Perl - Guide

Stichwortverzeichnis Suchen Perlboard.de

Inhaltsverzeichnis

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Erste Schritte

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Eine Einführung in Perl

Heute beginnen wir mit ein paar Grundlagen, die Ihnen den Start in die Arbeit mit Perl erleichtern sollen. Die Lektion von heute ist kurz; zuerst ein wenig Hintergrundwissen, und dann legen wir los und lassen Perl mit ein paar einfachen Skripts laufen. Insbesondere sind unsere Themen heute:

- Ein paar Hintergrundinformationen zu Perl: was Perl ist, wer es erfunden hat, wer heute damit arbeitet
- Warum Sie Perl überhaupt lernen sollten (andere Gründe als »mein Boß hat mich dazu verdonnert«)
- Etwas Code, damit Sie eine Vorstellung bekommen, wie Perl geschrieben und ausgeführt wird (und auf die restlichen Lektionen dieser Woche vorbereitet sind)

Machen wir kein großes Aufheben und fangen wir einfach an. Woche eins, Tag eins, Abschnitt eins - und los!

Was ist Perl, und wo kommt es her?

Perl hat nichts mit Perlen zu tun. Perl ist eigentlich eine Abkürzung. Es steht für ***P****ractical* ***E****xtraction and* ***R****eport* *L*anguage. Das könnte man mit »Praktische Extraktions- und Report-Sprache« übersetzen, wenn es nicht so schrecklich klänge. Denn es ist eine ziemlich treffende Beschreibung von Perls besonderen Stärken: ***Extraktion*** für das Anschauen von Dateien und das ***Herausziehen*** wichtiger Teile (zum Beispiel aktueller Daten aus einer HTML-Datei oder Benutzer- oder Host- Namen aus einer Netzwerk-Log-Datei); ***Report*** für das Generieren von Ausgaben und ***Reports*** über die gefundenen Informationen. Die Sprache ist ***praktisch***, weil es viel leichter ist, diese Art von Programmen in Perl zu schreiben als in einer Sprache wie C. Aber, ob englisch oder deutsch, niemand klammert sich heute mehr an die Abkürzung. Perl ist einfach Perl.

Perl wurde 1987 von Larry Wall entwickelt, einem Amerikaner, der schon damals in der Unix-Software-Welt ziemlich bekannt dafür war, dass er sowohl das frei erhältliche Programm ***patch*** als auch den Usenet Newsreader ***rn*** geschrieben hatte. Es geht die Geschichte um, dass Larry gerade an einer Aufgabe arbeitete, für die das Unix-Programm ***awk*** (das zu dieser Zeit populäre Extraktions- und Report-Programm) nicht ausreichte, ihm aber klar wurde, dass die Lösung seiner Aufgabe in einer Sprache wie C eine Menge Arbeit bedeuten würde. So erfand er eine eigene Skriptsprache - Perl. Er borgte sich dabei einiges aus diversen anderen Unixtools und -sprachen wie ***sed***, ***grep***, ***awk***, Shell-Skripts und, ja, C.

Außerdem wurde Perl, wie ***patch*** und ***rn***, frei an die Unix-Gemeinschaft gegeben. Die Unix-Gemeinschaft griff Perl dankbar auf. Jahrelang war Perl die bevorzugte Sprache für Unix-Systemadministratoren und andere Unix-Programmierer. Sie brauchten eine flexible, schnell zu programmierende Sprache zur Lösung von Aufgaben, die für Shell- Skripting zu komplex waren und deren Lösung in Sprachen wie C einen deutlichen Mehraufwand bedeutet hätte. Es lag an seiner Popularität als Unix-Sprache, dass Perl sich auch als Web-Sprache zum Erstellen von CGI-Skripts durchsetzte - so gut wie alle Webserver liefen früher auf Unixsystemen. Mit CGI (***Common Gateway Interface***) konnte - und kann - man mit Formularen oder anderen »webseitigen« Eingaben interaktiv Programme und Skripts auf dem Webserver ablaufen lassen. Perl paßte wunderbar in diese Nische, und als in den letzten Jahren das World Wide Web und CGI immer populärer wurden, wuchs auch das Interesse an Perl enorm. Selbst als Webserver dann auch auf anderen Systemen liefen, zog Perl mit ihnen zusammen um und blieb für diese Anwendung weiterhin populär.

Perl ist eigentlich das »Kind« von Larry Wall, der sich zunächst allein um Weiterentwicklung und Updates kümmerte. Mit dem plötzlichen Anstieg von Perls Beliebtheit kümmert sich jetzt eine engverzahnte Gruppe von freiwilligen Programmierern um diese Aufgaben. Diese Programmierer, darunter auch Larry Wall selbst, arbeiten weiter am Quellcode von Perl, portieren ihn auf Nicht-Unix- Plattformen, koordinieren Bugfixes und geben die »Standard« Perl Releases heraus (in der Datei ***Changes*** der Standard-Perl-Distribution finden Sie eine Namensliste dieser »Perl-Hauptdarsteller«). Perl gehört keiner einzelnen Organisation. Wie die GNU- Tools (***GNU-Emacs, GCC,***

GDB etc.) und das Betriebssystem Linux wird Perl auf einer freiwilligen Goodwill-Basis entwickelt und gemanagt. Es ist ebenfalls kostenlos. Alles, was Sie tun müssen, ist, es zu laden und zu installieren.

Die aktuelle Perl-Version ist Perl 5. Das ist auch die Version, die in diesem Buch behandelt wird. In allen Versionsnummern folgen auf die 5 ein Punkt und viele Zahlen. Das sind die Nummern der ***Patches****, kleineren »Neuauflagen«, die meistens in der Originalfassung gefundene Bugs beheben (obwohl sich manchmal auch einige neue Features einschleichen). Für die Unix- und Windows-Plattform war, während ich dieses Buch schrieb, Perl 5.005 aktuell, für den Macintosh lag »nur« Version 5.004 vor. Diese kleineren Nummern der Releases haben sich, wenn Sie all das hier lesen, mittlerweile möglicherweise geändert, an allen Perl-Portierungen wird weiter gearbeitet (außerdem können jederzeit separate weniger stabile Betaversionen und auch eine neue »offizielle Version« erscheinen). Sie müssen aber nicht jedes kleine Update mitmachen; wirklich wichtig ist vor allem, dass Sie Perl 5 installiert haben. Seit Perl 5.005 unterstützt der Kern auch Windows (anders als bei den verschiedenen oft inkompatiblen Portierungen, die es vorher gab). Wenn Sie mit Windows und eines älteren Perl-Release arbeiten, empfehle ich Ihnen dringend ein Update auf die neuere Version. Der neue Kern behebt viele Bugs, gleicht Unterschiede aus und erleichtert die Arbeit ungemein.*

Die Beispiele in diesem Buch wurden mit der Unix-Version 5.005_02 geschrieben und mit ActiveState's Perl for Windows build 508 und MacPerl 5.2.0r4 (einer 5.004-Portierung) getestet. In den Anhängen C, D und E erfahren Sie, wie man diese Perl-Versionen bekommt und installiert.

Warum Perl lernen?

Die Zahl der heute auf dem Markt erhältlichen Programmiersprachen ist enorm. Und wie es aussieht, behauptet jede von ihnen, sie könne all Ihre Probleme in der halben Zeit und zu einem Viertel der Kosten lösen und dabei noch Kaffee kochen .... Warum also Perl lernen und nicht eher eine dieser anderen Sprachen?

Das beste Argument lautet: Unterschiedliche Werkzeuge eignen sich unterschiedlich gut für unterschiedliche Aufgaben. Perl bietet Ihnen für eine Menge allgemeiner Aufgaben viele besonders gute Werkzeuge. Doch es gibt darüber hinaus noch ein paar mehr gute Gründe, Perl zu lernen und zu nutzen.

Perl ist nützlich

Perl ist wahrscheinlich die richtige Sprache für Sie, wenn eine der folgenden Aussagen auf Sie zutrifft:

- Sie sind ein Systemadministrator auf der Suche nach einer Allzweck- Skriptsprache
- Als Web-Administrator stehen Sie vor einem Dutzend CGI-Programmen, die Ihre Webdesigner von Ihnen geschrieben haben wollen.
- Sie sind ein recht versierter Unix-Anwender und möchten Ihr Wissen erweitern.
- Sie sind Programmierer und möchten schnell einen Prototyp für ein komplizierteres Programm schreiben.
- Sie möchten einfach nur eine Sprache, mit der Sie alles mögliche anstellen können.

Sie können mit Perl richtig sinnvolle Arbeit erledigen, und das ziemlich schnell.

Perl braucht keinen Software-Schnickschnack

Wenn Sie mit Perl arbeiten möchten, brauchen Sie kein hochglanzverpacktes Programm zu kaufen. Sie brauchen keinen Perl-Compiler oder integrierte Entwicklungsumgebung. Sie brauchen keinen Browser, der Perl unterstützt, und keinen Computer, der »Perl inside« sagt. Sie brauchen nur eines: den Standard-Perl- Interpreter. Und den bekommen Sie umsonst, ganz einfach, indem Sie sich ihn herunterladen. Wenn Sie einen Unix-Shell-Account bei einem Internet Service Provider haben, haben Sie ihn vielleicht schon.

Okay, ***eine*** andere Sache brauchen Sie noch: einen Texteditor, mit dem Sie Ihre Perl- Skripts schreiben. Irgendeiner ist aber bei jedem System, auf dem Perl läuft, schon dabei, also sind Sie nach wie vor auf der sicheren Seite.

Perl ist schnell zu programmieren

Perl ist eine Skriptsprache. Das bedeutet, dass Ihre Perl-Skripts reine Textdateien sind, die sofort ausgeführt werden, wenn Perl sie ablaufen läßt. Sie brauchen keinen Compiler, der Ihren Code wie bei einer Sprache wie C oder Java in ein anderes Format, etwa eine ausführbare oder Bytecode-Datei, verwandelt. Deswegen können Sie mit Perl schneller und einfacher als in C Ihre ersten Programme laufen lassen, debuggen und verändern.

Skriptsprachen werden in der Computersprache oft interpretierte Sprachen genannt. Obwohl Perl eine interpretierte Sprache zu sein scheint, weil seine Programme Skripts sind und der Perl-Interpreter diese Skripts ausführt, ist Perl in Wahrheit beides, eine kompilierte und eine interpretierte Sprache.

Wenn Sie ein Perl-Skript starten, liest Perl das ganze Skript ein, kompiliert es und führt dann das Ergebnis aus. Obwohl Perl dadurch sehr interpretiert aussieht und Sie Perl-Skripte ändern und neu ausführen können wie bei einer interpretierten Sprache, haben Sie in einem gewissen Maße auch Kontrolle über den Kompilierungsprozeß. Perl läuft auch schneller als eine interpretierte Sprache (obwohl es nicht ganz so schnell ist wie eine kompilierte).

Perl ist portierbar

Da Perl eine Skriptsprache ist, gilt: Ein Perl-Skript ist ein Perl-Skript, ist ein Perl- Skript, ganz egal auf welcher Plattform Sie es laufen lassen. Zwar gibt es Unterschiede in Perl auf verschiedenen Plattformen und auch ein paar Features, die nur auf bestimmten Plattformen funktionieren (ich werde jeweils darauf hinweisen), doch in den meisten Fällen kann man ein Skript von einer Plattform zur anderen verschieben, ohne es irgendwie zu ändern - ohne es aufwendig auf ein neues Betriebssystem zu portieren, ohne noch einmal den Quelltext kompilieren zu müssen.

Perl ist leistungsfähig

In Perl sind viele ziemlich komplexe Unix-Tools eingeflossen. Und es hat alle Features, die Sie in einer High-Level-Sprache erwarten (und viele, die Sie nicht erwarten). Fast alles, was Sie in einer anspruchsvollen Sprache wie C machen können, geht auch in Perl, obwohl es natürlich Aufgaben gibt, für die C besser als Perl geeignet ist und umgekehrt. Sie können in Perl strukturiert programmieren. Sie können in Perl fortgeschrittene Datenstrukturen bauen. Sie können in Perl objektorientiert programmieren.

Wenn Ihnen Perl allein nicht gut genug ist, gibt es außerdem umfassende Archive mit vielerlei Tools und Bibliotheken (***Module*** genannt) für diverse häufige Aufgaben. Module für Datenbanken, Interaktivität, Netzwerke, Verschlüsselung, Verbindungen zu anderen Sprachen - so ziemlich alles Erdenkliche ist in diesen Archiven erhältlich. Wenn Sie denken: »Dies und jenes habe ich jetzt mit Perl vor«, hatte in vielen Fällen schon jemand dieselbe Idee und hat seinen Code über die Archive zur freien Verfügung gestellt. Perls kooperative Natur hat zur Folge, dass es eine enorme Menge von Ressourcen gibt, die Sie für sich nutzen können.

Perl ist flexibel

Eins der Mottos von Perl ist: ***There's more than one way to do it*** (»Es gibt mehr als einen Weg, Dinge zu tun.«). Viele Wege führen zum Ziel. Darauf verweisen Perl-Leute oft mit der Abkürzung ***TMTOWTDI***. Merken Sie sich diesen Satz, denn ich werde im ganzen Buch immer wieder darauf zurückkommen. Die Philosophie beim Entwurf von Perl war, dass unterschiedliche Programmierer auch unterschiedliche Methoden haben, sich Problemen zu nähern und sie zu lösen. Deshalb stellt Ihnen Perl, anstatt von Ihnen zu verlangen, Ihre Denkweise an einen kleinen Satz von Befehlen und syntaktischen Strukturen anzupassen, eine enorme Anzahl von Konstruktionen und Abkürzungen zur Verfügung - von denen viele letztlich genau das gleiche machen wie andere, nur auf eine etwas andere Art und Weise.

Das macht Perl zu einer sehr umfangreichen, komplexen und komplizierten Sprache (als ich das erste Mal auf Perl traf, als junge, idealistische Programmiererin, war meine erste Reaktion: »Es ist so häßlich!«). Aber der Umfang von Perl und die Vielzahl seiner Möglichkeiten machen es auch extrem flexibel und angenehm im Gebrauch. Kompliziert kann es eigentlich nur werden, wenn Sie Perl-Skripts von anderen ***lesen*** müssen. Selbst eins zu schreiben, ist dagegen ziemlich einfach, denn es liegt ganz bei Ihnen, ob Sie ein ganz geradliniges, C-artiges Skript

schreiben - nur einige der vielen Perl-Abkürzungen verwenden und dabei komplett lesbar bleiben - oder ob Sie sich so schwer auf Perl-Seiteneffekte und versteckte Features stützen, dass Sie alle, die versuchen, Ihr Skript zu lesen, damit in den Wahnsinn oder zur Verzweiflung treiben (manchmal ist es recht lustig, auf diese Art einen ruhigen Nachmittag zu verbringen). Sie entscheiden, wie Sie Perl einsetzen möchten - Sie müssen nicht Ihre Denkweise ändern, um sich der Sprache anzupassen. Manchmal denke ich immer noch, Perl ist ja so häßlich - aber ich finde es auch wirklich cool, und es macht Spaß, damit zu arbeiten. Das kann ich von C nicht sagen, das zwar konsistenter und eleganter ist, aber meiner Meinung nach viel mehr Schinderei mit sich bringt.

Perl ist leicht zu lernen

Trotz des Umfangs und der Komplexität von Perl, trotz seiner zahllosen Features ist es nicht schwer zu lernen. Und zwar genau ***wegen*** der vielen verschiedenen Möglichkeiten, ein und dasselbe Problem zu lösen: Es gibt immer auch eine einfache. ***Alles*** von Perl zu beherrschen, kann ziemlich einschüchternd sein (und nur eine Handvoll Leute können von sich behaupten, wirklich jede Nuance der Sprache zu kennen). Aber die einfachen Teile von Perl sind tatsächlich sehr einfach (insbesondere wenn Sie schon etwas Programmiererfahrung haben), und Sie können sehr schnell ausreichend Perl lernen, um es sinnvoll einzusetzen. Sie gewinnen damit vielleicht keine Schönheitswettbewerbe, aber es wird funktionieren. Wenn Sie dann Fortschritte machen und vertrauter mit der Sprache werden, können Sie mehr und mehr Features und Abkürzungen hinzufügen und dann ganz besonders schön oder elegant oder kryptisch kurz schreiben, ganz wie Sie möchten.

Da draußen ist die Perl-Gemeinde

Entwicklung und Support von Perl finden jetzt seit etwa zehn Jahren auf freiwilliger Basis statt. Rund um die Welt widmen Programmierer ihre Zeit der Weiterentwicklung von Sprache und zusätzlichen Tools. Auf der einen Seite sieht das vielleicht sehr anarchistisch aus - da ist keine Firma verantwortlich zu machen, wenn etwas schiefgeht, keine Support-Nummer, die man anrufen kann, wenn man es nicht zum Laufen bringt. Auf der anderen Seite stellt die Perl-Gemeinschaft mittlerweile in so enormem Umfang hilfreiche Bibliotheken, bergeweise Dokumentation und FAQs (Frequently Asked Questions - häufig gestellte Fragen und die Antworten darauf) zur Verfügung, dass Sie die meisten Ihrer Fragen auch beantwortet und Hilfe bekommen, wenn Sie welche brauchen. Das werden Sie mit C nicht so haben, wo Sie normalerweise ein oder zwei Bücher kaufen oder sieben oder zwanzig und dann immer noch auf sich gestellt sind. Einer der größten Vorteile von Perl sind die Menschen, die damit arbeiten - und ihre außergewöhnliche Hilfsbereitschaft. Larry Wall hat sich übrigens immer gewünscht, dass die Perl-Gemeinde wie ein kleines Stück Himmel funktioniert. Bis jetzt ist ihm dieser Wunsch erfüllt worden, und auch Sie können nach der Lektüre dieses Buches dazu beitragen, dass das so bleibt. Bitte, lassen auch Sie sich von diesem Modell anstecken.

Erste Schritte zum eigenen Perl-Programm

Um Perl zu verwenden - damit zu programmieren und Perl-Skripts auszuführen -, müssen Sie es auf Ihrem System installiert haben. Das ist zum Glück nicht schwierig. Wie ich bereits erwähnt habe, können Sie Perl frei herunterladen. Es kostet Sie lediglich etwas Verbindungszeit mit dem Internet.

Wenn Sie Perl noch nicht installiert haben oder sich nicht sicher sind, starten Sie als erstes Ihren Webbrowser und gehen auf die Site `http://www.perl.com/`. Speichern Sie diese Adresse zu Ihren Bookmarks oder Favoriten: Sie ist die zentrale Sammelstelle für alles, was mit Perl zu tun hat. Sie finden dort nicht nur das Perl-Interpreter-Paket selbst, sondern auch das ***Comprehensive Perl Archive Network*** (»umfassendes Perl- Archivnetz«, abgekürzt ***CPAN***), Perl-News, Dokumentationen, Konferenzen und Wettbewerbe, Perl-Witze - so ziemlich alles, was Sie sich nur vorstellen können.

Auf der Seite `http://www.perl.com/latest.html` finden Sie aktuelle Informationen zu den neuesten Perl-Versionen für jede Plattform. Hier - und in den Anleitungen aus dem Perl-Paket selbst - steht, wie Sie Perl herunterladen und auf Ihrem System installieren.

Genauere Informationen bieten Ihnen die Anhänge C, D und E am Ende des Buchs. Dort ist im einzelnen erklärt, wie Sie Perl für Unix, Windows bzw. Macintosh installieren.

Bevor Sie weitermachen, sollten Sie Perl auf Ihrem System installiert und laufen haben.

Ein Beispiel: Das allgegenwärtige »*Hallo Welt*«

Eine alte Tradition in der Programmierwelt ist das »Hallo Welt«-Programm. Wenn ein Programmierer eine neue Sprache lernt, schreibt er als erstes ein Programm, das einfach nur »Hallo Welt!« auf den Bildschirm ausgibt. Es liegt mir fern, die erste zu sein, die von dieser Tradition abweicht. So ist das erste Perl-Skript, mit dem wir uns in diesem Buch befassen, schon ein Klassiker. In diesem Abschnitt schreiben Sie das »***Hallo Welt***«-Skript und führen es aus, und ich werde Ihnen dann erklären, was dort passiert.

Hallo Welt schreiben

Sie brauchen einen Texteditor. Keine professionelle Textverarbeitung-Software, sondern einfach einen Editor: Unter Unix tun es ***emacs***, ***vi*** oder ***pico***, in Windows das eingebaute ***Notepad***, die Shareware-Programme ***TextPad*** oder ***UltraEdit***; auf dem Mac können Sie den simplen in MacPerl eingebauten Texteditor nehmen oder das Mac-eigene ***SimpleText*** oder zum Beispiel die Shareware-Programme ***BBEdit*** oder ***Alpha***. Welchen Texteditor auch immer - starten Sie ihn, und tippen Sie die zwei Zeilen aus Listing 1.1 ein (na gut, drei, wenn Sie die leere in der Mitte mitzählen).

Schreiben Sie die Nummern und Doppelpunkte am Zeilenanfang nicht mit ab: das sind Zeilennummern. In allen Listings dieses Buchs stehen sie dort nur, damit ich Ihnen Zeile für Zeile erzählen kann, was im Skript geschieht.

Listing 1.1: Das Skript hallo.pl.

```
1: #!/usr/bin/perl -w
2:
3: print "Hallo Welt!\n";
```

Wenn Sie auf Unix arbeiten, achten Sie unbedingt auf die erste Zeile (umgangssprachlich »***sh'bang***« oder »***shebang***-Zeile« genannt: »***sh***'«für den ***Sh***arp, das Kreuz, und »***bang***« für das Ausrufezeichen). Diese Zeile sagt Unix, mit welchem Programm es das Skript ausführen soll. Der einzige Haken dabei ist, dass die Pfadangabe in dieser Zeile dem korrekten Pfad zum Perl-Interpreter entsprechen muss. Wenn Sie ihn in ***/usr/local/bin/perl*** oder woanders in Ihrem System installiert haben, fügen Sie Ihren Pfad anstelle des Pfadnamens in Zeile 1 im Listing 1.1 ein.

Wenn Sie mit Windows oder Mac arbeiten, ist diese ***Shebang***-Zeile normalerweise überhaupt nicht nötig. Sie stört aber auch nicht: durch das # am Anfang wird sie von Perl als Kommentar betrachtet und deshalb auf anderen Plattformen als Unix schlichtweg ignoriert.

Es ist sogar eine gute Idee, sich die Shebang-Zeile anzugewöhnen, auch wenn es nicht unbedingt notwendig ist - insbesondere, wenn Ihre Perl-Programme eventuell einmal auf Unix laufen sollen

*Wenn Sie vorhaben, Perl unter Windows für Web-CGI-Skripts einzusetzen, verlangen manche Webserver (wie z.B. Apache) eine **Shebang**-Zeile in Ihrem Perl-Skript (wenn auch eine im Windows-Stil, mit einem C: am Anfang). Noch ein Grund, sich das anzugewöhnen.*

Speichern Sie jetzt Ihr Skript als, sagen wir, ***hallo.pl***. Eigentlich können Sie die Datei nennen, wie Sie wollen, mit oder ohne die .pl-Erweiterung. Unter Windows sollten Sie das ***.pl*** aber lieber verwenden. Windows hat gern für jedes Programm eine eigene Dateinamensendung. Auf Unix-Rechnern sollten Sie Ihrem Skript natürlich nicht den Namen eines Unix-Befehls oder -Tools geben, nennen Sie es zum Beispiel nicht ***test*** - viele Unix-Varianten haben eine Routine mit diesem Namen, die dann anstatt Ihres Skripts ausgeführt würde.

Ihnen ist nicht nach Tippen? Alle Skripts aus diesem Buch finden Sie sowohl auf der Begleit-CD zu diesem Buch als auch auf der Webseite zu diesem Buch unter `http://www.typerl.com`. Sie können die dortigen Versionen verwenden, anstatt sie mühsam einzutippen. Allerdings lernt man meistens

viel besser, wie Perl-Skripts arbeiten, wenn man sie selbst eingibt (und dann die Fehler korrigieren muss, die dabei unvermeidbar auftauchen). Sie sollten das zumindest mit diesen ersten paar Skripts ausprobieren.

Hallo Welt ausführen

Der nächste Schritt ist, Perl wirklich einzusetzen und Ihr Skript auszuführen.

Auf Unix müssen Sie zuerst Ihr Skript ausführbar machen und dann einfach den Namen des Skripts in die Kommandozeile eingeben, etwa so (das Fettgedruckte ist, was Sie wirklich eintippen, das Prozentzeichen ist der Prompt):

```
% chmod +x hallo.pl
% hallo.pl
```

Abhängig von Ihrer Pfadvariablen müssen Sie beim Aufruf des Skripts eventuell das Verzeichnis mit eingeben, in dem es sich befindet, zum Beispiel:

```
% ./hallo.pl
```

Unter Windows starten Sie die MS-DOS-Eingabeaufforderung (oder einen anderen DOS-Prompt, wenn Sie mehrere haben) und tippen `perl -w` und den Namen des Skripts ein:

```
C:\perl\> perl -w hallo.pl
```

Unter Windows 95 müssen Sie jedesmal den vollen Perl-Befehl eingeben.

Wenn Sie mit Windows NT arbeiten und Perl so eingerichtet haben, dass ***.pl***-Dateien mit Perl verknüpft sind (wie in Anhang B beschrieben), dann brauchen Sie nur den Namen des Skripts eingeben:

```
C:\perl\> hallo.pl
```

In MacPerl wählen Sie zuerst ***Compiler Warnings*** im **Skript**-Menü und dann den Eintrag **Run Skript** aus dem **Skript**-Menü, um das Skript auszuführen.

Auf jeder Plattform sollten Sie jetzt den Satz »Hallo, Welt!« auf dem Bildschirm sehen.

Falls Sie die MPW-Version von MacPerl benutzen, können Sie Perl von der MPW-Kommandozeile starten. Dann gelten alle Optionen, die ich das ganze Buch hindurch beschreibe, als würden Sie die Unix-Version von Perl ausführen. In diesem Buch werde ich mich aber in erster Linie auf die Stand-alone-Version von MacPerl beziehen.

Was tun, wenn es nicht läuft

Was ist, wenn da kein »Hallo, Welt!« auf Ihrem Monitor steht? Vielleicht erhalten Sie statt dessen eine Perl-Fehlermeldung (zum Beispiel ***Can't find string terminator,*** »Kann das Ende der Zeichenkette nicht finden«). Prüfen Sie noch einmal Ihr Skript und stellen Sie sicher, dass Sie es korrekt eingetippt haben und dass alle Anführungszeichen, am Anfang und am Ende, richtig sind. Die erste Zeile beginnt mit einem Rautenzeichen #.

Wenn Sie die Meldung ***File not found*** (»Datei nicht gefunden«) oder ***Can't open perl script*** (»Kann Perl-Skript nicht öffnen«) erhalten, achten Sie darauf, dass Sie sich im gleichen Verzeichnis befinden wie Ihr Perl-Skript und

dass Sie den Dateinamen exakt so eintippen wie beim Speichern der Datei.

Wenn Sie die Meldung ***Command not found*** (»Befehl nicht gefunden«) erhalten, überprüfen Sie, ob Sie Perl richtig installiert haben, und wenn Sie unter Unix arbeiten, ob der Pfad in Ihrer ***Shebang***-Zeile dem korrekten Pfad zu Ihrem Perl-Interpreter entspricht.

Wenn Sie unter Unix die Meldung ***Permission denied*** (»Zugriff verweigert«) erhalten, prüfen Sie, ob Sie daran gedacht haben, Ihr Skript ausführbar zu machen (mit dem `chmod +x` Befehl).

Wenn Sie immer noch Probleme haben sollten und Ihre Version von Perl von einem Systemadministrator installiert worden ist, bitten Sie ihn um Hilfe. Vielleicht wurde Perl anders als Sie erwarten installiert, oder Ihnen ist die Ausführung von Perl-Skripts nicht gestattet.

Wie funktioniert das Skript?

Jetzt haben Sie also ein zwei Zeilen-Perl-Skript, das den Satz »Hallo, Welt« auf den Bildschirm schreibt. Das scheint dann auch schon alles - sieht doch ganz einfach aus -, und doch stecken hier wichtige Perl-Grundlagen drin.

Zuallererst, das Prinzip bleibt auch bei längeren Perl-Skripts dasselbe: Ein Perl-Skript ist eine Folge von Anweisungen, die nacheinander, von oben nach unten, ausgeführt werden (manchmal gibt es Abzweigungen zu Subroutinen, mehrmals ausgeführten Code in Schleifen oder Code aus externen separaten Bibliotheken, aber das hier ist die Grundidee).

Die erste Zeile im »Hallo, Welt«-Skript ist ein Kommentar. Mit Kommentaren beschreiben Sie Teile des Perl-Codes, um zu erklären, was diese tun, oder zur Erinnerung, was Sie noch machen müssen - im allgemeinen um Ihr Skript aus welchem Grund auch immer zu kommentieren. Kommentare werden von Perl ignoriert; sie sind ausschließlich für Sie und jeden anderen, der Ihren Code liest, gedacht. Es gehört zum guten Programmierstil, ein Skript ausreichend zu kommentieren, auch wenn Sie weniger Kommentare als Code in Ihrem Skript stehen haben.

Die Shebang-Zeile, also die erste Zeile in Listing 1.1, ist eine spezielle Art von Kommentaren auf Unix-Systemen. Perl-Kommentare beginnen mit dem Rautenzeichen (#), und alles von diesem Zeichen bis zum Zeilenende wird ignoriert. Wenn Ihr Kommentar mehrere Zeilen lang sein soll, müssen Sie alle Zeilen mit # beginnen.

> *Perl kennt zwar auch Kommentare, die über mehrere Zeilen gehen, doch sind diese vornehmlich für die internen Perl-Dokumentationen (PODs) und nicht für einfache Kommentare gedacht. Bleiben Sie am besten bei den Rauten.*

Die zweite Zeile im Code (Zeile 3 im Listing 1.1) ist ein Beispiel einer grundlegenden Perl-Anweisung: Es ist ein Aufruf der eingebauten Funktion `print`, die einfach den Satz »Hallo, Welt« auf den Bildschirm ausgibt (nun, eigentlich auf das Standardausgabegerät, und das ist in diesem Fall der Bildschirm. Morgen erfahren Sie mehr über die Standardausgabe. Das `\n` innerhalb der Anführungszeichen löst einen Zeilenvorschub aus, genau wie in C; ohne dieses `\n` würde Ihr Skript am Ende des Satzes »Hallo, Welt« aufhören und nicht schön ordentlich am Anfang der nächsten Zeile.

Beachten Sie das Semikolon am Ende der `print`-Anweisung. Die meisten einfachen Perl-Anweisungen enden mit einem Semikolon. Das ist wichtig - vergessen Sie es nicht.

Mehr über all diese Konzepte - Anweisungen, Kommentare, Funktionen, Ausgabe und so weiter - später in diesem Kapitel und morgen, am Tag 2.

Eine Anmerkung zu den Warnungen

Auf den verschiedenen Plattformen, auf denen Sie Perl laufen lassen können, werden Perl-Warnungen auf unterschiedliche Weise aktiviert:

- Unter Unix verwenden Sie die Option `-w` in der Shebang-Zeile.

- Unter Windows verwenden Sie die Option `-w` direkt beim Programmaufruf von Perl selbst.
- Auf dem Macintosh wählen Sie »***Compiler Warnings***« aus dem Skript-Menü.

Es ist eine ausgesprochen gute Idee, die Warnungen einzuschalten, wenn Sie Perl noch lernen (und oft auch, wenn Sie schon Erfahrung haben). Perl ist beim Verzeihen von eigenartigem oder oft auch falschem Code sehr großherzig - für Sie kann das stundenlange Fehlersuche bedeuten. Das Einschalten der Warnungen hilft, häufige Fehler und aus Sicht des Interpreters seltsame Stellen in Ihrem Code zu entdecken. Gewöhnen Sie es sich an, die Warnungen zu beachten, und Ihnen bleibt langfristig eine Menge Fehlersuche erspart.

Ein weiteres Beispiel: Echo

Schauen wir uns ein weiteres Beispiel an. Das Skript aus Listing 1.2 bittet Sie um eine Eingabe und wiederholt diese Eingabe dann wie ein Echo auf dem Bildschirm, etwa so:

```
% echo.pl
Echo? Hi Laura
Hi Laura
%
```

Auf Windows NT führen Sie das Skript mit dem Perl-Interpreter aus, also:

`c:>perl -w echo.pl`

*Wenn Sie nur **echo.pl** eingeben, wird es nicht immer funktionieren (wenn die Ausgabe nur `pl` ist, kommt der `echo`-Befehl von NT in die Quere).*

Listing 1.2 zeigt den Code des Skripts ***echo.pl***:

Listing 1.2: Das Skript echo.pl.

```
1: #!/usr/bin/perl -w
2: # gibt die Eingabe wieder aus
3:
4: print 'Echo? ';
5: $input = <STDIN>;
6: print $input;
```

Sie brauchen im Augenblick noch nicht jede Zeile dieses Skripts zu verstehen. Morgen, am Tag 2, werde ich alles bis in kleinste Detail erklären.

Tippen Sie das Skript erst einmal ganz ein, und starten Sie es. Wichtig ist im Moment, dass Sie eine generelle Vorstellung haben, wie es funktioniert. Im folgenden ein kleiner Schnelldurchlauf durch den Code:

Zeile 1 und 2 sind beides Kommentare. In der ersten steht der Shebang und in der zweiten eine kurze Erklärung, was das Skript leistet.

Zeile 4 schreibt `Echo?` auf den Bildschirm. Beachten Sie, dass anders als bei »Hallo Welt!\n« in dieser Zeichenfolge kein `\n` steht. Das deshalb, weil hier ja um eine Eingabe gebeten wird, der Cursor also, wie bei einem Prompt so üblich, am Ende der Zeile stehen soll.

Zeile 5 liest eine an der Tastatur eingegebene Zeile und speichert sie in der Variablen namens `$input`. Sie brauchen sich weder um die einzelnen eingetippten Zeichen zu kümmern noch darum, wann das Zeilenende auftaucht. Perl liest alles ein, bis der Anwender die Eingabetaste drückt.

Zeile 6 schließlich gibt den Wert der Variablen `$input`, sprich den eingegebenen Text, auf dem Bildschirm aus.

Ein drittes Beispiel: Das Krümelmonster

Lassen Sie uns, nur aus Spaß, noch ein weiteres Beispielprogramm schreiben. Früher, in den Tagen der textorientierten Computer-Terminals, geisterte eine Zeitlang ein Scherzprogramm namens »***the cookie monster***« (»das Krümelmonster«) herum. Das Krümelmonster-Programm verriegelte Ihr Terminal und sagte ohne Ende: »Gib mir einen Keks« (oder: »Ich will Kekse« oder etwas in der Art), und ganz egal, was Sie eintippten, es bestand immer nur darauf, dass es Kekse wolle. Der einzige Ausweg aus dem Programm war, wirklich KEKSE einzutippen - darauf kam man aber oft erst nach stundenlangem Versuchen.

Listing 1.3 zeigt eine simple Perl-Version des Krümelmonster-Programms.

Listing 1.3: Das Skript kekse.pl

```
1: #!/usr/bin/perl -w
2: #
3: # Krümelmonster
4:
5: $kekse = "";
6:
7: while ( $kekse ne "KEKSE") {
8:   print 'ich will KEKSE: ';
9:   chomp($kekse = <STDIN>);
10: }
11:
12: print "Mmmm. KEKSE.\n";
```

Das ist jetzt schon ein bißchen komplizierter als »Hallo Welt« oder »Echo«. Und so könnte es aussehen, wenn Sie es ausführen:

```
% kekse.pl
ich will KEKSE: asdf
ich will KEKSE: exit
ich will KEKSE: quit
ich will KEKSE: stop
ich will KEKSE: es reicht
ich will KEKSE: @*&#@(*&@$
ich will KEKSE: kekse
ich will KEKSE: KEKSE
Mmmm. KEKSE.
%
```

*Die letzte Zeile ist eine kleine Variation des traditionellen **Cookie-Monster**-Programms. Beachten Sie auch, dass man aus dieser Variante ziemlich einfach herauskommt: Ein simples `[Strg]-[C]` wird es sofort abbrechen (oder ein Datei-Stop-Skript in MacPerl). Das Originalprogramm war allerdings nicht annähernd so nett. Aber hey, es ist erst Tag Eins, so heimtückisch können und brauchen wir noch nicht sein.*

OK, und jetzt erkläre ich Ihnen Zeile für Zeile, was das Kekse-Skript macht:

- Zeile 2 und 3 sind Kommentare (das wird Ihnen mittlerweile klar sein).
- Zeile 5 initialisiert die Variable `$kekse` als einen leeren String »«.
- Zeile 7 ist der Anfang einer `while`-Schleife. Solange die Bedingung innerhalb der Klammern erfüllt ist, wird der Code innerhalb der geschweiften Klammern ausgeführt. Hier ist die Bedingung für die Schleife, dass die `$kekse`-Variable nicht das Wort KEKSE enthält, und zwar genau in dieser Schreibweise. Wurde `Kekse` oder `kekse` eingegeben, ist die Bedingung erfüllt, und die Schleife verlangt noch einmal KEKSE von uns. Mit `while` und anderen Schleifen befassen wir uns ausführlich am Tag 4.
- Zeile 8 bittet Sie um Kekse. Beachten Sie, dass hier kein Zeilenvorschubzeichen am Ende steht.
- Zeile 9 sieht wirklich eigenartig aus. Die Funktion `chomp`, über die Sie morgen mehr lernen werden, schneidet einfach das Zeilenvorschubzeichen (Eingabetaste) vom Ende Ihrer Eingabe ab und speichert sie in der Variablen `$kekse`.

Noch einmal: Falls Sie nicht jede Zeile verstehen, verfallen Sie nicht in Panik. Morgen wird alles klarer.

Besonderheiten im Deutschen

Deutsche Sprache, schwääre Sprache! Zumindest in der Computerwelt hat das Deutsche seine Tücken: ä, ö, ü und ß.

Ändern wir zum Beispiel unser Krümelmonster-Skript dahin, dass es zwar nach Keksen schreit, man aber »süße Kekse« eingeben muss, damit das Programm wieder aufhört:

```
7: while ( $kekse ne "süße Kekse") {
8:   print 'ich will ganz bestimmte Kekse: ';
9:   chomp($kekse = <STDIN>);
10: }
```

Dann kann es Ihnen gut passieren, dass Sie zwar ganz richtig »süße Kekse« eintippen, das Programm aber immer weiter quakt. Was hier passiert, sehen Sie, wenn Sie explizit süße Kekse verlangen:

```
7: while ( $kekse ne "süße Kekse") {
8:   print 'ich will süße Kekse: ';
9:   chomp($kekse = <STDIN>);
10: }
```

Im DOS-Fenster zum Beispiel werden auf einmal s³¯eKekse verlangt. Deswegen bringt die Eingabe von »süße Kekse« natürlich nichts. (Auch auf einem Unix-System kann es kryptisch werden: Unter X-Window kann es wunderbar funktionieren, auf der Konsole aber nicht mehr). Woran liegt das? Im ursprünglichen ASCII-Zeichensatz sind von 256 möglichen Zeichen-Codes nur die von 0 bis 127 festgelegt. Die übrigen Werte haben die verschiedenen Hersteller ganz unterschiedlich definiert, IBM hat sich etwas anderes ausgedacht als Apple, Microsoft wieder etwas anderes für DOS und noch etwas anderes für Windows und so weiter. Systemübergreifend »sicher« sind nur die folgenden (sichtbaren) Zeichen:

```
!"#$%&'()*+,-./
0123456789
:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_'
abcdefghijklmnopqrstuvwxyz
{|}~
```

Bei allen anderen Zeichen (also leider auch ä, ö, ü, ß, Ä, Ö und Ü) kann es also Schwierigkeiten geben.

Und was macht man da? Am sichersten ist, in Perl-Skripts gar keine anderen als die oben genannten Zeichen zu verwenden, dann kann, in welchem Zeichensatz auch immer, gar nichts schiefgehen. Also »suesse Kekse«.

Wenn Sie diese Schreibweise partout nicht ertragen, müssen Sie Ihre Skripts in dem Zeichensatz speichern, in dem Ihr Perl-Interpreter sie liest. Windows-Anwender können zum Beispiel mit einem Editor arbeiten, der Dateien auch in DOS-Code umwandeln kann. Oder sie schreiben ihre Skripts gleich ganz unter DOS (viel Spaß mit edit).

*Vergessen Sie aber nicht, dass Umlaute und ß dann zwar auf **Ihrem** System funktionieren, aber nach wie vor in Variablennamen nicht erlaubt und auf einem anderen System wieder eine mögliche Fehlerquelle sind. Deswegen verwende ich in diesem Buch die unschönere, aber sichere Schreibweise.*

Vertiefung

Vertiefung? Das war nicht gerade flach bis jetzt! Sie werden diesen Abschnitt am Ende jeder Lektion in diesem Buch finden. Hinter »Vertiefung« steht die Idee, dass es in Perl Dinge gibt, die Ihnen beizubringen ich keine Zeit

habe, oder noch andere Methoden, die zum selben Ziel führen (Sie erinnern sich: »***There's more than one way to do it***«). Der heutige Vertiefungsabschnitt gibt Ihnen Hinweise, wo Sie mehr lernen können - nämlich in der Online-Dokumentation, die bei Ihrem Interpreter dabei ist, oder im Internet unter `http://www.perl.com`.

Ein Großteil der Online-Dokumentation liegt in Form von *Manpages* vor (***man*** ist die Unix-Abkürzung für ***manual***, ***man pages*** wortwörtlich also »Handbuchseiten«). Ich werde das ganze Buch hindurch auf diese Manpages verweisen, zum Beispiel die ***perlfunc-*** oder ***perlop-***Manpages. Wenn Sie auf einem Unix-System arbeiten, können Sie diese Seiten normalerweise mit dem ***man***-Befehl aufrufen, zum Beispiel:

```
% man perlfunc
```

Die Inhalte aller *Manpages* sind auch im perl-eigenen ***POD***-Format erhältlich, eine spezielle Form von Perl-Dokumentation, die auf jeder Plattform gelesen oder mit in Perl eingebauten Konvertierungsprogrammen in Volltext oder HTML konvertiert werden kann (***POD*** steht für ***plain old documentation***, was man vielleicht mit »klare alte Dokumentation« übersetzen könnte, aber doch lieber einfach POD sagt ... Das POD-Format dient insbesondere dazu, beim Schreiben von Modulen die dazugehörige Dokumentation direkt in den Code des Moduls zu integrieren). Die POD-Seiten selbst sind im pod-Verzeichnis Ihrer Perl-Distribution gespeichert, und Sie können sie unter Unix oder Windows mit dem `perldoc`-Befehl und dem Namen einer Perl-*Manpage* lesen, etwa so:

```
% perldoc perlfunc
```

Wenn Sie etwas über eine bestimmte Perl-Funktion wie `print` oder `chomp` wissen möchten, verwenden Sie `perldoc` mit der Option `-f` folgendermaßen:

```
% perldoc -f print
```

In Windows-Distributionen finden Sie die *Manpages* immer häufiger nur als HTML- Dateien im Verzeichnis ***docs\ Perl***. Die Dateien haben die gleichen Namen wie die *Manpages* (zum Beispiel ***perlfunc.html***), Sie können sie also direkt mit Ihrem Webbrowser öffnen. Oder Sie öffnen zuerst die Datei ***perl.html*** (und setzen einen Bookmark), dort stehen gleich am Anfang die Links zu allen anderen *Manpages*, die Sie dann nur noch anzuklicken brauchen.

Auf dem Mac finden Sie alle Dokumentation zu Perl im Hilfemenü von MacPerl. Allerdings müssen Sie die Dateinamen erraten (die ***perlfunc-****Manpage* ist zum Beispiel unter »***Builtin functions***«, »Eingebaute Funktionen«, aufgelistet). ***POD***-Dateien liest MacPerl mit der Anwendung ***Shuck***.

Letzlich sind alle Perl-*Manpages* auch im Web unter `http://www.perl.com/CPAN-local/doc/manual/html/pod/` erhältlich. Ich finde es oft einfacher, die Perl-*Manpages* über das Web zu suchen und zu lesen als mit den ***perldoc*** oder ***man***-Befehlen (vielen Windows-Benutzern geht es wohl genauso).

Wenn Sie lieber richtiges Papier in den Händen haben möchten oder Ihre Englischkenntnisse für die *Manpages* nicht ausreichen, empfiehlt sich im allgemeinen das Buch ***Programmieren mit Perl*** (Wall, Christiansen and Schwartz, deutsche Übersetzung Klicman, O'Reilly, 1997), auch bekannt als das Kamelbuch (wegen dem Kamel auf seinem Cover). Das Kamelbuch ist die definitive Referenzbibel für Perl und beschreibt es in fast beängstigenden Details - es ist ein ganz schöner Wälzer. Ziel ist es, Ihnen alle Grundlagen und die allgemeine Praxis beizubringen. Wollen Sie aber einige der mehr esoterischen Features der Sprache erforschen, ist ***Programmieren mit Perl*** wahrscheinlich ein wichtiges Buch für Sie (meines ist zerfleddert und vollgekritzelt).

Zusammenfassung

Heute war eher ein »Hallo, wie geht es Ihnen«-Tag als ein harter Arbeitstag. In dieser Lektion haben Sie einiges über den Hintergrund und die Geschichte von Perl gelernt, warum es Spaß macht und warum es für Sie nützlich ist, es zu lernen. Nach all den Hintergrundinformationen in der ersten Hälfte der Lektion haben Sie einen ersten Blick darauf werfen können, wie ein Perl-Skript überhaupt aussieht und wie Sie es auf Ihrem System zum Laufen bekommen. Außerdem haben Sie ein paar Grundlagen über Kommentare, Perl-Anweisungen und den Ablauf von Perl-Programmen gelernt. Spätestens jetzt sollten Sie Perl auf Ihrem System installiert und startklar haben - von nun an ist alles Code.

Fragen und Antworten

Frage:
Wenn Perl so einfach zu lernen ist, warum brauche ich dann 21 Tage dafür?

Antwort:
Brauchen Sie vielleicht gar nicht. Wenn Sie 21 Tage Zeit und nichts anderes zu tun haben, können Sie eine ganze Menge mehr Perl lernen - mehr, als so manche Leute wissen, die sich selbst Perl-Programmierer nennen. Aber es ist auch gut möglich, dass Sie schon nach der ersten oder zweiten Woche genug Perl aufgeschnappt haben, um erst einmal klarzukommen, und Sie den schwierigeren Stoff ignorieren - bis Ihnen irgendwann abenteuerlicher zumute ist oder Sie mehr mit der Sprache machen müssen. Wenn Sie schon viel Programmiererfahrung mit einer anderen Sprache haben, kommen Sie durch die ersten Kapitel wahrscheinlich eine ganze Ecke schneller als nur ein Kapitel pro Tag. Eine Programmiererweisheit: Den Job so schnell wie möglich mit dem geringsten Arbeitsaufwand erledigen. Wenn Ihnen das schneller gelingt - wunderbar, das ist ganz im Sinne von Perl.

Frage:
Ich habe keine Programmiererfahrung, obwohl ich viel mit HTML gearbeitet habe und mich etwas mit JavaScript auskenne. Kann ich Perl lernen?

Antwort:
Ich wüßte nicht, was dagegen spräche. Obwohl ich dieses Buch für Leute geschrieben habe, die schon ein paar Programmierkenntnisse haben - wenn Sie sich in Ruhe durch das Buch, die Beispiele und Übungen arbeiten und selbst noch etwas experimentieren, sollte es Ihnen keine großen Schwierigkeiten bereiten. Perls Flexibilität macht es auch zu einer großartigen Sprache für Anfänger. Und wenn Sie sich mit anderen Webtechnologien schon auskennen, paßt Perl wunderbar dazu.

Frage:
Heißt der Quellcode, den ich schreibe, Perl-Programm oder Perl-Skript?

Antwort:
Das kommt darauf an, wie pedantisch Sie mit der Semantik sind. Man kann argumentieren, dass Programme kompiliert und Skripts interpretiert werden. Man habe ja schließlich C- und Java-Programme (und C- und Java-Compiler), aber JavaScript- oder AppleScript-Skripts. Und da Perl im wesentlichen eine interpretierte Sprache ist, ist der Code, den Sie schreiben, ein Perl-Skript und kein Perl-Programm. Man kann aber dagegenhalten, dass das, was Sie tun, Programmieren ist, und dieser Vorgang die Entwicklung eines Programms beinhaltet. Eine dritte Position wäre die Ansicht, dass es eigentlich überhaupt keine Rolle spielt. Diese Auffassung ist mir äußerst sympathisch, doch mein Verleger verlangt Konsistenz von mir; also bleibe ich beim ersten Argument und nenne sie Skripts.

Frage:
Ich habe den *Hallo-Welt*-Einzeiler eingetippt, aber er läuft einfach nicht!

Antwort:
Sind Sie sicher, dass Sie Perl auf Ihrem System installiert haben? Perl zu installieren und zum Laufen zu bringen, ist die Hauptaufgabe von heute. Wenn Sie an das Ende vom Buch blättern, finden Sie in den Anhängen Installationsanweisungen für Ihr System (vorausgesetzt, Sie arbeiten mit Unix, Windows oder MacOS). Außerdem können die Dokumente Ihrer Perl- Distribution eine große Hilfe sein, alles an den Start zu bringen.

Frage:
Wenn ich in Windows auf meine Datei *hallo.pl* doppelklicke, öffnet sich ganz kurz ein Fenster und verschwindet dann wieder. Wie kann ich die Ausgabe meines Programms sehen?

Antwort:
Führen Sie die Skripts aus einer Kommandozeile oder einem DOS-Prompt aus, nicht aus dem Explorer-Fenster. Starten Sie zuerst die MS-DOS- Eingabeaufforderung, wechseln Sie in das entsprechende Verzeichnis, und tippen Sie dann `perl -w` *und den Namen Ihres Skripts ein. Hier ein einfaches Beispiel:*

```
   C:\> cd ..\scripte
C:\scripte> perl -w hallo.pl
```

> Sollten Sie noch Schwierigkeiten haben, schauen Sie in die Windows Perl FAQs unter http://www.activestate.com/support/faqs/Win32.

Frage:
Ich arbeite unter Windows. Sie erwähnen, dass die #!- (*Shebang*-)Zeile am Anfang des Skripts eine reine Unix-Angelegenheit ist. Warum soll ich sie mit hineinnehmen, wenn ich nicht auf Unix arbeite und es auch nicht vorhabe?

Antwort:
Brauchen Sie nicht, wenn nicht zu erwarten ist, dass Ihre Perl-Skripts jemals auf Unix laufen (aber bedenken Sie, auch manche Webserver unter Windows verlangen danach). Weil die Shebang-Zeile mit einer Raute anfängt, ist sie eigentlich ein Kommentar, deswegen wird sie unter Windows ignoriert, und Sie können sie auch weglassen. Aber vielleicht sollen Ihre Skripts ja irgendwann doch einmal unter Unix eingesetzt werden - auf jeden Fall ist die Shebang-Zeile eine gute Angewohnheit.

Frage:
Das Skript *hallo.pl*- läuft wunderbar, aber *kekse.pl* gar nicht. Perl beschwert sich über die »*undefined subroutine chomp*«.

Antwort:
Klingt, als hätten Sie noch Perl 4. Dort gibt es die Funktion `chomp` *noch nicht, deswegen wird sie für eine »undefinierte Subroutine« gehalten. Geben Sie* `perl` *mit der Option* `-v` *ein, dann sagt Perl Ihnen »this is Perl« und die genaue Version. Es sollte Perl 5 sein (dieses Buch behandelt Perl 5, und Sie werden Schwierigkeiten bekommen, wenn Sie versuchen, Perl-5-Features mit Perl 4 zu nutzen). Überprüfen Sie noch einmal Ihre Installation, und stellen Sie sicher, dass Sie für alle Ihre Skripts Perl 5 verwenden.*

Frage:
Ich arbeite mit dem Editor emacs unter Unix. Gibt es einen speziellen Modus für Perl?

Antwort:
Selbstverständlich! Perl ist eine Unix-Sprache und eine populäre dazu. Sowohl in der Standard-GNU-emacs-Distribution als auch im Perl-Interpreter- Paket ist ein perl-mode dabei. Je nachdem, welche emacs-Version Sie benutzen und wie diese eingerichtet ist, kann eine .pl-Erweiterung oder eine Shebang-Zeile automatisch den perl-mode für Sie starten. Ansonsten schalten Sie ihn mit der Eingabe von `M-X perl-mode` *ein.*

Frage:
In Ihren Beispielen stehen in manchen print-Anweisungen doppelte Anführungszeichen und in anderen nur einfache. Warum?

Antwort:
Gut aufgepaßt! Dafür gibt es einen besonderen Grund, der damit zu tun hat, ob zwischen den Anführungszeichen Anweisungen wie `\n` *oder Variablennamen stehen. Sie werden den Unterschied morgen lernen.*

Workshop

Der Workshop-Abschnitt, ein Bestandteil jedes Kapitels, ist in zwei Teile gegliedert:

- Ein Quiz - hier können Sie überprüfen, ob Sie den Stoff des Kapitels verstanden haben.
- Übungen - hier können Sie Ihr neues Wissen wirklich anwenden. Nur das bringt Erfahrung in der Arbeit mit Perl.

Die Lösungen zu Quiz und Übungsaufgaben stehen weiter unten.

Quiz

1. Wofür steht Perl? Was bedeutet es?
2. Wer hat Perl ursprünglich geschrieben? Wer kümmert sich jetzt darum?
3. Welches ist die aktuelle Perl-Version?
4. Was sind die grundlegenden Unterschiede zwischen kompilierten und interpretierten Sprachen? Was davon ist Perl? Warum ist das nützlich?
5. Welche Aussage beschreibt Perl am besten?

- Perl ist eine kleine, mächtige, streng definierte Sprache mit einem Minimum an Konstrukten.
- Perl ist eine umfangreiche, mächtige, flexible Sprache mit vielen verschiedenen Möglichkeiten, etliche Dinge zu tun.

6. Was findet man auf der Webseite `http://www.perl.com`?
7. Was macht die Shebang-Zeile in einem Perl-Skript?
8. Wie erstellt man Kommentare in Perl?
9. Was sind Perl-Warnungen? Wie schalten Sie sie auf Ihrer Plattform ein? Warum sind sie nützlich?

Übungen

1. Versuchen Sie, die Programme ***hallo.pl***, ***echo.pl*** und ***kekse.pl*** auf die eine oder andere Art zu verändern. Übertreiben Sie es nicht: Entwickeln Sie nicht gleich ein Betriebssystem oder ähnliches, probieren Sie einfach ein paar Kleinigkeiten aus. Bauen Sie verschiedene Fehler in das Skript ein, und sehen Sie sich an, welche Fehlermeldungen Sie dann erhalten (geben Sie zum Beispiel Kommentare ohne das Rautenzeichen am Anfang ein, entfernen Sie die Anführungszeichen am Ende oder ein Semikolon). Machen Sie sich mit den verschiedenen Fehlermeldungen vertraut, die Perl ausgibt, wenn Sie Teile einer Anweisung vergessen.
2. Ändern Sie das ***Hallo Welt***-Skript so, dass es die Begrüßung zweimal ausgibt.
3. Ändern Sie das ***Hallo Welt***-Skript so, dass es die Begrüßung zweimal in derselben Zeile ausgibt.
4. FEHLERSUCHE: Was ist falsch an diesem Skript?

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

5. FEHLERSUCHE: Was ist falsch an diesem hier? (Tipp: Es sind gleich zwei Fehler)

```
    #!/usr/bin/perl -w
print 'Geben Sie Ihren Namen ein: '
# die Input-Daten speichern $input = <STDIN>;
print $input;
```

6. (Bonusfrage) Kombinieren Sie die ***Hallo-Welt-*** und ***Krümelmonster***-Beispiele so, dass das Skript Sie um die Eingabe Ihres Namens bittet und Ihnen dann immer wieder hallo sagt, bis Sie »bye bye« eintippen. Hier ein Beispiel, was das Skript ausgeben könnte:

```
    Geben Sie Ihren Namen ein: Laura
Hallo, Laura!
Geben Sie Ihren Namen ein: Anastasia
Hallo, Anastasia!
Geben Sie Ihren Namen ein: Turok der Allmächtige
Hallo, Turok der Allmächtige!
Geben Sie Ihren Namen ein: bye bye
hallo, bye bye!
%
```

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Perl steht für Practical Extraction and Report Language. Das heißt, mit dieser Sprache extrahiert man Teile aus Dateien und erstellt darüber Berichte (***reports***). Das Wort ***practical*** (praktisch) bedeutet, dass es für diese Art von Aufgaben eine sehr nützliche Sprache ist.
2. Larry Wall ist der Originalautor von Perl, und er ist weiterhin eng mit dessen Entwicklung befaßt. Perl wird in erster Linie von einer Gruppe freiwilliger Entwickler gepflegt und unterstützt.
3. Die exakte Antwort auf diese Frage fällt abhängig von der Version, die Sie installiert haben, unterschiedlich aus. Die aktuelle Hauptversion ist allerdings Perl 5, auf meiner Unix-Maschine ist die Version 5.004_04 installiert.
4. Kompilierte Sprachen verwenden ein Compiler-Programm, um den Programmquelltext in Maschinencode oder Bytecode zu konvertieren. Man läßt dann diesen Code laufen, um das Programm auszuführen. Bei

einem Interpreter ist aber der Quellcode gleichzeitig der finale Code. Der Interpreter liest die Quelldatei und führt sie aus, ohne ein Kompilat in einer Datei abzulegen.

1. Perl ist eine Kombination aus einer kompilierten und einer interpretierten Sprache. Es benimmt sich wie eine interpretierte Sprache, das heißt, es ist schnell zu schreiben, schnell zu ändern und auf verschiedene Plattformen portierbar; aber es kompiliert ebenfalls den Quellcode, bevor es ihn ausführt und hat deswegen die Fehlerkorrektur-Features und auch nahezu die Geschwindigkeit einer kompilierten Sprache.
5. Die zweite Aussage beschreibt Perl am besten:

- Perl ist eine umfangreiche, mächtige, flexible Sprache mit vielen verschiedenen Möglichkeiten, etliche Dinge zu tun.

6. Die Webseite `http://www.perl.com` ist die zentrale Sammelstelle für alles, was mit Perl zu tun hat: Dort schaut man nach der aktuellen Perl-Version, dem Comprehensive Perl Archive Network (Tools, Module und Utilities rund um Perl), Dokumentationen, FAQs und weiteren Informationen - so ziemlich allem, was Sie sich im Zusammenhang mit Perl nur wünschen können.
7. Die Shebang-Zeile wird unter Unix verwendet, um Unix zu sagen, mit welchem Programm es ein Skript ausführen soll. Sie enthält den Pfad zum Perl-Interpreter auf Ihrem System. Für andere Betriebssysteme sieht die Shebang-Zeile nur wie ein einfacher Kommentar aus und wird normalerweise ignoriert.
8. Kommentare in Perl beginnen mit einem #. Alles vom # bis zum Ende der Zeile wird ignoriert.
9. Perl-Warnungen sind spezielle Diagnosemeldungen, die helfen können, gebräuchliche Fehler zu beheben und Stellen zu finden, wo Sie vielleicht etwas anstellen, das zu unerwartetem Verhalten führt. Perl-Anfänger sind gut beraten, die Warnungen einzuschalten.
1. Um Warnungen unter Unix einzuschalten, verwenden Sie die Option ***-w*** mit dem Perl-Interpreter in der Shebang-Zeile.
1. Um Warnungen unter Windows einzuschalten, verwenden Sie die Option ***-w*** in der Perl-Kommandozeile.
1. In MacPerl schalten Sie Warnungen ein, indem Sie ***Compiler Warnings*** aus dem Skriptmenü wählen.

Lösungen zu den Übungen

1. Wenn Sie diese Übung tatsächlich durchgeführt haben, dann kennen Sie jetzt einige Fehlermeldungen.
2. Eine Beispiellösung:

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

3. Ihre Lösung könnte beispielsweise so aussehen:

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

4. Hier fehlt ein Rautenzeichen in der ersten Zeile des Skripts. Diese Zeile wird einen Fehler auslösen (unter Unix wird das Skript wahrscheinlich überhaupt nicht laufen).
5. Folgende zwei Fehler sollten Sie gefunden haben:

- Nach der ersten `print`-Anweisung fehlt das Semikolon am Zeilenende.
- Die nächste Zeile beginnt mit einem Kommentar - der ganze Text nach dem Rautezeichen wird für einen Kommentar gehalten und ignoriert.

6. Mit dem, was Sie heute gelernt haben, könnte Ihr Skript so aussehen:

```
    #!/usr/bin/perl -w
#
$name = "";
while ($name ne 'bye bye') {
  print 'Geben Sie Ihren Namen ein: ';
  chomp($name = <STDIN>);
  print "Hallo, ";
  print $name;
  print "!\n";
}
```

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Mit Strings und Zahlen arbeiten

Skalar ist ein besonderer Perl-Begriff für einen ***einzelnen*** Wert, ganz egal ob es sich dabei um eine Ziffer oder einen String (Zeichenfolge) handelt - solange es nur ***ein*** Wert ist, ist es ein Skalar. In diesem Kapitel lernen Sie, was es mit skalaren Daten und Variablen auf sich hat. Und ich werde Ihnen verschiedene Operatoren und Funktionen zur Arbeit mit Skalaren vorstellen. All dies gehört zu den wesentlichen Grundbausteinen eines Perl-Skripts.

Die Themen von heute sind:

- Zahlen und Strings
- Skalarvariablen: Definition, Verwendung und Wertzuweisung
- Einfache Berechnungen
- Vergleiche und Überprüfungen

Skalare Daten und Variablen

Perls Datentypenkonzept ist ziemlich flexibel: Anders als Sprachen wie C oder Java, mit ihren getrennten Typen für Integer (ganze Zahlen), Zeichen, Fließkommazahlen und so weiter (und strengen Regeln für Verwendung und Konvertierung dieser Datentypen) unterscheidet Perl lediglich zwischen zwei generellen Sorten von Daten - dafür aber nach einem ganz anderen Kriterium: Einzahl oder Mehrzahl, Singular oder Plural. ***Einzelne*** Werte wie ***eine*** Zahl oder ***eine*** Zeichenfolge sind ***skalare*** Daten. ***Mehrere*** Werte zusammen, ***Sammlungen*** von Werten wie zum Beispiel Arrays, heißen ***Listen***. Ob diese Listen wiederum nur aus Zahlen, nur aus Strings oder einem Mischmasch aus beidem bestehen, ist nicht so wichtig. Die Unterscheidung zwischen Skalaren und Listen hingegen spielt eine große Rolle. Einige Konstrukte in Perl verhalten sich ganz unterschiedlich, je nachdem ob Sie mit Skalaren oder Listen arbeiten. Heute allerdings bleiben wir erst einmal bei einfachen Zahlen und Strings und den Variablen dafür. Die Listendaten heben wir uns für später auf; der Unterschied ist dann auch einfacher zu verstehen.

Während ich heute ausschließlich von Zahlen und Strings als skalaren Daten rede, gibt es noch eine dritte Form: Referenzen sind auch eine Form von skalaren Daten. Aber Sie brauchen so früh in Ihrem Perl-Studium noch nichts über Referenzen zu wissen; deswegen werde ich sie jetzt erst einmal ignorieren. Mit Referenzen befassen wir uns viel später, am Tag 19.

Zahlen

Zahlen können Sie in Ihren Perl-Skripts eigentlich eintippen, wie Sie möchten. Beachten Sie nur, dass in Perl (wie im Englischen überhaupt) statt des Kommas ein Punkt geschrieben wird - 0,5 heißt auf Englisch 0.5. Dies alles sind gültige Zahlen in Perl:

```
4
3.2
0.2
.23434234
5.
7.0
1_123_456
10E2
45e-4
0xbeef
012
```

Integer- und Fließkommazahlen werden beide so dargestellt, wie man es gewohnt ist: Integerzahlen mit ganzen Zahlen, Fließkommazahlen mit einem Integer- und einem Dezimalteil, nur wie gesagt durch einen Punkt getrennt. Kommas sind nicht erlaubt. Bei Fließkommazahlen können Nullen vor oder nach dem Punkt weggelassen werden (statt `0.23434234` kann man also auch `.23434234` schreiben bzw. `5.` statt `5.0`). Die Unterstriche in `1_123_456` sind eine optionale Schreibweise für längere Zahlen (so wie wir lange Zahlen mit Punkten in Dreiergruppen aufteilen) und dienen allein der Lesbarkeit im Skript. (Die Unterstriche werden aus der Zahl entfernt, sobald sie ausgewertet wird - aber nur, wenn die Zahl direkt im Skript steht. Wenn sie irgendwie anders ins Skript eingelesen wird, funktioniert diese Umwandlung nicht. Als Benutzer kann man also nicht `1_123_456` an der Tastatur eingeben.) Exponenten werden durch ein großes oder kleines e eingeleitet, auf das der positive oder negative Exponent folgt. Hexadezimale Zahlen beginnen mit `0x` und oktale mit einer 0 (Null).

Perl unterscheidet nicht zwischen Integer- und Fließkommazahlen, ***signed*** und ***unsigned*** oder kurzen oder langen Zahlen. Eine Zahl ist ganz einfach immer skalar, und Perl konvertiert zwischen den Zahlentypen in Ihren Skripts je nach Bedarf.

Strings

Es scheint vielleicht etwas eigenartig, dass Strings skalare (einzelne) Daten sein sollen, wo ein String doch eine Folge von Buchstaben ist und in anderen Sprachen eigentlich in Arrays gespeichert wird. Aber Perl ist anders. In Perl sind Strings einfach eine Form von skalaren Daten.

Strings können auf zwei Arten dargestellt werden: als kein oder mehr Zeichen zwischen einfachen Anführungszeichen (`' '`) oder doppelten Anführungszeichen (`" "`), zum Beispiel:

```
'das hier ist ein String'
"das hier ist auch ein String"
""
"$fahr Grad Fahrenheit sind $cel Grad Celsius"
"Hallo, Welt!\n"
```

Strings können jede Art von ASCII-Daten enthalten, auch binäre Daten (hohe und niedrige ASCII-Zeichen). Allerdings enthalten Text-Strings häufig nur niedrige ASCII- Zeichen (normale Buchstaben, keine Akzente oder Sonderzeichen). Strings können beliebig lang sein. Die Länge Ihres Strings ist nur durch die Größe des Ihnen zur Verfügung stehenden Speichers begrenzt. Obwohl es eine Weile dauern und ziemlich viel Speicher fressen würde, das Gesamtwerk von Shakespeare oder einen 10- Megabyte-Unix-Kernel in einen einzigen Perl-String zu lesen, könnten Sie das sicherlich machen.

Es gibt zwei Unterschiede zwischen Strings in einfachen Anführungszeichen (auch ***single-quoted*** Strings genannt, ***quote*** ist englisch für Anführungszeichen, und was ein ***Single*** ist, brauche ich wohl nicht zu erklären) und denen in doppelten (***double- quoted*** Strings). Zum einen führt Perl auf einem String in doppelten Anführungszeichen ***Variableninterpolation*** aus: Das heißt, jeder Variablenname innerhalb der Anführungszeichen (wie `$fahr` und `$cel` in dem obigen Beispiel) wird durch die aktuellen Werte dieser Variablen ersetzt. Ein String in einfachen Anführungszeichen wie `'$fahr Grad Fahrenheit sind $cel Grad Celsius'` wird genau so ausgegeben, wie Sie es hier sehen, mit den Dollarzeichen und Variablennamen an Ort und Stelle. Wenn Sie wirklich ein Dollarzeichen in einem ***double-quoted*** String haben möchten, müssen Sie einen Backslash (umgekehrten Schrägstrich) davorsetzen: `"ein Doppelzimmer kostet pro Nacht \$14 Dollar"`.

Variableninterpolation funktioniert auch mit Strings, die Listenvariablen enthalten. Sie lernen mehr darüber an Tag 4, »Mit Listen und Arrays arbeiten«.

Der zweite Unterschied zwischen doppelten und einfachen Anführungszeichen ist, dass ***double-quoted*** Strings auch die in Tabelle 2.1 gezeigten Escape-Sequenzen interpolieren. Diese werden Ihnen vertraut vorkommen, wenn Sie C kennen; allerdings gibt es auch einige andere Escape-Sequenzen speziell für Perl. In ***single- quoted*** Strings werden Escape-Sequenzen größtenteils ausgegeben, wie sie eingetippt wurden: So wird aus `'Hallo Welt!\n'` dann `Hallo Welt!\n`, ohne Zeilenvorschub. Diese Regel hat zwei Ausnahmen: Mit `\'` und `\\` können Sie Anführungszeichen bzw. Backslashes in ***single-quoted*** Strings schreiben:

```
'Dies ist Laura\'s Skript'              # Ausgabe: Dies ist Laura's Skript
```

```
'Die Eingaben stehen in C:\\files\\input'
# Ausgabe: Die Eingaben stehen in C:\files\input'
```

Zeichen	Bedeutung
\n	Zeilenvorschub (***newline***)
\r	Wagenrücklauf (***carriage return***)
\t	Tabulator
\f	Seitenvorschub (***formfeed***)
\b	Rückschritt (***backspace***)
\a	Alarm (***bell***)
\e	Escape-Zeichen (***ESC***)
\0nn	Oktal (nn sind Ziffern), (\033 ist zum Beispiel ESC in Oktal)
\xnn	Hexadezimal (nn sind 0-9, a-f oder und A-F) (\x7f ist zum Beispiel DEL in Hexadezimal)
\cX	Steuerungszeichen (***control characters***), wobei X ein beliebiger Buchstabe ist (zum Beispiel ist \cC äquivalent mit [Ctrl]-[C] bzw. [Strg]-[C] auf deutschen Tastaturen)
\u	Großbuchstabe (***uppercase***), macht aus dem nächsten Buchstaben einen Großbuchstaben
\l	Kleinbuchstabe (***lowercase***), macht aus dem nächsten Buchstaben einen Kleinbuchstaben
\U	macht aus allen folgenden Buchstaben Großbuchstaben
\L	macht aus allen folgenden Buchstaben Kleinbuchstaben
\Q	Vor alle folgenden nichtalphanumerischen Zeichen wird ein Backslash gestellt (damit sie nicht mehr als Metazeichen betrachtet werden)
\E	Beendet \U, \L und \Q

Tabelle 2.1: ***Escape-Sequenzen für Strings***

Wir werden uns am Ende dieses Kapitels im Vertiefungsabschnitt mit \u, \l, \U, \L *und* \E *befassen.*

Leere Strings, das sind Strings ohne irgendein Zeichen, werden einfach mit zwei direkt aufeinanderfolgenden Anführungszeichen dargestellt. Beachten Sie, dass der Leerstring "" (ohne Inhalt = leer) nicht dasselbe ist wie ein String aus einem Leerzeichen (" ").

Was sollten Sie nun in einem Perl-Skript verwenden, double- oder single-quoted Strings? Das hängt ganz davon ab, was Sie mit dem String machen wollen. Wenn Sie Escape-Sequenzen und Variableninterpolation brauchen, nehmen Sie einen double- quoted String. Wenn Sie einen String mit einer Menge Dollarzeichen haben (zum Beispiel aktuelle Dollarsummen), nehmen Sie vielleicht lieber einen single-quoted String.

Konvertieren zwischen Zahlen und Strings

Weil sowohl Zahlen als auch Strings skalar sind, müssen Sie nicht explizit zwischen beiden hin- und herkonvertieren, sie sind austauschbar. Je nach Kontext wandelt Perl einen String automatisch in eine Zahl um und umgekehrt. So könnte zum Beispiel der String "14" zur Zahl 5 addiert werden, und das Ergebnis wäre die Zahl 19.

Das klingt anfangs vielleicht etwas komisch, aber es macht Dinge wie Ein- und Ausgabe sehr leicht. Sie brauchen eine über Tastatur eingegebene Zahl? Lesen Sie einfach ein, was getippt wurde, und verwenden Sie es dann als String. Sie brauchen kein ***scanf*** oder eine andere Funktion um alles vor- und zurückzukonvertieren. Perl macht das für Sie.

*Es gibt eine Ausnahme von der automatischen Konvertierung in Perl: Strings, die Oktal- oder Hexadezimalzahlen zu enthalten scheinen. Die Formate `0123` und `0xabc`, die Sie im Abschnitt über Zahlen kennengelernt haben, gelten nur für Zahlen, die Sie wirklich in Ihren Code tippen (in der Computerfachsprache **Literale** genannt). Von der Tastatur oder anderen Strings übergebene Strings in oktaler oder hexadezimaler Notation werden nicht automatisch in Zahlen umgewandelt. Das müssen Sie mit der Funktion `oct` dann selber machen:*

```
$zahl = '0x432';
print $zahl;      # Ausgabe: 0x432
$hexnum = oct $zahl;
print $hexnum;   # Ausgabe: 1074 (dezimales Äquivalent)
$zahl = '0123';
print (oct $zahl);  # Ausgabe: 83
```

Die Funktion `oct` kann aus dem Kontext schließen, ob der String eine Oktal- oder Hex-Zahl ist. Sie können auch die Funktion `hex` verwenden, allerdings nur für Hex-Zahlen.

Perl interpretiert auch scheinbar unsinnige Anweisungen wie `"foo" + 5` oder `"23skidoo" + 5`. In der ersten Anweisung wird der String zu 0 umgewandelt, und in der zweiten Anweisung wird 23 aus dem String verwendet und zu 5 addiert. Wenn Sie Perl-Warnungen (mit der Option ***-w*** oder ***Skripts--> Compiler Warnings*** in MacPerl) aktiviert haben, wird Perl sich allerdings über derartige Vorgänge beschweren, und das soll es ja auch.

Skalarvariablen

Skalare Daten werden in Skalarvariablen gespeichert. Skalarvariablen haben in Perl ein Dollarzeichen ($) am Anfang, gefolgt von einem oder mehreren alphanumerischen Zeichen oder Unterstrichen, wie zum Beispiel:

```
$i
$laenge
$Zinsen_im_Jahr
$max
$a56434
```

Die Regeln für gültige Variablennamen (für jede Perl-Variable, nicht nur Skalarvariablen) sind folgende:

- Das erste Zeichen nach dem `Präfix $` sollte ein Buchstabe des Alphabets oder ein Unterstrich sein. Umlaute (`ä`, `ö`, `ü`) und `ß` sind Sonderzeichen und nicht erlaubt. Andere Zeichen wie Ziffern oder Sonderzeichen wie `%`, `*` und so weiter sind in der Regel für spezielle Perl-Variablen reserviert.
- Nach diesem Zeichen (also ab dem dritten Zeichen) können Variablennamen alle anderen Buchstaben des Alphabets, Zahlen und Unterstriche enthalten (auch hier dürfen keine Umlaute und kein ß stehen).
- Variablennamen unterscheiden Groß- und Kleinschreibung - das heißt `$var` ist eine andere Variable als `$VAR` oder `$Var`.
- Variablennamen dürfen insgesamt höchstens 256 Zeichen lang sein. Nach dem Anfangs-`$` können Sie also noch 255 Zeichen verwenden, wenn Sie lustig sind (ich persönlich bekomme Kopfschmerzen, wenn ich mir so lange Variablennamen vorstelle, aber hey, wenn Ihnen das Spaß macht, nur zu).

Sie müssen Variablen in Perl nicht deklarieren oder initialisieren. Sie können sie verwenden, wie Sie sie brauchen. Skalarvariablen ohne Anfangswert haben den ***undefinierten Wert***, und der braucht Ihnen keine Sorgen zu machen - er wird immer entweder ein Leerstring oder ein Nullwert sein, je nachdem, wo Sie ihn benutzen (das ist wieder dieses Austauschbarkeits-Feature). Dennoch sollten Sie Ihre Variablen explizit initialisieren. Das gehört zum guten Umgangston in einer Programmiersprache. So wird Perl, wenn Sie die Warnungen eingeschaltet haben, undefinierte Variablen höflich bemäkeln.

Ich plaudere hier einfach so über den undefinierten Wert. Im Augenblick müssen Sie darüber nicht viel wissen. Später (genauer gesagt an Tag 4), lernen Sie mehr darüber und wie man eine Variable

auf Definiertheit überprüft (mit der `defined`-Funktion) oder undefiniert macht (mit der `undef`-Funktion).

Um einer Variable einen Wert zuzuweisen, verwenden Sie einen Zuweisungsoperator. Der häufigste ist das Gleichheitszeichen (=). Dieser Zuweisungsoperator weist ganz simpel einer Variablen einen Wert zu, mit dem Variablennamen auf der linken und dem Wert auf der rechten Seite, wie hier:

```
$i = 1;
```

Zuweisungsausdrücke werden von rechts nach links ausgewertet. So können Sie Zuweisungen auch aneinanderreihen wie in folgendem Beispiel (dabei wird `$b` der Wert 4 zugewiesen und danach erhält `$a` den Wert dieses Ausdrucks, also 4):

```
$a = $b = 4;
```

Perl hat viele C-ähnliche Kurzschreibweisen mit Zuweisungsoperatoren, die ich morgen näher beschreiben werde.

Wo gelten Variablen? Eine Variable im Hauptkörper eines Perl-Skripts ist global im Gültigkeitsbereich dieses Skripts (sie steht allen Teilen des Skripts zur Verfügung). Perl erlaubt Ihnen, lokale Variablen innerhalb von Subroutinen und Schleifen einzurichten. Außerdem können Sie Variablen auch mit Packages über mehrere Skripts hinweg verwenden, Sie müssen sie aber besonders deklarieren, damit sie nicht global sind. Aber das soll heute noch nicht Thema sein. Mehr über lokale Variablen lernen Sie am Tag 11, über Gültigkeitsbereiche am Tag 13.

Aufbau von Perl-Skripts

In der gestrigen Lektion habe ich skizziert, wie ein Perl-Skript eigentlich aussieht und wie Perl es ausführt. Verlassen wir kurz das Thema Daten und gehen etwas mehr ins Detail, zu den Grundregeln für die Anordnung von Daten, Variablen und anderen Operationen in Perl-Anweisungen und von Perl-Anweisungen in Skripts.

Perl-Skripts bestehen aus einer oder mehreren Anweisungen, die in der Regel nacheinander ausgeführt werden. Perl-Anweisungen können einfache Anweisungen sein, wie Variablenbelegung oder Ausdrücke, auf die wir etwas später am heutigen Tag noch zu sprechen kommen werden. Es können auch komplexere Anweisungen wie Bedingungen und Schleifen sein, mit denen wir uns am Tag 6 befassen. Einfache Anweisungen müssen mit einem Semikolon aufhören.

Abgesehen von dieser Semikolonregel kümmert sich Perl nicht groß um leere Stellen (Leerzeichen, Tabs, Zeilenvorschübe), solange es begreift, was Sie zu tun versuchen. Sie können ein komplettes Perl-Skript aus mehreren Anweisungen in eine einzige Zeile schreiben (und in der Tat sind Perl-Einzeiler, ausgeführt von der Kommandozeile, sogar weit verbreitet). Im allgemeinen werden Perl-Skripts aber in mehrere Zeilen geschrieben, eine Anweisung pro Zeile, mit einer gewissen Ordnung beim Zeileneinzug, um die Lesbarkeit zu verbessern. Wie Sie Ihren Quelltext einziehen, liegt ganz bei Ihnen, obwohl Perl-Programmierer beabsichtigen, sich auf einen C-ähnlichen Stil zu einigen. (Die ***perlstyle***-Manpage enthält Vorschläge, wie Larry Wall seinen Quelltext formatiert und ist auf jeden Fall lesenswert, auch wenn Sie einen anderen Formatierungsstil vorziehen.)

Perl-Anweisungen können auch Ausdrücke enthalten, wobei ein Ausdruck ganz einfach etwas ist, das einen Wert zum Ergebnis hat. `1 + 1` ist ein Ausdruck (der als 2 ausgewertet wird). Eine Variablenzuweisung (`$a = 1` zum Beispiel) ist ein Ausdruck, der den Wert des zugewiesenen Dings ergibt (also 1 zum Beispiel). Perl-Ausdrücke können überall dort verwendet werden, wo ein Wert erwartet wird, auch innerhalb von anderen Ausdrücken.

Arithmetische Operatoren

Operatoren sind nicht gerade eines der aufregendsten Perl-Themen, aber Sie brauchen sie zum Aufbauen von Ausdrücken. Perl stellt für Ausdrücke mit Skalaren einen recht robusten Satz von Operatoren zur Verfügung. Sie werden heute einige dieser Operatoren kennenlernen, andere erst morgen, am Tag 3.

Wir fangen mit den Rechenoperatoren an, die arithmetische Operationen auf numerischen Daten ausführen (man könnte auch sagen: die mit Zahlen rechnen). Strings werden bei Bedarf in Zahlen umgewandelt. Für Grundrechenoperationen stehen in Perl die in Tabelle 2.2 aufgeführten Operatoren zur Verfügung, bei denen die Operanden normalerweise auf beiden Seiten des Operators stehen, ganz wie man es erwarten würde.

Operator	Operation	Beispiel	Ergebnis
+	Addition	3 + 4	7
-	Subtraktion (2 Operanden)	4 - 2	2
-	Negation eines Operanden	-5	-5
*	Multiplikation	5 * 5	25
/	Division	15 / 4	3.75
**	Potenzieren	4**5	1024
%	Modulo (Rest)	15 % 4	3

Tabelle 2.2: Rechenoperatoren

Wenig hiervon sollte Sie überraschen, obwohl der Exponentoperator neu sein könnte. Für Potenzen ist der linke Operand die Basis und der rechte der Exponent, also `10**3` ist identisch mit `10E3` oder ***10 hoch 3.***

Für die Rangfolge der Operatoren gilt, was Sie bereits in der Schule gelernt haben: Multiplikation, Division und Modulo haben Vorrang vor Addition und Subtraktion. Allerdings hat die Negation einen höheren Rang als die Multiplikation, und die Potenz hat einen sogar noch höheren Rang (höherer Rang bedeutet, dass diese Ausdrucke zuerst ausgewertet werden). Sie werden morgen, am Tag 3, noch mehr über die Rangfolge der Operatoren erfahren.

Berechnungen und Nachkommapräzision

Alle Berechnungen werden in Perl mit Fließkommazahlen durchgeführt. Obwohl dies für simple Mathematik sehr bequem ist (man braucht sich nicht um das Konvertieren zwischen Integern und Fließkommazahlen zu kümmern), hat die Fließkommamathematik ein paar Haken, auf die Sie achten sollten.

Der erste ist, dass Divisionen immer Fließkommadivisionen sind. Der Ausdruck `15/4` hat das Ergebnis 3,75 und nicht 3 wie bei der Integerdivision (und was Sie erwarten würden, wenn Sie von C kommen). Wenn Sie wirklich ein ganzzahliges Ergebnis haben wollen, können Sie die Funktion ***int*** verwenden, um die Nachkommastellen zu entfernen, wie hier:

```
$ergebnis_als_int = int 15 / 4;          # Ergebnis: 3
```

Ein anderer Nebeneffekt der Fließkommadivision ist, dass Ihr Ergebnis manchmal viel präziser ist, als Sie möchten. Nehmen Sie zum Beispiel den simplen Ausdruck:

```
print 10 / 3;
```

Dieser Ausdruck hat die Zahl `3.33333333333333` zum Ergebnis. Das ist schön und gut, wenn Sie die Zahl `3.33333333333333` auch haben wollten, aber nicht wenn Ihnen `3.33` oder `3.3` genügt.

Perl hat keine eingebaute Funktion zum Runden, aber zwei ***print***-Funktionen, mit denen man sich behelfen kann: `printf` und `sprintf`, entliehen aus C, werden verwendet, um einen numerischen Wert innerhalb eines Strings zu formatieren. Die Funktion `printf` gibt wie `print` den Wert auf den Bildschirm aus, während `sprintf` lediglich einen String zurückgibt, den Sie einer Variablen zuweisen oder in einem anderen Ausdruck weiterverwenden können. Weil Perl fröhlich zwischen Zahlen und Strings hin- und herkonvertiert, können Sie mit diesen beiden Funktionen auch Zahlen »runden«. Um zum Beispiel `3.33333333333` als Zahl mit nur zwei Nachkommastellen auf den Monitor auszugeben, schreiben Sie:

```
printf("%.2f", 10/3);
```

Der `%.2f` Teil dieses Ausdrucks ist der wichtige; er sagt: »Drucke eine Fließkommazahl (`f`) mit `2` Dezimalstellen nach dem Komma (`.2`).«

Um einen Wert innerhalb Ihres Perl-Skripts zu »runden«, ohne etwas auf den Bildschirm auszugeben, verwenden Sie `sprintf` anstelle von `printf`:

```
$wert = sprintf("%.2f", $wert);        # runde $wert auf 2 Nachkommastellen
```

Morgen lernen Sie noch ein wenig mehr über `printf` und `sprintf`.

Der letzte Haken, den man zu Fließkommaberechnungen noch erwähnen muss, ist der sogenannte Rundungsfehler. Durch die Art, wie Fließkommazahlen gespeichert werden, können manchmal sehr simple Fließkommaberechnungen Ergebnisse haben, die zwar sehr nah dran, aber doch nicht das sind, was man erwarten würde. Zum Beispiel ergibt eine simple Operation wie `4.5 + 5.7` womöglich die Zahl `10.199999999999999` anstatt `10.2`, wie man (guten Rechts) annehmen könnte. Meistens ist das kein Problem, da Perl die Zahlen intern im Auge behält und `print` beim Ausgeben der Zahl sehr kleine Ungenauigkeiten selbständig ausgleicht. Wirklich problematisch wird es jedoch, wenn Sie versuchen, einen derartigen Ausdruck mit einer Konstanten zu vergleichen - ein Test, ob der Ausdruck `4.5 + 5.7` gleich `10.2` ist, könnte als falsch scheitern. Behalten Sie diesen Rundungsfehler im Kopf, wenn Sie mit Perl arbeiten - und achten Sie ganz besonders darauf, wenn Sie plötzlich unerwartete Ergebnisse erhalten.

Ein Beispiel: Fahrenheit in Celsius umrechnen

Mit Zahlen, Strings und Skalarvariablen im Repertoire können Sie nun anfangen, simple Perl-Skripts zu schreiben. Hier ist ein Skript, das Sie um einen Fahrenheit- Wert bittet und diesen dann in Celsius umrechnet. Wenn es läuft, sieht das Ganze dann so aus:

```
% temperatur.pl
Geben Sie eine Temperatur in Fahrenheit ein: 212
212 Grad Fahrenheit entsprechen 100 Grad Celsius
%
```

Listing 2.1 zeigt den Perl-Code für dieses Skript:

Listing 2.1: Das Skript temperatur.pl.

```
1:  #!/usr/bin/perl -w
2:
3:  $fahr = 0;
4:  $cel = 0;
5:
6:  print 'Geben Sie eine Temperatur in Fahrenheit ein: ';
7:  chomp ($fahr = <STDIN>);
8:  $cel = ($fahr - 32) * 5 / 9;
9:  print "$fahr Grad Fahrenheit entsprechen ";
10: printf("%d Grad Celsius\n", $cel);
```

Dieses Skript ist dem ***echo.pl-*** Skript von gestern sehr ähnlich, aber lassen Sie es uns Zeile für Zeile durchgehen, auf der Grundlage dessen, was Sie bisher über Skalare, Zahlen, Strings und Variablen gelernt haben.

Die Zeilen 3 und 4 initialisieren die Variablen, die wir in diesem Skript verwenden werden: `$fahr` für einen Fahrenheit-Wert und `$cel` für einen Grad-Celsius-Wert. Obwohl wir das Skript auch ohne Initialisierung der Variablen hätten schreiben können, haben wir so, wenn es fertig ist, eine nette Liste von allen Variablen in diesem Skript. Schön übersichtlich.

Zeile 6 schreibt die Eingabeaufforderung auf den Bildschirm. Hier habe ich einen ***single-quoted*** String genommen, weil es hier keine Variablen oder Escape-Sequenzen gibt, über die man sich Sorgen machen müßte, nur Buchstaben. Deswegen stehen hier nur einfache Anführungszeichen.

Zeile 7 liest eine Zeile Benutzereingaben von der Tastatur und speichert diesen String in der Skalarvariablen `$fahr`. Die Funktion `chomp` schneidet das Zeilenvorschubzeichen von diesem String ab. Diese Zeile ist vielleicht noch etwas schwierig, aber bleiben Sie dran, morgen erfahren Sie mehr über die Eingabe und die `chomp`-Funktion.

In Zeile 8 führen wir die Umrechnung des Wertes in `$fahr` durch und speichern das Ergebnis in der Skalarvariablen `$cel`. Beachten Sie, dass, obwohl die von der Tastatur eingelesenen Daten String-Form haben, Perl sich daraus nichts macht. Sie können einfach weitermachen und Berechnungen mit diesen Daten ausführen, als ob es Zahlen wären. Wenn Sie etwas Nichtnumerisches wie »philantrophisch« eingeben, werden die Perl-Warnungen natürlich mäkeln. Am Tag 6 werden Sie mehr darüber lernen, wie man Eingabefehler abfängt; für heute wollen wir einfach annehmen, dass alle Eingaben, die wir bekommen, auch die richtige, von uns erwartete Form haben.

Ohne die Klammern würden Multiplikation und Division zuerst durchgeführt. Wenn die Ausdrücke in einer anderen Reihenfolge ausgewertet werden sollen, dann setzen Sie Klammern. Hier haben wir Klammern verwendet, damit `32` vom Wert in `$fahr` subtrahiert wird, bevor man es mit `5 / 9` multipliziert.

Schließlich, in Zeile 9 und 10, geben wir das Ergebnis aus. In Zeile 6 nutzen wir die mittlerweile vertraute Funktion `print` (beachten Sie, dass die `$fahr`-Variable wegen Perls automatischer Variableninterpolation durch ihren aktuellen Wert ersetzt wird). In Zeile 10 verwenden wir zur Ausgabe die Funktion `printf`, damit wir bestimmen können, ***wie*** die Celsius-Temperatur ausgegeben wird. Mit einem normalen `print` wäre der Wert von `$cel` eine Fließkommazahl - und vielleicht eine sehr lange, je nachdem, wie die Berechnung ausgegangen ist. Durch den Einsatz von `printf` mit dem Format `%d` können wir den Celsius-Wert auf eine dezimale (ganze) Zahl begrenzen. Beachten Sie, dass, obwohl wir hier zwei Print-Anweisungen haben (zum einen `print`, zum anderen `printf`), die Ausgabe in einer einzelnen Zeile steht. Der erste ausgegebene String hat nicht mit `\n` aufgehört, also geht der zweite String in derselben Zeile weiter.

Morgen erfahren Sie mehr über `print` und `printf`.

Operatoren für Tests und Vergleiche

Mit Perls Vergleichsoperatoren erhalten Sie eine Aussage über die Beziehung zwischen zwei Zahlen oder Strings. Gleichheitsoperatoren prüfen, ob zwei Skalare gleich sind, relationale (oder Vergleichs-)Operatoren, ob einer »größer« ist als der andere. Schließlich gibt es in Perl auch noch logische Operatoren für boolesche (*wahr* oder *falsch*) Vergleiche. Diese brauchen Sie insbesondere für Bedingungs- und Schleifenoperationen, die wir am Tag 6 behandeln.

Was ist Wahrheit?

Nein, wir schweifen jetzt nicht in philosophische Debatten ab, doch bevor wir die Operatoren durchgehen, müssen Sie verstehen, was Perl unter ***wahr*** und ***falsch*** versteht.

