tcl bereits auf dem Rechner installiert ist. Anschließend muß das Modul Tk installiert werden (zu finden unter dem Eintrag *User_Interfaces* im Modulverzeichnis auf CPAN). Daraufhin sind die Erweiterungen sofort nutzbar. Da es Tk/Tcl (ebenso wie ja auch Perl) für eine Reihe verschiedener Rechnertypen kostenlos gibt, es mit Perl/Tk möglich, Anwendungen mit graphischer Benutzerschnittstelle zu schreiben, die sowohl unter X als auch unter Windows (oder auf Apple-Rechnern) funktionieren.

Ein einfaches Beispiel findet sich in *Perl in a Nutshell* im Kapitel Perl/Tk auf Seite 493:

```
#!/usr/bin/perl -w
use Tk;
my $mw = MainWindow->new;
$mw->Button(-text => "Hello World!", -command =>sub{exit})->pack;
MainLoop;
```

Das Nußschalen-Buch widmet den grundlegenden Eigenschaften von Perl/Tk rund fünfzig Seiten und ist daher hervorragend als Einführung geeignet.

Das folgende Code-Fragment zeigt (bis auf den Mustertext) den gesamten funktionstragenden Inhalt von `/usr/lib/perl5/site_perl/Tk/demos/widget_lib/texts.pl`

```
# texts.pl

use vars qw/$TOP/;

sub texts {

    # Create a top-level window that displays a basic text widget.

    my ($demo) = @_;
    $TOP = $MW->WidgetDemo(
        -name     => $demo,
        -text     => '',
        -title    => 'Text Demonstration - Basic Facilities',
        -iconname => 'texts',
    );

    my $t = $TOP->Scrolled(qw/Text -relief sunken -borderwidth 2 -setgrid true
                           -height 30 -scrollbars e/);
    $t->pack(qw/-expand yes -fill both/);

    $t->insert('0.0', 'This window is a text widget.  It displays
    # [... hier steht der dargestellte Text des Originalbeispiels ...]
    [...] visible.');

    $t->mark(qw/set insert 0.0/);

} # end texts

1;
```

7.2 modperl

Web-Server, die für jedes CGI-Programm den Perl-Interpreter aufrufen müssen, erzeugen Lastspitzen, die den Rechner schnell an die Grenzen seiner Leistungsfähigkeit bringen können, wenn in kurzer Zeit sehr viele Anfragen nach CGI-Programmen eintreffen. Unter solchen Umständen macht sich störend bemerkbar, daß für jedes Programm der komplette Zyklus aus Perl-Aufruf und Skript-Kompilierung neu durchlaufen werden muß. Mit dem Modul modperl wird dieses Problem umgangen, indem ein kompletter Perl-Interpreter in den bekannten Web-Server Apache (httpd) eingebunden wird. Auf diese Weise steht Perl ständig im Arbeitsspeicher zur Verfügung und muß nicht für jedes CGI-Programm neu aufgerufen werden.

8 Weiterführende Links

8.1 Literatur

Das Angebot an Literatur zu Perl übersteigt die oben genannten Titel bei weitem; an dieser Stelle erscheint daher kein statisches Literaturverzeichnis, sondern ein Verweis auf den Katalog von Lehmanns Buchhandlung. Geben Sie als Suchbegriff *Perl* ein und setzen Sie die Option *Ganze Wörter*, damit nicht auch Perlenstickereien gefunden werden. Die Buchhandlung ist erreichbar unter *www.lob.de*.

8.2 Perl auf Windows-basierenden Systemen

Von Wulf-Bernd Göhmann stammt die Materialsammlung zum Thema *Perl auf DOS- und Windows-Systemen*. Unter *http://userpage.fu-berlin.de/˜wbg* stehen nicht nur Bezugsquellen von Perl, sondern auch Hinweise auf GNU-Utilities (flex, bison, fileutils, findutils), die standardmäßig auf allen UNIX-Rechnern zu finden sind und auch auf Rechnern anderer Bauart die Arbeit sehr erleichtern können, sofern sie überhaupt vorhanden sind.

Von Wolfgang Lipp stammt eine weitere Seite mit Hinweisen zu Perl in Windows-Umgebungen. Sie ist unter *http://www.snafu.de/˜castor* zu finden und stellt neben Software-Information auch Angaben über Bestellmöglichkeiten amerikanischer Bücher zu Perl bereit.