Zum einem können jegliche skalare Daten auf ihre Wahrheit überprüft werden; das heißt Sie können nicht nur nachsehen, ob zwei Zahlen gleichwertig sind, sie können auch bestimmen, ob die Zahl `4` oder der String `"Thomas Jefferson"` wahr ist. Die Regel ist ganz einfach: Alle Skalare (alle Zahlen, Strings und Referenzen) sind ***wahr*** bis auf drei Ausnahmen:

- Der leere String (`""`)
- Null (`0`)
- Der undefinierte Wert (der sowieso meistens wie `""` oder `0` aussieht)

Diese drei sind ***falsch***.

Mit diesen Regeln im Kopf können Sie sich jetzt den Operatoren zuwenden.

Gleichheits- und Vergleichsoperatoren

Gleichheitsoperatoren überprüfen, ob zwei Daten gleich sind, relationale Operatoren ermitteln, ob ein Wert größer ist als der andere. Bei Zahlen ist das einfach: Verglichen wird in numerischer Reihenfolge. Bei Strings wird ein String für kleiner als ein anderer befunden, wenn dessen erstes Zeichen in der Reihe der ASCII-Zeichen früher

auftaucht als das erste Zeichen des anderen Strings (und umgekehrt für größer als). Im ASCII-Zeichen-Schema ist die Reihenfolge der Zeichen durch den zugehörigen ASCII- Code, eine Nummer von 0-255, festgelegt, auch für Zahlen, Leerstellen und Sonderzeichen, wobei die Großbuchstaben vor den Kleinbuchstaben kommen. Strings sind dann gleich, wenn sie vom Anfang bis zum Ende aus exakt den gleichen Zeichen bestehen.

Perl hat zwei Sätze von Gleichheits- und Vergleichsperatoren, einen für Zahlen und einen für Strings, wie in Tabelle 2.3 aufgelistet. Obwohl sie unterschiedliche Namen haben, werden Sie beide gleich verwendet, mit je einem Operand auf jeder Seite. All diese Operatoren geben `1` für *wahr* und `""` für *falsch* zurück.

Vergleich	Zahlenoperator	String-Operator
gleich	==	eq
nicht gleich	!=	ne
kleiner als	<	lt
größer als	>	gt
kleiner gleich	<=	le
größer gleich	>=	ge

Tabelle 2.3: Gleichheits- und Vergleichsoperatoren

Hier ein paar Beispiele für Zahlen- und String-Vergleiche:

```
4 < 5            # wahr
4 <= 4           # wahr
4 < 4            # falsch
5 < 4 + 5        # wahr (Addition zuerst ausgefuehrt)
6 < 10 > 15      # Syntaxfehler; Vergleiche sind nicht kombinierbar
'5' < 8          # wahr; '5' wird zu 5 konvertiert
'laut' < 'lauter'  # fuehrt unter -w zu Fehler. für Strings lt nehmen
'laut' lt 'lauter' # wahr
'laut' lt 'Laut'   # falsch; Gross- und Kleinschreibung
                   # wird unterschieden
'laut' eq 'laut '  # falsch, Leerzeichen spielen eine Rolle
```

Beachten Sie, dass keiner der Vergleichsausdrücke mit anderen Vergleichsausdrücken kombiniert werden kann. Während Sie einen Rechenausdruck `5 + 4 - 3` haben können, der von links nach rechts ausgewertet wird, ist ein Ausdruck `6 < 10 > 15` nicht möglich; er wird einen Syntaxfehler verursachen, weil er nicht auszuwerten ist.

Seien Sie vorsichtig, und vergessen Sie nicht, dass == der Gleichheitstest ist, nicht zu verwechseln mit =, dem Zuweisungsoperator. Letzterer ist ein Ausdruck, der ebenfalls *wahr* oder *falsch* zurückgeben kann, deshalb kann es sehr schwierig sein, Fehler aufzuspüren, wo Sie aus Versehen = statt == benutzt haben.

Sie fragen sich, wofür es zwei verschiedene Sätze von Vergleichsoperatoren gibt, wo Perl doch zwischen Zahlen und Strings automatisch konvertieren kann? Genau deswegen. Gerade weil Zahlen und Strings automatisch konvertiert werden können, brauchen Sie die Möglichkeit zu sagen: »Vergleiche die hier als Zahlen und nicht als String.«

Nehmen wir zum Beispiel an, es gäbe nur einen Satz Vergleichsoperatoren in Perl, wie in anderen Sprachen. Und sagen wir, wir hätten einen Ausdruck wie `'5' < 100`. Wie wird dieser Ausdruck ausgewertet? In anderen Sprachen wäre dieser Vergleich nicht einmal möglich, es wäre ein ungültiger Ausdruck. Nicht so in Perl, denn Zahlen und Strings können ineinander umgewandelt werden. Aber es gibt zwei Möglichkeiten, es korrekt auszuwerten. Wenn man `'5'`in eine Zahl konvertiert, erhält man `5 < 100`, das ist wahr. Wandelt man `100` in einen String um, erhält man `'5' < '100'`, das ist falsch, weil in der ASCII-Reihenfolge das Zeichen `5` erst nach dem Zeichen `1` steht. Um diese Zweideutigkeit zu vermeiden, brauchen wir zwei Sätze von Operatoren.

Zu vergessen, dass es zwei Sets von Vergleichsoperatoren gibt, ist einer der häufigeren Fehler bei Perl-Programmieranfängern (und einer der einen wahnsinnig machen kann, wenn zum Beispiel `'dies' == 'das'` beide Strings zu `0` konvertiert und dann *wahr* zurückgibt). Aktivierte Perl-Warnungen werden Sie gegebenenfalls auf

diesen Fehler aufmerksam machen (noch ein guter Grund, Warnungen in all Ihren Skripts angeschaltet zu lassen, zumindest bis Sie Ihren Programmierfähigkeiten ein bißchen mehr trauen.

Logische Operatoren

Logische Operatoren, auch boolesche Operatoren genannt, werten ihre Operanden auf den Grundlagen der Booleschen Algebra aus, geben also nach deren Regeln ***wahr*** oder ***falsch*** zurück. Das heißt für die Werte von x und y:

- `x AND y` liefert nur dann *wahr*, wenn sowohl x als auch y wahr sind.
- `x OR y` liefert dann *wahr*, wenn mindestens einer von beiden wahr ist.
- `NOT x` liefert *wahr*, wenn x falsch ist und umgekehrt.

Sie müssen `AND`, `OR` und `NOT` nicht groß schreiben. Die Großschreibung ist aber sehr gebräuchlich, um die Operatoren von den gleichnamigen Wörtern der englischen Sprache zu unterscheiden.

Typisch für Perl (die Qual der Wahl ist schließlich ein wichtiges Perl-Feature): Es gibt nicht nur einen Satz logischer Vergleiche, sondern zwei: einen aus C entliehenen und einen mit Perl-Schlüsselwörtern. Tabelle 2.4 zeigt beide Sätze dieser Operatoren:

C-Stil	Perl-Stil	Bedeutung
&&	AND	logisches UND
||	OR	logisches ODER
	NOT	logisches NICHT

Tabelle 2.4: Logische Vergleiche

Es gibt auch Operatoren für logisches XOR (entweder oder), `^` und `XOR`, aber im allgemeinen werden sie außerhalb von Bitmanipulation kaum gebraucht, deswegen habe ich sie hier nicht mit hineingenommen.

Der einzige Unterschied zwischen den zwei Operatorstilen liegt in ihrer Rangfolge. Die C-Stil-Operatoren stehen höher in der Rangfolge als die Perl-Stil-Operatoren. Dieser niedrige Rang der Perl-Stil-Operatoren kann Ihnen das Eintippen von ein paar Klammern ersparen, wenn diese Sie nerven. In vorhandenem Perl-Code werden Sie wahrscheinlich häufiger C-Stil-Operatoren finden. Die Perl-Stil-Operatoren sind zum einen recht neu, und zum anderen verwenden Programmierer, die an C gewöhnt sind, lieber die C-Stil-Codierung.

Beide Arten von logischem AND und NOT sind sogenannte Kurzschlußoperatoren, das heißt, wenn nach Auswerten der linken Seite des Ausdrucks das Endergebnis schon feststeht, wird die rechte Seite des Ausdrucks komplett ignoriert.

Sagen wir zum Beispiel, Sie hätten folgenden Ausdruck:

```
($x < $y) && ($y < $z)
```

Wenn der Ausdruck auf der linken Seite von `&&` falsch ist (wenn `$x` größer als `$y` ist), ist der Ausgang der rechten Seite des Ausdrucks irrelevant. Ganz egal, was dort herauskommt, der gesamte Ausdruck wird falsch sein (denn, Sie erinnern sich, logisches UND besagt, dass beide Seiten des Ausdrucks wahr sein müssen, damit der Ausdruck wahr sein kann). Um etwas Zeit zu sparen, verzichtet ein Kurzschlußoperator auf die Auswertung der rechten Seite des Ausdrucks, wenn die linke Seite (und damit der gesamte Ausdruck) falsch ist.

Ganz ähnlich wird bei ||, wenn die linke Seite des Ausdrucks als wahr ausgewertet wird, der ganze Ausdruck für wahr befunden, und die rechte Seite wird ignoriert.

Beide Formen von logischen Operatoren liefern ***falsch***, wenn sie zum Ergebnis falsch gekommen sind. Ist das Ergebnis ***wahr***, haben sie außerdem noch den Seiteneffekt, dass sie den zuletzt ausgewerteten Wert gleich mit zurückgeben (der, weil es keine Null oder kein Leerstring ist, noch als wahr gilt). Während dieser Nebeneffekt zuerst unsinnig erscheinen mag, wo Sie doch nur wissen wollen, ob der Wert des Ausdrucks ***wahr*** oder ***falsch*** ist, erlaubt er eine einfache Form der Auswahl zwischen verschiedenen Optionen oder Funktionsaufrufen oder ähnlichem:

```
$ergebnis = $a || $b || $c || $d;
```

In diesem Beispiel wird Perl die Liste von Variablen durchgehen und jede auf ihre »Wahrheit« überprüfen. Die erste, die ***wahr*** ergibt, hält den Ausdruck an (Kurzschluß), und der Wert von $ergebnis wird der Wert der letzten überprüften, also der ersten »wahren« Variablen.

Viele Perl-Programmier verwenden diese logischen Tests gerne als eine Art Bedingung wie im nächsten Beispiel, das Sie häufig sehen werden, wenn Sie sich den Perl-Code anderer Leute ansehen:

```
open(DATEI, 'Dateiname') || die 'kann Datei nicht oeffnen';
```

Auf der linken Seite des Ausdrucks wird mit `open` eine Datei geöffnet und ***wahr*** zurückgegeben, wenn die Datei erfolgreich geöffnet wurde. Auf der rechten Seite wird `die` (vom englischen ***to die***, sterben) eingesetzt, um das Skript bei einer Fehlermeldung sofort zu verlassen. Sie möchten das Skript natürlich nur verlassen, wenn die Datei nicht geöffnet werden konnte - das heißt, wenn `open` ***falsch*** zurückgibt. Da der || Ausdruck kurzschließt, wird es nur dann zum `die` auf der rechten Seite kommen, wenn die Datei nicht geöffnet werden konnte.

Ich werde darauf zurückkommen, wenn wir am Tag 6 über Bedingungen sprechen (und über `open` werden Sie am Tag 15 mehr lernen).

Vertiefung

Perl ist eine dermaßen umfangreiche Sprache - wenn ich wirklich ***alles*** erklären wollte, wäre dieses Buch so dick wie bestenfalls das Münchener, wahrscheinlicher aber wie das Berliner Telefonbuch (»Perl in 180 Tagen«?). In diesem Abschnitt reiße ich deswegen ein paar Sachen an, die Sie mit skalaren Daten noch machen können, ich in den vorigen Abschnitten aber nicht besprochen habe. Manches davon werden wir später in diesem Buch detaillierter betrachten, aber zum größten Teil sind dies Themen, die Sie selbst erforschen müssen, falls Sie mehr darüber wissen möchten.

Wie ich gestern erwähnt habe: Wenn ich auf die ***perlop-*** oder ***perlfunc-***Manpage **-** oder jede andere Perl-Manpage - verweise, finden Sie diese Seiten als Teil der Dokumentation bei Ihrem Perl-Interpreter oder im Web unter `http://www.perl.com/ CPAN-local/doc/manual/html/pod/`.

Anführungszeichen

Die Anführungszeichen `''` und `""` stellen in Perl lediglich eine Möglichkeit für das Erzeugen von Strings dar. Es gibt noch weitere Möglichkeiten, wie Sie in Perl auch ohne Anführungszeichen Strings erstellen können, von denen einige Variablen auswerten und andere nicht. Sie könnten zum Beispiel anstelle der einfachen Anführungszeichen bei `'Sein oder Nichtsein'` auch den Operater `q//` verwenden:

```
q/Sein oder Nichtsein/
```

Sie haben einen unerklärlichen Widerwillen gegen Anführungszeichen oder Slashes? Kein Problem. Sie können den `q//`-Operator mit jedem alphanumerischen Zeichen schreiben, solange Sie nur das gleiche Zeichen für den Anfang und das Ende des Strings nehmen und das Ganze mit einem `q` beginnen. Die folgenden Beispiele sind in Perl alle gleichwertig:

```
'Sein oder Nichtsein'
q/Sein oder Nichtsein/
```

```
q#Sein oder Nichtsein#
q?Sein oder Nichtsein?
```

Wie bei einfachen Anführungszeichen interpoliert der `q//` -Operator keine Variablen. Statt doppelten Anführungszeichen (für Variableninterpolation) können Sie in der gleichen Art und Weise den `qq//` -Operator einsetzen:

```
"zeige $zahl Zeilen"
qq/zeige $zahl Zeilen/
qq^zeige $zahl Zeilen^
```

Diese Formate sind recht praktisch, wenn man mit Strings arbeitet, die selbst viele Anführungszeichen enthalten. Mit `q` oder `qq` und einem beliebigen Begrenzungszeichen können Sie sich die Backslashes ersparen und die Anführungszeichen im String stehen lassen, wie sie sind. (Siehe auch die ***perlop-*** Manpage; dort finden Sie Details über die verschiedenen Operatoren, die statt Anführungszeichen verwendet werden können.)

Außerdem gibt es noch ***barewords*** (vom englischen ***bare***, »bloß, blank«): einzelne Wörter ohne Anführungszeichen, die in Perl keine andere Bedeutung haben, werden als Strings interpretiert. Von ihnen ist prinzipiell abzuraten, weil sie Skripts sehr schwer lesbar machen, sehr fehleranfällig sind und Sie nie wissen, wann eins Ihrer ***barewords*** vielleicht doch zum reservierten Wort in einer zukünftigen Version der Sprache wird. Vermeiden Sie sie. Bei aktivierten Perl-Warnungen wird Perl sich über sie auch beklagen.

Groß- und Kleinschreibung per Escape-Sequenz

In Tabelle 2.1 hatte ich die Sonderzeichen für interpolierte (double-quoted) Strings zusammengefaßt. Darunter waren auch ***Escape-Sequenzen*** für Groß- und Kleinschreibung (`\l, \u, \L, \U` und `\E`). Verwenden Sie diese, um die Groß- oder Kleinschreibung von Werten in Strings zu erzwingen.

Die Sonderzeichen `\l` und `\u` machen aus dem nachfolgenden Buchstaben im String einen Klein- bzw. Großbuchstaben. Sie könnten `\u` zum Beispiel verwenden, um sicherzustellen, dass ein Satzanfang groß geschrieben wird. `\L` und `\U` machen gleich eine Reihe von Zeichen innerhalb des Strings zu Groß- oder Kleinbuchstaben: Sie wandeln alle nachfolgenden Buchstaben im String entsprechend um, bis der String zu Ende ist oder ein `\E` auftaucht. Diese Sonderzeichen sind in Strings nicht unbedingt nützlich, aber sie werden es später, wenn Sie Muster für Such- und Ersetzungsprozeduren schreiben.

Mehr zur Variableninterpolation in Strings

Variableninterpolation ist die Fähigkeit von Perl, Verweise auf Variablen in Strings durch ihre aktuellen Werte zu ersetzen. Manchmal ist es allerdings für Perl schwierig zu begreifen, wo die Variable aufhört. Nehmen Sie zum Beispiel folgenden String:

```
"Dies ist ein $xes Programm...\n";
```

In diesem String heißt die zu interpolierende Variable eigentlich `$x` und enthält ein Adjektiv (das man vielleicht vorher aus »nett«, »langweilig« und »sinnlos« auswählen konnte). Weil aber der Ausgabestring etwas sagen soll wie `"Dies ist ein sinnloses Programm..."`, stößt der Variablenname direkt an das »`es`«, und Perl sucht nach einer Variablen namens `$xes` - und nicht nach `$x`. Es gibt eine Menge Auswege aus dieser Art von Problem - nicht zuletzt die Verkettung von mehreren Strings. Aber Perl hat auch eine direkte Lösung:

```
"Dies ist ein ${x}es Programm...\n";
```

Die geschweiften Klammern sind in diesem Fall einfach nur Begrenzungen, so dass Perl versteht, wo der Variablenname anfängt und endet; sie werden nicht ausgegeben. Solcher Gebrauch von geschweiften Klammern kann helfen, um Probleme bei der Variableninterpolation auszuklammern.

Zusammenfassung

Überall Zahlen und Strings! Heute haben Sie ziemlich viel über skalare Daten gelernt. Perl verwendet die Bezeichnung ***skalare Daten,*** um sich auf einzelne Objekte zu beziehen, insbesondere Zahlen und Strings.

Skalarvariablen, die mit einem $ beginnen, speichern skalare Daten.

Mit skalaren Daten können Sie Perl-Anweisungen erstellen, Berechnungen durchführen, zwei Werte vergleichen, Variablen Werte zuweisen, die Werte ändern und zwischen Zahlen und Strings hin- und herkonvertieren.

Nachfolgend noch einmal die eingebauten Funktionen, die Sie heute kennengelernt haben:

- `print` nimmt eine Liste von kommagetrennten Werten und Strings entgegen und gibt diese Werte an das Standardausgabegerät (`STDOUT`).
- `printf` nimmt Formatierungsanweisungen und beliebig viele Werte entgegen und gibt diese Werte entsprechend den Formatierungsanweisungen aus.
- `sprintf` tut dasselbe wie `printf`, nur dass es den formatierten String zurückgibt, ohne irgend etwas auszugeben.
- `chomp` mit einem String-Argument entfernt alle Zeilenvorschübe vom übergebenen String und gibt die Zahl von Zeichen, die es gelöscht hat, zurück.
- `int` übernimmt eine Zahl und gibt den ganzzahligen Teil dieser Zahl zurück (wobei es jede Nachkommastelle abschneidet).
- `oct` nimmt einen String und interpretiert ihn als Oktal- oder Hexadezimalzahl, je nachdem, ob der String mit `0` oder `0x` anfängt.
- `hex` nimmt einen String und interpretiert diesen als eine Hexadezimalzahl.

Einige dieser Funktionen werden wir morgen noch genauer untersuchen; ansonsten können Sie in der ***perlfunc***-Manpage (siehe Anhang A) nachschlagen, um mehr über diese Funktionen zu erfahren.

Fragen und Antworten

Frage:
Wie definiere ich eine Variable so, dass sie nur Zahlen enthalten kann?

Antwort:
Das können Sie nicht. Perl hat keine strengen Zahlentypen wie andere Sprachen. Die beste Näherung ist eine Skalarvariable, und die kann sowohl eine Zahl als auch einen String enthalten.

> *Meistens spielt das aber keine Rolle; Perl konvertiert für Sie Zahlen in Strings und umgekehrt.*

Frage:
Aber was ist, wenn ich irgendwann Daten in Stringform habe, die keine Zahlen sind, und ich damit irgendwelche Berechnungen vornehmen will?

Antwort:
Das Standardverhalten sieht, wie ich es im Abschnitt »Konvertieren zwischen Zahlen und Strings« beschrieben habe, so aus, dass Strings ohne numerischen Inhalt zu 0 werden. Für Strings, die mit Zahlen beginnen und dann mit Buchstaben weitergehen, werden die Buchstaben ignoriert.

> *Wenn Sie für Ihr Perl-Skript Warnungen eingeschaltet haben, dann wird Perl Sie warnen, wenn Sie versuchen, mit nichtnumerischen Daten zu rechnen, so dass Sie Ihr Skript korrigieren können, bevor das Problem überhaupt entsteht.*
>
> *Falls Sie befürchten, nichtnumerische Daten vom Anwender zu bekommen, sollten Sie die Eingaben beim Einlesen überprüfen. Wir haben dies in Ansätzen heute schon getan. An den Tagen 6 und 9 zeige ich Ihnen noch mehr Tricks zur Eingabeüberprüfung.*

Frage:
Meine Berechnungen geben Fließkommazahlen mit viel zu vielen Stellen nach dem Komma zurück. Wie beschränke ich Zahlen auf zwei Dezimalstellen?

Antwort:
Sehen Sie in dem Abschnitt über die arithmetischen Operatoren nach, wo ich erkläre, wie man genau dies mit

`printf` und `sprintf` erledigt.

Frage:
Aber `printf` und `sprintf` sind Stringfunktionen. Ich möchte sie auf Zahlen anwenden.

Antwort:
Sowohl Zahlen als auch Strings sind in Perl Skalare. Was Sie mit dem einen tun können, können Sie (in vielen Fällen) auch mit dem anderen tun. Nehmen Sie `printf` und `sprintf`.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Welche beiden Datentypen stehen Ihnen in Perl zur Verfügung?
2. Was für Daten sind skalar?
3. Was sind die Unterschiede zwischen Strings mit einfachen und Strings mit doppelten Anführungszeichen?
4. Welche der folgenden Variablen sind Skalarvariablen?

```
  $zaehler
$11foo
$_platzhalter
$back2thefuture
$lange_variable_zum_Speichern_von_wichtigen_Daten
```

5. Was ist der Unterschied zwischen den Operatoren = und ==?
6. Was ist der Unterschied zwischen einer Anweisung und einem Ausdruck?
7. Zu welchem Ergebnis wir der folgende Ausdruck ausgewertet?:

```
4 + 5 / 3**2 * 6
```

8. Wie formatiert man Zahlen in Perl?
9. Welche Werte repräsentieren in Perl den Wahrheitswert ***falsch***?
10. Warum gibt es unterschiedliche Operatoren für Zahlen- und für Stringvergleiche?
11. Erläutern Sie, was ein ***kurzschließender*** logischer Operator macht.

Übungen

1. Ändern Sie ***temperatur.pl***, so dass es Celsius zurück nach Fahrenheit konvertiert.
2. Schreiben Sie ein Programm, das Sie um die Eingabe der Länge und Breite eines Raums bittet und Ihnen dann die Größe in Quadratmetern ausgibt.
3. FEHLERSUCHE: Was ist falsch an diesem Programm?

```
    print 'Geben Sie das Wort foo ein: ';
chomp($input = <STDIN>);
if ($input = 'foo') {
   print "Danke!\n";
} else {
   print "Das ist nicht das Wort foo.\n";
}
```

4. FEHLERSUCHE: Und was ist mit dieser Version?

```
    print 'Geben Sie das Wort foo ein: ';
chomp($input = <STDIN>);
if ($input == 'foo') {
   print "Danke!\n";
} else {
```

```
    print "Das ist nicht das Wort foo.\n";
}
```

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Die zwei in Perl zur Verfügung stehenden Datentypen sind Skalare für individuelle, einzelne Dinge wie Zahlen oder Strings und Listen für Datensammlungen wie zum Beispiel Arrays.
2. Zahlen, Strings und Referenzen. Richtig ist Ihre Antwort aber auch, wenn Sie nur an Zahlen und Strings gedacht haben; über Referenzen haben wir noch nicht gesprochen.
3. Es gibt zwei Unterschiede zwischen ***single-*** und ***double-quoted*** Strings:
1. ***Double-quoted*** Strings können beliebig viele Sonderzeichen enthalten, ***single- quoted*** Strings nur `\'` und `\\`.
1. ***Double-quoted*** Strings werten die in ihnen enthaltene Variablen aus (das heißt, sie ersetzen die Variablen durch deren aktuellen Wert).
4. Alle Variablen in dieser Liste außer `$11foo` sind gültig. `$11foo` ist ungültig, weil es mit einer Zahl anfängt. (Perl kennt Variablen, die mit Zahlen beginnen, aber die sind alle für Perl reserviert).
5. Der =-Operator ist der Zuweisungsoperator, mit dem man einer Variablen einen Wert zuweist. Mit dem ==-Operator überprüft man die Gleichheit von zwei Zahlen.
6. Eine Anweisung ist in Perl eine einzelne Operation. Ein Ausdruck ist eine Anweisung, die einen Wert zurückgibt. Sie können oft in einer einzigen Perl- Anweisung etliche Ausdrücke verschachteln.
7. `7.33333333333333`, mit ein oder zwei Dreien mehr oder weniger.
8. Zum einfachen Abrunden von Zahlen (ohne sie auszugeben) verwenden Sie `sprintf`. Mit `printf` geben Sie Zahlen weniger präzise aus.
9. In Perl entsprechen drei Werte dem Wahrheitswert ***falsch***: `0`, der Leerstring `""` und der undefinierte Wert.
10. Perl hat unterschiedliche Operatoren für Zahlen und Strings, weil es skalare Werte automatisch konvertiert, mit beiden aber unterschiedlich umgehen muss.
11. Kurzschlußoperatoren werten die Operanden auf ihrer rechten Seite nur soweit wie nötig aus. Wenn der Wert links vom Operator schon den Wert des gesamten Ausdrucks bestimmt (zum Beispiel wenn die linke Seite von einem `&&` -Operator ***falsch*** ist), wird die Auswertung abgebrochen.

Lösungen zu den Übungen

1. Eine mögliche Lösung ist:

```
    #!/usr/bin/perl -w
$cel = 0;
$fahr = 0;
print 'Geben Sie eine Temperatur in Celsius ein: ';
chomp ($cel = <STDIN>);
$fahr = $cel * 9 / 5 + 32;
print "$cel Grad Celsius entsprechen ";
printf("%d Grad Fahrenheit \n", $fahr);
```

2. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
$breite = 0;
$laenge = 0;
$qm = 0;
print 'Geben Sie die Breite des Raums ein (Meter): ';
chomp ($breite = <STDIN>);
print 'Geben Sie die Laenge des Raums ein (Meter): ';
chomp ($laenge = <STDIN>);
$qm = $breite * $laenge;
print "Der Raum hat $qm Quadratmeter.\n";
```

3. Die Überprüfung innerhalb der Klammern für die `if`-Anweisung verwendet einen Zuweisungsoperator anstatt eines Gleichheitsoperators. Dieser Test wird immer wahr sein.

4. Diesmal prüft der Test innerhalb der Klammen auf Zahlengleichheit. Da hier aber Strings verglichen werden, braucht man statt dessen eine `eq` -Prüfung.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Weitere Skalare und Operatoren

Ein Skalar ist, wie Sie seit spätestens gestern wissen, etwas Einzelnes wie eine Zahl oder ein String. Gestern haben Sie schon einiges darüber erfahren, was man mit skalaren Daten und Operatoren machen kann. Heute werde ich Ihnen noch etwas mehr zeigen und das Thema abschließen. Am Ende befassen wir uns mit ein paar verwandten Themen. Heute steht folgendes an:

- Verschiedene Zuweisungsoperatoren
- Stringverkettung und -wiederholung
- Rangfolge der Operatoren
- Ein kurzer Überblick über Ein- und Ausgabe

Zuweisungsoperatoren

Gestern hatten wir bereits den grundlegenden Zuweisungsoperator = kennengelernt, der einer Variablen einen Wert zuweist. Zuweisungen verwendet man sehr häufig zur Änderung des Werts einer Variablen basierend auf ihrem aktuellen Wert.

```
$inc = $inc + 100;
```

zum Beispiel macht genau, was man erwarten würde: Es nimmt den Wert von `$inc`, addiert `100` dazu und speichert das Ergebnis dann wieder in `$inc`. Derartige Operationen kommen so oft vor, dass es dafür eine Kurzschreibweise, einen eigenen Operator gibt: Die Variablenbezeichnung kommt auf die linke Seite, in die Mitte der Operator += und auf die rechte Seite der Betrag, um den die Variable geändert werden soll:

```
$inc += 100;
```

Perl hat solche »Schnellzuweisungen« für alle Rechenoperatoren, für die (bis jetzt noch nicht besprochenen) Stringoperatoren und sogar für `&&` und `||`. Tabelle 3.1 zeigt ein paar dieser Zuweisungsoperatoren. Im Grunde hat so gut wie jeder Operator, der zwei Operanden hat, eine solche Zuweisungsversion. Dabei gilt allgemein:

```
Variable Zuweisungsoperator Ausdruck
```

ist äquivalent mit

```
Variable = Variable Operator Ausdruck
```

Es gibt nur einen Unterschied zwischen den beiden Formen. In der langen Version wird die Variable zweimal ausgewertet, in der Kurzversion nur einmal. Meistens macht das beim Ergebnis des Ausdrucks keinen Unterschied, aber denken Sie daran, wenn Sie einmal unerwartete Ergebnisse erhalten.

Operator	Beispiel	gleichbedeutend mit
+=	$x += 10	$x = $x + 10
-=	$x -= 10	$x = $x - 10
*=	$x *= 10	$x = $x * 10
/=	$x /= 10	$x = $x / 10
%=	$x %= 10	$x = $x % 10
**=	$x **= 10	$x = $x**10

Tabelle 3.1: ***Einige gebräuchliche Zuweisungsoperatoren***

Inkrement- und Dekrementoperatoren

Wie in C kann man mit den Operatoren ++ und -- eine Variable um 1 inkrementieren bzw. dekrementieren, das heißt, 1 addieren bzw. subtrahieren. Und wie in C können beide Operatoren entweder als Präfix (vor der Variablen, also ++$x) oder als Postfix (nach der Variablen, also $x++) gesetzt werden. Abhängig davon wird die Variable, bevor oder nachdem sie verwendet wurde, inkrementiert oder dekrementiert.

»Wie bitte?« denken Sie? Dann erkläre ich es Ihnen genauer: Die Operatoren ++ und -- verwendet man mit einer Skalarvariablen, um den Wert der Variablen um 1 zu erhöhen oder zu erniedrigen - sie sind eine Art Abkürzung der +=-Operatoren oder der -=-Operatoren. Beide Operatoren können vor der Variablen stehen - das wird dann Präfixnotation genannt:

```
++$x
```

Oder nach der Variablen (Postfixnotation):

```
$x++
```

Der Unterschied ist klein, aber fein: Die Position des Operators bestimmt, wann Perl beim Auswerten des Ausdrucks die Variable wirklich inkrementiert. Wenn Sie die Operatoren ganz alleinstehend - nur mit der Variablen, auf der operiert wird - verwenden, gibt es keinen Unterschied zwischen Postfix und Präfix. Die Variable wird inkrementiert, und Perl geht weiter. Wenn diese Operatoren aber auf der rechten Seite einer anderen Variablenzuweisung stehen, können Präfix- und Postfixnotation zu verschiedenen Ergebnissen führen. Schauen wir zum Beispiel auf das folgende Schnipsel Perl-Code:

```
$a = 1;
$b = 1;
$a = ++$b;
```

Nach diesen Anweisungen haben sowohl $a als auch $b den Wert 2. Warum? Die Präfixnotation bedeutet, dass der Wert von $b inkrementiert wird, bevor er $a zugewiesen wird. In diesem Ausdruck wird also zuerst $b auf 2 erhöht und dieser Wert dann $a zugewiesen.

Als Postfix sähe das anders aus:

```
$a = 1;
$b = 1;
$a = $b++;
```

Auch hier wird $b inkrementiert und erhält den Wert 2. Aber der Wert von $a bleibt 1. In der Postfixnotation wird der Wert von $b nämlich verwendet, bevor er inkrementiert wird. Perl wertet $b als 1 aus, weist diesen Wert $a zu und erhöht erst dann $b auf 2.

Um ganz und gar korrekt zu sein: Meine Reihenfolge, wie die Dinge passieren, ist falsch. Bei einer Variablenzuweisung muss alles auf der rechten Seite des =-Operators vor der Zuweisung ausgewertet werden. In Wirklichkeit wird $a also bis zum allerletzten Schritt gar nicht verändert, sondern Perl merkt sich den ursprünglichen Wert von $b, inkrementiert ganz brav und greift dann erst bei der Zuweisung an $a auf den Originalwert von $b zurück. Aber wenn Sie nicht mit wirklich komplexen Ausdrücken arbeiten, können Sie sich das Ganze auch so vorstellen, dass die Zuweisung vor der Inkrementierung stattfindet.

Stringverkettung und -wiederholung

String- und Textmanagement ist eine von Perls größten Stärken, und etliche der Beispiele in diesem Buch befassen sich mit der Arbeit mit Strings - in Strings suchen, Strings ändern, sie aus Dateien oder von der Tastatur lesen und sie an den Bildschirm, an Dateien oder über ein Netzwerk an einen Webbrowser schicken. Wir haben mit dem Thema erst angefangen und reden noch ganz allgemein über Strings.

Weil es ganz gut in die Lektion von heute paßt, möchte ich hier Perls Stringoperatoren erwähnen: `.` (Punkt) für Stringverkettung und `x` für Stringwiederholung

Anders als Java, das den +-Operator sowohl für die Addition als auch für die Stringverkettung nutzt, reserviert Perl + für die Verwendung mit Zahlen. Um zwei Strings aneinanderzufügen, verwenden Sie den .-Operator wie hier:

```
'es war einmal' . ' vor langer Zeit';
```

Dieser Ausdruck erstellt einen dritten String: `'es war einmal vor langer Zeit'`. Er ändert keinen der Originalstrings.

Sie können auch mehrere Strings zu einem einzelnen langen String verketten:

```
'eene ' . 'meene ' . 'muh... ' . 'und raus bist du!'
```

Der andere Stringoperator ist der x-Operator (nicht der X-Operator, es muss ein kleines x sein). Der x-Operator hat einen String auf der einen und eine Zahl auf der anderen Seite (aber wird beides bei Bedarf konvertieren) und erstellt dann einen neuen String: Der String links wird so oft wiederholt, wie die Zahl rechts vorgibt. Ein paar Beispiele:

```
'blah' x 4;  # 'blahblahblahblah'
'*' x 3;   # '***'
10 x 5;   # '1010101010'
```

In diesem letzten Beispiel wird die Zahl `10` zum String `'10'` konvertiert und dann fünfmal wiederholt.

Wofür soll das gut sein? Nehmen Sie an, Sie bräuchten für ein Bildschirmlayout eine bestimmte Zahl von Leerstellen oder Füllzeichen, wobei es aber möglich sein soll, dass die Breite dieses Layouts variiert. Oder stellen Sie sich vielleicht eine Art ASCII-Kunst vor, wo die Wiederholung von Buchstaben besondere Muster produziert (das hier ist Perl, Sie dürfen so eigenartiges Zeug machen, nur zu). Sollten Sie jemals einen String wiederholen müssen, egal wie oft, kann der x-Operator das für Sie erledigen.

Rangfolge und Assoziativität der Operatoren

Der ***Vorrang*** (***precedence***) der Operatoren legt fest, welche Operatoren in einem komplexen Ausdruck zuerst ausgewertet werden. Die ***Assoziativität*** bestimmt, wie gleichrangige Operatoren ausgewertet werden (linksassoziative Operatoren von links nach rechts, rechtsassoziative von rechts nach links, und bei nichtassoziativen Operatoren ist die Reihenfolge der Auswertung entweder nicht wichtig, nicht garantiert oder nicht einmal möglich). Tabelle 3.2 zeigt Vorrang und Assoziativität der verschiedenen Perl-Operatoren, wobei Operatoren mit höherem Vorrang (zuerst ausgewertet) in der Tabelle höher stehen als die mit niedrigerem Vorrang (später ausgewertet). Dies ist eine der Tabellen, die Sie bei der Arbeit mit Perl wahrscheinlich immer wieder brauchen werden - ich empfehle Ihnen, diese Seite wie auch immer zu markieren. Auswendig können müssen Sie das alles nicht, aber Sie sollten es schnell parat haben.

Sie können die Auswertungsreihenfolge eines Ausdrucks ändern, indem Sie Klammern setzen. Ausdrücke in Klammern werden vor denen außerhalb der Klammern ausgewertet.

Und noch etwas: In der Tabelle sind auch ein paar Operatoren aufgeführt, die Sie noch nicht kennen (und manche, die ich in diesem Buch nicht einmal behandeln werde). Ich habe bei den Operatoren, die ich später in diesem Buch noch erkläre, Verweise auf die entsprechenden Lektionen eingefügt.

Operator	Assoziativität	Bedeutung
->	links	Dereferenzierungsoperator (Tag 19)

++ --	nicht	Inkrement und Dekrement
**	rechts	Exponent
! ~ \ + -	rechts	logisches NICHT, bitweises NICHT, Referenz (Tag 19), unäres +, unäres -
=~ !~	links	Mustervergleich (Tag 9)
* / % x	links	Multiplikation, Division, Modulo, String-Wiederholung
+ - .	links	Addition, Subtraktion, Stringverkettung
<< >>	links	Bitverschiebung nach links, nach rechts
monadische Operatoren	nicht	funktionsähnliche (benannte monadische) Operatoren
< > <= >= lt gt lt le	nicht	relationale Operatoren, Vergleiche
== != <=> eq ne cmp	nicht	Gleichheit (<=> and cmp, Tag 8)
&	links	bitweises UND
| ^	links	bitweises ODER, bitweises XOR
&&	links	logisches UND im C-Stil
||	links	logisches ODER im C-Stil
..	nicht	Bereichsoperator (Tag 4)
?:	rechts	Bedingungsoperator (Tag 6)
= += -= *= /= etc.	rechts	Zuweisungsoperatoren
, =>	links	Kommaoperatoren (Tag 4)
Listen-Operatoren	nicht	Listenoperatoren im Listenkontext (Tag 4)
not	rechts	logisches NICHT
and	links	logisches UND
or xor	links	logisches ODER, EXKLUSIV-ODER

Tabelle 3.2: Operatorvorrang und -assoziativität

Ein Beispiel: Simple Statistik

Und jetzt ein Beispiel namens ***stats.pl***, das Sie um die Eingabe von Zahlen bittet, eine nach der anderen. Wenn Sie mit der Eingabe fertig sind, gibt es aus, wie viele Zahlen es waren, sowie die Summe aller Zahlen und den Durchschnitt. Es ist ein einfaches Statistik-Skript, aber es zeigt Vergleiche und Variablenzuweisungen (und wir werden auf diesem Skript später aufbauen). Hier ein Beispiel, wie es aussieht, wenn es läuft:

```
% stats.pl
Bitte eine Zahl eingeben: 7
Bitte eine Zahl eingeben: 4
Bitte eine Zahl eingeben: 20
Bitte eine Zahl eingeben: 1
Bitte eine Zahl eingeben: 0.5
Bitte eine Zahl eingeben:
Anzahl der eingegebenen Zahlen: 5
Summe der eingegebenen Zahlen: 32.5
Durchschnitt: 6.50
%
```

Listing 3.1 zeigt den Code des Statistikskripts.

Wie beim Temperaturumrechnungsbeispiel von gestern gehe ich einfach davon aus, dass Sie nur Zahlen eingeben. Alles andere, aus Versehen oder Absicht, wird zu Perl-Warnungen und möglichen Fehlern führen. Später in dieser Woche lernen Sie mehr über die Eingabeüberprüfung. Im Moment nehmen wir einfach an, wir hätten nur gute Daten zu verarbeiten.

Listing 3.1: Das Skript stats.pl

```
1: #!/usr/bin/perl -w
2:
3: $input = '';   # Benutzereingaben
4: $count = 0;    # Zaehler
5: $sum = 0;      # Summe
6: $avg = 0;      # Durchschnitt
7:
8: while () {
9:  print 'Bitte eine Zahl eingeben: ';
10:  chomp ($input = <STDIN>);
11:  if ($input ne '') {
12:   $count++;
13:   $sum += $input;
14:   }
15:  else { last; }
16: }
17:
18: $avg = $sum / $count;
19:
20: print "\nAnzahl der eingegebenen Zahlen: $count\n";
21: print "Summe der eingegebenen Zahlen: $sum\n";
22: printf("Durchschnitt: %.2f\n", $avg);
```

Dieses Skript hat drei Hauptabschnitte: einen zum Initialisieren der Variablen, einen zum Einlesen und Speichern der Eingaben und einen zum Berechnen des Durchschnitts und zur Ausgabe der Ergebnisse.

Dies ist der Initialisierungsabschnitt:

```
3: $input = '';   # Benutzereingaben
4: $count = 0;    # Zähler
5: $sum = 0;      # Summe
6: $avg = 0;  # Durchschnitt
```

Vier Skalarvariablen werden verwendet: `$input`, um die Eingaben zu speichern, wie sie hereinkommen, `$count`, um die Anzahl der eingegebenen Zahlen zu verfolgen (wegen englisch ***count***, »zählen«, heißen Zählervariablen häufig so). `$sum` und `$avg` sollen die Summe und den Durchschnitt speichern.

Im nächsten Abschnitt bitten wir um die Eingabe der Daten:

```
8: while () {
9:  print 'Bitte eine Zahl eingeben: ';
10:  chomp ($input = <STDIN>);
11:  if ($input ne '') {
12:   $count++;
13:   $sum += $input;
14:   }
15:  else { last; }
16: }
```

Dieser Teil des Skripts liest mit einer `while`-Schleife und einer `if`-Bedingung immer wieder Eingabedaten ein, und zwar solange, bis wir eine Leerzeile erhalten. Ich werde Schleifen und Bedingungen erst am Tag 6 behandeln, lassen Sie sich davon also nicht aufhalten. Vielleicht haben Sie `while`s und `if`s ja auch schon mal gesehen und erkennen, dass die `while`-Schleife hier keine Abbruchbedingung hat (die Klammern in Zeile 8 sind leer) und dadurch eine unendliche Schleife ist.

Im Körper der `while`-Schleife lesen wir mit Zeile 10 die aktuelle Eingabe ein (und ich weiß, Sie wollen immer noch wissen, was `chomp` und `<STDIN>` eigentlich machen, wir kommen bald dazu). Zeile 11 überprüft, ob die Eingabe etwas enthält. Beachten Sie, dass es sich hier um einen String-, nicht um einen Zahlenvergleich handelt (`ne`).

Wenn Eingabedaten vorhanden sind, inkrementieren wir in Zeile 12 die Variable `$count` und addieren in Zeile 13 den neu eingegebenen Wert zur Variablen `$sum`. So halten wir den Zähler und die Gesamtsumme immer auf dem laufenden. Mit der Berechnung des Durchschnitts warten wir bis zum Ende der Schleife.

Zeile 15 enthält die `else`-Klausel der Bedingung in Zeile 11 - wenn die Eingabe leer war, dann machen wir direkt hier weiter und steigen mit dem Schlüsselwort `last` aus der unendlichen `while` -Schleife aus (Genaueres über `last` erfahren Sie, wenn wir uns am Tag 6 mit Schleifen befassen).

Zu guter Letzt beenden wir alles mit der Berechnung des Durchschnitts und der Ausgabe der Ergebnisse:

```
18: $avg = $sum / $count;
19:
20: print "\nAnzahl der eingegebenen Zahlen: $count\n";
21: print "Summe der eingegebenen Zahlen: $sum\n";
22: printf("Durchschnitt: %.2f\n", $avg);
```

Zeile 18 ist einfach die Berechnung des Durchschnitts. Die Zeilen 20 bis 22 geben den Zähler, die Summe und den Durchschnitt aus. Beachten Sie das `\n` am Anfang der ersten `print` -Anweisung; es fügt eine extra Leerzeile vor der Ergebnisausgabe ein. Sie erinnern sich, dass `\n` überall in einem String stehen kann, nicht nur am Ende.

In der dritten Ergebniszeile verwenden wir wieder `printf` zur Formatierung der Zahlenausgabe. Diesmal haben wir ein `printf`-Format genommen, das den Wert als Fließkommazahl mit 2 Dezimalstellen ausgibt (`%.2f`). Im nächsten Abschnitt erfahren Sie mehr über `printf`.

Ein- und Ausgabe

Wir werden den heutigen Tag mit zwei Themen abschließen, die anfangs nicht zu all dem zu passen scheinen, worüber wir bis jetzt im Zusammenhang mit Skalaren gesprochen haben: den Umgang mit Eingabe und Ausgabe, ***Input*** und ***Output***. Ich komme hier aus einem wesentlichen Grund dazu: Sie sollen wissen, was in Ihren Skripts abläuft, wenn etwas von der Tastatur gelesen und auf den Bildschirm ausgegeben wird.

In diesem Abschnitt reden wir erst einmal über ganz einfache Eingabe und Ausgabe, im Laufe dieses Buches werden Sie mehr darüber erfahren, insbesondere auch über den Zugriff auf Dateien. Tag 15 schließlich ist ganz diesem Thema gewidmet.

Datei-Handles und Standardein- und -ausgabe

Zuerst etwas Terminologie: In den Skripts von heute und gestern haben Sie mit Ihrem Perl-Code Eingabedaten von der Tastatur eingelesen und Ausgaben an den Bildschirm geschickt. ***Tastatur*** und ***Bildschirm*** sind aber nicht unbedingt die besten Formulierungen, denn im Grunde genommen lesen Sie von einer Quelle namens ***Standardeingabegerät*** (oft auch nur kurz ***Standard Input*** oder Standardeingabe genannt) und schreiben zu einem Ziel namens ***Standardausgabegerät*** (***Standard Output***, Standardausgabe). Diese beiden Konzepte sind von Unix-Systemen entliehen. Wenn Sie mit Unix vertraut sind, wird Ihnen die Verwendung von ***Pipes*** und ***Filtern*** und ***Redirection*** geläufig sein. Kennen Sie nur Windows oder Mac, ergibt der Gedanke an ein Standardeingabe- oder -ausgabegerät für Sie wahrscheinlich nicht viel Sinn.

In jedem Fall aber arbeiten Sie, wenn Sie Daten von einer Quelle lesen oder auf ein Ziel schreiben, mit sogenannten ***Datei-Handles*** (***File-Handles***). Datei-Handles beziehen sich meist auf Dateien auf einem Speichergerät (Festplatte, Diskette etc.), es kommt aber auch vor, dass Daten von einer namenlosen Quelle kommen oder dort hingehen, zum Beispiel von oder zu einem anderen Programm wie einem Webserver. Zur allgemeinen Handhabung von Datenquellen und -zielen, die keine Dateien sind, bietet Ihnen Perl vordefinierte Datei-Handles für die Standardeingabe und die Standardausgabe an: `STDIN` und `STDOUT` (es gibt auch `STDERR` für Standard Error, aber das heben wir für später auf). Mit diesen beiden Datei-Handles kann man Daten von der Tastatur holen (`STDIN` ist der normale Eingabekanal) und an den Bildschirm ausgeben (`STDOUT` ist der normale Ausgabekanal).

Mit <STDIN> eine Zeile von der Standardeingabe lesen

In unseren bisherigen Skripts stand meistens eine Anweisung zum Einlesen der Tastatureingaben, die etwa so

aussah:

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

So etwas wird Ihnen in Perl-Code häufiger begegnen, auch wenn es oft in mehreren Zeilen steht, etwa so (beide Formen sind äquivalent):

```
$eingabezeile = <STDIN>
chomp($eingabezeile)
```

Sie wissen, dass `$eingabezeile` eine Skalarvariable ist und dass Sie ihr etwas zuweisen. Aber was?

Der `STDIN` -Teil dieser Zeile ist der vordefinierte Datei-Handle für die Standardeingabe. Sie brauchen diesen speziellen Datei-Handle weder besonders zu öffnen noch irgendwie zu organisieren: er steht Ihnen einfach zur Verfügung. Warum er komplett großgeschrieben ist? Das ist eine Perl-Konvention, damit man Datei-Handles nicht so leicht mit anderen Sachen durcheinanderbringt (wie zum Beispiel Schlüsselwörtern der Sprache).

Die spitzen Klammern um `STDIN` sind nötig, um Eingaben aus einem Datei-Handle zu lesen. Die <>-Zeichen werden auch oft Eingabeoperator genannt (wegen ihrer Form manchmal auch Diamantoperator). `<STDIN>` bedeutet also: »***Lies die Eingaben aus dem Datei-Handle*** `STDIN`.« In diesem besonderen Fall, wo Sie den `<STDIN>`-Ausdruck einer Variablen zuweisen, liest Perl eine Zeile von der Standardeingabe und stoppt, sobald es zu einem Zeilenvorschub (***Newline***) oder auf dem Macintosh zu einem Wagenrücklauf (***Carriage Return***) kommt. Anders als in C können Sie die Daten nicht explizit durchlaufen und jedes einzelne Zeichen daraufhin betrachten, ob es ein Zeilenende ist - Perl erledigt das für Sie. Alles, was Sie brauchen, ist `<STDIN>` und eine Skalarvariable, in der Sie die Eingabezeile dann speichern.

Was für `<STDIN>` eigentlich eine Zeile ist, wird von Perls ***Input Record Separator*** *festgelegt, dem Trennsymbol für Eingabedatensätze. Voreingestellt ist der Zeilenvorschub. Am Tag 9 erfahren Sie, wie man dieses Trennzeichen ändert. Bis dahin nehmen Sie einfach an, dass das Zeilenendezeichen auch tatsächlich das Ende einer Zeile ist.*

Dieses ganze Gerede über Ein- und Ausgabe führt uns endlich auch zu der Funktion `chomp` (vom englischen ***chomp,*** »nagen«). Wenn Sie eine Eingabezeile mit `<STDIN>` einlesen und in einer Variablen speichern, erhalten Sie alles, was eingetippt wurde, inklusive dem Zeilenvorschub am Ende. Den wollen Sie normalerweise aber dort gar nicht haben, es sei denn, Sie geben die Eingabe gleich wieder aus und brauchen ihn zur Formatierung. Die Perl-Funktion `chomp` nimmt einen String, und wenn das letzte Zeichen ein Zeilenvorschub ist, entfernt es diesen. Beachten Sie, dass `chomp` den Originalstring direkt verändert (anders als Stringverkettung und andere Stringfunktionen, die komplett neue Strings erstellen und die ursprünglichen Strings unverändert lassen). Deshalb können Sie `chomp` für sich allein in einer eigenen Zeile aufrufen, ohne der Variablen, die den String enthält, den Wert neu zuweisen zu müssen:

```
chomp($eingabezeile)
```

Frühere Perl-Versionen hatten eine ähnliche Funktion für die gleiche Aufgabe, genannt `chop`. Wenn Sie älteren Perl-Code lesen, werden Sie auf viel `chop` stoßen. Der Unterschied zwischen `chomp` und `chop` ist, dass `chop` das letzte Zeichen im String entfernt, ohne zu unterscheiden, ob es ein Zeilenvorschub ist oder nicht. `chomp` hingegen ist sicherer: Es schneidet nichts ab außer einen Zeilenvorschub.

Mit *print* auf die Standardausgabe schreiben

Sobald Sie Eingaben in Ihr Perl-Skript geholt haben, von `<STDIN>`, aus einer Datei oder von wo auch immer, können Sie mit diesen Eingaben machen, was Sie wollen. Irgendwann kommt dabei der Augenblick, wo Sie auch wieder Daten ausgeben möchten. Die zwei gebräuchlichsten Wege hierfür kennen Sie schon: `print` und `printf`.

Fangen wir mit `print` an. Die Funktion `print` kann beliebig viele Argumente übernehmen und schickt sie an die Standardausgabe (üblicherweise den Bildschirm). Bis jetzt haben wir immer nur ein Argument verwendet, aber Sie können auch mehrere Argumente übergeben, getrennt durch Kommata. Mehrere `print`- Argumente werden verkettet, bevor sie ausgegeben werden:

```
print 'Guten Morgen!';
print 1, 2, 3;    # gibt '123' aus
$a = 4;
print 1, ' ', $a;  # gibt "1 4" aus
print 1, " $a";    # gibt auch "1 4" aus
```

Genaugenommen sind die `print`-Argumente eine Liste, und es gibt einen Weg, zwischen Listenelementen besondere Zeichen ausgeben zu lassen. Sie werden morgen, am Tag 4, mehr darüber lernen.

Ich habe vorhin schon den Datei-Handle `<STDOUT>` als Möglichkeit aufgeführt, das Standardausgabegerät anzusteuern. Sie haben aber vielleicht bemerkt, dass wir Daten an den Bildschirm immer nur mit `print` ausgegeben haben, ohne uns jemals auf `<STDOUT>` beziehen zu müssen. Perl will Ihnen Zeit und Tipparbeit ersparen und geht deswegen davon aus, dass Sie die Standardausgabe benutzen wollen, wenn Sie `print` ohne einen expliziten Datei-Handle verwenden. Die beiden folgenden Anweisungen bewirken exakt das gleiche:

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

Am Tag 15, wenn wir über Datei-Handles sprechen, die mit echten Dateien verbunden sind, erfahren Sie mehr über die lange Version von `print`.

printf und *sprintf*

Zusätzlich zum alten Arbeitspferd `print` bietet Perl auch die Funktionen `printf` und `sprintf`. Diese beiden Funktionen sind äußerst nützlich für spezielle Formatierungen und Ausgaben von Zahlen. Sie arbeiten fast genauso wie die gleichnamigen Funktionen in C, aber seien Sie vorsichtig: `printf` ist bei weitem nicht so effizient wie `print`. Nehmen Sie also nicht einfach an, Sie können `printf` überall verwenden, weil Sie es so gewohnt sind. Verwenden Sie `printf` nur, wenn Sie dafür einen besonderen Grund haben.

Wie Sie seit gestern wissen, verwendet man die `printf`-Funktion, um formatierte Zahlen und Strings auszugeben, zum Beispiel an die Standardausgabe. `sprintf` formatiert einen String und gibt dann einfach nur den neuen String zurück, ist also geeigneter für Verschachtelungen innerhalb von Ausdrücken (tatsächlich ruft sogar `printf` für die Formatierungsarbeit `sprintf` auf).

An beide Funktionen, `printf` und `sprintf`, übergibt man zwei oder mehr Argumente: Das erste Argument ist ein String mit Formatierungscodes, das zweite und alle weiteren sind die Werte, auf die diese Codes angewendet werden sollen. Wir hatten vorhin ein Beispiel von `printf`, das eine Fließkommazahl auf zwei Dezimalstellen abgeschnitten hat:

```
printf("Durchschnitt: %.2f", $avg);
```

Wir haben auch schon gesehen, wie man auf den nächsten ganzzahligen Wert abrundet:

```
printf("%d Grad Celsius\n", $cel);
```

Gestern haben Sie zudem gesehen, wie man mit `sprintf` eine Fließkommazahl auf zwei Nachkommastellen beschränkt:

```
$wert = sprintf("%.2f", $wert);
```

Die Formatierungscodes folgen denselben Regeln wie die C-Versionen (obwohl der Längenspezifikator `*` nicht

unterstützt wird) und können recht komplex werden. Ein simpler Formatierungscode, den Sie in Perl verwenden könnten, hat folgende Syntax:

```
%l.px
```

Wobei `x` für einen Code steht, der auf den Wertetyp verweist. Am interessantesten sind für Sie wahrscheinlich der Formatierungscode `d` zur Ausgabe von Integern (ganzzahligen Werten) und der Code `f` für die Ausgabe von Fließkommazahlen. Das `l` und das `p` im Formatierungscode sind beide optional. `l` bezieht sich auf die Anzahl der Zeichen, die der Wert im Ergebnisstring aufnehmen soll (aufgefüllt mit Leerzeichen, wenn der Wert des Ausdrucks weniger als `l` Stelle hat), und `p` auf die Anzahl der Nachkommastellen in einer Fließkommazahl. Alle Zahlen werden auf das entsprechende Format gebracht.

Hier einige typische Beispiele, wie `sprintf` oder `printf` eingesetzt werden könnte:

```
$wert = 5.4349434;
printf("->%5d\n", $wert);          # ->   5
printf("->%11.5f\n", $wert);       # ->  5.43494
printf("%d\n", $wert);             # 5
printf("%.3f\n", $wert);           # 5.435
printf("%.1f\n", $wert);           # 5.4
```

Stehen mehrere Formatierungscodes hintereinander, werden sie von links nach rechts ausgewertet, wobei jeder Formatierungscode durch ein Argument ersetzt wird (daher sollten Sie gleichviel Formatierungscodes wie Extraargumente haben):

```
printf("Startwert: %.2f Endwert: %.2f\n", $start, $end);
```

Wenn in diesem Beispiel `$start` gleich `1.343` und `$end` gleich `5.33333` ist, wird die Anweisung folgendes ausgeben:

```
Startwert: 1.34 Endwert: 5.33
```

Wenn Sie mit den `printf`-Formatierungscodes aus C nicht vertraut sind, finden Sie mehr darüber in der `perlfunc`-Manpage (oder der ***printf***-Manpage).

Eine Anmerkung zum Gebrauch von Funktionen

Jetzt, da Sie die Funktionen `print` und `chomp` kennengelernt und einen kleinen Blick auf andere Funktionen wie `oct` und `int` und `sprintf` geworfen haben, ist ein guter Zeitpunkt, zu erklären, wie Funktionsaufrufe funktionieren. Wenn Sie genau aufgepaßt haben, haben Sie vielleicht bemerkt, dass ich die `print`-Funktion folgendermaßen aufrufe:

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

Ich habe aber `chomp` und `printf` folgendermaßen aufgerufen:

```
chomp($input = <STDIN>);
printf("Durchschnitt: %.2f", $avg);
```

Einmal stehen die Funktionsargumente in Klammern, das andere Mal nicht. Was ist nun richtig? Die Antwort ist: ***beides***. Klammern um die Funktionsargumente sind optional - solange Perl versteht, was Ihre Argumente sind, kommt es mit beiden Formen wunderbar zurecht.

Ich verwende in diesem Buch beide Varianten. Meine generelle Regel ist, dass ich keine Klammern setze, wenn eine Funktion ein einziges Argument entgegennimmt - `int` oder `oct` sind hierfür gute Beispiele. Wenn mehrere Argumente übergeben werden oder das Argument ein Ausdruck ist (wie häufig bei `printf` oder `chomp` der Fall), setze ich Klammern.

Je nachdem, womit Sie sich vertraut fühlen - und was Ihnen richtiger scheint - schaffen Sie sich ruhig Ihre eigene Regel für das Einklammern von Funktionsargumenten. Ich sollte allerdings erwähnen, dass die Regel »Klammern

sind optional, solange Perl versteht, was Ihre Argumente sind« gelegentlich schwierig zu begreifen ist, besonders bei mehreren Argumenten und noch mehr, wenn einige davon eingeklammerte Ausdrücke oder Listen sind. Und als wäre das noch nicht komplex genug: Einige von Perls eingebauten Funktionen sind eigentlich gar keine Funktionen, sondern Operatoren und haben deswegen einen anderen Vorrang als echte Funktionsaufrufe (mehr darüber im nachfolgenden Vertiefungsabschnitt). Es gibt genaue Rangfolgeregeln, wie komplexe Funktionsaufrufe ausgewertet werden. Doch die einfachste und sicherste Lösung, wenn Perl Ihre Argumente zu ignorieren scheint oder unerwartete Ergebnisse produziert, sind Klammern um alle Argumente. Die Aktivierung von Perl-Warnungen kann ebenfalls beim Abfangen einiger dieser Probleme hilfreich sein.

Vertiefung

Noch nicht genug von Strings und Zahlen? Sie wollen mehr? Kein Problem. Dieser Abschnitt behandelt ein paar andere bemerkenswerte Eigenheiten im Zusammenhang mit skalaren Daten, bevor wir mit Listen weitermachen.

Alle Operatoren werden in der ***perlop-****Manpage* besprochen, die Funktionen in der ***perlfunc-****Manpage*. Wie ich bereits erwähnt habe: Sie kommen zu all diesen Seiten mit dem Befehl `perldoc` oder über die Website `http://www.perl.com/CPAN-local/doc/ manual/html/pod`.

Nützliche Zahlen- und String-Funktionen

Perl enthält eine recht kleine Anzahl an Funktionen für eine Vielzahl von Zwecken. Diese Funktionen sind in Anhang A zusammengefaßt; die ***perlfunc-****Manpage* beschreibt sie noch detaillierter. Insbesondere bietet Perl nützliche Funktionen für Zahlen und Strings, inklusive der in Tabelle 3.3 aufgelisteten. Wir werden einige davon in den folgenden Kapiteln noch genauer untersuchen, andere können Sie selbst erforschen.

Funktion	Argument(e)	Was sie liefert
`abs`	WERT	Absolutwert von WERT
`atan2`	Y, X	Arkus-Tangens von Y/X
`chr`	ZAHL	Das Zeichen, das im ASCII-Zeichensatz für ZAHL festgelegt ist
`cos`	AUSDR	Cosinus von AUSDR in Bogenmaß
`exp`	AUSDR	e hoch AUSDR. Für allgemeine Potenzierung können Sie ** nehmen
`int`	AUSDR	Gibt den Integerteil von AUSDR zurück
`index`	STRING, TEILSTRING [, POSITION]	Gibt die Position zurück, an der TEILSTRING zuerst im STRING auftaucht. Wenn Sie die POSITION festlegen, beginnt dort die Suche
`lc`	AUSDR	Wandelt AUSDR in Kleinbuchstaben um
`lcfirst`	AUSDR	Wandelt das erste Zeichen in AUSDR in Kleinbuchstaben um
`length`	AUSDR	Länge von AUSDR in Bytes
`log`	AUSDR	Logarithmus von AUSDR (natürlicher Logarithmus zur Basis ***e***)
`ord`	AUSDR	ASCII-Wert des ersten Zeichens von AUSDR
`rand`	[AUSDR]	Zufallszahl zwischen 0 und AUSDR. Ohne AUSDR: zwischen 0 und 1
`reverse`	LISTE	Gibt in skalarem Kontext alle Zeichen der Liste »rückwärts« zurück
`rindex`	STRING, TEILSTRING [, POSITION]	Wie `index`, nur dass `rindex` von hinten anfängt: gibt die Position zurück, an der TEILSTRING zuletzt im STRING auftaucht
`sin`	AUSDRUCK	Sinus von AUSDR (in Bogenmaß)
`sqrt`	AUSDR	Quadratwurzel von AUSDR
`substr`	AUSDR, STARTPOS [, LAENGE]	Gibt einen Teilstring von AUSDR zurück, und zwar ab dem STARTPOS-ten Zeichen in AUSDR. Wenn Sie die LAENGE des Teilstrings nicht festlegen, erhalten Sie alle Zeichen ab STARTPOS

`uc`	AUSDR	Wandelt die Zeichen in AUSDR in Großbuchstaben um
`ucfirst`	AUSDR	Wandelt das erste Zeichen in AUSDR in einen Großbuchstaben um

Tabelle 3.3: Zahlen- und String-Funktionen

Bitweise Operatoren

Perl bietet den üblichen Satz C-ähnlicher Operatoren zum Spielen mit Bits in Integer- Zahlen: `~`, `<<`, `>>`, `&`. `|` und `^` . Auch für diese Operatoren gibt es Zuweisungsabkürzungen. Genaueres finden Sie in der ***perlop-****Manpage*.

Die Operatoren *cmp* und < = >

Zusätzlich zu den Vergleichsoperatoren, die ich im Abschnitt über Vergleiche beschrieben haben, verfügt Perl noch über die Operatoren `<=>` und `cmp`. Ersterer ist für Zahlen, der zweite für Strings. Beide geben `-1`, `0` oder `1` zurück, je nachdem, ob der linke Operator größer ist als der rechte, die Operatoren gleich sind oder der rechte Operator größer ist als der linke. Diese Operatoren verwendet man meistens zum Sortieren, worüber Sie am Tag 8 mehr erfahren werden.

Funktionen und funktionsähnliche Operatoren

Die in Perl eingebauten Funktionen bestehen aus zwei Gruppen: Funktionen und Operatoren, an die ein Argument übergeben wird und die nur wie Funktionen aussehen. Die funktionsähnlichen Operatoren stehen in der Mitte der Auswertungsrangfolge und verhalten sich in dieser Hinsicht wie Operatoren (während Funktionsaufrufe mit Klammern immer den höchsten Vorrang haben). Sehen Sie in der ***perlop-****Manpage* unter »Named Unary Operators« nach der Liste dieser Funktionen.

Zusammenfassung

Heute haben wir uns weiter durch die skalaren Daten gearbeitet. Sie haben weitere Operatortabellen gesehen und die Operatoren für die Zuweisung an Variablen, zur Änderung der Werte von Variablen und für die Verkettung und Wiederholung von Strings kennengelernt. Sie haben auch etwas über die Rangfolge von Operatoren gelernt, die bestimmt, welcher Operator zuerst abgearbeitet wird, wenn Sie mehrere Operatoren in einem Ausdruck stehen haben.

Am Ende der heutigen Lektion haben wir die Ein- und Ausgabe behandelt, insbesondere die Verwendung von `<STDIN>`, um Daten in ein Perl-Skript hineinzubekommen, und die verschiedenen ***print***-Funktionen, um sie wieder auszugeben. Und Sie haben bereits etwas über den Aufruf von Funktionen mit und ohne Klammern um die Argumente erfahren.

Folgende Perl-Funktionen wurden heute besprochen:

- `print` nimmt eine Liste von kommagetrennten Werten und Strings entgegen und gibt diese Werte an das Standardausgabegerät `<STDOUT>` aus.
- `printf` nimmt einen Formatierungsstring und beliebig viele Werte entgegen und gibt diese Werte entsprechend den Formatierungsanweisungen aus.
- `sprintf` macht dasselbe wie `printf`, außer dass es den formatierten String zurückgibt, ohne irgend etwas auszugeben.
- `chomp` mit einem String-Argument schneidet alle Zeilenvorschubzeichen von diesem String ab und gibt die Anzahl der entfernten Zeichen zurück.
- `chop` ist die ältere Version von `chomp`; es entfernt das letzte Zeichen vom String und gibt das gelöschte Zeichen zurück.

Schlagen Sie in der ***perlfunc-****Manpage* (siehe auch Anhang A) nach, um mehr über diese Funktionen zu erfahren.

Fragen und Antworten

Frage:
Ich möchte einen String Buchstabe für Buchstabe durchgehen und zählen, wie oft der Buchstabe »t« vorkommt. Wie kann ich das in Perl bewerkstelligen?

Antwort:
Nun ja, Sie könnten sich mit einer `for`-Schleife und einer Stringfunktion Buchstabe für Buchstabe durch den String arbeiten, aber das wäre wohl ein ziemlicher Overkill (und würde Sie als C-Programmierer outen, der bei Strings noch an nullterminierte Arrays denkt). Strings sind eine der besonderen Stärken von Perl, das über eingebaute Mechanismen zur Suche in Strings verfügt. Darum brauchen Sie diese entsetzlichen Buchstabe-für-Buchstabe-Vergleiche gar nicht. Am Tag 9 werden Sie mehr über diese Suchmechanismen, das sogenannte Pattern Matching, lernen. Bis dahin verschwenden Sie nicht soviel Zeit mit dem Iterieren über Strings - es geht wirklich viel einfacher.

Frage:
Kann ich `printf` genauso verwenden wie in C?

Antwort:
Ja, das können Sie, aber Sie sollten sich wirklich angewöhnen, statt dessen mit `print` zu arbeiten. Die `print`-Funktion ist effizienter und hilft beim Ausgleichen von Abrundungsfehlern in Fließkommazahlen. Es ist (ganz ehrlich) in den allermeisten Fällen die bessere Idee, `print` zu verwenden und auf `printf` nur in bestimmten Fällen (beispielsweise zum Runden) zurückzukommen. Mit `printf` können Sie alle Formatierungsanweisungen aus der Sprache C verwenden, außer `*`.

Frage:
Warum ist es wichtig, dass manche Funktionen Funktionen und manche eigentlich Operatoren sind? Benehmen sich nicht alle gleich?

Antwort:
Nein. Funktionen und Operatoren verhalten sich leicht unterschiedlich. Funktionen haben zum Beispiel einen höheren Rang. Außerdem können Argumente an einen Operator auf Vorrangsbasis gruppiert werden (was zu unerwarteten Ergebnissen führen kann). In den meisten Fällen aber sollte der Unterschied zwischen einer Funktion und einem Operator Ihnen nicht den Schlaf rauben.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist der Unterschied zwischen dem Postfix-Inkrementoperator (`$x++`) und dem Präfix-Inkrementoperator (`++$x`)?
2. Was legt der Operatorvorrang fest? Was besagt die Assoziativität?
3. Was ist ein Datei-Handle? Wofür braucht man Datei-Handles?
4. Definieren Sie Standardeingabe und -ausgabe. Wofür werden sie verwendet?
5. Was macht die Funktion `chomp`?
6. Was sind die Unterschiede zwischen `print`, `printf` und `sprintf`? Wann verwenden Sie welche Funktion?
7. Was bewirken die folgenden Operatoren?

```
.
**
ne
||
*=
```

Übungen

1. Schreiben Sie ein Programm, das beliebig viele Zeilen und jede Art von Eingabe, die durch Drücken der Eingabetaste beendet wird, entgegennimmt (also ähnlich arbeitet wie das Programm ***stats.pl***). Geben Sie

die Anzahl der eingegebenen Zeilen zurück.

2. FEHLERSUCHE: Was ist an dem folgenden Code-Fragment falsch?

```
    while () {
 print 'Einen Namen eingeben: ';
 chomp ($input = <INPUT>);
 if ($input ne '') {
   $namen++;
}
 else { last; }}
```

3. Schreiben Sie ein Programm, das mehrere Wörter aus verschiedenen Zeilen einliest und diese Wörter zu einem Gesamtstring zusammenfügt.
4. Schreiben Sie ein Programm, das einen String annimmt und ihn dann auf dem Bildschirm zentriert (gehen Sie von einer Zeile mit 80 Zeichen und einem String mit weniger als 80 Zeichen aus). Tipp: Die Funktion `length` liefert Ihnen die Länge eines Strings in Zeichen.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Präfixoperatoren inkrementieren den Wert einer Variablen vor seiner Verwendung, Postfixoperatoren danach.
2. Der Vorrang eines Operators bestimmt, welche Teile eines Ausdrucks zuerst ausgewertet werden, wenn dieser andere Ausdrücke enthält. Assoziativät bestimmt die Reihenfolge, in der Operatoren von gleichem Rang auswertet werden.
3. Ein Datei-Handle wird zum Einlesen bzw. Ausgeben von Daten von einer Quelle bzw. an ein Ziel verwendet, sei es eine Datei, die Tastatur, der Bildschirm oder ein anderes Gerät. Datei-Handles sind für Perl der normale Weg, mit Eingabe und Ausgabe und mit all diesen Geräten umzugehen.
4. Standardeingabe und Standardausgabe sind generell Eingabequellen und Ausgabeziele (das heißt nicht nur aus oder in Dateien). Sie werden meist verwendet, um Benutzereingaben von der Tastatur zu lesen oder etwas auf den Bildschirm zu schreiben.
5. Die Funktion `chomp` entfernt das Zeilenvorschubzeichen vom Ende eines Strings. Wenn dort kein Zeilenvorschub steht, macht `chomp` gar nichts.
6. Die `print`-Funktion ist der übliche Weg, etwas auf den Bildschirm oder andere Ziele auszugeben. Die Funktion `printf` gibt formatierte Strings auf bestimmte Ziele aus; `sprintf` formatiert einen String und liefert diesen formatierten String als Wert zurück anstatt ihn auszugeben.
7. Die Antworten sind:
1. `.` verkettet Strings
1. `**` erzeugt Exponentialzahlen
1. `ne` »ungleich« für Strings
1. `||` logisches ODER (C-Stil)
1. `*=` Multiplizieren und Zuweisen, dasselbe wie `$x = $x * $y`

Lösungen zu den Übungen

1. Hier eine mögliche Antwort:

```
    #!/usr/bin/perl -w
$input = ''; # Eingabe
$zeilen = 0; # Zeilenzaehler
while () {
 print 'Geben Sie etwas Text ein: ';
 chomp ($input = <STDIN>);
 if ($input ne '') {
   $zeilen++;
  }
 else { last; }
}
print "Zahl der eingegebenen Zeilen: $zeilen\n";
```

2. Das Datei-Handle für die Standardeingabe ist `STDIN`, nicht `INPUT`.
3. Hier eine mögliche Antwort:

```
    #!/usr/bin/perl -w
$input = ''; # Benutzereingaben
$satz = '';  # Ergebnis-Satz;
while () {
 print 'Geben Sie ein Wort ein: ';
 chomp ($input = <STDIN>);
 if ($input ne '') {
   $satz .= $input . ' ';
  }
 else { last; }
}
print "ergibt den Satz: $satz\n";
```

4. Hier eine mögliche Antwort:

```
    #!/usr/bin/perl -w
$input = "";  # Benutzereingaben
$leer = 0;    # Leerstellen drum herum
$laenge = 0 ; # Länge des Strings
print 'Geben Sie etwas Text ein: ';
chomp ($input = <STDIN>);
$laenge = length $input;
$leer = int((80 - $laenge) / 2);
print ' ' x $leer;
print $input;
print ' ' x $leer . "\n";
print '*' x 80;
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Mit Listen und Arrays arbeiten

Am Tag 2 und 3 haben wir uns in erster Linie mit Skalaren, also einzeln vorliegenden Daten, befaßt. Heute reden wir über Gruppen von Objekten, nämlich Listen und Arrays, und was Sie damit anstellen können. Sie lernen heute unter anderem,

- was Arrays und Listen sind, und in was für Variablen man sie speichert,
- wie man Arrays erstellt und gebraucht,
- was Listen- und Skalarkontext (für Sie und Perl) bedeutet,
- mehr über `<STDIN>` und wie Sie Daten in Listen einfügen und
- wie Sie Listen ausgeben.

Listendaten und -variablen

Wenn man skalare Daten als ***einzelne Objekte*** definiert, kann man sich Listendaten als eine ***Sammlung*** mehrerer Objekte - genauer gesagt als Satz von Skalaren - vorstellen. Genau wie der Begriff ***Skalar*** sowohl Zahlen als auch Strings umfaßt, bezieht sich der Begriff ***Liste*** üblicherweise auf zwei spezielle Konstrukte: Arrays und Hashes. Wir werden heute über Arrays reden und morgen tiefer in die Hashes einsteigen.

Ein Array ist eine Sammlung von beliebig vielen Skalaren. Auf jedes einzelne Element kann zugegriffen werden, indem man sich auf die Nummer seiner Position im Array (seinen Index) bezieht. Array-Indizes beginnen in Perl bei 0. Zur Illustration zeigt Abbildung 4.1 ein einfaches Array mit ein paar Zahlen und Strings darin.

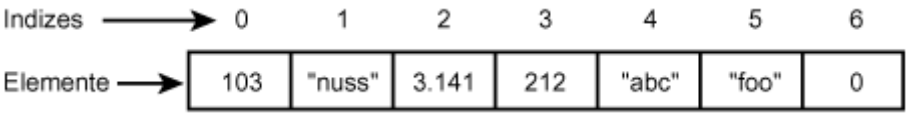

Abbildung 4.1: Anatomie eines Arrays

Arrays sind ***geordnet***. Das bedeutet, sie haben ein erstes Element, ein letztes, und alle Elemente dazwischen stehen in einer bestimmten Reihenfolge. Man kann die Reihenfolge der Elemente durch Sortierung ändern oder die Elemente durchlaufen, eins nach dem andern, vom Anfang bis zum Ende.

Arrays werden in Arrayvariablen gespeichert, wie Skalare in Skalarvariablen gespeichert werden. Eine Arrayvariable beginnt mit dem ***at***-Zeichen `@`. Abgesehen von diesem ersten Zeichen gelten für Namen von Arrayvariablen die gleichen Regeln wie für Skalarvariablen, nämlich die allgemeinen Variablennamensregeln:

- Variablennamen können insgesamt bis zu 256 Zeichen lang sein und Zahlen, Buchstaben und Unterstriche enthalten.
- Namen sind ***case-sensitiv***. Das heißt, es wird zwischen Groß- und Kleinbuchstaben unterschieden.
- Arrayvariablen können, anders als Skalarvariablen, mit einer Zahl anfangen, dürfen dann aber auch weiterhin nur noch andere Zahlen enthalten.

Die Namen von Skalar- und Arrayvariablen können nicht miteinander in Konflikt geraten. Die Skalarvariable `$x` ist eine andere Variable als die Arrayvariable `@x`.

Listendaten haben auch eine Rohform, die man (Überraschung!) eine Liste nennt. Eine Liste ist einfach eine Folge von Elementen. Sie können eine Liste einer Variablen zuweisen, eine Liste durchlaufen, um jedes ihrer Elemente auszugeben, sie in andere Listen verschachteln oder als Funktionsargument benutzen. Normalerweise werden Sie Listen zum Erstellen von Arrays und Hashes verwenden oder um Daten von einer Liste an eine oder mehrere

andere zu übergeben. Arrays sind einfach eine bestimmte Form von Liste, aber im wesentlichen das gleiche, ja austauschbar.

Definition und Gebrauch von Listen und Arrays

Was macht man mit einem Perl-Array? Man legt es an (***definiert*** es), weist es einer Arrayvariablen zu, packt Elemente hinein und liest oder nimmt sie wieder heraus, oder man zählt, wie lang das Array ist, das heißt wie viele Elemente es hat. Man kann noch eine ganze Menge mehr mit Arrays und ihren Daten anstellen, aber fangen wir mit den Grundlagen an.

Zur Definition von Perl-Arrays gibt es gar nicht viel zu sagen. Im Unterschied zu anderen Sprachen, in denen Arrays sorgfältig eingerichtet werden müssen, bevor man sie verwenden kann, erscheinen Arrays in Perl wie von Geisterhand dort, wo Sie sie brauchen, und sie wachsen und schrumpfen jederzeit und ganz von allein auf die richtige (also der aktuellen Anzahl der Elemente entsprechenden) Größe. In Perl- Arrays können Sie außerdem nicht nur jede Art von skalaren Daten speichern - Zahlen, alphanumerische Zeichen oder eine Mischung aus beiden -, sondern auch beliebig viele. Wie viele Elemente Sie in ein Array packen, bestimmen allein Sie und der Ihnen verfügbare Speicher.

Arrays können auch Referenzen enthalten, aber über die haben Sie noch nichts gelernt. Haben Sie Geduld, am Tag 19 wird alles klar.

Listen erstellen

Wie ich bereits erwähnt habe, bezieht sich der Begriff ***Liste*** auf einen unspezialisierten Satz von Daten, ein ***Array*** kann als eine Liste betrachtet werden, die in einer Arrayvariablen gespeichert ist. Sie können Listen überall verwenden, wo Arrays erwartet werden und umgekehrt.

Eine Liste erstellen Sie mit der Listensyntax: Tippen Sie die Elemente ein, setzen Sie Kommata dazwischen und alles zusammen in Klammern. Sie haben dann eine rohe Liste, mit der Sie ein Array oder einen Hash erstellen können, je nachdem, was für einer Variablen sie die Liste zuweisen. Betrachten wir jetzt ein paar Array-Beispiele. Hashes behandeln wir morgen.

Im folgenden Beispiel erstellen wir ein Array namens `@zahlen`, das vier Elemente hat:

```
@zahlen = (1, 2, 3, 4);
```

Listen aus Strings sind genauso einfach:

```
@strings = ('Petersilie', 'Salbei', 'Rosmarin', 'Thymian');
```

Die Listensyntax kann jede beliebige Mixtur aus Strings, Zahlen, Skalarvariablen, Ausdrücken, die Skalare liefern, und anderen Elementen enthalten:

```
@kram = ('garbonzo', 3.145, $zaehler, 'Schokolade', 4 / 7, $a++);
```

Für eine leere Liste schreiben Sie einfach nichts zwischen die Klammern (nicht einmal ein Leerzeichen):

```
@nichts = ();
```

Sie können Listen ineinander verschachteln, aber diese Unterlisten werden im endgültigen Array nicht festgehalten. Alle Elemente werden in ein einziges Array aufgedröselt und leere Unterlisten dabei entfernt:

```
@kombination = (1, 4, (6, 7, 8), (), (5, 8, 3));
# ergibt (1, 4, 6, 7, 8, 5, 8, 3);
```

Ähnlich verhält sich Perl beim Verschachteln von Arrayvariablen. Alle Elemente in den Unterarrays werden zu einer

einzigen größeren Liste verkettet:

```
@kombination2= (@zahlen, @strings);
# ergibt (1, 2, 3, 4, 'Petersilie', 'Salbei', 'Rosmarin', 'Thymian')
@zahlen = (@zahlen, 5);     # ergibt  (1, 2, 3, 4, 5);
```

Sagen wir, Sie definieren ein Array aus Ein-Wort-Strings, zum Beispiel Wochentagen oder eine Namensliste. Ein sehr gebräuchlicher Perl-Trick zum Erstellen von Ein-Wort- Arrays ist der Einsatz der speziellen `qw`-Syntax (von »***quote word«, »zitiere Wort«***). Die `qw`-Syntax erspart Ihnen das Eintippen der Anführungszeichen und Kommata und macht das Array um einiges lesbarer:

```
@monate = qw(
   Januar Februar Maerz
   April Mai Juni
   Juli August September
   Oktober November Dezember
);
```

Listen mit dem Bereichsoperator erstellen

Sie brauchen eine Liste aller Zahlen zwischen 1 und 1000? Hierfür alle Zahlen einzeln einzutippen, wäre schändliche Zeitverschwendung. Irgendeine Schleife, die eine Zahl nach der anderen hinzufügt, wäre da schon einfacher, aber immer noch ziemlich ***kludgy.*** (Ein ***kludge*** ist alter Freak-Jargon für eine nicht so optimale Lösung eines Problems. Einige irregeführte Einzelpersonen würden argumentieren, dass Perl selbst ein ***kludge*** ist. Doch das ignorieren wir mit einem Lächeln.)

In welcher Situation auch immer Sie denken: »Es muss einen einfachen Weg geben, das hinzukriegen«, stehen die Chancen wirklich gut, dass es einen solchen einfachen Weg in Perl auch gibt. In diesem Fall ist es der Bereichsoperator (`..`), auch ***Range***- Operator genannt. Um ein Array mit den Zahlen von 1 bis 1000 zu erstellen, schreiben Sie einfach:

```
@vieleZahlen = (1 .. 1000);
```

Das war's, Sie sind fertig. Das Array `@vieleZahlen` enthält jetzt 1000 Elemente, von 1 bis 1000. Der ***Range***-Operator arbeitet so, dass er vom linken Operanden jeweils um 1 bis zum rechten Operanden hochzählt (runterzählen kann man damit nicht).

Der Bereichsoperator funktioniert auch mit Buchstaben:

```
@alphabet = ('a' .. 'z');   # enthält 26 Elemente
```

Wir werden am Tag 6 noch einmal auf den Bereichsoperator zurückkommen, wenn wir uns mit Iterationen befassen.

Zuweisung und Listen

Bis jetzt haben wir die Listensyntax auf der rechten Seite des Zuweisungsausdrucks verwendet, um eine Liste zu erstellen und sie einer Arrayvariablen zuzuweisen. Sie können auf die linke Seite einer Zuweisung aber auch eine Liste von Variablenreferenzen stellen. Perl wird diesen Variablen dann auf Grundlage der Liste rechts Werte zuweisen.

Nehmen Sie zum Beispiel folgenden Ausdruck:

```
($a, $b) = (1, 2);
```

Auf beiden Seiten der Zuweisung haben wir hier eine Liste. Doch nach wie vor ist links die Seite, auf der Variablen gespeichert werden. Perl weist die Werte der rechten Liste den Variablen in der linken Liste zu, und zwar in der Reihenfolge, in der sie auftreten. So erhält `$a` den Wert `1` und `$b` den Wert `2`. Das ist natürlich genau das gleiche, als würden Sie die Werte in getrennte Zeilen schreiben, aber auf diese Art können Sie bequemer mehreren Variablen Werte zuweisen. Beachten Sie, dass die Zuweisungen parallel ablaufen, so dass man auch schreiben kann:

```
($x, $y) = ($y, $x);
```

Dieses Beispiel vertauscht die Werte von `$x` und `$y`. Nichts passiert vor dem anderen, deswegen wird es wie gewünscht funktionieren.

Wenn auf beiden Seiten eines Zuweisungsoperators Listen stehen, gilt die Regel, dass jede Variable auf der linken Seite einen Wert aus der rechten Seiten erhält. Wenn es links mehr Variablen als rechts Werte gibt, dann erhalten die übrigen Variablen den undefinierten Wert. Gibt es mehr Werte als Variablen, werden die übrigbleibenden Werte ignoriert.

Sie können auf die rechte Seite auch Arrayvariablen stellen. Das Array wird dann in seine Elemente zerlegt, und die Werte werden wie im Falle einer normalen Listensyntax zugewiesen:

```
($a, $b) = @zahlen;
```

Sie können Arrays sogar auf beiden Seiten in Listen verschachteln - sie werden in ihre Elemente aufgedröselt und nach den obigen Regeln zugewiesen - mit einer Ausnahme:

```
($a, @mehr) = (10, 11, 12, 13, 14);
```

In diesem Beispiel erhält `$a` den Wert `10` und `@mehr` erhält die Liste (`11, 12, 13, 14`). Arrayvariablen auf der linken Seite einer Listenzuweisung sind ***gefräßig*** - das heißt, sie speichern alle noch folgenden Elemente der Liste auf der rechten Seite der Zuweisung. Das ist wichtig, wenn Sie zum Beispiel folgendes Beispiel betrachten:

```
($a, @mehr, $b) = (10, 11, 12, 13, 14);
```

In diesem Fall erhält `$a` den Wert `10`, @mehr die Liste (`11, 12, 13, 14`) und `$b` den undefinierten Wert. Weil das Array alle verbleibenden Werte auf der rechten Seite »auffrißt«, bleibt keiner mehr übrig, der `$b` zugewiesen werden könnte.

Auf Array-Elemente zugreifen

Nun haben Sie also ein Array, vielleicht mit ein paar in Listenschreibweise vorgegebenen Elementen, oder einfach ein leeres Array, das Sie später mit Werten (beispielsweise Benutzereingaben) füllen möchten.

Um auf ein Array-Element zuzugreifen, verwenden Sie die `[]`-Syntax, die Sie vielleicht aus anderen Sprachen kennen: Der Index des Elements, auf das Sie zugreifen wollen, folgt in eckigen Klammern auf den Arraynamen:

```
$zahlen[4];
```

Dieses Beispiel liefert Ihnen den Wert des fünften Elements im Array `@zahlen`. (Sie erinnern sich, der Index startet bei 0.) Das folgende Beispiel ändert den Wert des ersten Elements in `10`:

```
$zahlen[0] = 10;
```

Halten Sie kurz an, und betrachten Sie diese Syntax etwas genauer. Kommt sie Ihnen komisch vor? Sie sollte. Wir beziehen uns hier auf das fünfte Element in einem Array namens `@zahlen`, aber wir benutzen etwas, das wie eine Skalarvariable aussieht. Das ist kein Druckfehler - so arbeitet die Syntax. Sie verwenden `@arrayname`, um sich auf das ganze Array zu beziehen, und `$arrayname[index]`, um auf ein einzelnes Element innerhalb dieses Arrays zuzugreifen.

Das bedeutet nicht, dass Sie `$arrayname` nicht verwenden können, um auf einen normalen Skalar zu verweisen. `$arrayname`, `$arrayname[index]` und `@arrayname` sind ganz verschiedene Dinge. Am besten merkt man sich das, indem man sich daran erinnert, dass die Elemente von Arrays ja Skalare sind und man deswegen eine Skalarvariable braucht, um auf sie zuzugreifen, auch wenn sie innerhalb eines Arrays stehen (die Perl-Warnungen geben Ihnen Bescheid, falls Sie das vergessen).

Array-Indizes beginnen bei 0 wie in anderen Sprachen auch, und ein Index kann nur eine ganze Zahl (ein Integer) sein. So bezieht sich `$arrayname[0]` auf das erste Element in einem Array, `$arrayname[1]` auf das zweite und so

weiter. Sie müssen nicht explizit eine Zahl als Index verwenden, Sie können auch eine Variable oder jeden anderen Ausdruck dafür nehmen, der zu einer Zahl ausgewertet wird:

```
$array[$zaehler];    # das Element an Position $zaehler;
```

Was passiert, wenn Sie versuchen, auf ein Element zuzugreifen, das gar nicht existiert - zum Beispiel auf Position 5 in einem dreielementigen Array? Wenn Sie die Warnungen aktiviert haben, erhalten Sie eine Fehlermeldung. Anderenfalls erhalten Sie einen undefinierten Wert (`0` oder `""`, je nach Kontext).

Wachsende Arrays

Wenn Sie hinter dem letzten Element eines Arrays ein weiteres Element anfügen, wird Perl das Array entsprechend vergrößern, das Element hinzufügen und eventuell dazwischenliegenden Elementen den undefinierten Wert zuweisen. Zum Beispiel:

```
@t = (1, 2, 3);
$t[5] = 10;
```

Das Array `@t` enthält nach diesen beiden Zeilen die Elemente `(1, 2, 3, undefiniert, undefiniert, 10)`. Wenn Sie versuchen, auf die undefinierten Werte in der Mitte zuzugreifen, wird Perl Sie warnen (vorausgesetzt, Sie haben die Warnungen angeschaltet).

Möchten Sie sichergehen, dass Sie es nur mit definierten Werten zu tun haben, können Sie Ihre Werte mit der Funktion `defined` überprüfen:

```
if (!defined $array[$index]) {
   print "Element $index ist undefiniert.\n";
}
```

Alternativ dazu können Sie ein Array-Element »leeren«, indem Sie es mit der Funktion `undef` auf den undefinierten Wert setzen:

```
if (defined $array[$index]) {
   undef($array[$index]);
}
```

Beachten Sie, dass das Array-Element, das Sie mit `undef` undefiniert gemacht haben, sich nach wie vor im Array befindet, nur dass es jetzt den undefinierten Wert hat. Um wirklich ein Element aus einem Array zu entfernen, müssen Sie es explizit löschen, indem Sie das Array neu konstruieren.

Die `undef`-Funktion kann übrigens überall eingesetzt werden, nicht nur in Arrays, um einer beliebigen Variablen den undefinierten Wert zuzuweisen. Man kann sie auch ohne jegliches Argument verwenden, dann gibt sie einfach den undefinierten Wert zurück, zum Beispiel:

```
@loch_in_der_mitte = (1, undef, undef, undef, 5);
```

Weil man auf diese Art so leicht einen undefinierten Wert erhält, bezeichnet man ihn üblicherweise einfach mit `undef` (und für den Rest des Buches werde ich genau das tun).

Das Ende eines Arrays finden

Weil Arrays in Perl jede beliebige Größe haben können, brauchen Sie eine Methode, das Ende eines Arrays zu finden. Schließlich muss zum Beispiel eine Schleife, die die Inhalte des Arrays durchläuft, wissen, wann Ende ist. Bequemerweise hat Perl hierfür eine eigene Syntax, die Ihnen die Indexnummer des letzten Array-Elements liefert: `$#array`. Mit dem Index des letzten Elements könnten Sie eine einfache Schleife bauen, die vom Index 0 bis zum letzten läuft. So könnten Sie zum Beispiel mit einem Konstrukt wie dem folgenden alle Elemente eines Arrays zeilenweise ausgeben:

```
$i = 0;
while ($i <= $#array) {
```

```
   print $array[$i++], "\n";
}
```

Für Operationen dieser Art verwendet man üblicherweise eine `for`- oder `foreach`-Schleife statt einer `while`-Schleife. Aber da Sie bis jetzt nur `while`-Schleifen gesehen haben, wollte ich hier keine andere nehmen. Wir werden uns `foreach` später in dieser Lektion noch ansehen.

Wenn Sie den Wert von `$#array` verändern, dann vergrößern bzw. verkleinern Sie das Array. Setzen Sie `$#array` auf einen größeren als den aktuellen Wert, erhalten alle neu hinzugekommenen Elemente den undefinierten Wert. Wenn Sie den Wert von `$#array` verkleinern, werden alle Elemente am Ende des Arrays verworfen.

Beachten Sie, dass man mit der Syntax `$#array` nicht die Länge des Arrays (oder die Anzahl der enthaltenen Werte) herausfindet. Weil Array-Indizes bei 0 beginnen, liefert Ihnen `$#array` einen Wert, der ***um eins kleiner ist als die Anzahl von Elementen im Array***. Wie man die Länge eines Arrays ermittelt, ist Thema des nächsten Abschnitts.

Die Länge eines Arrays herausfinden

Um die Anzahl der Elemente in einem Array zu ermitteln, verwenden Sie folgende Anweisung:

```
$anzahl_elemente = @array;
```

Denken Sie darüber jetzt nicht lange nach, merken Sie sich einfach: Um die Anzahl der Elemente in einem Array zu erhalten, verwendet man eine Skalarvariable und weist ihr eine Arrayvariable zu. Ich werde später erklären, warum das funktioniert (im Abschnitt »Listen- und Skalarkontexte«).

Listen und Arrays sortieren

In anderen Sprachen müssen Sie, wenn Sie die Inhalte eines Arrays sortieren möchten, vielleicht eine eigene Sortierroutine schreiben. Nicht so in Perl, da ist schon eine eingebaut. Zum Sortieren eines Arrays brauchen Sie lediglich die Funktion `sort`:

```
@geordnete_zahlen = sort @zahlen;
```

Diese Zuweisung sortiert das Array `@zahlen` und weist die neue Liste dann der Variablen `@geordnete_zahlen` zu. Das @zahlen-Array bleibt unverändert, also unsortiert.

Diese besonders simple `sort`-Variante sortiert die Inhalte von `@zahlen` in ASCII- Reihenfolge - das heißt, dass `5543` im Array vor `94` steht (weil 5 vor 9 kommt). Um das Array in numerischer Reihenfolge zu sortieren, setzen Sie nach `sort` einen speziellen Vergleichsausdruck:

```
@geordnete_zahlen = sort { $a <=> $b } @zahlen;
```

Der Teil in den geschweiften Klammern legt fest, wie `sort` das Array mit Hilfe des Vergleichsoperators <=> sortiert. Wir werden an Tag 8 zum Anpassen von `sort`- Routinen kommen. Jetzt können Sie sich erst einmal folgendes merken: Um ein Array nach Zahlen zu sortieren, verwenden Sie `sort` mit einem Vergleich. Um ein Array alphabetisch zu sortieren, nehmen Sie die Kurzform ohne den Vergleich.

Alle Elemente eines Arrays durchlaufen

Vorhin habe ich Ihnen in einem Beispiel eine `while`-Schleife gezeigt, die alle Elemente eines Arrays durchläuft, vom ersten Element zum letzten. Viel gebräuchlicher für eine solche ***Iteration über ein Array***, wie ein solcher Durchlauf auch genannt wird, ist allerdings die Verwendung einer `foreach`-Schleife wie hier:

```
foreach $x (@liste) {
   # mache etwas mit dem jeweiligen Element
```

```
}
```

Die `foreach`-Schleife wird für jedes Element der Liste innerhalb der Klammern einmal ausgeführt (hier ist es das Array `@liste`, aber es könnte auch eine rohe Liste, ein Bereich oder alles andere sein, was Ihnen als Ergebnis eine Liste liefert (zum Beispiel die `sort`-Funktion.) Bei jedem Listenelement wird der Wert dieses Elements einer Skalarvariablen (hier `$x`) zugewiesen und der Code zwischen den geschweiften Klammern ausgeführt. Sie könnten zum Beispiel eine `foreach`-Schleife verwenden, um jedes Listenelement auszugeben, sie alle zusammenzufügen oder sie auf den Wert `undef` zu überprüfen.

Mehr über `foreach` erfahren Sie am Tag 6 und morgen, wenn wir über Hashes reden.

Ein Beispiel: Mehr Statistik

Erinnern Sie sich an das simple Statistikskript von gestern? Man sollte nacheinander Zahlen eingeben, das Skript berechnete dann Anzahl, Summe und Durchschnitt. Lassen Sie uns dieses Skript heute etwas verändern, so dass es die eingegebenen Zahlen in ein Array speichert. Haben wir nämlich die Zahlen auch nach der Eingabe noch parat, können wir mehr mit ihnen anstellen - sie zum Beispiel sortieren oder den Median herausfinden (ein Zahlenwert, der die Zahlenliste so halbiert, dass die Hälfte der Zahlen kleiner und die andere Hälfte größer als dieser Wert ist).

Die neue Version des Skripts könnte zum Beispiel so ablaufen:

```
% mehrstats.pl
Bitte eine Zahl eingeben: 4
Bitte eine Zahl eingeben: 5
Bitte eine Zahl eingeben: 3
```

(Und so weiter, aus Platzgründen hier nicht aufgeführt.)

```
Bitte eine Zahl eingeben: 47
Bitte eine Zahl eingeben: 548
Bitte eine Zahl eingeben: 54
Bitte eine Zahl eingeben: 5485
Bitte eine Zahl eingeben:
Anzahl der eingegebenen Zahlen: 49
Summe der eingegebenen Zahlen: 10430
Hoechste Zahl: 5485
Niedrigste Zahl: 2
Durchschnitt: 212.86
Median: 45
%
```

Schon auf den ersten Blick erkennt man, worin sich dieses Statistikskript von der gestrigen Version unterscheidet:

- Es errechnet die höchste und die niedrigste der eingegebenen Zahlen.
- Es ermittelt den Median (der Wert in der Mitte einer sortierten Liste aller Zahlen).

Im Code finden wir einen weiteren bedeutenden Unterschied zwischen dieser und der letzten Version: Wir speichern die Eingabedaten diesmal in ein Array, anstatt sie einfach zu verwerfen. (Der Eingabevorgang hat sich nicht verändert: Wenn Sie das Skript ausführen, beenden Sie die Dateneingabe nach wie vor mit einer leeren Zeile). Listing 4.1 zeigt den Code für das neue Skript.

Listing 4.1: Das Skript mehrstats.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $input = '';  # Benutzereingabe
4:  @nums = ();   # Zahlen-Array
5:  $count = 0;   # Zaehler
6:  $sum = 0;     # Summe
7:  $avg = 0;     # Durchschnitt
8:  $med = 0;     # Median
9:
```

```
10: while () {
11:   print 'Bitte eine Zahl eingeben: ';
12:   chomp ($input = <STDIN>);
13:   if ($input ne '') {
14:     $nums[$count] = $input;
15:     $count++;
16:     $sum += $input;
17:    }
18:   else { last; }
19: }
20:
21: @nums = sort { $a <=> $b } @nums;
22: $avg = $sum / $count;
23: $med = $nums[$count /2];
24:
25: print "\nAnzahl der eingegebenen Zahlen: $count\n";
26: print "Summe der eingegebenen Zahlen: $sum\n";
27: print "Hoechste Zahl: $nums[$#nums]\n";
28: print "Niedrigste Zahl: $nums[0]\n";
29: printf("Durchschnitt: %.2f\n", $avg);
30: print "Median: $med\n";
```

Diese Version des Statistikskripts besteht aus vier Abschnitten: Initialisierung, Dateneingabe, Sortierung und Statistikberechnung und schließlich Ausgabe der Ergebnisse.

Der Initialisierungsabschnitt, Zeile 3 bis 7, ist derselbe wie im vorherigen Skript, außer dass wir zwei Variablen hinzugefügt haben: eine Arrayvariable (`@nums`) in Zeile 4 zum Speichern der numerischen Eingaben und die `$med`-Variable in Zeile 8 für den Medianwert. Wie bei anderen Variablen bräuchten wir `@nums` eigentlich nicht initialisieren, aber es sieht nett aus und verschafft uns am Anfang des Skripts einen Überblick über alle verwendeten Variablen.

Die Zeilen 10 bis 19 sind die neue `while`-Schleife für die Dateneingabe. Wenn Sie diese Version mit der von gestern vergleichen, werden Sie bemerken, dass hier eigentlich nicht viel Neues steht. Wir lesen immer noch eine Zahl pro Zeile ein, inkrementieren den Zähler `$count` und aktualisieren mit jeder Zahl die Variable `$sum` (und wir überprüfen auch immer noch nicht, ob die Eingaben wirklich nur Zahlen enthalten). Der einzige Unterschied ist die Zeile 14, in der wir bei jedem Schleifendurchlauf die neue Zahl dem Array `@nums` hinzufügen. Die `$count`-Variable erfüllt hier einen doppelten Zweck: Sie ist nicht nur Zähler der Eingaben, sondern auch Array-Index (beachten Sie, dass wir das Array korrekt bei `0` starten, weil wir `$count` erst danach, in Zeile 15, erhöhen.

Sind alle Eingaben gemacht, geht es weiter zu Zeile 21, wo wir das `@nums`-Array mit der numerischen `sort`-Routine sortieren, die ich vorhin beschrieben habe. Beachten Sie, dass wir die Variablen `$a` oder `$b` nicht deklarieren brauchen - diese Variablen sind lokale Variablen von `sort` und fallen unter den Tisch, sobald die Sortierung vollständig ist. Zeile 22 berechnet den Durchschnitt wie in unserer vorigen Version des Skripts, und Zeile 23 errechnet den Median, also den Wert in der Mitte einer sortierten Liste. Dafür müssen wir lediglich den Wert von `$count` durch zwei teilen und das Ergebnis dann als Array-Index angeben (im Falle einer ungeraden Anzahl von Elementen ist das Ergebnis natürlich keine ganze Zahl, aber dann schneidet Perl die Nachkommastellen einfach ab).

Damit kommen wir zur Auswertung in den Zeilen 25 bis 30. Zähler, Summe und Durchschnitt sind dieselben wie vorher, aber jetzt haben wir auch Höchstwert, kleinsten und Zentralwert hinzugefügt. Maximum und Minimum sind einfach: Da unser Array ja sortiert ist, brauchen wir nicht einmal etwas berechnen, wir nehmen nur das erste und das letzte Element des Arrays. Und da wir den Median schon seit Zeile 23 kennen, müssen wir ihn nur noch ausgeben wie alle anderen Werte.

Listen- und skalarer Kontext

Bevor wir die Arrays hinter uns lassen, möchte ich eine kleine Pause einlegen und das Thema Kontext behandeln. ***Kontext*** bezeichnet in Perl den Effekt, dass das Verhalten von Daten davon abhängt, was Sie mit ihnen zu machen versuchen. Viele an und für sich identische Daten »benehmen« sich je nach Kontext ganz unterschiedlich. Kontext kann sehr verwirrend sein, und wenn Sie damit durcheinandergeraten, können Sie Ergebnisse erhalten, die absolut sinnlos zu sein scheinen (oder Teile von Skripten anderer Programmierer produzieren scheinbar aus dem Nichts Resultate). Wenn Sie jedoch erst einmal verstanden haben, was Kontext in Perl bedeutet, und wissen, wie verschiedene Operationen damit arbeiten, können Sie auch komplexe Aufgaben sehr schnell erledigen, die in

anderen Sprachen viele Zeilen Code erfordern würden.

Wenn Sie ein erfahrener Programmierer sind und das Buch gerade nach Wichtigem durchsuchen, halten Sie genau hier an. Nehmen Sie sich die Zeit, diesen Abschnitt aufmerksam zu lesen und sicherzugehen, dass Sie es verstanden haben. Kontext kann sowohl Anfänger als auch erfahrene Perl-Programmierer in den Wahnsinn treiben.

Was ist ein Kontext?

Was also bedeutet ***Kontext***? Lassen Sie uns eine Analogie aus dem Deutschen verwenden: Nehmen Sie das Wort ***meinen***. Definieren Sie es, in 25 Wörtern oder weniger.

Was ist die Definition des Worts ***meinen***? Die richtige Antwort ist eigentlich: »Kommt drauf an, wie man es verwendet.« In dem Satz »Sie meinen nicht mich, sondern meinen Bruder« ist ***meinen*** einmal ein Verb (»Sie meinen«) und einmal ein Pronomen (»meinen Bruder«). »Sie meinen meinen Bruder« - dasselbe Wort, dieselbe Schreibweise, aber ganz verschiedene Bedeutungen, abhängig vom Zusammenhang - vom Kontext.

»Okay«, werden Sie sagen, »aber was hat das mit Perl zu tun?« Nehmen Sie den simplen Perl-Ausdruck `5 + 4`. Wozu wird dieser Ausdruck ausgewertet?

»Na, `9`«, sagen Sie und fragen sich, ob da ein Haken ist. Darauf können Sie wetten, es gibt einen: Was ist, wenn Sie diesen Ausdruck in eine Bedingung stellen, zum Beispiel so:

```
if (5 + 4) { ... }
```

Wozu wird `5 + 4` jetzt ausgewertet? Gut, zuerst immer noch zu `9`. Aber weil wir es hier als Bedingung verwenden, wird es auch als ***wahr*** ausgewertet (Sie erinnern sich, nur `0` und `""` sind ***falsch***). Im Kontext einer Bedingung - im Perl-Jargon Boolescher Kontext genannt - ist das Ergebnis von `5 + 4` ein Boolescher Wert, keine Zahl. Die `if`-Konstruktion erwartet als Ergebnis wahr oder falsch; Perl erkennt das und liefert wie (vom `if`, vielleicht gar nicht von Ihnen) gewünscht.

Auch die automatische Konvertierung zwischen Zahlen und Strings basiert auf Kontext. Bei Berechnungen erwartet Perl numerische Operanden (einen numerischen Kontext), deswegen konvertiert es alle Strings zu Zahlen. Stringoperatoren verlangen Stringkontext, und deshalb wandelt Perl numerische Werte in Strings um.

Numerischer, String- und Boolescher Kontext sind Formen des allgemeinen skalaren Kontexts. Um die Unterscheidung dieser drei brauchen Sie sich üblicherweise keine Sorgen zu machen, da Perl sie normalerweise erkennt und für Sie berücksichtigt.

Komplizierter wird es bei der Unterscheidung von Skalaren und Listen. Jede Operation in Perl wird entweder in skalarem oder in Listenkontext ausgewertet. Manche Perl-Operationen verlangen eindeutig skalaren, andere Listenkontext. Die Verwendung des falschen Datentyps würde bei diesen Operationen zu Fehlermeldungen führen. Doch eine dritte Sorte von Operationen kann sowohl in skalarem als auch in Listenkontext ausgewertet werden und sich in einem Kontext anders verhalten als in dem anderen. Und, um die Sache noch komplizierter zu machen, es gibt keine Standardregeln, wie Listen sich in skalarem Kontext verhalten und umgekehrt. Jede Operation hat ihre eigenen Regeln, und die müssen Sie sich merken oder nachsehen. Sobald Sie also mit einer Mischung aus Listen und Skalaren zu tun haben, sollten Sie sich für jede Operation in Perl drei Fragen stellen:

- In welchem Kontext bin ich (skalarer oder Listenkontext)?
- Welchen Datentyp verwende ich in diesem Kontext?
- Was passiert, wenn ich diesen Datentyp in diesem Kontext verwende?

Ich zeige Ihnen jetzt einige wenige Beispiele dieser »dritten Art«, in der Kontext so wichtig ist. Doch das Thema ist mit dieser Lektion nicht abgehakt, sondern fordert weiteren Fleiß Ihrerseits. Der Kontext spielt in Perl und in allen folgenden Lektionen so gut wie immer eine Rolle. Behalten Sie diese drei Fragen im Kopf und die Perl-Dokumentation griffbereit, dann ist es gar nicht so schwierig, wie es im Moment vielleicht scheint.

Noch einmal: die Anzahl der Elemente in einem Array herausfinden

Ein Beispiel haben Sie bereits gesehen, in dem der Kontext entscheidend war. Erinnern Sie sich, wie man die Länge eines Arrays herausfindet?

```
$anzahl_elemente = @array;
```

Ich habe Ihnen dazu gesagt, Sie sollen sich die Syntax einfach merken und sich keine Sorgen darum machen. Jetzt betrachten wir die Syntax genauer, denn es handelt sich um ein klassisches Beispiel, wie verwirrend Kontext in Perl sein kann. Die Variable `$anzahl_elemente` ist skalar, so dass die Zuweisung an diese Variable in skalarem Kontext ausgewertet wird - das heißt, auf der rechten Seite des =-Operators wird ein skalarer Wert erwartet (das ist die Antwort auf Ihre erste Frage: In welchem Kontext befinde ich mich? - In skalarem Kontext).

Auf der rechten Seite haben Sie eine Liste (das ist die Antwort auf die zweite Frage: Was für einen Datentyp benutze ich?). Also haben wir hier eine Liste in skalarem Kontext. Jetzt bleibt nur noch die Frage: »Was passiert mit einer Liste in diesem Kontext?« In diesem Fall wird die Anzahl von Elementen im Array `@array` der Variablen `$anzahl_elemente` zugewiesen. Wird eine Arrayvariable in skalarem Zuweisungskontext ausgewertet, liefert sie die Anzahl der Listenelemente.

Betrachten Sie diesen Vorgang nicht als Konvertierung der Liste in einen Skalar, denn das ist es nicht. Es findet keine Konvertierung statt. Die Liste benimmt sich in diesem Kontext einfach anders als sonst.

Überall, wo ein skalarer Wert erwartet wird und Sie eine Arrayvariable einfügen, erhalten Sie die Anzahl der Elemente dieses Arrays - zum Beispiel in jedem der folgenden Ausdrücke:

```
$med = $array[@array / 2];    # Median, erinnern Sie sich?
$total = @array1 + @array2 + @array3;
if (@array > 10) { ... }
while (@array) { ... }
```

Wenn Ihnen das verwirrend oder zu kryptisch aussieht, können Sie natürlich immer die simple Skalarzuweisung auf die eine und die Arrayvariable auf die andere Seite stellen. Oder wenn Sie die Länge des Arrays nur interessiert, weil Sie es mit einer Schleife durchlaufen möchten, können Sie die Frage nach der Anzahl der Elemente auch komplett umgehen und statt dessen `$#array` als Stoppwert verwenden.

Kontext und Zuweisung

Zuweisungen sind der häufigste Fall, in dem der Kontext wichtig wird - oder zumindest ist Kontext hier am einfachsten zu erklären. Der Kontext auf der linken und der rechten Seite des Zuweisungsoperators kann passen (Skalar = Skalar, Liste = Liste), oder eben nicht (Skalar = Liste, Liste = Skalar). In diesem Abschnitt werde ich ein paar der einfachen Regeln durchgehen, wie Zuweisungen mit Kontext umgehen.

Lassen Sie uns mit den einfachen Fällen anfangen, in denen der Kontext paßt. Die haben Sie bereits alle kennengelernt. Skalar an Skalar bedeutet, ein einzelnes Objekt an ein einzelnes Objekt zuzuweisen, wobei Strings und Zahlen gegebenenfalls konvertiert werden:

```
$x = 'foo';   # Skalarvariable, skalarer Wert
```

Auch Listen-an-Listen-Zuweisungen kennen Sie schon, sowohl mit Arrayvariablen auf der linken und Listensyntax auf der rechten Seite des Zuweisungsoperators als auch mit verschachtelten Listen:

```
@zahlen = (10, 20, 30);
($x, $y, $z) = @zahlen;
($a, @zahlen2) = @zahlen;
```

Jetzt wollen wir uns die Fälle ansehen, in denen der Kontext nicht paßt. Sagen wir, Sie versuchen, einen Skalar in einem Listenkontext zuzuweisen, wie in diesen Beispielen:

```
@array = 3.145;
($a, $b) = $c;
```

Das ist einfach! In diesen Fällen wird der skalare Wert rechts einfach in eine Liste umgewandelt und dann unter Verwendung der Listenzuweisungsregeln zugewiesen. Danach ist `@array` eine Liste mit einem einzigen Wert, `3.145`, `$a` hat den Wert von `$c`, und `$b` ist undefiniert.

Der vertrackteste Fall ist die Zuweisung einer Liste auf der rechten Seite an einen Skalar auf der linken. Sie haben gelernt, was passiert, wenn Sie eine Arrayvariable einer Skalarvariablen zuweisen:

```
$anzahl_elemente = @array;
```

Das funktioniert mit jedem Array:

```
$anzahl_elemente = sort @array;
```

Doch es funktioniert ***nur*** bei Arrays. Für ganz normale Listensyntax, also durch Kommas aufgeteilte Listen, ist die Regel anders:

```
$x = (2, 4, 8, 16);
```

In diesem Fall werden - wie beim Komma-Operator in C - alle Werte in der Liste, außer dem ***letzten*** ignoriert. Der Wert von `$x` wird hier `16`. (Das ist etwas anderes als die Zuweisung derselben Liste an (`$x`) - eine Listen-an-Listen-Zuweisung würde beim ***ersten*** Element, `2`, beginnen und unbenutzte Elemente verwerfen.)

Das Wichtigste, was Sie sich merken sollten, ist, dass es keine allgemeine Regel gibt, wie eine Liste sich in skalarem Kontext verhält - Sie müssen die einzelnen Regeln kennen oder greifbar haben. Behalten Sie die drei Fragen im Kopf; dann dürften Sie keine Schwierigkeiten haben.

Andere Kontexte

Es gibt noch ein paar andere Kontext-Situationen, die einer Betrachtung wert sind - und die Sie beim Arbeiten mit Listen und Skalaren beachten sollten.

Zuerst ist da der Boolesche Kontext, in dem eine Variable auf ihren Wahrheitswert überprüft wird (wie zum Beispiel im Bedingungsausdruck von `if` oder `while`). Sie haben schon gelernt, dass ein skalarer Wert in Booleschem Kontext ***wahr*** ist, wenn er irgendeinen Wert außer `""`, `0` oder `undef` hat.

Für Listen in Booleschem Kontext gilt eine ähnliche Regel - eine Liste mit irgendwelchen Elementen, einschließlich der undefinierten, ist in Booleschem Kontext ***wahr***. Eine leere Liste ist ***falsch***.

Zum zweiten ist der Kontext häufig sehr wichtig bei Funktionen. Viele Funktionen nehmen ihre Argumente als Liste entgegen und werten sie im Listenkontext aus. Wenn Sie Klammern um die Funktionsargumente setzen, wie wir gestern besprochen haben, ist das kein Problem - Sie übergeben der Funktion eine Liste mit Argumenten. Setzen Sie die Klammern aber nicht, versucht Perl, aus den gegebenen Argumenten selbst eine Liste zu bauen. Wenn diese Argumente aber Listen oder eingeklammerte Ausdrücke enthalten, kann Perl durcheinanderkommen. Betrachten wir ein Beispiel mit einer der Funktionen, die als Argument eine Liste erwarten, nämlich `print`:

```
priNT 4 + 5, 6, 'foo';
```

Hier werden die auszugebenden Argumente ausgewertet, als wären sie die Liste `(9, 6, 'foo')`. Im folgenden Fall ist es allerdings anders:

```
print (4 + 5), 6, 'foo';
```

Wegen der Klammern nimmt Perl an, dass der Ausdruck `4 + 5` das einzige Argument ist, und versteht nicht, was die `6` und das `'foo'` am Ende sollen. Aktivierte Perl- Warnungen würden diesen Fehler abfangen (und sich über die Verwendung von Konstanten in leerem Kontext beschweren). In solch einem Fall ist es am besten, die ganze Argumentenliste einzuklammern, damit nichts zweideutig ist:

```
print ((4 + 5), 6, 'foo');
```

Meistens kann Perl erkennen, ob Klammern einen Funktionsaufruf, einen Ausdruck oder eine Liste meinen. In den meisten übrigen Fällen weisen die Warnungen Sie auf die Zweideutigkeiten hin. Aber achten Sie auf diese Unterschiede und den Kontext, wenn Sie Perl-Skripts schreiben.

Die Funktion *scalar*

»Und was ist, wenn ich wirklich mal eine Liste im Skalarkontext verwenden möchte?« fragen Sie sich jetzt vielleicht. Keine Angst, Sie brauchen dafür keine langen Umwege (zum Beispiel eine temporäre Skalarvariable, nur um die Liste in skalaren Kontext zu bekommen). Mit der Funktion `scalar` können Sie eine Liste immer dazu zwingen, in skalarem Kontext ausgewertet zu werden. Nehmen Sie zum Beispiel die beiden folgenden Anweisungen:

```
print @zahlen;
print scalar(@zahlen);
```

Die `print`-Funktion wertet ihre Argumente im Listenkontext aus (deshalb können Sie mehrere durch Kommata getrennte Argumente angeben). Die erste dieser beiden Anweisungen expandiert nun das Array `@zahlen` in Listenkontext und gibt die Werte des Arrays aus. Die zweite wertet `@zahlen` in skalarem Kontext aus und liefert Ihnen in diesem Fall die Anzahl der Elemente von `@zahlen`.

Eingabe, Ausgabe und Listen

Wie gestern betrachten wir zum Abschluß der Lektion einige Eingabe-/Ausgabe- Themen, diesmal mit besonderem Augenmerk auf den Listenkontext:

- `<STDIN>` im Listenkontext
- Ausgabe von Listen

<STDIN> im Listenkontext

Gestern habe ich Ihnen gezeigt, wie man mit `<STDIN>` Daten von der Standardeingabe liest. Bis jetzt haben wir `<STDIN>` wie folgt verwendet:

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

Bei genauem Betrachten erkennen Sie, dass wir `<STDIN>` hier in skalarem Kontext verwenden. Wie viele andere Perl-Operationen verhält sich auch der Eingabeoperator <> in Listenkontext anders als in skalarem.

In Skalarkontext liest `<STDIN>` eine Eingabezeile bis zum Zeilenvorschub. In Listenkontext aber liest `<STDIN>` immer weiter und speichert jede Zeile als ein Element in eine Liste, und zwar so lange, bis es zu einem Dateiende (***end-of-file-***)Zeichen kommt:

```
@liste = <STDIN>;
```

Nun werden Sie fragen, wo Tastatureingaben denn ein Dateiendezeichen haben sollen. Setzen Sie es mit der Tastenkombination `[Strg]-[D]` (bzw. `[Strg]-[Z]` in Windows). Wenn Sie `<STDIN>` im Listenkontext verwenden, wartet Perl bei der Dateneingabe, bis Sie mit `[Strg]-[D]` bzw. `[Strg]-[Z]` sagen: »Hier ist das Dateiende«, speichert Ihre Daten dann in eine Liste und geht weiter im Skript.

Wirklich sinnvoll ist dieses Verhalten von `<STDIN>` in Listenkontext vor allem, wenn Sie aus Dateien lesen, also auch wirklich ein Dateiende haben. Für Tastatureingaben verwendet man normalerweise `<STDIN>` in skalarem Kontext. Aber es ist wichtig, sich auch für die Eingabe die Unterschiede zwischen skalarem und Listenkontext klarzumachen. Sie sind, wie bei so vielem in Perl, erheblich.

Listen ausgeben

In den Beispielen dieses Kapitels haben wir zur Ausgabe von Listen mit Schleifen gearbeitet, die jedes Listenelement betrachtet und dann ausgegeben haben. Wenn Sie die Elemente einer Liste allerdings gar nicht

verändern, sondern wirklich nur ausgeben möchten, gibt es einen leichteren Weg: Die `print`-Funktion geht grundsätzlich davon aus, dass ihre Argumente in Listenkontext stehen, und das können Sie ausnutzen. Nehmen wir zum Beispiel eine simple Liste von 1 bis 9:

```
@liste = (1..9);
```

Wenn Sie diese Liste einfach mit `print @liste` ausgeben, erhalten Sie folgende Ausgabe:

```
123456789
```

Es wird kein Zeilenvorschub ans Ende gesetzt. Die Werte der Liste werden lediglich verkettet, nichts weiter.

Wenn Sie die Elemente nun aber durch Leerzeichen getrennt ausgeben wollen? Sie könnten eine `while`- oder `foreach`-Schleife verwenden. Doch es geht auch einfacher - durch Variableninterpolation. Gestern haben wir die Variableninterpolation für Strings besprochen, durch die in einem String `"dies ist der String $zaehler"` die Variable `$zaehler` durch ihren aktuellen Wert ersetzt wurde. Variableninterpolation funktioniert auch mit Array- oder Hash-Variablen - die Inhalte des Arrays oder Hash werden dabei nacheinander, getrennt durch ein Leerzeichen, ausgegeben. Setzen Sie zum Beispiel `@liste` und einen Zeilenvorschub in Anführungszeichen:

```
print "@liste\n";
```

Sie erhalten die Liste mit Leerzeichen zwischen den Elementen und einem Zeilenvorschub am Ende:

```
1 2 3 4 5 6 7 8 9
```

Auch die Ausgabe von Hash-Variablen funktioniert auf diese Art, nur würde der Hash zuerst in seine Schlüssel und Werte zerlegt. Variableninterpolation von Listenvariablen macht es sehr einfach, die Inhalte einer Liste auszugeben, ohne auf Schleifen zurückgreifen zu müssen. Beachten Sie, dass Sie deswegen in einem ***double-quoted*** String vor ein `@`-Zeichen einen Backslash setzen müssen, damit Perl nicht glaubt, es wäre der Beginn einer Arrayvariablen und nach diesem gar nicht existierenden Array sucht (und sich dann beschwert, dass es keines findet). Perl-Warnungen geben Ihnen Bescheid, wenn Ihnen dieser Fehler unterläuft.

*Eine andere Möglichkeit, die Ausgabe von Listen zu steuern, besteht darin, spezielle globale Perl-Variablen für die Trennsymbole für Ausgabefelder (**Output field separator**) und Ausgabedatensätze (**Output record separator**) zu setzen. Mehr über diese speziellen Variablen erfahren Sie im Vertiefungsabschnitt.*

Vertiefung

Sie haben in dieser Lektion viel gelernt, doch es gibt noch einiges mehr über Arrays und Hashes, das ich noch nicht besprochen habe (wirklich!). Dieser Abschnitt faßt einige Eigenschaften von Listen, Arrays und Hashes zusammen, die ich im Hauptteil dieser Lektion nicht behandelt habe.

Negative Array-Indizes

Im Array-Zugriffsausdruck `$array[index]` ist der (bei 0 beginnende) Index normalerweise die Position des Elements im Array. Sie können aber auch negative Array-Indizes verwenden, wie hier:

```
$array[-1];
```

Negative Array-Indizes zählen vom Ende des Arrays zurück: Der Index `-1` bezieht sich auf die letzte Position im Array (genau wie `$#array`), `-2` auf die vorletzte und so weiter. Sie können mit negativen Indizes genauso auf die Elemente zugreifen wie mit positiven, obwohl es vielleicht aus Lesbarkeitsgründen die bessere Idee wäre, bei den positiven zu bleiben.

Mehr über Bereiche

Wir haben in dieser Lektion mit dem Bereichsoperator `..` Zahlenlisten erstellt. Einige Eigenschaften von Bereichen habe ich Ihnen bis jetzt vorenthalten. So können Sie Bereiche zum Beispiel auch mit Buchstaben, genauer gesagt mit ASCII-Zeichen erzeugen - die Liste, die Sie erhalten, beginnt mit dem ersten Operanden, endet mit dem zweiten und enthält dazwischen alle Zeichen, die in der ASCII-Tabelle zwischen den beiden Operanden liegen. Das Ergebnis des Bereichs `'a .. z'` ist zum Beispiel eine Liste mit 26 Elementen, nämlich den Kleinbuchstaben von a bis z .

Sie können auch Zahlen und Buchstaben oder mehrere Buchstaben kombinieren, der Bereichsoperator liefert Ihnen auf geradezu magische Art und Weise die Werte zwischen den höheren und niedrigeren Werten.

Außerdem kann der Bereichsoperator auch in skalarem Kontext verwendet werden. Dann gibt er einen Booleschen Wert zurück, was in manchen Schleifen oder zur »Simulation« von `awk`- oder `sed`-ähnlichem Verhalten nützlich sein kann. Sehen Sie in der ***perlop***-Manpage unter ***Range-Operator*** nach weiteren Informationen.

chomp und *chop* mit Listen

Die Funktionen zum Entfernen von Zeilenvorschubs- oder anderen Zeichen vom Ende eines Strings, `chomp` und `chop`, akzeptieren auch eine Liste als Argument. In diesem Fall arbeiten sie sich durch alle Listenelemente und entfernen jeweils den Zeilenvorschub oder das letzte Zeichen von jedem Element. Das kann zum Beispiel nützlich sein, wenn Sie alle Zeilenvorschübe aus Daten entfernen möchten, die Sie aus einer Datei gelesen haben.

In der ***perlfunc***-Manpage finden Sie weitere Informationen über `chomp` und `chop`.

Trennsymbole für Ausgabefelder, Datensätze und Listen

In Perl ist ein Satz von globalen Spezialvariablen eingebaut, mit denen man Perls Verhalten in bestimmten Situationen steuern kann. Sie werden im Verlauf dieses Buchs viele dieser Variablen kennenlernen; eine vollständige Liste finden Sie auf der ***perlvar***-Manpage.

Relevant für das heutige Thema sind die Variablen ***Output field separator*** (Trennsymbol für Ausgabefelder), ***Output record separator*** (Trennsymbol für Ausgabedatensätze) und ***List separator*** (Listentrennsymbol). Mit diesen drei globalen Variablen können Sie Perls Standardverhalten bei der Ausgabe von Listen verändern. Tabelle 4.1 erläutert diese Variablen.

Variable	Aufgabe, Voreinstellung
`$,`	***Output field separator,*** Ausgabefeld-Trennsymbol: was zwischen Listenelementen ausgegeben wird. Voreinstellung ist leer, also kein Zeichen.
`$\`	***Output record separator***, Trennsymbol für Ausgabedatensätze: was am Ende einer Liste ausgegeben wird. Voreinstellung ist leer, also kein Zeichen.
`$"`	***List separator***, Listen-Trennsymbol: wie ***Output field separator***, außer dass es nur auf Listenvariablen in interpolierten Strings angewendet wird. Voreinstellung ist ein einzelnes Leerzeichen.

Tabelle 4.1: ***Globale Ausgabevariablen***

Wie Sie im Abschnitt »Listen ausgeben« gelernt haben, verkettet Perl alle Elemente einer Liste, wenn Sie die Liste ausgeben:

```
print (1,2,3);  # gibt "123" aus
```

In Wirklichkeit setzt Perl den Wert des Ausgabefeld-Trennsymbols zwischen die Elemente und den des Ausgabedatensatz-Trennsymbols ans Ende der Liste. Da diese beiden Variablen standardmäßig leer sind, wird zwischen den Elementen und am Ende auch nichts ausgegeben - es sei denn, Sie verändern diese Variablen und legen zum Beispiel folgendes Ausgabeverhalten fest:

```
$, = '*';
$\ = "\n";
print (1,2,3);  # gibt "1*2*3\n" aus
```

Das Listentrennsymbol bestimmt wie das Ausgabefeld-Trennsymbol aussieht, was zwischen Listenelementen ausgegeben wird, gilt aber nur für Listenvariablen in interpolierten Strings. Voreingestelltes Listentrennsymbol ist ein Leerzeichen - deswegen werden bei der Interpolation von Array- und Hash-Variablen standardmäßig Leerzeichen zwischen die Elemente gesetzt, wie Sie im Abschnitt »Listen ausgeben« gesehen haben. Möchten Sie dieses Ausgabeverhalten verändern, setzen Sie einfach das Listentrennsymbol auf einen Wert Ihrer Wahl.

Leerer Kontext

Zusätzlich zu Listen-, skalarem und Booleschem Kontext kennt Perl auch einen ***leeren*** Kontext - das sind die Stellen, an denen Perl überhaupt nichts erwartet. Wenn Sie zum Beispiel einen String als Anweisung in Ihr Skript setzen:

```
"das bringt nichts";
```

dann bringt das wirklich nichts - die einzige Auswirkung ist (bei aktivierten Perl- Warnungen) die Fehlermeldung `useless use of a constant in void context` (sinnlose Verwendung einer Konstante in leerem Kontext). Eine Fehlermeldung zu ***void context*** warnt Sie also, wenn Sie einen Ausdruck an einer Stelle verwenden, an der Perl gar nichts erwartet.

Zusammenfassung

Heute war Listentag. Wie Sie heute gelernt haben, ist eine Liste einfach eine in Klammern gesetzte Reihe von durch Kommata getrennten Skalaren. Weisen Sie eine Liste einer Arrayvariablen `@array` zu, können Sie mit der Array-Zugriffssyntax `@array[index]` auf die einzelnen Elemente dieses Arrays zugreifen. Wir haben des weiteren besprochen, wie man die letzte Position im Array (`$#array`) und die Anzahl der Elemente (`$elements = @array`) herausfindet. Diesen Abschnitt haben wir mit ein paar Bemerkungen zu Listensyntax und Listenzuweisungen abgeschlossen, mit denen Sie Variablen auf beiden Seiten eines Zuweisungsausdrucks parallel Werte zuweisen können.

Dann sind wir das Thema Kontext angegangen, das eine wichtige Rolle spielt, weil Perl viele Dinge im Skalarkontext anders auswertet als im Listenkontext. Sie haben sich mit drei Fragen zum Verstehen eines Kontexts befaßt: Welcher Kontext wird in einem Ausdruck erwartet, welchen Datentyp haben die Daten, und was machen diese Daten in diesem Kontext?

Schließlich haben wir uns weiter mit einfacher Eingabe/Ausgabe, heute in bezug auf Listen, befaßt. Morgen werden wir Ihr Listengrundwissen mit der Behandlung von Hashes vervollständigen. (An Tag 19 werden wir noch fortgeschrittenere Datenstrukturen betrachten und dazu noch einmal auf die Listen zurückkommen - aber mit Listen, Arrays und Hashes im Perl-Repertoire können Sie schon eine Menge anstellen.)

Im folgenden noch einmal die Perl-Funktionen, die Sie heute kennengelernt haben:

- `qw` nimmt eine Liste von durch Leerzeichen getrennten Strings entgegen und gibt eine Liste der einzelnen Stringelementen zurück. `qw` erspart Ihnen das Eintippen vieler Anführungszeichen und Kommata, wenn Sie eine lange Liste von Strings zu definieren haben.
- `defined` nimmt eine Variable oder ein Listenelement entgegen und gibt ***wahr*** zurück, wenn der Wert an dieser Stelle nicht der undefinierte ist.
- `undef` nimmt eine Variable oder ein Listenelement entgegen und setzt sie/es auf den undefinierten Wert. Ohne Argumente gibt `undef` nur den undefinierten Wert zurück.
- `sort` nimmt eine Liste als Argument entgegen, sortiert diese Liste in ASCII- Reihenfolge und gibt die sortierte Liste zurück. Setzt man die Anweisung `{ $a <=> $b }` vor die Liste, sortiert es in numerischer Reihenfolge.
- `scalar` wertet eine Liste in Skalarkontext aus.

Mehr Informationen finden Sie in der ***perlfunc***-Manpage bzw. im Anhang A.

Fragen und Antworten

Frage:
Was ist der Unterschied zwischen dem undefinierten Wert und `undef`?

Antwort:
Der Unterschied ist gering. Der undefinierte Wert ist der Wert, den Perl Variablen, Array-Elementen oder Hash-Werten zuweist, wenn für sie kein Wert vorhanden ist - wenn Sie eine Variable verwenden, ohne sie zu initialisieren, oder wenn Sie die Länge eines Arrays vergrößern, ohne entsprechend Elemente einzufügen. Möchten Sie den undefinierten Wert explizit verwenden, zum Beispiel um eine Variable undefiniert zu machen oder den undefinierten Wert in ein Array aufzunehmen, nehmen Sie dafür die Funktion `undef`*. Wegen der engen Beziehung zwischen dem undefinierten Wert und* `undef` *ist es weit verbreitet, dass* `undef` *für den undefinierten Wert steht (wie in »Die letzten drei Elemente dieses Arrays sind* `undef`*«). Sie werden in Ihrem Code nichts falsch machen, wenn Sie* `undef` *überall dort verwenden, wo Sie einen undefinierten Wert haben möchten.*

Frage:
Ich möchte ein Array von Arrays erstellen.

Antwort:
Das können Sie nicht. Besser gesagt, jetzt noch nicht. Für Arrays von Arrays oder Arrays von Hashes oder jede Art von verschachtelten Datenstrukturen brauchen Sie Referenzen - und etwas Geduld. Wir werden Referenzen an Tag 19 behandeln.

Frage:
Oh je! Ich verstehe Skalar- und Listenkontext nicht. Wenn verschiedene Operationen verschiedene Dinge machen können und es keine Regeln gibt, wie Listen und Skalare sich im jeweils anderen Kontext verhalten, heißt das, dass ich mir für jede Operation merken muss, was sie in jedem Kontext macht?

Antwort:
Äh, ja. Nein. Naja, ungefähr. Wenn Sie die Frage nach dem Kontext absolut abscheulich finden, können Sie die esoterischeren Kontexterscheinungsformen in so gut wie jedem Skript weitestgehend vermeiden, merken Sie sich nur die wichtigsten Besonderheiten (wie man zum Beispiel die Länge eines Arrays herausfindet), und schlagen Sie den Rest nach, wenn irgend etwas nicht richtig funktioniert. Wenn Sie allerdings Skripts anderer Programmierer lesen, müssen Sie vermutlich auf den Kontext achten oder sogar nach verborgenem Kontext suchen.

Frage:
Ich möchte die Anzahl der Elemente in einer Liste herausfinden, also habe ich geschrieben »`length @array...`«

Antwort:
Hören Sie gleich hier auf. Die `length`*-Funktion ist eine nette Funktion - für Strings und Zahlen. Um die Anzahl der Elemente in einer Liste herauszufinden, sollten Sie die Arrayvariable in skalarem Kontext auswerten:*
`$anzahl_elemente = @array.`

Frage:
Wie durchsuche ich ein Array nach einem bestimmten Element?

Antwort:
Eine Möglichkeit wäre, das Array mit einer `foreach-` *oder* `while`*-Schleife zu durchlaufen und nacheinander jedes Element zu überprüfen. Perl hat aber auch eine Funktion namens* `grep` *(nach dem Unix-Suchbefehl), die das für Sie übernimmt. Am Tag 8 werden Sie mehr über* `grep` *erfahren.*

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Erklären Sie die Unterschiede zwischen Listen und Arrays.
2. Was ist das Ergebnis dieser Liste?

```
@liste = (1, (), (4, 3), $foo, ((), 10, 5 + 4), (), (''));
```

3. Was bewirken die folgenden Perl-Anweisungen? Begründen Sie Ihre Antworten.

```
  ($x, $y, $z) = ('a', 'b');
($u, @mehr, $v) = (1 .. 10);
$zahlen[4] = (1, 2, 3);  # @zahlen enthaelt anfangs (10,9,8)
undef $zahlen[4];
$zahlen[$#zahlen];
$foo = @zahlen;
@mehr = 4;
```

4. Nach welcher Regel wird eine Liste in einen Skalar konvertiert?
5. Wie sortiert man ein Array?
6. Was ist Unterschied zwischen `<STDIN>` in skalarem und in Listenkontext? Warum ist das wichtig?

Übungen

1. Schreiben Sie ein Skript, das den Benutzer zur Eingabe von zwei Zahlen auffordert und dann ein Array aus den Zahlen dazwischen erstellt (der Benutzer soll sowohl die niedrigere als auch die höhere Zahl als erste eingeben können).
2. Schreiben Sie ein Skript, das Sie um zwei Arrays bittet und dann ein drittes Array erstellt, das nur die Elemente enthält, die in ***beiden*** anderen Arrays vorkommen (die Schnittmenge dieser Arrays).
3. FEHLERSUCHE: Was ist falsch an diesem Skript? (Tipp: Es könnten mehrere Fehler sein!)

```
    print 'Geben Sie eine Zahlenliste ein: ';
chomp($in = <STDIN>);
@zahlen = split(" ", $in);
@sortiert = sort @zahlen;
print "eingegebene Zahlen: @sortiert\n";
$anz_zahlen = $#zahlen;
print "Anzahl eingegebener Zahlen: $anz_zahlen\n";
```

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Eine Liste ist einfach ein Satz von skalaren Elementen. Ein Array ist eine geordnete Liste, auf deren Positionen über Indizes zugegriffen werden kann.
2. Das Ergebnis der Liste ist: `(1, 4, 3, $foo, 10, 9, '')`. Dabei wird für `$foo` der aktuelle Wert von `$foo` eingesetzt.
3. Die Antworten sind:

1. a. `$x` wird `'a'`, `$y` wird `'b'`, `$z` wird undefiniert. Zuweisungen an Listen auf der linken Seite finden parallel statt, die Werte auf der rechten Seite werden den korrespondierenden Variablen auf der linken Seite zugewiesen.

1. b. `$u` wird `1`, `@mehr` wird `(2,3,4,5,6,7,8,9,10)`, `$v` wird undefiniert. Arrayvariablen auf der linken Seite einer Listenzuweisung sind gefräßig: Sie »schlucken« alle verbleibenden Werte auf der rechten Seite.

1. c. `$zahlen[4]` wird `3`. Beim Zuweisen einer rohen Liste an einen Skalar weist Perl nur den letzten Wert der Liste zu und ignoriert alle vorigen Werte.

1. Wenn der ursprüngliche Wert von `@zahlen (10,9,8)` war, ist der neue Wert von `@zahlen` jetzt `(10,9,8,undef,3)`. Wie gesagt werden bei Zuweisung von Listensyntax in skalarem Kontext alle Werte außer dem letzten ignoriert.

1. d. `$zahlen[4]` wird auf den undefinierten Wert gesetzt (vorher war es `3`).

1. e. `$zahlen[$#zahlen]` verweist auf den Wert an der letzten Position der Liste.
1. f. `$foo` enthält nach dieser Zuweisung die Anzahl der Elemente im Array `@zahlen`.
1. g. Die `4` wird zu einer Liste »befördert«, und `@mehr` wird eine Liste mit einem Element: `(4)`.
4. Fangfrage! Es gibt gar keine Regel für die Konvertierung einer Liste in einen Skalar. Außerdem gibt es eine solche Konvertierung überhaupt nicht. Listen verhalten sich in skalarem Kontext verschieden, je nachdem wie Sie sie verwenden.
5. Man sortiert ein Array mit der `sort`-Funktion:

```
    @sortiert = sort @array;
```

6. In skalarem Kontext liest `<STDIN>` eine Eingabezeile ein, die dann zu Ende ist, wenn der Benutzer die Eingabetaste drückt, und speichert diese Zeile in einer Skalarvariablen. In einem Listenkontext liest `<STDIN>` alle Zeilen von der Standardeingabe, bis es auf ein Dateiendezeichen stößt, und speichert jede Zeile als ein Element in einer Liste. Das sind zwei ganz verschiedene Arten, Daten in Ihr Programm einzulesen.

Lösungen zu den Übungen

1. Hier eine mögliche Antwort:

```
    #!/usr/bin/perl -w
$eins = 0;
$zwei = 0;
print 'Geben Sie eine Bereichsgrenze ein: ';
chomp ($eins = <STDIN>);
print 'Geben Sie die andere Bereichsgrenze ein: ';
chomp ($zwei = <STDIN>);
if ($eins < $zwei) {
    @array = ($eins .. $zwei);
} else {
    @array = ($zwei .. $eins);
}
print "@array \n"
```

2. Hier eine Lösung, die von `foreach`-Schleifen Gebrauch macht:

```
    #!/usr/bin/perl -w
$input = '' ; # Zwischenspeicher für Eingaben
@array1 = (); # erstes Array
@array2 = (); # zweites Array
@final = ();  # Schnittmenge
print 'Geben Sie das erste Array ein: ';
chomp($input = <STDIN>);
@array1 = split(' ', $input);
print 'Geben Sie das zweite Array ein: ';
chomp($input = <STDIN>);
@array2 = split(' ', $input);
foreach $el (@array1) {
    foreach $el2 (@array2) {
        if (defined $el2 && $el eq $el2) {
            $final[$#final+1] = $el;
            undef $el2;
            last;
        }
    }
}
print "@final\n";
```

1. Anstatt der Zeile, die `$el` dem letzten Element im Array zuweist, könnten Sie auch die `push`-Funktion verwenden, die genau das gleiche tut (aber etwas einfacher zu lesen ist):

```
    push @final $el;
```

3. Fangfrage! Der einzige Fehler ist hier in der Zeile $anz_zahlen = $# zahlen. Die Annahme, dass `$#zahlen` die Anzahl der Elementen sei, ist falsch (`$#zahlen` liefert den höchsten Index, und der ist um eins kleiner als die Anzahl von Elementen). Verwenden Sie statt dessen:

```
$anz_zahlen = @zahlen
```

oder

```
$anz_zahlen = $#zahlen + 1.
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Mit Hashes arbeiten

Neben Arrays und Listen gibt es in Perl eine weitere Datenstruktur, in der Sie Daten auflisten können: die assoziativen Arrays, kurz Hashes genannt. In vielen Situationen - abhängig von der Art der Daten und ihrer Verwendung - eignen sich Hashes weit besser als Arrays, um Daten zu speichern und auf sie zuzugreifen.

Heute befassen wir uns mit Hashes. Die Themen sind heute:

- Unterschiede zwischen Arrays und Hashes
- Definition von Hashes
- Zugriff auf Elemente in einem Hash
- Hashes und Kontext
- Und: wie Sie mit `split` einen String in eine Liste (oder einen Hash) konvertieren

Hashes im Vergleich zu Arrays und Listen

Sie haben gestern gelernt, dass eine Liste ein Satz von Skalaren und ein Array eine (nach Elementposition) geordnete Liste ist. Mit einem Hash kann man ebenfalls eine Sammlung von Daten darstellen, jedoch werden die Daten auf eine andere Art organisiert.

Ein Hash ist ein ***ungeordneter*** Satz von Paaren aus Schlüsseln und Werten. Jedem Schlüssel ist ein Wert zugeordnet, wobei sowohl Schlüssel als auch Wert eine beliebige Art von skalarem Wert sein können (siehe Abbildung 5.1). Sie können auf ein Element (sprich ein Paar) in einem Hash zugreifen, indem Sie sich auf den Schlüssel beziehen.

Weder die Schlüssel noch die Werte stehen in irgendeiner Ordnung - Sie können nicht auf das erste oder letzte Element in einem Hash verweisen.

Hashes sind in vieler Hinsicht nützlicher als Arrays, vor allem wenn man auf ein Element lieber mit einer expliziten Bezeichnung (einem Hash-Schlüssel) als mit einer bloßen Nummer (einem Array-Index) zugreifen möchte.

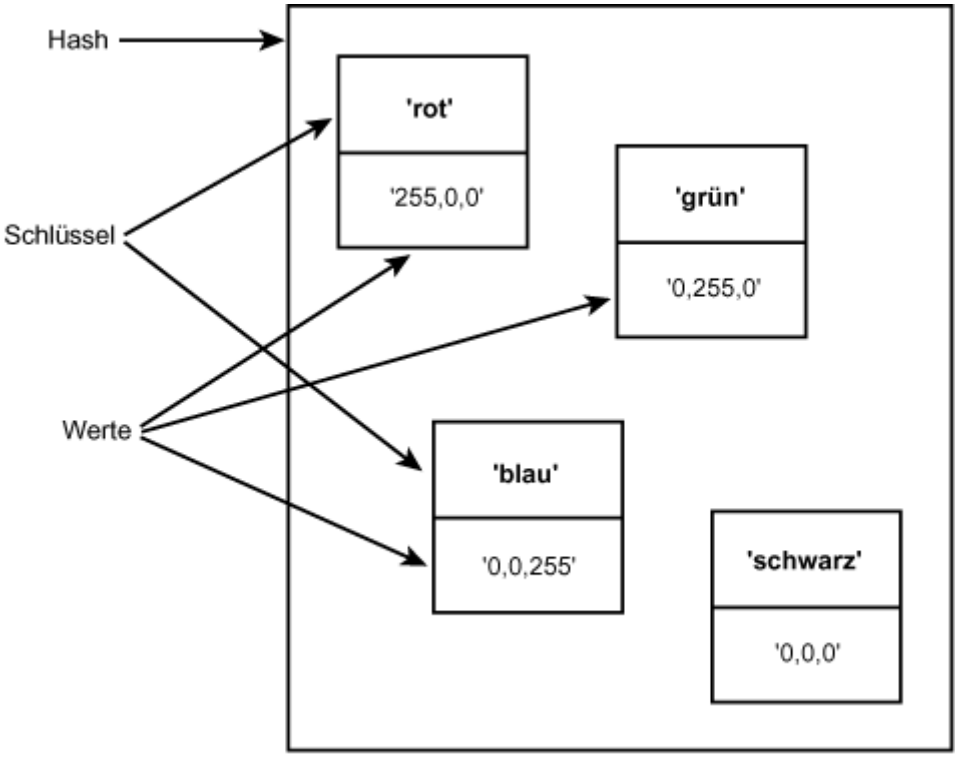

Abbildung 5.1: Hashes

*Hashes werden auch assoziative Arrays genannt. »Assoziatives Array« ist sogar die ursprüngliche, korrekte Bezeichnung, die im Grunde besser beschreibt, was Hashes eigentlich sind (die Schlüssel werden mit ihren Werten assoziiert). Doch mittlerweile bevorzugen viele Perl-Programmierer (wie ich auch) den viel kürzeren und weniger zungenbrecherischen Namen **Hash**.*

Wie Arrays haben Hashes ihre eigenen, als solche gekennzeichneten Variablen: Diese beginnen mit einem Prozentzeichen (%) und folgen denselben Regeln wie Arrayvariablen. Wie bei allen Variablen ist die Hash-Variable `%x` etwas anderes als die Arrayvariable `@x` oder die Skalarvariable `$x`.

Hashes

Hashes sind Listen und den Arrays in Erstellung und Gebrauch sehr ähnlich. Doch weil Hashes ihre Daten anders speichern, haben sie ein paar Besonderheiten. Wenn Sie zum Beispiel Daten in ein Hash packen, müssen Sie bei jedem Element auf zwei Skalare achten (den Schlüssel und den Wert). Und weil Hashes nicht geordnet sind, ist das Sortiern von Hash-Elementen etwas aufwendiger. Außerdem verhalten sich Hashes in skalarem Kontext anders als Arrays. Lesen Sie weiter, ich werde Ihnen all dies jetzt erklären.

Hashes und Listensyntax

Die Listensyntax (Klammer auf, durch Kommata getrennte Elemente, Klammer zu) kann zum Erstellen von Hashes ebenso verwendet werden wie für Arrays. Alles was Sie tun müssen, ist, Ihre Daten innerhalb von Klammern aufzulisten und dann eine Hash-Variable auf die linke Seite der Zuweisung zu stellen:

```
%paare = ('rot', 255, 'grün', 150, 'blau', 0);
```

Mit einer Arrayvariablen ergäbe dies ein Array von sechs Elementen. Mit einer HashVariablen (`%paare`) werden die Elemente dem Hash paarweise zugewiesen: Das erste Element ist ein Schlüssel, das zweite sein Wert, das dritte Element ist der zweite Schlüssel und das vierte dessen Wert und so weiter und so fort. Wenn die Anzahl der Elemente ungerade ist, wird das letzte ignoriert.

Nun kann man bei dieser Schreibweise kaum auf den ersten Blick erkennen, was in der Liste ein Schlüssel und was ein Wert ist (und je länger die Liste ist, mit der Sie den Hash initialisieren, desto schwieriger wird es). Viele Perl-Programmierer setzen deswegen in ihrer Listensyntax für Hashes die Schlüssel/Wert-Paare jeweils in eigene Zeilen:

```
%temps = (
   'Boston', 22,
   'New York', 18,
   'Miami', 32,
   'Portland', 25,
    # und so weiter...
);
```

Noch besser als diese Formatierung ist der =>-Operator, der sich exakt genauso verhält wie das Komma zwischen Schlüssel und Wert. Die Verbindung zwischen den Schlüsseln und Werten wird durch den =>-Operator noch deutlicher. Das erste Beispiel mit den Farben sieht dann wie folgt aus:

```
%paare = ('rot'=>255, 'grün'=>150, 'blau'=>0);
```

Und das zweite mit den Städten:

```
%temps = (
   'Boston' => 22,
   'New York' => 18,
   'Miami' => 32,
   'Portland' => 25,
    # und so weiter...
);
```

Und noch etwas: Perl geht bei jedem Schlüssel eines Hash-Elements davon aus, dass es sich um Strings handelt. Und weil der Schlüssel ohnehin ein String sein muss, können Sie sich etwas Tipparbeit sparen und die Anführungszeichen auch weglassen:

```
%paare = (rot=>255, grün=>150, blau=>0);
```

Wenn der Schlüssel allerdings ein Leerzeichen enthält, müssen Sie die Anführungszeichen setzen (so smart ist Perl auch wieder nicht).

Wie bei Arrays erzeugt das Zuweisen einer leeren Liste `()` an eine Hash-Variable einen leeren Hash:

```
%hash = ();  #keine Schlüssel und keine Werte
```

Konvertieren zwischen Arrays, Listen und Hashes

Eine zweite Möglichkeit, einen Hash zu erzeugen, ist die Initialisierung mit einem Array oder einer Liste. Weil die »Rohform« der Inhalte von Hashes und Arrays jeweils Listen sind, können Sie ohne Probleme zwischen den beiden hin- und herkopieren:

```
@zeug = ('eins', 1, 'zwei', 2);
%paarweises_zeug = @zeug
```

In diesem Beispiel werden durch die Zuweisung des Arrays `@zeug` an den Hash `%paarweises_zeug` die Array-Elemente in eine Liste expandiert, dann in zwei Schlüssel/Wert-Paare zerlegt und im Hash gespeichert. Das Ergebnis wäre das gleiche, wenn Sie alle Elemente in Listenform eingetippt hätten. Passen Sie aber auf die Anzahl der Elemente auf - ist sie ungerade, wird das letzte Element ignoriert und nicht in den Hash übernommen (Perl-Warnungen geben Ihnen Bescheid, wenn das passiert).

Und wie ist es mit der Rückwandlung von einem Hash in eine Liste? Im folgenden Beispiel weisen Sie einen Hash einem Array zu:

```
@zeug = %paarweises_zeug;
```

Wenn Sie einen Hash auf die rechte Seite einer Listenzuweisung stellen oder vielmehr wann immer Sie einen Hash

in einer Situation verwenden, in der eine rohe Liste erwartet wird, »dröselt« Perl den Hash in seine einzelnen Elemente auf (Schlüssel, Wert, Schlüssel, Wert und so fort). Diese Liste wird dann dem Array `@zeug` zugewiesen.

Dieses schöne Aufschlüsselungsverhalten hat einen Haken: Weil Hashes nicht geordnet sind, stehen die Schlüssel/Wert-Paare, die Sie aus einem Hash herausziehen, höchstwahrscheinlich weder in derselben Reihenfolge, in der Sie sie eingefügt haben, noch sind sie nach einem anderen augenscheinlich sinnvollen Kriterium sortiert. Hash-Elemente werden in einem internen Format gespeichert, das den Zugriff sehr schnell vornimmt (die Geschwindigkeit ist sozusagen Perls einziges Anordnungskriterium), und in dieser internen, so gut wie unvorhersagbaren Reihenfolge werden sie auch wieder hervorgeholt. Wenn Sie eine Liste aus einem Hash in einer bestimmten Reihenfolge brauchen, müssen Sie eine Schleife bauen, die die Elemente nach Ihren Vorgaben gezielt aus dem Hash zieht (mehr darüber später).

Auf Hash-Elemente zugreifen

Anders als Arrays, die lediglich Werte in einer bestimmten Reihenfolge enthalten, bestehen Hashes wie gesagt aus Schlüssel/Wert-Paaren. Um auf einen Wert in einem Hash zuzugreifen, müssen Sie seinen Schlüssel (auch ***Key*** genannt) kennen. Mit dem Schlüssel in geschweiften Klammern ({}) können Sie folgendermaßen auf einen Hash-Wert zugreifen:

```
print $temps{'Portland'};
$temps{'Portland'} = 50;
```

Sie sehen, dass diese Syntax der Array-Zugriffssyntax `$array[]` sehr ähnlich ist - Sie greifen mit einer Skalarvariablen (`$temps`) und geschweiften Klammern um den Schlüsselnamen (anstatt eckiger um einen Array-Index) auf einen skalaren Wert innerhalb des Hash zu. Der Schlüssel innerhalb der Klammern sollte ein String sein (hier haben wir einen ***single-quoted*** String genommen), Ziffern werden gegebenenfalls in Strings umgewandelt. Außerdem können Sie, wenn der Schlüssel nur ein einziges Wort enthält, die Anführungszeichen weglassen, Perl versteht auch so, was Sie meinen:

```
$temps{Portland} = 50;    # ist das gleiche wie $temps{'Portland'} = 50;
```

Wie bei den Arrays kommt der Variablenname in der Zugriffssyntax nicht mit gleichnamigen Skalarvariablen in Konflikt. Jede der folgenden Variablen verweist auf etwas anderes, obwohl der Variablenname immer derselbe ist:

```
$name        # ein Skalar
@name        # ein ganzes Array
%name        # ein ganzer Hash
$name[$index]   # der Skalar im Array @name an der Stelle $index
$name{'key'}    # der Skalar im Hash %name mit dem Schlüssel 'key'
```

Hash-Elemente löschen

Mit der eben besprochenen Elementzugriffssyntax können Sie ein Hash-Element hinzufügen, darauf zugreifen und es ändern. Aber wie werden Sie Elemente, die Sie nicht mehr brauchen, wieder los? Dafür stellt Perl die Funktion `delete` zur Verfügung. Diese Funktion nimmt den Verweis auf ein Hash-Element entgegen (im allgemeinen einen Hash-Zugriffsausdruck wie `$hashname{'key'}`), löscht sowohl den Schlüssel als auch den ihm zugeordneten Wert und gibt den gelöschten Wert zurück. Das bedeutet, dass Sie mit `delete` ein Element nicht nur löschen, sondern auch von einem Hash zu einem anderen verschieben können (es also aus dem einen löschen und dem anderen hinzufügen), wie in folgendem Beispiel:

```
$hash2{$key} = delete $hash{$key};
```

Sie können auch überprüfen, ob ein bestimmtes Schlüssel/Wert-Paar in einem Hash existiert: Die Funktion `exists` durchsucht einen Hash nach einem ihr übergebenen Schlüssel und gibt ***wahr*** zurück, wenn sie ihn findet. Beachten Sie, dass der zu dem gefundenen Schlüssel zugehörige Wert sehr wohl undefiniert sein könnte - `exists` prüft wirklich nur, ob der Schlüssel vorhanden ist. Verwenden Sie `exists` folgendermaßen.

```
if (exists $hashname{$key})  { $hashname{$key}++; }
```

Dieser Ausdruck prüft, ob der Schlüssel `$key` vorhanden ist, und wenn ja, inkrementiert er den diesem Schlüssel

zugeordneten Wert (vorausgesetzt natürlich, dieser Wert ist eine Zahl).

Auf alle Werte in einem Hash zugreifen

Angenommen, Sie möchten alle Werte in einem Array oder einer Liste durchgehen, jeden einzelnen auf eine bestimmte Eigenschaft überprüfen und unter bestimmten Bedingungen verwenden. Bei einem Array würden Sie mit Element 0 anfangen und den Vorgang so lange wiederholen, bis Sie beim letzten Element der Liste angelangt sind (oder eine `foreach`-Schleife verwenden). Aber wie geht das mit Hashes? Eine Reihenfolge gibt es nicht, und die Schlüssel können irgendwelche skalaren Werte sein. Was Sie brauchen, ist eine Methode, ein paar Informationen aus dem Hash zu ziehen, mit denen Sie die Struktur dann durchlaufen können.

Für dieses Problem stehen die Funktionen `keys` und `values` zur Verfügung. Diese Funktionen nehmen beide einen Hash als Argument entgegen und geben eine Liste zurück - `keys` eine Liste der Schlüssel und `values` eine Liste der Werte im angegebenen Hash. Wenn Sie dann mit dieser Liste und `foreach` oder einer anderen Schleife auf jedes einzelne Hash-Element zugreifen, kommen Sie an wirklich alle Elemente - auch die, die Sie vielleicht längst vergessen haben.

Nehmen wir zum Beispiel einen Hash mit einer nach Städtenamen aufgeschlüsselten Liste von Temperaturen (wie wir es bereits vorhin in einem Beispiel hatten). Diese Liste möchten wir jetzt alphabetisch nach Städten sortiert ausgeben. Dafür erstellen wir mit `keys` eine Liste aller Schlüssel, sortieren diese Liste mit `sort` und geben die sortierten Schlüssel und ihre Werte dann in einer `foreach`-Schleife aus, etwa so:

```
foreach $stadt (sort keys %temps) {
    print "$stadt: $temps{$stadt} grad\n";
}
```

Diese Schleife durchläuft nacheinander jedes Element der sortierten Schlüsselliste und weist es der Variablen `$stadt` zu. Im Schleifenkörper können Sie sich dann mit dieser Variablen auf das aktuelle Element beziehen.

Hashes und Kontext

Lassen Sie uns auf den Kontext zurückkommen und betrachten, wie Hashes sich in den verschiedenen Kontexten verhalten. Zumeist gelten dieselben Regeln wie für Listen, doch es gibt ein paar Ausnahmen.

Ich habe Ihnen gezeigt, wie man mit Listensyntax einen Hash erstellt, wobei der Hash die Elemente dann als Schlüssel/Wert-Paare speichert wie hier:

```
%paare = (rot=>255, grün=>150, blau=>0);
```

Im umgekehrten Fall, wenn Sie einen Hash verwenden, wo eine Liste erwartet wird, wird der Hash (in beliebiger Reihenfolge) zurück in seine Einzelteile zerlegt und folgt dann den gleichen Regeln wie alle anderen Listen.

```
@farben = %paare;      # ergibt ein Array aus allen Elementen
($x, $y, $z) = %paare; # die ersten drei Elemente des aufgelösten
                       # Hash werden Variablen zugewiesen,
                       # verbleibende Elemente werden ignoriert
print %paare;          # zerlegt den Hash in seine Elemente und
                       # gibt sie aus
```

Immer wenn Sie einen Hash in einem Listenkontext verwenden - zum Beispiel auf der rechten Seite einer Zuweisung an ein Array -, wird der Hash in eine Liste seiner Einzelteile »aufgedröselt«, und diese Liste benimmt sich dann wie in jedem anderen Listen- oder skalaren Kontext auch. Es gib allerdings einen Sonderfall:

```
$x = %paare;
```

Auf den ersten Blick könnte man meinen, dies sei das Hash-Äquivalent zu `$x = @array` (Sie erinnern sich, damit ermitteln Sie die Anzahl der Elemente in einem Array). Aber Perl verhält sich hier anders als bei Arrays - das Ergebnis `$x` ist hier nämlich eine Beschreibung des internen Zustands der Hash-Tabelle, was in 99 % der Fälle wahrscheinlich nicht das ist, was Sie wollen. Um die Anzahl der Elemente (Schlüssel/Wert-Paare) in einem Hash zu erhalten, verwenden Sie statt dessen die `keys`-Funktion und weisen die Schlüsselliste einer Skalarvariablen zu:

```
$x = keys %paare;
```

Die Funktion `keys` gibt eine Liste der Schlüssel im Hash zurück, die dann in skalarem Kontext ausgewertet die Anzahl der Elemente liefert.

*Sind Sie neugierig, was ich mit »eine Beschreibung des internen Zustands der Hash-Tabelle« meine? Okay, dann werde ich es kurz erklären. Die Zuweisung einer Hash-Variablen in skalarem Kontext liefert Ihnen zwei Zahlen, getrennt durch einen Schrägstrich. Die zweite Zahl ist die Summe der zur Verfügung stehenden **Slots**, das heißt Speicherstellen, die für die interne Hashtabelle reserviert wurden (oft auch »buckets« genannt). Die erste Zahl ist die Anzahl der tatsächlich von den Daten genutzten **Slots**. Sinnvoll werden diese beiden Zahlen, wenn Sie wissen möchten, wie effizient eine Hash-Tabelle ist: Eine Hash-Beschreibung von 4/100 würde bedeuten, dass der Hash nur 4 von 100 bereitgestellten Buckets verwendet: schlechte Nachricht über die Effizienz Ihres Skripts. Den Aufbau fortgeschrittener - und effizienter - Datenstrukturen werden wir an Tag 19 behandeln.*

Ein Beispiel: Häufigkeiten im Statistikprogramm

Ändern wir noch einmal unser Statistikskript. Erweitern wir es diesmal dahin, dass es sich merkt, wie oft jede Zahl jeweils eingegeben wurde, und das Ergebnis schließlich als Balkendiagramm darstellt. Hier ein Beispiel, wie das Diagramm aussehen könnte (abgesehen von diesem Histogramm ist die Ausgabe dieselbe wie vorher; deswegen werde ich das hier nicht noch einmal alles aufführen):

Häufigkeit der einzelnen Zahlen:

```
1  | *****
2  | **********
3  | ******************
4  | ****************
5  | ***********
6  | ****
43 | *
62 | *
```

Um das Vorkommen jeder Zahl in unserem Skript zu verfolgen, verwenden wir einen Hash mit den neu eingegebenen Zahlen als Schlüssel und den Häufigkeiten, mit denen die Zahlen auftauchen, als Werte. Der Diagrammabschnitt des Skripts durchläuft dann alle Elemente dieses Hash und gibt in einer grafischen Darstellung aus, wie oft jede Zahl eingegeben wurde.

Listing 5.1 zeigt den Perl-Code für unser neues Skript:

Listing 5.1: nochmehrstats.pl.

```
1:  #!/usr/bin/perl -w
2:
3:  $input = '';  # Benutzereingabe: Zahl
4:  @nums = ();   # Array: Zahlen;
5:  %freq = ();   # Hash: Zahl-Haeufigkeit
6:  $count = 0;   # Anzahl aller Zahlen
7:  $sum = 0;     # Summe
8:  $avg = 0;     # Durchschnitt
9:  $med = 0;     # Median
10: $maxspace = 0;# maximaler Platz für die Schluessel
11:
12: while () {
13:   print 'Geben Sie eine Zahl ein: ';
14:   chomp ($input = <STDIN>);
15:   if ($input ne '') {
16:     $nums[$count] = $input;
17:     $freq{$input}++;
18:     $count++;
19:     $sum += $input;
```

```
20:    }
21:   else { last; }
22: }
23:
24: @nums = sort { $a <=> $b } @nums;
25: $avg = $sum / $count;
26: $med = $nums[$count /2];
27:
28: print "\n Anzahl der eingegebenen Zahlen: $count\n";
29: print "Summe der Zahlen: $sum\n";
30: print "Kleinste Zahl: $nums[0]\n";
31: print "Groesste Zahl: $nums[$#nums]\n";
32: printf("Durchschnitt: %.2f\n", $avg);
33: print "Mittelwert: $med\n\n";
34: print "Haeufigkeit der einzelnen Zahlen:\n";
35:
36: $maxspace = (length $nums[$#nums]) + 1;
37:
38: foreach $key (sort { $a <=> $b } keys %freq) {
39:   print $key;
40:   print ' ' x ($maxspace - length $key);
41:   print '| ', '*' x $freq{$key}, "\n";
42: }
```

Dieses Skript unterscheidet sich nicht sehr vom vorigen; die einzigen Änderungen stehen in den Zeilen 5, 10, 17 und im letzten Abschnitt von Zeile 36 bis 42. Betrachten Sie diese Zeilen einmal etwas genauer, und beachten Sie auch, wo und wie sie sich in das Skript fügen, das wir bereits geschrieben haben.

Die Zeilen 5 und 10 sind ganz einfach; sie definieren lediglich neue Variablen, die wir später im Skript verwenden: Der Hash `%freq` speichert die Häufigkeit der eingegebenen Zahlen, und die Variable `$maxspace` enthält einen Zwischenwert zur Formatierung des Diagramms (mehr dazu, wenn wir zum Erstellen des Diagramms kommen).

Zeile 17 ist da viel interessanter. Sie steht innerhalb der Schleife, mit der wir die Daten einlesen. In der Zeile davor haben wir die zuletzt eingegebene Zahl dem Zahlenarray hinzugefügt. Die Zahl selbst ist der Schlüssel, und wie oft sie bis jetzt eingegeben wurde, ist der Wert. In Zeile 17 verwenden wir die eingegebene Zahl als Schlüssel des Häufigkeits-Hash. Wenn die Zahl bereits als Schlüssel vorhanden ist, erhöhen wir den ihr zugeordneten Wert um 1 (mit dem ++-Operator). Wenn kein solcher Schlüssel existiert, unsere Zahl also noch nicht im Hash enthalten ist, fügen wir sie damit hinzu und erhöhen den Wert auf 1.

Bei jedem weiteren Durchlauf inkrementieren wir nur dann den Wert (sprich die Häufigkeit), wenn genau diese Zahl wieder in den Benutzereingaben auftaucht.

So haben wir nach Beendigung der Eingabeschleife schließlich einen Hash, der jede eingegebene Zahl genau einmal als Schlüssel und ihre jeweilige Häufigkeit als Werte enthält. Jetzt muss nur noch ein Diagramm mit diesen Daten ausgegeben werden.

Anstatt wie in den bisherigen Beispielen Schritt für Schritt die Zeilen 36 bis 42 durchzugehen, möchte ich Ihnen nun zeigen, in welchen Schritten ich diese Schleife ***geschrieben*** habe. So bekommen Sie einen Einblick in meine Denkweise und die Entstehung dieser Schleife. Damit wird deutlicher, warum sie ist, wie sie ist.

Als erstes möchte ich die Werte in die richtige Reihenfolge bringen. Also fange ich mit einer `foreach`-Schleife an, ähnlich der, die ich heute im Abschnitt »Auf alle Werte in einem Hash zugreifen« beschrieben habe.

```
foreach $key (sort { $a <=> $b } keys %freq) {
  print "Schluessel: $key Wert: $freq{$key}\n";
}
```

In dieser Schleife verwende ich foreach, um auf jeden Hash-Schlüssel zuzugreifen. In welcher Reihenfolge das geschieht, wird jedoch von dem eingeklammerten Ausdruck in der ersten Zeile kontrolliert. Der `keys %freq`-Teil erstellt eine Liste aller Schlüssel des Hash, `sort` sortiert sie (Sie erinnern sich, `sort` sortiert standardmäßig in ASCII- Reihenfolge, erst das Hinzufügen von `{$a <=> $b}` erzwingt eine numerische Sortierung). Das Ergebnis ist, dass der Hash vom kleinsten zum größten Schlüssel durchgegangen wird.

Innerhalb der Schleife muss ich dann nur noch die Schlüssel und die Werte ausgeben. Mit ein paar einfachen Daten

erhielte ich dann eine Ausgabe wie diese:

```
Schluessel: 2 Wert: 4
Schluessel: 3 Wert: 5
Schluessel: 4 Wert: 3
Schluessel: 5 Wert: 1
```

Das ist zwar eine gute Darstellung der Werte im `%freq`-Hash, aber kein Histogramm. Mein zweiter Schritt ist die Veränderung der `print`-Anweisung. Ich verwende den wunderbaren String-Wiederholungsoperator (`x`), um die der Häufigkeit der Zahlen entsprechende Anzahl Sternchen auszugeben:

```
foreach $key (sort { $a <=> $b } keys %freq) {
  print '$key |', '*' x $freq{$key}, "\n";
}
```

Damit komme ich der Sache schon näher. Die Ausgabe sähe jetzt etwas so aus:

```
2 | ****
3 | *****
4 | ***
5 | *
```

Problematisch wird es aber, wenn die eingegebenen Zahlen größer als 9 sind. Wenn nicht alle Schlüssel gleich viele Stellen haben, würde mein schönes Diagramm geradezu aufgeschraubt. Wenn ich zufällig eine vierstellige Zahl zwischen meinen ein- und zweistelligen Zahlen hätte, sähe das Histogramm wie folgt aus:

```
2 | ****
3 | *****
4 | ***
5 | *
13 | **
24 | *
45 | ***
2345 | *
```

Was also tun? Ich muss dafür sorgen, dass vor dem Pipe-Zeichen (|) immer die richtige Anzahl Leerzeichen steht, damit die Sternchen im Histogramm in derselben Spalte beginnen. Ich habe dieses Problem mit der Funktion `length` gelöst, die die Anzahl der Zeichen (genaugenommen der Bytes) in einem skalaren Wert liefert.

Zunächst muss ich herausfinden, wie breit die Schlüsselspalte überhaupt sein soll, das heißt wie viele Stellen der »breiteste« Schlüssel hat. Ich nehme dafür die Länge der größten Zahl im Array `@zahlen` und addiere eine 1, um ein Leerzeichen am Ende anzuhängen:

```
$maxspace = (length $nums[$#nums]) + 1;
```

Innerhalb der Schleife verändere ich noch einmal meine Print-Anweisungen. Diesmal teile ich dem String-Wiederholungsoperator mit, wie viele Leerzeichen nach einem Schlüssel ausgegeben werden sollen - Spaltenbreite minus Schlüssellänge. So wird der Unterschied zwischen kleineren und der größten Zahl stets korrekt mit Leerzeichen aufgefüllt, und ich kann mit der Ausgabe der Pipes und Sternchen weitermachen:

```
foreach $key (sort { $a <=> $b } keys %freq) {
  print $key;                                  # den Schluessel ausgeben
  print ' ' x ($max_breite - length $key);  # bis zur max.-Breite mit
                                               # Leerzeichen auffuellen
  print '| ', '*' x $freq{$key}, "\n";       # die Sternchen ausgeben
}
```

Jetzt sieht das Histogramm aus, wie ich es Ihnen ganz am Anfang gezeigt habe. Ich bin fertig. Den kompletten Code haben Sie in Listing 5.1 bereits gesehen.

Die Art, wie ich hier das Diagramm formatiert habe, ist nicht gerade elegant. Sie sollten sich diese

*Methode nicht zum Vorbild nehmen, wenn Sie das Ausgabeformat Ihrer Daten festlegen, insbesondere wenn Sie dabei mit mehr als den paar Zeichen in diesem Beispiel zu tun haben. Perl (Sie erinnern sich, die **praktische Extraktions- und Report**-Sprache) hat einen Satz spezieller Datenformatierungs-Prozeduren, mit denen sich solche Reports viel einfacher erstellen lassen. Im HTML-Zeitalter wird mit Perl-Formatierungen nicht mehr viel gearbeitet, doch an Tag 20 gebe ich Ihnen zumindest eine kleine Kostprobe.*

Einen String mit *split* in Teilstrings zerlegen

Über die Tastatur eingegebene Daten sind meist recht unkompliziert zu verarbeiten - Sie brauchen sie nur einer Variablen zuweisen und können dann damit anstellen, was immer Sie wollen. Aber in vielen Fällen sind - insbesondere aus Dateien eingelesene - Eingabedaten nicht so einfach zu handhaben. Was ist, wenn Sie Daten mit nicht einer, sondern gleich zehn Zahlen pro Zeile bekommen? Was ist, wenn Sie sich für einen Teil in der Mitte der Zeile interessieren, sich aus dem Rest aber überhaupt nichts machen?

Normalerweise erhalten Sie Ihre Daten in irgendeiner rohen Form, aus der Sie die interessanten Dinge selbst »herauspicken« und speichern müssen. Dafür stellt Perl Ihnen die Funktion `split` zur Verfügung, die einen gegebenen String nach Ihren Vorgaben in eine Liste von Teilstrings aufteilt (***splittet***).

Ihre Vorgaben sind dabei insbesondere die Zeichen oder Zeichenfolgen, bei denen `split` sozusagen »die Schere ansetzen« und den String aufteilen soll. Sie können hier (mit regulären Ausdrücken) die raffiniertesten Suchmuster festlegen. Heute allerdings betrachten wir nur das einfachste aller Trennzeichen: das Leerzeichen, das nicht nur bei Leerzeichen, sondern allen Leerstellen (auch ***Whitespace***, »weißer Raum« genannt und als Sammelbegriff für Leerzeichen, Tabulator, Zeilenvorschub, Wagenrücklauf, Seitenvorschub und vertikalen Tabulator gebraucht) trennt.

Sie übergeben der `split`-Funktion zwei Strings als Argumente - der erste enthält das Trennzeichen, und der zweite ist der String, den Sie bei jedem Vorkommen des Trennzeichens splitten möchten. Die `split`-Funktion teilt den String entsprechend auf und gibt Ihnen eine Liste der Teilstrings zurück, die Sie beispielsweise einer Arrayvariablen zuweisen und dann weiterarbeiten können. Der folgende Perl-Code zum Beispiel zerlegt die Zahlenfolge im String `$zahlenfolge` in ein Array von Zahlen:

```
$zahlenfolge = '34 23 56 34 78 38 90';
@zahlen = split(' ', $zahlenfolge);
```

So! Jetzt können Sie mit den Zahlen im Array `@zahlen` nach Belieben herumspielen.

Ein weiteres Beispiel: Eine alphabetische Namensliste

Zum Abschluß dieser Lektion wollen wir Hashes und `split` zusammen in einem kleinen Beispiel einsetzen, das eine Namensliste einliest, diese Namen in einen nach Nachnamen aufgeschlüsselten Hash packt und sie dann mit dem Nachnamen zuerst und in alphabetischer Reihenfolge ausgibt. Das Ganze könnte zum Beispiel so aussehen:

```
Geben Sie einen Namen ein (Vor- und Nachname): Umberto Eco
Geben Sie einen Namen ein (Vor- und Nachname): Kurt Vonnegut
Geben Sie einen Namen ein (Vor- und Nachname): Fjodor Dostojewski
Geben Sie einen Namen ein (Vor- und Nachname): Albert Camus
Geben Sie einen Namen ein (Vor- und Nachname): Paul Auster
Geben Sie einen Namen ein (Vor- und Nachname): George Orwell
Geben Sie einen Namen ein (Vor- und Nachname):
Auster, Paul
Camus, Albert
Dostojewski, Fjodor
Eco, Umberto
Orwell, George
Vonnegut, Kurt
```

Listing 5.2 zeigt das zugehörige Skript:

Listing 5.2: Das Skript namen.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $in = '';      # temp Input
4:  %names = ();   # Hash Namen
5:  $fn = '';      # temp Vorname
6:  $ln = '';      # temp Nachname
7:
8:  while () {
9:      print 'Geben Sie einen Namen ein (Vor- und Nachname): ';
10:     chomp($in = <STDIN>);
11:     if ($in ne '') {
12:         ($fn, $ln) = split(' ', $in);
13:         $names{$ln} = $fn;
14:     }
15:     else { last; }
16: }
17:
18: foreach $lastname (sort keys %names) {
19:     print "$lastname, $names{$lastname}\n";
20: }
```

Dieses Skript besteht aus drei Abschnitten: die Variablen initialisieren, die Daten einlesen und sie sortiert wieder ausgeben. Der Initialisierungsteil sollte mittlerweile klar sein, aber vielleicht fragen Sie sich, was das `temp` in den Kommentaren bedeutet. Es steht für ***temporär***. Wie Sie wissen, dient der gesamte Initialisierungsabschnitt im Grunde nur der Übersichtlichkeit. Mit einem kurzen `temp` im Kommentar möchte ich hier auf den ersten Blick klarmachen, dass diese Variablen nur Zwischenspeicher für die Vor- und Nachnamen sind - die letztlich ja im Hash `%names` landen.

In Zeile 8 bis 16 lesen wir die Daten ein, und zwar wie Sie es bereits aus dem Statistikskript kennen - mit einer `while`-Schleife, einem `if` zum Überprüfen auf eine Leerzeile und `<STDIN>` in skalarem Kontext. Anders als im Statistikskript packen wir die eingegebenen Strings hier nicht in ein Array, sondern splitten sie in Zeile 12 in zwei temporäre Skalarvariablen, `$fn` und `$ln`, auf. In Zeile 13 fügen wir die Inhalte dieser beiden Variablen dem Hash `%names` hinzu, den Nachnamen als Schlüssel und den Vornamen als Wert.

Wenn alle Daten eingegeben sind, ist unser Hash komplett, und wir können ihn ausgeben. Auch diese Syntax haben Sie bereits gesehen, zuletzt in dem Histogrammbeispiel weiter oben in dieser Lektion. Diesmal sortieren wir die Schlüssel aber in alphabetischer Reihenfolge, deswegen genügt hier die einfachere Form von `sort`. Die `print`-Anweisung in Zeile 19 verwendet schließlich die Variable `$lastname` (die bei jedem `foreach`-Durchlauf den aktuellen Schlüssel enthält), um den Nachnamen und den diesem Schlüssel im Hash zugeordneten Vornamen auszugeben.

Wenn die Schleifen Sie verwirren, versuchen Sie nur die anderen Teile der Beispiele zu verstehen. Morgen, am Tag 6, befassen wir uns ausführlich mit `while` und `foreach`. Dann erfahren Sie ganz genau, was die Schleifen in den Beispielen eigentlich machen.

Vertiefung

Wie Arrays sind auch Hashes Listen; deswegen brauchen wir in dieser Lektion eigentlich nicht viel tiefer zu gehen. Doch eine Funktion, die für den Gebrauch mit Hashes nützlich sein kann, möchte ich hier erwähnen: `each`.

Die Funktion `keys` nimmt einen Hash als Argument entgegen und gibt eine Liste der Schlüssel im Hash zurück; `values` macht das gleiche mit den Hash-Werten. Übergibt man der Funktion `each` einen Hash als Argument, gibt sie ein Schlüssel/Wert-Paar als zweielementige Liste zurück: Das erste Element ist ein Key und das zweite der Wert. Das Besondere ist, dass Sie mit mehrmaligem Aufrufen von `each` den gesamten Hash durchgehen können. Wie bei allen Hash-Elementen ist die Reihenfolge der Paare nicht geordnet. Wenn `each` alle Elemente aus dem Hash gelesen hat, gibt es eine leere Liste `()` zurück.

Zusammenfassung

Heute haben wir mit der Behandlung der Hashes Ihr Grundwissen über Listendaten vervollständigt. Hashes sind Arrays und Listen sehr ähnlich - mit der Ausnahme, dass sie Daten in Schlüssel/Wert-Paaren anordnen, anstatt sie in einer numerischen Reihenfolge zu speichern. Wir haben besprochen, wie man einen Hash in einer Hash-

Variablen `%hash` speichert, auf einen Wert mit `$hash{'key'}` zugreift, Schlüssel aus dem Hash löscht und mit einer `foreach`-Schleife und der `keys`-Funktion alle Hash- Elemente durchgehen kann.

Im folgenden noch einmal die Perl-Funktionen, die Sie heute kennengelernt haben:

- `exists` nimmt einen Hash-Schlüssel entgegen und gibt ***wahr*** zurück, wenn dieser im Hash als Schlüssel vorhanden ist (auch wenn der diesem Schlüssel zugeordnete Wert undefiniert ist).
- `delete` nimmt einen Hash-Schlüssel entgegen und löscht diesen Schlüssel samt Wert aus dem Hash. Anders als `undef`, das einem Element in einem Hash oder Array den undefinierten Wert zuweist, entfernt `delete` das gesamte Paar (Schlüssel und Wert).
- `keys` nimmt einen Hash entgegen und gibt eine Liste aller Schlüssel in diesem Hash zurück.
- `values` nimmt einen Hash entgegen und gibt eine Liste aller Werte in diesem Hash zurück.
- `split` nimmt zwei Strings entgegen und splittet den zweiten String an den Stellen, an denen das im ersten String angegebene Trennzeichen auftaucht, in eine Liste von Teilstrings. Mit einem dritten Argument, einer Zahl, kann man festlegen, wie viele Elemente die von `split` erzeugte Liste höchstens enthalten darf.

Mehr Informationen finden Sie in der ***perlfunc***-Manpage bzw. im Anhang A.

Fragen & Antworten

Frage:
Diese verschiedenen Variablenzeichen! Wie soll ich die denn auseinanderhalten!

Antwort:
Je öfter Sie sie benutzen, desto einfacher wird es auch, sich zu merken, welches wofür verwendet wird. Vielleicht hilft es, beim Skalarvariablenzeichen `$` an den Buchstaben S wie Skalar zu denken (oder wenn Sie in `$` nur Dollars sehen - Dollars sind Zahlen und skalar). Das at- Zeichen @ sieht ein bißchen aus wie ein kleines a. A steht für Array. Und das Prozentzeichen % für Hashes besteht aus einem Schrägstrich mit zwei Punkten - einem für den Schlüssel und einem für den Wert. Wenn Sie auf Arrays oder Hashes zugreifen, denken Sie daran, was Sie haben möchten: Wollen Sie ein einzelnes Element, nehmen Sie `$`. Wollen Sie mehrere (eine Liste), verwenden Sie @.

Frage:
Hashes sind bloß assoziative Arrays, oder nicht? Es sind doch nicht wirklich Hash- Tabellen?

Antwort:
Doch. »Assoziatives Array« ist eine andere, in früheren Perl-Versionen sogar »offizielle« Bezeichnung für einen Hash (»Hash« hat sich durchgesetzt, weil es sich bequemer aussprechen und tippen läßt als »assoziatives Array«, zumindest fanden das die Perl-Programmierer). Hashes sind intern aber auch tatsächlich als echte Hash-Tabellen implementiert und haben insbesondere bei riesigen Datenmengen alle Geschwindigkeitsvorteile eines Hash-Verfahrens gegenüber einem auf Schlüsselvergleich basierenden Verfahren.

Frage:
Sie verwenden in allen Beispielen einen Hash-Key, um auf einen Wert zuzugreifen. Gibt es auch einen Weg, mit einem Wert an einen Schlüssel zu kommen?

Antwort:
Nein. Also, es gibt keine Funktion, die das tut. Sie könnten mit einer `foreach`- Schleife und den Hash-Schlüsseln den Hash durchlaufen, auf einen bestimmten Wert überprüfen und so den entsprechenden Schlüssel herausfinden. Aber bedenken Sie, dass verschiedene Schlüssel durchaus den gleichen Wert haben können, die Beziehung von einem Wert zu seinem Schlüssel also nicht die gleiche ist wie die eines Schlüssels zu seinem Wert.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Erklären Sie die Unterschiede zwischen Listen, Arrays und Hashes.
2. Worauf beziehen sich die folgenden Variablen?

```
    $foo
@foo
%foo
$foo{'key'}
```

3. Was sind die Ergebnisse der folgenden Perl-Anweisungen? Begründen Sie Ihre Antwort.

```
    %zeug = qw(1 eins 2 zwei 3 drei 4 vier);
@zahlen = %zeug
$foo = %zeug;
```

4. Was passiert, wenn Sie einen Hash in einem Listenkontext verwenden? Und in skalarem Kontext?
5. Wie sortiert man einen Hash?
6. Erklären Sie die Unterschiede zwischen den Funktionen `keys`, `values` und `each`.
7. Wozu dient `split`?

Übungen

1. Verändern Sie das Statistikprogramm so, dass der Anwender alle Zahlen in einer Zeile eingeben kann.
2. Schreiben Sie ein Skript, das um die Eingabe eines Satzes bittet und dann die Anzahl der Buchstaben und Wörter sowie den Satz in umgekehrter Wortreihenfolge ausgibt (Tipp: Verwenden Sie die Funktionen `length`, `split` und `reverse`).
3. Schreiben Sie das Skript ***namen.pl*** um, so dass es auch mit zweiten Vornamen oder Initialien umgehen kann (z.B. Jean Paul Sartre oder William S. Burroughs).

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Eine Liste ist eine Sammlung von Skalaren, ein Array ist eine geordnete Liste von Skalaren, auf die man mit ihrer Position in der Liste zugreift, und ein Hash ist eine ungeordnete Liste von Schlüssel/Wert-Paaren, auf die man mit dem Schlüssel zugreift.
2. Die Antworten sind:
1. `$foo` ist eine Skalarvariable.
1. `@foo` ist eine Arrayvariable.
1. `%foo` ist eine Hash-Variable.
1. `$foo{'key'}` ist der dem Schlüssel `'key'` zugeordnete Wert im Hash `%foo`.
3. Die Antworten sind:
1. a. Der Hash `%zeug` erhält vier Schlüssel/Wert-Paare: `'1'/'eins'`, `'2'/'zwei'`, `'3'/'drei'` und `'4'/'vier'`. Die `qw`-Funktion setzt die Anführungszeichen vor und nach jedem Element.
1. b. Die Schlüssel/Wert-Paare in `%zeug` werden in eine Liste zerlegt und im Array `@zahlen` gespeichert (Schlüssel, Wert, Schlüssel, Wert und so weiter)
1. c. `$foo` enthält einen Code, der den internen Zustand des Hash beschreibt.
4. Die Verwendung eines Hash in einem Listenkontext »zerlegt« in einer internen Reihenfolge den Hash in seine Schlüssel und Werte. In skalarem Kontext liefert der Hash zwei Zahlen, die Auskunft über den internen Zustand des Hash geben.
5. Man kann einen Hash nicht sortieren, weil ein Hash nicht geordnet ist. Man kann allerdings eine Liste seiner Schlüssel sortieren und über diese Liste dann auf jeden Schlüssel und den entsprechenden Wert zugreifen.
6. Die Funktion `keys` liefert Ihnen eine Liste aller Schlüssel im Hash, die Funktion `values` eine Liste aller Werte. Die Funktion `each` liefert eine Liste von je einem Schlüssel/Wert-Paar. Mit Hilfe aufeinanderfolgender Aufrufe von `each` können Sie alle Paare im Hash durchgehen.
7. `split` splittet einen String in eine Liste von Teilstrings. `Split` wird im allgemeinen zum Einlesen von Daten verwendet, die man nicht direkt einer Variablen zuweisen kann, was häufig der Fall ist, wenn man sie aus Dateien liest.

Lösungen zu den Übungen

1. Der einzige Unterschied zur ursprünglichen Version von ***stats.pl*** besteht hier in der Eingabeschleife. Sie könnten zum Beispiel die erste `while`-Schleife durch folgenden Code ersetzen, der mit Hilfe von `split` die eingegebene Zahlenreihe in eine Liste einzelner Zahlen zerlegt, sie im Array `@zahlen` speichert und dann mit `foreach` durchläuft:

```
    print 'Geben Sie Ihre Zahlen ein, alle in einer Zeile, ';
print "durch Leerzeichen getrennt: \n";
chomp ($input = <STDIN>);
@nums = split(' ', $input);
$count = @nums;
foreach $num (@nums) {
    $freq{$num}++;
    $sum += $num;
}
```

2. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
#
# Satz-Statistik
$in = '' ; # temp Input
@sent = (); # Satz
$words = 0; # Anzahl Woerter
@reversed = 90; # Satz rueckwaerts
print 'Geben Sie einen Satz ein: ';
chomp($in = <STDIN>);
print 'Anzahl der Zeichen: ';
print length $in;
@sent = split(' ', $in);
$words = @sent;
print "\nAnzahl der Woerter: $words\n";
@reversed = reverse @sent;
print "der Satz rueckwaerts: \n";
print "@reversed\n";
```

3. Das Skript könnte folgendermaßen aussehen:

```
    #!/usr/bin/perl -w
$in = '';      # temp. Eingabe
%names = ();   # Hash: Namen
@raw = ();     # temp: rohe Woerter
$fn = '';      # Vorname
while () {
    print 'Geben Sie einen Namen ein (Vor- und Nachname): ';
    chomp($in = <STDIN>);
    if ($in ne '') {
         @raw = split(' ', $in);
         if ($#raw == 1) {  # Normalfall: zwei Woerter
             $names{$raw[1]} = $raw[0];
         } else {  # den Vornamen zusammensetzen
             $fn = '';
             $i = 0;
             while($i < $#raw) {
                 $fn .= $raw[$i++] . ' ';
             }
             $names{$raw[$#raw]} = $fn;
         }
    }
    else { last; }
}
foreach $lastname (sort keys %names) {
    print "$lastname, $names{$lastname}\n";
}
```

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Bedingungen und Schleifen

Mit Hilfe von Bedingungen und Schleifen steuern Sie die Ausführung von Anweisungsblöcken in Perl-Skripten. Ohne solche Strukturen würde Ihr Perl-Skript einfach von oben nach unten durchlaufen und nacheinander alle Anweisungen abarbeiten, bis es zu Ende ist. Sie könnten nicht überprüfen, ob etwas einen bestimmten Wert hat, und dann gegebenenfalls zu einem anderen Teil des Codes abzweigen, Sie könnten nicht festlegen, ob und wie oft ein Anweisungsblock noch einmal ausgeführt wird - ohne Kontrollstrukturen wäre Perl sehr langweilig.

In dieser Lektion behandeln wir die verschiedenen Bedingungs- und Schleifenkonstrukte, die Sie für Ihre Perl-Skripts brauchen. Unsere heutigen Themen sind:

- Eine Einführung in Anweisungsblöcke
- Die Bedingungen `if`, `if...else`, `if...elsif` und `unless`...
- Die Schleifen `while`, `do...while` und `until`
- Die Schleifen `for` und `foreach`
- Die Steuerung von Schleifen mit `next`, `last`, `redo` und ***Labels***
- Die Verwendung der Spezialvariablen `$_` als Abkürzung für viele Operationen
- Mit <> aus Dateien lesen

Komplexe Anweisungen und Blöcke

Bedingungen und Schleifen werden manchmal auch ***komplexe Anweisungen*** genannt, und zwar deshalb, weil Bedingungen und Schleifen im Vergleich zu einzelnen Anweisungen, die mit einem Semikolon enden (wie zum Beispiel `$x = 5;` oder `$array[0] = "erstes";`), wie soll ich sagen ... eben komplexer sind. Der wahrscheinlich bedeutendste Unterschied zwischen einfachen und komplexen Anweisungen ist allerdings, dass letztere mit ganzen ***Blöcken*** von Perl-Code arbeiten.

Ein Block ist einfach eine in geschweifte Klammern {} gesetzte Folge von beliebigen Perl-Anweisungen. Innerhalb eines Blocks können einfache Anweisungen oder andere Blöcke stehen, also alles, was auch außerhalb des Blocks auftauchen kann. Wie Anweisungen in einem Skript werden die Anweisungen in einem Block nacheinander ausgeführt. Zum Beispiel:

```
while (Bedingung) { # Beginn des Blocks
   Anweisung;
   Anweisung;
   if (Bedingung) { # Beginn des if-Blocks
      Anweisung;
   }                          # Ende des if-Blocks
   # ... weitere Anweisungen
}                             # Ende des Blocks
```

Eine andere Eigenschaft von Blöcken ist, dass Perl hinter der letzten Anweisung im Block nicht unbedingt ein Semikolon verlangt. Es ist aber eine gute Idee, das Semikolon trotzdem zu setzen - dann müssen Sie nicht mehr darauf achten, wenn Sie später weitere Anweisungen hinzufügen.

Blöcke, die nicht an eine Bedingung oder Schleife gebunden sind, werden ***bare blocks*** oder freistehende Blöcke genannt und nur einmal ausgeführt. ***Bare blocks*** können durchaus nützlich sein, besonders wenn Sie sie mit einem ***Label*** versehen, doch jetzt werden wir uns erst einmal auf die Blöcke konzentrieren, die zu komplexen Anweisungen gehören.

Bedingungen

Mit Hilfe von Bedingungsanweisungen können Sie festlegen, welche Blöcke Perl ausführen soll, wenn ein Ausdruck einen bestimmten Wert hat. Sie setzen eine Bedingung auf, und Perl überprüft, ob diese Bedingung erfüllt ist, das heißt, ob der Bedingungsausdruck ***wahr*** ist. Wenn der Ausdruck ***wahr*** ist, führt Perl den direkt folgenden Block aus, wenn er ***falsch*** ist, überspringt Perl diesen Block und macht beim nächsten Block oder Teil des Skripts weiter. Anders als bei Schleifen wird jeder Block nur einmal ausgeführt.

if, *if...else*, und *if...elsif*

Die häufigste Formen der Bedingung sind `if` (***wenn***) und seine Varianten `if...else` (***wenn...sonst***) und `if...elsif` (***wenn...sonst wenn***). Die `if`-Anweisung sieht wie folgt aus:

```
if ( Bedingungsausdruck) {
   # Anweisungen
}
```

`Bedingungsausdruck` kann jeder beliebige Ausdruck sein; er wird in Booleschem skalaren Kontext auf seinen Wahrheitswert überprüft. Sie erinnern sich, dass alles außer `""`, `0` und `undef` als ***wahr*** angesehen wird. Ist der `Bedingungsausdruck` ***wahr***, wird der Anweisungsblock ausgeführt. Ist er ***falsch***, passiert gar nichts, und Perl macht mit der nächsten Anweisung unter dem `if`-Block weiter.

Beachten Sie, dass anders als in C oder Java auf die Bedingung (auch bei `else` und `elsif`) immer ein Block folgen muss, auch wenn er nur eine einzige Anweisung enthält. Sie müssen die geschweiften Klammern immer setzen - aber nicht unbedingt in verschiedenen Zeilen, wie ich es eben gezeigt habe. ***Wo*** die Klammern stehen, bleibt Ihrem Geschmack überlassen.

Um einen alternativen Block ausführen zu lassen, wenn die Bedingung nicht erfüllt ist, verwenden Sie `if...else`:

```
if ( Bedingungsausdruck ) {
   # Anweisungen, die ausgefuehrt werden,
   # wenn Bedingungsausdruck wahr ist
} else {
   # Anweisungen, die ausgefuehrt werden,
   # wenn Bedingungsausdruck falsch ist
}
```

Wie in anderen Sprachen auch, können Sie `if` und `else` auch ineinander verschachteln:

```
if ( Bedingungsausdruck 1 ) {
   # Anweisung 1
} else {
   if ( Bedingungsausdruck 2 ) {
      # Anweisung 2
   } else {
      if ( Bedingungsausdruck 3 ) {
         # Anweisung 3
      } else {
         # und so weiter...
      }
   }
}
```

Um Ihnen etwas Tipparbeit zu ersparen oder unübersichtlich viele Einzüge zu vermeiden, bietet Perl eine dritte Form von `if`-Bedingungen, das `elsif`, das diese Art von Operationen um einiges kompakter macht:

```
if ( Bedingungsausdruck 1 ) {
   # Anweisung 1
} elsif ( Bedingungsausdruck 2 ) {
   # Anweisung 2
} elsif ( Bedingungsausdruck 3 ) {
   # Anweisung 3
} else (
   # Anweisung fuer alle anderen Faelle
}
```

Beachten Sie, dass sobald die Bedingung eines `elsif` ***wahr*** ist, alle noch folgenden `elsif`-Blöcke übersprungen, das heißt nicht einmal ausgewertet werden. Bei verschachtelten `if...else`-Anweisungen ist es ebenso: In jeder Folge von Blöcken wird jeweils nur der erste ausgeführt, dessen Bedingung erfüllt ist.

Was ist mit Fallunterscheidungskonstrukten wie `switch` oder `case`? Perl hat von sich aus keine Anweisung für `switch` (das ist wahrscheinlich das einzige Konstrukt, für das andere Sprachen eine eigene Anweisung haben, Perl aber nicht). Allerdings gibt es verschiedene Möglichkeiten, mit vorhandenen Perl-Konstrukten eine `switch`-Anweisung zu »simulieren«. Ein paar davon werden wir im Abschnitt »Vertiefung« am Ende dieser Lektion durchgehen.

unless

Die `unless`-Anweisung ist eine Art umgekehrtes `if`. Manchmal soll eine Operation nur ausgeführt werden, wenn eine Bedingung ***nicht*** erfüllt ist - was bedeutet, dass in einer normalen `if...else`-Anweisung all das ***gute*** Zeug in den `else`-Teil wandern würde wie hier:

```
if ( Bedingungsausdruck ) {
   # mache nichts
} else {
   # mache das hier
}
```

Das ist nicht unbedingt die optimale Lösung. Sie könnten natürlich den Bedingungsausdruck negieren (mit `not` oder `!`):

```
if (not Bedingungsausdruck ) {
   # mache das hier
}
```

So müßten Sie es in anderen Sprachen machen und Ihre Denkweise an die Syntax anpassen. Perl aber zieht es vor, wenn Sie so denken, wie Sie zu denken gewohnt sind, und bietet Ihnen deshalb eine Alternative. Wenn Sie denken: »Mache das und das, außer unter dieser Bedingung«, also eine Operation nur ausführen möchten, wenn diese Bedingung nicht erfüllt ist, nehmen Sie einfach `unless` (vom Englischen ***unless, wenn nicht*** oder ***außer wenn***):

```
unless ( Bedingungsausdruck ) {
   # mache das hier
}
```

Mit diesem `unless` wird der Block nur dann ausgeführt, wenn der `Bedingungsausdruck` ***falsch*** ist. Ist er aber ***wahr***, wird der Block übersprungen. Sie können einem `unless` auch ein `else` hinzufügen, wenn Sie möchten (aber kein `elsif`, und so etwas wie `elsunless` gibt es zum Glück auch nicht).

Der Bedingungsoperator

Manche Bedingungen sind so kurz, dass es viel zuviel Aufwand wäre, all diese Klammern und Worte auf sie zu verschwenden. Manchmal ist es sinnvoll, eine Bedingung in einen anderen Ausdruck zu betten (was mit `if` oder `unless` nicht möglich ist, weil diese keine Werte zurückliefern). Verwenden Sie in solchen Situationen den Bedingungsoperator ?...:.... Wie bei `if...else` brauchen Sie einen Bedingungsausdruck und zwei Angaben - was Perl bei erfüllter und bei nicht erfüllter Bedingung tun soll. Das Ganze schreiben Sie dann in folgender Form:

```
Bedingungsausdruck ? wert_wenn_wahr : wert_wenn_falsch;
```

Hier wird der Bedingungsausdruck auf seine Wahrheit überprüft, genau wie bei `if`, und wenn er ***wahr*** ist, dann wird der Ausdruck `wert_wenn_wahr` ausgewertet (und der Wert zurückgegeben), anderenfalls wird wert_wenn_falsch ausgewertet und zurückgegeben. Anders als bei `if` und `unless` sind wert_wenn_wahr und wert_wenn_falsch einzelne Ausdrücke, keine Blöcke. Zum Beispiel können Sie mit der folgenden Zeile ganz schnell den höheren von zwei Werten herausfinden:

```
$max = $x > $y ? $x : $y;
```

Dieser Ausdruck sieht nach, ob der Wert von `$x` größer ist als der von `$y`. Wenn ja, gibt er `$x` zurück. Wenn `$x` kleiner oder gleich `$y` ist, gibt er `$y` zurück. Der zurückgegebene Wert wird dann der Variablen `$max` zugewiesen. Mit `if` und `else` würde der gleiche Vorgang so aussehen:

```
if ($x > $y) {
   $max = $x;
} else {
   $max = $y;
}
```

*Der Bedingungsoperator wird manchmal auch **triadischer** Operator genannt, weil er drei Operanden hat (**monadische** Operatoren haben einen, **dyadische** zwei und **triadische** drei Operanden).*

Kontrollflußsteuerung mit logischen Operatoren

Am Tag 2 habe ich Ihnen Perls logische Operatoren `&&`, `||`, `and` und `or` erklärt und kurz erwähnt, dass Sie damit Bedingungsanweisungen konstruieren können. Lassen Sie uns diese Operatoren hier noch einmal betrachten, damit Sie ein Gefühl dafür bekommen, wie sie arbeiten. Nehmen Sie den folgenden Ausdruck:

```
$wert = $dies || $das;
```

Um zu verstehen, wie dieser Ausdruck funktioniert, müssen Sie sich an zwei Eigenschaften von logischen Operatoren erinnern: dass sie »kurzschließen« und dass sie den Wert des zuletzt ausgewerteten Ausdrucks zurückgeben. So wird in dem obigen Beispiel zuerst `$dies`, der linke Operand von `||`, überprüft, und dann gibt es drei Möglichkeiten:

- Wenn `$dies` irgendeinen anderen Wert als `0` oder `""` hat, ist der Ausdruck ***wahr***. Weil der `||`-Operator kurzschließt, wird `$das` gar nicht mehr ausgewertet. Der Operator gibt das Ergebnis der letzten Auswertung zurück - den Wert von `$dies`, der dann der Variablen `$wert` zugewiesen wird.
- Hat `$dies` den Wert `0` oder `""`, ist der Ausdruck ***falsch***. Perl wertet dann `$das` aus. Wenn `$das` ***wahr*** ist, ist der gesamte Ausdruck ***wahr,*** und der Operator gibt den zuletzt ermittelten Wert zurück - `$wert` erhält den Wert von `$das`.
- Wenn alle beide, `$dies` und `$das`, falsch sind, gibt der Operator ***falsch*** zurück, und `$wert` erhält einen `0`- oder `""`-Wert.

Mit `if...else` könnten Sie diese Anweisung wie folgt schreiben:

```
if ($dies) { $wert = $dies; }
else {$wert = $das; }
```

Mit dem Bedingungsoperator könnten Sie schreiben:

```
$wert = $dies ? $dies : $das
```

Aber beide brauchen mehr Platz und sind schwieriger zu begreifen - zumindest schwieriger als der logische Ausdruck, der sich fast wie Klartext liest: dies oder das. Und schon sind Sie fertig.

Wie ich am Tag 2 schon erwähnt habe, werden Ihnen solche für eine Entscheidung eingesetzten logischen Operatoren am häufigsten beim Öffnen von Dateien begegnen:

```
open(DATEI, "Dateiname") or die "Kann Datei nicht oeffnen\n";
```

Wenn die Funktion `open` eine Datei erfolgreich geöffnet hat, gibt sie ***wahr*** zurück, und deswegen wird der Teil nach dem `or` übersprungen, die Funktion `die` (»***stirb«***) also gar nicht erst ausgeführt. Mehr hierzu am Tag 15.

while-Schleifen

Mit den verschiedenen `if`-Anweisungen in Perl steuert man den Programmfluß, indem man (je nachdem, ob eine Bedingung erfüllt ist oder nicht) zu verschiedenen Teilen des Skripts abzweigt. Die zweite Steuerungsmöglichkeit sind Schleifen - die ein und denselben Anweisungsblock immer wieder ausführen und erst dann aufhören, wenn eine bestimmte Bedingung erfüllt ist. Perl hat zwei generelle Arten von Schleifen, die beide ungefähr dasselbe machen: `while`-Schleifen laufen solange, bis eine Bedingung erfüllt ist, bei `for`-Schleifen ist die Anzahl der Durchläufe von Anfang an festgelegt. Man kann jede `while`-Schleife auch als `for`-Schleife formulieren und umgekehrt, doch paßt je nach Situation meist die eine besser als die andere.

Wir beginnen mit den `while`-Schleifen. Davon hat Perl drei verschiedene: `while`, `do...while` und `until`.

while

Die Grundform einer Schleife in Perl ist die `while`-Schleife mit einem Bedingungsausdruck und einem Anweisungsblock:

```
while ( Bedingungsausdruck ) {
   # zu wiederholende Anweisungen
}
```

In der `while`-Schleife wird der Bedingungsausdruck ausgewertet, und wenn er ***wahr*** ist, werden die Anweisungen im Block ausgeführt. Danach wird der Bedingungsausdruck wieder ausgewertet, und wenn er immer noch ***wahr*** ist, wird der Anweisungsblock wieder ausgeführt. Dieser Vorgang wiederholt sich so lange, bis der Bedingungsausdruck ***falsch*** ergibt. Als Beispiel zeige ich Ihnen noch einmal die `while`- Schleife aus dem Krümelmonster-Skript, das Sie bereits am ersten Tag gesehen haben:

```
while ( $kekse ne "KEKSE") {
   print 'ich will KEKSE: ';
   chomp($kekse = <STDIN>);
}
```

Hier wiederholt sich der Eingabevorgang (die `print`- und die `<STDIN>`-Anweisung) so lange, bis die Eingabe den String `"KEKSE"` enthält. Sie könnten diese Schleifenanweisung auch so lesen: »Solange der Wert von `$kekse` nicht gleich dem String `"KEKSE"` ist, tue folgendes.«

Hier ein anderes Beispiel von Tag 4, das mit einer temporären Variablen `$i` für den Array-Index ein Array durchläuft:

```
$i = 0;
while ($i <= $#array) {
   print $array[$i++], "\n";
}
```

In diesem Fall ist die Bedingung, dass `$i` kleiner oder gleich dem größten Array-Index ist. Innerhalb des Schleifenblocks geben wir das aktuelle Array-Element aus und inkrementieren `$i`. So bestimmen wir, wie oft die Schleife durchlaufen werden soll: solange wie `$i` kleiner oder gleich dem größten Index in `@array` ist, also `$#array + 1`, (weil wir ja bei 0 anfangen).

Beachten Sie beim Schreiben von `while`-Schleifen, dass innerhalb der Schleife etwas passieren muss, dass sie näher an ihr Ende bringt. Wenn Sie hier zum Beispiel vergessen, `$i` zu erhöhen, wird die Schleifenbedingung nie erfüllt, und die Schleife läuft und läuft und läuft ohne Ende.

Schleifen, die nicht enden, werden Endlosschleifen genannt und können - wenn man sie bewußt verwendet - durchaus von Nutzen sein. Zum Beispiel ist eine `while`- Schleife ohne Bedingung eine absichtlich endlose Schleife. Sie haben ein paar davon in den bisherigen Beispielen gesehen. Diese hier ist aus den Statistikskripten:

```
while () {
   print 'Bitte eine Zahl eingeben: ';
   chomp ($input = <STDIN>);
   if ($input ne '') {
      $zahlen[$count] = $input;
      $count++;
```

```
      $sum += $input;
   }
  else { last; }
}
```

Diese Schleife liest bei jedem Durchlauf eine Zeile von der Standardeingabe und kann niemals aufgrund einer Schleifenbedingung enden, da es keine Bedingung gibt. Doch innerhalb der Schleife überprüfen wir die Eingabe mit `if...else`. Wenn `$input` ein leerer String ist, unsere `if`-Bedingung also nicht erfüllt ist, wird der `else`-Block ausgeführt, der nur aus dem Schleifensteuerbefehl `last` besteht. Dieses `last` bricht die Schleife ab und geht zum nächsten Teil des Skripts weiter. Es gibt in Perl drei solcher Schleifensteuerbefehle: `last`, `next` und `redo`, die wir später in diesem Kapitel, im Abschnitt »Schleifen steuern«, genauer betrachten werden.

Ich hätte diese Schleife auch so schreiben können, dass das `while` eine richtige Bedingung hätte und zum richtigen Zeitpunkt abbrechen würde. In diesem speziellen Beispiel fand ich es aber einfacher, die Schleife eben auf diese Art zu konstruieren. Perl zwingt Ihnen auch bei Schleifen oder Bedingungen kein bestimmtes Denkmuster auf, Sie können Ihr Skript so bauen, wie Sie es für die jeweilige Aufgabe am besten halten.

until

Genau wie `unless` das Gegenteil von `if` ist, ist `until` das Gegenteil von `while`. `Until` sieht genauso aus wie `while`, mit einem Bedingungsausdruck und einem Block:

```
until ( Bedingungsausdruck ) {
   #  Anweisungen
}
```

Der einzige Unterschied ist die Bedeutung der Bedingung - mit `while` wird die Schleife so lange durchlaufen, wie die der Bedingungsausdruck ***wahr*** ist. Mit `until` wird sie so lange ausgeführt, wie der Bedingungsausdruck ***falsch*** ist. ***Until*** ist englisch und heißt ***bis*** - »Bis diese Bedingung wahr ist, tue das hier.«

do

Die dritte Form von `while`-Schleifen ist `do`. Bei `while` und `until` wird die Schleifenbedingung ausgewertet, ***bevor*** der Block ausgeführt wird - wenn der Bedingungsausdruck also von Anfang an ***falsch*** (oder bei `unless` ***wahr***) ist, bricht die Schleife sofort ab und macht gar nichts. Manchmal aber möchten Sie einen Anweisungsblock zunächst einmal ausführen und erst danach entscheiden, wie es weitergeht. Hierfür gibt es `do`. `do`-Schleifen haben einen anderen Aufbau als `while`- und `until`-Schleifen und sehen etwa wie folgt aus (beachten Sie das Semikolon am Ende, bei `do` ist es Pflicht):

```
do {
   # Schleifenblock
} while (Bedingungsausdruck);
```

Dasselbe geht auch mit `until`:

```
do {
   # Schleifenblock
} until (Bedingungsausdruck);
```

Mit diesen beiden Anweisungen werden die Blöcke ***vor*** der Auswertung des Bedingungsausdrucks ausgeführt. Selbst wenn der Bedingungsausdruck ***falsch*** (bei `while`) oder ***wahr*** (bei `until`) ist, wird der Anweisungsblock mindestens einmal ausgeführt.

*In Wirklichkeit ist `do` eine Funktion, die hier nur so tut, als wäre sie eine Schleife (deshalb brauchen Sie auch das Semikolon). Meistens verhält `do` sich genau wie eine Schleife, nur wenn Sie mit Schleifensteuerbefehlen (wie `last` oder `next`) oder mit **Labels** (Sprungmarken) arbeiten wollen, müssen Sie statt `do` eine »echte« `while`- oder `until`-Schleife verwenden. Schleifensteuerbefehle und Labels besprechen wir noch in dieser Lektion.*

Ein Beispiel: Zahlen raten

In diesem Beispiel spielen wir ein kleines Spiel. Perl bittet Sie um eine Zahl, »zieht« dann eine Zufallszahl zwischen 1 und Ihrer Zahl, und Sie sollen raten, welche es gezogen hat, zum Beispiel:

```
% zahlraten.pl
Geben Sie die hoechste Zahl ein: 50
  Ihr Tipp? (eine Zahl zwischen 1 und 50): 25
Zu hoch!
  Ihr Tipp? (eine Zahl zwischen 1 und 50): 10
Zu niedrig!
  Ihr Tipp? (eine Zahl zwischen 1 und 50): 17
Zu hoch!
  Ihr Tipp? (eine Zahl zwischen 1 und 50): 13
Zu hoch!
  Ihr Tipp? (eine Zahl zwischen 1 und 50): 12
Richtig!
Gratuliere!  Sie haben mit 5 Versuchen die richtige Zahl erraten.
%
```

Das Skript dazu arbeitet mit zwei endlosen `while`-Schleifen, einer Menge `if`- Bedingungen und der `rand`-Funktion zur Ermittlung einer Zufallszahl. Listing 6.1 zeigt den Code.

Listing 6.1: Das Skript zahlraten.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $top = 0;   # hoechste Zahl
4:  $num = 0;  # Zufallszahl
5:  $count = 0; # zaehlt Versuche
6:  $guess = "";  # aktueller Tipp
7:
8:  while () {
9:      print 'Geben Sie die hoechste Zahl ein: ';
10:     chomp($top = <STDIN>);
11:     if ($top == 0 || $top eq '0') {
12:         print "Das ist keine gute Zahl.\n";
13:     }
14:     else { last; }
15: }
16:
17: srand;
18: $num = int(rand $top) + 1;
19:
20: while () {
21:     print " Ihr Tipp? (eine Zahl zwischen 1 und $top): ";
22:     chomp($guess = <STDIN>);
23:     if ($guess == 0 || $guess eq '0') {
24:         print "Das ist keine gute Zahl.\n";
25:     } elsif ($guess < $num) {
26:         print "Zu niedrig!\n";
27:         $count++;
28:     } elsif ($guess > $num) {
29:         print "Zu hoch!\n";
30:         $count++;
31:     } else {
32:         print "\a\aRichtig!  \n";
33:         $count++;
34:         last;
35:     }
36: }
37: print "Gratuliere!  Sie haben mit $count Versuchen";
38: print " die richtige Zahl erraten.\n";
```

Dieses Skript hat vier Teile: Initialisierung, Eingabe der höchsten Zahl, Auswahl der zu ratenden Zahl und dann das Raten selbst. Den Initialisierungsteil überspringe ich diesmal ganz; Sie werden mittlerweile wissen, wie man Skalarvariablen zuweist.

In Zeile 8 bis 15 erfragen wir, wie hoch die Zufallszahl höchstens sein darf. Diese `while`-Schleife ist endlos, doch die Bedingung in Zeile 11 soll sicherstellen, dass Sie nicht 0 eingeben. Tun Sie es doch, wird eben nicht das `last` im `else`-Block, sondern die Eingabeschleife von vorn ausgeführt. Denken Sie daran, dass Perl bei der Umwandlung von Strings in Zahlen einen String in 0 konvertiert, wenn es keine numerischen Daten in ihm findet. Deswegen fängt diese Bedingung sowohl eine 0 als auch alle anderen nichtnumerischen Eingaben ab. Perl-Warnungen würden sich bei der Eingabe eines Strings trotzdem über »`not a number`« (»keine Zahl«) beschweren. Wenn wir uns in ein paar Tagen mit Mustervergleichen (Pattern Matching) befassen, werden Sie einen besseren Weg kennenlernen, diese Warnungen zu umgehen.

Jedenfalls generieren wir, sobald wir eine brauchbare Zahl haben, in Zeile 17 und 18 die geheime Zahl, und zwar mit den eingebauten Funktionen `srand` und `rand`. Ohne Argumente setzt die `srand`-Funktion den Zufallsgenerator auf die aktuelle Uhrzeit, damit wir jedesmal andere Zahlen bekommen (ansonsten wäre es in der Tat ein sehr langweiliges Spiel). Die Funktion `rand` generiert dann eine Zufallszahl zwischen 0 und einem Argument, hier unserem `$top` (einschließlich 0 ohne `$top`). Wir schneiden diese Zahl auf einen Integer zu, addieren `1`, damit wir keine Nullen, aber gelegentlich auch die eingegebene Höchstzahl erhalten, und speichern sie in der Variablen `$num`.

In den Zeilen 20 bis 34 wird geraten. Hier überprüfen wir mit `if...elsif`- Anweisungen drei Dinge:

- Ob der Vorschlag eine 0 ist (Zeile 23); in diesem Fall geben wir eine Warnung aus.
- Ob der Vorschlag kleiner als die geheime Zahl ist; in diesem Fall geben wir »Zu niedrig!« aus.
- Ob der Vorschlag größer als die geheime Zahl ist; in diesem Fall geben wir »Zu hoch!« aus.

Wenn der Vorschlag weder zu niedrig noch zu hoch, aber eine gültige Zahl ist, dann muss es die richtige Zahl sein - also piepen wir zweimal (okay, Sie vielleicht nicht, aber die Escape-Sequenz `\a` gibt einen Piepton aus), Perl schreibt einen Glückwünsch auf den Bildschirm und hört auf.

Während der Benutzer Zahlen rät, merken wir uns die Anzahl der Versuche in einer `$count`-Variablen. Wir wollen aber nur gültige Versuche zählen, also inkrementieren wir `$count` nur in den drei Fällen gültiger Zahlen (zu niedrig, zu hoch, richtig). `$count` wird dann mit der Glückwunschmeldung ausgegeben.

for-Schleifen

Geht es um die Wiederholung eines Anweisungsblocks, bietet `while` als »Mutter aller Schleifen« immer einen Weg - sie führt einen Block immer wieder aus, und zwar so lange (bzw. bis) die Schleifenbedingung erfüllt ist. Die zweite Art von Schleife, die `for`- Schleife, löst das gleiche Problem auf etwas andere Art und Weise. Bei `for`-Schleifen wird der Anweisungsblock n-mal hintereinander ausgeführt, dann wird die Schleife beendet.

Sie könnten jede `while`-Schleife auch als `for`-Schleife formulieren und umgekehrt. Aber für manche Aufgaben bietet sich die eine Form eher an als die andere.

Perl kennt zwei `for`-Schleifen: eine allgemeine C-ähnliche `for`-Schleife und eine dem Shell-Skripting entliehene `foreach`-Schleife, die einen Block »für jedes« (***for each***) Element einer Liste wiederholt.

for

Die `for`-Schleife ist in Perl dieselbe wie in C. Sie setzen eine Lauf- oder Zählervariable (zum Beispiel `$i`) auf einen beliebigen Startwert, überprüfen eine Bedingung, führen den Schleifenblock aus, wenn diese erfüllt ist, ändern dann den Wert der Laufvariablen, überprüfen wieder die Bedingung, durchlaufen die Schleife gegebenenfalls von neuem, ändern wieder die Laufvariable - und so weiter:

```
for ( Startwert; Bedingungsausdruck; Aenderung ) {
   # Anweisungen
}
```

In diesem Beispiel ist `Startwert` der Ausdruck zum Initialisieren der Laufvariablen, der `Bedingungsausdruck` entscheidet, ob die Iteration weitergeht, und `Aenderung` bestimmt, wie der Wert der Laufvariablen nach jedem Schleifendurchlauf geändert wird. Beim ersten Durchlauf wird der Zähler initialisiert, die Bedingung ausgewertet und, wenn sie ***wahr*** liefert, der Anweisungsblock ausgeführt. Beim zweiten Durchlauf wird die

Änderungsanweisung ausgeführt, die Bedingung wieder überprüft und wenn sie immer noch ***wahr*** ist, der Block wieder ausgeführt. So geht es immer weiter, bis der Bedingungsausdruck ein ***falsch*** zurückgibt.

Für fünf Schleifendurchläufe könnten Sie zum Beispiel eine `for`-Schleife wie diese verwenden:

```
for ( $i = 1; $i <= 5; $i++) {
   print "Durchlauf $i\n";
}
```

Dieser Code-Schnipsel produziert folgende Ausgabe:

```
Durchlauf 1
Durchlauf 2
Durchlauf 3
Durchlauf 4
Durchlauf 5
```

Sie könnten diese Schleife selbstverständlich auch als `while`-Schleife schreiben:

```
$i = 1;
while ($i <= 5) {
   print "Durchlauf $i\n";
   $i++;
}
```

Beide Schleifen würden fünfmal durchlaufen, doch die `for`-Schleife ist kompakter und übersichtlicher. Sie listet sozusagen fein säuberlich auf, wo sie anfängt, wo sie aufhört und welche Schritte sie dazwischen unternimmt. Für manche Aufgaben eignet sie sich besser als eine `while`-Schleife - beispielsweise, wenn Sie einen begrenzten Satz von Werten durchgehen, etwa zum Bearbeiten aller Elemente in einem Array.

Sie können die Initialisierung, den Bedingungsausdruck oder die Änderungsanweisung oder alle drei im oberen Teil der `for`-Schleife weglassen, nicht jedoch die Semikola. Eine endlose `for`-Schleife könnte zum Beispiel folgendermaßen aussehen:

```
for ($i = 0; ; $i++) {
   # Anweisungen
}
```

Oder einfach so:

```
for (;;) {
   # Anweisungen
}
```

Die erste Schleife initialisiert den Schleifenzähler `$i` und erhöht ihn nach jedem Durchlauf um 1. Die zweite Schleife hat nicht einmal eine Zählervariable, sie wird einfach nur durchlaufen. Um aus diesen Schleifen herauszukommen, müssen Sie die Abbruchbedingung innerhalb der Blöcke festlegen.

Sie wollen die Schleife über mehrere Zähler laufen lassen? Setzen Sie einfach Kommas zwischen Ihre Zählerausdrücke:

```
for ($i=0, $j=1; $i < 10, j$ < $total; $i++, $j++ ) {
   # Anweisungen
}
```

In diesem Fall wird die Schleife abgebrochen, wenn einer der beiden Bedingungsausdrücke ***wahr*** zurückgibt.

foreach

Die `for`-Schleife bietet sich an, wenn Sie auf einen konkreten Endpunkt überprüfen können - zum Beispiel den höchsten Index in einem Array oder eine bestimmte Zahl. `foreach` ist eine Kurzversion von `for` ohne expliziten Zähler. `foreach` durchläuft eine Liste und führt den Anweisungsblock so viele Male aus, wie diese Liste Elemente

hat.

Sie kennen `foreach` bereits - gestern haben wir uns damit durch Listen von Hash- Schlüsseln gearbeitet. Hier nun die ausführliche »Gebrauchsanweisung« für `foreach`: Sie übergeben der `foreach`-Schleife eine Liste - zum Beispiel eine Liste aller Schlüssel in einem Hash oder einen Bereich. `foreach` durchläuft die Liste, setzt dabei eine temporäre Variable nacheinander auf den Wert jedes Elements und führt für jedes Element in der Liste den Anweisungsblock aus.

Hier ein simples Äquivalent zur `for`-Schleife von vorhin, die den Zähler fünfmal ausgegeben hat:

```
foreach $zahl (1 .. 5) {
   print "Durchlauf $zahl\n";
}
```

Hier erstellt der Bereichsoperator `..` eine Liste der Zahlen 1 bis 5, und die `foreach`- Schleife arbeitet sich durch diese Liste, wobei sie die Zahlen nacheinander der Variablen `$zahl` zuweist.

`Foreach`-Schleifen eignen sich außerordentlich gut zum Durchlaufen von Listen, zum Beispiel um die Schlüssel und Werte eines Hash auszugeben, wie Sie gestern gelernt haben:

```
foreach $key (sort keys %hashname) {
   print "Schlüssel: $key Wert: $hashname{$key}\n";
}
```

Wie Sie sehen, habe ich die `$key`-Variablen hier nicht initialisiert (auch die `$zahl`- Variable im vorangehenden Beispiel wurde nicht initialisiert). Das ist nicht nötig, weil die Variable zwischen `foreach` und der Liste eine lokale Variable innerhalb der Schleife ist - das heißt, vor und nach der Schleife existiert sie gar nicht. Falls sie doch schon existiert, Sie also vor der Schleife eine gleichnamige Variable verwenden, benutzt `foreach` sie nur vorübergehend und stellt ihren ursprünglichen Wert wieder her, sobald die Schleife verlassen wird. Sie können sich die `foreach`-Variable als eine Art »Kritzel«-Variable vorstellen, die einzig und allein dem kurzen Speichern von Elementen dient und beim Schleifenende weggeworfen wird.

Wenn Sie die Werte der Listenelemente gar nicht brauchen, sondern lediglich die Liste durchgehen möchten, können Sie diese Variable auch einfach weglassen.

Schleifen steuern

In `while`- und `for`-Schleifen stehen die Bedingungen, unter denen die Schleife durchlaufen bzw. abgebrochen werden soll, ganz am Anfang. In vielen Fällen ist das auch genau das Richtige. Wenn Sie aber mit komplexeren Schleifen arbeiten oder mit Endlosschleifen spielen, wie ich es in einigen der bisherigen Beispiele getan habe, möchten Sie vielleicht auch mitten in der Schleife entscheiden können, wie es von diesem Punkt aus weitergehen soll. Und natürlich hat Perl da etwas für Sie: Schleifensteuerbefehle.

> *Wie ich schon erwähnt habe, können Sie in `do`-Schleifen **keine** Schleifensteuerbefehle oder Labels (Sprungmarken) einsetzen. Dafür müßten Sie die `do`-Schleife zu einer `while`-, `until`-, `for`- oder `foreach`-Schleife umschreiben.*

Mit Schleifensteuerbefehlen steuern Sie den Ablauf einer Schleife. Zwei von ihnen haben Sie bereits gesehen: `next` und `last`, um die Schleife neu zu starten oder sie komplett abzubrechen. Außerdem gibt es in Perl `redo` und Labels, mit denen Sie Ihren Schleifen einen Namen geben können.

last, *next* und *redo*

Die Grundbausteine der Schleifensteuerung sind die Schlüsselwörter `last`, `next` und `redo`. Sobald eins von ihnen in einer `while`- oder `for`-Schleife auftritt, unterbricht Perl die normale Schleifenausführung.

Sie können jeden dieser drei Befehle mit einem Label oder für sich allein verwenden. Mit einem Label beziehen sie sich auf die mit diesem Label versehene Schleife ohne Label auf die innerste Schleife (mehr über Labels später). Im einzelnen bewirken sie folgendes:

- Mit `last` wird die Schleife sofort abgebrochen und verlassen (wie bei `break` in C). Der nächste Teil des Skripts wird ausgeführt.
- Mit `next` wird der aktuelle Durchlauf abgebrochen, es wird zurück zum Schleifenanfang gesprungen, und der nächste Durchlauf wird mit der Überprüfung der Bedingung gestartet. Die Anweisung entspricht dem `continue`- Befehl in C. Das Schlüsselwort `next` ist ein bequemer Weg, unter bestimmten Bedingungen den Code im Rest des Anweisungsblocks zu überspringen.
- bei `redo` wird der aktuelle Schleifendurchlauf abgebrochen und wieder an den Anfang gegangen. Der Unterschied zwischen `redo` und `next` ist, dass `next` den Bedingungsausdruck am Schleifenanfang noch einmal auswertet (und bei `for`- Schleifen die Änderungsanweisung ausführt). `Redo` hingegen startet erst am Anfang des Blocks neu, ohne den Bedingungsausdruck am Schleifenanfang noch einmal auszuwerten. Sie können sich den Unterschied so vorstellen, dass `redo` denselben Durchlauf noch einmal wiederholt, während `next` zum nächsten springt (und das ist tatsächlich exakt das, was passiert).

Schauen wir uns noch einmal das Zahlenratespiel an. Wir könnten die zweite `while`- Schleife auch so schreiben:

```
while () {
    print "Ihr Tipp? (eine Zahl zwischen 1 und $top): ";
    chomp($guess = <STDIN>);
    if ($guess == 0 || $guess eq '0') { # wenn Eingabe 0 oder String
        print "Das ist keine gute Zahl.\n";
        next;
    }
    if ($guess < $num)  { # wenn Eingabe zu niedrig
        print "Zu niedrig!\n";
        $count++;
        next;
    }
    if ($guess > $num) { # wenn Eingabe zu hoch
        print "Zu hoch!\n";
        $count++;
        next;
    }
    $count++;
    last;
}
```

Weil diese `while`-Schleife endlos ist, müssen wir mit Schleifensteuerbefehlen innerhalb des Schleifenkörpers festlegen, wann sie abgebrochen werden soll - in unserem Fall, wenn die richtige Zahl eingegeben wurde. In dem Schleifenblock wird die Eingabe auf drei verschiedene Dinge überprüft: ob sie gleich 0 ist, ob sie kleiner als die geheime Zahl ist oder ob sie größer als die geheime Zahl ist. In jeder dieser `if`-Anweisungen überspringen wir, wenn eins dieser Kriterien zutrifft, mit `next` sofort die restlichen Anweisungen im Block und gehen zurück zum Schleifenanfang (stünde dort ein Bedingungsausdruck, würden wir ihn jetzt aufs neue auswerten). Wenn die eingegebene Zahl allen drei Überprüfungen standhält, dann ist es die richtige, und wir können die Schleife mit `last` abbrechen.

Im allgemeinen sind Schleifensteuerbefehle nur notwendig, wenn Sie innerhalb der Schleife auf bestimmte Bedingungen überprüfen, die den normalen Ablauf der Schleife unterbrechen sollen, oder um aus einer Endlosschleife herauszukommen.

Labels

Schleifensteuerbefehle ohne Labels beziehen sich auf die innerste Schleife, das heißt, sie unterbrechen die sie unmittelbar umschließende Schleife. Manchmal haben Sie aber mehrere ineinandergeschachtelte Schleifen und möchten unter einer bestimmten Bedingung aus mehreren Schleifen aussteigen. Aus diesem Grund können Sie Schleifen mit Labels kennzeichnen und dann mit `last`, `next` und `redo` zu diesen Schleifen springen.

Labels stehen am Anfang der Schleife und werden vereinbarungsgemäß großgeschrieben, damit man sie nicht mit Perl-Schlüsselwörtern durcheinanderbringt. Verwenden Sie einen Doppelpunkt, um das Label von der Schleife zu trennen:

```
LABEL: while (Bedingungsausdruck) {
   #...
}
```

Innerhalb der Schleife verwenden Sie dann `last`, `next` oder `redo` mit dem Label:

```
LABEL: while (Bedingung_1) {
   #...
   while (Bedingung_2) {
      # ...
      if (noch_eine_Bedingung) {
         last LABEL;
      }
   }
}
```

Ihre Labels können Sie nennen, wie Sie wollen, mit zwei Ausnahmen: `BEGIN` und `END` sind für die Paketkonstruktion und -dekonstruktion reserviert. Mit Paketen (englisch Packages) werden wir uns in diesem Buch nicht befassen; lediglich am Tag 13 werde ich Ihnen erklären, was ein Paket überhaupt ist. Wenn Sie bereit sind, in fortgeschrittene Perl-Konzepte einzusteigen, schauen Sie in der ***perlmod***-Manpage nach weitergehenden Informationen über Pakete und Module.

Hier ein einfaches Beispiel ohne Labels. Die äußere `while`-Schleife überprüft, ob die Variable `$exit` ungleich dem String `'n'` ist. Die innere `while`-Schleife ist eine Endlosschleife.

```
while ($exit ne 'n') {
   while () {
        print 'Geben Sie eine Zahl ein: ';
        chomp($zahl = <STDIN>);
        unless ($zahl eq '0' ) { # die Zahl 0 ist erlaubt
           if ($zahl == 0 ) {    # wenn Eingabe ein String
              print "Keine Strings.  Zahlen von 0 bis 9, bitte.\n";
              next;
           }
        }
        # andere Anweisungen
        last;
    }
    print 'Eine weitere Zahl ausprobieren (j/n)?: ';
    chomp ($exit = <STDIN>);
}
```

In diesem Beispiel verlassen `next` und `last` die sie unmittelbar einschließende Schleife - also die innere, endlose Schleife, in der Sie die Zahl eingeben. Die äußere Schleife wird nur in einem einzigen Fall verlassen: wenn Sie `$exit` am Ende der äußeren Schleife auf `'n'` setzen.

Sagen wir, Sie wollten diesem Skript noch die Möglichkeit hinzufügen, dass man mit der Eingabe von `exit` alle Schleifen und das Skript beenden kann. Dazu würden Sie vor die äußere Schleife ein Label setzen:

```
AUSSEN: while ($exit ne 'n') {
   # etc.
}
```

In der inneren Schleife würden Sie dann `last` mit dem Label verwenden:

```
AUSSEN: while ($exit ne 'n') {
   while () {
        print 'Geben Sie eine Zahl ein (Beenden mit exit): ';
        chomp($zahl = <STDIN>);
        if ($zahl eq "exit" ) { # schluss, aus, vorbei!
           last AUSSEN;
        }
        # etc.
    }
 #  mehr...
}
```

Wenn der Benutzer hier am 'Geben Sie eine Zahl ein'- Prompt »exit« eintippt, bezieht sich der last-Befehl auf die Schleife mit dem Label AUSSEN - beendet also in diesem Fall die äußere Schleife.

Beachten Sie, dass Schleifensteuerbefehle sich auf Schleifen beziehen und nicht auf eine bestimmte Position im Skript (wie etwa goto, falls Ihnen das etwas sagt). Am besten stellt man sich die Labels weniger als Sprungmarken, als vielmehr als Bezeichner für Schleifen, die man anspringen kann, vor. Wenn Sie mit einem Schleifensteuerbefehl aus einer Schleife herausspringen, springen Sie in Wirklichkeit nicht zu dem Label, sondern zur Anweisung direkt hinter der Schleife mit dem Label.

Die Variable *$_*

Glückwunsch! Sie wissen jetzt so gut wie alles, was Sie über Schleifen in Perl wissen müssen (das übrige erfahren Sie im Abschnitt »Vertiefung«). Weil wir in dieser Lektion noch etwas Platz haben, beende ich sie heute mit zwei allgemeinen Themen: der speziellen Variablen $_ und einer einfachen Methode, Dateien zu lesen. Beides werden wir in den weiteren Kapiteln dieses Buches häufig gebrauchen.

Fangen wir mit der $_-Variablen an. Diese spezielle Variable können Sie sich als einen Standardplatzhalter für skalare Werte vorstellen. Viele Perl-Konstrukte verwenden $_, wenn Sie keine eigene Skalarvariable angeben. Sie können das ausnutzen und Ihre Skripts kürzer und effizienter machen - manchmal allerdings auf Kosten der Lesbarkeit.

Betrachten wir ein Beispiel: foreach. Sie erinnern sich, dass die foreach-Schleife mit einer temporären Variablen arbeitet, in der die Listenelemente nacheinander zwischengespeichert werden. Wenn Sie keine temporäre Variable angeben, speichert Perl die Werte in $_ . Innerhalb der Schleife können Sie sich dann auf $_ beziehen, um an diesen Wert heranzukommen. Statt

```
foreach $key (sort keys %hash) {
   print "Schlüssel: $key Wert: $hash{$key}\n";
}
```

können Sie genausogut schreiben:

```
foreach (sort keys %hash) {
   print "Schlüssel: $_ Wert: $hash{$_}\n";
}
```

Auch viele Funktionen arbeiten mit dem Wert von $_ , wenn Sie ihnen keine Argumente übergeben - print und chomp zum Beispiel. Wundern Sie sich also nicht über Anweisungen wie:

```
print;
```

Fügen Sie einfach in Gedanken das $_ hinzu:

```
print $_;
```

Wir werden uns die $_-Variable noch genauer ansehen - im nächsten Abschnitt und am Tag 9.

Mit *<>* und *while*-Schleifen aus Dateien lesen

In den Beispielen der letzten Tage haben wir Eingabedaten mit <STDIN> von der Tastatur gelesen. Wir haben den Datei-Handle für die Standardeingabe sowohl in skalarem als auch in Listenkontext verwendet, und der Unterschied zwischen <STDIN> in skalarem Kontext (Zeile für Zeile) und Listenkontext (bis zum Dateiendezeichen) sollte Ihnen halbwegs klar sein.

Bestehen Ihre Eingabedaten allerdings aus mehr als nur ein paar Zeilen, ist es äußerst mühsam, sie alle über die Tastatur einzutippen. Das Statistikskript von gestern ist ein gutes Beispiel dafür - es dauert eine ganze Weile, alle Zahlen einzugeben, und wollten wir auch nur einen einzigen Wert hinzufügen, müßten wir komplett von vorne anfangen.

Viel praktischer wäre es, die Daten in eine eigene Datei zu speichern und von dort zu lesen, wenn das Skript ausgeführt wird. Dafür gibt es in Perl zwei Möglichkeiten: Zum einen können Sie eine bestimmte Datei innerhalb Ihres Skripts öffnen und lesen. Diesem Thema habe ich eine ganze Lektion gewidmet, nämlich Tag 15. Es gibt aber noch einen schnelleren Weg, Daten aus beliebigen Daten in ein Perl-Skript einzulesen. Diese Technik, die auf dem Perl-Befehl `line` beruht, bringe ich Ihnen heute bei.

Wir haben den Zeileneingabeoperator <> bis jetzt immer in Zusammenhang mit dem Datei-Handle `<STDIN>` verwendet. Wenn Sie <> aber ohne ein Datei-Handle einsetzen, holt Perl sich den Input aus der Datei, die Sie in der Kommandozeile angeben. Ein <> ohne Datei-Handle bedeutet im Grunde nichts anderes als: »Nimm alle in der Kommandozeile genannten Dateien, öffne sie, verkette sie miteinander und lies sie, als wären sie eine einzige Datei.«

Genaugenommen entnimmt Perl die Namen der zu öffnenden und zu lesenden Dateien von einer speziellen Variablen namens `@ARGV`, einem Array mit Dateinamen oder anderen in der Kommandozeile angegebenen Werten, und Sie könnten `@ARGV` im Skript noch verändern. Aber fürs erste nehmen wir einfach an, dass `@ARGV` die in der Kommandozeile angegebenen Dateinamen enthält und <> mit diesen arbeitet. Mehr über `@ARGV` erfahren Sie an Tag 15.

Hier ein Beispiel, das die in der Kommandozeile angegebenen Dateien zeilenweise einliest und jede Zeile nacheinander ausgibt:

```
while (defined($input = <>)) {
   print "$input";
}
```

*Wenn Sie mit MacPerl arbeiten, haben Sie vielleicht gar keine Kommandozeile und wissen nicht recht, was Sie jetzt machen sollen. Aber keine Angst, es geht auch in MacPerl: Speichern Sie Ihr Skript als **Droplet** (diese Option finden Sie im **Save**-Dialog, Menü **Type**). Ist Ihr Skript als **Droplet** gespeichert, können Sie die zu lesenden Dateien per **Drag & Drop** auf das Skript-Icon ziehen - MacPerl wird starten und diese Dateien in das Skript lesen.*

Sagen wir, Sie hätten dieses Beispiel als `echodatei.pl` gespeichert und möchten die Datei `eine_Datei.txt` ausgeben. Von der Kommandozeile rufen Sie das Skript folgendermaßen auf:

```
% echodatei.pl eine_Datei.txt
```

Wenn Sie mehrere Dateien ausgeben wollen, hängen Sie einfach alle anderen Dateinamen hinten an die Kommandozeile an:

```
% echodatei.pl eine_Datei.txt noch_eine_Datei.txt Datei_3.txt
```

Perl wird all diese Dateien öffnen und eine nach der anderen ausgeben.

Gehen wir diese `while`-Schleifenbedingung noch einmal von innen nach außen durch, damit Sie verstehen, was hier vor sich geht. Das `$input = <>` wird Ihnen vertraut vorkommen; es ist ähnlich wie beim Lesen einer Zeile mit `<STDIN>` in skalarem Kontext. Vielleicht erinnern Sie sich noch, dass die `defined`-Funktion ***wahr*** oder ***falsch*** zurückgibt, je nachdem, ob ein Argument definiert ist oder nicht (das heißt, nicht den undefinierten Wert enthält). Hier verwenden wir `defined`, um die Schleife beim Dateiende abzubrechen - für jede Zeile der Datei bekommen wir einen gültigen Wert, doch am Ende der Datei liefert <> undefiniert zurück - `$input` wird ebenfalls undefiniert, die `defined`-Funktion liefert ***falsch,*** und die `while`-Schleife wird gestoppt.

Rein technisch gesehen brauchen Sie den `defined`-Teil gar nicht. Perl versteht auch so, wo das Dateiende ist, und hört automatisch auf zu lesen. Wenn Sie aber die Warnungen eingeschaltet haben, beschwert Perl sich, dass Sie nicht ausdrücklich auf das Dateiende geprüft haben. Sie können beides umgehen, die Warnung und den Aufruf von `defined`, wenn Sie `$_` als Eingabevariable nehmen (siehe unten).

Wie bei `<STDIN>` können die leeren spitzen Klammern sowohl in skalarem als auch im Listenkontext verwendet werden. In skalarem Kontext lesen Sie die Eingabedateien zeilenweise ein (wobei das Zeilenende ein Zeilenvorschub oder - auf dem Mac - ein ***Carriage Return*** ist). Im Listenkontext wird jede Zeile der Datei (bzw. der Dateien) als ein Element der Liste gespeichert.

Eine noch kürzere Version des `echodatei`-Skripts macht von der `$_`-Variablen Gebrauch. Sie können die Variable `$input` durch `$_` ersetzen und auf die temporäre Variable und die `defined`-Funktion komplett verzichten:

```
while (<>) {
   print;
}
```

Wenn die Bedingung einer `while`-Schleife nichts als einen Zeileneingabeoperator enthält, liest diese `while`-Schleife die Eingabedateien Zeile für Zeile ein, weist jede Zeile nacheinander der Variablen `$_` zu und bricht ab, wenn `<>` undefiniert ist, ohne dass Sie darauf ausdrücklich überprüfen müßten - Sie erhalten deswegen keine Fehlermeldung. Diesen Mechanismus können Sie eigentlich mit jeder Eingabequelle verwenden - es funktioniert auch mit `<STDIN>` oder einem anderen Datei-Handle.

Beachten Sie allerdings, dass es die `while`-Schleife ist - nicht der `<>`-Operator -, die die Zeilen nacheinander `$_` zuweist und auf das Dateiende überprüft. Sie können zum Beispiel nicht `do chomp(<>)` schreiben; dieser Funktionsaufruf würde die aktuelle Zeile nicht in `$_` speichern. Dieses besondere Feature haben nur `while`- und `for`-Schleifen (`for`-Schleifen werden als »verkleidete« `while`-Schleifen betrachtet).

Dieser Mechanismus ist ein sehr gebräuchlicher Weg, Daten in ein Perl-Skript einzulesen. In vielen Perl-Skripten werden Ihnen gleich am Skript-Anfang solche Schleifen begegnen, die Daten aus Dateien in ein Array oder einen Hash speichern.

Betrachten wir ein weiteres Beispiel für den Einsatz von `<>` und `$_` zum Einlesen von Daten. Gestern hatten wir ein Skript, das den Benutzer um die Eingabe von ein paar Namen bittet und diese dann in ein Array speichert. Die Eingabeschleife dieses Skripts sah wie folgt aus:

```
while () {
     print 'Geben Sie einen Namen ein (Vor- und Nachname): ';
     chomp($in = <STDIN>);
     if ($in ne '') {
         ($fn, $ln) = split(' ', $in);
         $namen{$ln} = $fn;
     }
     else { last; }
}
```

Jetzt möchten wir diese Namen aus einer Datei einlesen. Zu diesem Zweck streichen wir die Eingabeaufforderung, den Aufruf von `<STDIN>` und die Überprüfung auf eine leere Eingabe:

```
while (defined($in = <>)) {
    chomp($in);
    ($fn, $ln) = split(" ", $in);
    $namen{$ln} = $fn;
}
```

Mit Hilfe der `$_`-Variablen können wir das Skript noch weiter kürzen:

```
while (<>) {
    chomp;
    ($fn, $ln) = split(' ');
    $namen{$ln} = $fn;
}
```

Obwohl sie kein einziges Mal explizit auftaucht, wird die $_-Variable hier sehr viel verwendet: Die while-Schleife weist ihr eine Zeile aus der Input-Datei zu und überprüft, ob dieser Wert definiert ist. Dann greift chomp auf diesen Wert zu und schneidet den Zeilenvorschub ab. Und split sieht in $_ nach, was es denn splitten soll. Derartige Abkürzungen sind in Perl-Skripten sehr häufig.

Wir könnten sogar die split-Funktion noch weiter abkürzen und das leere Argument weglassen. Ohne Argumente trennt split den String in $_ an den Leerstellen:

```
($fn, $ln) = split;
```

Vertiefung

Wie in den bisherigen Lektionen habe ich Ihnen auch zu Bedingungen und Schleifen noch nicht alles gesagt. Zum Teil möchte ich dies jetzt nachholen und Ihnen einige der bisher ausgelassenen Konzepte vorstellen. Ansonsten steht es Ihnen natürlich frei, auf eigene Faust weiterzuforschen.

Modifikatoren für Bedingungen und Schleifen

Alle Bedingungen und Schleifen, die Sie heute kennengelernt haben, sind (mit Ausnahme von do) komplexe Anweisungen - sie operieren auf Blöcken aus anderen Anweisungen und benötigen kein Semikolon am Zeilenende. Daneben besitzt Perl auch einen Satz sogenannter Modifikatoren (***Modifier***) für einfache Anweisungen, mit denen Sie bedingungs- und schleifenähnliche Anweisungen erstellen können. Manchmal helfen Modifikatoren, einfache Bedingungen und Schleifen noch kürzer auszudrücken oder die Logik einer Anweisung besser darzustellen.

Es gibt vier mögliche Modifikatoren: if, unless, while und until. Jeden dieser ***Modifier*** können Sie an eine einfache Anweisung anhängen, direkt vor dem Semikolon. Hier ein paar Beispiele:

```
print "$wert" if ($wert < 10);
$z = $x / $y if ($y > 0);
$i++ while ($i < 10);
print $wert until ($wert++ > $maxwert)
```

Mit einem Bedingungsmodifikator (if oder unless) wird der vordere Teil der Anweisung nur ausgeführt, wenn der Bedingungsausdruck ***wahr*** (oder bei unless ***falsch***) ist. Mit einem Schleifenmodifikator (while oder until) wird die Anweisung so lange ausgeführt, wie der Ausdruck ***wahr*** (bzw. bei until ***falsch***) ist.

Beachten Sie, dass Anweisungen mit while- oder until-Modifikatoren nicht dasselbe sind wie normale while- oder until-Schleifen. Sie können sie weder mit Schleifensteuerbefehlen wie next oder last kontrollieren noch ihnen Schleifenlabels zuweisen.

Die do-Schleifen, die Sie vorhin in diesem Abschnitt kennengelernt haben, sind genau genommen Anweisungen mit Schleifenmodifikatoren. Do ist eine Funktion, die einen Anweisungsblock ausführt. Mit den Modifikatoren while und until bestimmen Sie, wie oft diese Ausführung wiederholt wird. Deshalb können Sie in do-Schleifen keine Schleifensteuerbefehle verwenden.

continue-Blöcke

Ein continue-Block ist ein optionaler Anweisungsblock nach einer Schleife, der ausgeführt wird, wenn der Schleifenblock zu Ende ausgeführt oder die Schleife mit next unterbrochen wurde. Der continue-Block wird nicht ausgeführt, wenn Sie die Schleife mit last oder redo abbrechen. Nach der Ausführung des continue-Blocks macht Perl mit dem nächsten Schleifendurchlauf (also der Auswertung der Schleifenbedingung) weiter.

Sie könnten einen continue-Block zum Beispiel verwenden, um einen Fehler zu beheben, der die Schleife unterbrochen hat. Danach läuft die Schleife normal weiter. Das ganze sieht folgendermaßen aus (wobei die while-Schleife hier auch eine for- oder foreach-Schleife sein könnte):

```
while ( Bedingung ) {
   # Anweisungen
   if (noch eine Bedingung) {
```

```
      # Fehler!
      next;  # (springt zum continue-Block)
   }
   # weitere Anweisungen
} continue {
   # behebe Fehler
   # (danach geht's am Schleifenanfang weiter)
}
```

Beachten Sie, dass `continue` *in Perl etwas anderes ist als das* `continue` *in C - dessen Perl-Pendant ist* `next`*.*

Konstruktion von *switch*- oder *case*-Anweisungen

Bemerkenswerterweise hat Perl keine explizite `switch`- (oder, je nach Ihrer Lieblingssprache, auch `case`-) Anweisung. Eine `switch`-Anweisung (Schalter, Weiche) ist eine Fallunterscheidung, mit der Sie einen Wert überprüfen, und viel kompakter, übersichtlicher und oft auch effizienter als eine Konstruktion aus zahllosen `if` und `elsif`, um festzulegen, was bei welchem Ergebnis geschehen soll.

Ein solches `switch`-Konstrukt können Sie in Perl jedoch »nachahmen«. Hier erweisen sich freistehende Blöcke (***bare blocks***) als äußerst nützlich. Für freistehende Blöcke gelten die gleichen Regeln wie für Schleifenblöcke; deswegen können Sie sie mit Labels versehen, mit `redo` erneut ausführen oder mit `last` und `next` verlassen (`next` führt einen eventuell vorhandenen `continue`-Block aus, `last` nicht). Das nutzen wir aus und schreiben unser »Perl-`Switch`« zum Beispiel so:

```
SWITCH: {
   $a eq "eins" && do {
                       $a = 1;
                       last SWITCH;
                     };
   $a eq "zwei" && do {
                       $a = 2;
                       last SWITCH;
                     };
   $a eq "drei" && do {
                       $a = 3;
                       last SWITCH;
                       };
# und so weiter
}
```

Mit Pattern Matching (Mustervergleichen) geht es noch kürzer. Ich komme darauf zurück, wenn wir uns an Tag 9 mit Pattern Matching und regulären Ausdrücken befassen.

goto

Ja, Perl unterstützt das verschriene `goto` (englisch »gehe zu«). Es ist in keinem Fall empfehlenswert, doch wenn Sie denn unbedingt wollen, können Sie `goto` auf drei Arten einsetzen:

- `goto LABEL`, wobei das Label irgendwo in Ihrem Skript stehen kann, in diesem Fall also wirklich eine Sprungmarke ist.
- `goto AUSDRUCK`, wobei der Ausdruck zu einem Labelnamen ausgewertet werden muss.
- `goto &NAME`, was nicht wirklich ein `goto` ist - es ersetzt eine laufende Unterroutine durch die Unterroutine `&NAME`. Es wird beim automatischen Laden von Subroutinen in Pakete genutzt.

Falls Sie sich für weitere `goto`-Details interessieren, schauen Sie in die ***perlsyn***- Manpage.

Zusammenfassung

Mit Bedingungen und Schleifen können Sie je nach Situation entscheiden, wie sich Ihr Skript verhalten soll. Diese Kontrollstrukturen sind so wichtig, dass wir schon seit dem zweiten Tag - lange bevor wir zu dieser Lektion gekommen sind - in den Beispielen mit ihnen gearbeitet haben.

Bedingungsanweisungen zweigen zu unterschiedlichen Anweisungsblöcken ab, abhängig davon, ob eine Bedingung erfüllt ist oder nicht. Sie haben die Konstrukte `if`, `if...else` und `if...elsif` kennengelernt, den Bedingungsoperator (`?...:`), der in andere Ausdrücke eingebaut werden kann, und Sie haben gesehen, wie Sie logische Operatoren (`&&`, `||`, `and` und `or`) zur Steuerung des Programmflusses einsetzen können.

Als nächstes haben wir Schleifen behandelt: insbesondere die `while`-Schleifen, die einen Anweisungsblock so lange ausführen, wie die Schleifenbedingung ***wahr*** ist. Sie haben das Gegenstück, die `until`-Schleife, kennengelernt, die abbricht, wenn die Bedingung ***wahr*** ist. In diesem Zusammenhang haben wir auch die `do`-Schleifen (`do...while` und `do...until`) und ihre Besonderheiten besprochen - beispielsweise dass sie gar keine echten Schleifen sind.

Die zweite Art von Schleife ist die `for`-Schleife, die ebenfalls Anweisungsblöcke wiederholt, aber geeigneter ist, wenn die Anzahl der Durchläufe von vornherein feststeht. Sie haben die `for`-Schleife mit ihrer C-ähnlichen Zählersyntax und die `foreach`-Schleife zum Durchlaufen aller Elemente in einer Liste kennengelernt.

Dann habe ich erklärt, wie Sie mit den Schleifensteuerbefehlen `next`, `last` und `redo` die Ausführung eines Blocks abbrechen und Teile der Schleife überspringen sowie in verschachtelten Schleifen eine bestimmte, mit einem Label versehene Schleife ansteuern können.

Schließlich haben wir die Lektion wie die beiden letzten Lektionen mit weiteren Anmerkungen zum Thema Eingabe beendet. Diesmal haben Sie gelernt, wie Sie mit der `<>`-Syntax aus Dateien lesen und mit der Variablen `$_` viele Operationen abkürzen können.

Mit der heutigen Lektion haben Sie sich die Grundlagen zu Perl vollständig angeeignet. Morgen werden wir die Woche mit einigen längeren Beispielen abschließen. Nächste Woche geht es nicht bloß weiter, nächste Woche legen wir richtig los und widmen uns einigen der mächtigsten und aufregendsten Perl-Features, darunter Pattern Matching (Mustervergleiche) und allerlei Listenakrobatik.

Sie haben heute folgende Perl-Funktionen kennengelernt:

- `do`, eine Funktion, die sich wie eine Schleife mit einem `while` oder `until` am Ende benimmt
- `rand`, das eine Zufallszahl zwischen 0 und ihrem Argument generiert,
- `srand`, den von `rand` verwendeten Zufallsgenerator initialisiert. Ohne Argument nimmt `srand` die aktuelle Uhrzeit als Startwert.

Mehr Details finden Sie - wie bereits erwähnt - in der ***perlfunc***-Manpage.

Fragen und Antworten

Frage:
Ich habe den Eindruck, dass `for`-Schleifen auch als `while`-Schleifen geschrieben werden könnten und umgekehrt.

Antwort:
Ja, das könnten sie mit Sicherheit. Aber es geht darum, was Sie sich vorstellen, eine »wiederhole x-mal«- oder eine »wiederhole so lange, bis«- Schleife. Sie sollen dann nicht noch darüber nachdenken müssen, wie Sie die eine in die andere umformulieren. Es ist eine der angenehmsten Eigenschaften von Perl, dass es in der Lage ist, sich an Ihre Denkweise anzupassen. Übrigens kennen sogar die weit weniger »entgegenkommenden« Sprachen C und Java `while`- und `for`-Schleifen.

Frage:
Ich habe versucht, mit `continue` eine Schleife abzubrechen, und Perl hat mit Fehlermeldungen um sich geworfen. Was habe ich falsch gemacht?

Antwort:

Sie haben vergessen, dass `continue` *in Perl nicht zum Abbrechen von Schleifen verwendet wird (oder Sie haben diesen Abschnitt übersprungen). Die Entsprechung zum* `continue` *von C ist in Perl* `next`*. Verwenden Sie* `continue` *nur als optionalen Anweisungsblock, der am Ende eines Blocks ausgeführt werden soll (siehe Vertiefungsabschnitt).*

Frage:
Ich habe mir Ihre beiden Beispiele für das Lesen aus Dateien angesehen. Eines der Beispiele verwendet `$_`, das andere nicht. Das erste ist zwar kürzer, aber wenn man nicht weiß, welche Operationen mit `$_` arbeiten, ist es kaum zu verstehen. Ich finde, dass die Lesbarkeit eines Skripts ein paar mehr Buchstaben wert ist.

Antwort:
Das ist definitiv ein guter Grundsatz, dem auch viele Perl-Programmierer folgen. »Schleichwege« über die `$_`*-Variable machen das Skript zwar kürzer, aber bestimmt auch weniger verständlich. In manchen Fällen - wie zum Beispiel für Eingaben mit* `<>` *- ist der Einsatz von* `$_` *jedoch eine Art weitverbreitete »Redensart«, die, sobald Sie sich daran gewöhnt haben, auch vernünftig und verständlich scheint. Wenn Sie Skripts von anderen lesen oder verändern müssen, werden Sie wahrscheinlich recht bald auf mysteriöses* `$_`*-Verhalten stoßen. Achten Sie also auf* `$_`*, setzen Sie es ein, wo es wirklich angemessen ist, oder vermeiden Sie es, wenn es Ihrer Meinung nach die Lesbarkeit zu sehr einschränkt. Die Entscheidung liegt ganz bei Ihnen.*

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist ein Block? Was kann ein Block enthalten?
2. Was ist der Unterschied zwischen `if` und `unless`?
3. Wodurch wird eine `while`-Schleife abgebrochen?
4. Warum sind `do`-Schleifen anders als `while`- oder `until`-Schleifen?
5. Welche drei Ausdrücke enthält die Klammer der `for`-Schleife? Was machen sie?
6. Erklären Sie den Unterschied zwischen `next`, `last` und `redo`.
7. Nennen Sie drei Situationen, in denen die Variable `$_` verwendet werden kann.
8. Worin unterscheidet sich `<>` von `<STDIN>`?

Übungen

1. Schreiben Sie ein Divisions-Skript, in dem Sie um die Eingabe von zwei Zahlen bitten und prüfen, dass keine der eingegebenen Zahlen negativ und die zweite Zahl nicht Null ist. Wenn beide Zahlen diesen Anforderungen entsprechen, teilen Sie die erste durch die zweite Zahl, und geben Sie das Ergebnis aus.
2. FEHLERSUCHE: Was stimmt nicht an diesem Skript? (Tipp: Es könnten auch mehrere Fehler sein)

```
    if ($wert == 4) then { print $wert; }
elseif ($wert > 4) { print "mehr als 4"; }
```

3. FEHLERSUCHE: Und was ist falsch an diesem hier (es könnten auch hier mehrere Fehler sein)?

```
    for ($i = 0, $i < $max, $i++) {
   $werte[$i] = "";
}
```

4. FEHLERSUCHE: Und was ist mit diesem hier?

```
    while ($i < $max) {
   $werte[$i] = 0;
}
```

5. Schreiben Sie das Skript `namen.pl` so um, dass es die Namen aus einer Datei liest (wenn Sie gestern die

Übungsaufgabe 3 gelöst haben, also auch mit Zweitnamen umgehen können, schreiben Sie diese Version entsprechend neu).

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Ein Block ist eine von geschweiften Klammern (`{}`) umgebene Gruppe von Perl- Anweisungen (die andere Blöcke enthalten kann). Blöcke werden meistens im Zusammenhang mit Bedingungen und Schleifen gebraucht, können aber auch allein, als freistehende Blöcke, verwendet werden (***bare blocks***).
2. Eine `if`-Anweisung führt ihren Block aus, wenn der Bedingungsausdruck zu ***wahr*** ausgewertet wird, `unless` führt ihn aus, wenn die Bedingung ***falsch*** ergibt.
3. Die `while`-Schleife bricht ab, wenn ihr Bedingungsausdruck ***falsch*** zurückgibt.
4. `do`-Schleifen unterscheiden sich in zweierlei Hinsicht von `while-` oder `until`- Schleifen. Zum einen werden ihre Blöcke vor der ersten Überprüfung, also mindestens einmal, ausgeführt. Zum anderen können Sie keine Schleifensteuerbefehle wie `last` oder `next` in einer `do`-Schleife verwenden, weil `do` gar keine echte Schleife, sondern eigentlich eine Funktion mit einem Modifikator ist.
5. Die drei Teile in den Klammern der `for`-Schleife sind:
1. a) Ein Ausdruck zum Initialisieren der Laufvariablen/des Schleifenzählers, wie `$i = 0.`
1. b) Ein Bedingungsausdruck, der die Anzahl der Durchläufe bestimmt, wie zum Beispiel `$i < $max.`
1. c) Ein Inkrementierungsausdruck, der den Schleifenzähler verändert (damit die Schleifenbedingung irgendwann nicht mehr erfüllt ist), zum Beispiel `$i++.`
6. `next`, `last` und `redo` sind Schleifensteuerbefehele. `next` bricht den aktuellen Durchlauf ab und startet am Schleifenanfang den nächsten, inklusive der Überprüfung der Bedingung in einer `for-` oder `while`-Schleife. Auch `redo` bricht den aktuellen Durchlauf ab, startet aber am Anfang des Blocks, ohne die Schleifenbedingung zu überprüfen. `last` beendet die Schleife ebenso komplett wie abrupt - es prüft nichts, startet nichts, es steigt einfach aus.
7. Hier ein paar Situationen, in denen Perl `$_` verwendet, wenn Sie keine Variable angegeben:
1. `while (<>)` weist `$_` jede Zeile einzeln zu.
1. `chomp` entfernt den Zeilenvorschub vom String in `$_`.
1. `foreach` verwendet `$_` als temporäre Schleifenvariable.
8. <> wird verwendet, um die Inhalte von Dateien einzulesen, die über die Kommandozeile angegeben (und in @ARGV gespeichert) werden. Mit `<STDIN>` liest man Daten von der Standardeingabe (normalerweise der Tastatur) ein.

Lösungen zu den Übungen

1. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
$zahl1 = 0;
$zahl2 = 0;
while () {
    print 'Geben Sie die erste Zahl ein: ';
    chomp($zahl1 = <STDIN>);
    print 'Geben Sie die zweite Zahl ein: ';
    chomp($zahl2 = <STDIN>);
    if ($zahl1 < 0 || $zahl2 < 0) {
        print "Keine negativen Zahlen!\n";
        next;
    } elsif ( $zahl2 == 0) {
        print "Die zweite Zahl darf nicht 0 sein!\n";
        next;
    } else { last; }
}
print "$zahl1 geteilt durch $zahl2 ist ";
printf("%.2f\n", $zahl1 / $zahl2 );
```

2. Es sind zwei Fehler: `elseif` ist kein gültiges Perl-Schlüsselwort, nehmen Sie statt dessen `elsif`. `then` ist

ebenfalls kein gültiges Perl-Schlüsselwort.

3. Die Ausdrücke innerhalb des `for`-Bedingungsausdrucks müssen mit Semikola, nicht mit Kommata getrennt werden.
4. Syntaktisch ist diese Schleife korrekt, aber sie wird nie enden - `$i` wird innerhalb der Schleife nicht inkrementiert.
5. Hier eine Antwort:

```
    #!/usr/bin/perl -w
%names = ();    # Hash: Namen
@raw = ();      # temp: rohe Woerter
$fn = '';       # Vorname
while (<>) {
    chomp;
    @raw = split(" ", $_);
    if ($#raw == 1) {  # Normalfall: zwei Woerter
        $names{$raw[1]} = $raw[0];
    } else {  # den Vornamen zusammensetzen
        $fn = '';
        for ($i = 0; $i < $#raw; $i++) {
            $fn .= $raw[$i] . " ";
        }
        $names{$raw[$#raw]} = $fn;
    }
}
foreach $lastname (sort keys %names) {
    print "$lastname, $names{$lastname}\n";
}
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Ein paar längere Beispiele

Widmen wir den letzten Tag der Woche ein paar weiteren Beispielen. Sie werden in diesem Kapitel nicht viel Neues lernen, auch Übungen und Quiz gibt es heute nicht. Betrachten Sie es als kurze Verschnaufpause, in der Sie sich vollständige Perl-Skripts ganz in Ruhe ansehen können. Insgesamt hat das Buch drei dieser Beispiellektionen (immer am Ende der Wochen), die Ihr neues Wissen zementieren sollen.

Wir besprechen heute drei Perl-Skripts:

- Eine weitere Version von ***stats.pl*** mit einem komplexeren Histogramm
- Ein Skript zum Buchstabieren von Zahlen
- Ein Skript zum Konvertieren von Textdateien in einfache Webseiten

Statistik mit verbessertem Histogramm

Verändern wir also noch einmal das Statistikprogramm, an dem wir die ganze Woche lang gearbeitet haben. Die Version von gestern erzeugte ein horizontales Histogramm, das etwa so aussah:

```
Haeufigkeit der einzelnen Zahlen:
1  | *****
2  | *************
3  | ********************
4  | ****************
5  | ***********
6  | ****
43 | *
62 | *
```

Heute wollen wir ein Diagramm mit vertikalen Balken ausgeben:

```
     *
     *
     *
     *
     *
     * *
     * * *
     * * *         *     *     *
     * * *   * * *     *     *
   * * * *   * * *     *     *
   * * * *   * * *     *     *
 * * * * * * * * *     *     *   *
 * * * * * * * * *  *  *  *  *  *  *  *
----------------------------------------
 1 2 3 4 5 6 7 8 9 12 23 25 34 37 39 42
```

Diese Diagrammform ist sehr viel schwieriger zu erstellen als die horizontale. Wir erstellen sie mit zwei verschachtelten `for`-Schleifen und großer Sorgfalt beim Zählen, damit auch alles an der richtigen Stelle landet.

Noch etwas ist in dieser Version von ***stats.pl*** anders: Sie holt sich die Daten aus einer Datei, anstatt sie am Prompt vom Benutzer eintippen zu lassen. Wie bei allen Skripts, die aus einer Datei lesen, müssen Sie diese Datei beim Aufruf des Skripts in der Kommandozeile angeben:

```
% statsfinal.pl daten.txt
```

In der Datei (hier ***daten.txt***) steht jede Zahl in einer einzelnen Zeile.

*Sie finden **daten.txt** auch auf der beiliegenden CD, wenn Sie keine Lust haben, die Zahlen selbst einzutippen.*

Listing 7.1 zeigt den Code unseres endgültigen Statistikskripts. So viel, wie wir bereits damit gearbeitet haben, sollte es ihnen vertraut sein. Konzentrieren Sie sich auf die beiden Teile des Skripts, die anders sind als in der letzten Version: die Eingabeschleife, die die Daten aus der Datei liest (Zeile 14 bis 21), und den Code zur Ausgabe des neuen Histogramms (Zeile 36 bis 49).

Listing 7.1: Das Skript statsfinal.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $input = '';  # Benutzereingabe: Zahl
4:  @nums = ();   # Array: Nums;
5:  %freq = ();   # Hash: Zahl-Haeufigkeit
6:  $maxfreq = 0; # hoechste Haeufigkeit
7:  $count = 0;   # Anzahl aller Nums
8:  $sum = 0;     # Summe
9:  $avg = 0;     # Durchschnitt
10: $med = 0;     # Median
11: @keys = ();   # temp Keys
12: $totalspace = 0; # gesamte Breite des Histogramms
13:
14: while (defined ($input = <>)) {
15:     chomp ($input);
16:     $nums[$count] = $input;
17:     $freq{$input}++;
18:     if ($maxfreq < $freq{$input}) { $maxfreq = $freq{$input} }
19:     $count++;
20:     $sum += $input;
21: }
22: @nums = sort { $a <=> $b } @nums;
23:
24: $avg = $sum / $count;
25: $med = $nums[$count /2];
26:
27: print "\nAnzahl der eingegebenen Nums: $count\n";
28: print "Summe der Nums: $sum\n";
29: print "Kleinste Zahl: $nums[0]\n";
30: print "Groesste Zahl: $nums[$#nums]\n";
31: printf("Durchschnitt: %.2f\n", $avg);
32: print "Mittelwert: $med\n\n";
33:
34: @keys = sort { $a <=> $b } keys %freq;
35:
36: for ($i = $maxfreq; $i > 0; $i--) {
37:     foreach $zahl (@keys) {
38:         $space = (length $zahl);
39:         if ($freq{$zahl} >= $i) {
40:             print( (" " x $space) . "*");
41:         } else {
42:             print " " x (($space) + 1);
43:         }
44:         if ($i == $maxfreq) { $totalspace += $space + 1; }
45:     }
46:     print "\n";
47: }
48: print "-" x $totalspace;
49: print "\n @keys\n";
```

Da Sie den <>-Operator zum Einlesen von Daten aus Dateien bereits kennen, dürften die Zeilen 14 bis 21 Sie nicht überraschen. Beachten Sie, dass wir jede Zeile der Datei (also jede Zahl) der `$input`-Variablen zuweisen und diesen Wert dann im gesamten Block verwenden.

Warum `$input` und nicht `$_`? Wir hätten auch `$_` einsetzen können, aber viele der Anweisungen in diesem Block brauchen einen wirklichen Variablenbezug (sie greifen nicht von allein auf `$_` zurück). In diesem Beispiel würde der Code durch `$_` nur wenig kürzer, aber um einiges schwieriger zu lesen. Deshalb ist es hier die wohl bessere Idee, sich zugunsten der Lesbarkeit für eine eigene Variable zu entscheiden.

> *Je mehr Möglichkeiten ich in diesem Buch erkläre, desto mehr Möglichkeiten stehen Ihnen auch zur Auswahl - dass Perl ein besonderes Feature anbietet, bedeutet noch lange nicht, dass Sie es auch verwenden müssen. Man sollte immer abwägen zwischen sehr kurzem Code, den nur noch Perl-Experten entziffern können, und längerem, vielleicht gar weniger effizientem, aber dafür lesbarem Code. Betrachten Sie Ihr Perl-Skript als besonders gut gemacht, wenn jemand anderes es einfach durchlesen kann.*

Abgesehen davon, dass die Daten hier aus einer Datei anstatt von der Standardeingabe gelesen werden, unterscheidet sich dieser `while`-Block nur noch in einem Punkt von der letzten Version: Die neu hinzugefügte Anweisung in Zeile 18 berechnet den Wert von `$maxfreq`, die maximale Häufigkeit. Diese gibt an, wie oft die Zahl vorkommt, die am häufigsten aufgetaucht ist. Mit diesem Wert legen wir später die Gesamthöhe des Diagramms fest. Hier vergleichen wir die maximale Häufigkeit lediglich mit der aktuellen und verändern `$maxfreq`, wenn letztere größer ist.

Weiter unten im Skript - nachdem wir sortiert, summiert und die ersten Ergebnisse ausgegeben haben - kommen wir zum Histogrammteil, der etwas einschüchternden Anhäufung von Schleifen in den Zeilen 36 bis 49.

Ein Histogramm mit horizontalen Balken ist viel einfacher zu erstellen als eins mit vertikalen. Für das horizontale Histogramm brauchten wir vorgestern nur die Schlüssel im `%freq`-Hash zu durchlaufen und die entsprechende Zahl Sternchen auszudrucken (und das Ganze etwas zu formatieren). Für ein vertikales Histogramm müssen wir von Anfang an viel mehr Informationen über das Gesamtlayout parat haben, weil die Zeilen, die wir zeichnen, in keinem direkten Zusammenhang mit einem bestimmten Schlüssel oder Wert im Hash stehen. Außerdem müssen wir auch die Leerstellen für die Formatierung beachten.

Wir bauen das Histogramm mit zwei Schleifen. Die äußere, eine `for`-Schleife, kontrolliert die Anzahl der auszugebenden Zeilen, das heißt die Höhe von oben nach unten. Die zweite, eine `foreach`-Schleife, bewegt sich innerhalb jeder Zeile von links nach rechts und schreibt entweder ein Sternchen oder ein Leerzeichen in jede Spalte. Mit diesen beiden verschachtelten Schleifen (der `for`-Schleife außen und der `foreach`- Schleife innen) können wir uns von Zeile zu Zeile und von links nach rechts bewegen und sowohl die Höhe als auch die Breite des Histogramms von unseren Daten abhängig machen.

Zuerst erstellen wir in Zeile 34 eine sortierte Liste aller Schlüssel des `%freq`-Hash, vor allem aus Bequemlichkeit und damit die `for`-Schleifen wenigstens etwas durchschaubarer werden.

In Zeile 36 startet unsere äußere `for`-Schleife. Die Gesamthöhe des Diagramms wird von der am häufigsten auftretenden Zahl in unseren Daten bestimmt. Hier kommt der beim Einlesen ermittelte Wert von `$maxfreq` zum Einsatz. Die äußere `for`-Schleife beginnt bei diesem Höchstwert und arbeitet sich hinunter bis zur 0, wo sie abbricht. So erhalten wir die richtige Anzahl Zeilen.

Die innere Schleife ist für die einzelnen Zeilen zuständig. Sie durchläuft die Schlüssel des `%freq`-Hash (also unsere Zahlen) und überprüft, ob in der aktuellen Zeile bei dieser Zahl ein `*` oder ein Leerzeichen gesetzt werden muss. Wir achten auch hier auf die Formatierung und fügen für die Spalten mit mehrstelligen Zahlen entsprechend mehr Leerstellen hinzu (`$space` wird für eine Zahl 333 einen anderen Wert haben als für 1).

Zeile für Zeile machen wir ab Zeile 38 folgendes:

- In Zeile 38 berechnen wir die für diese Spalte benötigten Leerstellen, abhängig davon, wie viele Stellen der aktuelle Wert hat.
- Zeile 39 entscheidet, ob ein Sternchen ausgegeben werden muss. Hier wird überprüft, ob die Häufigkeit der Zahl, die wir gerade betrachten, unserer vertikalen Position im Histogramm entspricht (der Häufigkeitswert also größer oder gleich der augenblicklichen Höhe des Balkens, dem aktuellen Wert von `$i`, ist). Je weiter

wir im Histogramm nach unten gehen, das heißt je kleiner `$i` wird, desto mehr Zahlen erfüllen dieses Kriterium.
- Wenn nach Zeile 39 ein Sternchen gesetzt werden muss, gibt Zeile 40 sowohl das Sternchen als auch die für die korrekte Spaltenbreite nötigen Leerzeichen aus.
- Wenn kein Sternchen erforderlich ist, geben wir mit den Zeilen 41 und 42 die richtige Anzahl von Füllzeichen aus: Spaltenbreite plus ein Leerzeichen extra.
- Zeile 44 ist knifflig. Diese Zeile berechnet die Gesamtbreite des Histogramms, basierend auf der Anzahl der Stellen aller Zahlen und den Leerzeichen dazwischen. Das Ergebnis brauchen wir zwar erst später, in Zeile 48, zum Ausgeben einer Trennlinie, aber wenn wir schon einmal mitten in der Schleife sind, können wir es auch gleich jetzt berechnen. Diese Schleife macht folgendes: Wenn `$i` gleich `$maxfreq` ist - das heißt, wenn wir die erste Zeile des Diagramms erstellen -, addiert diese Schleife die Anzahl der in der aktuellen Spalte erforderlichen Leerstellen zur Variablen `$totalspace`. Nach dem ersten vollständigen Durchlauf der `foreach`-Schleife enthält `$totalspace` genau die Gesamtbreite des Histogramms.
- Wenn wir mit einer Histogrammzeile fertig sind, gibt die äußere `for`-Schleife schließlich in Zeile 46 einen Zeilenvorschub aus - und beginnt alles wieder von vorn.

Wenn wir die Sternchen des Histogramms ausgegeben haben, fehlen nur noch die Bezeichnungen für die Spalten. Hier simulieren wir mit Bindestrichen eine Querlinie (die Anzahl der erforderlichen Bindestriche haben wir bereits mit dem Wert von `$totalspace` berechnet) und schreiben unsere Zahlen darunter: einen String, der die Variable `@keys` interpoliert und alle Elemente von `@keys`, durch je ein Leerzeichen getrennt, ausgibt.

Kompliziert verschachtelte Schleifen sind häufig nur schwer nachzuvollziehen. Manchmal reicht eine Erklärung wie meine einfach nicht aus. Wenn dieses Beispiel Sie auch jetzt noch verwirrt, versuchen Sie, es mit ein paar kleinen Beispielzahlen selbst »durchzuspielen«. Verfolgen Sie Schritt für Schritt und Schleife für Schleife die »Entwicklung« der verschiedenen Werte - das hilft Ihnen, die Zusammenhänge zwischen den einzelnen Variablen klarer zu erkennen.

Ein Zahlenbuchstabierer

Unser zweites Beispiel hat im wirklichen Leben nicht besonders viel Sinn, aber es zeigt ein paar komplexere Einsatzmöglichkeiten für `if` und `while`. Dieses Skript bittet Sie um die Eingabe einer einstelligen Zahl (und fängt Strings oder mehrstellige Zahlen ab). Dann schreibt es die Zahl als Wort aus, also `2` als `zwei` und `5` als `fuenf` und so weiter. Schließlich fragt es, ob Sie eine weitere Zahl eingeben möchten, und wenn ja, wiederholt es den gesamten Prozeß. Hier ein Beispiel, wie es ablaufen könnte:

```
% zahlenbuchstabierer.pl
Geben Sie die zu buchstabierende Zahl ein: buh
Keine Strings. Eine Zahl von 0 bis 9 bitte.
Geben Sie die zu buchstabierende Zahl ein: 45
Zu hoch. 0 bis 9 bitte.
Geben Sie die zu buchstabierende Zahl ein:: 6
6 ist sechs.
Eine weitere Zahl versuchen (j/n)?: weiss nicht
j oder n bitte.
Eine weitere Zahl versuchen (j/n)?: j
Geben Sie die zu buchstabierende Zahl ein: 3
3 ist drei.
Eine weitere Zahl versuchen (j/n)?: n
```

Listing 7.2 zeigt den kompletten Code.

Listing 7.2: Das Skript zahlenbuchstabierer.pl.

```
1:  #!/usr/bin/perl -w
2:  # zahlenbuchstabierer:  buchstabiert Zahlen
3:  # Simple Version, nur einstellige Zahlen
4:
5:  $num = 0;    # Zahl
6:  $exit = "";  # Programm verlassen? j oder n.
7:
8:  while ($exit ne "n") {
9:
10:     while () {
```

```
11:         print 'Geben Sie die zu buchstabierende Zahl ein: ';
12:         chomp($num = <STDIN>);
13:         if ($num ne "0" && $num == 0) { # wenn $num ein String
14:             print "Keine Strings. Eine Zahl von 0 bis 9 bitte.\n";
15:             next;
16:         }
17:         if ($num > 9) { # wenn $num mehrstellige Zahl
18:             print "Zu hoch. 0 bis 9 bitte.\n";
19:             next;
20:         }
21:         if ($num < 0) { # wenn $num negative Zahl
22:             print "Keine negativen Zahlen. 0 bis 9 bitte.\n";
23:             next;
24:         }
25:         last;
26:     }
27:
28:     print "$num ist ";
29:     if ($num == 1) { print 'eins'; }
30:     elsif ($num == 2) { print 'zwei'; }
31:     elsif ($num == 3) { print 'drei'; }
32:     elsif ($num == 4) { print 'vier'; }
33:     elsif ($num == 5) { print 'fuenf'; }
34:     elsif ($num == 6) { print 'sechs'; }
35:     elsif ($num == 7) { print 'sieben'; }
36:     elsif ($num == 8) { print 'acht'; }
37:     elsif ($num == 9) { print 'neun'; }
38:     elsif ($num == 0) { print 'null'; }
39:     print "\n";
40:
41:     while () {
42:         print 'Eine weitere Zahl versuchen (j/n)?: ';
43:         chomp ($exit = <STDIN>);
44:         $exit = lc $exit;
45:         if ($exit ne 'j' && $exit ne 'n') {
46:             print "j oder n bitte.\n";
47:         }
48:         else { last; }
49:     }
50: }
```

Beginnen wir bei der äußeren Schleife, und arbeiten wir uns nach innen vor. Die erste `while`-Schleife, von Zeile 8 bis 50, enthält fast das gesamte Skript, denn sie soll, wenn am `ja/nein`-Prompt `j` eingegeben wird, auch fast das gesamte Skript noch einmal ausführen. Weil wir sie mindestens einmal laufen lassen möchten, muss die Bedingung in Zeile 8 am Anfang wahr sein (in weiser Voraussicht haben wir die Variable `$exit` dementsprechend initialisiert).

Die zweite `while`-Schleife, von Zeile 10 bis 26, überprüft die Benutzereingaben. Das erste `if` stellt sicher, dass Sie keine Strings eingegeben haben (wenn `$num` nicht das Zeichen `0` ist, aber im numerischen Vergleich zu 0 ausgewertet wird, muss `$num` ein String sein). Das zweite `if` sieht nach, ob Sie eine Zahl größer 9 eingegeben haben (diese Version des Skripts buchstabiert nur einstellige Zahlen), und das dritte prüft auf negative Zahlen. Wenn eine dieser Bedingungen erfüllt ist, überspringen wir mit dem Schleifensteuerbefehl `next` den Rest des Blocks und gehen zurück zum Anfang der Schleife, in der wir uns unmittelbar befinden - das ist in diesem Fall immer noch die endlose `while`-Schleife in den Zeilen 10 bis 26. Wenn die Eingabe unseren Kriterien entspricht, also kein String und eine Zahl zwischen 0 und 9 ist, dann steigen wir mit `last` aus dieser `while`-Schleife aus und machen mit der ersten ihr folgenden Anweisung weiter.

Diese Anweisung steht in Zeile 28. Wir geben den ersten Teil der Ergebnismeldung aus und gehen dann durch einen Satz von `if`- und `elsif`-Anweisungen, die für die aktuelle Zahl den entsprechenden String auf den Bildschirm schreiben. Dieser Abschnitt (Zeile 29 bis 38) wäre ein klassischer Fall für eine `switch`-Anweisung, mit der wir hier nicht immer wieder das gleiche eintippen müßten (den Programmiergöttern sei Dank für Copy-and-Paste).

Sobald wir unser Zahlwort ausgegeben haben, fragen wir in der letzten `while`-Schleife (Zeile 41 bis 49), ob der gesamte Vorgang von vorn beginnen soll. Als Antwort erwarten wir ein `j` oder ein `n`. Beachten Sie den Aufruf der Funktion `lc` in Zeile 44 - wenn der Benutzer Großbuchstaben eingibt, wandelt `lc` sie vor der Überprüfung in Kleinbuchstaben um - damit akzeptieren wir also auch `J` oder `N`, ohne explizit darauf prüfen zu müssen.

Wie Sie sehen, bestimmen wir hier nicht, was bei einem `j` oder `n` passieren soll, wir überprüfen nur, ob tatsächlich `j` oder `n` eingegeben wurde. Mehr ist gar nicht nötig. Sobald `$exit` einen gültigen Wert hat, endet die äußere `while`-Schleife, und wir kommen direkt zum Anfang, zur Schleifenbedingung in Zeile 8, zurück. Hier entscheiden wir über den weiteren Ablauf: Bei einem `n` bricht die äußere `while`- Schleife wegen nicht erfüllter Schleifenbedingung ab, und das Skript ist beendet. Anderenfalls beginnen wir den Durchlauf von vorn und machen so lange weiter, bis der Anwender ein `n` eingibt.

Simple Text-zu-HTML-Konvertierung

Beenden wir dieses Kapitel mit einem etwas nützlicheren Skript: ***webseite.pl*** liest eine einfache Textdatei, fragt vom Benutzer ein paar grundlegende Einstellungen ab und spuckt dann eine HTML-Version der Datei aus. Dieses Skript ist kein besonders raffinierter HTML-Generator. Es läßt Sie lediglich Vorder- und Hintergrundfarbe, eine Überschrift sowie Ihre E-Mail-Adresse vorgeben, und im Text selbst fügt es nur Absatz-Tags an den richtigen Stellen ein. Aber Sie erhalten immerhin eine grundlegende HTML-Vorlage, mit der Sie weiterarbeiten können.

Der Ablauf

Bevor es Ihren Text in HTML konvertiert, bittet das ***webseite***-Skript Sie um ein paar Angaben:

- Titel der Seite (`<TITLE>...</TITLE>` in HTML)
- Hintergrund- und Textfarbe. Zur Wahl stehen hier nur die von HTML unterstützten Namen der 16 Grundfarben, alles andere fangen wir ab. Hierzu geben wir außerdem etwas rudimentäre Online-Hilfe.
- eine Überschrift am Anfang der Seite (`<H1>...</H1>` in HTML)
- eine E-Mail-Adresse, die als Link am unteren Ende der fertigen HTML-Seite eingefügt wird.

Ein Ablauf von ***webseite.pl*** könnte zum Beispiel folgendermaßen aussehen:

```
% webseite.pl heine.txt
Geben Sie den Titel Ihrer Webseite ein: Heinrich Heine, Reisebilder, Dritter Teil
Geben Sie die Hintergrundfarbe ein (? zeigt Auswahl): ?
Zur Wahl stehen folgende Farben:
black (schwarz), maroon (dunkelbraun),
green (gruen), olive (olivgruen),
navy (dunkelblau), purple (violett),
teal (blaugruen), gray (grau),
silver (silbergrau), red (rot),
lime (hellgruen), yellow (gelb),
blue (blau), fuchsia (pink),
aqua (hellblau), white (weiss),
oder Return fuer keine Farbe
Geben Sie die Hintergrundfarbe ein (? zeigt Auswahl): white
Geben Sie Textfarbe ein (? zeigt Auswahl): black
Geben Sie eine Ueberschrift ein: Kapitel XXVI
Geben Sie Ihre E-Mail-Adresse ein: lemay@lne.com
*****************************
<HTML>
<HEAD>
<TITLE>Heinrich Heine, Reisebilder, Dritter Teil</TITLE>
</HEAD>
<BODY BGCOLOR="white" TEXT="black">
<H1>Kapitel XXVI</H1>
<P>
Die Natur wollte wissen, wie sie aussieht, und sie erschuf Goethe.
```

(... und so weiter, hier aus Platzgründen nicht aufgeführt).

```
<P>
Es liegt Wahrheit in diesen Worten und ich bin sogar der Meinung, dass Goethe manchmal seine Sa
<P>
<HR>
<ADDRESS><A HREF="mailto:lemay@lne.com">lemay@lne.com</A></ADDRESS>
</BODY>
</HTML>
```

Der fertige HTML-Code wird von diesem Skript nur auf den Bildschirm ausgegeben - das nützt Ihnen herzlich wenig, ich weiß - und bitte um etwas Geduld. Wie Sie Daten in eine Datei statt an die Standardausgabe leiten, behandeln wir an Tag 15. Bis dahin könnten Sie den hier ausgegebenen HTML-Text mit Copy-and-Paste in einen Texteditor übernehmen und als HTML-Datei speichern. Wenn Sie diese Datei dann in einen Webbrowser laden, sehen Sie das Ergebnis von ***webseite.pl***. Unser Beispiel ergäbe etwa folgendes Bild:

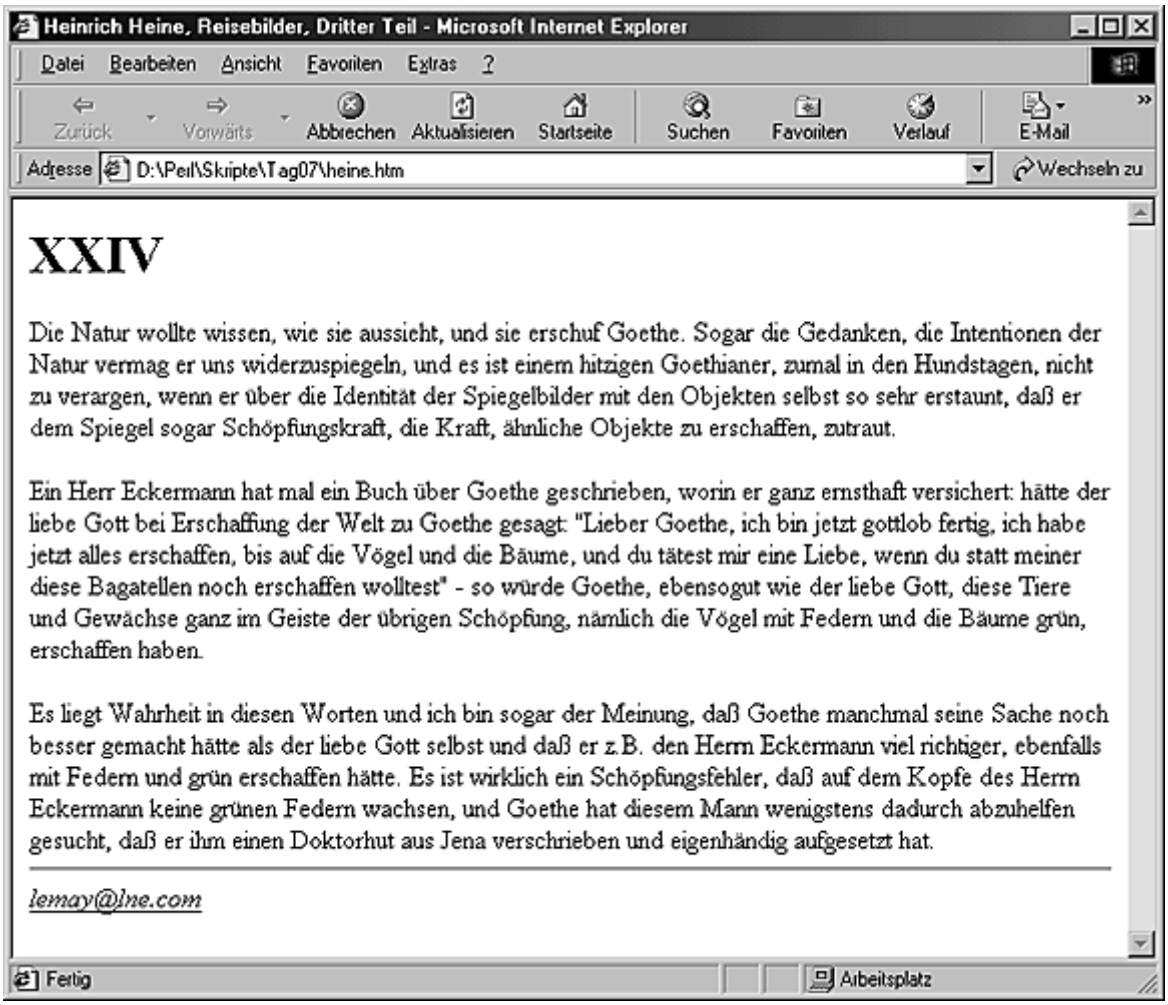

Abbildung 7.1: Das Ergebnis des Skripts webseite.pl

Die Eingabedatei

Eine Bemerkung zu der Textdatei, die Sie an ***webseite.pl*** zum Konvertieren übergeben: Das Skript geht davon aus, dass diese aus Absätzen besteht, die durch Leerzeilen voneinander getrennt sind. Hier zum Beispiel der Inhalt der Datei ***heine.txt***, die ich für das Beispiel oben verwendet habe:

```
Die Natur wollte wissen, wie sie aussieht, und sie erschuf Goethe. Sogar die Gedanken, die Inte
Ein Herr Eckermann hat mal ein Buch über Goethe geschrieben, worin er ganz ernsthaft versichert
Es liegt Wahrheit in diesen Worten und ich bin sogar der Meinung, dass Goethe manchmal seine Sa
```

Ich habe hier ausnahmsweise auf die ASCII-sichere Umschreibung von Umlauten und ß verzichtet - schließlich geht es bei Webseiten nicht zuletzt um die Ästhetik der Darstellung. Am Tag 1 habe ich erklärt, warum Ihr Perl-Interpreter damit nicht ohne weiteres umgehen kann. Wenn Sie diese Textdatei in dem Zeichensatz speichern, den Ihr Perl-Interpeter voraussetzt, werden Sie innerhalb Ihres Systems erst einmal keine Schwierigkeiten haben - beachten Sie jedoch, dass diese Datei auf dem Rechner Ihrer Brieffreundin in Kopenhagen (oder auch nur bei Ihrem Nachbarn, mit dessen Betriebssystem Sie nie arbeiten würden) zu ganz anderen, unerwünschten Ergebnissen führen kann. Übrigens haben Umlaute auch in HTML-Code eigentlich nichts zu.

Das Skript

Listing 7.3 zeigt den Code für unser Skript.

Listing 7.3: webseite.pl

```
1:  #!/usr/bin/perl -w
2:  #
3:  # webseite:  simple Textdatei-zu HTML-Konvertierung
4:  # *sehr* simpel.  Keine Sonderzeichen, Links, Fett- oder
5:  # andere Formatierungen, etc.
6:  # Leerzeile in Textdatei == Absatz in HTML-Code
7:
8:  $title = '';                    # <TITLE>, Titel
9:  $bgcolor = '';                  # BGCOLOR, Hintergrundfarbe
10: $text = '';                     # TEXT, Textfarbe
11: $head = '';                     # <H1>, erste Ueberschrift
12: $mail = '';                     # E-Mail-Adresse
13:
14: print " Geben Sie den Titel Ihrer Webseite ein: ";
15: chomp($title = <STDIN>);
16:
17: foreach $farbe ('Hintergrund', 'Text') { # laeuft zweimal: einmal
18:                                          # je Farbe
19:     $in = '';                   # temp. Eingabe
20:     while () {
21:        print "Geben Sie die ${farbe}farbe ein (? zeigt Auswahl): ";
22:         chomp($in = <STDIN>);
23:         $in = lc $in;
24:
25:         if ($in eq '?') {       # Hilfe anzeigen
26:             print "\nZur Wahl stehen folgende Farben:\n";
27:             print "black (schwarz), maroon (dunkelbraun),\n";
28:             print "green (gruen), olive (olivgruen),\n";
29:             print "navy (dunkelblau), purple (violett),\n";
30:             print "teal (blaugruen), gray (grau),\n";
31:             print "silver (silbergrau), red (rot),\n";
32:             print "lime (hellgruen), yellow (gelb),\n";
33:             print "blue (blau), fuchsia (pink),\n";
34:             print "aqua (hellblau), white (weiss),\n";
35:             print "oder Return fuer keine Farbe\n\n";
36:             next;
37:         } elsif ($in eq '' or
38:                  $in eq 'black' or
39:                  $in eq 'maroon' or
40:                  $in eq 'green' or
41:                  $in eq 'olive' or
41:                  $in eq 'navy' or
42:                  $in eq 'purple' or
43:                  $in eq 'teal' or
44:                  $in eq 'gray' or
45:                  $in eq 'silver' or
46:                  $in eq 'red' or
47:                  $in eq 'lime' or
48:                  $in eq 'yellow' or
49:                  $in eq 'blue' or
50:                  $in eq 'fuchsia' or
51:                  $in eq 'aqua' or
52:                  $in eq 'white') { last; }
53:         else {
54:             print "Das ist kein gueltiger Farbname.\n";
55:         }
56:     }
57:
58:     if ($farbe eq 'Hintergrund') {  # Hintergrundfarbe
59:         $bgcolor = $in;
60:     } else {                        # Textfarbe
61:         $text = $in;
62:     }
63: }
64:
65: print "Geben Sie eine Ueberschrift ein: ";
```

```
66: chomp($head = <STDIN>);
67:
68: print "Geben Sie Ihre E-Mail-Adresse ein: ";
69: chomp($mail = <STDIN>);
70:
71: print '*' x 30;
72:                               # jetzt geht der HTML-Code los:
73: print "\n<HTML>\n<HEAD>\n<TITLE>$title</TITLE>\n";
74: print "</HEAD>\n<BODY";
75: if ($bgcolor ne '') { print qq( BGCOLOR="$bgcolor"); }
76: if ($text ne '') { print qq( TEXT="$text"); }
77: print ">\n";
78: print "<H1>$head</H1>\n<P>";
79:
80: while (<>) {
81:     if ($_ eq "\n") {
82:         print "<p>\n";
83:     } else {
84:         print $_;
85:     }
86: }
87:
88: print qq(<HR>\n
89:          <ADDRESS><A HREF="mailto:$mail">$mail</A></ADDRESS>\n);
90: print "</BODY>\n</HTML>\n";
```

Dieses Skript ist nicht besonders komplex oder syntaktisch raffiniert. Es verwendet nicht einmal Arrays oder Hashes (Wofür auch? Es gibt nichts, was hier gespeichert oder berechnet werden müßte) - nur viele, viele Schleifen und Bedingungen.

Betrachten wir zuerst die große `foreach`-Schleife, die in Zeile 17 beginnt. Diese Schleife ist zuständig für die Frage nach Hintergrund- und Textfarbe. Weil sich diese beiden Eingabeaufforderungen exakt gleich verhalten, wollte ich denselben Code nicht für jede einzelne Aufforderung wiederholen (insbesondere, weil die `if`-Bedingungen in Zeile 37 bis 53 wirklich reichlich sind). Später werden Sie lernen, wie man sich derartig wiederholenden Code in eine Subroutine packt und dann lediglich diese Subroutine zweimal aufruft. Doch jetzt, wo wir viel über Schleifen, aber nichts über Subroutinen wissen, habe ich mich für eine `foreach`-Schleife entschieden.

Die Schleife wird zweimal durchlaufen, einmal für den String `'Hintergrund'` und einmal für den String `'Text'`. Wir benutzen diese Strings für die Eingabeaufforderung und später zur Zuweisung der eingegebenen Werte an die entsprechende Variable (`$bgcolor` oder `$text`).

Innerhalb der `foreach`-Schleife haben wir eine andere, eine unendliche `while`- Schleife, die die Eingabeaufforderungen so lange wiederholt, bis wir akzeptable Eingaben haben (Fehlerbehandlung ist immer eine gute Programmierübung). Der Benutzer kann 18 verschiedene Dinge eingeben: eine der sechzehn Grundfarben, ein Fragezeichen oder gar nichts.

Die Bedingungen in Zeile 25 bis 55 durchlaufen jede dieser Möglichkeiten, zuerst die Eingabe von `?`. Als Antwort auf ein Fragezeichen müssen wir lediglich eine hilfreiche Mitteilung ausgeben und dann mit `next` zum nächsten Durchlauf der `while`-Schleife springen (also, die Eingabeaufforderung neu anzeigen und auf weitere Eingaben warten).

Die nächste Überprüfung (ab Zeile 37) stellt sicher, dass wir korrekte Eingaben haben, nämlich entweder einen Zeilenvorschub (dann haben wir eine leere Eingabe und legen keine Farbe fest[1]) oder eine der sechzehn Grundfarben. Beachten Sie, dass hier immer auf kleingeschriebene Farbnamen überprüft wird. In Zeile 23 haben wir mit der Funktion `lc` die Eingabestrings komplett in Kleinbuchstaben umgewandelt; deswegen kann der Benutzer auch `BLACK` oder `Black` eingeben - wir haben alle Möglichkeiten zu einer zusammengefaßt (aber bequemerweise die Eingabe von `?` nicht berührt).

Wenn die Eingabe einer dieser Vorgaben entspricht, steigen wir in Zeile 52 mit `last` aus der `while`-Schleife aus (Sie erinnern sich, `next` und `last` beziehen sich ohne Labels auf die nächste sie umgebende Schleife - hier also auf die `while`-, nicht auf die `foreach`-Schleife). Wenn uns die Eingabe nicht paßt, springen wir zum letzten `else`- Fall in Zeile 53, geben eine Fehlermeldung aus und beginnen die `while`-Schleife von vorn.

Die letzte Überprüfung in der `foreach`-Schleife (Zeile 58 bis 62) stellt fest, ob es sich bei der Eingabe um den Wert

für die Hintergrund- oder die Textfarbe handelt, und weist den Farbwert dann der entsprechenden Variablen zu.

Der Rest des Skripts, von Zeile 73 bis zum Ende, gibt den Anfang unserer HTML-Datei aus, liest und konvertiert die in der Kommandozeile angegebene Textdatei und setzt die abschließenden HTML-Tags ans Ende. Beachten Sie die Überprüfungen in Zeile 75 und 76: Wenn `$bgcolor` und `$text` keine Werte haben, schreiben wir diese Attribute gar nicht erst in den HTML-Tag `<BODY>`. (Einfacher wäre es, sie dort stehen zu lassen, `BGCOLOR=""` oder `TEXT=""`, aber das hätte nicht besonders nett ausgesehen.)

Beachten Sie auch die Verwendung der Funktion `qq`. Sie haben `qq` an Tag 2 im Vertiefungsabschnitt kennengelernt. Mit der `qq`-Funktion erstellt man einen ***double- quoted*** String, ohne doppelte Anführungszeichen zu verwenden. Wenn ich hier doppelte Anführungszeichen verwendet hätte, hätte ich vor die Anführungszeichen im String selbst einen Backslash setzen müssen. Ich finde, so sieht es besser aus.

Die Zeilen 80 bis 86 lesen (mit `<>`) die Textdatei und geben dann einfach alles wieder aus, nur mit Absatz-Tags (`<P>`) statt Leerzeilen. Eine robustere Version dieses Skripts würde nach Sonderzeichen suchen (Akzenten, Umlauten und so weiter) und sie durch ihre entsprechenden HTML-Codes ersetzen - aber diese Aufgabe läßt sich mit Pattern Matching (Mustervergleich) viel einfacher lösen; deswegen heben wir uns das für später auf.

Zum Abschluß bleibt nur noch die Ausgabe des E-Mail-Links (mit einem HTML `mailto` und Link-Tags) und der Tags zum Beenden der HTML-Datei (`</BODY></HTML>`).

Zusammenfassung

Programmierlehrbüchern mangelt es selten an wortreichen Erklärungen, aber häufig an konkreten Beispielen. Ich möchte mich nicht um ausführliche Erklärungen drücken (wer lacht da?), aber diese Lektion - und die zwei weiteren an den Tagen 14 und 21 - zeigt Ihnen etwas längeren Code, der ohne thematische Trennung die Techniken aus den vorangegangenen Lektionen anwendet und vielleicht sogar eine mehr oder weniger nützliche Aufgabe erfüllt (obwohl ich nicht sicher bin, wie oft Sie eine Zahl werden buchstabieren müssen).

Wir haben uns heute drei Skripts angesehen: ***statsfinal.pl***, eine weitere Neuauflage des vertrauten Statistik-Skripts, gab - nach sorgfältigem Maßnehmen - mit Hilfe zweier verschachtelter `for`-Schleifen ein vertikales Histogramm aus. Das zweite, ***zahlenbuchstabierer.pl***, arbeitete mit sehr vielen Bedingungen, um schließlich eine einstellige Zahl zu buchstabieren. Das dritte, ***webseite.pl***, nahm in der Kommandozeile eine Textdatei entgegen, fragte nach ein paar Angaben und konvertierte den Inhalt der Textdatei in eine Webseite.

Herzlichen Glückwunsch, Sie haben die erste Woche dieses Drei-Wochen-Buches und bereits ein mächtiges Stück der Sprache bewältigt. Darauf bauen wir weiter auf. Auf zur Woche 2!

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Listen und Strings manipulieren

Wir beginnen die zweite Woche mit der Verfeinerung Ihres Wissens über Listen, Arrays, Hashes und Strings - und wie Sie die darin enthaltenen Daten manipulieren können. Insbesondere werde ich Ihnen heute etliche Perl-Funktionen vorstellen, die es sich lohnt, im Repertoire zu haben:

Die Themen von heute sind:

- Array- und Hash-Segmente (***Slices***)
- Listen sortieren
- Listen durchsuchen
- Listenelemente hinzufügen oder entfernen
- Listenelemente verändern
- Verschiedene Stringmanipulationen: `reverse`, Substrings finden und extrahieren

Array- und Hash-Segmente

An den Tagen 4 und 5 haben Sie gelernt, wie Sie mit Arrays und Hashes arbeiten können. Perl kennt aber auch Mechanismen zur Manipulation von Teilmengen von Arrays und Hashes. Eine solche Teilmenge eines Arrays oder Hash, die selbst wieder ein Array oder Hash darstellt, heißt Segment oder ***Slice*** (Scheibe, Schnitte).

Der Zugriff auf ein Array-Segment ist dem auf ein einzelnes Element sehr ähnlich, auch auf ein ***Slice*** beziehen Sie sich mit eckigen Klammern. Es gibt jedoch zwei signifikante Unterschiede:

```
@array = (1,2,3,4,5);
$ein_wert = $array[0];   # $ein_wert ist 1
@slice = @array[0,1,2];   # @slice ist (1,2,3)
```

Sehen Sie die Unterschiede? Bei der Elementzugriffssyntax `$array[0]` nutzen Sie das Präfix `$` und eine Indexnummer in rechteckigen Klammern für den Zugriff auf ein Element in der Arrayvariablen. Das Ergebnis speichern Sie in der Skalarvariablen `$ein_wert`. In der Segmentschreibweise setzen Sie ein `@` an den Anfang des Arrayvariablennamens und eine Liste von Indizes in die Klammern. Das Ergebnis ist im Beispiel eine dreielementige Liste, die Sie im Array `@slice` speichern.

Sie können in die Klammern für ein Segment jeden beliebigen Ausdruck stellen, solange er nur zu einer Liste von gültigen Indizes evaluiert (das heißt: ausgewertet) wird. Der Bereichsoperator zum Beispiel eignet sich besonders gut für Segmente aus aufeinanderfolgenden Elementen:

```
@monate = qw (januar, februar, maerz, april, mai, juni, juli,
              august, september, oktober, november, dezember);
@viertes_quartal = @monate[9..11];
```

Beachten Sie, dass Sie nicht aus Versehen ein Array-Segment mit einem einzigen Index verwenden, wenn Sie eigentlich nur auf ein einzelnes Element zugreifen wollen:

```
$ein_wert = @array[5];
```

Diese Schreibweise ist ein häufiger Fehler. Sie ziehen hier nämlich eine einelementige Liste aus dem Array `@array` und versuchen sie dann in skalarem Kontext auszuwerten. Gegebenenfalls machen eingeschaltete Warnungen Sie darauf aufmerksam.

Hash-Segmente sind den Array-Segmenten sehr ähnlich, erfordern aber eine andere Syntax.

```
%hashslice = @hash{'dies','das','anderes'};
```

Die Schlüssel, die Sie extrahieren wollen, werden in den geschweiften Klassen aufgelistet, wie einzelne Hash-Schlüssel, aber ein Array-Variablensymbol wird vorangestellt. Der Grund ist, dass das Ergebnis eine Liste von Schlüsseln und Werten ist (in der korrekten Reihenfolge: Schlüssel, Wert, Schlüssel, Wert und so weiter). Durch Zuweisung an eine Hash-Variable kann man dieses Segment wieder in ein Hash verwandeln. Was Sie auf keinen Fall schreiben sollten, ist:

```
%hashslice = %hash{'dies','das','anderes'};
```

Diese Zeile würde (selbst ohne aktive Warnungen) zu einer Fehlermeldung führen. Auch Hash-Segmente sind Arrays. Allgemein gilt: Um aus einem Array oder Hash einen Skalar zu extrahieren, verwenden Sie die Skalarschreibweise ($), um ein Segment zu extrahieren, verwenden Sie die Arrayschreibweise (@).

Listen sortieren

Sie haben schon gesehen, wie man mit der Funktion `sort` eine Liste sortiert, zum Beispiel die Schlüssel in einem Hash:

```
@keys = sort keys %hash
```

Standardmäßig sortiert `sort` eine Liste in ASCII-Reihenfolge. Um eine Liste aus Zahlen in numerischer Reihenfolge zu sortieren, müssen Sie zwischen `sort` und der zu sortierenden Liste einen zusätzlichen Ausdruck einfügen:

```
@keys = sort { $a <=> $b } keys %hash;
```

Was macht dieser Extraausdruck? Wofür stehen `$a` und `$b`? In den vorangehenden Kapiteln habe ich Ihnen gesagt, dass Sie diese Zeile einfach auswendig lernen sollten. Jetzt aber erkläre ich, was das alles bedeutet.

Der Teil in den geschweiften Klammern legt fest, wie `sort` die Liste sortieren soll. Genauer gesagt vergleicht er zwei Elemente und teilt `sort` das Ergebnis mit: -1, wenn das erste Element kleiner ist als das zweite, 0, wenn beide Elemente gleich sind, und +1, wenn das erste größer ist als das zweite ist.

Diesen Vergleich können in Perl zwei Operatoren erledigen: `<=>` und `cmp`. Warum zwei Operatoren? Aus dem gleichen Grund, aus dem es zwei Sätze von Gleichheits- und Vergleichsoperatoren gibt: einen für Strings (`cmp`) und einen für Zahlen (`<=>`).

Der Operator `cmp` arbeitet auf Strings. Er gibt -1, 0 oder 1 zurück, je nachdem, ob der erste Operand (in ASCII-Hinsicht) kleiner als, gleich oder größer als der zweite ist. Wenn Sie Zeilen schinden wollten, könnten Sie `cmp` auch auf folgende Art schreiben, in einem »warum einfach, wenn's auch kompliziert geht«-Wettbewerb hätten Sie damit gute Chancen:

```
$ergebnis = '';
if ($a lt $b ) { $ergebnis = -1; }
elsif ($a gt $b { $ergebnis = 1; }
else { $ergebnis = 0; }
```

Der Operator `<=>`, manchmal wegen seines Aussehens auch Raumschiffoperator (***Spaceship-Operator***) genannt, macht genau das gleiche, nur eben mit Zahlen.

Doch was sind `$a` und `$b`? Es sind temporäre Variablen von `sort` - Sie müssen sie nicht deklarieren, und sie verschwinden von selbst, sobald die Sortierung erledigt ist. Beim Sortieren der Liste enthalten `$a` und `$b` jeweils die zwei Elemente, die gerade miteinander verglichen werden.

Weil `$a` und `$b` spezielle Variablen der `sort`-Routine sind und auf zwei Elemente in der Liste verweisen, seien Sie vorsichtig mit eventuellen anderen Variablen namens `$a` und `$b`. Wenn Sie `$a`

oder `$b` innerhalb Ihrer Sortierungsroutine verändern, kann das zu unerwünschten Ergebnissen führen.

Solange Sie nichts anderes festlegen, verwendet `sort` den `cmp`-Operator für den Wertevergleich. Mit `{ $a <=> $b }` werden die Werte numerisch verglichen und sortiert.

Im übrigen wird standardmäßig aufsteigend sortiert. Wenn Sie absteigend sortieren möchten, vertauschen Sie einfach `$a` und `$b`:

```
@keys = sort { $b <=> $a } keys %hash; #groesste Keys zuerst
```

Mit einer `sort`-Routine und einer `foreach`-Schleife können Sie einen Hash nach Schlüsseln sortiert ausgeben. Etwas komplizierter ist es, die Ausgabe nach den Hash- Werten zu sortieren. Wenn Sie lediglich die Werte brauchen, erstellen Sie mit der Funktion `values` eine Liste dieser Werte und sortieren sie dann. Auf diese Weise verlieren Sie jedoch den Zugriff auf die ursprünglichen Schlüssel. Es gibt keine Möglichkeit mehr, einen Wert wieder seinem Schlüssel zuzuordnen. Was tun? Sie können die Schlüssel in Wertreihenfolge sortieren und dann beides ausgeben, die Schlüssel und ihre Werte:

```
foreach $key (sort {$zeugs{$a} cmp $zeugs{$b}} keys %zeugs) {
   print "$key, $zeugs{$key}\n";
}
```

Hier sortieren wir nicht die Schlüssel (wir vergleichen nicht einfach `$a` und `$b`), sondern die mit den Schlüsseln verbundenen Werte (`$zeugs{$a}` und `$zeugs{$b}`). Das Ergebnis ist eine nach den Werten sortierte Liste der Schlüssel im Hash, über die wir dann mit `foreach` iterieren können.

Suchen

Listen sortieren ist einfach - denn es gibt eine Funktion dafür. Eine Liste hingegen nach einem bestimmten Element oder einem Teil eines Elements zu durchsuchen, ist nicht ganz so einfach, denn - TMTOWTDI[1] - Sie haben definitiv mehr als eine Möglichkeit. Zum Beispiel könnten Sie eine Liste von Strings mit `foreach` durchlaufen und jedes Element mit dem gesuchten String vergleichen, etwa so:

```
chomp($suchwort = <STDIN>);  # wonach wir suchen
foreach $el (@strings) {
   if ($suchwort eq $el) {
      $gefunden = 1;
   }
}
```

Wenn Sie nicht nach ganzen Listenelementen, sondern in den Listenelementen nach einem Teilstring suchen, nutzt Ihnen der `eq`-Operator allerdings herzlich wenig. Hilfreicher ist hier eine der Stringfunktionen, die ich Ihnen am Tag 3 kurz vorgestellt habe - die Funktion `index` durchsucht einen String nach einem Teilstring und gibt die Position zurück, an der der Teilstring zuerst im String vorkommt. Findet sie den Teilstring nicht, liefert sie -1 (Sie erinnern sich, dass Stringpositionen, wie Arrays, bei 0 beginnen):

```
foreach $el (@strings) {
   if ((index $el, $suchwort) >= 0) {   # -1 heisst nicht gefunden
      $gefunden = 1;
   }
}
```

Effizienter und noch leistungsfähiger ist die Suche nach Teilstrings mit Hilfe von Mustern. Pattern Matching, das Arbeiten mit Mustern, werden wir aber erst morgen kennenlernen; halten Sie sich daher nicht allzusehr mit der folgenden Syntax auf.

```
foreach $el (@strings) {
   if ($el =- /$suchwort/) {
      $gefunden = 1;
   }
}
```

Es geht sogar noch kürzer. Die Perl-Funktion `grep` (benannt nach dem gleichnamigen Unix-Tool) dient insbesondere dem Durchsuchen von Listen. Wie viele andere Perl- Features auch benimmt `grep` sich in einem Listenkontext anders als in skalarem Kontext.

Sie übergeben der Funktion `grep` ein Suchkriterium (das kann ein Block oder ein Ausdruck sein) und eine Liste. Im Listenkontext gibt `grep` eine neue Liste der Elemente zurück, für die das Suchkriterium ***wahr*** ergab. Hier zum Beispiel holen wir uns alle Elemente größer 100 aus der Liste `@zahlen`:

```
@hohe = grep { $_ > 100 } @zahlen;
```

Beachten Sie unsere Freundin, die `$_`-Variable. Die Funktion `grep` nimmt jeweils ein Listenelement, weist es `$_` zu und wertet dann den Ausdruck in den geschweiften Klammern in Booleschem Kontext aus. Ist das Ergebnis ***wahr***, erfüllt das jeweilige Element unser Suchkriterium und wird der Ergebnisliste hinzugefügt. Ist das Ergebnis ***falsch,*** geht `grep` zum nächsten Element der Liste `@zahlen` - und macht so weiter, bis sie zum Ende der Liste kommt.

Sie können als Suchkriterium jeden beliebigen Ausdruck oder Block verwenden (er wird dann in Booleschem Kontext ausgewertet), nur sollte `grep` damit etwas anfangen können (nämlich anhand dessen die neue Liste erstellen). Außerdem müssen Sie nach einfachen Ausdrücken ein Komma setzen; nach Blöcken ist das Komma nicht unbedingt notwendig

`Grep` arbeitet auch mit regulären Ausdrücken. In dieses Thema steigen wir zwar erst morgen ein, aber ich gebe Ihnen, sozusagen als kleinen Vorgeschmack, hier schon mal ein Beispiel:

```
@ixe = grep /x/, @worte;
```

Gesucht werden die Zeichen zwischen den Schrägstrichen (hier nur der Buchstabe `x`). In diesem Beispiel speichert `grep` alle Elemente des Arrays `@worte`, die den Buchstaben `x` enthalten, in das Array `@ixe`. Sie können zwischen die Schrägstriche des Patterns (des Suchmusters) auch eine Variable setzen:

```
print "Suchen nach? ";
chomp($suchwort = <STDIN>);
@gefunden = grep /$suchwort/, @worte;
```

Das Pattern kann beliebig viele Zeichen enthalten; `grep` wird genau diese Zeichenfolge in jedem Listenelement suchen. So findet zum Beispiel `/der/` alle Elemente, in denen »der« enthalten ist (also »der«, »deren« und »oder«, nicht aber »Der« oder »Erde«). Wie Sie mit gewissen Sonderzeichen noch ausgeklügeltere Suchkriterien definieren, besprechen wir morgen.

In skalarem Kontext verhält sich `grep` ähnlich wie in einem Listenkontext, nur dass es keine Liste aller gefundenen Elemente zurückgibt, sondern die Zahl der Treffer (0 bedeutet »nichts gefunden«).

Man verwendet die `grep`-Funktion meistens mit Listen oder Arrays, aber nichts hindert Sie daran, sie auch auf Hashes anzuwenden, solange Sie nur Ihr Suchkriterium richtig formulieren. Die folgende Zeile findet zum Beispiel mit Hilfe von `grep` alle Hash-Schlüssel, bei denen entweder der Schlüssel oder der zugehörige Wert größer 100 ist:

```
@hohe_keys = grep { $_ > 100 or $numhash{$_} > 100 } keys %numhash;
```

Ein Beispiel: Mehr Namen

Am Tag 5 hatten wir ein einfaches Beispiel, das Namen (Vor- und Nachnamen) eingelesen, gesplittet und nach Nachnamen aufgeschlüsselt in einem Hash gespeichert hat. Am Tag 6 haben wir die Namen mit dem Zeileneingabeoperator <> aus einer externen Datei in ein Hash gelesen. Heute wollen wir das Namensskript weiter ausbauen und fügen eine große `while`-Schleife hinzu, die dem Benutzer vier Optionen anbietet:

- Namensliste nach Nachnamen sortieren und ausgeben
- Namensliste nach Vornamen sortieren und ausgeben
- Vor- oder Nachname suchen (exakte Übereinstimmung)
- Programm beenden

Wenn Sie eine der ersten drei Optionen auswählen, führt das Programm den entsprechenden Befehl aus und zeigt Ihnen danach erneut das Menü an, so dass Sie eine andere Option wählen können. Nur die vierte Option beendet das Programm. Hier ein Beispiel, wie diese neue Version, ***mehrnamen.pl,*** auf dem Bildschirm aussehen könnte:

```
% mehrnamen.pl namen.txt
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen (exakte Suche)
4. Programm beenden
Geben Sie eine Zahl ein: 1
Burroughs, William S.
Salinger, J. D.
Sartre, Jean Paul
Schiller, Friedrich
Shakespeare, William
Vonnegut, Kurt
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
4. Beenden
Geben Sie eine Zahl ein: 2
Friedrich Schiller
J. D.  Salinger
Jean Paul  Sartre
Kurt Vonnegut
William Shakespeare
William S.  Burroughs
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
4. Beenden
Geben Sie eine Zahl ein: 3
Wonach suchen? Will
gefundene Namen:
   William S.  Burroughs
   William Shakespeare
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
4. Beenden
Geben Sie eine Zahl ein: 4
%
```

Listing 8.1 zeigt den Code für unser und Such- und Sortierskript:

Listing 8.1: mehrnamen.pl

```
1:  #!/usr/bin/perl -w
2:
3:  %names = ();                    # Hash mit Namen
4:  @raw = ();                      # die einzelnen Wörter
5:  $fn = "";                       # Vorname
6:  $exit = 1;                      # Programm verlassen?
7:  $in = '';                       # temp Input
8:  @keys = ();                     # temp Treffer-Keys
9:  @n = ();                        # temp Name
10: $search = '';                   # wonach gesucht wird
11:
12: while (<>) {
13:     chomp;
14:     @raw = split(" ", $_);
15:     if ($#raw == 1) {  # Normalfall
16:         $names{$raw[1]} = $raw[0];
17:     } else {  # erstelle den Vornamen
18:         $fn = "";
19:         for ($i = 0; $i < $#raw; $i++) {
20:             $fn .= $raw[$i] . " ";
21:         }
22:         $names{$raw[$#raw]} = $fn;
23:     }
24: }
```

```
25:
26: while ($exit) {
27:
28:     print "\n1. Namensliste nach Nachnamen sortieren\n";
29:     print "2. Namensliste nach Vornamen sortieren\n";
30:     print "3. Namen suchen\n";
31:     print "4. Beenden\n\n";
32:     print "Geben Sie eine Zahl ein: ";
33:
34:     chomp($in = <STDIN>);
35:
36:     if ($in eq '1') {       # nach Nachnamen sortieren und ausgeben
37:
38:         foreach $name (sort keys %names) {
39:             print "$name, $names{$name}\n";
40:         }
41:
42:     } elsif ($in eq '2') {   # nach Vornamen sortieren und ausgeben
43:
44:         @keys = sort { $names{$a} cmp $names{$b} } keys %names;
45:         foreach $name (@keys) {
46:             print "$names{$name} $name\n";
47:         }
48:
49:     } elsif ($in eq '3') {      # Namen finden (1 oder mehr)
50:
51:         print "Wonach suchen? ";
52:         chomp($search = <STDIN>);
53:
54:         @keys = ();  # @keys fuer Search zuruecksetzen
55:         while (@n = each %names) {
56:             if (grep /$search/, @n) {
57:                 $keys[++$#keys] = $n[0];
58:             }
59:         }
60:         if (@keys) {
61:             print "gefundene Namen: \n";
62:             foreach $name (sort @keys) {
63:                 print "   $names{$name} $name\n";
64:             }
65:         } else {
66:             print "Nichts gefunden.\n";
67:         }
68:
69:    } elsif ($in eq '4') {      # Ende
70:       $exit = 0;
71:    } else {
72:       print "Eingabefehler.  1 bis 4 bitte.\n";
73:    }
74: }
```

Den Rahmen dieses Skripts bildet eine große `while`-Schleife, die sich um die Optionen 1 bis 4 kümmert. Nachdem sie die Auswahlmöglichkeiten ausgegeben und um die Eingabe der entsprechenden Zahl gebeten hat, enthält die `while`-Schleife ein paar Überprüfungen der Eingabe. Wenn die Eingabe nicht 1, 2 , 3 oder 4 war, übernimmt das letzte `else` in Zeile 71 und beginnt von vorne (da wir auf diese Zahlen mit einem Stringvergleich überprüfen, würde ein versehentlicher Buchstabe keine Warnungen produzieren).

Viel interessanter ist, was bei jeder einzelnen Option passiert. Die beiden Sortieroptionen (Option 1 und 2 in Zeile 36 bzw. 42) verwenden jeweils eine unterschiedliche Form der `sort`-Funktion, die Sie in dieser Lektion bereits kennengelernt haben. Die dritte Option, die in den Namen nach dem eingegebenen String sucht, arbeitet mit `grep`.

Betrachten wir zuerst die beiden Sortieroptionen. Unser Namens-Hash ist nach Nachnamen aufgeschlüsselt, also ist das Sortieren nach Nachnamen ganz einfach. Tatsächlich ist die `foreach`-Schleife in Zeile 38 bis 40 genau dieselbe wie in den bisherigen Versionen dieses Beispiels.

Den Hash nach Vornamen zu sortieren (die zweite Option) ist da schon schwieriger. Doch im Abschnitt »Sortieren« habe ich Ihnen ja gerade gezeigt, wie man das macht - die dort gezeigte Technik können Sie hier in Zeile 44 bis 47 anwenden. Sie erstellen mit einer `sort`-Routine, die nach Werten sortiert, eine temporäre Liste der Schlüssel, durchlaufen diese mit `foreach` und geben die Namen in der richtigen Reihenfolge aus.

Womit wir bei der Suche wären. In Zeile 51 und 52 fragen wir, wonach gesucht werden soll. Diesen Suchstring speichern wir in der Variablen `$search`.

Auf den ersten Blick denken Sie vielleicht, diese Suche wäre ganz einfach - nur `grep` und das Pattern `/$search/`. Der Haken ist aber, dass wir in einem Hash sowohl die Schlüssel als auch die Werte durchsuchen müssen.

Wollten wir nur in den Nachnamen suchen, könnten wir einfach mit Hilfe der `keys`- Funktion die Schlüssel durchsuchen, etwa so:

```
@treffer = grep /$search/, keys %names;
```

Um an die Werte zu kommen, könnten wir mit einer `foreach`-Schleife die Schlüssel durchlaufen und jeden Schlüssel und seinen Wert nach dem gefragten String durchsuchen. Dieser Ansatz wäre nicht falsch. Aber in diesem besonderen Beispiel wollte ich `grep` verwenden, deswegen habe ich einen vielleicht ungewöhnlich anmutenden Weg gewählt (ich nenne es lieber einen kreativen Weg): Ich nehme die `each`-Funktion, die, wie Sie am Tag 5 gelernt haben, mir eine Liste aus Schlüssel/ Wert-Paaren liefert. Dann verwende ich `grep` mit dieser Liste und speichere die Schlüssel für später.

Das mache ich in Zeile 55 bis 59. Schauen wir uns die Zeilen noch mal genauer an:

```
55:         while (@n = each %names) {
56:             if (grep /$search/, @n) {
57:                 $keys[++$#keys] = $n[0];
58:             }
59:         }
```

Von `each` erhalten wir eine zweielementige Liste mit einem Schlüssel/Wert-Paar. Rufen wir die `each`-Funktion mehrmals auf, arbeitet sie sich durch alle Schlüssel und Werte im Hash. Die `while`-Schleife in Zeile 55 wird so oft durchlaufen, wie Schlüssel/ Wert-Paare im Hash sind, wobei jedes Paar unserem temporären Namensarray `@n` zugewiesen wird. Wenn kein Paar mehr übrig ist, gibt `each` die leere Liste `()` zurück, `@n` evaluiert zu ***falsch***, und die `while`-Schleife stoppt.

Innerhalb der `while`-Schleife überprüfen wir mit einer `if`-Bedingung, `grep` und dem Pattern `/$search/`, ob der Suchstring in einem der beiden Elemente von `@n` vorkommt. Wir verwenden `grep` hier in skalarem Kontext, deswegen liefert `grep` uns die Anzahl der Treffer. Wenn `grep` nichts finden konnte, gibt es 0 zurück, und wir fahren fort mit dem nächsten `while`-Durchlauf. Wenn es aber etwas gefunden hat, liefert es eine Zahl ungleich Null, unsere Bedingung ist erfüllt, und wir machen mit Zeile 57 weiter.

In Zeile 57 holen wir uns den in `$n[0]` gespeicherten Schlüssel und hängen ihn ans Ende des Arrays `@keys` an (wenn wir den Schlüssel haben, haben wir auch Zugriff auf den Wert, um den brauchen wir uns also keine Sorgen zu machen). Sie erinnern sich, dass `$#keys` uns den höchsten Index im Array liefert, `++$#keys` sich also auf die Position nach der letzten Position bezieht. Beachten Sie die Präfixschreibweise - so können wir den Index inkrementieren, bevor wir dem Array an dieser Stelle etwas zuweisen. Das machen wir erst danach: Wir fügen dem Array als neues letztes Element unseren Schlüssel hinzu.

Puh! Diese Zeile ist ein Beispiel dafür, wieviel Information man in eine einzige Zeile stopfen kann. Zum Glück gibt es einen viel leichteren Weg, ein Element am Ende einer Liste anzufügen: die Funktion `push`, mit der wir uns später in dieser Lektion noch befassen.

Nachdem die Liste mit den Trefferschlüsseln erstellt ist, müssen wir sie nur noch ausgeben. In diesem Fall überprüfen wir erst, ob `@keys` überhaupt etwas enthält (Zeile 60), und wenn dem so ist, sortieren wir es und geben die Namen aus. Anderenfalls teilt eine kleine Meldung mit, dass nichts gefunden wurde (Zeile 66).

Listenelemente hinzufügen oder entfernen

Sie können Listen-, Array- und Hash-Elemente immer mit den Methoden hinzufügen und entfernen, die ich an den Tagen 4 und 5 beschrieben habe. Aber in manchen Situationen sind diese Standardwege zum Hinzufügen und Entfernen von Elementen etwas unbequem, schlecht zu lesen oder uneffizient. Hilfsbereit wie immer, stellt Perl Ihnen einige Funktionen zu Verfügung, die Ihnen die Arbeit mit Listen erleichtern werden. Zu diesen Funktionen

gehören:

- `push` und `pop`: ein Element am Ende der Liste hinzufügen und entfernen
- `shift` und `unshift`: ein Element am Anfang der Liste hinzufügen und entfernen
- `splice`: ein Element irgendwo in der Liste hinzufügen oder entfernen

push und *pop*

Mit den Funktionen `push` und `pop` kann man Elemente am Ende einer Liste hinzufügen oder entfernen. Das heißt, sie betreffen die höchsten Indexpositionen dieser Liste. Wenn Sie jemals mit S***tacks*** (Stapeln) gearbeitet haben und Ihnen die »***Last In First Out***«-Idee ein Begriff ist,[2] werden Sie `push` und `pop` kennen. Hatten Sie noch nie mit S***tacks*** zu tun, stellen Sie sich einen Stapel Teller vor. Wenn Sie einen Teller auf diesen Stapel legen, ist das auch der Teller, den Sie als erstes wieder herunternehmen (Sie werden wahrscheinlich nicht irgendeinen aus der Mitte herausziehen). ***Stack*** ist englisch und heißt nichts anderes als Stapel - `push` entspricht »Teller rauf« und `pop` »Teller runter«. Ganz einfach. Was beim Tellerstapel oben ist, ist bei Arrays das Ende, das heißt `push` fügt ein Element am Ende des Arrays an, und `pop` nimmt dieses Element wieder weg.

Die Funktion `push` braucht zwei Argumente: eine Liste, die verändert, und eine Liste von Elementen, die angefügt werden soll. Die Originalliste wird an Ort und Stelle verändert und `push` gibt die neue Zahl der Elemente in der veränderten Liste zurück (sozusagen, wie viele Teller jetzt in dem Stapel sind). So haben wir zum Beispiel im vorigen Abschnitt diesen häßlichen Code gebraucht, um dem Array `@keys` einen Schlüssel hinzuzufügen:

```
$keys[++$#keys] = $n[0];
```

Genau das gleiche könnten wir mit `push` so schreiben:

```
push @keys, $n[0];
```

Weniger Zeichen und leichter zu verstehen. Beachten Sie, dass das zweite Argument hier ein Skalar ist (der dann in eine einelementige Liste konvertiert wird); es könnte auch eine Liste sein. So können Sie mehrere Listen ganz einfach zusammenfügen:

```
push @ergebnis_liste, @liste1;
```

Die Funktion `pop` macht das Gegenteil von `push`: Sie entfernt das letzte Element aus der Liste oder dem Array. Der `pop`-Funktion muss man nur ein Argument übergeben, nämlich die zu verändernde Liste. `Pop` entfernt das letzte Element und gibt dieses Element zurück.

```
$letztes = pop @array;
```

Wie bei `push` wird die Liste an Ort und Stelle verändert und hat nach dem `pop` ein Element weniger. Ein Weg, alle Elemente von einem Array in ein anderes zu verschieben, könnte zum Beispiel so aussehen:

```
while (@alt) {
   push @neu, pop @alt;
}
```

Beachten Sie, dass mit diesem Beispiel das neue Array genau die umgekehrte Reihenfolge hätte, denn das letzte Element im alten Array ist das erste, das ins neue Array verschoben wird. Zum Umkehren einer Liste eignet sich allerdings die Funktion `reverse` weit besser, und nach wie vor verschieben Sie eine Liste am einfachsten mit einer Zuweisung. Sie könnten obigen Code aber zum Beispiel verwenden, wenn Sie mit jedem Element noch etwas machen wollen, während sie es von einem Array in das andere schieben:

```
while (@alt) {
   push @neu, 2 * pop @alt;
}
```

shift und *unshift*

Wenn `push` und `pop` Elemente am Listenende hinzufügen oder entfernen, wäre es doch nett, auch Funktionen zu haben, die das gleiche am Anfang der Liste machen. Kein Problem! Das erledigen `shift` und `unshift` (von englisch ***shift***: verschieben, wegrücken, ändern) sie schieben alle Elemente eine Position hoch oder runter (oder wie man sich das bei Arrays eher vorstellt, nach rechts oder links). Wie bei `push` und `pop` wird die Liste direkt geändert.

Die Funktion `shift` nimmt ein Argument entgegen, und zwar eine Liste. Sie entfernt das erste Element dieser Liste und schiebt alle anderen Elemente eine Position nach unten (links). Zurück gibt sie das entfernte Element. Auch das folgende Beispiel verschiebt Elemente von einem Array in ein anderes, jedoch wird die Reihenfolge hier nicht umgekehrt:

```
while (@alt) {
   push @neu, shift @alt;
}
```

Die Funktion `unshift` ist das Listenanfangspendant zu `push`. Man übergibt Ihr zwei Argumente: die zu ändernde Liste und die anzufügende Liste. Die geänderte Liste enthält dann die zugefügten Elemente, gefolgt von den alten Listenelementen, die soweit nach oben (oder rechts) verschoben wurden, wie Elemente hinzugefügt worden sind. Zurück gibt `unshift` die Anzahl der Elemente in der neuen Liste:

```
$anzahl_user = unshift @user, @neue_user;
```

splice

`push` und `pop` verändern Elemente am Ende einer Liste, `shift` und `unshift` Elemente am Anfang. Die Funktion `splice` (spleissen, verbinden) ist ein Allzweckmittel, wenn Sie irgendwo in der Liste Elemente hinzufügen, entfernen oder ersetzen wollen. Sie übergeben `splice` vier Argumente:

- das zu ändernde Array
- den Offset, die Position im Array, ab der Elemente hinzugefügt oder entfernt werden
- die Anzahl zu entfernender oder zu ersetzender Elemente. Wenn dieses Argument nicht enthalten ist, ändert `splice` jedes Element vom Offset an
- die Listenelemente, die dem Array hinzugefügt werden, wenn vorhanden

Der Offset ist genaugenommen der Abstand zwischen dem Anfang eines Arrays oder Strings und einer bestimmten Position. Er gibt an, wie viele Elemente oder Zeichen Sie nach rechts gehen müssen, um von der ersten zur gewünschten Position zu gelangen - mit einem Offset 0 bleiben Sie am Anfang. Da Arrays und Strings ebenfalls bei 0 beginnen, bedeutet Offset letztlich nichts anderes als Index der Zielposition und umgekehrt.[3]

Betrachten wir zuerst, wie man mit `splice` Elemente aus einem Array entfernt. Bei jedem der folgenden Beispiele gehen wir von einer Liste mit zehn Elementen, den Zahlen von 0 bis 9, aus:

```
@zahlen = 0 .. 9;
```

Zum Entfernen der Elemente 5, 6 und 7 nimmt man als Offset 5 und als Länge 3. Ein Listenargument gibt es nicht, weil wir ja nichts hinzufügen:

```
splice(@zahlen, 5, 3);  # ergibt (0,1,2,3,4,8,9)
```

Um alle Elemente ab Position 5 (also bis zum Listenende) zu entfernen, lassen Sie einfach das Längenargument weg:

```
splice(@zahlen, 5);  # ergibt (0,1,2,3,4)
```

Wollen Sie Elemente ersetzen, übergeben Sie `splice` als viertes Argument eine Liste mit den einzufügenden Elementen. Hier zum Beispiel ersetzen Sie die Elemente 5, 6 und 7 durch die Elemente »fuenf«, »sechs« und

»sieben«:

```
splice(@zahlen, 5, 3, qw(fuenf sechs sieben));
                        # ergibt (0,1,2,3,4,fuenf,sechs,sieben,8,9)
```

Ob Sie genauso viele Elemente entfernen, wie Sie hinzufügen, kümmert Perl nicht. Es fügt einfach die Elemente der übergebenen Liste an der durch den Offset festgelegten Position ein und rückt alle anderen Elemente entsprechend zurecht. Im folgenden Beispiel löscht `splice` die Elemente 5, 6 und 7 und setzt den String »nicht vorhanden« an ihre Stelle:

```
splice(@zahlen, 5, 3, "nicht vorhanden");
                        # ergibt (0,1,2,3,4,"nicht vorhanden",8,9)
```

Das Array hat dann nur noch acht Elemente.

Um dem Array Elemente hinzuzufügen, ohne welche zu entfernen, übergeben Sie als Länge 0. Hier zum Beispiel fügen wir nach Element 5 ein paar Zahlen ein:

```
splice(@zahlen, 5, 0, (5.1, 5.2, 5.3));
                       # ergibt  (0,1,2,3,4,5,5.1,5.2,5.3,6,7,8,9)
```

Wie die anderen Funktionen zur Manipulation von Listen verändert die Funktion `splice` die Liste direkt und gibt eine Liste der entfernten Elemente zurück. Diese Eigenschaft können Sie ausnutzen, um eine Liste in mehrere Teile zu zerlegen. Sagen wir zum Beispiel, Sie hätten eine Liste `@gesamtliste`, die Sie in zwei Listen `@liste1` und `@liste2` zerlegen möchten. Das erste Element in `@gesamtliste` ist die Anzahl der Elemente, die Sie in die `@liste1` verschieben wollen, alle übrigen Elemente sollen in `@liste2`. Sie könnten das mit den folgenden drei Zeilen lösen. Die erste Zeile entfernt ab Position 1 so viele Elemente aus `@gesamtliste`, wie das Element 0 festlegt, und weist sie `@liste1` zu. Die zweite entfernt das Element 0, und die dritte speichert die verbleibenden Elemente in `@liste2`:

```
@liste1 = splice(@gesamtliste, 1, $gesamtliste[0]);
shift @gesamtliste;
@liste2 = @gesamtliste;
```

Weitere Möglichkeiten zur Listenmanipulation

Warten Sie, das war noch nicht alles. Ich habe noch mehr Funktionen, die beim Manipulieren von Listen ganz nützlich sein können: `reverse`, `join` und `map`.

reverse

Die Funktion `reverse` nimmt jedes Element einer Liste und färbt es blau. Kleiner Scherz. In Wahrheit kehrt `reverse` die Reihenfolge der Listenelemente um:

```
@andersrum = reverse @liste;
```

Wie bereits gesagt können Sie die Reihenfolge statt mit `reverse` auch mit anderen Methoden wie `shift`, `unshift`, `push`, `pop` oder bloßem Umsortieren umkehren (aber versuchen Sie, Arrayelemente so wenig wie möglich hin- und herzuschieben, der Überblick geht schnell verloren).

Die `reverse`-Funktion arbeitet auch reibungslos mit Skalaren. In diesem Fall kehrt es die Reihenfolge der Zeichen im String um. Wir kommen gleich im Abschnitt »Strings manipulieren« darauf zurück.

join

Die Funktion `split` spaltet einen String in mehrere Listenelemente auf. Die Funktion `join` (verbinden, vereinigen) tut das Gegenteil: `join` fügt Listenelemente zu einem einzigen String zusammen, wobei Sie als Trennzeichen eine beliebige Zeichenfolge festlegen können. Wenn Sie zum Beispiel eine Zahlenfolge mit `split` in eine Liste zerlegt haben:

```
@liste = split(' ', "1 2 3 4 5 6 7 8 9 10";
```

dann könnten Sie sie mit `join` wieder zurück in einen String verwandeln, in dem die Zahlen durch Leerzeichen voneinander getrennt sind:

```
$string = join(' ',@liste);
```

Oder Sie könnten die Elemente durch Pluszeichen voneinander trennen:

```
$string = join('+',@liste);
```

Oder Sie trennen sie gar nicht:

```
$string = join('',@liste);
```

map

Sie wissen bereits, wie man mit `grep` die Elemente einer Liste, die ein bestimmtes Suchkriterium erfüllen, in einer anderen Liste speichert. Die Funktion `map` arbeitet ganz ähnlich, nur dass `map` nicht bestimme Elemente aus einer Liste auswählt, sondern einen gegebenen Ausdruck oder Ausdrucksblock auf jedes Listenelement anwendet und alle Ergebnisse in einer neuen Liste sammelt.

Sagen wir zum Beispiel, Sie haben eine Liste aus Zahlen von 1 bis 10 und Sie möchten eine Liste mit den Quadraten all dieser Zahlen erstellen. Sie könnten dafür eine `foreach`-Schleife und `push` nehmen:

```
foreach $zahl (1 .. 10) {
   push @quadrate, ($zahl*$zahl);
}
```

Derselbe Vorgang würde mit `map` so aussehen:

```
@zahlen = (1..10);
@quadrate = map { $_**2 } @zahlen;
```

Wie `grep` schnappt sich `map` nacheinander die Elemente der Liste und weist diese nacheinander `$_` zu. Anders als `grep` wertet es den Ausdruck dann nicht in Booleschem, sondern in Listenkontext aus und speichert jedes Ergebnis in der neuen Liste - auch Null, auch Werte, die anders sind als das ursprüngliche Listenelement in `$_` und sogar mehrere Elemente gleichzeitig. Stellen Sie sich `map` als einen Filter für jedes Listenelement vor, wobei das Ergebnis des Filters sein kann, was immer Sie möchten.[4]

Der »Filter«-Teil von `map` kann entweder ein einzelner Ausdruck oder ein ganzer Block sein (nach einem einfachen Ausdruck müssen Sie ein Komma setzen, nach einem Block nicht unbedingt). Bei einem einfachen Ausdruck gibt `map` für jedes Listenelement das Ergebnis dieses Ausdrucks zurück. Bei einem Block liefert es das Ergebnis des zuletzt ausgewerteten Ausdrucks im Block - sorgen Sie also dafür, dass die letzte Evaluierung im Block das gewünschte Ergebnis hat.[5] Im folgenden Beispiel ersetzt `map` alle negativen Zahlen durch 0 und jede 5 durch die Zahlen 2 und 3. Ich habe das ganze etwas leichter lesbar formatiert. Weder bei `map` noch bei `grep` muss der gesamte Code in einer Zeile stehen:

```
@neue_zahlen = map {
               if ($_ < 0) {
                   0;
               } elsif ($_ == 5) {
                   (3,2);
               } else { $_; }
} @zahlen;
```

Strings manipulieren

In dieser Lektion ging es bisher vornehmlich um die Manipulation von Listen und Listeninhalten mittels verschiedener Perl-Funktionen. Perl hat aber auch einige sehr nützliche Funktionen zum Manipulieren von Strings

(ein paar davon habe ich bereits am Tag 2 vorgestellt). In diesem Abschnitt wollen wir uns einige dieser Funktionen, `reverse`, `index`, `rindex` und `substr`, genauer ansehen.

Jede dieser Funktionen dient der Veränderung von Strings. In vielen Fällen sind andere Methoden wahrscheinlich geeigneter - der Punktoperator (`.`) zum Verketten, Variablenwerte und -interpolation zum Erstellen oder Patterns zum Suchen und Durchsuchen von Strings. Doch in manchen Fällen sind die folgenden Funktionen zumindest einfacher zu handhaben, insbesondere wenn Sie ähnliche Funktionen zur Stringmanipulation von anderen Sprachen her gewohnt sind.

reverse

Sie kennen die `reverse`-Funktion schon im Zusammenhang mit Listen. In einem Listenkontext kehrt sie die Reihenfolge der Elemente in der Liste um. In skalarem Kontext verhält `reverse` sich anders: Hier wird die Reihenfolge der Zeichen in einer Zahl oder einem String umgekehrt.

> *Aus dem String »NIE GRUB RAMSES MARBURG EIN« macht `reverse` also »NIE GRUBRAM SESMAR BURG EIN«. Beachten Sie, dass die Reihenfolge **aller** Zeichen umgekehrt wird und ein eventueller Zeilenvorschub am Ende eines Strings in seiner Umkehrung am Anfang steht. Wenn Sie diesen Effekt vermeiden wollen, verwenden Sie vorher `chomp`.*

Der Unterschied zwischen `reverse` in Skalar- und Listenkontext kann manchmal etwas verwirren. Nehmen Sie zum Beispiel folgenden Code:

```
foreach $string (@liste) {
   push @andersrum, reverse $string;
}
```

Auf den ersten Blick sieht das Beispiel so aus, als nehme es alle Stringelemente aus dem Array `@liste`, kehre jedes einzelne um und schiebe das Ergebnis in das Array `@andersrum`. Wenn Sie den Code jedoch ausführen, enthält `@andersrum` exakt dasselbe wie `@liste`. Warum? Weil das zweite Argument der Funktion `push` eine Liste zu sein hat, wird `reverse` hier im Listen- statt im skalaren Kontext aufgerufen. Der String in `$string` wird folglich als eine einelementige Liste interpretiert, die auch rückwärts nur dieses eine Element enthält. Die Zeichen innerhalb dieses Elements bleiben unberührt. Dieses »Mißverständnis« kann die `scalar`-Funktion klären:

```
foreach $string (@liste) {
   push @andersrum, scalar (reverse $string);
}
```

index und *rindex*

Mit den Funktionen `index` und `rindex` suchen Sie in Strings nach Teilstrings. Wenn Sie ihnen zwei Strings übergeben (einen zu durchsuchenden und einen zu suchenden), liefern sie die Position des zweiten Strings im ersten zurück oder, wenn er nicht gefunden wurde, -1. Positionen beginnen auch in Strings bei 0.[6] Anders als bei Arrays bezeichnen sie aber die Stellen ***zwischen*** den Zeichen, das heißt Position 0 ist die Stelle unmittelbar vor dem ersten Zeichen des Strings. Sie können mit `index` oder `rindex` `grep`-ähnlichen Code wie diesen hier schreiben:

```
foreach $str (@liste) {
   if ((index $str, $suchwort) != -1) {
      push @treffer, $str;
}
```

Der Unterschied zwischen `index` und `rindex` sind der Startpunkt und die Richtung der Suche. Die Funktion `index` beginnt ihre Suche am Anfang des Strings und findet die Position, an der der gesuchte Teilstring das erste Mal auftaucht. `rindex` hingegen »rollt den String von hinten auf«: Es beginnt am Ende und findet die letzte Position des Teilstrings.

Beiden Funktionen können Sie ein optionales drittes Argument übergeben, das die Position festlegt, an der die Suche starten soll. Wenn Sie zum Beispiel mit `index` bereits etwas gefunden haben, können Sie `index` noch einmal aufrufen und da anfangen, wo Sie aufgehört haben.

substr

Die Funktion `substr` (von Substring) extrahiert einen Teilstring aus einem String und gibt ihn zurück. Sie rufen `substr` mit folgenden drei Argumenten auf:

- Dem String, auf den Sie sich beziehen
- Dem Offset, ab dem die Extraktion beginnen soll. Wenn Sie hier eine negative Zahl übergeben, zählt `substr` die Positionen vom Ende ab.
- Der Länge des zu extrahierenden Teilstrings. Wenn Sie keine Länge angeben, werden alle Zeichen vom Offset bis zum Ende des Strings extrahiert.

Der Originalstring wird von `substr` nicht verändert.

Sie verstehen nur Bahnhof? Dann extrahieren wir die ersten vier Zeichen aus dem String »Bahnhof« und speichern sie in `$teilstring`:

```
$string = "bahnhof";
$teilstring= substr($string, 0, 4);  #$teilstring ist "bahn"
```

Oder die Zeichen 5 und 6:

```
$teilstring= substr($string, 4, 2);  #$teilstring ist "ho"
$teilstring= substr($string, -3, 2);  #$teilstring ist "ho"
```

Das Besondere an `substr` ist aber, dass Sie es auch auf die linke Seite einer Zuweisung stellen können. Dann extrahiert es keine Teilstrings, sondern macht geradezu das Gegenteil: Auf der linken Seite einer Zuweisung beziehen sich die Argumente (String, Offset, Länge des Teilstrings) nicht auf das Extrahieren, sondern das Ersetzen von Teilstrings. Der String auf der rechten Seite kann größer oder kleiner als der zu ersetzende sein, das ist Perl ganz egal:

```
substr($string, 4, 3) = "fahrkarten"; #$string ist "bahnfahrkarten"
substr($string, -1, 1) = "";          #$string ist "bahnfahrkarte"
substr($string, 0, 13) = "$teilstring "; #$string ist "ho "
```

Möchten Sie einen Teilstring überall in einem String suchen und ersetzen, könnten Sie zum Beispiel eine `while`-Schleife, `index` und `substr` verwenden:

```
$str = "Dies ist ein Test-String. Den wir veraendern wollen.";
$pos = 0;
$teilstr = "i";
$ersatz = "*";
while ($pos < (length $str) and $pos != -1) {
    $pos = index($str, $teilstr, $pos);
    if ($pos != -1 ) {
        substr($str,$pos,length $teilstr) = $ersatz;
        $pos++;
    }
}
```

Klammern Sie sich aber nicht an diesen Code: Für solche Operationen gibt es bessere und kürzere Wege. Insbesondere reguläre Ausdrücke komprimieren diese gesamte Schleife in einzige Zeile:

```
$str =~ s/$teilstr/$ersatz/g;
```

Vertiefung

In dieser Lektion habe ich Ihnen viele recht gebräuchliche Funktionen zur String- und Listenmanipulation

vorgestellt. Von diesen wiederum habe ich nur die gebräuchlicheren Anwendungen beschrieben. Wenn Sie Interesse an den Details dieser (oder der in diesem Buch nicht beschriebenen) Funktionen haben, schauen Sie auch in Anhang A oder die ***perlfunc***-Manpage.

Einige Eigenschaften der Funktionen habe ich vor allem deshalb nicht angesprochen, weil ich Sie dann noch öfter auf später vertrösten müßte, als ich es ohnehin schon tue. Zum Beispiel kann die Sortierroutine für die `sort`-Funktion (der Teil in den geschweiften Klammern) durch den Namen einer Subroutine ersetzt werden. So könnten Sie einmal eine ganz ausgefuchste `sort`-Routine schreiben und diese an verschiedenen Stellen immer wieder verwenden.

Die Funktionen `pop` und `shift` können auch ohne Argumente verwendet werden. Auf welche Liste sie sich dann beziehen, hängt davon ab, wo im Skript sie zum Einsatz kommen: Im Hauptkörper des Skripts verändern `pop` und `shift` ohne Argumente das `@ARGV`-Array (das die für das Skript gedachten Argumente enthält). Innerhalb einer Subroutine wirken `pop` und `shift` sich auf `@_` aus, einer Liste, in der die Argumente der Subroutine landen (und hier mein Verweis auf später: Mit Subroutinen befassen wir uns an Tag 11).

Zusammenfassung

Perl lernen ist mehr als nur die Syntax des Sprachkerns zu lernen. Die Perl- Funktionen tragen viel zur Funktionalität von Perl-Skripts bei. Oft gibt es bessere Wege als diese Funktionen, aber in bestimmten Fällen sind sie wiederum sehr nützlich. Das gilt für alle Funktionen, die Sie heute kennengelernt haben. Das meiste können Sie auch ganz anders machen, aber vielleicht nicht so schnell, effizient oder übersichtlich.

Wir haben heute verschiedene Funktionen zum Manipulieren von Listen und Strings besprochen und wie man sie zur Lösung einfacher Aufgaben einsetzt. Sie haben außerdem etwas über Array- und Hash-Segmente gelernt, mit denen Sie mehrere Elemente auf einmal aus einer Liste oder aus einem Hash herausziehen können.

Die neuen Funktionen waren heute:

- `sort`, sortiert Listen
- `grep`, extrahiert Listenelemente, auf die ein bestimmtes Kriterium zutrifft
- `push` und `pop`, entfernt Elemente vom Ende einer Liste bzw. fügt sie dort hinzu
- `shift` und `unshift`, entfernt Elemente vom Anfang einer Liste bzw. fügt sie dort hinzu
- `splice`, Hinzufügen, Entfernen oder Ersetzen von Elementen an jeder Position einer Liste
- `reverse`, kehrt die Reihenfolge der Elemente in der Liste oder in skalarem Kontext, der Zeichen im String, um
- `join`, das Gegenteil von `split`, fügt Listenelemente zu einem String zusammen
- `map`, führt auf jedem Listenelement eine oder mehrere Anweisungen aus und erstellt eine neue Liste mit den Ergebnissen
- `index` und `rindex`, finden die Position eines Strings in einem anderen String
- `substr`, zum Entfernen, Hinzufügen oder Ersetzen eines Strings durch einen anderen

Fragen und Antworten

Frage:
Die meisten heute beschriebenen Funktionen scheinen nur mit Listen reibungslos zu arbeiten. Kann ich sie auch mit Hashes einsetzen?

Antwort:
Hashes und Listen sind insoweit austauschbar, als ein Hash in seine einzelnen Komponenten aufgeschlüsselt wird, wenn man ihn als Liste verwendet. Die Elemente werden wieder zu Schlüssel-Wert-Paaren, wenn aus der Liste wieder ein Hash wird. Ganz technisch gesehen können Sie also viele der Funktionen auch mit Hashes benutzen. Allerdings kommt dabei vielleicht nicht gerade das heraus, was Sie sich davon versprechen. Zum Beispiel wird `pop` *mit einem Hash diesen Hash in eine Liste umwandeln und dann das letzte Element dieser Liste entfernen (den letzten Key im Hash) - nur wissen Sie ja gar nicht, in welcher Reihenfolge die Keys im Hash stehen; deswegen ist das Ergebnis nicht vorherzusagen. Hashes manipulieren Sie wohl besser mit Hash- Segmenten oder der* `delete`*-Funktion, oder Sie wenden die Funktionen von heute auf Listen aus den Hash-Schlüsseln und -werten an.*

Frage:
Ich will mit `reverse` eine Liste aus Strings umkehren. Aber die Reihenfolge ändert sich nicht; es ist, als hätte ich die Funktion nie aufgerufen.

Antwort:
Sind Sie sicher, dass Sie `reverse` auf einen String anwenden und nicht auf eine einelementige Liste? Eine Liste mit einem einzigen Element ist auch eine Liste, deren Reihenfolge `reverse` Ihnen artig umkehrt - was den String aber nicht verändert. Achten Sie darauf, dass Sie `reverse` in skalarem Kontext aufrufen, oder erzwingen Sie ihn mit der `scalar`-Funktion.

Frage:
In einer der letzten Lektionen haben Sie die Variable `$"` beschrieben, mit der man das Trennzeichen zwischen Listenelementen festlegt, beispielsweise beim Interpolieren einer Liste in einen String. Was ist der Unterschied zwischen dieser Technik und der `join`-Funktion?

Antwort:
Beim Ergebnis gibt es keinen Unterschied. Beide machen aus einer Liste einen String mit Trennzeichen zwischen den Listenelementen. Allerdings ist `join` hier effizienter, weil der fragliche String nicht erst interpoliert werden muss.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist ein Segment (oder ***Slice***)? Wie wirkt sich ein Segment auf die Originalliste aus?
2. Wofür werden die Variablen `$a` und `$b` in `sort`-Routinen verwendet?
3. Was machen die Operatoren `<=>` und `cmp`? Was ist der Unterschied zwischen den beiden?
4. Das erste Argument der Funktion `grep` ist ein Ausdruck oder Block. Wozu dient dieser Block?
5. Wie fügt man mit `splice` einem Array Elemente hinzu? Wie ersetzt man zwei Elemente durch eine Liste aus vier Elementen?
6. Wie gibt man aus dem Block in `map` einen Wert zurück?

Übungen

1. Schreiben Sie folgende Ausdrücke neu, und zwar mit `splice`:

```
  push @liste, 1;
push @liste, (2,3,4);
$list[5] = "foo";
shift @liste;
```

2. FEHLERSUCHE: Was ist falsch an diesem Code (Tipp: Es könnten mehrere Fehler sein)?

```
  while ($i <= $#liste) {
 $str = @novel[$i++];
 push @ergebnis, reverse $str;
}
```

3. FEHLERSUCHE: Und was ist hiermit?

```
  while ($pos < (length $str) and $pos != -1) {
  $pos = index($str, $such, $pos);
  if ($pos != -1 ) {
      $count++;
  }
}
```

4. Schreiben Sie eine neue Version des folgenden Ausdrucks. Verwenden Sie `foreach` und `push`.

```
@liste2 = grep {$_ < 5 } @liste;
```

5. Schreiben Sie ein Skript, das um die Eingabe eines Strings und eines einzelnen Zeichens bittet und dann ausgibt, wie oft dieses Zeichen in dem String vorkommt. Verwenden Sie die Funktion `index`.
6. Schreiben Sie das Skript aus Übung 5 noch einmal, diesmal ohne `index` (oder Patterns, falls Sie sich damit schon auskennen). Tipp: Probieren Sie's mit `grep`.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Ein ***Slice*** oder Segment ist eine Art Teilarray von Elementen aus einem Array oder Hash. Segmente haben keine Auswirkung auf die Originalliste; sie kopieren die ausgewählten Elemente in eine neue Liste, anders als bei Verwendung von `splice`, das die Originalliste (bzw. das Array) dauerhaft verändert.
2. Die Variablen `$a` und `$b` in einer `sort`-Routine sind lokale Variablen dieser Routine und enthalten die Werte der beiden jeweils zu vergleichenden Elemente.
3. Die Operatoren <=> und `cmp` haben zwei Operanden und geben -1 zurück, wenn der erste kleiner ist als der zweite, 0, wenn die beiden gleich sind, und 1, wenn der erste größer ist als der zweite. Wenn dieses Verhalten auch nicht allzu sinnvoll für den normalen »Hausgebrauch« scheinen mag, so sind diese Operatoren für `sort`- Routinen doch von großem Nutzen, denn dort werden genau diese Angaben für die Sortierung gebraucht.
1. Der Unterschied zwischen <=> und `cmp` ist der gleiche wie der zwischen == und `eq`: <=> nimmt man für Zahlen, `cmp` für Strings.
4. Der Ausdruck oder Block, den man als Argument an `grep` übergibt, ist ein Kriterium, ob ein bestimmtes Listenelement (wie in `$_` gespeichert) in der Ergebnisliste gespeichert werden soll: Evaluiert der Ausdruck (oder Block) zu ***wahr***, wird das Element gespeichert.
5. Man fügt einem Array mit `splice` Elemente hinzu, indem man als Längenargument (das dritte Argument) 0 angibt. So werden keine Elemente entfernt und die neuen Elemente an der angegebenen Startposition eingefügt:

```
    splice(@array, 0, 0, (1,2,3,4);
# fügt (1,2,3,4) am Array-Anfang ein
```

6. Damit ein Ausdrucksblock in `map` einen bestimmten Wert zurückgibt, muss der im Block zuletzt ausgewertete Ausdruck diesen Wert liefern. Das Ergebnis der letzten Auswertung ist auch der Rückgabewert des Blocks und wird dann an die neue Liste weitergegeben.

Lösungen zu den Übungen

1. Hier die Antworten:

```
   splice(@liste, $#liste+1, 0, 1);
splice(@liste, $#liste+1, 0, (2,3,4));
splice(@liste, 5, 1,"foo");
splice(@liste, 0, 1);
```

2. Dieser Code hat wirklich mehrere Fehler, den ersten in der zweiten Zeile: Der Array-Zugriffsausdruck ist hier in Wahrheit ein Array-Segment. Sie erhalten hier daher keinen String, sondern ein Array mit einem einzigen Element. Perl- Warnungen fangen diesen Fehler ab.
1. Doch selbst wenn Sie diesen Fehler beheben, wird der String nicht umgekehrt. Denn in der dritten Zeile wird `reverse` in einem Listenkontext aufgerufen und der String dadurch als einelementige Liste interpretiert. Stellen Sie `reverse` mit der `scalar`-Funktion in skalaren Kontext.
3. Irgendwo in der Schleife müssen Sie die aktuelle Position erhöhen, sonst findet die Schleife immer denselben Substring an immer derselben Stelle, wieder und wieder - endlos.
4. Hier eine Antwort:

```
    foreach (@liste) {
   if ($_ < 5) {
       push @liste2, $_;
   }
}
```

5. Hier eine Antwort:

```
    #!/usr/bin/perl -w
$str = ''; #String
$key = ''; #gesuchtes Zeichen
$pos = 0;   #temp Position
$count = 0; #Zaehler
print "Geben Sie den zu durchsuchenden String ein: ";
chomp($str = <STDIN>);
print "Geben Sie das zu zählende Zeichen ein: ";
chomp($key = <STDIN>);
while ($pos < (length $str) and $pos != -1) {
    $pos = index($str, $key, $pos);
    if ($pos != -1 ) {
        $count++;
        $pos++;
    }
}
print "$key kommt im String $count mal vor.\n";
```

6. Der knifflige Teil an dieser Übung ist die Zeile, in der `split` den String in eine Liste mit den Zeichen umwandelt. Haben Sie erst einmal eine Liste, ist auch der Einsatz von `grep` ganz einfach.

```
    #!/usr/bin/perl -w
$str = ''; #String
$key = ''; #gesuchtes Zeichen
$count = 0; #Zaehler
print "Geben Sie den zu durchsuchenden String ein: ";
chomp($str = <STDIN>);
print "Geben Sie das zu zählende Zeichen ein: ";
chomp($key = <STDIN>);
@chars = split('',$str);
$count = grep {$_ eq $key} @chars;
print "$key kommt im String $count mal vor.\n";
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Pattern Matching mit regulären Ausdrücken

Sie haben also den Stoff bis hierher gelesen und gelernt? Erliegen Sie jetzt allerdings nicht dem Glauben, dass Sie damit schon eine gute Basis für die Perl-Programmierung haben, nur weil Ihnen die meisten Konzepte, die auch vielen anderen Sprachen gemein sind, bereits begegnet sind. Denn wenn Sie dieses Buch vor dem heutigen Kapitel zur Seite legen und mit dem bisher Erlernten versuchen, Perl zu nutzen, entgeht Ihnen einer der leistungsfähigsten und flexibelsten Aspekte von Perl - Pattern Matching mit Hilfe regulärer Ausdrücke. Pattern Matching ist mehr als nur die Suche nach einer Zeichenfolge in Ihren Daten; es ist eine Art, Daten zu sehen und zu verarbeiten, die unglaublich effizient ist und sich erstaunlich leicht programmieren läßt. Perl zu lernen, ohne Bekanntschaft mit regulären Ausdrücken zu machen, ist wie Snowboard fahren, ohne mit Schnee in Berührung zu kommen. In anderen Worten, hören Sie hier noch nicht auf - der gute Teil kommt erst.

Heute beschäftigen wir uns eingehend mit regulären Ausdrücken, warum sie so nützlich sind, wie sie aufgebaut werden und wie sie funktionieren. Morgen setzen wir dann die Diskussion fort und kommen zu den fortschrittlicheren Anwendungen von regulären Ausdrücken. Heute jedoch lernen Sie:

- was sich hinter Pattern Matching und regulären Ausdrücken verbirgt und warum sie so nützlich sind
- wie man einfache reguläre Ausdrücke mit Einzelzeichen-Suchläufen und Pattern Matching-Operatoren aufbaut
- wie man mit Zeichengruppen sucht
- wie man mehrere Vorkommen eines Zeichens findet
- wie man Suchmuster in Bedingungen und Schleifen verwendet

Sinn und Zweck des Pattern Matching

Pattern Matching ist eine Technik, mit der ein String, bestehend aus Text oder binären Daten, nach einer Zeichenfolge durchsucht wird. Die Zeichenfolge wird dabei in Form eines speziellen Suchmusters angegeben. Wenn Sie zum Beispiel in Ihrer Textverarbeitung mit dem Suchen-Befehl nach einem Zeichenstring in einer Datei suchen oder sich im Web einer Suchmaschine bedienen, um etwas zu finden, setzen Sie bereits eine vereinfachte Form des Pattern Matching ein: Ihr Kriterium lautet »suche diese Zeichen«. In oben genannten Umgebungen können Sie Ihre Kriterien üblicherweise Ihren Bedürfnissen anpassen, zum Beispiel können Sie nach diesem oder jenem suchen oder nach dem und dem, aber nicht nach jenem, Sie können nach ganzen Wörtern suchen oder nur nach den Wörtern, die einen Schriftgrad von 12 haben und unterstrichen sind. Pattern Matching in Perl ist noch weitaus komplizierter. Hier können Sie nämlich hochspezielle Kombinationen an Suchkriterien definieren, und das mit unwahrscheinlich wenig Code, wobei Sie eine spezielle Minisprache zur Musterdefinition, die sogenannten regulären Ausdrücke, verwenden.

Die regulären Ausdrücke von Perl - in Anlehnung an den englischen Begriff »regular expression« oft auch nur ***regex*** oder ***RE*** genannt - basieren auf den regulären Ausdrücken vieler Unix-Tools, wie zum Beispiel `grep` und `sed`. Und wie bei so vielen Merkmalen, die Perl von anderen Programmen übernommen hat, wurden auch hier leichte Änderungen und viele zusätzliche Ergänzungen vorgenommen. Wenn Sie bereits mit regulären Ausdrücken vertraut sind, werden Sie keine Schwierigkeiten mit den regulären Ausdrücken in Perl haben, da Sie in der überwiegenden Zahl der Fälle die gleichen Regeln anwenden können (auch wenn es einige Fallen gibt, vor denen man sich hüten muss, besonders wenn Sie vorher hauptsächlich mit komplizierten regulären Ausdrücken gearbeitet haben).

*Der Begriff **regulärer Ausdruck** mag auf den ersten Blick etwas unsinnig erscheinen. Erstens handelt es sich dabei nicht um richtige Ausdrücke, und zweitens läßt sich nur schwer erklären, was so regulär daran ist. Doch Sie sollten sich nicht allzu lange mit dem Namen aufhalten. Der Begriff **regulärer Ausdruck** entstammt der Mathematik und beschreibt die Sprache, mit der Sie Muster für*

das Pattern Matching in Perl schreiben.

Die obigen Beispiele mit der Suchmaschine oder dem Suchen-Befehl sollten Ihnen zeigen, was mit Pattern Matching alles möglich ist. Sie sollten jetzt aber nicht schlußfolgern, dass man Pattern Matching nur für das Suchen nach Textstellen verwenden kann. Der Leistungsumfang der regulären Ausdrücke in Perl umfaßt:

- Eingabe-Validierung - es wird sichergestellt, dass der Benutzer die von Ihnen gewünschten Daten eingegeben hat.
- Prüfung, ob die Eingabe in dem korrekten spezifischen Format erfolgt ist, zum Beispiel ob E-Mail-Adressen aus den richtigen Komponenten bestehen.
- Herausziehen bestimmter Teile einer Datei, die einem bestimmten Kriterium entsprechen (Sie könnten zum Beispiel alle Überschriften extrahieren, um ein Inhaltsverzeichnis zu erstellen, oder alle Links einer HTML-Datei).
- Zerlegung eines Strings auf der Basis bestimmter Trennzeichen-Felder (und oft komplizierter verschachtelter Trennzeichen-Felder).
- Feststellen von Unregelmäßigkeiten in einer Datenmenge - zum Beispiel mehrere Leerzeichen, wo sie nicht hingehören, doppelte Wörter, Fehler in der Formatierung.
- Das Zählen der Vorkommen eines Musters in einem String.
- Suchen&Ersetzen-Operationen - einen String suchen, der mit einem vorgegebenen Muster übereinstimmt, und ihn durch einen anderen String ersetzen.

Dies ist natürlich nur eine kleine Liste der Möglichkeiten - Sie können reguläre Ausdrücke in Perl für eine Vielzahl von Aufgaben einsetzen. Allgemein läßt sich sagen, dass Sie für eine Aufgabe, bei der Sie einen String oder einen Text durchgehen, am besten Perl mit seinen regulären Ausdrücken einsetzen. Viele der Operationen zum Suchen von Strings, die Sie gestern kennengelernt haben, lassen sich mit Mustern viel besser lösen.

Pattern-Matching-Operatoren und -Ausdrücke

Beim Pattern Matching legen Sie zuerst einmal fest, was Sie suchen wollen, dann schreiben Sie einen regulären Ausdruck für die Suche und setzen anschließend das Muster in einer Situation ein, in der das Ergebnis der Suche (gefunden oder nicht gefunden) von Nutzen ist. Wie wir es schon von Perl gewohnt sind, ist die Art und Weise, wie Sie Muster einsetzen, davon abhängig, wo und in welchem Kontext Sie das Muster verwenden.

Lassen Sie uns mit einem ziemlich einfachen Beispiel beginnen - Muster in einem Booleschen skalaren Kontext, bei dem der Ausdruck ***wahr*** zurückliefert, wenn der String das Muster enthält.

Zum Erstellen eines solchen Musters verwenden wir zwei Operatoren: den Operator für reguläre Ausdrücke `m//` und den Pattern-Matching-Operator `=~`:

```
if ($string =~ m/foo/) {
   # tue etwas ...
}
```

Der Test der `if`-Anweisung besagt: Liefere ***wahr*** zurück, wenn der String in der Variablen `$string` das Muster `foo` enthält. Beachten Sie, dass es sich bei dem Operator `=~` nicht um einen Zuweisungsoperator handelt, auch wenn er so aussieht. `=~` wird ausschließlich für den Mustervergleich verwendet und bedeutet: »Suche das Muster auf der rechten Seite im String auf der linken Seite.« Manchmal wird dieser Operator auch der ***Binding***-Operator genannt.

Das Suchmuster selbst steht zwischen den Slash-Zeichen von `m//`. Dieses spezielle Muster ist eines der einfachsten, das Sie erzeugen können, da es nur aus drei Sonderzeichen in Folge besteht (später werden Sie noch erfahren, was eigentlich eine Übereinstimmung (englisch ***match***) ist und was nicht). Ein Muster könnte auch folgendermaßen aussehen: `m/.*\d+/` oder `m/^[+-]?\d+\.?\d*$/` oder aus einem sonstwie unverständlichen Satz an Zeichen bestehen. Aber bitte keine Panik -- ich werde Ihnen in Bälde zeigen, wie man diese Muster dechiffriert.

Für diese Art von Muster ist das `m` optional und kann weggelassen werden (was in der Regel auch geschieht). Wenn Sie den Inhalt der Standardvariablen `$_` durchsuchen, können Sie außerdem die Variable und den Operator `=~` fortlassen. Sie werden in Perl häufig auf Kurzversionen wie die folgende stoßen:

```
if (/^\d+/) { # ...
```

Die ausgeschriebene Langversion dazu wäre:

```
if ($_ =~ m/^\d+/)  { # ...
```

Einen einfachen Fall dieser Art haben Sie bereits gestern im Zusammenhang mit der `grep`-Funktion kennengelernt, die mit Hilfe von Mustern einen String-Abschnitt innerhalb des Listenelements `$_` sucht.

```
@foodinge = grep /foo/, @strings;
```

Diese Zeile wiederum entspricht der ausgeschriebenen Form:

```
@foodinge = grep { $_ =~ /foo/ } @strings;
```

Im Laufe unserer heutigen Lektion werden Sie mehrere Möglichkeiten kennenlernen, Muster in unterschiedlichen Kontexten und aus verschiedenen Gründen einzusetzen. Dabei liegt die Hauptarbeit im Erlernen der Syntax der regulären Ausdrücke. Lassen Sie uns also mit diesem Aspekt beginnen.

Einfache Muster

Beginnen wir mit einigen der einfachsten und grundlegendsten Muster, die Sie erzeugen können: Muster, die mit bestimmten Zeichenfolgen übereinstimmen, Muster, die nur an bestimmten Positionen im String vorkommen können, und kombinierte Mustern, die mit Hilfe sogenannter Alternationen erstellt werden.

Zeichenfolgen

Zu einem der einfachsten Muster, für die Übereinstimmungen gesucht werden sollen, gehören die Zeichenfolgen:

```
/foo/
/dies oder das/
/   /
/Laura/
/Muster, die speziellen Folgen entsprechen/
```

All diese Muster werden gefunden (match), wenn Ihre Daten diese Zeichen in genau der angegebenen Reihenfolge enthalten. Dabei müssen alle Zeichen, auch die Leerzeichen deckungsgleich sein. Das Wort `oder` im zweiten Muster hat dabei keine besondere Bedeutung (es ist kein logisches `ODER`). Dieses Muster wird nur gefunden, wenn die Daten irgendwo den String »`dies oder das`« enthalten.

Beachten Sie, dass Zeichen in Mustern irgendwo in einem String gefunden werden können. Wortgrenzen werden nicht berücksichtigt - das Muster `/es/` wird sowohl in dem String »`es war einmal`« als auch in dem String »`diese Unterschiede sind keine`« gefunden. Das Muster `/es /` wird jedoch nur in dem ersten String gefunden, da es ein Leerzeichen enthält und die Zeichen `e`, `s` und das Leerzeichen nur in dem ersten String in genau der vorgegebenen Reihenfolge auftreten.

Beim Pattern Matching ist die Groß- und Kleinschreibung zu beachten: `/kazoo/` wird nur `kazoo` finden und nicht `Kazoo` oder `KAZOO`. Um bei einer Suche die Unterscheidung in Groß- und Kleinschreibung aufzuheben, müssen Sie nach dem Muster direkt die Option `i` eingeben (`i` steht für ***Groß- und Kleinschreibung ignorieren***):

```
/kazoo/i   # sucht nach allen Versionen in Groß- und Kleinschreibung
```

Sie können aber auch speziell Muster erzeugen, die entweder nach Groß- oder nach Kleinbuchstaben suchen. Darauf möchte ich aber erst im nächsten Abschnitt eingehen.

Sie können in den Mustern nahezu alle alphanumerischen Zeichen sowie die Escape- Darstellungen für binäre Daten (oktale und hexadezimale Escape-Zeichen) verwenden. Es gibt jedoch eine Reihe von Zeichen, die nur zusammen mit einem vorangehenden Escape-Zeichen in Mustern verwendet werden können. Diese Zeichen werden auch Metazeichen genannt und beziehen sich auf die Sprache des Pattern Matching und nicht auf das

literale Zeichen.

Die folgenden Metazeichen sollten Sie in Ihren Mustern beachten:

```
^ $

. +

? *

{ (

) \

/ |

[
```

Wenn Sie irgendwann tatsächlich ein Metazeichen in einem String suchen wollen - zum Beispiel das Fragezeichen (?) - müssen Sie dem Metazeichen das Escape-Zeichen Backslash (\) voranstellen :

```
/\?/  # sucht nach einem Fragezeichen
```

Übereinstimmung an Wort- und Zeilengrenzen

Wenn Sie Muster zum Vergleichen einer Zeichenfolge erzeugen, können diese Zeichen irgendwo innerhalb des Strings vorkommen, und die Suche führt zum Erfolg. Manchmal jedoch möchten Sie einen Mustervergleich nur für Zeichen an einer bestimmten Position durchführen - zum Beispiel `/es/` nur als alleinstehendes Wort erkennen oder `/kazoo/` nur zu Beginn einer Zeile (das heißt dem Beginn eines Strings).

> *Ich gehe dabei davon aus, dass es sich bei den Daten, die Sie durchsuchen, um eine einzelne Eingabezeile handelt, die aus einem einzigen String ohne Neue-Zeile-Zeichen besteht. Unter dieser Voraussetzung sind die Begriffe **String**, **Zeile** und **Daten** synonym. Morgen werden wir dann sehen, wie Muster mit mehrzeiligen Daten umgehen.*

Um Muster an einer bestimmten Position zu suchen, müssen Sie einen Musteranker verwenden. Der Musteranker für den Beginn eines Strings lautet ^. Ein Beispiel:

```
/^kazoo/  # Übereinstimmung nur, wenn kazoo am Anfang einer Zeile steht
```

Der Musteranker für das Ende eines Strings lautet $:

```
/Ende$/   # Übereinstimmung nur, wenn Ende am Ende der Zeile steht
```

Auch hier sollten Sie sich das Muster als eine Folge von Elementen vorstellen, bei der jeder Teil des Musters mit den Daten, die durchsucht werden sollen, übereinstimmen muss. Die Pattern-Matching-Routinen in Perl starten den Suchvorgang an der Position, die direkt vor dem ersten Zeichen liegt und damit dem Zeichen ^ entspricht. Danach wird jedes Zeichen einzeln überprüft bis zum Ende der Zeile, das dem $- Zeichen entspricht. Folgt auf das Ende des Strings eine neue Zeile (angezeigt durch ein Neue-Zeile-Zeichen), befindet sich die durch $ markierte Position direkt vor diesem Zeichen.

Anhand eines Beispiels möchte ich Ihnen zeigen, was passiert, wenn Sie versuchen, das Muster `/^foo/` in dem String »`sein oder nicht sein`« zu finden (was natürlich nicht klappt, uns aber trotzdem einen Versuch wert sein soll). Perl startet mit dem Anfang der Zeile, die dem ^-Zeichen entspricht. Dieser Teil der Zeile ist somit ***wahr***. Danach erfolgt der Test des ersten Zeichens. Das Muster erwartet dort ein `f`, findet aber ein `s` vor. Deshalb wird der Mustervergleich abgebrochen und ***falsch*** zurückgeliefert.

Und was passiert, wenn Sie versuchen, obiges Muster auf den String »fob« anzuwenden? Der Mustervergleich führt etwas weiter - übereinstimmend sind der Anfang der Zeile, das f und das o, aber dann schlägt der Mustervergleich bei b fehl. Denken Sie auch daran, dass /^foo/ in dem String »-foo« nicht gefunden wird - denn foo steht nicht direkt am Anfang der Zeile, wie es vom Muster erwartet wird. Übereinstimmung ist nur gegeben, wenn alle vier Elemente des Musters dem String entsprechen.

Hier einige interessante, aber knifflige Verwendungen von ^ und $. Können Sie raten, wann es bei diesen Beispielen zu einer Übereinstimmung kommt?

```
/^/
/^1$/
/^$/
```

Das erste Muster entspricht allen Strings, die am Anfang einer Zeile beginnen. Es wäre schon ein recht seltsamer String, der nicht mit dem Anfang einer Zeile beginnen würde. Deshalb entspricht dieses Muster allen Strings, sogar einem leeren String.

Das zweite Muster erwartet am Anfang einer Zeile die Zahl 1 und anschließend das Ende der Zeile. Eine Übereinstimmung gibt es also nur, wenn der String aus einer 1 besteht, wirklich nur aus einer 1 - dazu gehören weder »123« noch »foo 1« und auch nicht »1«.

Das dritte Muster geht davon aus, dass auf den Beginn der Zeile direkt das Ende der Zeile folgt - das heißt, dass es keine eigentlichen Daten gibt. Dieses Muster sucht nach einer leeren Zeile. Denken Sie jedoch daran, dass $ direkt vor dem Zeichen für »Neue Zeile« steht. Deshalb wird sowohl »« als auch »\n« gefunden.

Eine weitere Grenze, die beim Pattern Matching von Bedeutung ist, ist die Wortgrenze. Unter einer Wortgrenze versteht man die Position, die zwischen einem Wortzeichen (ein Buchstabe, eine Zahl oder ein Unterstrich) und einem anderen Zeichen (Leerzeichen oder Interpunktionszeichen) liegt. Eine Wortgrenze wird durch das Escape-Zeichen[1] \b angezeigt. Mit dem Muster /\bes\b finden Sie nur das ganze Wort es im String. Vorkommen, bei denen die Zeichen e und s in der Mitte eines Wortes stehen (wie in »Vergessenheit«) werden übergangen. Sie können mit \b sowohl auf den Anfang als auch auf das Ende eines Wortes Bezug nehmen. Das Muster / \bes/ zum Beispiel führt sowohl in dem String »es war einmal« als auch in »die Situation eskaliert« oder sogar »so sei es!« zu einer Übereinstimmung, aber nicht in »dieses Beispiel« oder »in Vergessenheit geraten«.

Sie können auch nach einem Muster suchen, das nicht an einer Wortgrenze steht. Dazu gibt es das \B. Demzufolge wird das Muster /\Bes/ nur gefunden, wenn die Zeichen e und s innerhalb eines Wortes und nicht zu Beginn stehen.

Alternativen vergleichen

Manchmal werden Sie nach mehr als einem Muster im gleichen String suchen und dann prüfen wollen, ob alle Muster oder überhaupt eines davon gefunden wurde. Sie könnten dieses Problem dadurch lösen, dass Sie mehrere Pattern-Matching- Ausdrükken mit Hilfe von Perls logischen regulären Ausdrücken für das Boolesche UND (&& oder and) und ODER (|| oder or) kombinieren, beispielsweise zu:

```
if (($in =~ /dies/) || ($in =~ /das/)) { ...
```

Wenn der durchsuchte String /dies/ oder /das/ enthält, liefert der ganze Test ***wahr*** zurück.

Für die ODER-Suche (vergleiche dies oder das Muster - eine Übereinstimmung reicht) gibt es AUCH ein Metazeichen, das Sie in den regulären Audrücken verwenden können: das Pipe-Zeichen (|). So könnte man den langen if-Test aus dem obigen Beispiel auch folgendermaßen schreiben:

```
if ($in =~ /dies|das/) { ...
```

Die Verwendung des |-Zeichens innerhalb eines Musters wird auch als ***Alternation*** bezeichnet, da es Ihnen erlaubt, verschiedene Muster zu vergleichen. Für ein solches Muster wird der Wert ***wahr*** zurückgegeben, wenn eines der Muster übereinstimmt.

Alle Ankerzeichen, die Sie mit einem Alternationszeichen verwenden, beziehen sich nur auf das Muster, das auf der gleichen Seite wie das Pipe-Zeichen steht. So bedeutet zum Beispiel das Muster `/^dies|das/`, dass das »`dies` am Anfang der Zeile« oder das »`das` irgendwo« zu suchen ist und nicht, dass »`dies`« oder »`das`« am Anfang der Zeile zu stehen hat. Wenn Sie diese letztgenannte Mustervariante wünschen, könnten Sie `/^dies|^das/` schreiben, die elegantere Lösung wäre aber, Ihre Muster in Klammern zu setzen:

```
/^(dies|das)/
```

Bei diesem Muster überprüft Perl zuerst den Anfang der Zeile und testet dann alle einzelnen Zeichen von »`dies`«. Gibt es dabei keine Übereinstimmung, geht Perl wieder zurück zum Anfang der Zeile und versucht, für »`das`« eine Übereinstimmung zu finden. Bei dem Muster `/^dies|das/` wird zuerst versucht, eine Übereinstimmung für alles, was auf der linken Seite der Alternation steht (Anfang der Zeile gefolgt von »`dies`«), zu finden. Ist der Versuch fehlgeschlagen, geht Perl zurück und durchsucht den ganzen String nach »`das`«.

Eine noch bessere Lösung wäre es, nur die Elemente zusammenzufassen, die sich im Muster unterscheiden, das heißt nicht nur das `^`-Zeichen für den Anfang der Zeile, sondern auch das `d`-Zeichen:

```
/^d(ies|as)/
```

Diese letzte Variante bedeutet, dass Perl die Alternation nur dann vornimmt, wenn bereits `d` zu Beginn der Zeile erkannt wurde. Damit kann man das Zurückgehen für unnötige Vergleiche minimieren. Für reguläre Ausdrücke gilt die Regel: Je weniger Perl für den Mustervergleich aufwenden muss, um so besser.

Sie können alle Arten von Alternativen in einem Muster zusammenfassen. So sucht zum Beispiel das Muster `/(1.|2.|3.|4.) Mal/` nach »`1. Mal`«, »`2. Mal`« und so weiter - solange wie die Daten eine der Alternativen innerhalb der Klammern sowie den String »` Mal`« enthalten (man beachte das Leerzeichen).

Mit Zeichengruppen vergleichen

So weit, so gut? Die bisher erstellten regulären Ausdrücke werden wohl auch Sie nicht als besonders komplex einstufen, vor allem nicht, wenn Sie jedes Muster - Zeichen um Zeichen, Alternative um Alternative unter Berücksichtigung etwaiger Zusammenfassungen - aus der Sicht von Perl betrachten. In diesem Abschnitt möchte ich auf einige der Kurzformen der regulären Ausdrücke zu sprechen kommen, die Sie zum Beschreiben und Zusammenfassen von verschiedenen Arten von Zeichen verwenden können.

Zeichenklassen

Angenommen Sie haben einen String und wollten eines von vier Wörtern in diesem String suchen: `Aal`, `Mal`, `Tal` und `Wal`. Sie könnten dazu folgendes eingeben:

```
/Aal|Mal|Tal|Wal/
```

Das funktioniert natürlich. Perl würde den ganzen String nach `Aal` durchsuchen, dann nach `Mal`, dann nach `Tal` und so weiter. Es geht aber auch kürzer - sowohl für Sie bei der Eingabe als auch für Perl bei der Ausführung. Dazu müssen Sie die Zeichen so zusammenfassen, dass Sie das `al` nicht jedesmal separat anführen:

```
/(A|M|T|W)al/
```

Bei diesem Muster durchsucht Perl den gesamten String nach `A`, `M`, `T` oder `W` und erst, wenn es eines dieser Zeichen findet, versucht es, eine Übereinstimmung mit `al` zu erzielen. Das ist wesentlich effizienter!

Diese Art von Muster - bei der eine große Anzahl an Alternativen einzelner Zeichen besteht - ist derart häufig, dass es hierfür in den regulären Ausdrücken eine eigene Syntax gibt. Der Satz an alternativen Zeichen wird dabei als ***Zeichenklasse*** bezeichnet, die Sie in eckige Klammern setzen. Mit Hilfe einer Zeichenklasse würde unser Beispiel von oben wie folgt geschrieben:

```
/[AMTW]al/
```

Damit sparen Sie eine Reihe von Zeichen, und es ist darüber hinaus auch noch leichter zu lesen. Dieses Beispiel wird von Perl genauso gelesen und abgearbeitet wie das mit den Alternationszeichen: Erst werden die Zeichen innerhalb der Zeichenklasse abgeglichen und dann alle Zeichen außerhalb davon.

Die Zeichen, die innerhalb einer Zeichenklasse auftauchen können, unterliegen anderen Regeln als die, die außerhalb stehen. Die meisten Metazeichen werden innerhalb einer Zeichenklasse zu einem ganz gewöhnlichen Zeichen (abgesehen von dem Zeichen für die schließende eckige Klammer, die aus verständlichen Gründen mit einem Escape-Zeichen versehen sein muss: dem ***Caret*** (^), das nicht an erster Stelle stehen darf, oder einem Bindestrich, dem innerhalb einer Zeichenklasse eine besondere Bedeutung zukommt). Ein Beispiel zur Kontrolle der Interpunktion am Ende eines Satzes (Interpunktion nach einer Wortgrenze und vor zwei Leerzeichen) könnte wie folgt aussehen:

```
/\b[.!?]  /
```

Außerhalb der Zeichenklasse haben der Punkt (.) und das Fragezeichen (?) eine besondere Bedeutung, innerhalb davon sind sie jedoch nur einfache Zeichen.

Bereiche

Angenommen Sie wollten nach allen Kleinbuchstaben von a bis f suchen (zum Beispiel in einer Hexadezimalzahl). Ihre Eingabe könnte lauten:

```
/[abcdef]/
```

Das sieht doch sehr nach einem Bereich aus? Sie können innerhalb von Zeichenklassen Bereiche definieren, verwenden dazu allerdings nicht den aus Tag 4 bekannten Bereichsoperator (..). Reguläre Ausdrücke verwenden für Bereiche den Bindestrich (weshalb Sie ihn auch mit einem Escape-Zeichen versehen müssen, wenn Sie konkret nach einem Bindestrich suchen). Ein Bereichsmuster für a bis f würde folgendermaßen aussehen:

```
/[a-f]/
```

Sie können als Bereich eine beliebige Anzahl von Zeichen oder Zahlen eingeben: /[0-9]/, /[a-z]/ oder /[A-Z]/. Die Bereiche lassen sich sogar kombinieren. Dabei entspricht / [0-9a-z]/ der Langform / [0123456789abcdefghijklmnopqrstuvwxyz]/.

Negierte Zeichenklassen

Eckige Klammern definieren eine Klasse von zu vergleichenden Zeichen in einem Muster. Sie können aber auch einen Satz von Zeichen definieren, der nicht verglichen werden soll, eine sogenannte negierte Zeichenklasse. Dazu müssen Sie lediglich dafür Sorge tragen, dass das erste Zeichen in der Zeichenklasse ein Caret-Zeichen (^) ist. Das Beispiel für einen Mustervergleich, bei dem alles außer A oder B verglichen wird, lautet:

```
/[^AB]/
```

Beachten Sie, dass das Caret innerhalb einer Zeichenklasse eine andere Bedeutung hat als außerhalb. Innerhalb wird damit eine negierte Zeichenklasse definiert und außerhalb der Anfang einer Zeile.

Wenn Sie irgendwann einmal über eine Zeichenklasse nach dem Caret-Zeichen suchen wollen, können Sie das problemlos - Sie müssen jedoch sicherstellen, dass es nicht das erste Zeichen ist oder es mit einem Escape-Zeichen versehen (am besten verwenden Sie für beide Fälle ein Escape-Zeichen, damit Sie sich nicht so viele Regeln merken müssen):

```
/[\^?.%]/    #  sucht nach ^, ?, ., %
```

Sie sollten sich diese Syntax merken, da Sie wahrscheinlich ziemlich häufig von negierten Zeichenklassen in regulären Ausdrücken Gebrauch machen werden. Eine Feinheit gilt es jedoch zu beachten: Negierte Zeichenklassen negieren nicht den ganzen Wert des Musters. Wenn also /[12]/ bedeutet: »Liefere ***wahr*** zurück, wenn die Daten die Zeichen 1 oder 2 enthalten«, bedeutet /[^12]/ nicht: »Liefere ***wahr*** zurück, wenn die Daten weder 1 oder 2 enthalten«. Wäre dies der Fall, würde es zu einer Übereinstimmung kommen, auch wenn der untersuchte String

leer wäre. Was negierte Zeichenklassen eigentlich aussagen, ist: »Suche nach allen Zeichen, die in dem Muster nicht aufgeführt sind.« Es muss also zumindest ein richtiges Zeichen gefunden werden, damit eine negierte Zeichenklasse greift.

Besondere Klassen

Sollten Bereichsangaben für Zeichenklassen Ihnen immer noch zuviel Tipparbeit sein, gibt es als Erleichterung einige besondere Klassen (auch negierte Zeichenklassen), die einen eigenen Escape-Code aufweisen. Sie werden häufig auf diese besonderen Klassen in regulären Ausdrücken stoßen, besonders in Fällen, wo Zahlen in bestimmten Formaten verglichen werden. Beachten Sie, dass diese speziellen Codes nicht von ekkigen Klammern eingeschlossen sein müssen; Sie können sie so, wie sie in der untenstehenden Tabelle aufgeführt sind, verwenden.

In Tabelle 9.1 finden Sie eine Liste der Codes für die besonderen Zeichenklassen.

Code	Äquivalente Zeichenklasse	Bedeutung
\d	[0-9]	Alle Ziffern
\D	[^ 0-9]	Alle Zeichen außer Ziffern
\w	[0-9a-zA-z_]	Alle »Wortzeichen« (alphanumerische Zeichen und _)
\W	[^ 0-9a-zA-z_]	Alle Zeichen außer »Wortzeichen«
\s	[\t\n\r\f]	Whitespace-Zeichen (Leerzeichen, Tabulator, Neue Zeile, Wagenrücklauf, Blattvorschub)
\S	[^ \t\n\r\f]	Alle Zeichen außer Whitespace-Zeichen

Tabelle 9.1: ***Codes für Zeichenklassen***

Die Wortzeichen (`\w` und `\W`) bedürfen einer Erklärung. Warum sollte ein Unterstrich ein Wortzeichen sein, die anderen Interpunktionszeichen aber nicht? Um genau zu sein, haben Wortzeichen sehr wenig mit Wörtern zu tun. Es handelt sich dabei um alle zulässigen Zeichen für die Vergabe eines Variablennamens: Zahlen, Buchstaben und Unterstriche. Alle anderen Zeichen gehören nicht zu den Wortzeichen.

Sie können diese Zeichencodes überall dort verwenden, wo Sie einen besonderen Zeichentyp benötigen. Der Code `\d` bezieht sich zum Beispiel auf alle Ziffern. Mit `\d` können Sie ein Muster erzeugen, das drei Ziffern vergleicht (`/\d\d\d/`) oder drei Ziffern, ein Leerzeichen und sechs weitere Ziffern (`/\d\d\d \d\d\d\d\d\d/`) - eine Telefonnummer wie 089 121212. Wie es auch ohne die lästigen Wiederholungen geht, werden Sie bald erfahren, wenn wir zu den Quantifizierern kommen.

Mit . ein beliebiges Zeichen finden

Die am breitesten gefaßte Zeichenklasse, die Sie definieren können, vergleicht nach nur einem, dafür aber beliebigen Zeichen. Hierfür verwenden Sie das Punktzeichen (`.`). Das folgende Muster ergibt für alle Zeilen, die wirklich nur ein Zeichen enthalten, eine Übereinstimmung:

```
/^.$/
```

Eigentlich kommt der Punkt vor allem in Mustern mit Quantifizierern (die wir gleich behandeln werden) zum Einsatz. Sie können den Punkt aber auch nutzen, um zum Beispiel Felder einer bestimmten Länge anzuzeigen:

```
/^..:/
```

Dieses Muster führt nur zu einer Übereinstimmung, wenn die Zeile mit zwei Zeichen und einem Doppelpunkt beginnt.

Mehr zu dem Punktoperator nach dem folgenden kleinen Beispiel.

Ein Beispiel: Den Zahlenbuchstabierer optimieren

Erinnern Sie sich noch an das `zahlenbuchstabierer`-Skript von vorgestern? Dieses Skript übernahm eine einstellige Zahl und wandelte sie in ein Wort um. Vielleicht erinnern Sie sich auch noch an die Stelle, als ich erwähnte, dass es hier einfacher wäre, reguläre Ausdrücke zu verwenden. Und nachdem Sie jetzt einiges Grundwissen zu den regulären Ausdrücken erworben haben, lassen Sie uns eine Neufassung dieses Skripts erstellen, bei der die ganzen `if`-Anweisungen durch reguläre Ausdrücke ersetzt werden.

Und da wir gerade dabei sind, können wir auch den Teil des Zahlenbuchstabierers überarbeiten, der die Eingabe überprüft. Wir können, was die Eingabevalidierung betrifft, mit regulären Ausdrücken wesentlich mehr machen - bis hin zur Absurdität. Ja, wir werden in diesem Skript bei der Eingabevalidierung der Absurdität sogar sehr nahe kommen. Diese Version testet auf eine Reihe von möglichen Eingaben und gibt dabei verschiedene (manchmal recht sarkastische) Kommentare aus:

```
% zahlenbuchstabierer2.pl
Geben Sie die zu buchstabierende Zahl ein (0-9): foo
Sie können mich nicht täuschen.  Die Eingabe enthält Buchstaben.
Geben Sie die zu buchstabierende Zahl ein (0-9): 45foo
Sie können mich nicht täuschen.  Die Eingabe enthält Buchstaben.
Geben Sie die zu buchstabierende Zahl ein (0-9): ###
huh?  Das sieht *wirklich* nicht nach einer Zahl aus
Geben Sie die zu buchstabierende Zahl ein (0-9): -45
Das ist eine negative Zahl.  Bitte nur positive Zahlen!
Geben Sie die zu buchstabierende Zahl ein (0-9): 789
Zu groß!  0 bis 9 bitte.
Geben Sie die zu buchstabierende Zahl ein (0-9): 4
Danke!
Die Zahl 4 schreibt sich vier
Eine weitere Zahl versuchen (j/n)?: x
j oder n, bitte
Eine weitere Zahl versuchen (j/n)?: n
%
```

Anstatt Ihnen jetzt das Skript ganz zu zeigen und jede Zeile einzeln durchzugehen, möchte ich einen anderen Weg einschlagen: Ich zeigen Ihnen Teile der alten und der neuen `zahlenbuchstabierer`-Version, erläutere sie und bringe das Beispiel am Ende, so dass Sie sich dann einen Gesamteindruck verschaffen können.

Beginnen wir mit der Schleife, die eine Zahl als Eingabe erwartet. So sah die Schleife in der alten Version des Zahlenbuchstabierers aus:

```
while () {
       print 'Geben Sie die zu buchstabierende Zahl ein: ';
       chomp($num = <STDIN>);
       if ($num ne "0" && $num == 0) { # wenn $num ein String
            print "Keine Strings. Eine Zahl von 0 bis 9 bitte.\n";
            next;
        }
        if ($num > 9) { # wenn $num mehrstellige Zahl
            print "Zu hoch. 0 bis 9 bitte.\n";
            next;
        }
        if ($num < 0) { # wenn $num negative Zahl
            print "Keine negativen Zahlen. 0 bis 9 bitte.\n";
            next;
        }
        last;
    }
```

Wir können diese drei Tests in der Schleife ohne größere Probleme durch reguläre Ausdrücken, die sogar verständlicher sind, ersetzen - und wir können darüber hinaus auf noch schwierigere Sachen prüfen. Unsere neue Schleife prüft, ob die drei folgenden Hauptbedingungen gegeben sind:

- ob die Eingabe aus einer einzelnen Zahl, und damit meine ich nur einer Ziffer, besteht (womit wir bereits fertig wären).
- ob die Eingabe statt einer Zahl irgendein anderes Zeichen enthält

- ob die Zahl größer als 9 ist.

Der zweite Test kann in mehrere kleinere Tests zerlegt werden, die dann auf alphabetische Zeichen, negative Zahlen (beginnen mit einem -), Fließkommazahlen (mit einem Dezimalpunkt) oder absolut abwegige Zeichen testen. Sehen Sie im folgenden die neue Version unserer Schleife, die zusätzlich von der Variablen $_ Gebrauch macht, um uns etwas Tipparbeit bei den Tests des Mustervergleichs zu ersparen:[2]

```
1: while () {
2:     print 'Geben Sie die zu buchstabierende Zahl ein (0-9): ';
3:     chomp($_ = <STDIN>);
4:     if (/^\d$/) {  # korrekte Eingabe
5:         print "Danke!\n";
6:         last;
7:     } elsif (/^$/) {
8:         print "Sie haben nichts eingegeben.\n";
9:     } elsif (/\D/) { # keine Zahlen
10:        if (/[a-zA-z]/) { # Buchstaben
11:            print "Sie können mich nicht täuschen.  Die Eingabe enthält
                      Buchstaben.\n";
12:        } elsif (/^-\d/) { # negative Zahlen
13:            print "Das ist eine negative Zahl. Bitte nur positive
                      Zahlen!\n";
14:        } elsif (/\./) { # Dezimalzahlen
15:            print "Das sieht sehr nach einer Dezimalzahl aus.\n";
16:            print "Ich kann Dezimalzahlen nicht in Worten ausgeben.
                      Versuchen Sie eine neue Zahl.\n";
17:        } elsif (/[\W_]/) {  # andere Zeichen
18:            print "huh?  Das sieht *wirklich* nicht nach einer Zahl
                     aus\n";
19:        }
20:    } elsif ($_ > 9) {
21:        print "Zu groß!  0 bis 9, bitte.\n";
22:    }
23:  }
```

Werfen wir jetzt, Zeile für Zeile, einen Blick auf die regulären Ausdrücke, damit Sie wissen, auf was getestet wird:

- Zeile 4: `/^\d$/`
  Dieses Muster prüft, ob die Eingabe aus einer einstelligen Zahl besteht - das heißt, ob die Eingabe genau unseren Erwartungen entspricht. Ich habe dieses Muster als erstes definiert, damit für den Fall, dass der Benutzer eine korrekte Eingabe vornimmt, nicht eine Menge Zeit damit vergeudet wird, die ganzen Optionen auf Korrektheit zu prüfen. Wird bereits hier eine Übereinstimmung festgestellt und die Eingabe war korrekt, kann die Schleife an dieser Stelle direkt mit `last` verlassen werden.
- Zeile 7: `/^$/`
  Wie Sie bereits im Abschnitt zum Mustervergleich an Grenzen gelernt haben, überprüft dieses Muster auf eine leere Zeile - die Sie immer dann erhalten, wenn bei der Eingabeaufforderung die Eingabetaste betätigt wird, ohne dass vorher etwas eingegeben wurde.
- Zeile 9: `/\D/`
  Dieser Zeichencode steht für die Zeichenklasse »alle Zeichen außer Zahlen«. Wenn bei der Eingabeaufforderung irgend etwas eingegeben wird, das keine Zahl ist, zum Beispiel eine Kombination von Zahlen und Buchstaben, nur Buchstaben oder Zeichen wie -, . oder $ trifft dieses Muster zu. Anschließend wird in eine Reihe von Untertests verzweigt, um die Eingabe von speziellen nicht-numerischen Zeichen abzufangen.
- Zeile 10: `/[a-zA-Z]/`
  Dieser Zeichenklassenbereich sucht nach den Zeichen des Alphabets. Ich habe hier absichtlich auf die Verwendung des Codes `\w` verzichtet, da dieser den Unterstrich mit eingeschlossen hätte. Den jedoch möchte ich zusammen mit allen anderen Zeichen in einem separaten Test unterbringen.
- Zeile 12: `/^-\d/`
  Hier prüfen wir auf einen Strich am Anfang der Zeile direkt gefolgt von einer Ziffer. Das ist der Test auf negative Zahlen
- Zeile 14: `/\./`
  Bei einer Eingabe, die einen Punkt enthält, handelt es sich höchstwahrscheinlich um eine Fließkommazahl. Beachten Sie, dass der Punkt hier mit einem Escape- Zeichen versehen werden muss, um einen tatsächlich Punkt zu repräsentieren (ansonsten stellt der Punkt in Mustern ein Metazeichen dar).
- Zeile 17: `/[\W_]/`

Hier verwenden wir eine Zeichenklasse für zwei Abfragen: alle Zeichen, die keine Wortzeichen sind (0-9, a-z, A-Z) sowie den Unterstrich. Dies ist unsere Auffanglösung für alle anderen Zeichen, die eventuell eingegeben wurden.

- Zeile 20: `hier kein Muster`
  Diese Zeile fängt alle Eingaben ab, bei denen es sich um eine Zahl handelt (und die nicht durch die meisten der vorherigen Tests abgefangen werden konnten), die jedoch aus mehr als einer Ziffer bestehen. Hier wird nur geprüft, ob die Zahl größer als 9 ist. Es gibt zwar auch für diesen Fall ein Muster, doch haben wir dieses noch nicht kennengelernt.

Der nächste Teil des alten `zahlenbuchstabierer`-Skripts bestand aus einem Satz von `if...elsif`-Schleifen, die den Eingabewert mit einem Zahlenstring verglichen. Mit Hilfe von regulären Ausdrücken, der Standardvariablen `$_` und logischen Ausdrücken in der Funktion als Bedingung, können wir folgende verschachtelte `if`-Anweisungen:

```
if ($num == 1) { print 'eins'; }
    elsif ($num == 2) { print 'zwei'; }
    elsif ($num == 3) { print 'drei'; }
    elsif ($num == 4) { print 'vier'; }
    # ... andere Zahlen aus Platzgründen fortgelassen
}
```

auf einen Satz von folgenden logischen Anweisungen reduzieren:

```
/1/ && print 'eins';
/2/ && print 'zwei';
/3/ && print 'drei';
/4/ && print 'vier';
# ... und so weiter
```

Cool, nicht wahr? Es erinnert stark an `switch`-Anweisungen und ist nachweislich leichter zu lesen.

Abschließend schreiben wir unsere kleine Ja-oder-nein-Schleife neu, die das gesamte Skript bei Bedarf wiederholt. Die alte Version lautete:

```
while () {
    print 'Eine weitere Zahl versuchen (j/n)?: ';
    chomp ($exit = <STDIN>);
    $exit = lc $exit;
    if ($exit ne 'j' && $exit ne 'n') {
        print "j oder n, bitte\n";
    }
    else { last; }
}
```

Eigentlich ist an dieser Version nicht allzuviel falsch, aber da wir uns in diesem Kapitel mit Pattern Matching beschäftigen, wollen wir auch hier Muster einsetzen:

```
while () {
        print 'Eine weitere Zahl versuchen (j/n)?: ';
        chomp ($exit = <STDIN>);
        $exit = lc $exit;
        if ($exit =~ /^[jn]/) {
            last;
        }
        else {
            print "j oder n, bitte\n";
        }
    }
```

Beachten Sie die Unterschiede zwischen dieser Schleife und der Eingabeschleife. In der Eingabeschleife haben wir die Eingabe in der Variablen `$_` abgelegt, so dass wir in der `if`-Bedingung nur das Muster unterzubringen brauchten. Hier testen wir auf den String in der Variablen `$exit`, so dass wir den Operator `=~` verwenden müssen. Im Muster selbst testen wir, ob die Eingabe entweder `j` oder `n` lautete (J und N werden mit der Funktion `lc` zu Kleinbuchstaben konvertiert). Wenn ja, verlassen wir die Schleife und kehren zu der äußeren Schleife zurück, die das Skript, wenn nötig, erneut durchläuft.

In diesem Beispiel habe ich eine ganze Menge regulärer Ausdrücke verwendet und viele davon eigentlich unbegründet. An dieser Stelle möchte ich Sie darauf hinweisen, dass Sie nicht um jeden Preis reguläre Ausdrücke verwenden sollten, nur weil sie cool sind. Die Perl-Maschinerie der regulären Ausdrücke ist besonders leistungsfähig, wenn es um komplizierte Sachverhalte geht. Sie erzeugen damit jedoch nicht den effizientesten Code, wenn Sie sie für einfache Probleme einsetzen. In einem solchen Fall sind einfache Tests und `if`*-Anweisungen oft schneller in der Ausführung als reguläre Ausdrücke. Wenn Sie also bei Ihrem Code auch auf die Effizienz achten wollen, sollten Sie sich das merken.*

Listing 9.1 enthält den vollständigen Code für die neue Version von `zahlenbuchstabierer.pl`:

Listing 9.1: Das Skript zahlenbuchstabierer2.pl[3]

```
#!/usr/bin/perl -w
# Zahlenbuchstabierer:  gibt Zahlen in Worten aus
# einfache Version für einstellige Zahlen
$exit = "";  # ob das Skript verlassen werden soll oder nicht.
while ($exit ne "n") {
    while () {
        print 'Geben Sie die zu buchstabierende Zahl ein (0-9): ';
        chomp($_ = <STDIN>);
        if (/^\d$/) {
            print "Danke!\n";
            last;
        } elsif (/^$/) {
            print "Sie haben nichts eingegeben.\n";
        } elsif (/\D/) {        # Keine Zahlen
            if (/[a-zA-z]/) { # Buchstaben
                print "Sie können mich nicht täuschen. Die Eingabe enthält
                       Buchstaben.\n";
            } elsif (/^-\d/) { # negative Zahlen
                print "Das ist eine negative Zahl.  Bitte nur positive
                       Zahlen!\n";
            } elsif (/\./) { # Dezimalzahlen
                print "Das sieht sehr nach einer Dezimalzahl aus.\n";
                print "Ich kann Dezimalzahlen nicht in Worten ausgeben.
                       Versuchen Sie eine neue Zahl.\n";
            } elsif (/[\W_]/) {  # andere Zeichen
                print "huh?  Das sieht *wirklich* nicht nach einer Zahl
                       aus\n";
            }
        } elsif ($_ > 9) {
            print "Zu groß!  0 bis 9, bitte.\n";
        }
    }
    print "Die Zahl $_ schreibt sich ";
    /1/ && print 'eins';
    /2/ && print 'zwei';
    /3/ && print 'drei';
    /4/ && print 'vier';
    /5/ && print 'fünf';
    /6/ && print 'sechs';
    /7/ && print 'sieben';
    /8/ && print 'acht';
    /9/ && print 'neun';
    /0/ && print 'null';
    print "\n";
    while () {
        print 'Eine weitere Zahl versuchen (j/n)?: ';
        chomp ($exit = <STDIN>);
        $exit = lc $exit;
        if ($exit =~ /^[yn]/) {
            last;
        }
        else {
            print "j oder n, bitte\n";
        }
```

```
    }
}
```

Mehrere Übereinstimmungen von Zeichen finden

Sind Sie bereit, weiterzumachen? Dann komme ich zum zweiten Teil der Syntax für reguläre Ausdrücke: den Quantifizierern. Im Gegensatz zu den Mustern, die Sie bisher kennengelernt haben und die sich auf einzelne Dinge beziehungsweise Gruppen von einzelnen Dingen beziehen, können Quantifizierer für mehrere Vorkommen - oder keine Vorkommen - stehen. Als Quantifizierer bezeichnet man bestimmte Metazeichen in regulären Ausdrücken, die angeben, wie oft ein Zeichen oder eine Zeichenfolge in dem Suchmuster vorkommen soll.

Die regulären Ausdrücke von Perl kennen drei Metazeichen für Quantifizierer: `?`, `*` und `+`. Die Metazeichen beziehen sich auf die Häufigkeit des Musters (ein Zeichen oder eine Zeichenfolge) auf das sie direkt folgen.

Optionale Zeichen mit ?

Lassen Sie uns mit dem `?` anfangen. Damit können Sie Sequenzen finden, in denen das direkt vor dem Fragezeichen stehende Zeichen enthalten sein oder fehlen kann. Betrachten wir zum Beispiel folgendes Muster:

```
/wah?r/
```

Das Fragezeichen in diesem Muster bezieht sich auf das direkt davorstehende Zeichen (`h`). Dieses Muster würde sowohl in dem String »`das ist nicht wahr`« als auch in dem String »`es war einmal`« eine Übereinstimmung finden, da sowohl `wahr` als auch `war` diesem Muster entsprechen. Der String, den Sie durchsuchen, muss ein `w`, ein `a` und ein `r` enthalten, das `h` hingegen ist optional.

Auch hier sollten Sie sich vergegenwärtigen, wie der String verarbeitet wird. Zuerst wird das `w` und dann das `a` überprüft. Dann folgt das dritte Zeichen. Ist es ein `h`, dann gibt es keine Probleme, und wir gehen im String und im Muster zum nächsten Zeichen über (das `r`). Ist es kein `h`, ist das auch kein Problem. Dann gehen wir im Muster ein Zeichen weiter und schauen, ob es hiermit eine Übereinstimmung gibt.

Mit Hilfe von Klammern können Sie Sequenzen von optionalen Zeichen zusammenstellen:

```
/Milch(bar)?/
```

Durch die Klammern wird die ganze Zeichensequenz (`bar`) optional, so dass dieses Muster sowohl auf `Milch` als auch auf `Milchbar` prüft. Das, was vor dem Fragezeichen steht, ist optional, unabhängig davon, ob es ein einziges Zeichen oder eine Gruppe von Zeichen ist.

> *Da stellt sich die Frage, wozu man ein Muster wie dieses erzeugen sollte? Es scheint doch, als ob der `(bar)`-Teil dieses Musters ohne Bedeutung ist und der `/Milch/`-Teil - mit weniger Zeichen - genausogut zum Ziel führt. In diesen einfachen Fällen, in denen wir nur herauszufinden versuchen, ob etwas gefunden wird oder nicht, spielt es keine große Rolle. Morgen jedoch, wenn Sie lernen, wie Sie das gefundene Objekt extrahieren und damit komplexere Muster erstellen, wird die Unterscheidung deutlicher.*

Sie können das Fragezeichen auch zusammen mit Zeichenklassen verwenden:

```
/Punkt \d?/
```

Dieses Muster findet für die Strings »`Punkt 1`«, »`Punkt 9`« und so weiter genauso wie für den String »`Punkt `« (man beachte das Leerzeichen) eine Übereinstimmung. Jedes Zeichen der Zeichenklasse kann entweder einmal oder keinmal vorkommen, damit das Muster eine Übereinstimmung findet.

Mehrere optionale Zeichen mit *

Der zweite Quantifizierer ist der Multiplikator, das *-Zeichen, das ähnlich dem ? funktioniert. Im Gegensatz zum Fragezeichen, das nur eine oder keine Instanz des vorangehenden Zeichens erlaubt, ist bei dem * keine oder eine beliebige Anzahl des Zeichens möglich. Betrachten wir folgendes Muster:

```
/xy?z/
```

In diesem Beispiel sind das x und das z erforderlich, das y aber kann beliebig oft oder auch gar nicht enthalten sein. Dieses Muster findet `xyz`, `xyyz`, `xyyyyyyyyyyz` oder auch nur `xz` ohne das y.

Wie bei dem ? können Sie auch das * zusammen mit Zeichenfolgen oder Zeichenklassen verwenden. Eines der Hauptanwendungsbereiche ist die Kombination des *-Zeichens mit dem Punktzeichen. Damit drücken Sie aus, dass an dieser Position eine beliebige Anzahl von beliebigen Zeichen stehen kann:

```
/dies.*/
```

Mit diesem Muster finden Sie die Strings »`dieses`«, »`dies ist nicht mein Problem, kümmere Du dich drum`« oder einfache nur »`dies`« - zur Erinnerung, es muss kein Zeichen am Ende stehen, damit eine Übereinstimmung gefunden wird.

Ein häufiger Fehler ist, zu vergessen, dass * für »keine oder mehrere Instanzen« steht, und es wie folgt zu verwenden:

```
if (/^[0-9]*$/) {
    # soll Zahlen enthalten
}
```

Dieses Muster soll nur zu einer Übereinstimmung führen, wenn die Eingabe Zahlen enthält, und zwar nur Zahlen. Erreicht wird damit jedoch, dass dieses Muster zwar Zahlen wie »`7`«, »`1540`« und »`15443`« findet, aber auch leere Strings akzeptiert, da das *- Zeichen bedeutet, dass eine Übereinstimmung auch dann gegeben ist, wenn keine Zahl vorhanden ist. Normalerweise würden Sie das + statt dem * benutzen, wenn irgend etwas zumindest einmal vorhanden sein soll.

Beachten Sie, dass dieses Beispiel »finde keine oder mehrere Zahlen« nicht bedeutet, dass jeder String, der keine Zahlen aufweist, gefunden wird. So wird zum Beispiel der String »`Lederhosen`« zu keinem Ergebnis führen. Ein Muster, das auf keine oder mehrere Zahlen prüft, umfaßt nicht automatisch andere Prüfungen. Wenn Sie eine Übereinstimmung für Zeichen statt für Zahlen möchten, müssen Sie das im Muster auch entsprechend definieren. Bei regulären Ausdrücken müssen Sie sehr genau bei der Angabe Ihrer Musterdefinitionen sein.

Mindestens eine Instanz mit +

Das Metazeichen + ist in seiner Funktionsweise mit dem Zeichen * identisch - bis auf einen kleinen Unterschied: Anstatt keine oder mehrere Instanzen des gegebenen Zeichens oder der Zeichenfolge zu prüfen, verlangt +, dass das Zeichen oder die Zeichenfolge mindestens einmal zu finden ist (eine oder mehrere Instanzen). Betrachten wir das obige Beispiel:

```
/xy+z/
```

Dieses Muster führt für folgende Strings zu einer Übereinstimmung: `xyz`, `xyyz`, `xyyyyyyz`, aber nicht für `xz`. Das y muss mindestens einmal vorhanden sein.

Wie schon bei * und ? können Sie auch das +-Zeichen mit Zeichenfolgen und Zeichenklassen verwenden.

Die Anzahl der Wiederholungen beschränken

Bei den Metazeichen + und * kann das gegebene Zeichen oder die Zeichenfolge beliebig oft vorhanden sein, es gibt nach oben keine Begrenzung (Zeichen mit ? können maximal nur einmal vorhanden sein). Wie aber gehen Sie vor, wenn Sie eine Übereinstimmung nur für eine spezielle Anzahl von Wiederholungen suchen? Was, wenn Sie für das gesuchte Muster eine Ober- beziehungsweise Untergrenze angeben wollen, so dass eine Wiederholung zuviel oder zuwenig nicht zu einer Übereinstimmung führt? Um die Anzahl der zu findenden Wiederholungen festzulegen,

können Sie den optionalen Metazeichen geschweifte Klammern wie folgt einsetzen:

```
/\d{1,4} /
```

Dieses Muster wird gefunden, wenn die durchsuchten Daten eine Ziffer, zwei Ziffern, drei Ziffern oder vier Ziffern, jeweils gefolgt von einem Leerzeichen, enthalten. Weder mehr Ziffern noch keine Ziffer sind als Eingabe zulässig. Die erste Zahl in den geschweiften Klammern gibt die minimale Anzahl und die zweite die maximale Anzahl der Wiederholungen an. Durch Angabe nur einer Zahl können Sie eine feste Zahl von Wiederholungen angeben:

```
/a{5}b/
```

Ein Muster wie dieses sucht nach genau fünf `a`'s in einer Reihe, gefolgt von einem `b` - nicht mehr und nicht weniger. Es entspricht damit dem Muster `/aaaaab/`. Ein weniger spezifischer Einsatzbereich von `{}` wäre beispielsweise:

```
/DM \d+\.\d{2}/
```

Können Sie nach Studium dieses Musters selbst herausfinden, wie eine Übereinstimmung aussehen müßte? Ich habe hier eine Reihe von Escape-Zeichen verwendet, so dass Sie vielleicht etwas verwirrt sind. Zuerst wird die Zeichenfolge »`DM` « gesucht, dann ein oder mehrere Dezimalzahlen `\d+`, gefolgt von einem Punkt (`.`). Abschließend dürfen dann nur noch zwei und nicht mehr Dezimalzahlen stehen. Setzen Sie diese Aussagen zusammen, und Sie werden feststellen, dass dieses Muster einem Geldbetrag entspricht - `DM 45.23` wären demnach korrekt wie auch `DM 0.45` oder `DM 15.00`. Die Angabe von `DM .45` oder `DM 34.2` hingegen würden nicht erkannt, da dieses Muster mindestens eine Zahl links des Dezimalpunktes erwartet und genau zwei Zahlen rechts davon.

Kommen wir zurück zu den geschweiften Klammern. Sie können damit auch nur eine Untergrenze festlegen und die Anzahl der Wiederholungen nach oben offenlassen. Lassen Sie dazu die Zahlenangabe für das Maximum fort, und behalten Sie das Komma bei:

```
/ba{4,}t/
```

Dieses Muster sucht ein `b`, mindestens vier oder mehr Instanzen von `a` und dann den Buchstaben `t`. Drei Instanzen von `a` werden nicht gefunden, zwanzig hingegen problemlos.

Zu Ihrer Information: Sie können auch `+`, `*` und `?` mit Hilfe der geschweiften Klammern ausdrücken:

```
/x{0,1}/   # entspricht /x?/
/x{0,}/    # entspricht /x*/
/x{1,}/    # entspricht /x+/
```

Mehr zum Erstellen von Mustern

Wir haben dieses Kapitel mit einem allgemeinen Überblick über die Verwendung von Mustern in Perl-Skripts begonnen, wobei wir einen `if`-Test und den Operator `=~` verwendet haben (den Sie beim Durchsuchen von `$_` fortlassen können). Und da Sie jetzt eine Vorstellung davon haben, wie Sie mit der Syntax der regulären Ausdrücke Muster erstellen, möchte ich zu Perl zurückkehren und beschreiben, wie Sie Muster in Ihren Perl-Skripts einsetzen können. Außerdem zeige ich Ihnen, wie Sie Variablen in den Mustern interpolieren und Muster in Schleifen verwenden.

Muster und Variablen

In allen bisher betrachteten Beispielen haben wir Muster als hartcodierten Satz von Zeichen in `if`-Bedingungen verwendet. Was aber, wenn Sie je nach Eingabe verschiedene Dinge vergleichen wollen? Wie können Sie das Suchmuster bei Ausführung des Skripts erzeugen oder ändern?

Problemlos! Muster können wie Anführungszeichen Variablen enthalten, und der Wert der Variablen wird im Muster ersetzt:

```
$muster = "^\d{3}$";
if (/$muster/) {...
```

Die betreffende Variable kann einen String eines beliebigen Musters enthalten, einschließlich Metazeichen. Sie können diese Technik verwenden, um Muster auf verschiedene Art und Weise zu kombinieren oder um nach Mustern auf der Basis einer Eingabe zu suchen. Sehen Sie im folgenden ein Beispiel für ein einfaches Skript, das Sie auffordert, ein Muster und die zu durchsuchenden Daten einzugeben. Je nachdem, ob es zu einer Übereinstimmung kommt, liefert das Skript ***wahr*** oder ***falsch*** zurück.

```
#!/usr/bin/perl -w
print 'Geben Sie ein Muster ein: ';
chomp($muster = <STDIN>);
print 'Geben Sie den String ein: ';
chomp($in = <STDIN>);
if ($in =~ /$muster/) { print "wahr\n"; }
else { print "falsch\n"; }
```

Sie werden dieses Skript (oder ein ähnliches) mit zunehmenden Kenntnissen über reguläre Ausdrücke sehr nützlich finden.

Muster und Schleifen

Eine Möglichkeit, Muster in Perl-Skripts einzusetzen, besteht darin, sie wie bisher in `if`-Bedingungen zu verwenden. In skalarem Booleschen Kontext evaluieren die Tests zu ***wahr*** oder ***falsch***, je nachdem ob in den Daten eine Übereinstimmung zu dem Muster gefunden wird oder nicht. Eine andere Möglichkeit, ein Muster zu verwenden, wäre als Bedingung in einer Schleife mit der Option `/g` am Ende des Musters:

```
while (/muster/g)  {
   #  Schleife
}
```

Die Option `/g` wird verwendet, um alle Muster in dem gegebenen String zu prüfen (hier `$_`, Sie können aber auch den Operator `=~` verwenden, um in anderen Daten eine Übereinstimmung zu finden). Im Falle eines `if`-Tests ist die Option `/g` nicht von Belang. Der Test wird bei der ersten gefundenen Übereinstimmung ***wahr*** zurückliefern. Im Falle einer `while`- (oder `for`-) Schleife bewirkt das `/g`, dass die Bedingung bei jedem Vorkommen des Musters im String ***wahr*** ist - und die Anweisungen im Block werden ebensooft ausgeführt.

> *Wir sprechen immer noch davon, Muster in einem skalaren Kontext zu verwenden. Hier bewirkt das `/g` lediglich interessante Varianten in Schleifen. Zu den Mustern im Listenkontext kommen wir morgen.*

Ein weiteres Beispiel: Zählen

Im folgenden sehen Sie ein Skript, das von der eben besprochenen Möglichkeit, Muster in Schleifen zu verwenden, Gebrauch macht. Das Skript verarbeitet eine oder mehrere Dateien und zählt, wie oft das gesuchte Muster in der Datei vorkommt. Mit diesem Skript könnten Sie zum Beispiel zählen, wie oft Ihr Name in einer Datei erwähnt wird, oder herausfinden, wie viele Besucher Ihrer Website von America Online (`aol.com`) kamen. Ich habe dieses Skript mit einem Entwurf dieses Kapitels getestet und festgestellt, dass ich das Wort Muster bisher über 180mal verwendet habe.

Listing 9.2 zeigt dieses einfache Skript:

Listing 9.2: zaehlen.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $pat = ""; # wonach gesucht wird
4:  $count = 0; # Anzahl der Vorkommen
```

```
5:
6:  print 'Nach was soll gesucht werden? ';
7:  chomp($pat = <STDIN>);
8:  while (<>) {
9:      while (/$pat/g) {
10:         $count++;
11:     }
12: }
13:
14: print "/$pat/ $count mal gefunden.\n";
```

Wie bei allen bisher erstellten Skripts, die Dateien mit <> durchlaufen, müssen Sie auch dieses von der Befehlszeile mit dem Namen einer Datei aufrufen:

```
% zaehlen.pl logdatei
Nach was soll gesucht werden? aol.com
/aol.com/ 3456 mal gefunden.
%
```

Nichts in Listing 9.2 sollte für Sie absolut neu sein, auch wenn es einige Punkte gibt, die Beachtung verdienen. Zur Erinnerung: `while` zusammen mit dem Zeileneingabeoperator (<>) weist die einzelnen Zeilen der Eingabe der Standardvariablen `$_` zu. Da Muster standardmäßig diesen Wert vergleichen, benötigen wir keine temporäre Variable, die jede Eingabezeile aufnimmt. Die erste `while`-Schleife (Zeile 8) liest die Zeilen der Eingabedateien. Die zweite `while`-Schleife durchsucht jede einzelne Zeile der Eingabe und inkrementiert `$count` jedesmal, wenn das Muster in einer Zeile gefunden wird. So erhalten wir die Gesamtzahl der Vorkommen des vorgegebenen Musters, sowohl für jede Zeile als auch für alle Zeilen in der Eingabe zusammen.

Eine wichtige Sache zu diesem Skript möchte ich noch anmerken: Wenn Sie eine Suche nach einer Textpassage statt nach einem einzigen Wort durchführen - zum Beispiel alle Vorkommen Ihres Vor- und Nachnamens suchen - kann es passieren, dass die Textpassage über zwei Zeilen geht. Ein Skript wie dieses wird solche Vorkommen nicht finden, da keine der Zeilen das komplette Suchmuster enthält. Morgen lernen Sie, wie Sie nach einem Muster suchen, dass sich über mehrere Zeilen erstreckt.

Musterpriorität

Vielleicht erinnern Sie sich noch an unsere kleine Tabelle in Kapitel 3, »Weitere Skalare und Operatoren«, die die Prioritäten der verschiedenen Operatoren verdeutlichte und es Ihnen erlaubte, herauszufinden, welche Teile eines größeren Ausdrucks zuerst ausgewertet werden. Metazeichen in Mustern unterliegen ebenfalls Prioritätsregeln, die festlegen, auf welche Zeichen oder Zeichengruppen sich die Metazeichen beziehen. In Tabelle 9.2 sind diese Prioritäten aufgeführt, wobei die Zeichen weiter oben in der Tabelle enger binden als die Zeichen weiter unten.

Zeichen	Bedeutung
( )	Gruppe und Speicher
? + * { }	Quantifizierer
x \x $ ^ (?=) (?!)	Zeichen, Anker, Vorausschau
|	Alternativen

Tabelle 9.2: Prioritäten der Metazeichen für Muster

Wie schon bei den Ausdrücken können Sie mit `()` gruppieren, damit diese als Folge ausgewertet werden.

Noch sind Ihnen nicht alle Metazeichen bekannt. Morgen werden wir mehr davon kennenlernen.

Vertiefung

In diesem Kapitel haben Sie die Grundlagen der regulären Ausdrücke kennengelernt. Morgen stelle ich Ihnen weitere Einsatzbereiche für reguläre Ausdrücke vor. Für den Fall, dass Sie noch mehr Informationen zu dem bisher Gelernten benötigen, möchte ich Ihnen die Dokumentation ***perlre***-Manpage ans Herz legen.

Weitere Einsatzbereiche für Muster

Zu Beginn dieses Kapitels haben Sie gelernt, dass der Operator =~ Muster mit skalaren Variablen vergleicht. Neben =~ können Sie auch !~ verwenden:

```
$Sache !~ = /muster/;
```

!~ ist die logische Negation von =~. Mit anderen Worten, es wird nur ***wahr*** zurückgeliefert, wenn das Muster NICHT in $Sache gefunden wird.

Eine weitere nützliche Funktion für Muster ist die Funktion pos, die an sich der Funktion index sehr ähnlich ist - nicht aber in Zusammenhang mit Mustern. Sie können diese Funktion verwenden, um die exakte Position zu bestimmen, an der eine Übereinstimmung gefunden wurde (mit Hilfe von m//g), oder um eine Mustersuche an einer bestimmten Position in einem String zu starten. Die pos-Funktion übernimmt einen skalaren Wert (oft eine Variable) als Argument und liefert den Offset des Zeichens ***nach*** dem letzten Zeichen der Übereinstimmung zurück.
Ein Beispiel:

```
$finde = "123 345 456 346";
while ($finde =~ /3/g) {
   print pos $finde,"\n";
}
```

Dieses Codefragment gibt alle Positionen des Strings $finde aus, an denen die Zahl 3 erscheint (3, 5, 13).

Weitere Informationen zu der pos-Funktion finden Sie in der Hilfsdokumentation ***perlre***-Manpage.

Musterbegrenzer und Escape-Zeichen

Alle Muster, die wir bisher betrachtet haben, begannen und endeten mit einem Slash und enthielten in der Mitte die Zeichen oder Metazeichen, die überprüft werden sollten. Die Slash-Zeichen selbst sind ebenfalls Metazeichen. Das bedeutet, dass, wenn Sie nach einem Slash suchen wollen, Sie ihm einen Backslash voranstellen müssen. Dies ist für Muster mit vielen Slash-Zeichen recht mühselig - zum Beispiel für Unix- Pfadnamen, die alle von Slash-Zeichen getrennt werden. Da kann es schnell passieren, dass Sie mit einem Muster arbeiten, das folgendermaßen aussieht:

```
/\/usr(\/local)*\/bin\//;
```

Das liest sich nur sehr schwer (schwerer als so mancher andere reguläre Ausdruck). Zum Glück hat Perl dafür eine Lösung: Sie müssen ein Muster nicht unbedingt in // einschließen. Sie können dafür jedes beliebige nichtalphanumerische Zeichen verwenden. Der einzige Haken dabei ist, dass Sie bei Verwendung eines anderen Zeichens das m im Ausdruck m// nicht weglassen dürfen (Sie können die Begrenzer auch substituieren, müssen dann aber s/// verwenden). Auch für die neuen Begrenzer gilt, dass Sie diesen Escape-Zeichen voranstellen müssen, wenn Sie sie innerhalb des Musters verwenden wollen. So könnte zum Beispiel der obige Ausdruck auch wie folgt geschrieben werden:

```
m%/usr(/local)*/bin/%;
```

Es kann Ihnen auch passieren, dass ihr Suchmuster eine Reihe von nichtalphanumerischen Zeichen enthält, die gleichzeitig Metazeichen für Muster sind. Dies bedingt eine ganze Menge von Backslash-Zeichen für die Metazeichen, so dass das Muster nur schwer zu lesen ist. Mit dem Escape-Zeichen \Q können Sie die Musterverarbeitung für eine bestimmte Anzahl an Zeichen ausschalten und mit \E wieder einschalten. Wenn Sie zum Beispiel nach einem Muster suchen, das die folgenden Zeichen {(^*)} (aus was für Gründen auch immer) enthält, würde das folgende Muster genau nach diesen literalen Zeichen suchen:

```
/\Q{(^*)}\E/;
```

Die Musterverarbeitung mit `\Q` auszuschalten, ist auch für die Variableninterpolation innerhalb von Mustern nützlich, um ungewöhnliche Ergebnisse bei der Eingabe von Suchmustern zu verhindern:

```
/Von:\s*\Q$von\E/;
```

Zusammenfassung

Pattern Matching und reguläre Ausdrücke sind zweifelsohne eine der größten Stärken von Perl. Andere Sprachen mögen zwar über Bibliotheken und Funktionen für reguläre Ausdrücken verfügen, aber nur Perl allein arbeitet so intensiv mit Pattern Matching, einer Technik, die eng mit vielen anderen Aspekten der Sprache verbunden ist. Perl ohne reguläre Ausdrücke ist lediglich eine weitere seltsam anmutende Sprache. Perl mit regulären Ausdrücken hingegen ist unglaublich leistungsstark.

Heute haben Sie alles über Muster für Pattern-Matching-Operationen kennengelernt: wie man sie erstellt, sie verwendet, Teile davon speichert und sie mit anderen Teilen von Perl zusammen verwendet. Ich habe Ihnen die verschiedenen Metazeichen vorgestellt, die Sie innerhalb der regulären Ausdrücke verwenden können: Metazeichen zum Verankern eines Musters (`^`, `$`, `\B`, `\b`), zum Erzeugen einer Zeichenklasse (`[]` und `[^]`), zum Wechseln zwischen verschiedenen Mustern (`|`) und zum Suchen von mehrfachen Vorkommen eines Zeichens (`+`, `*`, `?`).

Mit Hilfe des Ausdrucks `m//` können Sie Muster auf Strings anwenden. Standardmäßig greifen Muster dabei auf den in der Variablen `$_` gespeicherten String zurück, es sei denn, Sie verwenden den Operator `=~`, um das Muster auf eine andere Variable anzuwenden.

Morgen werden wir das hier Besprochene weiter vertiefen, die bekannten Muster um weitere Muster ergänzen und neue und bessere Möglichkeiten kennenlernen, die Muster einzusetzen.

Fragen und Antworten

Frage:
Was ist der Unterschied zwischen `m//` und nur `//`?

Antwort:
Im Grunde genommen gibt es keinen Unterschied. Das `m` ist optional, es sei denn Sie verwenden ein anderes Zeichen als Musterbegrenzer. Ansonsten bewirken beide Formen dasselbe.

Frage:
Alternation bewirkt eine logische ODER-Verknüpfung in einem Muster. Wie erreiche ich ein logisches UND?

Antwort:
Am einfachsten verwenden Sie dazu mehrere Muster und verbinden sie mit dem `&&`- oder `and`-Operator:

```
/muster1/ && /muster2/;
```

Wenn Sie wissen, in welcher Reihenfolge die zwei Muster erscheinen, können Sie auch folgendes eingeben:

```
/muster1.*muster2/
```

Frage:
Ich habe ein Muster, das nach Zahlen sucht: `/\d*/`. Es sucht nach Zahlen, aber darüber hinaus auch nach allen anderen Strings. Was mache ich falsch?

Antwort:
Sie verwenden `*`, wo Sie eigentlich `+` meinen. Zur Erinnerung: `*` bedeutet »keine oder mehrere Vorkommen«. Das heißt, dass auch wenn Ihr String keine Zahl aufweist, immer noch eine Übereinstimmung gegeben ist - Sie haben dann eben »keine« Vorkommen. `+` wird verwendet, wenn das Muster zumindest einmal vorkommen soll.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Definieren Sie die Begriffe »Pattern Matching« und »reguläre Ausdrücke«.
2. Für welche Art von Aufgaben wird Pattern Matching sinnvollerweise eingesetzt? Nennen Sie drei!
3. Was bewirken die folgenden drei Muster?

```
  /ice\s*cream/
/\d\d\d/
/^\d+$/
/ab?c[,.:]d/
/xy|yz+/
/[\d\s]{2,3}/
/"[^"]"/
```

4. Gehen Sie davon aus, dass `$_` den Wert `123 kazoo kazoo 456` enthält. Wie lautet das Ergebnis der folgenden Ausdrücke?

```
  if (/kaz/) {  # wahr oder falsch?
while (/kaz/g) {  # was passiert?
if (/^\d+/) {  # wahr oder falsch?
if (/^\d?\s/) {  # wahr oder falsch?
if (/\d{4}/) {  # wahr oder falsch?
```

Übungen

1. Schreiben Sie Muster für die folgenden Aufgaben:

- Die ersten Wörter in einem Satz finden (das sind die Wörter, die mit einem Großbuchstaben beginnen und auf die Kombination Leerzeichen-Punkt folgen)
- Prozentangaben finden (alle Dezimalzahlen gefolgt von einem Prozentzeichen)
- Alle Zahlen finden (mit oder ohne Dezimalpunkt, positiv oder negativ)

2. FEHLERSUCHE: Was ist an folgendem Code falsch?

```
 print 'Geben Sie einen String ein: ';
chomp($input = <STDIN>);
print 'Wonach soll gesucht werden? ';
chomp($pat = <STDIN>);
if (/$pat/) {
   # Muster gefunden, bearbeiten!
}
```

3. FEHLERSUCHE: Und was ist an folgendem Code falsch?

```
 print 'Wonach soll gesucht werden? ';
chomp($pat = <STDIN>);
while (<>) {
    while (/$pat/) {
        $count++;
    }
}
```

4. Gestern haben wir ein Skript namens `mehrnamen.pl` erstellt, mit dem Sie eine Liste von Namen sortieren und nach verschiedenen Teilen durchsuchen konnten. Der Suchteil bestand aus einem verworrenen Mechanismus aus `each` und `grep`, um das Muster zu finden. Schreiben Sie diesen Teil des Skripts neu und verwenden Sie statt dessen Muster.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. ***Pattern Matchin***g ist ein Perl-Konzept, bei dem Muster formuliert werden, die dann auf einen String oder einen anderen Satz von Daten angewendet werden. ***Reguläre Ausdrücke*** sind eine Sprache, mit der diese Muster geschrieben werden.
2. Es gibt viele Anwendungsbereiche für Pattern Matching - sie werden nur durch Ihre Vorstellungskraft begrenzt. Hier eine kleine Auswahl:

1. a. Eingabeüberprüfung
1. b. Vorkommen in einem String zählen
1. c. Daten aus einem String auf der Basis bestimmter Kriterien extrahieren
1. d. Einen String in verschiedene Elemente zerlegen
1. e. Ein spezielles Muster durch einen anderen String ersetzen
1. f. Regelmäßige (oder unregelmäßige) Muster in einem Datensatz suchen

3. Die Antworten lauten wie folgt:

1. a. Dieses Muster findet die Zeichen `ice` und `cream`, wenn diese durch kein oder mehrere Leerzeichen getrennt sind.
1. b. Dieses Muster findet drei Ziffern in einer Reihe.
1. c. Dieses Muster findet eine oder mehrere Ziffern, die allein in einer Zeile stehen.
1. d. Dieses Muster findet ein `a`, ein optionales `b`, ein `c`, ein Komma, Punkt oder Doppelpunkt und ein `d`. Gefunden wird also `ac.d` oder auch `abc,d`, aber nicht `abcd`.
1. e. Dieses Muster findet entweder `xy` oder `y` mit einem oder mehreren `z`.
1. f. Dieses Muster findet entweder eine Ziffer oder ein Leerzeichen, das mindestens zweimal, aber nicht öfter als dreimal auftritt.
1. g. Dieses Muster findet alle Zeichen zwischen dem öffnenden und dem schließenden Anführungszeichen.

4. Die Antworten lauten:

1. a. ***wahr***
1. b. Die Schleife wird für jedes Vorkommen von `kaz` in dem String durchlaufen (hier zweimal).
1. *c.* ***wahr***. Das Muster findet eine oder mehrere Ziffern zu Beginn der Zeile.
1. *d.* ***falsch***. Dieses Muster findet 0 oder eine Ziffer zu Beginn der Zeile gefolgt von einem Leerzeichen. Die drei Ziffern in unserem String werden nicht gefunden.
1. *e.* ***falsch***. Dieses Muster findet vier Ziffern in einer Reihe; wir haben jedoch hier nur drei Ziffern.

Lösungen zu den Übungen

1. Wie meistens bei Perl gibt es mehrere Lösungswege zu einem Problem. Hier einige der möglichen Lösungen:

```
  /[.!?"]\s+[A-Z]\w+\b/
/d+%/
/[+-]\d+\.?\d+/
```

2. Die Variable, auf die das Muster angewendet wird, und die Variable, in der sich die eigentlichen Daten befinden, sind nicht identisch. Das Muster in der `if`- Anweisung versucht das Muster mit `$_` abzugleichen, aber wegen der zweiten Zeile befindet sich die eigentliche Eingabe in `$input`. Verwenden Sie statt dessen diesen `if`-Test:

```
if ($input =~ /$spat/)  {
```

3. Diese Übung war gemein, da der Code syntaktisch korrekt ist. Die zweite `while`- Schleife - das ist die mit dem Muster - sucht nach dem Muster in `$_`, was korrekt ist. Aber der Test ist ein einfacher ***wahr***/***falsch***-Test: Existiert dieses Muster? Die erste Zeile, die dieses Muster enthält, wird ***wahr*** liefern, und dann wird `$count` inkrementiert. Danach wird der Test jedoch erneut durchgeführt, er ist immer noch ***wahr,*** und der Zähler wird wieder inkrementiert - immer und immer wieder. Es gibt nichts, was die Iteration der Schleife stoppt.
1. Mit der Option `/g` hinter dem Muster in der `while`-Schleife erreichen Sie den besonderen Fall, dass die `while`-Schleife nur so oft durchlaufen wird, wie das Muster im String gefunden wurde, und dann abbricht.

Wenn Sie Muster innerhalb von Schleifen verwenden, dürfen Sie das `/g` nicht vergessen.

4. Der einzige Teil, der geändert werden muss, sind die Zeilen, die das `@keys`-Array erzeugen (und in denen mit `grep` nach dem Muster gesucht wird). Hier werden wir eine `foreach`-Schleife verwenden und in dieser Schlüssel und Wert testen. Wir fügen noch die Option `/i` hinzu, um nicht zwischen Groß- und Kleinschreibung unterscheiden zu müssen, und setzen dann die `@keys`-Liste auf die leere Liste zurück, so dass sie nicht von einem Suchlauf zum nächsten immer länger wird. Hier sehen Sie die neue Version:

```
 } elsif ($in eq '3') {      # sucht einen Namen (1 oder mehr)
    print "Wonach soll gesucht werden? ";
    chomp($search = <STDIN>);
    @keys = ();
    foreach (keys %names) {
        if (/$search/i or $names{$_} =~ /$search/i) {
            push @keys, $_;
        }
    }
    if (@keys) {
        print "Gefundene Namen: \n";
        foreach $name (sort @keys) {
            print "   $names{$name} $name\n";
        }
    } else {
        print "Keine gefunden.\n";
    }
}
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Erweiterte Möglichkeiten regulärer Ausdrücke

Gestern haben wir uns den Grundlagen der regulären Ausdrücke (regular expressions) gewidmet. Sie haben die wichtigsten Metazeichen kennengelernt und wissen jetzt, wie man sie verwendet, um Muster in Strings zu finden. Heute, im zweiten Teil unserer `regex`-Saga, bauen wir auf diesem Wissen auf und untersuchen etwas komplexere Möglichkeiten für den Einsatz regulärer Ausdrücke. In diesem Kapitel zeige ich Ihnen,

- wie man das, was durch einen regulären Ausdruck gefunden wurde, extrahiert,
- wie man Muster in skalarem und in Listenkontexten verwendet,
- wie man Muster für Suchen&Ersetzen-Operationen verwendet,
- wie man die `split`-Funktion sonst noch einsetzen kann und
- wie man Muster über mehrere Zeilen vergleicht.

Übereinstimmungen extrahieren

Mit Hilfe von Mustern im Booleschen skalaren Kontext einer `if`- oder einer Schleifenbedingung können Sie feststellen, ob Ihr Muster mit einem Teil eines Strings übereinstimmt oder nicht. Auf diese Frage gibt es jedoch nur zwei Antworten: *Ja* oder *Nein*. Das mag zwar zur Überprüfung der Eingabe oder zum Ermitteln mehrerer Vorkommen eines Musters in einem String ganz nützlich sein, ist aber nur eine Seite der Medaille. Die Antworten *Ja* oder *Nein* sind zwar recht zweckmäßig, aber noch nützlicher ist es, wenn man herausfinden kann, welche Daten genau dem Muster entsprechen, um dann diese Daten später in dem Muster wiederzuverwenden oder eine Liste aller gefundenen Übereinstimmungen aufzubauen.

Ob das Vorkommen, das Sie mit einem Muster finden, nutzbringend ist oder nicht, hängt natürlich vom Muster ab. Lautet Ihr Muster `/abc/`, dann werden auch die gefundenen Daten `abc` lauten, was Ihnen natürlich bereits vorher bekannt war. Wenn jedoch Ihr Muster ungefähr so aussieht `/\+.*/` (suche ein + dann eine beliebige Anzahl von Zeichen), kann Ihre Übereinstimmung aus einem beliebigen Satz von Zeichen bestehen, die zufällig nach einem +-Zeichen stehen. Dass man auf die gefundenen Vorkommen zugreifen kann, ist eine ganz wichtige Eigenschaft der regulären Ausdrücke.

In Perl gibt es mehrere Wege, um auf Fundstellen zuzugreifen, und verschiedene Möglichkeiten, diese - hat man sie erst einmal gefunden - zu nutzen. Gibt es eine Übereinstimmung, können Sie weiter hinten in dem gleichen Muster darauf Bezug nehmen, Sie können die Übereinstimmung in einer skalaren Variablen speichern oder eine Liste aller Übereinstimmungen anlegen. In diesem Abschnitt werden wir diese Möglichkeiten der Reihe nach besprechen.

Mit Klammern Rückbezüge herstellen

Gestern habe ich Ihnen gezeigt, wie Sie mit Klammern Teile des Musters zusammenfassen können oder wie sich damit die Prioritäten beim Mustervergleich ändern lassen. Der dritte und vielleicht wichtigste Einsatzbereich von Klammern ist, die gefundene Übereinstimmung zu speichern, um dann weiter hinten zum Aufbau eines komplexeren regulären Ausdrucks auf diese Übereinstimmung Bezug zu nehmen. Dieser Mechanismus, Übereinstimmungen zu sichern, wird in der Sprache der regulären Ausdrücke oft auch als ***Rückbezug*** bezeichnet.

Betrachten wir dazu folgendes Beispiel. Angenommen wir suchen Zeilen, die mit dem gleichen Wort beginnen und enden. Dabei ist es unerheblich, welches Wort das ist, solange es am Anfang und am Ende der Zeile gleich lautet. Sie könnten dazu zwei Mustertests, eine Schleife und mehrere `if`-Anweisungen aufsetzen. Besser wäre es jedoch, zu Beginn der Zeile das erste Wort abzufragen, den Wert zu speichern und dann zu schauen, ob dieses Wort auch am Ende der Zeile vorkommt. Und so sähe ein derartiger Ausdruck aus:

```
/^(\S+)\s.*\1$/
```

Ich möchte Ihnen diesen Ausdruck Zeichen für Zeichen aufschlüsseln. Das erste Zeichen ist ein Caret (^) und

bezieht sich auf den Anfang der Zeile. Direkt daran schließt sich eine Klammer an, die ein Muster einschließt, das für später gespeichert werden soll. `\S` ist ein beliebiges Nicht-Whitespace-Zeichen, und `\S+` bedeutet ein oder mehrere beliebige Nicht-Whitespace-Zeichen. Die schließende Klammer beendet den Teil, der gespeichert werden soll. Können Sie mir noch folgen? Das Muster innerhalb der Klammer schaut am Anfang der Zeile nach einer Reihe von Zeichen, gefolgt von einem Whitespace-Zeichen (Leerzeichen, Tabulator etc.).

Doch fahren wir fort. Anschließend haben wir ein einziges Whitespace-Zeichen (`\s`), kein oder mehrere Zeichen beliebiger Art (`.*`), `\1` und dann das Zeichen für Ende der Zeile (`$`). Doch wozu der Ausdruck `\1`? Dieser Code ist eine Referenz auf das, was wir als Übereinstimmung in der Klammer gefunden haben. `\1` besagt: »Setze das, was du im ersten Klammerpaar gefunden hast, hierher.« Damit wird genau das, was wir innerhalb dieser Klammer gefunden haben, später in dem Muster selbst erscheinen. Damit das gesamte Muster ***wahr*** zurückliefert, muss die referenzierte Übereinstimmung also auch am Ende der Zeile erscheinen. Alle Teile des Musters müssen gefunden werden, nicht nur der Klammerinhalt.

Angenommen, Sie haben folgende Zeile:

```
»Perl ist zur schnellen Skripterstellung am besten geeignet.
Wenn Sie Ihre Arbeit gut machen wollen, wählen Sie Perl.«
```

(Gut, ich gebe zu, das sind zwei Zeilen. Nehmen Sie jedoch einfach an, es wäre eine einzige Zeile innerhalb der Variablen `$_`). Wird das obige Muster, angewendet auf diese Zeile, eine Übereinstimmung (gleiches Wort am Anfang und am Ende der Zeile) finden? Schauen wir mal. Das Muster sucht zuerst den Zeilenanfang, dann alle Nicht- Whitespace-Zeichen bis zum ersten einfachen Whitespace-Zeichen. Das Wort »Perl« und das anschließende Leerzeichen stimmen mit dem Muster überein. (Beachten Sie, dass das Whitespace-Zeichen außerhalb der Klammer steht und somit nicht mitgespeichert wird.) Der nächste Teil des Musters (`.*`) schluckt alle Zeichen bis zum Ende der Zeile. Hiernach halten wir Ausschau nach einem Leerzeichen und einem weiteren Vorkommen der zuvor gefundenen Übereinstimmung, die wir am Anfang der Zeile gespeichert haben. Denken Sie daran, hinter `\1` verbirgt sich die am Anfang gefundene Übereinstimmung, so dass wir nach dem Wort `Perl` suchen. Nachdem wir `Perl` gefunden haben, erwartet das Muster mit `$` noch das Ende der Zeile - aber nein, was ist das? Die Zeile ist ja noch gar nicht zu Ende! Da steht ja noch ein Punkt! Und wegen diesem Punkt liefert das gesamte Muster ***falsch*** zurück.

Wir können das jedoch problemlos beheben. Durch folgende Änderung am Muster kann man die Interpunktion berücksichtigen:

```
/^(\S+)\s.*\s\1[.!?"]$/
```

Am wichtigsten ist jedoch dabei die Tatsache, dass sich `\1` auf das bezieht, was durch das in Klammern stehende Muster gefunden wurde. Das ganze Muster ist nur erfolgreich, wenn der ermittelte Klammerinhalt noch einmal dort im String auftaucht, wo es durch die Referenz `\1` vorgegeben wird.

Vielleicht wundern Sie sich, warum es gerade `\1` heißt. Die Bezeichnung `\1` resultiert daraus, dass es nur eine Klammer gibt, in der ein Muster gespeichert wird. Möglich sind durchaus mehrere gespeicherte Übereinstimmungen in einem Muster, die dann jeweils von Klammern eingeschlossen sind und mit den Zahlen `\1`, `\2`, `\3` und so weiter angesprochen werden (gezählt wird von links nach rechts). Die Zahlen werden jeweils den ***öffnenden*** Klammern zugewiesen - was bedeutet, dass Sie eine Klammer nicht schließen müssen, bevor Sie eine neue öffnen. Sie können somit die Muster, für die Sie eine Übereinstimmung suchen, verschachteln und mit der Zahl dann auf die jeweilige Übereinstimmung Bezug nehmen.

Seien Sie jedoch vorsichtig mit dem Einsatz von Klammern - unabhängig davon, ob Sie damit Gruppen definieren, um die Prioritäten zu ändern, oder Übereinstimmungen sichern wollen - Perl wird auf alle Fälle die Werte speichern. Sie können dies verhindern, indem Sie eine besondere Form der Klammer verwenden: `(?:muster)` anstelle von `(muster)`. Mehr dazu erfahren Sie im Abschnitt »Vertiefung« zu diesem Kapitel.

Übereinstimmungen in Variablen speichern

Rückbezüge ermöglichen es Ihnen, eine für einen Teil des Musters gefundene Übereinstimmung im Muster selbst wieder zu referenzieren. Sie können auf den bereits gefundenen Teil eines Musters, aber auch von außerhalb des Musters zugreifen. Zusätzlich zu der Numerierung der Rückbezüge durch `\1`, `\2` und so weiter weist Perl den Werten der Teilfundstellen Skalarvariablen `$1`, `$2` und so weiter zu. Dieser Mechanismus erweist sich als besonders

praktisch, wenn man Teile aus einem String oder anderen Daten extrahieren möchte. Sehen Sie dazu folgendes Beispiel, bei dem das erste Wort aus einem String herausgelöst wird:

```
if (/^(\S+)\s/) {
   print "Erstes Wort: $1\n";
}
```

Hierbei wird Perl, wenn es eine Entsprechung zu dem Muster findet (das heißt, wenn es einen Zeilenanfang gefolgt von einem oder mehreren Nicht-Whitespace-Zeichen und einem Whitespace-Zeichen findet), »`Erstes Wort:`« plus der gefundenen Übereinstimmung ausgeben. Wenn die Daten kein erstes Wort aufweisen (zum Beispiel wenn es sich um einen leeren String handelt oder es kein Whitespace- Zeichen in dem String gibt), wird nichts ausgegeben, da der Test ***falsch*** zurückliefert. Wie bereits erwähnt, muss das ganze Muster und nicht nur der Teil in Klammern stimmen, damit ***wahr*** zurückgeliefert werden kann.

Auf eines möchte ich Sie im Zusammenhang mit den Übereinstimmungsvariablen noch hinweisen: Die Werte dieser Variablen sind lokal zu ihrem Block, können nur gelesen werden und sind sehr kurzlebig. Schon beim nächsten Versuch, eine Übereinstimmung zu finden, verschwindet ihr alter Wert. Die Werte verschwinden auch, wenn ein Block endet. Mit anderen Worten, wenn Sie die Werte dieser Variablen ändern wollen, müssen Sie sie zur Sicherung in einer anderen Variablen oder in einer Liste ablegen. Übereinstimmungsvariablen dienen nur der vorübergehenden Speicherung.

Übereinstimmungen und Kontext

Bis jetzt sind uns Muster nur in einem skalaren Kontext und dort vornehmlich in booleschen Tests begegnet. Für die Verwendung von Mustern in einem skalaren booleschen Kontext gibt es zwei Regeln:

- Ein Test wie `/abc/` liefert 1 (***wahr***) zurück, wenn das Muster in dem gegebenen String (`$_` wenn es keine Variable ist oder `=~` für alles andere) gefunden wurde, ansonsten ***falsch***. Sinnvoll vor allem in Bedingungen.
- Die Option `/g` nach dem Muster ermöglicht die Iteration durch den String. Das Muster liefert für jede Übereinstimmung 1 (***wahr***) zurück und ***falsch*** am Ende des Strings. Wird in `while`-Schleifen eingesetzt.

In beiden Fälle füllen Klammern innerhalb des Musters die Übereinstimmungsvariablen `$1`, `$2`, `$3` und so weiter, und Sie können diese Werte dann innerhalb des Bedingungs- oder Schleifenblocks oder bis zum nächsten Mustervergleich verwenden.

Für Muster in einem Listenkontext gelten allerdings andere Regeln (Überraschung, Überraschung):

- Ein Muster, das in Klammern Teilmuster enthält, die gespeichert werden, gibt eine Liste der ersten Teilmuster zurück, die eine Übereinstimmung ergaben (`$1`, `$2`, `$3` und so weiter) - und natürlich auch die einzelnen Variablen. Gibt es in dem Muster keine von Klammern eingeschlossenen Teilmuster, wird die Liste (`1`) zurückgegeben (das heißt eine Liste mit einem Element: der Zahl 1).
- Die Option `/g` nach dem Muster liefert eine Liste aller Teilmuster zurück, für die im String eine Übereinstimmung gefunden wurde.
- Wird das Muster überhaupt nicht gefunden, ist das Ergebnis eine leere Liste `()`.

Da Muster in einem Listenkontext die gefundenen Übereinstimmungen als Liste zurückgeben, könnten Sie die Muster nutzen, um Ihre Daten in Elemente aufzusplitten. Hier eine Möglichkeit, einen Namen in die beiden Bestandteile Vor- und Nachname zu zerlegen:

```
($fn, $ln) = /^(\w+)\s+(\S+)$/
```

Eine Bemerkung zur Gier

In Kapitel 2, »Mit Strings und Zahlen arbeiten«, hatten wir eine kurze Diskussion über die Wahrheit, und jetzt werden wir über Gier sprechen. Vielleicht kommen wir weiter hinten im Buch auch noch auf Gerechtigkeit und Neid zu sprechen.

Doch Humor beiseite, eine heikles Problem beim Extrahieren von Mustern betrifft die Art und Weise, wie sich die quantifizierenden Metazeichen verhalten. Die Metazeichen `+`, `*`, `?` und `{}` werden auch gierige Metazeichen

genannt, da sie im Falle einer Übereinstimmung so viele Zeichen wie möglich in die Übereinstimmung mit aufnehmen bis hin zum letzten Zeichen in der Zeile.

Normalerweise verhalten sich Muster so, dass sie, wenn sie gefundene Übereinstimmungen zurückliefern sollen (das heißt, wenn das Muster in einem Listenkontext verwendet wird), die erste Übereinstimmung zurückliefern. Betrachten wir beispielsweise folgenden Ausdruck:

```
@x = /(\d\d\d)/;
```

Angenommen die Daten in `$_` sehen folgendermaßen aus:

```
3443 32 784 2344 123 78932
```

Das Array `@x` endet als eine Liste von einem Element, den ersten drei Ziffern, in diesem Fall `344`. Das Muster bricht immer nach dem ersten positiven Vergleich ab.

Die Quantifizierer `*`, `?` und `{}` unterliegen allerdings anderen Regeln. Nehmen wir als Beispiel die Daten von oben und versuchen wir eine Übereinstimmung für folgendes Muster zu finden:

```
/(\d*)/;
```

Da * als »keines oder mehrere der vorhergehenden Zeichen« definiert ist, könnten Sie davon ausgehen, dass der Vergleich mit dem Muster abgebrochen wird, sobald diese Bedingung erfüllt ist, das heißt, sobald das erste passende Zeichen eingelesen wurde - was in unserem Zahlenbeispiel von oben die Zahl 3 zurückliefern würde. Die Ziffer entspricht zwar dem Muster, aber man darf nicht vergessen, dass * ein gieriger Quantifizierer ist, der bis an die Grenze des Möglichen alle Zeichen aufsaugt. Das Ergebnis eines Vergleichs dieses Musters mit dem Zahlenstring lautet daher 3443. Der *-Quantifizierer geht Zahl um Zahl vor, bis er auf ein Leerzeichen trifft. Ein Leerzeichen ist keine Zahl, und deshalb muss der Quantifizierer hier stoppen.

Hier sehen Sie ein noch schwierigeres Beispiel:

```
/'(.*)'/
```

Auf den ersten Blick scheint dieses Muster nach Zeichen in Anführungszeichen zu suchen (und `$1` mit diesen Zeichen zu füllen). Versuchen Sie jedoch einmal, dieses Muster auf folgenden String anzuwenden:

```
"Sie sagte, 'Ich möchte diesen Käfer nicht essen,' und dann schlug sie mich."
```

Da die Sequenz `.*` gierig ist, vergleicht sie alle Zeichen zwischen den einfachen Anführungszeichen, liefert sie zurück (`Ich möchte diesen Käfer nicht essen`) und fährt dann damit fort, die restlichen Zeichen bis zum Ende der Zeile zu vergleichen. Da die Sequenz dabei das Anführungszeichen nicht verglichen hat, geht Perl rückwärts und versucht verschiedene Zeichen, bis es auf das Anführungszeichen stößt. Damit erhalten Sie zwar das erwartete Ergebnis, doch Perl wird viel Zeit aufwenden müssen, um zu dem Ergebnis zu gelangen. Es kann sogar noch schlimmer werden. Wenn Sie nämlich mehrere Anführungszeichen im String haben, wird Perl das erste akzeptieren, auf das es beim Rückwärtsgehen stößt. Nehmen wir folgenden String als Beispiel:

```
"'Ich verachte dich,' sagte sie und warf eine Kanne nach mir. 'Ich wünschte du wärest tot.'"
```

Wenn Sie versuchen, das gleiche Muster auf diesen String anzuwenden, wird `$1` letztlich folgenden String enthalten:

```
Ich verachte dich,' sagte sie und warf eine Kanne nach mir. 'Ich wünschte du wärest tot.
```

Angenommen Sie wollten mit dem Muster ursprünglich nur nach dem Inhalt der ersten Anführungszeichen suchen (nur die Wörter `Ich verachte dich`), dann entspricht dieses Ergebnis absolut nicht Ihren Erwartungen.

Quantifizierer scheinen auf den ersten Blick eine clevere Lösung zum Füllen einer Lücke zwischen zwei Mustern zu sein. Aufgrund ihres gierigen Verhaltens sind sie dafür jedoch häufig absolut ungeeignet, und Sie werden nur frustriert werden, wenn Sie versuchen, sie dafür einzusetzen. Die bessere Lösung - sowohl um sicherzustellen, dass

nur gefunden wird, wonach gesucht wurde, als auch um Perl davon abzuhalten, unnötige Zeit mit dem Rückwärtsdurchlaufen eines Strings zu verbringen - ist die Verwendung von negierten Zeichenklassen anstatt der Quantifizierer. Betrachten Sie das Problem einmal nicht als »alle Zeichen zwischen dem öffnenden und dem schließenden Anführungszeichen«, sondern als »ein öffnendes Anführungszeichen, dann einige Zeichen, die kein Anführungszeichen sind, und dann ein schließendes Anführungszeichen«.

Wenn Sie diesen Gedanken in ein Muster umsetzen, sieht das folgendermaßen aus:

```
/"([^"]+)"/
```

Das sind zwar einige Zeichen mehr, und es ist insgesamt etwas schwieriger zu lesen, aber dafür liefert das Muster auch garantiert die Zeichen zwischen den Anführungszeichen zurück und frißt nicht gierig alle Zeichen nach dem schließenden Anführungszeichen, die dann ein Rückwärtssuchen erforderliche machen. Merken Sie sich diese Regel: Wenn Sie ein Muster zwischen Begrenzern vergleichen wollen, verwenden Sie am besten eine negierte Zeichenklasse mit dem schließenden Begrenzungszeichen innerhalb der ekkigen Klammern.

Ein zweiter, weniger effizienter Weg, das gierige Verhalten der Metazeichen `+`, `*`, `?` und `{}` zu unterdrücken, besteht darin, besondere, nichtgierige (»faule«) Versionen dieser Metazeichen zu verwenden: `+?`, `*?`, `??` und `{}?`. Mehr dazu im Abschnitt »Vertiefung«.

Muster für Suchen&Ersetzen-Operationen

Reguläre Ausdrücke lassen sich nicht nur sehr gut zum Suchen eines Musters oder zum Ablegen von Übereinstimmungen in Listen oder Variablen oder ähnlichem nutzen, sondern vor allem auch für das Ersetzen der Vorkommen des Musters durch einen anderen String. Dies entspricht den Suchen&Ersetzen-Operationen Ihres bevorzugten Textverarbeitungssystems oder Editors plus der geballten Leistungsfähigkeit und Flexibilität der regulären Ausdrücke.

Die Syntax, mit der Sie nach etwas suchen, um es dann mit einem anderen Muster zu ersetzen, lautet:

```
s/muster/ersetzung/
```

In dieser Syntax ist `muster` ein regulärer Ausdruck und `ersetzung` der String, durch den die gefundenen Vorkommen zu ersetzen sind. Fehlt die Angabe des Strings, durch den ersetzt werden soll, werden die gefundenen Übereinstimmungen aus dem Gesamtstring gelöscht. Ein Beispiel:

```
S/\s+/ /   # ersetze ein oder mehrere Whitespaces durch ein Leerzeichen
```

Wie bei den normalen Mustern sucht und ersetzt diese Syntax standardmäßig in `$_`. Verwenden Sie den Operator `=~`, wenn Sie in einem anderen String suchen wollen.

Die Suchen&Ersetzen-Syntax ersetzt nur die erste Übereinstimmung und liefert dann 1 zurück. Wird am Ende die Option `/g` (steht für global) angegeben, ersetzt Perl alle Vorkommen des Musters in dem String:

```
s/--/[md]/g  # ersetze zwei Bindestriche durch den Gedankenstrich [md]
```

Sie können, wie bei normalen Mustern, am Ende auch die Option `/i` verwenden, mit der die Suche unabhängig von der Groß- und Kleinschreibung durchgeführt wird (Vorsicht! Das bedeutet jedoch nicht, dass die Ersetzung sich dabei jeweils der Schreibweise anpaßt):

```
s/a/b/gi;   # ersetze [Aa] durch b, global
```

Lassen Sie sich nicht davon abhalten, innerhalb der Suchen&Ersetzen-Muster Klammern oder Übereinstimmungsvariablen zu verwenden. Man kann sie in vielfältiger Weise sinnvoll einsetzen:

```
s/^(\S+\b)/=$1=/g  # setzt =-Zeichen um das erste Wort
s/^(\S+)(\s.*)(\S+)$/$3$2$1/  # tauscht das erste mit dem letzten Wort
```

Anhänger von `sed` werden vermutlich die Verwendung von `/1` und `/2` und so weiter in dem Ersetzungsteil der Suchen&Ersetzen-Operationen befürworten. Doch auch wenn dies in Perl funktioniert (vor allem weil Perl Ihnen diese Referenzen durch Variablen ersetzt), sollten Sie sich deren Verwendung allmählich abgewöhnen. Offiziell ist der Ersetzungsteil in dem Ausdruck `s///` ein ganz normaler String in doppelten Anführungszeichen, und `\1` bedeutet in diesem Kontext an sich etwas anderes.

Mehr zu split

Erinnern Sie sich noch an die `split`-Funktion aus Kapitel 5, »Mit Hashes arbeiten«? Wir haben mit `split` die Vor- und Nachnamen in getrennten Listen ausgeben lassen:

```
($fn, $ln) = split(" ", $in);
```

Damals habe ich Ihnen erklärt, dass die Verwendung von `split` mit einem Leerzeichen in Anführungszeichen ein Sonderfall sei, der sich nur auf Daten anwenden lasse, in denen die Felder durch Whitespace-Zeichen getrennt sind. Um `split` für Daten zu verwenden, die durch irgendein anderes Zeichen getrennt sind oder die einer komplizierteren Verarbeitung bedürfen, um die Elemente zu finden, müssen Sie `split` um einen regulären Ausdruck für das zu vergleichende Muster ergänzen:

```
($fn, $ln) = split(/\s+/, $in);                    # bei Whitespace zerlegen
@nums = split(//, $num);                  # zerlege 123 in (1,2,3)
@fields = split(/\s*,\s*/, $in);                    # zerlege durch Kommata getrennte Felder,
                    # mit oder ohne Whitespace um das Komma
```

Das erste Beispiel, das den String dort aufteilt, wo ein oder mehrere Leerzeichen auftauchen, entspricht im Verhalten zum einem unserem Beispiel aus Kapitel 5 mit dem Leerzeichen in Anführungszeichen und zum anderen einer Version, in der `split` ohne irgendein Muster verwendet wird (die Syntax mit dem Leerzeichen in Anführungszeichen lehnt sich an das Unix-Tool `awk` an, das Strings auf gleiche Art und Weise in Teilstrings zerlegt).

Sie können `split` als drittes Argument auch eine Zahl mitgeben, die angibt, in wie viele Teile die Daten zerlegt werden sollen:

```
($ln, $fn, $daten{$ln}) = split(/,/ $in, 3);
```

Dieser reguläre Ausdruck wäre beispielsweise für folgende Daten sehr nützlich:

```
Jones,Tom,braun,blau,64,32
```

Der `split`-Befehl zerlegt die Daten beim Komma in drei Elemente: den Nachnamen, den Vornamen und den Rest. Die Zuweisung wird den Nachnamen und den Vornamen jeweils in Skalarvariablen ablegen und »den Rest« in einem Hash, das den Nachnamen als Schlüssel verwendet.

Normalerweise enthalten die Teilstrings, die in der endgültigen Liste gespeichert werden, nichts, was dem Suchmuster entspricht. Wenn Sie jedoch im Muster Klammern verwenden, wird alles, was innerhalb dieser Klammern dem Suchmuster entspricht, auch in der endgültigen Liste erscheinen, wobei jede Entsprechung ein eigenes Listenelement darstellt. Angenommen Sie hätten folgenden String:

```
1:34:96:54:0
```

Wenn Sie diesen String an den Doppelpunkten zerlegen (mit dem Muster `/:/`), erhalten Sie eine Liste aller Elemente, die keine Doppelpunkte sind, also eine Liste aller Zahlen. Mit dem Muster `/(:)/` erhalten Sie folgende Liste, die sowohl Teilstrings enthält, die nicht Bestandteil des Musters sind, als auch solche, die dem Klammerinhalt des Musters entsprechen:

```
1, ':', 34, ':', 96, ':', 54, ':', 0
```

split zusammen mit normalem Pattern Matching sollte es Ihnen ermöglichen, Ihre Strings auf fast jede beliebige Weise zu zerlegen. Greifen Sie mit Mustern und Rückbezügen auf die gewünschten Teile zu, und zerlegen Sie den String dann mit split und unter Angabe der Teile, die Sie nicht benötigen, in seine Bestandteile. Jeder, der die Suchmuster korrekt einsetzt und das Format der Eingabedaten versteht, wird keine Schwierigkeiten haben, Daten fast jeden Formats mit nur wenigen Codezeilen zu verarbeiten. In einer Sprache wie C wäre dies nur mit einem wesentlich größeren Aufwand zu bewerkstelligen.

Pattern Matching über mehrere Zeilen

Bis jetzt sind wir beim Pattern Matching von der Annahme ausgegangen, dass sich der Vergleich nur auf einzelne Zeilen (Strings) beschränkt, die aus einer Datei oder von der Tastatur aus eingelesen werden. Diese Annahme besagt, dass der String, der durchsucht wird, keine Zeichen für Zeilenvorschub oder Wagenrücklauf enthält, und dass die Anker für Anfang und Ende der Zeile sich auf den Anfang und das Ende des Strings selbst beziehen. Für den while-Code (<>), den wir bisher geschrieben haben, ist diese Annahme ziemlich sinnvoll.

Oft jedoch wollen Sie ein Muster über mehrere Zeilen hinweg vergleichen, vor allem, wenn Ihre Eingabedaten aus Sätzen oder Absätzen bestehen, deren Zeilengrenzen von der jeweiligen Textformatierung abhängen. Wenn Sie zum Beispiel alle Vorkommen der Bezeichnung »Exegetic Frobulator 5000« auf einer Webseite suchen, wollen Sie nicht nur die Bezeichnungen finden, die auf einer logischen Zeile stehen, sondern auch die, die sich über die Zeilengrenze hinweg in die nächste Zeile erstrecken.

Dazu müssen Sie in zwei Schritten vorgehen. Zuerst müssen Sie Ihre Eingaberoutinen dahingehend ändern, dass die gesamte Eingabe als ein String eingelesen und nicht Zeile für Zeile verarbeitet wird. Damit erhalten Sie einen extrem langen String mit Zeichen für »Neue Zeile« und »Wagenrücklauf«. Zweitens müssen Sie je nach Suchmuster, das Sie verwenden, Perl mitteilen, dass die Neue-Zeile-Zeichen anders zu handhaben sind.

Mehrere Zeilen der Eingabe speichern

Es gibt mehrere Möglichkeiten, Ihre gesamte Eingabe als einen String einzulesen. In einem Listenkontext könnten Sie <> verwenden:

```
@input = <>;
```

Die obige Zeile birgt allerdings ein großes Gefahrenpotential. Wenn nämlich Ihre Eingabe sehr, sehr groß ist, könnte der gesamte zur Verfügung stehende Speicher Ihres Systems in dem Versuch, all diese Daten einzulesen, aufgebraucht werden. Und es gibt keine Möglichkeit, mittendrin abzubrechen. Etwas weniger aggressiv ist der Ansatz, die vornehmlich in Absätzen vorliegenden Daten mit Hilfe der Sondervariablen $/ einzulesen. Wenn Sie $/ auf einen Null-String setzen ($/ = "";), dann liest Perl einen Absatz einschließlich der Neue-Zeile-Zeichen ein und hört auf, sobald es auf zwei oder mehr Neue-Zeile-Zeichen in einer Reihe trifft (dabei gehen wir davon aus, dass Ihre Eingabedaten eine oder mehrere Leerzeilen zwischen den Absätzen enthalten):

```
$/ = "";
while (<>) {   # liest einen Absatz keine, Zeile
   # $_ enthält den gesamten Absatz und nicht nur eine Zeile
}
```

Der dritte Weg, mehrere Zeilen in einen einzigen String einzulesen, besteht darin, verschachtelte while-Schleifen zu verwenden und die eingelesenen Zeilen an einen Eingabestring anzuhängen, bis ein bestimmter Begrenzer erreicht ist. In streng codierten HTML-Dateien zum Beispiel endet ein Absatz mit einem </P>-Tag, so dass Sie in diesen Dateien die Eingabedaten bis zu diesem Punkt einlesen könnten:

```
while (<>) {
   if (/(.*)<\/P>/) {
      $in.=$1
   } else {
      $in.=$_
   }
}
```

Eingabedaten mit Neue-Zeile-Zeichen

Haben Sie erst einmal Ihre mehrzeiligen Eingabedaten in einem String untergebracht und entweder in $_ oder einer anderen Skalarvariablen gespeichert, können Sie damit beginnen, diese Daten nach Mustern zu durchsuchen, die sich über mehrere Zeilen erstrecken. Einiges gilt es jedoch zu beachten, wenn Sie eine Mustersuche bei Daten mit eingebetteten Neue-Zeile-Zeichen durchführen wollen:

- Die Zeichenklasse /s umfaßt die sogenannten Whitespace-Zeichen einschließlich des Neue-Zeile-Zeichens und des Zeilenvorschubs. Bei Verwendung dieser Zeichenklasse haben Sie keine Probleme, Muster wie /George\s+Washington/ zu finden, auch wenn diese nicht in einer Zeile stehen, sondern sich auf zwei Zeilen verteilen.
- Die Ankerzeichen ^ und $ beziehen sich auf den Anfang oder das Ende eines Strings, nicht jedoch auf eingebettete Neue-Zeile-Zeichen. Wenn diese Ankerzeichen ihre Bedeutung auch in einem String beibehalten sollen, der aus mehreren Zeilen besteht, müssen Sie die Option /m verwenden.
- Das Metazeichen Punkt (.) bricht in der Regel am Neue-Zeile-Zeichen ab! Dieses Verhalten läßt sich jedoch mit der Option /s ändern.

Der letzte Punkt hierbei ist am problematischsten. Betrachten wir folgendes Muster, das den Quantifizierer .* verwendet, um nach einer anfänglichen »Von: «-Überschrift den Rest einer Zeile auszulesen:

```
/Von: (.*)/
```

Dieses Muster sucht zuerst nach den Zeichen Von: und füllt dann $1 mit dem Rest der Zeile. Dies bereitet in der Regel bei einem String, der am Ende einer Zeile aufhört, auch keinerlei Probleme. Wenn der String jedoch mehrere Zeilen lang ist, wird eine Übereinstimmung nur bis zum ersten Neue-Zeile-Zeichen (\n) erzielt, denn das Punktzeichen berücksichtigt in der Regel die Zeichen für *Neue Zeile*.

Sie könnten das Problem umgehen, indem Sie das Muster ändern, so dass es nach einem oder mehreren Wörtern oder Whitespace-Zeichen sucht, und den Punkt auf diese Weise ganz vermeiden. Das wäre jedoch sehr arbeitsaufwendig. Was Sie hier benötigen, ist die Option /s am Ende Ihres Musters, die Perl anweist, das Metazeichen Punkt um das Neue-Zeile-Zeichen (\n) zu ergänzen. Die Option /s hat keinen Einfluß auf das Verhalten des restlichen Mustervergleichs - ^ und $ repräsentieren weiterhin die Zeichen für Anfang und Ende des Strings.

Wenn Sie einen regulären Ausdruck mit den Zeichen ^ und $ verwenden, werden Sie mehrzeilige Strings unter Umständen anders behandeln wollen als einzeilige Strings. Standardmäßig beziehen sich ^ und $ auf den Anfang und das Ende des Strings, wobei die Zeichen für eine neue Zeile ignoriert werden. Wenn Sie jedoch die Option /m verwenden, bezieht sich ^ nicht nur auf den Anfang der des Strings, sondern auch auf den Anfang einer Zeile (die Position direkt nach einem \n), und $ betrifft sowohl das Ende des Strings als auch das Ende der Zeile (die Position direkt vor dem \n). Mit anderen Worten, wenn Ihr String vier Textzeilen enthält, wird das Zeichen ^ viermal zu einer Übereinstimmung führen. Das gleiche gilt für $. Dazu ein Beispiel:

```
while (/^(\w)/mg) {
   print "$1\n";
}
```

Diese while-Schleife gibt das erste Wort jeder Zeile in $_ aus, unabhängig davon, ob die Eingabe eine Zeile oder mehrere enthält.

Wenn Sie jedoch die Option /m verwenden und ausnahmsweise den Anfang oder das Ende des Strings abfragen wollen, lassen sich ^ und $ dafür nicht länger verwenden. Aber keine Angst, auch dafür hat Perl eine Lösung parat: Mit \A und \Z beziehen Sie sich auf den Anfang und das Ende des Strings, unabhängig vom Status von /m.

Sie können problemlos die Optionen /s und /m zusammen verwenden. Sie müssen sich lediglich merken, dass /s das Verhalten des Punktes beeinflußt und /m das Verhalten von ^ und $. Darüber hinaus stellen eingebettete Neue-Zeile-Zeichen in Strings kein Problem für Pattern Matching dar.

Eine Zusammenfassung der Optionen und Escape- Zeichen

Im Verlaufe dieses Kapitels habe ich Ihnen diverse Optionen vorgestellt, die Sie zusammen mit den Mustern verwenden können, sowie eine Reihe von besonderen Escape-Zeichen, die innerhalb von Mustern verwendet

werden können.

In Tabelle 10.1 finden Sie die Optionen, die Sie an das Ende eines Musters (`m//` oder nur `//`) setzen können, aber auch jene, die zusammen mit dem Ersetzungsausdruck gültig sind (`s///`).

*Ich habe in diesem Kapitel nicht alle der hier aufgeführten Optionen besprochen. Einige werden Ihnen noch im Abschnitt »Vertiefung« vorgestellt. Falls Sie selbst nachforschen wollen, welche Möglichkeiten Ihnen bestimmte Optionen im Detail bieten, sollten Sie dazu in die **perlre**-Manpage schauen.*

Option	Verwendung
`g`	sucht alle Vorkommen (nicht nur eines)
`i`	sucht nach Groß- und Kleinbuchstaben
`m`	verwendet `^` und $ für *Neue Zeile*
`o`	interpoliert das Muster einmal (steigert die Effizienz)
`s`	(.) Punkt schließt die Neue-Zeile-Zeichen mit ein
`x`	erweitert die regulären Ausdrücke (kann Kommentare und Whitespace-Zeichen mit einschließen)
`e`	bewertet Ersetzung als Perl-Ausdruck (nur Substitution mit s//)

Tabelle 10.1: *Optionen zum Pattern Matching und Suchen&Ersetzen*

Tabelle 10.2 enthält die besonderen Escape-Zeichen, die in den regulären Ausdrücken zusätzlich zu den normalen String-Escape-Zeichen (`\t`, `\n`, Backslash, der Metazeichen als gewöhnliches Zeichen interpretiert) verwendet werden können.

Escape-Zeichen	Verwendung
`\A`	Anfang eines Strings
`\Z`	Ende eines Strings
`\w`	Wortzeichen
`\W`	Nichtwortzeichen
`\b`	Wortgrenze
`\B`	Nichtwortgrenze
`\s`	Whitespace-Zeichen
`\S`	Nicht-Whitespace-Zeichen
`\d`	Ziffer
`\D`	Nichtziffer
`\Q`	Alle Sonderzeichen mit einem Escape-Zeichen versehen
`\E`	Beendet eine `\Q`-Sequenz

Tabelle 10.2: Escape-Zeichen für Pattern Matching

Ein Beispiel: Der Grafik-Extraktor

Beenden möchte ich meine bisherigen Ausführungen mit einem Beispiel für einen ziemlich umfangreichen regulären Ausdruck (eigentlich sind es zwei Ausdrücke) innerhalb eines Perl-Skripts. Dieses Skript erhält als Eingabe eine HTML-Datei, geht die Datei durch und sucht nach eingebetteten Grafiken (mit Hilfe des <IMG>-Tags in HTML). Danach gibt es eine Liste der Grafiken in der Seite aus sowie eine Liste der verschiedenen Attribute zu der jeweiligen Grafik (ihre Position, Breite oder Höhe, Textalternative etc.). Die Ausgabe des Skripts wird in etwa so aussehen:

```
---------------
Grafik:  title.gif
   HSPACE: 4
   VPSACE: 4
   ALT: *
---------------
Grafik: smbullet.gif
   ALT: *
---------------
Grafik:  rib_bar_wh.gif
   BORDER: 0
   HSPACE: 4
   WIDTH; 50
   HEIGHT: 50
   ALT: --
```

Falls Sie mit HTML nicht vertraut sind, sehen Sie hier ein Beispiel für das <IMG>-Tag, das an einer beliebigen Stelle in einer HTML-Datei eingefügt werden kann:

```
<IMG SRC="imgfile.gif" WIDTH=50 HEIGHT=75 ALT="Pinguine">
```

Dieses Tag weist einige knifflige Besonderheiten auf, die Ihre Aufgabe schwieriger machen, als sie auf den ersten Blick scheint. Zum einen kann das Tag selbst in Groß- oder in Kleinbuchstaben vorkommen, und es kann sich über mehr als eine Zeile erstrecken. Die Attribute (die Schlüssel/Werte-Paare nach dem IMG-Teil) können ebenfalls groß oder klein geschrieben sein, können Leerzeichen links und rechts des Gleichheitszeichen haben und in Anführungszeichen stehen oder nicht. Die Werte können Leerzeichen enthalten (müssen dann allerdings von Anführungszeichen umschlossen sein). Es ist nur ein Attribut erforderlich, das SRC-Attribut, aber es gibt weitere Attribute, die in beliebiger Reihenfolge erscheinen können.

All diese Möglichkeiten sind dafür verantwortlich, dass der reguläre Ausdruck wesentlich komplexer wird, als wenn damit nur der Inhalt zwischen dem öffnenden und dem schließenden Tag aufgenommen werden sollte. Um genau zu sein, habe ich in diesem Skript die Aufgabe auf zwei reguläre Ausdrücke verteilt: Einer sucht das IMG- Tag in der Datei und zieht es heraus, und ein anderer extrahiert und parst die einzelnen Attribute.

Listing 10.1 enthält den Code zu diesem Skript. Versuchen Sie schon einmal, es durchzugehen, um ein Gefühl für den Aufbau zu bekommen. Aber sorgen Sie sich nicht zu sehr, wenn Sie die Muster noch nicht vollständig verstehen.

Listing 10.1: Das Skript img.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $/ = "";     # Absatzeingabe-Modus
4:  $raw = "";   # rohe Attribute
5:  %atts = ();  # Attribute
6:
7:  while (<>) {
8:      while (/<IMG\s+([^>]+)>/ig ) {
9:          $raw = $1;
10:         while ($raw =~ /([^ =]+)\s*=\s*("([^"]+)"|[^\s]+\s*)/ig) {
11:            if (defined $3) {
12:               $atts{ uc($1) } = $3;
13:            } else { $atts{ uc($1)} = $2; }
14:         }
15:         if ($raw =~ /ISMAP/i) {
16:             $atts{'ISMAP'}= "Yes";
17:         }
18:
19:         print '-' x 15;
```

```
20:         print "\nGrafik:  $atts{'SRC'}\n";
21:         foreach $key ("WIDTH", "HEIGHT",
22:                       "BORDER", "VSPACE", "HSPACE",
23:                       "ALIGN", "ALT", "LOWSRC", "ISMAP") {
24:              if (exists($atts{$key})) {
25:                 $atts{$key} =~ s/[\s]*\n/ /g;
26:                 print "   $key: $atts{$key}\n";
27:              }
28:         }
29:         %atts = ();
30:     }
31: }
```

Dieses Skript besteht aus zwei Teilen: einem Abschnitt, der die Daten aus der Eingabe herauszieht, und einem Abschnitt, der in Protokollform ausgibt, was gefunden wurde.

Der erste Teil verwendet eine Reihe von verschachtelten `while`-Schleifen, um die HTML-Datei durchzugehen: Zeile 7 durchläuft die gesamte Eingabe, Zeile 8 sucht in der Eingabe nach jedem Vorkommen des Image-Tags, und Zeile 10 durchläuft jedes Attribut im `<IMG>`-Tag und speichert es in einem Hash namens `%atts` mit dem Attributnamen als Schlüssel.

Nachdem der Hash `%atts` gefüllt ist, müssen wir nur noch die Werte ausgeben. Da ich möchte, dass sie in einer speziellen Reihenfolge ausgegeben werden, habe ich in Zeile 21 mit einer `foreach`-Schleife angegeben, in welcher Reihenfolge die Schlüssel auszugeben sind.

Unser Hauptaugenmerk in diesem Skript soll aber auf den regulären Ausdrücken in den Zeilen 8 und 10 liegen. Betrachten wir also diese zwei Muster im Detail. Zeile 8 lautet:

```
while (/<IMG\s+([^>]+)>/ig) {
```

Gehen wir diesen regulären Ausdruck zeichenweise durch: Zuerst suchen wir nach den Zeichen `<IMG` (der Öffnungsteil des Tags) und dann nach einem oder mehreren Whitespace-Zeichen, gefolgt von einem oder mehreren Zeichen, die nicht > sind. Das Muster selbst wird mit dem eigentlichen >-Zeichen, das das Ende des Tags signalisiert, beendet.

Beachten Sie die Klammern um den Teil, der die »ein oder mehrere Zeichen, die nicht > sind« repräsentiert - genau dieser Teil interessiert uns besonders, da er die Attribute für die Grafik enthält. Dieser Teil des Musters wird herausgezogen und gespeichert, so dass er später in dem Rumpf der Schleife verwendet werden kann.

Eine Bemerkung wert sind auch die Optionen am Ende des Musters: `/i` steht für eine Suche, die Groß- und Kleinschreibung unberücksichtigt läßt (gesucht wird sowohl `<img...>` als auch `<IMG...>`, und `/g` steht für eine globale Suche (wir durchlaufen für jedes `<IMG`, auf das wir stoßen, einmal die `while`-Schleife innerhalb der Schleife). Beachten Sie, dass wir `/s` oder `/m` eigentlich nicht explizit angeben müssen, um die Suche über Zeilengrenzen hinweg auszuführen - HTML erfordert Zeilenumbrüche nur für Whitespace, und unser Muster verwendet `/s` für alle Whitespace-Zeichen, so dass wir in dieser Hinsicht auf der sicheren Seite sind. Wir werden kein Problem mit Tags haben, die sich über mehrere Zeilen erstrecken.

Im Rumpf der Schleife können wir die Attribute-Liste in der Variablen `$raw` (Zeile 9) speichern. Wir sind dazu gezwungen, da die Werte von `$1`, `$2`, `$3` und so weiter alle nur kurzlebig sind - sie werden beim nächsten Mustervergleich zurückgesetzt. Damit kommen wir zu Zeile 10 und dem folgenden wirklich fast undurchschaubaren regulären Ausdruck:

```
while ($raw =~ /([^ =]+)\s*=\s*("([^"]+)"|[^\s]+\s*)/ig) {
```

Dieser reguläre Ausdruck läßt sich in vier wichtige Teile gliedern, die ich in Abbildung 10.1 veranschaulichen möchte:

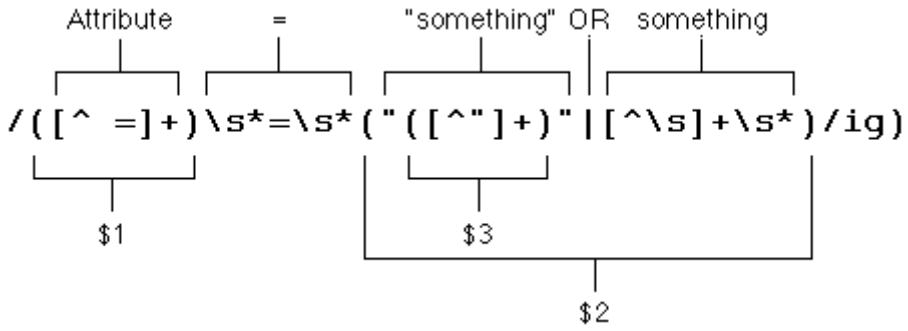

Abbildung 10.1: Die Teile des regulären Ausdrucks in img.pl

Die Teile lassen sich auch wie folgt beschreiben:

- Der Attributname - ein Satz an Zeichen, der weder ein Leerzeichen noch ein Gleichheitszeichen ist. Wir speichern den Attributnamen in `$1`.
- Ein Gleichheitszeichen, das links und rechts optional von einem Whitespace- Zeichen begrenzt sein kann.
- Eines von zwei Werte-Formaten: Das erste ist ein Anführungszeichen gefolgt von einer Reihe von Zeichen und einem weiteren Anführungszeichen. Damit sind die Attribute in Anführungszeichen abgedeckt (zum Beispiel `ALT="ein alternativer Text"`).
- Eine Reihe von Nicht-Whitespace-Zeichen gefolgt von optionalen Whitespace- Zeichen. Damit sind die Fälle abgedeckt, in denen keine Anführungszeichen verwendet werden (zum Beispiel `HEIGHT=100`).

Beachten Sie die Position der Klammern für die Werte, und vergegenwärtigen Sie sich die Regel für Übereinstimmungsvariablen: Die Zahlen werden auf der Basis der öffnenden Klammer vergeben. Deshalb repräsentiert `$2` den kompletten Wert, während `$3` der Wert minus der Angabe in Anführungszeichen ist - falls der Wert überhaupt Anführungszeichen hatte.

In den Zeilen 11 bis 14 werden beide Fälle berücksichtigt, indem wir testen, ob `$3` gesetzt ist. Wenn ja, hat unser Wert Anführungszeichen, und der Teil, der nicht in den Anführungszeichen steht, wird im Hash `%atts` gespeichert. Ist `$3` nicht definiert, steht unser Wert nicht in Anführungszeichen, und wir können statt dessen `$2` in `%atts` ablegen. Außerdem möchte ich Sie darauf hinweisen, dass wir die Funktion `uc` verwendet haben, um unsere Attributnamen in Großbuchstaben zu wandeln, bevor sie gespeichert werden.

Die Zeilen 15 bis 17 behandeln einen Sonderfall des `<IMG>`-Tags: Das Attribut `ISMAP` übernimmt keinen Wert. Es zeigt lediglich an, ob es sich bei der Grafik um eine Imagemap handelt. (Eine Imagemap ist eine anklickbare Grafik, das heißt, Sie können verschiedene Bereiche der Grafik anklicken und lösen damit unterschiedliche Aktionen aus.) Diese Art von anklickbarer Grafik wird in HTML normalerweise nicht mehr verwendet (das Tag wurde durch ein anderes Tag ersetzt), aber der Vollständigkeit halber habe ich es mit aufgenommen. Wir mußten dies zu einem Sonderfall machen, da es nicht durch den Ausdruck in Zeile 10 abgefangen wird.

Nach Ausführung all dieser Schleifen und Muster haben wir einen Hash vorliegen, in dem alle gefundenen Attribute des `<IMG>`-Tags gespeichert sind. Uns bleibt nur noch, diese Werte auszugeben. Die Schleife in Zeile 21 durchläuft alle möglichen Werte für das `<IMG>`-Tag in der Reihenfolge, in der sie ausgegeben werden. Jedes `<IMG>`, das wir in der Datei finden, kann potentiell eine Teilmenge dieser Attribute aufweisen, die bis auf `SRC` alle optional sind. Deshalb müssen wir in Zeile 24 einen Test durchführen, um sicherzustellen, dass das Attribut auch tatsächlich existiert, bevor es ausgegeben wird. Die Funktion `exists` testet einen Hash, um festzustellen, ob ein Schlüssel existiert, und liefert ***wahr*** oder ***falsch*** zurück.

Ungewöhnlich ist auch die Zeile 25, in der wir eine schnelle Suchen&Ersetzen- Operation auf dem Wert durchführen. Damit wollen wir die Fälle abfangen, in denen sich das Attribut auf mehrere Zeilen verteilt - der Wert also ein Neue-Zeile-Zeichen enthält, das in der endgültigen Tabelle nicht mit ausgegeben werden soll. Dieser kleine reguläre Ausdruck sucht optionale Whitespace-Zeichen gefolgt von einem Neue-Zeile- Zeichen und ersetzt sie durch ein einfaches Leerzeichen.

Zum Schluß in Zeile 29 löschen wir den Attribut-Hash für die nächste Runde und das nächste `<IMG>`-Tag.

Dieses Skript ist zwar kurz, zeigt aber vorbildlich, für welche Art Aufgaben sich Perl besonders gut eignet:

komplizierte Muster in Texten zu suchen und diese dann in anspruchsvollen Protokollen auszugeben. Wenn Sie die gleiche Aufgabe in C lösen wollten, würden Sie mit Sicherheit mehr als 30 Zeilen dafür benötigen.

Tips zum Erstellen regulärer Ausdrücke

Je nach der Komplexität Ihrer Daten oder der Aufgabe, die damit zu bewerkstelligen ist, kann die Formulierung eines regulären Ausdrucks sehr einfach sein oder mehrerer Überarbeitungen bedürfen. Hier möchte ich Ihnen einige Tips geben, die Ihnen helfen sollen, Suchmuster zu verwenden:

- Machen Sie sich mit den Daten vertraut. Versuchen Sie ein Gefühl für die verschiedenen Erscheinungsformen der Muster zu bekommen und zuerst einmal einen Satz konsistenter Regeln aufzustellen, bevor Sie dazu übergehen, eigene Muster zu schreiben.
- Verwenden Sie mehrere reguläre Ausdrücke, wo nötig. Manchmal ist es einfacher, die Aufgabe in mehreren Teilaufgaben zu lösen als zu versuchen, alles auf einmal zu bewältigen.
- Vergessen Sie `split` nicht. Einige Aufgaben lassen sich besser mit `split` lösen (alles außer einem Muster entfernen) als mit einfachen Mustervergleichen. Und umgekehrt.
- Verwenden Sie Klammern, um das (und nur das), was Sie benötigen, herauszuziehen. Wenn Sie zum Beispiel die Anführungszeichen in einem Vergleich nicht benötigen, setzen Sie die Klammern innerhalb der Anführungszeichen. Speichern Sie nur, was Sie unbedingt brauchen, und Sie ersparen es sich, die nicht benötigten Teile später löschen zu müssen.
- Verwenden Sie negierte Zeichenklassen anstelle von Quantifizierern. Denken Sie daran, dass Quantifizierer wie `.*` gierig sind, das heißt alle Zeichen bis zum Ende der Zeile schlucken und Ihnen das Leben schwer machen können. Ganz abgesehen davon, dass sie auch für Perl selbst einen zusätzlichen Arbeitsaufwand bedeuten. Vermeiden Sie diese Konstrukte überall dort, wo sich eine negierte Zeichenklasse besser einsetzen läßt.
- Denken Sie daran, dass `*` und `?` für »kein oder mehrere« beziehungsweise »kein oder ein« Zeichen stehen. Dies bedeutet, dass das Suchmuster auch dann erfolgreich sein kann, wenn das Zeichen gar nicht vorkommt. Wenn Ihr Muster erfordert, dass ein Zeichen mindestens einmal vorhanden ist, verwenden Sie `+` anstelle von `*` oder `?`.
- Behalten Sie auch das Prinzip der Alternation im Auge. Das Alternationszeichen (`|`) kann für hochkomplexe Muster, die mehrere komplexe alternative Fälle aufweisen, sehr praktisch sein.
- Ziehen Sie auch in Erwägung, Ihre Aufgabe ohne einen regulären Ausdruck zu lösen. Reguläre Ausdrücke sind unglaublich leistungsstark, was allerdings auf Kosten der Performance geht. Das sollte man vor allem bei leichten Aufgaben wie zum Beispiel einfachen Tests berücksichtigen.
- Verlieren Sie nicht den Kopf. Reguläre Ausdrücke können unglaublich leistungsstark sein, sie können einen aber auch an den Rand des Wahnsinns treiben, wenn mal wieder nichts klappt. Wenn Sie sich bei einem regulären Ausdruck verrannt haben, legen Sie eine Pause ein, und versuchen Sie, das Problem von einem anderen Gesichtspunkt aus zu betrachten.

Vertiefung

Reguläre Ausdrücke gehören zu den Themen, mit denen man ein ganzes Buch füllen und trotzdem noch Fragen offenlassen kann. In den Kapiteln von gestern und heute habe ich Ihnen die Grundlagen vermittelt, wie man reguläre Ausdrücke aufbaut und in eigenen Programmen verwendet. Es gibt jedoch noch viele Punkte, die ich nicht angesprochen habe, unter anderem eine Unmenge von Metazeichen und Perl- spezifischen regulären Ausdrücken. In diesem Abschnitt möchte ich Ihnen einen Überblick über einige davon geben.

Weitere Informationen zu anderen Aspekten der regulären Ausdrücke in Perl finden Sie in der ***perlre***-Manpage, die ziemlich aufschlußreich ist. Wenn Sie feststellen, dass Ihnen die Arbeit mit regulären Ausdrücken großen Spaß macht und Sie des Englischen mächtig sind, sollten Sie die Anschaffung des Buches »***Reguläre Ausdrücke***« (Jeffrey Friedl, O'Reilly Verlag) in Erwägung ziehen. Dieses Buch beschreibt erstaunlich detailliert reguläre Ausdrücke aller Art - sowohl von Perl als auch von anderen Sprachen.

Weitere Metazeichen

Mit den Metazeichen, die ich gestern und heute beschrieben habe, habe ich Ihnen den größten Teil der elementaren Zeichen vorgestellt, die in den meisten regulären Ausdrücken (nicht nur bei Perl) Verwendung finden. Perl umfaßt eine Reihe von zusätzlichen Metazeichen, die noch andere Möglichkeiten bieten, komplexe Muster zu erstellen (oder die gefundenen Muster zu verarbeiten).

Als erstes möchte ich hier die nichtgierigen Versionen der Quantifizierer *, +, ? und {} nennen. Wie Sie in diesem Kapitel gelernt haben, gehören die Quantifizierer eigentlich zu den gierigen Metazeichen. Sie vergleichen alle Zeichen und damit weit mehr, als von Ihnen erwartet - was Ihnen manchmal zum Nachteil gereichen kann, wenn Sie herausfinden wollen, wonach das Muster eigentlich sucht. Zu diesen Quantifizierern bietet Perl Ihnen nun als Gegenstücke die nichtgierigen Quantifizierer (manchmal auch faule Quantifizierer genannt): *?, +?, ?? und {}?. Diese Quantifizierer vergleichen die Mindestanzahl der Zeichen, die für den Mustervergleich nötig sind, während die regulären Quantifizierer die maximale Zahl der Zeichen vergleicht. Dies kann in manchen Situationen nützlich sein; Sie sollten jedoch nicht vergessen, wenn möglich auch mit negierten Zeichenklassen zu arbeiten. Die faulen Quantifizierer sind nicht so effizient wie eine negierte Zeichenklasse und führen oft zu unerwarteten Ergebnissen.

Das Konstrukt (?:muster) ist eine Variante zu den Klammern, mit denen Muster zusammengefaßt und die Ergebnisse in den Übereinstimmungsvariablen $1, $2, $3 etc. gespeichert werden. Wenn Sie Klammern verwenden, um einen Ausdruck als Gruppe zusammenzufassen, wird das Ergebnis automatisch auch gesichert, ob Sie wollen oder nicht. Mit dem Konstrukt (?:muster) hingegen wird der Ausdruck als Einheit zusammengefaßt und ausgewertet, das Ergebnis jedoch nicht gespeichert. Dies ist effizienter als die Verwendung der normalen Klammern, wenn das Ergebnis keine Rolle spielt.

Mit dem Konstrukt (?o) können Sie die Optionen für das Pattern Matching innerhalb des Musters selbst verwenden. So können Sie damit zum Beispiel bestimmte Teile des Ausdrucks von der Unterscheidung der Groß- und Kleinschreibung befreien. Das o in dem Konstrukt kann eine beliebige gültige Option für den Mustervergleich sein.

Die Voraussschau ist eine Besonderheit der regulären Ausdrücke in Perl, die es Perl erlaubt, den String kurz zu überfliegen und zu schauen, ob es nicht irgendwo eine Übereinstimmung mit dem Muster gibt, ohne dass dabei die Position im String geändert oder irgend etwas den geklammerten Teilen des Musters hinzugefügt wird. Das entspricht in etwa der Aussage: »Wenn der nächste Teil dieses Musters X enthält, dann stimmt dieser Teil überein«, ohne dass dabei die Position verlassen wird. Mit (?=muster) erzeugen Sie ein positives Muster für die Vorausschau (wenn muster im weiteren Verlauf des Strings eine Übereinstimmung hat, dann hat der vorangegangene Teil des Musters auch eine Übereinstimmung). Das Gegenteil davon ist ein negatives Muster (?!muster). Es stimmt nur dann überein, wenn zu dem muster keine Übereinstimmung gefunden wird.

Besondere Variablen

Zusätzlich zu den Übereinstimmungsvariablen $1, $2 und so weiter gibt es in Perl noch die Variablen $', $& und $´, die einen Kontext für die Textfundstellen des Musters bereithalten. $' bezieht sich dabei auf den Text bis zur Fundstelle, $& steht für den gefundenen Text und $´ für den Text nach der Fundstelle (beachten Sie den Apostroph, der nicht mit dem einfachen Anführungszeichen ' identisch ist). Im Gegensatz zu den kurzlebigen Übereinstimmungsvariablen halten diese Variablen ihre Werte bis zum nächsten erfolgreichen Vergleich fest - unabhängig davon, ob der ursprüngliche String sich geändert hat oder nicht. All diese Variablen zehren jedoch enorm an der Leistung, deshalb sollten Sie sie wenn möglich vermeiden.

Die Variable $+ gibt die höchste Zahl der definierten Übereinstimmungsvariablen an. Wurden zum Beispiel $1 und $2, jedoch nicht $3, mit Werten gefüllt, wird $+ auf 2 gesetzt.

Optionen

Im Verlaufe dieses Kapitels haben Sie den größten Teil der Optionen für reguläre Ausdrücke in Perl kennengelernt (sowohl m// als auch s///). Zwei sind jedoch noch unerwähnt geblieben: /x für erweiterte reguläre Ausdrücke und /o, um wiederholtes Kompilieren des gleichen regulären Ausdrucks zu vermeiden.

Mit der Option /x können Sie reguläre Ausdrücke zur besseren Lesbarkeit mit Whitespace-Zeichen und Kommentaren versehen. Wenn Sie normalerweise Leerzeichen in einem Muster verwenden, werden diese Leerzeichen als Teil des Musters selbst betrachtet. Mit der Option /x werden nicht nur alle Leerzeichen und Neue-Zeile-Zeichen ignoriert, es ist damit auch möglich, Kommentare zu den einzelnen Zeilen des regulären Ausdrucks einzubauen. So könnten wir zum Beispiel den regulären Ausdruck in unserem Skript ***img.pl***, der folgendermaßen aussah:

```
while ($raw =~ /([^ =]+)\s*=\s*("([^"]+)"|[^\s]+\s*)/ig) {
```

auch wie folgt schreiben:

```
while ($raw =~ /([^ =]+)     # sucht den Attributnamen
                \s*=\s*      # sucht Gleichheitszeichen mit und ohne
                             # Whitespaces
                ("([^"]+)"|  # sucht und extrahiert Werte in
                             # Anführungszeichen
                [^\s]+\s*)   # oder sucht Werte nicht in Anführungszeichen
                /igx) {
```

Die Verwendung von erweiterten regulären Ausdrücken kann eine große Hilfe sein und die Lesbarkeit eines regulären Ausdrucks verbessern.

Schließlich gibt es noch die Option `/o`, die dazu dient, Perl beim Kompilieren und Einlesen eines regulären Ausdrucks, der über eine Skalarvariable interpoliert wurde, zu optimieren. Betrachten wir folgendes Codefragment:

```
while (<>) {
   if (/$muster/) {
     ...
   }
}
```

In diesem Fragment wird das in `$muster` gespeicherte Muster interpoliert und in ein richtiges Muster kompiliert, das Perl versteht. Das Problem dabei ist jedoch, dass dieses Muster sich innerhalb einer Schleife befindet und sich deshalb der Prozeß bei jedem Durchlauf der Schleife wiederholt. Mit der Angabe von `/o` am Ende des Musters teilen Sie Perl mit, dass sich das Muster nicht ändert und deshalb nur einmal kompiliert werden muss, um dann einfach wiederverwendet zu werden:

```
if (/$muster/o) {  # nur einmal kompilieren
```

Informationen zu diesen Metazeichen, Variablen und Optionen finden Sie auch in der Hilfsdokumentation ***perlre-*** Manpage.

Zusammenfassung

Im heutigen Kapitel haben wir, aufbauend auf den Grundlagen von gestern, das Thema der regulären Ausdrücke vertieft. Wir haben angesprochen, wie man Fundstellen aus einer Pattern-Matching-Operation mit Hilfe von Klammern herauszieht und wie man durch die Verwendung von Rückbezügen und Übereinstimmungsvariablen gefundene Vorkommen speichert, um später auf sie zuzugreifen.

Im Rahmen dieser Diskussion haben Sie Pattern Matching in verschiedenen Kontexten kennengelernt (skalare Kontexte liefern ***wahr*** oder ***falsch*** zurück, Listenkontexte liefern Listen der Übereinstimmungen zurück). Außerdem habe ich Ihnen etwas über das gierige Verhalten der quantifizierenden Metazeichen und die erweiterten Möglichkeiten der `split`-Funktion erzählt. Wenn Sie die beiden Kapitel zu den regulären Ausdrücken durchgearbeitet haben, sollten Sie genug über reguläre Ausdrücke wissen, um so ziemlich jedes Muster mit einem beliebigen Satz an Daten zu vergleichen.

Fragen und Antworten

Frage:
Ich habe hier einige Codezeilen, die in zwei Schritten versuchen, etwas aus einem String herauszuziehen. Das erste Muster legt einen Teilstring in `$1` ab, und dann durchsucht das zweite Muster `$1` nach einem weiteren Muster. Das zweite Muster führt nie zu einer Übereinstimmung, und die Ausgabe von `$1` zeigt, dass die Variable leer ist. Wenn diese Variable aber leer ist, so hätte das zweite Muster eigentlich gar nicht erst verglichen werden sollen. Was läuft hier falsch?

Antwort:
Das klingt, als wenn Sie ungefähr folgendes versucht hätten:

```
    if ($string =~ /ein langes Muster mit einem {Teilmuster} darin/) {
   if ($1 =~ /ein zweites Muster/) {
      # verarbeitet zweites Muster
   }
```

```
}
```

Leider geht das so nicht. Die Variable `$1` (oder jede andere Variable) ist unglaublich kurzlebig. Jedesmal, wenn Sie einen regulären Ausdruck verwenden, setzt Perl alle Übereinstimmungsvariablen zurück. Bei diesem speziellen Beispiel hier gibt es in der ersten Zeile eine Übereinstimmung, und die Variable `$1` wird mit dem Inhalt der Klammern gefüllt. Sobald Sie jedoch einen neuen Mustervergleich durchführen (in der zweiten Zeile), verschwindet der Wert von `$1`, was konkret bedeutet, dass das zweite Muster nie zu einer Übereinstimmung führen kann. Das Geheimnis liegt darin, sicherzustellen, dass die Werte aller Übereinstimmungsvariablen irgendwo anders abgelegt werden, wenn Sie wiederverwendet werden sollen. In diesem besonderen Fall reicht es aus, eine temporäre Variable hinzuzufügen und diese dann zu durchsuchen:

```
 if ($string =~ /ein langes Muster mit einem {Teilmuster} darin/) {
$tmp = $1;
if ($tmp =~ /ein zweites Muster/) {
   # verarbeitet zweites Muster
}
}
```

Frage:
Ich habe einige Skripts gesehen, die eine $*-Variable auf 1 gesetzt haben, um einen Mustervergleich über mehrere Zeilen durchzuführen - wozu sie die Option `/ m` verwenden. Was bedeutet $*, und kann ich es auch verwenden?

Antwort:
In früheren Versionen von Perl hat man `$*` gesetzt, um Perl anzuweisen, die Bedeutung von `^` und `$` zu ändern. In aktuellen Perl-Versionen jedoch sollten Sie statt dessen die Option `/m` verwenden. `$*` wurde lediglich aus Gründen der Rückwärtskompatibilität beibehalten.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Angenommen `$_` enthält `123 kazoo kazoo 456`. Wie lauten die Ergebnisse der folgenden Ausdrücke?

```
    @matches = /(\b[^\d]+)\b/g;
@matches = /\b[^\d]+\b/;
s/\d{3}/xxx/;
s/\d{3}/xxx/g;
$matches = s/\d{3}/xxx/g;
if (/\d+(.*)\d+/) { print $1;}
@matches = split(/z/);
@matches = split(" ",$_ 3);
```

2. Wie lautet die Regel für die Numerierung der Rückbezüge und der Übereinstimmungsvariablen?
3. Wie lange bestehen die Werte der Übereinstimmungsvariablen?
4. Wie können Sie das gierige Verhalten der Quantifizierer + und * unterbinden?
5. Was bewirken die folgenden Optionen?

```
/g

/i

/o

/s
```

Übungen

1. FEHLERSUCHE: Was ist an diesem Codefragment falsch?

```
    while (<>) {
   $input =~ /pat\s/path /;
}
```

2. FEHLERSUCHE: Was ist an diesem Codefragment falsch?

```
    @matches = /\b[^\d]+\b/;
```

3. Schreiben Sie ein Skript, das doppelte Wörter (»das das« oder »ein ein«) in der Eingabe findet und diese durch ein Vorkommen desselben Wortes ersetzt. Suchen Sie dabei auch nach doppelten Wörtern über zwei Zeilen.
4. Schreiben Sie ein Skript, das Akronyme in der Eingabe aufschlüsselt (ersetzen Sie zum Beispiel »HTML« durch »HTML (Hypertext Markup Language)«. Verwenden Sie die folgenden Akronyme und deren Ersetzungen:

```
    HTML (HyperText Markup Language)
ICBM (InterContinental Ballistic Missile)
EEPROM (Electrically-erasable programmable read-only memory)
SCUBA (self-contained underwater breathing apparatus)
FAQ (Frequently Asked Questions)
```

5. Modifizieren Sie das Skript ***img.pl***, so dass es Links anstelle von Grafiken herauszieht und protokolliert. TIP: Links haben ungefähr folgendes Format:

```
    <A HREF="url_des_Links">Text des Links</A>
```

1. Links können folgende Attribute enthalten: `NAME`, `REL`, `REV`, `TARGET` und `TITLE`.
1. Stellen Sie sicher, dass Sie sowohl den Inhalt des Link-Tags protokollieren als auch den Text zwischen den öffnenden und schließenden Tags.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Die Antworten lauten

    a. (`"kazoo"`, `"kazoo"`)

    b. (`1`). Ein Muster in einem Listenkontext ohne Teile des Musters in Klammern oder mit der Option `/g` liefert im Falle einer Übereinstimmung (`1`) zurück

    c. Der String wird geändert in »`xxx kazoo kazoo 456`«

    d. Der String wird geändert in »`xxx kazoo kazoo xxx`«

    e. Die Variable `$matches` wird auf 2 gesetzt (die Anzahl der vorgenommenen Änderungen)

    f. »`kazoo kazoo 45`« (behalten Sie die gierigen Quantifizierer im Auge)

    g. (»`123 ka`«, »`oo ka`« »`oo 456`«)

    h. (»`123`«, »`kazoo`«, »`kazoo 456`«)

2. Die Numerierung der Rückbezüge und der Übereinstimmungsvariablen basieren auf den öffnenden Klammern. Die Klammern eines Musters können ineinander verschachtelt werden.
3. Übereinstimmungsvariablen sind extrem kurzlebig. Ihre Werte sind nur bis zum nächsten Mustervergleich oder bis zum Ende des Blocks gültig.
4. Zwei Möglichkeiten: Am besten vermeiden Sie gierige Quantifizierer durch die Verwendung des Punktzeichens (`.`) oder durch Einsatz negierter Zeichenklassen. Der zweite Weg führt über die nichtgierigen Versionen der Quantifizierer (`+?` und `*?`).
5. Die Antworten lauten:

   a. Die Option `/g` bedeutet »global«. Das Muster wird im ganzen String gesucht (die Suche wird nicht nach dem ersten Vorkommen abgebrochen). Je nach Kontext ist der weitere Verlauf unterschiedlich.

   b. Die Option `/i` unterdrückt die Unterscheidung nach Groß- und Kleinschreibung bei der Suche.

   c. Die Option `/o` bedeutet: »Kompiliere dieses Muster nur einmal«. Sie ist besonders nützlich zum Optimieren von Mustern mit eingebetteten Variablen.

   d. Die Option `/s` ermöglicht es dem Punktzeichen (`.`), das Neue-Zeile-Zeichen (`\n`) zu übergehen.

Lösungen zu den Übungen

1. Das Muster besteht aus zwei Teilen (ein Muster und eine Ersetzung) aber keinem führenden `s`. Die korrekte Version hierfür sieht folgendermaßen aus:

```
    while (<>) {
   $input =~ s/pat\s/path /;
}
```

2. Fangfrage! An diesem Code ist eigentlich nichts falsch, außer dass er wahrscheinlich nicht das macht, was Sie erwarten. Ein Muster in einem Listenkontext, der keine Teilmuster in Klammern aufweist, ergibt den Wert (`1`), wenn das Muster zu einer Übereinstimmung führt. Um eine Liste der Übereinstimmungen zu sichern, müssen Sie irgendwo im Muster Klammern setzen.
3. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
#
# Sucht doppelte Wörter ohne Rücksicht auf Zeilenumbrüche
# diese Version sucht Groß- und Kleinbuchstaben, aber berücksichtigt
# keine Zeichensetzung oder mehr als zwei Vorkommen des gleichen Wortes.
$/ = "";  # Absatzeingabe-Modus
while (<>) {
  s/\b(\w+)\s+\1\b/$1/ig;
  print;
}
```

4. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
%acs = (
       "HTML" => "HyperText Markup Language",
       "ICBM" => "InterContinental Ballistic Missile",
       "EEPROM" => "Electrically-erasable programmable read-only memory",
       "SCUBA" => "self-contained underwater breathing apparatus",
       "FAQ" => "Frequently Asked Questions",
       );
while (<>) {
    foreach $key (keys %acs) {
        s/$key/$key ($acs{$key})/gi;
    }
    print;
}
```

5. Hier ist eine Lösung:

```
    #!/usr/bin/perl -w
# sucht und extrahiert Links
# Lässt Link-Text mit eingebettetem HTML unberücksichtigt
$/ = "";     # Absatzeingabe-Modus
$raw = "";   # rohe Attribute
$linktext = ""; # Link-Text
%atts = ();  # Attribute
while (<>) {
    while (/<A\s+([^>]+)>([^<]+)<\/A>/ig) {
        $raw = $1;
        $linktext = $2;
        $linktext =~ s/[\s]*\n/ /g;
        while ($raw =~ /([^\s=]+)\s*=\s*("([^"]+)"|[^\s]+\s*)/ig) {
           if (defined $3) {
              $atts{ uc($1) } = $3;
           } else { $atts{ uc($1)} = $2; }
        }
        print '-' x 15;
        print "\nLink-Text: $linktext\n";
        foreach $key ("HREF", "NAME", "TITLE",
                      "REL", "REV", "TARGET") {
             if (exists($atts{$key})) {
                $atts{$key} =~ s/[\s]*\n/ /g;
                print "   $key: $atts{$key}\n";
             }
        }
        %atts = ();
    }
}
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Subroutinen erstellen und verwenden

Wir vervollständigen Ihr Wissen über den Perl-Sprachkern heute mit dem Thema Subroutinen, Funktionen und lokale Variablen. Wenn Sie bestimmten Code immer wieder verwenden, müssen Sie ihn nicht jedesmal in Ihr Skript tippen - Sie können ihn in eine Subroutine packen und diese, wann immer Sie sie brauchen, aufrufen - genau wie Sie es von Perl-Funktionen kennen.

Heute erkläre ich Ihnen unter anderem:

- den Unterschied zwischen benutzerdefinierten Subroutinen und Perls Standardfunktionen
- wie Sie einfache Subroutinen definieren und aufrufen
- wie Sie lokale Variablen für Subroutinen deklarieren und verwenden
- wie Subroutinen Werte zurückgeben
- wie Sie Argumente an Subroutinen übergeben
- wie Sie Subroutinen in verschiedenen Kontexten einsetzen

Subroutinen und Funktionen

Eine Funktion ist im allgemeinen ein Stück Code, das auf irgendwelchen Daten einen oder mehrere Vorgänge ausführt. Man ruft eine Funktion auf, indem man im Skript ihren Namen nennt und ihr Argumente übergibt. Wenn Perl das Skript ausführt und auf einen Funktionsaufruf stößt, wechselt Perl zur entsprechenden Funktionsdefinition, führt die dortigen Anweisungen aus und kehrt dann an die Stelle zurück, an der es das Skript verlassen hat. Das gilt für alle Arten von Funktionen.

Wir haben in den bisherigen Kapiteln bereits viel mit Perls eigenen Funktionen wie `print`, `sort`, `keys`, `chomp` etc. gearbeitet. Diese sind in Perls Standardbibliothek definiert, so dass sie in allen Perl-Programmen zur Verfügung stehen. Eine zweite Art von Funktionen sind in zusätzlichen Perl-Modulen oder -Bibliotheken definiert. Um eine solche Funktion aufrufen zu können, müssen Sie am Anfang Ihres Skripts das zugehörige Modul geladen haben - wie das geht, zeige ich Ihnen am Tag 13.

Eine dritte Art von Funktionen sind die Subroutinen, die Sie selbst definieren. Die Begriffe *Funktion* und *Subroutine* bedeuten in Perl im Grunde das gleiche.[1] Manche Programmierer nennen Subroutinen auch benutzerdefininierte Funktionen, um sie von Perls eigenen Funktionen zu unterscheiden, in anderen Zusammenhängen ist eine Unterscheidung gar nicht nötig. Ich nenne in diesem Buch die Funktionen, die Sie in Ihren eigenen Programmen (und später auch Modulen) selbst definieren, Subroutinen. Funktionen nenne ich die, die Sie woanders herbekommen - aus der Perl- Standardbibliothek oder optionalen Modulen.

Was nützt Ihnen eine Subroutine? Sobald Sie mehr als nur ein paar Zeilen Code in Ihren Skripts wiederholen müssen, haben Sie einen guten Grund, diesen Code in eine Subroutine zu packen. Das erspart Ihnen Tipparbeit. Doch auch wenn Sie den Code nur einmal brauchen, kann es - besonders bei komplexen Aufgaben - von Nutzen sein, das Skript aufzuteilen und die verschiedenen Teilaufgaben in eigenen Subroutinen zu lösen. Klare Namen für die einzelnen Vorgänge (`diagramm_ausgeben` oder `max_finden`) steigern die Übersichtlichkeit Ihres Skripts. Eventuelle Probleme lassen sich schneller isolieren: Sie können jede Subroutine für sich allein schreiben, testen und überprüfen - und sicher sein, dass sie, wenn Sie sie schließlich in Ihr Gesamtskript einfügen, auch funktioniert, wie Sie erwarten. Subroutinen sind eine Frage des Programmierstils - je mehr Einzelprobleme Sie auch in eigenen Subroutinen lösen, desto übersichtlicher und nachvollziehbarer werden Ihre Skripts für Sie und andere.

Einfache Subroutinen definieren und aufrufen

Die einfachste Form von Subroutine nimmt keine Argumente entgegen, verwendet keine lokalen Variablen, gibt keinen Wert zurück - und hat nicht besonders viel Sinn. Am Beispiel einer solchen Subroutine zeige ich Ihnen im folgenden Abschnitt, wie Sie Subroutinen deklarieren, definieren und aufrufen.

Eine Beispielsubroutine

Nehmen wir ein ganz einfaches Beispiel. Erinnern Sie sich an das Skript ***temperatur.pl*** von Tag 2, das nach einer Temperatur in Fahrenheit gefragt und sie in Celsius umgerechnet hat? Die eigentliche Berechnung stand mitten im Skript, aber wir hätten sie auch in eine Subroutine stellen können:

```
1#!/usr/bin/perl -w
$fahr = 0;
$cel = 0;
print 'Geben Sie eine Temperatur in Fahrenheit ein ';
chomp ($fahr = <STDIN>);
&f2c(); #Fahrenheit zu Celsius
print "$fahr Grad Fahrenheit entsprechen ";
printf("%d Grad Celsius\n", $cel);
sub f2c {
   $cel = ($fahr - 32) * 5 / 9;
}
```

Sehen Sie sich genau an, wie wir die Subroutine hier einsetzen. Perl führt das Skript wie gewohnt Zeile für Zeile aus, bis es zu dem Verweis auf eine Subroutine (hier `&f2c`) gelangt. Jetzt wechselt Perl zur Subroutinendefinition (den letzten Zeilen im Skript), führt den Block der Subroutine aus und kehrt dann dorthin zurück, wo die normale Ausführung unterbrochen wurde. In unserem Beispiel bedeutet dies, dass Perl, nachdem es von der Tastatur die Temperatur eingelesen hat, zur Subroutine `&f2c` wechselt, den Wert in Celsius umrechnet und dann das Ergebnis ausgibt.

> *Wenn ich sage, Perl wechselt zur Subroutine, meine ich nicht, dass Perl wirklich zu dieser Skriptzeile springt. Bevor Perl ein Skript ausführt, liest es das gesamte Skript in den Speicher und informiert sich über die Definitionen der Subroutinen. Später springt die Ausführung dann nur noch zu den Definitionen der Subroutinen und wieder zurück.*

Subroutinen definieren

An diesem simplen Beispiel haben Sie gesehen, wie man Subroutinen definiert:

```
sub subroutinenname {
   Anweisungen;
   ...
}
```

Eine Subroutinendefinition beginnt mit dem Wort `sub`, gefolgt vom Namen der Subroutine, gefolgt von einem Block. Der Block ist, wie Sie von Schleifen und Bedingungen her kennen, eine mit geschweiften Klammern umgebene Folge von Perl-Anweisungen. Ein Beispiel:

```
sub zahl_holen {
   print 'Bitte eine Zahl eingeben: ';
   chomp($zahl = <STDIN>);
}
```

Der Name einer Subroutine kann aus beliebig vielen alphanumerischen Zeichen und Unterstrichen bestehen, kommt nicht in Konflikt mit gleichnamigen Skalar-, Array- oder Hash-Variablen und unterscheidet zwischen Groß- und Kleinbuchstaben.

Subroutinendefinitionen können Sie überall dorthin stellen, wo auch eine normale Anweisung stehen kann - an den Anfang Ihres Skripts, ans Ende oder mitten hinein (sogar in andere Blöcke). Im allgemeinen empfiehlt es sich jedoch aus Gründen der Übersichtlichkeit, sie alle zusammen an den Anfang oder das Ende des Skripts zu setzen.

Subroutinen aufrufen

Mit einem kaufmännischen Und (`&`) vor dem Namen der Subroutine und optionalen runden Klammern danach rufen

Sie eine Subroutine auf:

```
&f2c;   #Fahrenheit zu Celsius
&zahl_holen()
```

In die Klammern kommen die Argumente, die Sie der Subroutine übergeben (mehr dazu später).

Das kaufmännische Und (&) ist optional; Sie können Subroutinen auch ohne & aufrufen. Einige Programmierer verwenden das &, weil es die Unterscheidung von Perl-eigenen Funktionen und selbst definierten (oder aus Modulen importierten) Subroutinen erleichtert. Ich rufe in diesem Buch Subroutinen immer mit & auf.

Es gibt einen Ausnahmefall, in dem das kaufmännische Und nicht optional ist: Wenn Sie sich indirekt auf eine Subroutine beziehen, sie aber nicht aufrufen, also zum Beispiel mit `defined` *überprüfen, ob die Subroutine definiert ist, dann müssen Sie das* `&` *verwenden.*

Außerdem könnten Sie in einigen Fällen auch die Klammern weglassen (insbesondere, wenn die Subroutine an einer früheren Stelle im Skript oder importierten Modul bereits deklariert wurde). Über die Deklaration von Subroutinen erfahren Sie im Vertiefungsabschnitt mehr. Ich verwende die Klammern in jedem Fall, um Verwirrung zu vermeiden - denn verkehrt sind sie nie.

Sie können eine Subroutine nicht nur aus dem Hauptkörper Ihres Skripts aufrufen, sondern auch aus anderen Subroutinen - die Sie vielleicht aus wieder anderen Subroutinen aufgerufen haben. In Perl können Sie Subroutinenaufrufe so tief verschachteln, wie Ihr Speicher erlaubt. Auf ein paar Dinge wie die Argumente oder die Gültigkeitsbereiche der lokalen Variablen müssen Sie dabei noch achten, aber dazu kommen wir noch.

Ein Beispiel: Statistik in Subroutinen

Mehr oder weniger zum Spaß habe ich mir die letzte Version unseres Statistikskripts geschnappt, in seine Einzelteile zerlegt und jedes in eine Subroutine gepackt. Im Hauptkörper des Skripts werden jetzt nur noch Subroutinen aufgerufen. Am Verhalten des Skripts hat sich nichts geändert, nur an seinem Aufbau. Das Ergebnis können Sie in Listing 11.1 bewundern.

Listing 11.1: Das Skript statsfunk.pl

```
1:  #!/usr/bin/perl -w
2:
3:  &initvars();
4:  &getinput();
5:  &printresults();
6:
7:  sub initvars {
8:      $input = "";     # temp Input
9:      @nums = ();      # Array: Zahlen
10:     %freq = ();      # Hash: Zahl-Haeufigkeit
11:     $maxfreq = 0;    # hoechste Haeufigkeit
12:     $count = 0;      # Anzahl Zahlen
13:     $sum = 0;        # Summe
14:     $avg = 0;        # Durchschnitt
15:     $med = 0;        # Median
16:     @keys = ();      # temp keys
17:     $totalspace = 0; # gesamte Breite des Histogramms
18: }
19:
20: sub getinput {
21:     while (defined ($input = <>)) {
22:         chomp ($input);
23:         $nums[$count] = $input;
24:         $freq{$input}++;
25:         if ($maxfreq < $freq{$input}) { $maxfreq = $freq{$input} }
26:         $count++;
27:         $sum += $input;
28:     }
```

```
29:
30: }
31:
32: sub printresults {
33:     @nums = sort { $a <=> $b } @nums;
34:
35:     $avg = $sum / $count;
36:     $med = $nums[$count /2];
37:
38      print "\nAnzahl der eingegebenen Zahlen: $count\n";
39:     print "Summe der Zahlen: $sum\n";
40:     print "Kleinste Zahl: $nums[0]\n";
41:     print "Groesste Zahl: $nums[$#nums]\n";
42:     printf("Durchschnitt: %.2f\n", $avg);
43:     print "Mittelwert: $med\n\n";
44:     &printhist();
45: }
46:
47: sub printhist {
48:     @keys = sort { $a <=> $b } keys %freq;
49:
50:     for ($i = $maxfreq; $i > 0; $i--) {
51:         foreach $num (@keys) {
52:             $space = (length $num);
53:             if ($freq{$num} >= $i) {
54:                 print( (" " x $space) . "*");
55:             } else {
56:                 print " " x (($space) + 1);
57:             }
58:             if ($i == $maxfreq) { $totalspace += $space + 1; }
59:         }
60:         print "\n";
61:     }
62:     print "-" x $totalspace;
63:     print "\n @keys\n";
64: }
```

Da diese Version des Skripts eigentlich nichts anderes macht als die letzte, gibt es hier nur ein paar Dinge anzumerken:

- Der einzige Teil des Skripts, der nicht in einer Subroutine steht, sind die Zeilen 3 bis 5, die Aufrufe der Subroutinen: Die erste initialisiert die Variablen, die zweite liest die Zahlen aus einer Datei, und die dritte berechnet die Ergebnisse und gibt sie aus.
- Im Skript sind vier Subroutinen definiert, aber nur drei werden am Anfang aufgerufen. Erst die Subroutine `printresults` ruft am Ende des Blocks die Subroutine `printhist` auf.

Aus Subroutinen Werte zurückgeben

Mit einem Aufruf der Subroutine allein, als einziger Anweisung in einer Zeile, wird die Subroutine ausgeführt - und das war's dann auch schon. Viel nützlicher sind Subroutinen, die einen Wert zurückgeben: Sie können sie in Ausdrücke oder Anweisungen einbauen oder als Argument an andere Subroutinen übergeben.

Wir haben schon besprochen, dass ein Block das Ergebnis der letzten Auswertung zurückgibt. Der Block, der eine Subroutine definiert, macht da keinen Unterschied. So liest zum Beispiel das folgende Skript zwei Zahlen ein und addiert sie:

```
$sum = &sumnums();
print "Summe: $sum\n";
sub sumnums {                                # Zahlen addieren
    print 'Geben Sie eine Zahl ein: ';
    chomp($zahl1 = <STDIN>);
    print 'Geben Sie eine weiter Zahl ein: ';
    chomp($zahl2 = <STDIN>);
    $zahl1 + $zahl2;
}
```

In der ersten Zeile haben wir hier einen Subroutinenaufruf auf der rechten Seite einer Zuweisung. Was wir hier

zuweisen ist nicht etwa die Subroutine selbst (wie sollte das aussehen?), sondern ihren Rückgabewert. Das Ergebnis der Addition (`$zahl1 + $zahl2`) in der letzten Zeile der Subroutine `&sumnums` ist der Wert, den die Subroutine zurückgibt, der an `$sum` zugewiesen und von `print` ausgegeben wird.

Der Haken an diesem Verhalten ist, dass die letzte Anweisung im Block zwar häufig, aber nicht immer den Wert liefert, den die Subroutine zurückgeben soll. Die Regel besagt, dass ein Block das Ergebnis der letzten Auswertung zurückgibt - und was zuletzt ausgewertet wird, muss nicht immer die letzte Anweisung; es könnte zum Beispiel auch die Schleifenbedingung einer `while`- oder `for`-Schleife oder ein Schleifensteuerbefehl sein.

Weil der Rückgabewert einer Subroutine nicht immer klar ist, empfiehlt es sich, in der letzten Zeile mit `return` (zurückgeben) explizit festzulegen, was sie zurückgeben soll. Die Funktion `return` nimmt einen Ausdruck als Argument entgegen, wertet ihn aus und gibt das Ergebnis zurück. Die letzte Zeile von `&sumnums` sähe dann so aus:

```
return $zahl1 + $zahl2;
```

Sie möchten mehrere Werte aus einer Subroutine zurückgeben? Kein Problem. Geben Sie einfach eine Liste zurück (aber achten Sie im Hauptteil Ihres Skripts darauf, dass Sie mit dieser Liste richtig umgehen). Das folgende Code-Fragment zum Beispiel ruft eine Subroutine auf, die die Werte aus einem Array verarbeitet und eine Liste von drei Werten zurückgibt, die dann den Elementen `$max`, `$min` und `$count` zugewiesen werden (wie wir der Subroutine das Array `@zahlen` übergeben, soll uns hier noch nicht interessieren):

```
($max, $min, $count) = &berechne(@zahlen);
# ...
sub berechne {
    # ...
   return ($wert1, $wert2, $wert3);
}
```

Und wenn Sie mehrere Listen aus einer Subroutine zurückgeben möchten? Das geht nicht oder zumindest nicht so einfach. Die Funktion `return` löst in einem Listenargument alle eventuellen Unterlisten und Hashes in ihre Elemente auf - sie interpoliert Arrays und Hashes in eine lineare Liste von Skalaren. Dieses Verhalten ist keine Besonderheit von `return`, sondern Perls allgemeine »Datenliefermethode« von und an Subroutinen (mehr dazu später). Brauchen Sie von einer Subroutine wirklich mehrere Listen, müssen Sie dafür sorgen, dass sich die gelieferte lineare Liste außerhalb der Subroutine wieder in die Teillisten zurückverwandeln läßt.

Lokale Variablen in Subroutinen

Ein Perl-Skript in Subroutinen aufzuteilen, die mit denselben globalen Variablen arbeiten, ist im Grunde eine reine Codeformatierungsübung. Vom besseren Überblick einmal abgesehen, bringen Ihnen solche Subroutinen keine wesentlichen Vorteile. Unser Ziel jedoch sind Subroutinen als eigenständige Einheiten, die eigene Variablen haben, nur die ihnen übergebenen Argumenten bearbeiteten und die angeforderten Ergebnisse liefern. Solche Subroutinen sind leichter zu handhaben, wiederzuverwenden und zu debuggen. Diesem Ziel werden wir uns jetzt Schritt für Schritt annähern; und wir beginnen mit den lokalen Variablen.

Bisher hatten wir fast ausschließlich mit globalen Variablen zu tun. Global bedeutet, dass diese (Skalar-, Array- und Hash-) Variablen überall im Skript zur Verfügung stehen und so lange existieren, wie das Skript läuft. Sie haben zwar bereits ein paar Ausnahmen gesehen - zum Beispiel die Schleifenvariable von `foreach` oder die Treffervariablen in regulären Ausdrücken - doch größtenteils gehörten unsere Variablen dem globalen Gültigkeitsbereich an.

Lokale Variablen sind, in Verbindung mit Subroutinen, Variablen, die erst beim Start der Subroutine zu existieren beginnen, nur in der Subroutine zur Verfügung stehen und wieder verschwinden, sobald die Subroutine beendet ist. Ansonsten sind es ganz normale Variablen - sie sehen weder anders aus, noch werden sie anders verwendet als ihre globalen Pendants.

Sie haben in Perl zwei Möglichkeiten, lokale Variablen zu erstellen. Die einfachere behandeln wir hier, die andere (und worin sie sich unterscheidet) am Tag 13.

Zum Deklarieren einer lokalen Variablen für eine Subroutine setzen Sie den Operator `my` vor die Variable, wenn Sie sie initialisieren oder das erste Mal verwenden:

```
my $x = 1;  # $x ist jetzt lokal
```

Wenn Sie mehrere lokale Variablen deklarieren möchten, müssen Sie die Liste in Klammern setzen, sonst wirkt sich

```
my ($a, $b, $c);   # drei lokale Variablen, alle undefiniert.
```

Im vorigen Abschnitt haben wir eine Subroutine erstellt, die zwei Zahlen erfragt und addiert hat. Die beiden Variablen `$zahl1` und `$zahl2` waren global. Mit einer zusätzlichen Zeile machen wir sie zu lokalen Variablen:

```
$sum = &sumnums();
print "Summe: $sum\n";
sub sumnums {
    my ($zahl1, $zahl2);
    print 'Geben Sie eine Zahl ein: ';
    chomp($zahl1 = <STDIN>);
    print 'Geben Sie eine weiter Zahl ein: ';
    chomp($zahl2 = <STDIN>);
    return ($zahl1 + $zahl2);
}
```

In dieser Version deklariert die Zeile my ($zahl1, $zahl2) zwei lokale Variablen. Diese sind nur innerhalb der Subroutine sumnums existent und sichtbar.

Was passiert, wenn vor dem Subroutinenaufruf schon eine globale Variable `$zahl1` vorhanden war? Perl hat damit kein Problem. Eine mit `my` deklarierte Variable verdeckt eine globale Variable gleichen Namens vor »ihrer Subroutine«. Oder andersherum formuliert: In einer Subroutine ist eine globale Variable unsichtbar, sobald in der Subroutine eine gleichnamige lokale Variable deklariert wird. Wenn Perl die Subroutine verläßt, verschwindet die `my`-Variable wieder, und Sie haben einen freien Blick (und Zugriff) auf die globale Variable. Weil dies - vor allem bei der Fehlersuche - sehr verwirren kann, ist es im allgemeinen eine gute Idee, Ihre lokalen Variablen anders zu nennen als Ihre globalen.

Über den Paketnamen kommen Sie in einer Subroutine aber auch an versteckte globale Variablen heran. Wir besprechen das an Tag 13.

Betrachten wir folgendes Beispiel:

```
#!/usr/bin/perl -w
$wert = 0;
print "vor subaufruf - \$wert ist $wert\n";
&subaufruf();
print "nach subaufruf - \$wert ist $wert\n";
sub subaufruf {
    my ($wert);
    $wert = 10;
    print "innerhalb von subaufruf - \$wert ist $wert\n";
}
```

Dieses Skriptchen produziert folgende Ausgabe:

```
vor subaufruf - $wert ist 0
innerhalb von subaufruf - $wert ist 10
nach subaufruf - $wert ist 0
```

Sie sehen es selbst: Die lokale Variable `$wert`, die in der Subroutine deklariert wird, hat mit der globalen nichts zu tun. Sie in der Subroutine zu verändern, wirkt sich nicht im geringsten auf die globale Namensschwester aus.

Wenn wir jetzt aus `subaufruf` eine weitere Subroutine namens `subaufruf2` aufrufen und dort wieder eine `my`-Variable `$wert` mit 0 initialisieren und inkrementieren würden, hätten wir drei völlig verschiedene Variablen namens `$wert` und folgende Ausgabe:

```
vor subaufruf - $wert ist 0
```

```
innerhalb von subaufruf - $wert ist 1
innerhalb von subaufruf2 - $wert ist 1
nach subaufruf - $wert ist 0
```

Auch wenn Sie Subroutinen ineinander verschachteln - eine Subroutine aus einer anderen Subroutine aufrufen -, sind die in der aufrufenden Subroutine deklarierten `my`- Variablen in der aufgerufenen Subroutine unsichtbar und umgekehrt.[3] Hier unterscheidet sich Perl von vielen anderen Sprachen, die die Gültigkeitsbereiche von lokalen Variablen in verschachtelten Subroutinen »kaskadieren«. `my`-Variablen existieren in Perl wirklich einzig und allein in der Subroutine, in der sie deklariert werden, nirgendwo sonst.

Für die Computerexperten unter den Lesern heißt dies, dass die `my`-Variablen lexikalische statt dynamische Gültigkeit haben. Sie existieren nur innerhalb des Blocks, in dem sie definiert sind. Dazu gleich mehr.

Werte an Subroutinen übergeben

Jetzt wissen Sie, wie Subroutinen Werte zurückliefern und Daten in lokalen Variablen speichern. Fehlt nur noch, wie man ihnen Werte übergibt. Damit kommen wir auf die Argumente zu sprechen.

Argumente übergeben

Das Konzept der Datenübergabe an eine Subroutine (und zurück) ist in Perl ebenso schlicht wie ergreifend. Während Sie in anderen Sprachen schon beim Schreiben einer Subroutine genau festlegen müssen, wie viele Argumente welchen Typs sie entgegennehmen kann, packt bzw. expandiert Perl einfach alle Ihre Argumente in eine einzige lineare Liste. Bei skalaren Argumenten ist das keine besondere Aktion:

```
&meine_subroutine(1, 2, 3);
```

`&meine_subroutine` erhält in diesem Fall eine Liste von drei numerischen Werten. Achten Sie aber auf Listenargumente:

```
&meine_subroutine(@liste, @andere_liste);
```

In diesem Fall expandiert Perl die beiden Arrayvariablen und speichert alle ihre Elemente nacheinander in einer einzigen Liste - der Liste, die es an die Subroutine übergibt. Die Identität der Arrays geht in dieser Liste verloren (die Inhalte sind zwar noch alle da, aber wenn Sie nicht vorgesorgt haben, läßt sich nicht mehr herausfinden, aus welchem Array sie stammen).

Bei Hashes ist es ähnlich: Ein Hash wird (wie immer, wenn Sie ihn einer Liste zuweisen) in seine Bestandteile zerlegt. Seine Schlüssel und Werte werden der Liste hinzugefügt, die die Subroutine schließlich entgegennimmt.

Doch was, wenn Ihre Argumente wirklich aus mehreren Arrays oder Hashes bestehen sollen? Keine Panik, es gibt ein paar Möglichkeiten, Perls Datenübergabeverhalten zu umgehen. Eine Möglichkeit wäre, Ihre Arrays oder Hashes als globale Variablen zu speichern und sich in der Subroutine einfach auf sie zu beziehen. Mit ein paar vorbereitenden Tricks könnten Sie die Teillisten auch aus der Argumentliste rekonstruieren (zum Beispiel, indem Sie als Argument auch die Länge der Arrays mit übergeben). Die dritte und wahrscheinlich die beste Möglichkeit ist, die Argumente als Referenzen zu übergeben. Die Struktur der Arrays und Hashes bliebe dabei erhalten - mit Referenzen befassen wir uns an Tag 19.

Argumente in der Subroutine entgegennehmen

Okay, die Argumente wurden in einer linearen Liste an die Subroutine übergeben. Und nun? Wie kommen Sie vom Körper der Subroutine aus an diese Parameterliste heran?[4]

Die an eine Subroutine übergebene Liste (auch Parameterliste dieser Subroutine genannt) wird im lokalen Spezialarray `@_` gespeichert. Auf dieses Array können Sie mit den bekannten Techniken zugreifen. `@_` ist eine lokale Variable der Subroutine, das heißt ihr Inhalt ist nur innerhalb der Subroutine sichtbar. Wenn Sie aus einer Subroutine heraus eine andere aufrufen, erhält diese zweite Subroutine ihr eigenes `@_`- Array.

Hier ein Beispiel einer Subroutine, die zwei Argumente addiert:

```
&addiere(2,3);  #Aufruf der Subroutine
sub addiere {
   return $_[0] + $_[1];
}
```

Die beiden Argumente an die Subroutine - hier 2 und 3 - landen im Array `@_`. Mit den Ausdrücken `$_[0]` und `$_[1]` greifen Sie auf die ersten beiden Elemente dieses Arrays zu. Genau wie `$foo[0]` und `$foo` sich auf unterschiedliche Dinge beziehen, ist auch `$_[0]` etwas anderes als `$_` (`$_[0]` ist das erste Element in der Argumentliste der Subroutine, `$_` ist die Default-String-Spezialvariable).

Diese Subroutine ist nicht gerade intelligent - sie addiert lediglich die ersten beiden Argumente, ganz egal, wie viele Sie ihr übergeben haben. Weil Sie nicht kontrollieren können, wie viele Argumente eine Subroutine entgegennehmen kann, müßten Sie beim Aufruf auf die korrekte Anzahl achten oder in der Subroutine die Zahl der Argumente (also der Elemente in `@_`) überprüfen oder die Subroutine so umschreiben, dass sie etwas großzügiger mit mehreren Argumenten umgeht. Sie könnten sie zum Beispiel alle Argumente addieren lassen, wie viele es auch sein mögen:

```
sub addiere {
   my $sum = 0;
   foreach $i (@_) {
      $sum += $i;
   }
```

In den neueren Perl-Versionen (seit Release 5.003) haben Sie die Möglichkeit, bei der Deklaration einer Subroutine mit sogenannten Prototypen auch Art und Anzahl der Argumente zu bestimmen. Da dieses Feature relativ neu ist, möchten Sie sich in Ihren Skripts aber vielleicht nicht darauf verlassen. *Mehr zu Prototypen auf Seite 332.*

Perl hat keine eigene Methode zum expliziten Benennen von eingehenden Argumenten, aber ein sehr gebräuchlicher »Trick« macht praktisch das gleiche: Weisen Sie als erstes die Argumente von `@_` lokalen Variablen zu:

```
sub foo {
   my($max, $min, $inc) = @_;
   ...
}
```

Dann können Sie sich auf die Argumente mit einem leichter zu merkenden Variablennamen beziehen und die Positionen der einzelnen Argumente im Array `@_` getrost vergessen. Eine andere (sehr gebräuchliche) Möglichkeit führt zum gleichen Ergebnis: Erinnern Sie sich an die Funktion `shift`? Bequemerweise entfernt `shift`, wenn es ohne Argumente innerhalb einer Subroutine aufgerufen wird, das erste Element von `@_` und gibt es zurück (`pop` macht dasselbe mit dem letzten Element von `@_`):

```
sub foo {
   my $max = shift;
   my $min = shift;
   my $inc = shift;
   ...
}
```

Eine Anmerkung zu den Werten in @_

Die Werte von `@_` sind sogenannte Referenzparameter: Sie verweisen implizit auf die eigentlichen skalaren Argumente, die von außen übergeben wurden. Das bedeutet, dass, wenn Sie direkt im Array `@_` ein Element verändern, unter Umständen auch einen Wert außerhalb der Subroutine verändern. Betrachten wir am besten, was folgendes Beispiel ausgibt:

```
#!/usr/bin/perl -w
$a = "a";
@b = ($a, 0);
%c = ('rom', '1', 'berlin', '2');
print "\n vor subaufruf:  \$a = $a   \@b = @b   \%c = ";
foreach (keys %c) {
    print "$_ : $c{$_}  ";
}
&subaufruf($a, 10, @b, %c,);
print "\n nach subaufruf:  \$a = $a  \@b = @b  \%c = ";
foreach (keys %c) {
    print "$_ : $c{$_}  ";
}
sub subaufruf {
    my ($a, $b, $c, $d, $e ) = @_;
    print "\n start subaufruf: \@ = @_ ";
    $a = "my a";
    $b = 20;
    $c = "my b1";
    $_[0] = "AAA";
    $_[2] = B1;
    $_[3] = B2;
    $_[4] = "PARIS";
    print "\n ende subaufruf: \$a = $a  \$b = $b  \$c = $c \n";
    print "                  \@ = @_";
}
```

Dieser Code produziert folgenden Output (ohne dass wir mit `subaufruf` etwas zurückgeben):

```
 vor subaufruf:  $a = a   @b = a 0   %c = berlin : 2  rom : 1
 start subaufruf: @ = a 10 a 0 berlin 2 rom 1
 ende subaufruf: $a = my a  $b = 20  $c = my b1
                 @ = AAA 10 B1 B2 PARIS 2 rom 1
 nach subaufruf:  $a = AAA  @b = B1 B2  %c = berlin : 2  rom : 1
```

Hier sehen Sie, wie die globalen Variablen durch die Zuweisungen an `$_[0]` etc. verändert werden. Es ist, als stünden die Variablen selbst in der Parameterliste `@_` - so etwas nennt man dann Referenzübergabe. Nur auf den Hash haben wir keinen Einfluß (`$_[4] = "PARIS"` wirkt sich nicht auf `%c` aus), wir erhalten lediglich die aktuellen Werte seiner Elemente (ja, die Werte der Schlüssel und die Werte der Werte). So etwas nennt man dann Wertübergabe.

Am Anfang von `subaufruf` weisen wir `@_` einer `my`-Liste von lokalen Variablen zu.[5] Weil wir hier im Grunde eine Kopie der Parameterwerte machen, berühren wir keine der Originalvariablen, wenn wir die (Kopien in den) lokalen Variablen verändern - das ist die einfachere, verständlichere und empfehlenswerte Art des Umgangs mit unseren Argumenten.

Subroutinen und Kontext

Parameterübergabe, Subroutinendefinition, Rückgabewerte - all dies sind auch in anderen Sprachen wichtige Fragen bei der Definition von Funktionen. Aber das hier ist Perl; es muss noch ein paar Eigenar(tigkei)ten geben. Sie wissen bereits, dass Arrays und Hashes in einer Subroutinen-Parameterliste ihre Identität verlieren, weil sie aufgedröselt werden. Eine andere Eigenart von Perl ist der Kontext.

Da Subroutinen sowohl in Skalar- als auch in Listenkontext aufgerufen werden und sowohl einen Skalar als auch eine Liste zurückgeben können, spielt die Frage nach dem Kontext auch beim Definieren von Subroutinen eine Rolle - oder zumindest beim Aufrufen von Subroutinen: Rufen Sie Subroutinen, die Listen zurückgeben, nicht in Skalarkontext auf, wenn Sie nicht genau wissen, zu welchen Ergebnissen das führt.

Vielleicht ***möchten*** Sie aber gerade eine Subroutine schreiben, die sich je nach Kontext, in dem sie aufgerufen wird, unterschiedlich verhält. Dann hat Perl eine nette Funktion für Sie: `wantarray`. Die Funktion `wantarray` gibt ***wahr*** zurück, wenn die momentan laufende Subroutine in Listenkontext aufgerufen wurde, ***falsch*** wenn in Skalarkontext (»`wantlist`«, »will Liste« wäre ein passenderer Name, aber aus historischen Gründen heißt sie eben `wantarray`). So können Sie zum Beispiel vom Kontext abhängig machen, welchen Wert Sie aus einer Subroutine zurückgeben:

```
sub skalar_oder_liste {
    # ...
    if (wantarray()) {
        return @liste;
    } else { return $skalar}
}
```

Gehen Sie mit diesem Feature aber vorsichtig um. Bedenken Sie, welche Verwirrung schon Perl-Funktionen stiften können, die sich je nach Kontext unterschiedlich verhalten (und so manchen Blick in die Dokumentation erfordern). In vielen Fällen ist es besser, einen der Subroutine selbst entsprechenden Wert zurückzugeben und mit diesem in der Anweisung, die die Subroutine aufgerufen hat, entsprechend umzugehen.

Ein weiteres Beispiel: Statistik mit Menüführung

Verändern wir zum zweiten Mal in dieser Lektion unser gutes altes Statistikbeispiel und schöpfen wir dabei aus dem gesamten Wissen, das Sie im Laufe dieses Buches angehäuft haben. Diesmal soll das Skript nicht alles hintereinander weg ausführen, sondern die verschiedenen Operationen erst einmal in einem Menü anzeigen. Sie haben dann die Wahl, welche dieser Operationen Sie auf den aus Ihrer Input-Datei gelesenen Zahlen ausführen möchten.

Anders als im Skript ***mehrnamen.pl*** vom Tag 8 steuern wir dieses Menü hier aber nicht über eine große `while`-Schleife und diversen Bedingungen, sondern über Subroutinen. Die verschiedenen Berechnungen verschieben wir in die Subroutinen, in denen sie gebraucht werden.

Weil dieses Skript ziemlich lang ist, zeige ich Ihnen diesmal nicht zuerst den kompletten Code und bespreche ihn dann hinterher, sondern ich fange mit den Erklärungen der einzelnen Schritte an und stelle das vollständige Listing ans Ende dieses Abschnitts.

Beginnen wir mit dem Hauptkörper des Skripts. In unseren bisherigen Versionen mussten wir erst einmal eine Menge Variablen initialisieren - globale Variablen. Diesmal ist unser Initialsierungsteil nur noch zwei Zeilen lang, denn wir brauchen nur zwei globale Variablen. Eine für den Input (die Zahlen, die wir aus der angegeben Datei lesen) und eine, mit der wir stets im Blick haben, ob das Skript beendet werden soll oder nicht. Alle anderen deklarieren wir in den Subroutinen als lokale Variablen:

```
#!/usr/bin/perl -w
@zahlen = (); #Input aus Datei, eine Zahl pro Zeile
$choice = "";
&getinput();
```

Das Skript beginnt mit dem Aufruf der Subroutine `&getinput`. Diese Subroutine liest die Daten aus der Eingabedatei und speichert sie im Array `@nums`. Diese Version von `getinput` ist wesentlich kürzer als die letzte - wir kümmern uns hier nicht um die Häufigkeit der einzelnen Werte, ihre Gesamtsumme etc., und wir verwenden `$_` anstatt einer temporären Input-Variablen. Das einzige, was wir außer dem Einlesen hier noch erledigen, ist das Sortieren der Zahlen im Array:

```
sub getinput {
    my $count = 0;
    while (<>) {
        chomp;
        $nums[$count] = $_;
        $count++;
    }
    @nums = sort { $a <=> $b } @nums;
}
```

Nachdem wir das Input gespeichert haben, kommen wir zum Kern unseres menügesteuerten Skripts, einer `while`-Schleife, die für jede gewählte Menüoption jeweils die entsprechende Subroutine aufruft:

```
&getinput();
while ($choice !~ /q/i) {
    $choice = &printmenu();
  SWITCH: {
      $choice =~ /^1/ && do {
```

```
                    &printdata(); last SWITCH; };
      $choice =~ /^2/ && do {
                    &countsum (); last SWITCH; };
      $choice =~ /^3/ && do {
                    &maxmin(); last SWITCH; };
      $choice =~ /^4/ && do {
                    &meanmed(); last SWITCH; };
      $choice =~ /^5/ && do {
                    &printhist(); last SWITCH; };
  }
}
```

Schauen Sie! Ein Switch! Hier »simulieren« wir mit Pattern Matching, `do`-Anweisungen und dem Label (`SWITCH`) eine `switch`-Anweisung. Bei jedem möglichen Wert von `$choice`, 1 bis 5, rufen wir eine andere Subroutine auf und verlassen mit `last SWITCH` den gesamten Block. Beachten Sie, dass ein Block mit einem Label zwar keine Schleife ist, Sie ihn aber mit den Schleifensteuerbefehlen genauso verlassen können.

Welchen Wert `$choice` in dem `SWITCH`-Block überhaupt hat, bringen wir vorher mit dem Subroutinenaufruf `&printmenu` in Erfahrung. Beachten Sie, dass die `while`- Schleife immer wieder das Menü ausgibt und die gewählte Operation ausführt - bis `printmenu` ein `q` zurückliefert. Dann bricht die Schleife ab, und das Skript wird beendet.

Die Subroutine `printmenu` zeigt einfach die einzelnen Optionen an, liest und überprüft den vom Benutzer eingegebenen Wert und liefert diesen zurück:

```
sub printmenu {
    my $in = "";
    print "Was moechten Sie ausgeben? (Beenden mit Q): \n";
    print "1. eine Liste der Zahlen \n";
    print "2. die Anzahl und Summe der Zahlen\n";
    print "3. die kleinste und die groesste Zahl\n";
    print "4. den Durchschnitts- und den Medianwert\n";
    print "5. ein Diagramm, wie oft jede Zahl vorkommt.\n";
    while () {
        print "\nIhre Auswahl   --> ";
        chomp($in = <STDIN>);
        if ($in =~ /^\d$/ || $in =~ /^q$/i) {
            return $in;
        } else {
            print "Ungueltige Eingabe. 1-5 oder Q, bitte.\n";
 }
    }
}
```

Gehen wir die im Menü gezeigten Optionen von oben nach unten durch. Möchte der Benutzer sich eine Liste der Zahlen anzeigen lassen, gibt er eine `1` ein, `$choice` erhält den Wert eins, und der `SWITCH`-Block ruft `&printdata` auf. Die Subroutine `printdata` durchläuft einfach das (globale) Array `@nums` und gibt die Elemente aus. Weil wir es aber übersichtlich lieben, möchten wir nach jeweils zehn Zahlen eine neue Zeile beginnen. Deshalb verfolgen wir in einer lokalen Variablen `$i` die Anzahl der Elemente, und wenn sie bei 10 angelangt ist, geben wir einen Zeilenvorschub aus und setzen sie zurück auf `1`:

```
sub printdata {
    my $i = 1;
    print "die Zahlen: \n";
    foreach $num (@nums) {
        print "$num ";
        if ($i == 10) {
            print "\n";
            $i = 1;
        } else { $i++; }
    }
    print "\n\n";
}
```

Die zweite Menüoption ist die Ausgabe von Anzahl und Gesamtsumme unserer Zahlen beziehungsweise der Aufruf der Subroutine `&countsum`:

```
sub countsum {
    print "Anzahl der Zahlen: ", scalar(@nums), "\n";
    print "Summe der Zahlen: ", &sumnums(),"\n\n";
}
```

Diese Subroutine wiederum ruft `&sumnums()` auf, die die Summe aller Zahlen berechnet:

```
sub sumnums {
    my $sum = 0;
    foreach $num  (@nums) {
        $sum += $num;
    }
    return $sum;
}
```

In den bisherigen Versionen des Skripts haben wir die Summe gleich beim Einlesen der Zahlen berechnet, hier hingegen machen wir daraus einen separaten Vorgang. Sie könnten einwenden, dass das weniger effizient ist, zumal wir die Summe ja auch zur Berechnung des Durchschnittswerts brauchen - doch andererseits sparen wir uns dadurch Arbeit, bis sie wirklich nötig ist.

Die dritte Möglichkeit ist die Ausgabe der kleinsten und der größten Zahl. Hierfür müssen wir gar nicht viel herumrechnen - da unser Array sortiert ist, finden wir kleinstes und größtes Element am Anfang bzw. am Ende:

```
sub maxmin {
    print "Kleinste Zahl: $nums[0]\n";
    print "Groesste Zahl: $nums[$#nums]\n\n";
}
```

Entscheidet der Benutzer sich für die vierte Option, ermitteln wir Durchschnittswert und Median wie in den bisherigen Versionen des Skripts (nur dass wir uns die Summe von einer Subroutine liefern lassen):

```
sub meanmed {
    printf("Durchschnitt: %.2f\n", &sumnums() / scalar(@nums));
    print "Median: $nums[@nums / 2]\n\n";
}
```

Bleibt nur noch die fünfte Möglichkeit, das Histogramm auszugeben. Wie in den bisherigen Versionen ist dies der komplexeste Teil des Skripts. Hier allerdings packen wir alles, was mit dem Histogramm zu tun hat, in eine Subroutine `printhist`, anstatt die nötigen Werte über das gesamte Skript zu verteilen. Das bedeutet zum einen mehr lokale Variablen und Rechenarbeit als für die anderen Subroutinen. Zum anderen wird aber der Einleseprozeß nicht von irgendwelchen Häufigkeitsberechnungen verlangsamt, die am Ende vielleicht gar niemand braucht - und wir haben nicht während des gesamten Skriptablaufs den Häufigkeits-Hash im Speicher hängen:

```
sub printhist {
    my %freq = ();
    my $maxfreq = 0;
    my @keys = ();
    my $space = 0;
    my $totalspace = 0;
    my $num;
    my $i;
    # freq-Hash erstellen, maxfreq festlegen
    foreach $num (@nums) {
        $freq{$num}++;
        if ($maxfreq < $freq{$num}) {
            $maxfreq = $freq{$num}
        }
    }
# Hash ausgeben
    @keys = sort { $a <=> $b } keys %freq;
    for ($i = $maxfreq; $i > 0; $i--) {
        foreach $num (@keys) {
            $space = (length $num);
            if ($freq{$num} >= $i) {
                print( (" " x $space) . "*");
            } else {
                print " " x (($space) + 1);
```

```
            }
            if ($i == $maxfreq) {
                $totalspace += $space + 1;
            }
        }
        print "\n";
    }
    print "-" x $totalspace;
    print "\n @keys\n\n";
}
```

Abgesehen davon, dass wir alle Häufigkeitsberechnungen in dieser Subroutine zusammengefaßt haben, hat sich an dem Vorgang selbst nicht viel geändert. Er ist nur etwas eigenständiger geworden.

Das war's schon. Listing 11.2 zeigt das gesamte Skript.

Listing 11.2: Das Skript statsmenue.pl

```
#!/usr/bin/perl -w
@nums = (); # Input aus Datei, eine Zahl pro Zeile
$choice = "";
# Hauptskript
&getinput();
while ($choice !~ /q/i) {
    $choice = &printmenu();
  SWITCH: {
      $choice =~ /^1/ && do {
                       &printdata(); last SWITCH; };
      $choice =~ /^2/ && do {
                       &countsum (); last SWITCH; };
      $choice =~ /^3/ && do {
                       &maxmin(); last SWITCH; };
      $choice =~ /^4/ && do {
                       &meanmed(); last SWITCH; };
      $choice =~ /^5/ && do {
                       &printhist(); last SWITCH; };
  }
}
# Input aus Datei lesen und dann sortieren
sub getinput {
    my $count = 0;
    while (<>) {
        chomp;
        $nums[$count] = $_;
        $count++;
    }
    @nums = sort { $a <=> $b } @nums;
}
# das Menue. Beenden mit Q
sub printmenu {
    my $in = "";
    print "Was moechten Sie ausgeben? (Beenden mit Q): \n";
    print "1. eine Liste der Zahlen \n";
    print "2. die Anzahl und Summe der Zahlen\n";
    print "3. die kleinste und die groesste Zahl\n";
    print "4. den Durchschnitts- und den Medianwert\n";
    print "5. ein Diagramm, wie oft jede Zahl vorkommt.\n";
    while () {
        print "\nIhre Auswahl   --> ";
        chomp($in = <STDIN>);
        if ($in =~ /^\d$/ || $in =~ /^q$/i) {
            return $in;
        } else {
            print "Ungueltige Eingabe. 1-5 oder Q, bitte.\n";
 }
    }
}
# die Daten ausgeben, zehn Zahlen pro Zeile
sub printdata {
    my $i = 1;
    print "die Zahlen: \n";
```

```
    foreach $num (@nums) {
        print "$num ";
        if ($i == 10) {
            print "\n";
            $i = 1;
        } else { $i++; }
    }
    print "\n\n";
}
# Anzahl und Summe der Elemente ausgeben
sub countsum {
    print "Anzahl der Zahlen: ", scalar(@nums), "\n";
    print "Summe der Zahlen: ", &sumnums(),"\n\n";
}
# Summe der Zahlen berechnen
sub sumnums {
    my $sum = 0;
    foreach $num  (@nums) {
        $sum += $num;
    }
    return $sum;
}
# kleinste und groesste Zahl ausgeben
sub maxmin {
    print "Kleinste Zahl: $nums[0]\n";
    print "Groesste Zahl: $nums[$#nums]\n\n";
}
# Durchschnitt und Median ausgeben
sub meanmed {
    printf("Durchschnitt: %.2f\n", &sumnums() / scalar(@nums));
    print "Median: $nums[@nums / 2]\n\n";
}
# Das Histogramm ausgeben. Haeufigkeits-Hash erstellen und ausgeben
sub printhist {
    my %freq = ();
    my $maxfreq = 0;
    my @keys = ();
    my $space = 0;
    my $totalspace = 0;
    my $num;
    my $i;
    # freq-Hash erstellen, maxfreq festlegen
    foreach $num (@nums) {
        $freq{$num}++;
        if ($maxfreq < $freq{$num}) {
            $maxfreq = $freq{$num}
        }
    }
    # Hash ausgeben
    @keys = sort { $a <=> $b } keys %freq;
    for ($i = $maxfreq; $i > 0; $i--) {
        foreach $num (@keys) {
            $space = (length $num);
            if ($freq{$num} >= $i) {
                print( (" " x $space) . "*");
            } else {
                print " " x (($space) + 1);
            }
            if ($i == $maxfreq) {
                $totalspace += $space + 1;
            }
        }
        print "\n";
    }
    print "-" x $totalspace;
    print "\n @keys\n\n";
}
```

Vertiefung

Wie immer habe ich Ihnen noch nicht alles gesagt und möchte Sie in diesem Abschnitt zumindest darauf aufmerksam machen.

In der ***perlsub***-Manpage sind Perl-Subroutinen und wie Sie sie definieren, deklarieren und verwenden, detailliert beschrieben. Weil `my` ein Operator ist, finden Sie mehr Informationen über `my` in der ***perlfunc-***Manpage (wir werden uns aber auch am Tag 13 im Zusammenhang mit Gültigkeitsbereichen noch einmal mit ihm befassen). Sehen Sie also in diese Seiten, wenn die folgenden Themen Sie genauer interessieren.

Die Klammern um Argumente weglassen

Beim Aufrufen von Perl-eigenen Funktionen können Sie die Klammern um die Argumente auch weglassen, wenn Perl dann noch verstehen kann, wo die Argumente anfangen und aufhören. Das gleiche gilt auch für Ihre eigenen Subroutinen, allerdings nur unter zwei weiteren Bedingungen:

- Sie rufen die Subroutine auch ohne das `&`-Präfix auf (mit dem Präfix würde Perl den aktuellen Wert von `@_` an die Subroutine übergeben).
- Sie haben die Subroutine im Skript bereits vor diesem Aufruf deklariert.

Es ist Perl eigentlich gleichgültig, ob Sie eine Subroutine im Skript vor oder nach ihrem Aufruf deklarieren (in einigen Sprachen ***müssen*** Sie sie zuerst deklariert haben). Die Ausnahme von dieser Regel ist das Weglassen der Klammern.

Hier kommt der kleine, aber feine Unterschied zwischen Deklaration und Definition von Subroutinen ins Spiel. Jede Subroutinendefinition ist gleichzeitig auch ihre Deklaration; sagt also Perl: »Hallo, es gibt hier eine Subroutine mit diesem Namen.« Der dann folgende Block legt fest, was die Subroutine im einzelnen macht. Sie können eine Subroutine aber auch ohne diesen Block »vordeklarieren«:

```
sub meine_subroutine;
```

Dann weiß Perl, dass mit `meine_subroutine` eine Subroutine gemeint ist, und durchsucht das Skript nach ihrer Definition.

Eine auf diese Art vordeklarierte Subroutine können Sie auch ohne Klammern um die Argumente aufrufen, wenn die eigentliche Definition erst am Ende des Skripts steht (aber irgendwo muss sie stehen, vergessen Sie sie nicht).

Es ist unter Perl-Programmierern allerdings gängige Praxis, die Klammern immer zu setzen, ob sie unbedingt nötig sind oder nicht, weil Subroutinenaufruf mit Klammern leichter lesbar und weniger fehlerträchtig - also empfehlenswert - ist.

Mit @_ Argumente an Subroutinen übergeben

Wie Sie wissen, ist `@_` das lokale Spezialarray einer Subroutine, das alle Argumente an diese Subroutine enthält. Sie können `@_` aber nicht nur innerhalb, sondern auch außerhalb der Subroutine zum Festlegen der Argumente verwenden.

Wenn Sie `@_` zum Beispiel im Hauptkörper Ihres Skripts ein paar Elemente zuweisen und dann eine vordeklarierte Subroutine ohne Argumente aufrufen, übergibt Perl dieser Subroutine das Array `@_` mit den von Ihnen gesetzten Werten:

```
sub meine_subroutine;
@_ = ('dies', 'das', 'sonstwas');
&meine_subroutine; # uebergibt den aktuellen Wert von @_
```

Weil `&meine_subroutine` hier ohne Klammern, aber mit Präfix aufgerufen wird, übergibt Perl ihr die Elemente von `@_` als Parameterliste. Das kann nützlich sein, wenn Sie zum Beispiel zehn verschiedene Subroutinen mit den gleichen Argumenten aufrufen möchten.

Bei verschachtelten Subroutinen funktioniert es genauso. Sie können die Parameter einer Subroutine (nämlich den aktuellen Inhalt von `@_`) einfach an die nächste Subroutine weitergeben. Beachten Sie aber, dass die Subroutine bereits deklariert sein muss und dass Sie ihr »selbst gefülltes« `@_` überschreiben, sobald Sie die Subroutine mit irgendeinem Argument aufrufen, und sei es nur ein leeres Paar Klammern.

Anonyme Subroutinen

Anonyme Subroutinen sind Subroutinen ohne Namen, die ähnlich funktionieren wie die Pointer (Zeiger) auf Funktionen in C und sozusagen erst zur Laufzeit zum Leben erwachen. Mit anonymen Subroutinen können Sie Referenzen auf eine Subroutine erzeugen und über diese Referenzen auf sie zugreifen - aber das geht hier zu weit. Wir befassen uns mit anonymen Subroutinen am Tag 19, wenn wir Referenzen behandeln.

Prototypen

Ich habe am Anfang des Buches erwähnt, dass kleinere Perl-Updates in erster Linie Fehler beheben, doch manchmal auch ganze neue Features mitbringen. Eins dieser Features sind Prototypen, Bestandteil von Perl seit Version 5.003. Mit Prototypen können Sie Ihre Subroutinen so deklarieren, dass sie wie Perl-eigene Funktionen aussehen und arbeiten - das heißt, dass Sie für eine Subroutine Typ und Anzahl der Argumente festlegen können, statt sie jede x-beliebige Parameterliste entgegennehmen zu lassen.

Prototypen wirken sich nur auf Subroutinen aus, die ohne das `&`-Präfix aufgerufen werden. Sie können dieselbe Subroutine auch mit `&` aufrufen, nur wird der Prototyp dann ignoriert. Wofür Sie sich jeweils entscheiden, bleibt Ihrer Lust und Laune überlassen.

Eine Subroutine mit Prototyp deklarieren oder definieren Sie folgendermaßen:

```
sub NAME (PROTOTYP);   #nur (Vor-) Deklaration
sub NAME (PROTOTYP) #Deklaration mit anschließender Definition
         { ...      # Definition
}
```

Wobei `NAME` hier für den Namen Ihrer Subroutine steht. Den `PROTOTYP` setzen Sie aus folgenden Sonderzeichen zusammen, die sich auf Anzahl und Typ der erlaubten Argumente repräsentieren:

- `$` steht für eine Skalarvariable, `@` für ein Array, `%` für einen Hash. Ohne Backslash müssen `@` und `%` am Ende stehen, weil sie Listenkontext erzwingen und alle restlichen Argumente »auffressen«.
- Ein Semikolon trennt ein erforderliches von einem optionalen Argument.
- Ein mit einem Backslash versehenes Zeichen fordert ein Argument, das mit diesem Zeichen beginnt.

So verlangt zum Beispiel eine mit dem Prototyp `($$)` deklarierte Subroutine zwei Skalarvariablen als Argumente. Der Prototyp `($$;@)` fordert zwei Skalare und eine optionale Liste[6], genauer eine Arrayvariable (die ja mit einem `@` beginnt).

So sehr Prototypen Ihnen auch gefallen mögen - bedenken Sie, dass sie nur in neueren Perl-Versionen implementiert sind. Das System, auf dem Sie Ihre Skripts laufen lassen, muss also mindestens Perl 5.003 installiert haben.

Mehr Informationen zu Prototypen finden Sie in der ***perlsub***-Manpage.

Die Funktion *caller*

Eine Funktion, die ich im Kapitel »übergangen« habe, ist `caller`. Die Funktion `caller` gibt Informationen, von wo aus eine Subroutine aufgerufen wurde (das kann bei der Fehlersuche behilflich sein). Für mehr Details sehen Sie in die ***perlfunc***-Manpage.

Zusammenfassung

Eine Subroutine ist eine Möglichkeit, häufig gebrauchten Code zusammenzufassen oder ein langes Skript in kleinere »Portionen« aufzuteilen. Heute haben Sie gelernt, wie Sie Subroutinen deklarieren, definieren, aufrufen und Werte in sie hinein und wieder heraus bekommen. In diesem Zusammenhang haben wir auch über den Gültigkeitsbereich von globalen und lokalen Variablen gesprochen.

Subroutinen definieren Sie mit dem Schlüsselwort `sub`, dem Namen der Subroutine und einem Block, der die

auszuführenden Anweisungen enthält. Innerhalb dieses Blocks können Sie mit `my` lokale Variablen deklarieren, die nur für diese Subroutine sichtbar und verfügbar sind. An die übergebenen Argumente kommen Sie heran, indem Sie auf das lokale Spezialarray `@_` zugreifen, und mit der Funktion `return` geben Sie einen Wert oder eine Wertliste aus der Subroutine zurück. Wenn es relevant ist, in welchem Kontext die Subroutine aufgerufen wurde, ermitteln Sie ihn mit der Funktion `wantarray`.

Sie rufen Subroutinen mit einem optionalen `&`-Präfix, dem Namen der Subroutine und den Argumenten in Klammern auf. Perl gibt alle Argumente in einer linearen, flachen Liste an das Array `@_`. Hashes und verschachtelte Listen werden dabei expandiert.

Fragen und Antworten

Frage:
Was ist der Unterschied zwischen Subroutinen und Funktionen?

Antwort:
Im Grunde gibt es keinen Unterschied, beide bezeichnen vom Konzept her dasselbe. Deswegen kann man die beiden Begriffe zur Unterscheidung zwischen bestimmten Arten von Funktionen/Subroutinen verwenden. Ich zum Beispiel unterscheide in diesem Buch zwischen den in Perl eingebauten und Ihren selbstdefinierten Funktionen.

> Der Unterschied zwischen selbstdefinierten und Perl-eigenen Funktionen ist aber nicht nur ein begrifflicher. Anders als Perl-Funktionen erwarten Subroutinen (zumindest ohne Prototypen) keine bestimmte Anzahl und Art von Argumenten. Auch die Regeln, wann Sie die Klammern um die Argumente weglassen können, sind unterschiedlich. Doch in zukünftigen Perl-Versionen wird das Verhalten von Subroutinen dem der Perl-eigenen Funktionen mehr und mehr ähneln - die Perl-Autoren arbeiten daran.

Frage:
Subroutinenaufrufe mit `&` sehen irgendwie seltsam aus. Kann ich das `&` weglassen?

Antwort:
Ja, das `&`-Präfix ist optional. Ob Sie es verwenden oder nicht, liegt ganz allein bei Ihnen.

Frage:
Ich möchte zwei Arrays an eine Subroutine übergeben und auch zwei Arrays zurückbekommen. Aber die beiden Arrays werden auf dem Hin- und Rückweg in einer einzelnen Liste zusammengeschmissen. Wie halte ich sie auseinander?

Antwort:
Am besten machen Sie das mit Referenzen, und die besprechen wir am Tag 19. Eine andere Möglichkeit wäre, innerhalb der Subroutine einfach zwei globale Arrays zu verändern, anstatt ihr die Daten in einer Liste zu übergeben.

Frage:
Aber globale Variablen sind böse, schlecht und verwerflich.

Antwort:
Na, das kommt darauf an, mit wem Sie sprechen. Manche Probleme lassen sich in Perl durchaus am besten mit globalen Variablen lösen. Wenn Sie vorsichtig mit ihnen umgehen, sie »sauber« deklarieren und die Grenzen ihrer Gültigkeitsbereiche klar ziehen, dann kommen Sie um die Nachteile von globalen Variablen auch herum. Wir befassen uns übermorgen genauer mit Gültigkeitsbereichen.

Frage:
Ich verstehe das nicht, sind Perl-Subroutinen nun *call-by-value* oder *call-by- reference*?

Antwort:
Weil Perl keine formalen Parameter hat, ist das gar nicht so leicht zu beantworten. Prinzipiell ist die Übergabe der Parameterliste an die Subroutine eine Referenzübergabe (call-by-reference): Wenn Sie direkt auf `@_` zugreifen und zum Beispiel eine Skalarvariable verändern, ändert sich ihr Wert auch außerhalb der Subroutine. Beachten Sie

aber, dass Hashes und verschachtelte Arrays expandiert werden - bei Arrays haben Sie dann bestenfalls eine Referenzübergabe der Einzelteile, bei Hashes eine reine Wertübergabe (call-by-value). Am einfachsten ist, alle Elemente von `@_` lokalen Variablen zuzuweisen und mit diesen weiterzuarbeiten. Das entspräche einer Call-by-value-Übergabe aller Argumente - Ihren Originalvariablen könnte nichts mehr passieren.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Nennen Sie zwei Gründe, warum Subroutinen nützlich sind.
2. Wie ruft man eine Subroutine auf?
4. Spielt es eine Rolle, ob eine Subroutine vor ihrem Aufruf definiert wurde? Wo im Skript sollten Subroutinendefinitionen stehen?
5. Welchen Wert gibt eine Subroutine zurück, wenn Sie ihn nicht mit `return` explizit festlegen?
6. Ist der Wert, den `return` aus einer Subroutine zurückgibt, ein Skalar oder eine Liste?
7. In welchen Teilen eines Perl-Skripts sind mit `my` deklarierte Variablen verfügbar?
8. Was passiert, wenn Sie mit `my` eine lokale Variable deklarieren und es bereits eine gleichnamige globale Variable gibt?
9. Was geschieht mit einem Hash, wenn man ihn an eine Subroutine übergibt?
10. Wozu dient `@_`? Wo haben Sie Zugriff auf `@_`?
11. Wie geben Sie Argumenten in einer Perl-Subroutine einen Namen?
12. Was macht `wantarray`? Wozu könnten Sie es gebrauchen?

Übungen

1. Schreiben Sie eine Subroutine, die nichts macht, als ihre Argumente auszugeben, jedes in einer eigenen Zeile mit Zeilennummer.
2. Schreiben Sie eine Subroutine, die einen String entgegennimmt und in umgekehrter Wortreihenfolge zurückgibt (`zum Beispiel` - `Beispiel zum`).
3. Schreiben Sie eine Subroutine, die eine Liste von Zahlen entgegennimmt und die entsprechenden Quadratzahlen zurückgibt. Wenn der String nichtnumerische Elemente enthält, löschen Sie diese aus der Rückgabeliste.
4. FEHLERSUCHE: Was ist falsch an diesem Skript?

```
    @gemeinsam = &schnittmenge(@liste1, @liste2);
sub schnittmenge {
   my (@eins, @zwei) = @_;
   my @final = ();
   OUTER: foreach my $el1 (@eins) {
      foreach my $el2 (@zwei) {
         if ($el1 eq $el2) {
             @final = (@final, $el1);
             next OUTER;
         }
      }
   }
   return @final;
}
```

5. FEHLERSUCHE: Und was ist mit diesem hier?

```
    $wie_oft= &suche(@input, $suchwort);
sub suche {
   my (@in, $suchwort) = @_;
   my $count = 0;
   foreach my $str (@in) {
       while ($str =~ /$suchwort/og) {
           $count++;
       }
```

```
    }
    return $count;
}
```

6. Schreiben Sie eine Subroutine, die, wenn in Skalarkontext aufgerufen, eine Zeile von der Tastatur einliest und sie in umgekehrter Zeichenreihenfolge ausgibt. Wenn man sie in Listenkontext aufruft, soll sie mehrere Zeilen in eine Liste lesen und die Reihenfolge der ***Zeilen*** umkehren (die letzte Zeile zuerst und so weiter, die einzelnen Strings selbst sollen bleiben, wie sie sind).
7. Nehmen Sie das Grafik-Extraktionsskript von gestern, und schreiben Sie es diesmal mit Subroutinen.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Subroutinen sind aus mehreren Gründen nützlich:

- Lange und komplexe Skripts werden übersichtlicher, wenn man sie in einzelne Subroutinen aufteilt.
- Ein Skript wird lesbarer, wenn man einzelnen Vorgängen einen Namen gibt - den Subroutinennamen.
- In Subroutinen kann man lokale Variablen deklarieren, die effizienter und leichter zu kontrollieren sind als globale Variablen.
- Wenn man denselben Code mehrmals benötigt, braucht man ihn nicht immer wieder einzutippen, sondern kann ihn in eine Subroutine packen und bei Bedarf aufrufen.
- Kleinere Teile Code, die sich auf eine Aufgabe konzentrieren, sind einfacher zu schreiben und zu debuggen.

2. Man ruft Subroutinen mit einer der folgenden Formen auf:

```
    &subroutine();    # beides: &-Präfix und Klammern
&subroutine(1,2)  # mit Argumenten
subroutine(1,2)   # & ist optional
```

3. Eine Subroutinendefinition hat folgende Form:

```
    sub NAME {
   # Körper der Subroutine
}
```

4. Nein, es spielt keine Rolle, ob eine Subroutine vor oder nach ihrem Aufruf definiert wird. In einigen Sonderfällen (wenn zum Beispiel die Klammern um die Argumente weggelassen werden) muss sie aber vor ihrem Aufruf wenigstens deklariert worden sein.
1. Überall, wo im Skript eine normale Anweisung stehen kann, kann man auch eine Subroutine definieren. Üblich ist allerdings, alle Subroutinendefinitionen an den Anfang oder das Ende des Skripts zu stellen.
5. Ohne ein explizites `return` gibt die Subroutine das Ergebnis der letzten Auswertung im Block zurück.
6. Fangfrage! Ob `return` einen Skalar oder eine Liste zurückgibt, bestimmen Sie.
7. `My`-Variablen sind nur in dem Block, in dem sie deklariert wurden, sichtbar und verfügbar, nirgendwo sonst - nicht einmal in von dort aufgerufenen Subroutinen.
8. Mit `my` deklarierte Variablen verstecken gleichnamige globale Variablen. Erst wenn Perl den Gültigkeitsbereich der `my`-Variablen verläßt, sind ihre globalen »Namensvettern« wieder verfügbar.
9. Alle Argumente an eine Subroutine werden als eine einzige lineare Liste übergeben (und kommen als solche zurück). Wie in jedem Listenkontext werden Arrays und Hashes in einer solchen Liste interpoliert - ein Hash also in seine einzelnen Schlüssel und Werte expandiert.
10. `@_` ist die Parameterliste der aktuellen Subroutine - das lokale Spezialarray, das alle Argumente an diese Subroutine enthält. Normalerweise greift man innerhalb der Subroutine darauf zu, aber man kann `@_` auch außerhalb verwenden (siehe Abschnitt »Prototypen«).
11. Perl kennt keine formalen Parameter. Wenn Sie Ihren Argumenten einen Namen geben möchten, müssen Sie die entsprechenden Elemente von `@_` lokalen Variablen zuweisen (die Sie ja benennen können, wie Sie wollen). Allerdings entspricht das einer ***Call-by-value***-Wertübergabe, Sie arbeiten also nur mit den Werten der Argumente weiter, nicht mit den Originalargumenten außerhalb der Subroutine.
12. Die Funktion `wantarray` liefert ***wahr*** zurück, wenn die Subroutine in Listenkontext aufgerufen wurde. Mit

ihr könnten Sie herausfinden, in welchem Kontext die Subroutine aufgerufen wurde, und je nachdem (ob Listen- oder Skalarkontext) ein entsprechendes Ergebnis zurückgeben.

Lösungen zu den Übungen

1. Hier ist eine Antwort:

```
sub print_argumente {
my $zeile = 1;
foreach my $arg (@_) {
    print "$zeile. $arg\n";
    $zeile++;
}
}
```

2. Hier eine mögliche Antwort:

```
sub string_umkehren {
my @str = split(/\s+/, $_[0]);
return join(" ", reverse @str);
}
```

3. Hier eine Antwort

```
sub quadrate {
my @ergebnis = ();
foreach my $el (@_) {
    if ($el !~ /\D+/) {
        @ergebnis = (@ergebnis, $el**2);
    }
}
return @ergebnis;
}
```

4. Dieses Skript nimmt irrtümlicherweise an, dass die beiden Argumente `liste1` und `liste2` innerhalb der Subroutine noch zwei Listen (`@eins` und `@zwei`) seien. Sind sie aber nicht - all ihre Elemente werden fein säuberlich in ***einer*** Liste aneinandergereiht, an `@_` weitergegeben und `@eins` zugewiesen. Für dieses Beispiel wären (momentan) zwei globale Variablen besser geeignet.
5. Das Problem in diesem Skript liegt in der Reihenfolge der Argumente. Weil die - ich kann es gar nicht oft genug sagen - lineare Parameterliste zuerst alle Elemente von `@input` und dann das `$suchwort` enthält, weisen wir in der Subroutine auch das Suchwort der lokalen Variablen `@input` zu. Sie erinnern sich, wie gefräßig Listenvariablen auf der linken Seite einer Zuweisung sind - sie schlucken alle noch vorhandenen Elemente, und für das lokale `$suchwort` bleibt nichts mehr übrig. Tauschen Sie die Argumente einfach um, und weisen Sie zuerst das Suchwort und dann die Liste zu.
6. Machen Sie's zum Beispiel so:

```
sub rueckwaerts {
my $in = "";
if (wantarray()) {  # Listenkontext
    my @inliste = ();
    while () {
        print 'Ihre Eingaben bitte: ';
        $in = <STDIN>;
        if ($in ne "\n") {
            @inliste = ($in, @inliste); # Reihenfolge
                                        # umkehren
        }
        else { last; }
    }
    return @inliste;
}
else {            # Skalarkontext
    print 'Ihre Eingabe bitte: ';
    chomp($in = <STDIN>);
    return reverse $in;
}
```

```
}
```

7. Was halten Sie hiervon:

```
    #!/usr/bin/perl -w
$/ = "";     # Absatzeingabe-Modus
while (<>) {
    while (/<IMG\s([^>]+)>/ig) {
        &bild_atts_ermitteln($1);
    }
}
sub bild_atts_ermitteln {
    my $roh = $_[0];
    my %atts = ();
    while ($roh =~ /([^\s=]+)\s*=\s*("([^"]+)"|[^\s]+\s*)/ig) {
        if (defined $3) {
            $atts{ uc($1) } = $3;
        } else { $atts{ uc($1)} = $2; }
    }
    if ($roh =~ /ISMAP/i) {
        $atts{'ISMAP'}= "Yes";
    }
    &print_atts(%atts);
}
sub print_atts {
    my %atts = @_;
    print '-' x 15;
    print "\nImage:  $atts{'SRC'}\n";
    foreach my $key ("WIDTH", "HEIGHT",
                     "BORDER", "VSPACE", "HSPACE",
                     "ALIGN", "ALT", "LOWSRC", "ISMAP") {
        if (exists($atts{$key})) {
            $atts{$key} =~ s/[\s]*\n/ /g;
            print "   $key: $atts{$key}\n";
        }
    }
}
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

---

1

Von Funktionen wird erwartet, dass sie einen sinnvollen Wert zurückgeben, von Subroutinen aber nicht. Doch irgend etwas geben Subroutinen in Perl immer zurück - und über den Sinn läßt sich streiten.

2

Denn der `my`-Operator hat eine festere Bindung als das Komma.

3

Selbst wenn eine Subroutine sich selber aufruft, erhält sie mit jedem Aufruf eine neue Kopie der `my`-Variablen.

4

Parameter = Argument.

5

Wir machen sozusagen aus der Referenz- eine Wertübergabe.

6

Auch wenn das Semikolon in gewisser Weise überflüssig ist, weil Listen natürlich leer sein können.

Perl-Skripts debuggen

Auch als guter (!) Programmierer müssen Sie damit rechnen, dass Ihre Programme - und seien sie noch so kurz - Fehler aufweisen. Einige davon sind einfacher Art - Syntaxfehler, Endlosschleifen oder Beanstandungen (Option `-w`). Andere hingegen sind versteckt und liefern unverständliche oder gar keine Ergebnisse.

Ein Weg, herauszufinden, was sich an jeder Stelle in Ihrem Skript abspielt, führt über `print`-Anweisungen. Mit Hilfe von `print`-Anweisungen können Sie die aktuellen Werte der Variablen kontrollieren und feststellen, ob Schleifen und Bedingungen auch ausgeführt werden. Es gibt jedoch noch einen anderen Weg, der besonders für längere Skripts empfehlenswert ist. Perl verfügt über einen Quellcode-Debugger, mit dem Sie die Ausführung eines Skripts schrittweise verfolgen und die verschiedenen Werte der Variablen während der Ausführung des Skripts ausgeben lassen können. Mit dem Debugger können Sie die Fehler in Ihrem Code schneller finden als mit `print`-Anweisungen.

In diesem Kapitel zeige ich Ihnen, wie Sie den Perl-Debugger einsetzen. Im einzelnen sehen Sie anhand eines kurzen Beispiels, wie Sie den Debugger normalerweise verwenden

- lernen Sie, wie Sie den Debugger starten und ausführen,
- verfolgen Sie die Ausführung Ihres Programms,
- gehen Sie Ihr Skript schrittweise durch,
- lernen Sie, Ihren Quelltext auf verschiedene Weisen anzeigen zu lassen,
- drucken Sie die Werte der Variablen aus und
- setzen Sie Haltepunkte, um die Ausführung an bestimmten Punkten zu unterbrechen.

Einsatz des Debuggers: Ein einfaches Beispiel

Die Arbeitsweise des Perl-Debuggers läßt sich wahrscheinlich am besten anhand eines kleinen typischen Beispiels demonstrieren. Dazu gehen wir schrittweise durch die Ausführung eines einfachen Skripts mit Subroutinen, in diesem Falle das Namensskript aus Kapitel 6, »Bedingungen und Schleifen«, das eine Liste von Namen einliest, Sie bittet einen Suchbegriff einzugeben und die Namen zurückgibt, die dem Suchbegriff entsprechen.

Wenn Sie ein Skript bei eingeschaltetem Perl-Debugger ausführen, landen Sie direkt im Debugger und erhalten in etwa folgende Zeilen:

```
% perl -d  statsfunk.pl statsdaten.txt
Loading DB routines from perl5db.pl version 1.01
Emacs support available.
Enter h or 'h h' for help.
main::(statsfunk.pl:3):       &initvars();
  DB<1>
```

Wenn Sie noch nicht wissen sollten, wie Sie den Debugger starten, zerbrechen Sie sich darüber nicht den Kopf. Ich werde gleich nach diesem kleinen Exkurs darauf eingehen.

Die Schlußzeile `DB<1>` ist die Eingabeaufforderung des Debuggers. Hier geben Sie Ihre Befehle ein. Die Zahl `1` besagt, dass dies der erste Befehl ist. Befehle lassen sich durch Eingabe der Befehlsnummer wiederholen.

Die Zeile vor der Eingabeaufforderung gibt die aktuelle Codezeile an, die Perl anschließend ausführt. Der Teil zur Linken bezieht sich auf das aktuelle Paket, den Namen der Datei und die Zeilennummer. Der Teil zur Rechten ist

die eigentliche Codezeile einschließlich der Kommentare zu der Zeile.

Um eine Codezeile auszuführen, können Sie den Befehl `n` oder den Befehl `s` verwenden. Der `s`-Befehl geht etwas mehr in die Tiefe, da damit auch Subroutinen ausgeführt werden. Mit dem `n`-Befehl bleiben Sie bei der schrittweisen Ausführung auf der obersten Ebene Ihres Skripts. Subroutinen werden im Hintergrund abgearbeitet. Beide Befehle lassen sich wiederholen, indem man bei jeder Eingabeaufforderung die Eingabe-taste betätigt:

```
main::(statsfunk.pl:3):       &initvars();
  DB<1> s
main::initvars(statsfunk.pl:8):           $input = "";
  DB<1>
main::initvars(statsfunk.pl:9):           @nums = ();
  DB<1>
main::initvars(statsfunk.pl:10):          %freq = ();
  DB<1>
```

In diesem Beispiel fällt auf, dass, sobald die Ausführung des Skripts in die `&initvars()`-Subroutine verzweigt (erfolgt in unserem Beispiel gleich in der ersten Zeile), die Informationen auf der linken Seite der Ausgabe den Namen dieser Subroutine widerspiegeln. So wissen Sie jederzeit, wo im Skript Sie sich befinden. Falls Sie aber die Übersicht verloren haben oder anhand der einzelnen Zeile Ihre Position im Skript nicht festlegen können, ist es möglich mit Hilfe des Befehls `l` (steht für ***list***) weitere Codezeilen anzeigen zu lassen:

```
DB<1> l
10==>       %freq = ();      # Hash: Zahl-Haeufigkeit
11:         $maxfreq = 0;    # hoechste Haeufigkeit
12:         $count = 0;      # Anzahl Zahlen
13:         $sum = 0;        # Summe
14:         $avg = 0;        # Durchschnitt
15:         $med = 0;        # Median
16:         @keys = ();      # temp keys
17:         $totalspace = 0; # gesamte Breite des Histogramms
18      }
19
```

Mit `l` lassen sich die auf die aktuelle Zeile folgenden zehn Zeilen am Bildschirm ausgeben. Durch Eingabe eines Minuszeichens können Sie auch die zehn Zeilen vor der aktuellen Zeile sichtbar machen. Durch mehrmaliges Eingeben von `l` und - können Sie sich den Quellcode auflisten lassen, die folgenden Zeilen und zurückliegende. Beachten Sie jedoch, dass Sie damit lediglich die Zeilen anzeigen lassen und Ihre Codezeile im Kontext sehen können - der Code selbst wird damit nicht ausgeführt.

Beim Durchschreiten des Codes können Sie die Werte aller Variablen - Skalare, Arrays, Hashes und so weiter - mit dem Befehl `x` ausgeben lassen. Im nächsten Beispiel hat die Subroutine `&getinput()` gerade eine Zeile aus der Eingabedatei eingelesen, diese Zeile in der Variablen `$input` abgelegt und anschließend das Zeichen für »Neue Zeile« aus dieser Zeile entfernt. Bei `DB<5>` wird der Wert von `$index` und die aktuellen Werte von `@nums` ausgegeben (die Array-Indizes stehen links und die eigentlichen Elemente rechts). Beachten Sie, dass die Codezeile (hier Zeile 23) angezeigt wird, bevor sie ausgeführt wird, so dass der Wert von `$input` noch nicht in `@nums` abgelegt ist.

```
main::getinput(statsfunk.pl:21):          while (defined ($input = <>)) {
  DB<5> s
main::getinput(statsfunk.pl:22):              chomp ($input);
  DB<5>
main::getinput(statsfunk.pl:23):              $nums[$count] = $input;
  DB<5> x input
0  5
DB<6> x @nums
0  1
1  4
2  3
3  4
DB<7>
```

Mit dem `x`-Befehl können Sie auch, wie im folgenden Beispiel zu sehen, Perl-Code ausführen, um die Anzahl der Elemente in `@raw` zu ermitteln oder das erste Element anzuzeigen:

```
DB<3> x scalar(@nums)
0  4
  DB<4> x $raw[0]
0  1
```

Wenn Sie Ihr Skript mit `s` oder `n` debuggen, wird Ihnen jede Zeile bis ins kleinste Detail angezeigt - manchmal sogar ausführlicher, als Sie es benötigen. Innerhalb einer Subroutine können Sie mit dem Befehl `r` das schrittweise Debuggen der Subroutine aufheben, den Rest der Subroutine ausführen und dann dorthin zurückkehren, von wo die Subroutine aufgerufen wurde. Dies kann bei verschachtelten Subroutinen wiederum eine Subroutine sein.

Sie können mit der Eingabe von `c` den Debug-Prozess jederzeit abbrechen. Anschließend führt Perl den Rest des Skripts ohne weitere Unterbrechungen aus (es sei denn, Sie haben explizit Haltepunkte in Ihrem Skript aufgenommen, um zum Beispiel eine Eingabe einzulesen.)

Zusätzlich zu der Möglichkeit, die Ausführung Ihres Skripts zeilenweise zu verfolgen, können Sie die Ausführung mit Haltepunkten steuern. Unter einem Haltepunkt versteht man eine Markierung in einer Codezeile oder zu Beginn einer Subroutine. Mit dem Befehl `c` wird das Skript bis zu einem gesetzten Haltepunkt ausgeführt und dort unterbrochen. Am Haltepunkt können Sie dann mit `n` oder `s` den Code zeilenweise durchgehen, mit `x` die Variablen ausgeben oder mit `c` den nächsten Haltepunkt ansteuern.

Betrachten wir beispielsweise das Skript ***statsmenue.p1***, mit dem wir gestern das ***stats***-Skript in mehrere Subroutinen unterteilt haben. Die Subroutine `&countsum()` gibt die Anzahl und die Summe der Daten aus. Um die Summe zu berechnen, ruft sie die Subroutine `&sumnums()`auf. Sie setzen einen Haltepunkt bei der Subroutine `&sumnums()`, indem Sie den Befehl `b` gefolgt von dem Namen der Subroutine eingeben. Dann führen Sie mit `c` das Skript bis zu diesem Haltepunkt aus.

```
# perl -d statsmenue.pl statsdaten.txt
Loading DB routines from perl5db.pl version 1.01
Emacs support available.
Enter h or 'h h' for help.
main::(statsmenue.pl:3): @nums = (); # array of numbers;
  DB<1> b sumnums
  DB<2> c
Was moechten Sie ausgeben? (Beenden mit Q):
1. eine Liste der Zahlen
2. die Anzahl und Summe der Zahlen
3. die kleinste und die groesste Zahl
4. den Durchschnitts- und den Medianwert
5. ein Diagramm, wie oft jede Zahl vorkommt

Ihre Auswahl --> 2
Anzahl der Zahlen: 70
main::sumnums (statsmenue.pl:72):     my $sum = 0;
  DB<2>
```

Und wie, meinen Sie, sollen Sie sich die Namen all Ihrer Subroutinen merken? Kein Problem. Mit der Eingabe von `S` können Sie alle verfügbaren Subroutinen ausgeben lassen. Insbesondere `S main` wird die von Ihnen definierten Subroutinen anzeigen (wundern Sie sich jedoch nicht, wenn auch hier einige zusätzliche Perl-Routinen auftauchen). Betrachten Sie das folgende Beispiel:

```
DB<7> S main
main::BEGIN
main::countsum
main::getinput
main::maxmin
main::meanmed
main::printdata
main::printhist
main::printmenu
main::sumnums
  DB<8>
```

Der `main`-Teil heißt so in Anlehnung an das `main`-Paket, in dem sich all Ihre Variablen und Subroutinen standardmäßig befinden. Auf Pakete werden wir im folgenden Kapitel 13 noch näher eingehen.

Wenn Sie einen Haltepunkt in einer bestimmten Zeile setzen wollen, können Sie mit `l` das Skript listenförmig ausgeben, um die Zeile zu ermitteln, und dann diese Zeilennummer zusammen mit dem Befehl `b` verwenden:

```
DB<3> l
43:         print "5. ein Diagramm, wie oft jede Zahl vorkommt.\n";
44:         while () {
45:             print "\nIhre Auswahl   --> ";
46:             chomp($in = <STDIN>);
47:             if ($in =~ /^\d$/ || $in =~ /^q$/i) {
48:                 return $in;
49              } else {
50:                 print "Ungueltige Eingabe. 1-5 oder Q, bitte.\n";
51              }
DB<3>
```

Beachtenswert ist auch, dass der Debugger die Ausführung des Skripts zur Laufzeit verfolgen kann. Mit den Befehlen `n` und `s` können Sie zwar jede Anweisung einzeln ausführen lassen, aber manchmal möchte man die Zeilen nicht einzeln durchgehen, sondern sich trotzdem die einzelnen Anweisungen bei der Ausführung ausgeben lassen. Der Befehl `t` schaltet die Verfolgung ein und aus. Wir verfolgen hier die Subroutine `&printdata()`, wobei ein Haltepunkt an die Spitze der `foreach`-Schleife gesetzt wurde:

```
DB<1> b 59
  DB<2> t
    Trace = on
  DB<2> c
Was moechten Sie ausgeben? (Beenden mit Q):
1. eine Liste der Zahlen
2. die Anzahl und Summe der Zahlen
3. die kleinste und die groesste Zahl
4. den Durchschnitts- und den Medianwert
5. ein Diagramm, wie oft jede Zahl vorkommt

Ihre Auswahl --> 1
die Zahlen:
1 main::printdata(statsmenue.pl:59):         foreach $num (@nums) {
DB<2> c
main::printdata(statsmenue.pl:60):               print "$num ";
1 main::printdata(statsmenue.pl:61):             if ($i == 10) {
main::printdata(statsmenue.pl:64):               } else { $i++; }
main::printdata(statsmenue.pl:59):           foreach $num (@nums) {
  DB<2>
```

An diesem Beispiel möchte ich Ihnen zeigen, dass bei der Verfolgung nicht nur die Ausgabe des Skripts angezeigt (man beachte die `1` am Zeilenanfang in der Mitte der Ausgabe direkt nach dem Befehl `c`), sondern auch jede Skriptzeile ausgegeben wird. Würden wir hier erneut `c` eingeben, würde die `foreach`-Schleife erneut durchlaufen, und wir erhielten die gleiche Ausgabe.

Verlassen wird der Debugger mit `q` (einfach nur `q`).

```
DB<18> q
%
```

Soweit - so gut. Nach diesem Exkurs sollten Sie eine ungefähre Vorstellung davon haben, wie der Debugger funktioniert. Der Rest dieses Kapitels ist den Details der spezielleren Befehle gewidmet.

Den Debugger starten und ausführen

Der für Perl mitgelieferte Debugger wird von der Befehlszeile aus mit der Option `-d` gestartet. Während Sie Ihre Perl-Skript unter Unix oder Windows NT lediglich durch Angabe des Skriptnamens aufgerufen haben, so müssen Sie zum Debuggen Perl explizit aufrufen, gefolgt von dem Skriptnamen und den Argumenten. Wenn Sie also

normalerweise ein Skript wie folgt aufrufen:

```
% meinSkript.pl namen.txt
```

lautet der Aufruf für den Debugger wie folgt:

```
% perl -d meinSkript.pl namen.txt
```

Wenn Sie der Meinung sind, dass Sie den Debugger recht häufig für ein bestimmtes ziemlich schwieriges Skript einsetzen werden, können Sie alternativ die Option `-d` auch in der ***shebang***-Zeile des Skripts aufnehmen.

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

Vergessen Sie nicht, die Option wieder zu entfernen, sobald Sie mit dem Debuggen fertig sind.

Um den Debugger in MacPerl einzuschalten, wählen Sie im **Script**-Menü den Befehl **Perl Debugger**, speichern Sie Ihr Skript, und führen Sie es dann wie gewohnt aus. Wenn Sie vorhaben, Ihr Skript als Droplet zu verwenden (das Dateien als Eingabe akzeptiert), dann vergessen Sie nicht das Skript als Droplet zu speichern.

Beachten Sie, dass vor Aufruf Ihres Debuggers das Skript frei von Syntaxfehlern und Warnungen sein muss. Sie müssen diese fatalen Fehler beseitigen, bevor Sie Ihr Skript debuggen können. Da diese Aufgabe Ihnen ohnehin nicht erspart bliebe, sollte die Bürde nicht zu schwer sein.

Während der Debugger läuft, können Sie jederzeit mit dem Befehl `h` Hilfe anfordern. Damit wird eine Liste der möglichen Befehle ausgegeben. Rollt Ihnen der Bildschirm zu schnell, steht Ihnen der Befehl `|h` zur Verfügung (pausiert nach jeder Seite). Hilfe gibt es auch zu jedem einzelnen Befehl (`h` plus Argument). Alle Befehle, die dem Argument entsprechen, werden dann wie folgt ausgegeben:

```
DB<3> h c
c [line|sub]    Continue; optionally inserts a one-time-only breakpoint
                at the specified position.
command         Execute as a perl statement in current package.
  DB<4>
```

Jeder Debugger-Befehl hat eine bestimmte Nummer (im obigen Beispiel hatte der Befehl `h c` die Nummer 3). Mit einem Ausrufezeichen und der Befehlsnummer können Sie jederzeit auf einen der vorangegangenen Befehle Bezug nehmen.

```
DB<4> !3
```

Einen Überblick über die letzten paar Befehle erhalten Sie mit `H` und einer Zahl, der ein Minuszeichen vorangestellt wurde:

```
DB<13> H -3
13: H-3
12: b sumnums
11: x @nums
  DB<14>
```

Sie verlassen den Debugger mit `q`. Ist die Ausführung Ihres Perl-Skripts abgeschlossen, können Sie mit `R` die Ausführung erneut starten. Beachten Sie, dass `R` je nach Umgebung und Befehlszeilenargumenten, die für das Skript verwendet wurden, nicht funktioniert.

Die Ausführung verfolgen

Verfolgen bedeutet, dass Sie jede Zeile Ihres Skripts angezeigt bekommen, während sie von Perl ausgeführt wird. Wird eine Zeile dabei mehrmals hintereinander ausgeführt, wie zum Beispiel in einer Schleife, wird jeder Durchgang von Perl angezeigt. Bei sehr komplexen Skripten ist die Ausgabe deshalb oft umfangreicher als

notwendig. Aber mit Haltepunkten an bestimmten Positionen mag es zeitweise recht nützlich sein, die genaue Reihenfolge der Ausführung des Skripts zu verfolgen.

Um zwischen den Verfolgungsmodi in Ihrem Perl-Skript hin- und herzuschalten, steht Ihnen `t` zur Verfügung. Ist die Verfolgung ausgeschaltet, wird sie mit `t` eingeschaltet und umgekehrt. Die folgende Ausgabe zeigt zum Beispiel das Ergebnis einer ausgeführten Schleife mit ausgeschalteter und mit eingeschalteter Verfolgung (der Haltepunkt befindet sich in Zeile 59, einer `foreach`-Schleife).

```
main::printdata(statsmenue.pl:59):         foreach $num (@nums) {
  DB<4> c
2 main::printdata(statsmenue.pl:59):           foreach $num (@nums) {
  DB<4> t
Trace = on
  DB<4> c
main::printdata(statsmenue.pl:60):                 print "$num ";
2 main::printdata(statsmenue.pl:61):               if ($i == 10) {
main::printdata(statsmenue.pl:64):                 } else { $i++; }
main::printdata(statsmenue.pl:59):             foreach $num (@nums) {
```

Bei eingeschalteter Verfolgung können Sie die Ausführung Ihres Skripts mitverfolgen. Eine Stapelverfolgung zeigt Ihnen, wo Sie bereits gewesen sind - im Falle von verschachtelten Subroutinen zeigt sie Ihnen die Subroutine, die die von Ihnen zur Zeit ausgeführte Subroutine aufgerufen hat, und alle, die eventuell noch darüber angeordnet sind - bis zur obersten Ebene Ihres Skripts. Die Stapelverfolgung wird mit dem Befehl `T` aktiviert:

```
DB<4> T
@ = main::sumnums() called from file 'statsmenue.pl' line 68
$ = main::countsum() called from file 'statsmenue.pl' line 13
  DB<4>
```

Die Zeichen zu Beginn dieser Zeilen geben den Kontext an, in dem die Subroutine aufgerufen wurde. So zeigt die erste Zeile dieser Stapelverfolgung, dass die Subroutine `&sumnums()` (die aktuelle Routine) in einem Listenkontext (angezeigt durch das `@`- Zeichen am Zeilenanfang) von der Routine `&countsum()` aus aufgerufen wurde. Die Routine `&countsum()` wiederum wurde von dem `main`-Teil des Skripts in einem skalaren Kontext aufgerufen (angezeigt durch das `$` am Zeilenanfang).

Schritt für Schritt durch das Skript

Um den Code Ihres Skripts schrittweise zu durchwandern, müssen Sie einen der Befehle `s` oder `n` verwenden. Mit der Eingabetaste wiederholen Sie Ihre vorige Eingabe (`s` oder `n`). Bei jedem Schritt zeigt Perl die Codezeile an, die es als nächstes ausführen wird (nicht die aktuell ausgeführte Codezeile):

```
DB<1> s
main::getinput(statsmenue.pl:29):                  $nums[$count] = $_;
  DB<1>
```

Der Unterschied zwischen `s` und `n` liegt darin, dass der Befehl `s` bei der Überwachung in die Ausführung untergeordneter Subroutinen verzweigt, während `n` diese Subroutinen zwar ausführt, jedoch bei der schrittweisen Überwachung auf der gleichen Ebene bleibt.

Um die Ausführung einer Subroutine in Einzelschritten abzubrechen, den Rest der aktuellen Subroutine auszuführen und zu der Anweisung zurückzukehren, die die Subroutine ursprünglich aufgerufen hat, verwenden Sie den Befehl `r`.

Um die Einzelschrittausführung für den gesamten Code aufzuheben, verwenden Sie den Befehl `c`.

Den Quelltext auflisten

Sie können den aktuell ausgeführten Quelltext mit Zeilennummern ausgeben lassen, um sich den Kontext zu der aktuellen Codezeile anzeigen zu lassen oder um nach einer speziellen Zeile zu suchen, bei der ein Haltepunkt gesetzt werden soll.

Die nächsten zehn Zeilen des Codes erscheinen bei Eingabe des Befehls `l`:

```
DB<15> l
79==>           $sum += $num;
80          }
81:         return $sum;
82      }
83
84  # kleinste und groesste Zahl ausgeben
85  sub maxmin {
86:         print "Kleinste Zahl: $nums[0]\n";
87:         print "Groesste Zahl: $nums[$#nums]\n\n";
88      }
  DB<15>
```

Weitere Aufrufe von `l` werden die jeweils folgenden zehn Zeilen anzeigen. Sie können `l` aber auch zusammen mit einer Zeilennummer verwenden, um genau diese Zeile anzuzeigen, oder mit einem Zeilenbereich (zum Beispiel `1-4`), um speziell diese Zeilen auszugeben, oder mit dem Namen einer Subroutine, um die ersten zehn Zeilen dieser Subroutine aufzulisten.

Um in der Anzeige einige Zeilen zurückzuwandern, gibt es den Befehl `-`. Wie schon bei dem Befehl `l` können Sie mit mehrmaliger Eingabe von `-` auch weiter zurückwandern.

Mit dem Befehl `w` erhalten Sie einen Ausschnitt um die aktuelle Zeile (oder die spezifizierte Zeile, falls angegeben): Es werden einige Zeilen davor und einige Zeilen danach eingeblendet. Ein Pfeil (`==>`) markiert die aktuelle Zeilenposition:

```
DB<3> w
24      # Input aus Datei lesen und dann sortieren
25      sub getinput {
26:         my $count = 0;
27==>       while (<>) {
28:             chomp;
29:             $nums[$count] = $_;
30:             $count++;
31:          }
32:         @nums = sort { $a <=> $b } @nums;
33:      }
```

Mit einem Mustervergleich suchen Sie nach einer bestimmten Zeile im Quelltext. `/ daten/` wird zum Beispiel nach dem ersten Vorkommen des Wortes `daten` suchen. Mit `?muster?` durchsuchen Sie die Datei rückwärts.

Einen letzten nützlichen Befehl zum Auflisten möchte ich Ihnen noch vorstellen: `S`. Damit werden Ihnen alle Subroutinen, die im Skript auftauchen, angezeigt. Die meisten dieser Subroutinen sind Perl- oder Debugger-spezifisch. `S` zusammen mit einem Paketnamen (zum Beispiel `main`) wird jedoch nur die Subroutinen in dem Paket ausgeben:

```
DB<20> S main
main::BEGIN
main::get_muster
main::namen_lesen
main::suche_muster
```

Morgen werde ich Ihnen mehr zu den Paketen erzählen.

Variablen ausgeben

In dem Quelltext zu Ihrem Skript herumzublättern ist nicht nur schön und praktisch, es hilft Ihnen auch, herauszufinden, was in Perl abläuft, wenn Ihr Skript ausgeführt wird. Die folgenden Befehle dienen dazu, die Werte der Variablen auszugeben:

`X` gibt alle Variablen in dem aktuellen Paket aus. Da viele dieser Variablen Perl- spezifisch sind (einschließlich spezieller Variablen wie `@_`, `$_` und `$1`), kann die Liste ziemlich lang werden. `X` zusammen mit dem Namen einer

Variablen gibt alle Variablen aus, die mit dem Suchnamen übereinstimmen. Beachten Sie, dass Sie nur den Namen selbst angeben und nicht die Präfixe wie `$`, `@` oder `%`. `X foo` gibt die Werte aller Variablen aus, deren Namen `foo` lautet (`$foo`, `@foo` und `%foo`).

Der Befehl `V` wird verwendet wie `X`. Zusätzlich kann jedoch ein optionaler Paketname angegeben werden, um die Variablen dieses Pakets auszugeben. Diese Eigenschaft ist allerdings erst von Belang, wenn Sie mit Paketen arbeiten. Der Vollständigkeit halber möchte ich Sie hier jedoch erwähnen.

Der Befehl `X` birgt das Problem, dass lokale Variablen innerhalb von Subroutinen offensichtlich nicht erkannt werden. Um die Werte von lokalen Variablen auszugeben oder kleinere Perl-Fragmente auszuführen, an deren Ergebnis Sie interessiert sind, verwenden Sie den `x`-Befehl:

```
DB<3> x $input
0  'Dante Alighieri'
```

Wenn Sie Arrays oder Hashes ausgeben, wird der Inhalt des Arrays oder des Hash angezeigt. Die Ausgabe von `X` und `V` ist etwas einfacher zu lesen als die von `x`, besonders im Falle von Hashes. Und so sieht ein Hash bei der Verwendung von `X` aus:

```
DB<4> X %names
%names = (
   'Adams' => 'Douglas'
   'Alexander' => 'Lloyd'
   'Alighieri' => 'Dante'
   'Asimov' => 'Isaac'
   'Barker' => 'Clive'
   'Bradbury' => 'Ray'
   'Bronte' => 'Emily'
)
```

Haltepunkte setzen

Das Setzen von Haltepunkten innerhalb eines Skripts erlaubt es Ihnen, das Skript normal auszuführen und an bestimmten Punkten (in der Regel dort, wo alles anfängt, falsch zu laufen) anzuhalten. Sie können beliebig viele Haltepunkte in Ihrem Skript setzen und dann mit den Befehlen zur Einzelschrittausführung oder zur Variablenausgabe das Problem einkreisen. Nehmen Sie mit `c` die Ausführung des Codes nach dem Haltepunkt wieder auf.

Sie setzen einen Haltepunkt mit dem Befehl `b`. Wird der Befehl ergänzt um den Namen einer Subroutine, befindet sich der Haltepunkt an der ersten Anweisung innerhalb dieser Subroutine. Der Befehl `b` zusammen mit einer Zeilennummer setzt den Haltepunkt in genau dieser Zeile. Ohne Argumente wird mit `b` ein Haltepunkt in der aktuellen Zeile gesetzt. Im Quellcode-Listing erscheinen Haltepunkte als ein kleingeschriebenes `b` (in unserem Beispiel in Zeile 33):

```
DB<19> w 33
30      }
31
32      sub suche_muster {
33:b        my $key = $_[0];
34:         my $gefunden = 0;
35:         foreach $ln (sort keys %namen) {
36==>           if ($ln =~ /$key/o || $namen{$ln} =~ /$key/o) {
37:                 print "$ln, $namen{$ln}\n";
38:                 $gefunden = 1;
39              }
```

Der Befehl `L` dient dazu, alle Haltepunkte, die Sie gesetzt haben, auf einmal auszugeben:

```
DB<22> L
namessub.pl:
 9:         my @raw = ();  # raw Liste von Namen
   break if (1)
 33:        my $key = $_[0];
   break if (1)
```

Gelöscht wird ein Haltepunkt, indem die Zeilennummer oder der Subroutinenname zusammen mit dem Befehl `d` verwendet wird. Mit `D` werden alle gesetzten Haltepunkte auf einmal gelöscht.

```
DB<22> d 9
DB<23> L
namessub.pl:
 33:        my $key = $_[0];
   break if (1)
```

Weitere Befehle

Bis jetzt habe ich Ihnen vornehmlich die Befehle vorgestellt, die Ihnen den Einstieg in die Arbeit mit dem Debugger erleichtern und die Sie wahrscheinlich am häufigsten anwenden. Neben den hier vorgestellten Befehlen können Sie mit Perl aber auch noch bedingte Haltepunkte setzen, den Wert der Variablen ändern, Aktionen in bestimmten Zeilen ausführen und mehr oder weniger komplette Perl-Skripts eingeben und deren Ausführung interaktiv verfolgen. Sobald Sie mit dem Perl-Debugger etwas vertrauter sind, werden Sie den Befehl `h` garantiert öfter verwenden und, wenn nötig, die ***perldebug***-Manpage zu Rate ziehen.

Zu guter Letzt

Der Debugger wird Ihnen helfen, Probleme in Ihrem Code zu lokalisieren. Die fleißige Verwendung der Option `-w` ermöglicht es, viele Probleme zu verhindern, bevor das Skript überhaupt gestartet wird (oder bevor man den Debugger überhaupt konsultieren muss, um herauszufinden, was falsch läuft). Machen Sie es sich also zur Gewohnheit, wann immer möglich, `-w` zu verwenden.

Vertiefung

In diesem Kapitel habe ich Ihnen die wichtigsten Debugger-Befehle vorgestellt. Wenn Sie so richtig in die Arbeit mit dem Debugger einsteigen, sollten Sie in der ***perldebug-*** Manpage oder der Online-Hilfe nachschauen, welche weiteren Befehle und Optionen Ihnen zur Verfügung stehen.

Der Einsatz verschiedener Debugger

In Perl können Sie das Verhalten des Debuggers anpassen. Sie können aber auch ein ganz anderes Debugger-System verwenden. Der Schalter `-d` zusammen mit einem Doppelpunkt und dem Namen des Moduls bindet das Modul als neuen Debugger ein. So können Sie zum Beispiel mit dem Modul `Devel::DProf` von CPAN Laufzeitprofile Ihrer Perl-Skripten erstellen (testen, wie lange Ihre Subroutinen dauern, um herauszufinden, wo Ihr Code noch effizienter sein könnte). Sobald Sie das Modul installiert haben, können Sie es wie folgt aufrufen:

```
% perl -d:DProf dasSkript.pl
```

Die Datei `perl5db.pl` enthält den Code für den Debugger. Sie können diese Datei kopieren und Ihren Ansprüchen gemäß modifizieren. Weitere Informationen finden Sie in der ***perldebug-***Manpage oder den Dokumentationen zu dem `DProf`-Modul.

Perl interaktiv

Sie können den Debugger nutzen, um Perl in einer Art interaktivem Modus auszuführen. So können Sie Befehle testen und deren Ausgabe direkt verfolgen. Sie benötigen dafür nicht einmal ein richtiges Skript. Und so sieht ein einfacher Befehl aus, mit dem Sie den Debugger ohne ein Skript zum Debuggen laden:

```
% perl -d -e1
```

> *Um genau zu sein, Sie haben Perl damit ein Skript zur Ausführung übergeben. Dieses Skript ist jedoch nur ein Zeichen lang: `1`. Mit der Option `-e` werden Perl-Skripts direkt von der Befehlszeile ausgeführt. Dieses Thema schneiden wir noch einmal am Ende des Buches in Kapitel 20 an.*

Häufige Fallen und Fragen

In den letzten zwei Wochen habe ich mich bemüht, Sie auf häufige Fehler hinzuweisen, die fast allen Perl-Anfängern (und auch erfahrenen Programmierern) immer wieder unterlaufen. In der ***perltraps-***Manpage finden Sie ebenfalls eine Liste der häufigsten Fallen und darüber hinaus noch vieles mehr. Schon ein kurzes Studieren dieser Seite wird Ihnen viele interessante Hinweise liefern, wenn es darum geht, Probleme bei schwierigem Code zu lösen.

Die Perl-Dokumentation umfaßt außerdem einen umfangreichen Satz an FAQ- Dateien (häufig gestellte Fragen). Bevor Sie sich vor Verzweiflung über ein bestimmtes Problem die Haare raufen, sehen Sie lieber einmal in den FAQs nach. Beginnen Sie dabei mit der ***perlfaq-***Manpage, und lassen Sie sich von dort aus leiten.

Zusammenfassung

Heute haben wir zwar nicht besonders viel über Perl selbst erfahren, dafür aber um so mehr über den dazugehörigen Befehlszeilen-Debugger. Sie wissen jetzt, wie Sie den Debugger aufrufen und starten, wie Sie jede Zeile Ihres Skripts einzeln ausführen, den Quelltext ausgeben, Haltepunkte setzen, die Ausführung verfolgen und Informationen über verschiedene Teile Ihres Skripts während seiner Ausführung einholen.

Mit dem Debugger gibt es nur wenige Probleme, die Ihnen verborgen bleiben. Es lassen sich Probleme damit schneller feststellen als mit `print`-Anweisungen.

Fragen und Antworten

Frage:
Ich verwende emacs. Gibt es eine Möglichkeit, den Perl-Debugger zusammen mit emacs zu verwenden?

Antwort:
Aber klar. In der Datei cperl-mode.el finden Sie massenweise Material darüber, wie man Perl in den emacs einbindet. Diese Datei gehört zum Standardumfang von Perl und befindet sich in dem Verzeichnis emacs.

Frage:
Ich bin eher an Debugger mit grafischer Oberfläche gewöhnt. All diese Befehle auf Befehlszeilenebene nerven mich. Gibt es für Perl auch visuelle Debugger?

Antwort:
Wenn Sie den ActiveState-Port von Perl für Windows verwenden, steht Ihnen über ActiveState ein phantastischer visueller Perl-Debugger zur Verfügung. Weitere Informationen finden Sie unter `http://www.activestate.com`.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Wozu dient der Debugger? Warum ist er nützlicher als zum Beispiel `-w` oder `print`?
2. Wie starten Sie den Perl-Debugger?
3. Nennen Sie drei Möglichkeiten, um Codefragmente im Debugger auszugeben.
4. Wie können Sie die Werte von Variablen im Debugger ausgeben?
5. Worin unterscheidet sich die Verfolgung von der Einzelschrittausführung des Codes?
6. Inwiefern unterscheiden sich die Befehle `X` und `V` zum Ausgeben der Variablen?

Übungen

Geben Sie folgendes Skript ein:

```
#!/usr/bin/perl -w
my @foo = (2,5,3,7,4,3,4,3,2,3,9);
foreach $wert (0..10) {
    &multiplizieren($wert, $foo[$wert]);
}
sub multiplizieren {
    my ($num, $val) = @_;
    print "$num mal $val ist gleich ", $num * $val, "\n";
}
```

Lassen Sie den Debugger darüber laufen und führen Sie folgende Debugger- Operationen durch:

1. Schalten Sie mit `t` die Verfolgung ein, und lassen Sie mit `c` die Ergebnisse anzeigen.
2. Geben Sie `R` ein, um das Skript erneut auszuführen. Gehen Sie mit `n` schrittweise durch das Skript. Wenn Sie sich innerhalb der `foreach`-Schleife befinden, lassen Sie die Werte für `$wert` mehrmals ausgeben.
3. Geben Sie `R` ein, um das Skript erneut auszuführen. Gehen Sie mit `s` schrittweise durch das Skript. Geben Sie innerhalb der Subroutine `&multiplizieren()` die Werte von `$num` und `$val` aus. Kehren Sie mit `r` wieder aus der Subroutine `&multiplizieren()` zurück.
4. Geben Sie `R` ein, um das Skript erneut auszuführen. Setzen Sie mit dem Befehl `b` in der Subroutine `&multiplizieren()`einen Haltepunkt. Lassen Sie den Haltepunkt mit `L` anzeigen. Führen Sie mit `c` das Programm bis zum Haltepunkt aus. Löschen Sie mit `d` den Haltepunkt wieder.
5. FEHLERSUCHE: Gehen Sie mit dem Debugger in folgendem Skript auf Fehlersuche:

```
    #!/usr/bin/perl -w
@foo = (2,5,3,7,4,3,4,3,2,9);
while ($i < $#foo) {
    &multiplizieren($i, $foo[$i]);
}
sub multiplizieren {
    my ($num, $val) = @_;
    print "$num mal $val ist gleich ", $num * $val, "\n";
}
```

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Der Befehl `-w` dient dazu, Syntaxfehler oder schlechten Programmierstil zu finden (oder zu vermeiden). Der Perl-Debugger unterstützt Sie bei allen anderen Problemen: wenn Arrays nicht eingerichtet oder ausgegeben werden, Werte nicht übereinstimmen oder die Ausführung des Skripts nicht der erstrebten Logik entspricht. Die gleiche Hilfestellung - allerdings in reduzierter Form - erhalten Sie mit der `print`-Anweisung, die jedoch mit mehr Aufwand verbunden ist und oft größere Schwierigkeiten hat, das Problem zu orten. Mit dem Debugger können Sie auch die Werte von Variablen ändern und beliebige Codefragmente ausführen, während das Skript gerade in Ausführung ist - all dies ist mit einer einfachen `print`-Anweisung nicht möglich.
2. Um den Perl-Debugger zu starten, geben Sie in der Befehlszeile `perl` ein, gefolgt von der Option `-d`, dem Namen Ihres Skripts und etwaiger Skriptargumente. Wenn Sie mit MacPerl arbeiten, müssen Sie den Perl-Debugger über das Menü **Script** aufrufen.
3. Es gibt verschiedene Möglichkeiten, Code im Debugger aufzulisten:

- `l` allein zeigt die nachfolgenden Codezeilen an
- `l` zusammen mit einer Zeilennummer zeigt genau diese Zeile an
- `l` zusammen mit einem Zeilenbereich zeigt genau diesen Bereich an
- `-` bringt eine Reihe von voranstehenden Codezeilen zur Anzeige
- `w` zeigt einige Zeilen vor und einige Zeilen nach der aktuellen Zeile an

4. Die Werte von globalen Variablen und von Paket-Variablen geben Sie mit dem Befehl `X` gefolgt vom

Variablennamen (ohne die Zeichen `$`, `@` oder `%`) aus. Andere Variablen und die Ergebnisse von Perl-Ausdrücken (wie zum Beispiel `$hash{'key'}`) geben Sie mit dem Befehl `x` aus.

5. Wenn Sie beim Debuggen die Verfolgung einschalten, wird jede Codezeile bei der Ausführung angezeigt - unabhängig davon, ob die Ausführung in Einzelschritten erfolgt oder nicht. Bei der Einzelschrittausführung wird jede Zeile direkt vor der Ausführung angezeigt. Wenn Sie bei der Einzelschrittausführung den Befehl `n` verwenden, überspringen Sie die Subroutinen. Diese werden zwar ausgeführt, aber ihr Inhalt wird nicht angezeigt.
6. Der Befehl `X` gibt die Variablen für das aktuelle Paket (`main`) aus. Der Befehl `V` gibt Variablen in einem beliebigen gegebenen Paket aus (wenn Sie später mit Paketen arbeiten, wird Ihnen dieser Befehl noch sehr dienlich sein).
7. Ein Haltepunkt ist eine Markierung an einer beliebigen Stelle in Ihrem Perl-Code. Wenn Sie den Code in Ihrem Debugger ausführen, läuft Perl, bis es auf einen Haltepunkt stößt. Dort wird die Ausführung angehalten. Von hier aus können Sie den Code in Einzelschritten durchwandern, Werte von Variablen ausgeben oder bis zum nächsten Haltepunkt die Ausführung fortsetzen.

Lösungen zu den Übungen

1. 1- Hierzu gibt es keine Antworten. Die Punkte symbolisieren die Schritte, die im Debugger durchgeführt werden sollen.
5. Es gibt zwei Fehler im Skript:

- `$i` wurde nicht initialisiert, so dass die Subroutine `&multiplizieren()` Initialisierungsfehler auslöst, wenn sie versucht, den Wert von `$num` auszugeben.
- `$i` wird nicht inkrementiert. Das bedeutet, dass `i` immer den Wert `0` behält und in einer Endlosschleife hängenbleibt.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Gültigkeitsbereiche , Module und das Importieren von Code

In der heutigen Lektion kommen wir auf die Themen Raum und Zeit zu sprechen - eine Ergänzung unserer früheren Diskussionen von Wahrheit und Gier. Mit Raum beziehe ich mich auf die sogenannten ***Namensbereiche*** von Variablen: Sie erfahren, wie Variablennamen über lokale und globale Gültigkeitsbereiche hinweg verwaltet werden und wie man mit Paketen globale Variablen programmübergreifend verwalten kann. Verbunden mit dem Thema Raum ist die Fähigkeit, Code von anderen Modulen entweder zur Kompilier- oder zur Laufzeit in Ihr Skript zu importieren - womit wir beim Thema ***Zeit*** wären. Heute erfahren wir,

- warum globale Variablen Probleme verursachen können und wie sich diese Variablen kontrollieren lassen,
- mehr über lokale Variablen: mehr als nur `my` und Subroutinen,
- wie man externen Code importiert und verwendet und
- wie man Module der Standardbibliothek und von CPAN verwendet.

Globale Variablen und Pakete

Der Gültigkeitsbereich einer Variablen bezieht sich, allgemein ausgedrückt, auf die Verfügbarkeit und die Existenz einer Variablen in Ihrem Skript. Von einem ***globalen Gültigkeitsbereich*** spricht man, wenn eine Variable in allen Teile eines Skripts verfügbar ist und so lange existiert, wie das Skript ausgeführt wird. Von einem ***lokalen Gültigkeitsbereich*** spricht man hingegen, wenn eine Variable nur begrenzt gültig ist und, je nachdem welcher Teil Ihres Perl-Skripts gerade ausgeführt wird, existiert oder nicht.

Wir beginnen dieses Kapitel mit einem Überblick über globale Variablen und den globalen Gültigkeitsbereich. Anschließend widmen wir uns dem lokalen Gültigkeitsbereich.

Globale Variablen und ihre Probleme

Das ganze Buch hindurch haben wir in den meisten Beispielen globale Variablen und deren globalen Gültigkeitsbereich verwendet - mit Ausnahme von gelegentlichen lokalen Variablen in Subroutinen. Und dafür gibt es auch einen guten Grund: Globale Variablen lassen sich leicht erstellen und verwenden. Jede Variable, die nicht explizit mit `my` (oder, wie Sie bald lernen werden, mit `local`) deklariert wurde, ist automatisch eine globale Variable. Unabhängig vom Kontext, in dem Sie verwendet wurde, steht sie zu jedem Zeitpunkt in Ihrem Skript zur Verfügung.

So lassen sich schnell einfache Skripts schreiben. Doch im gleichen Zuge, in dem Ihre Skripts an Umfang zunehmen, wird die Verwendung von globalen Variablen immer problematischer. Es gibt mehr Variablen, die überwacht werden müssen, und mehr Variablen, die Platz belegen. Globale Variablen, die auf unerfindliche Weise mitten in einem Skript auftauchen, sind teilweise schwer zu debuggen - und es stellt sich die Frage, wann diese Variable aktualisiert wird und von welchem Teil des Skripts? Wissen Sie überhaupt noch, wozu diese globale Variable eingesetzt wurde?

Bei der Erstellung größerer Skripts unter Verwendung von globalen Variablen besteht darüber hinaus die nicht unerhebliche Gefahr, dass Sie zufällig einen Variablennamen vergeben, der bereits irgendwo in Ihrem Skript verwendet wurde. Dieses Problem erschwert nicht nur das Debuggen Ihres Skripts, es wirkt sich auch nachteilig aus, wenn Sie Ihre Skripts in den Code von anderen Programmentwicklern integrieren oder wiederverwendbare Perl-Bibliotheken erzeugen wollen. Damit wird das Risiko gleichlautender Variablennamen in mehreren Codeteilen zu einem sehr realen und nur schwer zu lösenden Problem.

Die beste Möglichkeit, Namenskonflikte durch globale Variablen zu umgehen, besteht darin, auf globale Variablen zu verzichten. Organisieren Sie Ihre Skripts in Subroutinen, und deklarieren Sie Ihre Variablen als lokal zu diesen Subroutinen. Daten, die von mehreren Subroutinen benötigt werden, können mit Hilfe von Argumenten von

Subroutine an Subroutine weitergereicht werden. Viele Software- Entwickler sind der Meinung, dass alle Programme - wie klein und spezialisiert sie auch sein mögen - nach diesem Prinzip geschrieben werden sollten, denn das Vermeiden von globalen Variablen entspräche professionellem Software-Design.

In der Realität hingegen benutzt jeder hin und wieder globale Variablen. Besonders in Situationen, in denen jeder Teil des Skripts auf die gleichen gespeicherten Daten einer Liste oder einer anderen Struktur zugreifen muss. Damit kommen wir zu einer weiteren Möglichkeit, globale Variablen zu organisieren und zu verwalten: den Paketen.

Was ist ein Paket?

Ein ***Paket*** ist eine Möglichkeit, Ihre globalen Variablen so zusammenzufassen, dass sie im eigentlichen Sinne nicht länger global sind. Mit anderen Worten, jedes Paket definiert seinen eigenen Namensbereich. Mit Paketen können Sie steuern, welche globalen Variablen anderen Paketen verfügbar sein sollen und welche nicht. Damit lassen sich die Probleme mit gleichlautenden Variablennamen über verschiedene Codeteilen hinweg vermeiden.

Die Chancen stehen nicht schlecht, dass Sie Ihre eigenen Pakete nur entwickeln müssen, wenn Sie Perl-Module, Bibliotheken oder Klassen in objektorientierter Perl- Programmierung erstellen - alles Themen, die weit über den Rahmen dieses Buches hinausgehen. Doch auch, wenn Sie keine eigenen Pakete entwickeln, sind Pakete überall präsent, wo Perl-Skripts geschrieben oder ausgeführt werden, auch wenn Sie es nicht merken. Mit minimalen Kenntnissen der Funktionsweise von Paketen im Hinterkopf wird es Ihnen nicht nur leichterfallen, zu verstehen, wie Perl Variablen und Namensbereiche in Ihren Skripts handhabt, sondern auch wie Code aus Modulen importiert wird. Außerdem wird Ihnen dieses Wissen zugute kommen, wenn Sie soweit sind, dass Sie Ihren Code in Bibliotheken oder Module umwandeln wollen. Machen Sie sich so früh wie möglich mit den zugrundeliegenden Regeln vertraut, dann haben Sie es später um so leichter.

Wie Pakete und Variablen zusammenarbeiten

Der Gedanke, der den Paketen zugrunde liegt, besteht darin, dass jede Zeile in Perl in der Umgebung des aktuellen Pakets kompiliert wird - wobei es sich bei dem Paket um das Standardpaket oder ein selbst definiertes Paket handeln kann. Jedes Paket kommt mit einem Satz an Variablennamen (auch ***Symboltabelle*** genannt), der bestimmt, ob eine Variable für Perl verfügbar ist und wie der aktuelle Wert der Variablen lautet. Die Symboltabelle enthält alle Namen, die Sie möglicherweise in Ihrem Skript verwenden - Skalare, Arrays, Hashes und Subroutinen.

Wenn Sie in Ihrem Skript auf einen Variablennamen Bezug nehmen (zum Beispiel `$x`), wird Perl versuchen, diese Variable in der Symboltabelle des aktuellen Pakets zu finden. Wenn Sie Perl anweisen, das Paket zu wechseln, wird `$x` in dem neuen Paket gesucht. Sie können Variablen aber auch über ihre kompletten Paketnamen ansprechen. Damit teilen Sie Perl mit, in welcher Symboltabelle sich die Variable befindet und wie ihr Wert lautet. Paketnamen bestehen aus dem Namen des Pakets, zwei Doppelpunkten und dem Namen der Variablen. Das Sonderzeichen, das anzeigt, ob die Variable ein Skalar, eine Liste, ein Hash etc. ist, steht noch vor dem Paket.

So würden Sie sich zum Beispiel mit `$main::x` auf eine Skalarvariable `$x` beziehen, die in dem Paket `main` gespeichert ist. `$x` ohne weitere Angaben würde die Skalarvariable `$x` in dem aktuellen Paket bezeichnen. Beide Variablen tragen den gleichen Namen, aber da sie in verschiedenen Paketen existieren, haben deren Werte nichts miteinander zu tun (und existieren in anderen Paketen überhaupt nicht).

Das von Ihnen bisher verwendete Standardpaket lautet `main`, wasIhnen vielleicht bisher verborgen geblieben ist. Wenn Sie globale Variablen in Skripts erzeugen und verwenden, die nicht explizit ein eigenes Paket definieren, werden diese Variablen tatsächlich in dem Paket `main` erzeugt (vielleicht ist Ihnen dies bereits in den Fehlermeldungen aufgefallen - in einigen heißt es `Name main::foo used only once...`). Die ganze Zeit, wenn ich also von globalen Variablen gesprochen habe, war ich nicht ganz korrekt: Globale Variablen sind streng genommen Variablen des Pakets `main`.

Beachten Sie, dass die Verteilung von globalen Variablen auf Pakete deren Verwaltung nicht unbedingt vereinfacht, wenn Sie sehr viele davon verwenden. Hundert globale Variablen in `main` sind genauso schwierig zu verwalten wie hundert globale Variablen in einem neuen Paket namens `meinPaket`. Die Verwendung lokaler Variablen und das Übergeben von Daten zwischen einzelnen Subroutinen bleibt auch weiterhin die beste Empfehlung für Ihre eigene kleine Perl-Welt.

Für die Erzeugung eines neuen Pakets oder zum Wechseln zwischen Paketen gibt es die Funktion `package`:

```
package meinpaket;    #  definiert oder wechselt zu einem Paket, das nicht
                      # main lautet
```

Der Gültigkeitsbereich von Paketdefinitionen entspricht in etwa dem von lokalen Variablen: Paketdefinitionen innerhalb einer Subroutine oder eines Blocks kompilieren den ganzen Code innerhalb dieser Subroutine oder dieses Blocks als Teil des neuen Pakets und kehren dann zu dem umschließenden Paket zurück. Durch den Aufruf von `package` zu Beginn eines Skripts definieren Sie ein neues Paket für den gesamten Block.

Wie ich bereits vorher erwähnt habe, definieren Sie eigene Pakete vornehmlich dann, wenn Sie Code für eigene Bibliotheken oder Module aufsetzen. Einige Punkte in Bezug auf Pakete sollten Sie sich merken:

- Jeder Variablenname und -wert wird für das aktuelle Paket in einer Symboltabelle gespeichert.
- Sie können auf einen Variablennamen Bezug nehmen, indem Sie entweder nur den Variablennamen (für das aktuelle Paket) oder den Variablennamen mit dem vorangestellten Paketnamen angeben. Damit legen Sie fest, welche Symboltabelle Perl nach dem Wert der Variablen überprüft.
- Das Standard-Paket lautet `main`.

Globale Variablen, die keinem Paket angehören

Ein Möglichkeit, globale Variablen zu erzeugen, die keine Namenskonflikte provozieren, besteht darin, Pakete zu erzeugen. Es gibt jedoch noch einen anderen sehr geläufigen Trick, mit dem man globale Variablen erzeugen kann: Man deklariert die globalen Variablen als lokal zu ihrem eigenen Skript.

Wenn Sie Ihre globalen Variablen mit dem Modifizierer `my` deklarieren, wie es bei lokalen Variablen innerhalb von Subroutinen der Fall ist, dann gehören diese globalen Variablen keinem Paket an, nicht einmal `main`. Dennoch sind sie in Ihrem Skript immer noch global und für alle Teile des Skripts verfügbar (einschließlich der Subroutinen). Es treten jedoch keine Namenskonflikte mit anderen Variablen aus Paketen auf, auch nicht für die Variablen aus dem Paket `main`. Globale Variablen, die mit `my` deklariert wurden, haben auch leistungsmäßig einen Vorteil, denn Perl muss nicht bei jeder Referenzierung dieser Variablen auf die dazugehörige Symboltabelle zugreifen.

Aufgrund der Vorteile von globalen Variablen, die außerhalb von Paketen deklariert wurden, wird in der Perl-Programmierung empfohlen, möglichst in allen, außer vielleicht den einfachsten Perl-Skripts, diese Art von Variablen zu verwenden. Dazu müssen Sie in Ihren eigenen Skripts lediglich bei der Deklaration der globalen Variablen (genau wie bei den lokalen Variablen) den Modifizierer `my` mit angeben:

```
my %namen = (); # globales Hash für Namen
```

Perl verfügt außerdem über einen besonderen Mechanismus, der dafür sorgt, dass Sie all Ihre Variablen ordnungsgemäß verwenden - seien es lokale Variablen innerhalb von Subroutinen oder globale Variablen, die nicht in einem Paket definiert wurden. Nehmen Sie als oberste Zeile Ihres Skripts »`use strict`;« mit auf, um diesen Mechanismus einzuschalten:

```
#!/usr/bin/perl -w
use strict;
my $x = '';  # OK
@foo = ();   # bricht Ausführung des Skripts ab
# Rest Ihres Skripts
```

Wenn Sie `use strict` verwenden, wird Perl bei der Ausführung Ihres Skripts eine Fehlermeldung ausgeben, wenn es auf nicht zuzuordnende globale Variablen trifft, und das Programm abbrechen. Tatsächlich wird `use strict` bei allen Variablen, die nicht mit `my` deklariert, ohne Paketnamen referenziert oder von irgendwoher importiert wurden, eine Fehlermeldung ausgeben. Betrachten Sie `use strict` als einen noch strengeren Auslöser von Variablenwarnungen als `-w`.

Es gibt jedoch einen seltsamen Nebeneffekt des Befehls `use strict`: Er moniert auch die Platzhaltervariable in einer `foreach`-Schleife, so zum Beispiel `$key` in folgendem Beispiel:

```
foreach $key (keys %namen) {
  ...
}
```

Der Grund dafür liegt darin, dass die implizit lokale `foreach`-Variable im Grunde genommen eine globale Variable ist, die vorgibt, lokal zu sein. Sie ist zwar nur in der `foreach`-Schleife verfügbar und verhält sich somit wie eine lokale Variable, aber intern wird sie anders deklariert (die zwei Möglichkeiten, lokale Variablen zu deklarieren, beschreibe ich weiter hinten in diesem Kapitel im Abschnitt »Lokale Variablen mit `my` und `local`«.) Um Fehlermeldungen von `use strict` zu vermeiden, müssen Sie ein »`my`« vor den Variablennamen setzen:

```
foreach my $key (keys %namen) {
  ...
}
```

Im weiteren Verlauf dieses Buches wird in allen Beispielen der Befehl `use strict` verwendet, und globale Variablen werden als `my`-Variablen deklariert.

Lokaler Gültigkeitsbereich und Variablen

Eine lokale Variable steht, wie Sie bereits in Kapitel 11, »Subroutinen erstellen und verwenden«, erfahren haben, nur einem bestimmten Teil des Skripts zur Verfügung. Nachdem die Ausführung dieses Teils des Skripts abgeschlossen ist, hört die Variable auf zu existieren. Darüber hinaus sind lokale Variablen nur beschränkt für andere Teile des Skripts verfügbar. So können sie privat zu dem Gültigkeitsbereich sein, in dem sie definiert wurden, oder nur für die Teile des Skripts zur Verfügung stehen, die zeitgleich mit dem Teil des Skripts ausgeführt werden, in dem die Variable definiert wurde.

Lokale Variablen und lokaler Gültigkeitsbereich

In Kapitel 11 haben wir mit `my` definierte lokale Variablen innerhalb von Subroutinen betrachtet. Eine Subroutine definiert einen lokalen Gültigkeitsbereich. Die lokale Variable steht dann dem ganzen Code, der innerhalb dieser Subroutine definiert wurde, zur Verfügung.

Eine Subroutine ist jedoch nicht der einzige Ort, in dem ein lokaler Gültigkeitsbereich erzeugt werden kann. Jeder Block, der von geschweiften Klammern eingeschlossen ist, definiert einen lokalen Gültigkeitsbereich, und jede Variable, die in diesem Block definiert wurde, verliert nach diesem Block ihren Gültigkeitsbereich. Damit ist es möglich, lokale Variablen innerhalb von Schleifen oder Bedingungen und sogar innerhalb von freien Blöcken zu definieren, wenn es dort Codeteile gibt, die von einem neuen lokalen Gültigkeitsbereich profitieren.

Die folgenden Codefragmente definieren alle einen lokalen Gültigkeitsbereich. Jede Deklaration der Variablen `$s` erfolgt lokal zum einschließenden Block, und jedes `$x` unterscheidet sich von allen anderen Versionen von `$x`. Beachten Sie vor allem die Verwendung der lokalen Variablen in der `foreach`-Schleife. Dort ist die Variable lokal zur gesamten Schleife und nicht zu jeder Iteration. `$x` wird wie erwartet fünfmal inkrementiert:

```
if ($foo) {      # Bedingung
    my $x = 0;
    print "X in der Bedingung: $x\n";
    $x++;
}
foreach (1..5) {  # Schleife
    my $x += $_;
    print "X in der Schleife: $x\n";
}
{                # freier Block
    my $x = 0;
    print "X im freien Block: $x\n";
    $x++;
}
```

Es gibt noch eine weitere Regel im Zusammenhang mit Variablen und Gültigkeitsbereich, die Sie sich merken sollten: Lokale Variablen, die den gleichen Namen wie globale Variablen tragen, verbergen die Werte der gleichlautenden globalen Variable. Den ganzen lokalen Gültigkeitsbereich hindurch gilt der Wert der lokalen Variablen, und mit dem Verlassen des lokalen Gültigkeitsbereichs treten die ursprüngliche globale Variable und ihr Wert wieder in Kraft.

Diese Eigenheit ist einerseits sehr bequem, kann andererseits aber auch sehr verwirrend sein. Im allgemeinen wird

empfohlen, gleiche Namen für lokale und globale Variablen zu vermeiden, es sei denn, es bestehen plausible Gründe dafür.

Bei Bedarf können Sie jedoch durch Angabe des kompletten Paketnamens jederzeit auf die globale Variable Bezug nehmen, auch innerhalb eines lokalen Gültigkeitsbereichs. Voraussetzung ist allerdings, dass Sie weder `use strict` verwendet noch Ihre globalen Variablen mit `my` deklariert haben:

```
$foo = 0;  # global
{
   my $foo = 1;        # lokal
   print "$foo\n";     # gibt 1 aus
   print "$main::foo\n";  # gibt 0 aus
}
print "$foo\n";        # gibt 0 aus
```

Lokale Variablen mit my und local

Lokale Variablen können nicht nur mit `my`, sondern auch mit `local` definiert werden. Der Modifizierer `local` wird auf die gleiche Weise wie der Modifizierer `my` verwendet und kann eine oder mehrere Variablen einschließen:

```
local ($x, $y);
```

Da stellt sich die Frage nach dem Unterschied. Der auffälligste Unterschied zwischen diesen beiden Arten von lokalen Variablen ist der, dass der Gültigkeitsbereich für lokale Variablen, die mit `local` definiert wurden, durch die Ausführung des Skripts festgelegt wird und nicht durch die Anordnung des Codes. Eine `my`-Variable ist nur bis zum nächsten umschließenden Block oder zur nächsten Subroutinendefinition gültig. Wenn Sie von Ihrer Subroutine aus eine neue Subroutine aufrufen, hat die zweite Subroutine keinen Zugriff auf die Variablen der ersten Subroutine. Lokale Variablen, die mit `local` deklariert wurden, sind innerhalb des aktuellen Blocks oder der aktuellen Subroutine sowie für alle darin enthaltenen Subroutinen gültig.

> *Technisch betrachtet haben `my`-Variablen einen lexikalischen und `local`-Variablen einen dynamischen Gültigkeitsbereich. Diese Begriffe müssen Sie nicht lernen, es sei denn Sie sind ein Informatiker oder wollen anderen Informatikern imponieren.*

Ein Beispiel zu den unterschiedlichen Gültigkeitsbereichen von `my`- und `local`- Variablen finden Sie im Abschnitt »Vertiefung« - für den Fall, dass Sie die Unterschiede zwischen `local` und `my` genauer untersuchen wollen. Am besten fahren Sie jedoch damit, Ihre lokalen Variablen mit `my` und nicht mit `local` zu deklarieren, da `my`-Variablen sich enger an die Definition eines lokalen Gültigkeitsbereichs, wie er in anderen Programmiersprachen verwendet wird, anlehnen und leichter zu verwenden und zu verwalten sind.

Perl-Module verwenden

Erfolgreich ein Perl-Skript zu schreiben, besteht zur einen Hälfte darin, bewußt Code aufzusetzen, und zur anderen Hälfte darin, bewußt Code nicht aufzusetzen. Genauer gesagt, sollten Sie wissen, wann Sie von den vorgegebenen Perl-Funktionen Gebrauch machen oder die Bibliotheken und Module anderer Programmierer nutzen sollten, um sich die Programmierung leichter zu machen.

Wenn Sie in Perl eine Aufgabe zu bewältigen haben, die ziemlich komplex scheint, aber unter Umständen bereits von anderen Programmierern zu früheren Zeiten in Angriff genommen wurde, stehen die Chancen recht gut, dass Ihnen bereits jemand die Arbeit abgenommen hat. Und in guter Perl-Manier könnte dieser Jemand seinen Code verpackt in einem Modul oder einer Bibliothek zum Herunterladen im Internet zur Verfügung gestellt haben. Handelt es sich um eine häufig zu erledigende Aufgabe von allgemeinem Interessse, so könnte das passende Modul sogar Teil der ausgelieferten Standardversion von Perl sein. Die meiste Zeit werden Sie wahrscheinlich damit verbringen, sich diese Bibliotheken zunutze zu machen, indem Sie sie in Ihre Perl-Skripts importieren und einige Codezeilen hinzufügen, um so den importierten Code Ihrer speziellen Situation anzupassen. Und das war's dann auch schon.

In etlichen der folgenden Kapitel dieses Buches werden wir einer Reihe von Modulen begegnen, die Ihnen als Teil der Standardversion von Perl, als Teil der Version für Ihre spezifische Plattform oder als herunterladbare Datei des Comprehensive Perl Archive Network (CPAN) zur Verfügung stehen. In diesem Abschnitt lernen Sie deshalb die Grundlagen zur Nutzung dieser Module kennen: Sie erfahren, was ein Modul ist, wie man es importiert und wie man es in eigenen Skripts verwendet.

Zur Terminologie

Zuerst möchte ich einige Begriffe klären. Ich habe mit den Begriffen ***Funktion***, ***Bibliothek***, ***Modul*** und ***Paket*** um mich geworfen, und deshalb ist es notwendig, genau zu klären, was damit jeweils gemeint ist.

Eine vordefinierte ***Funktion*** ist, wie ich bereits erwähnt habe, eine Funktion, die mit Perl ausgeliefert wird und Ihnen zum Einbauen in Ihre Skripts zur Verfügung steht. Sie brauchen keine speziellen Schritte auszuführen, um eine vordefinierte Funktion aufzurufen.

Eine Perl-***Bibliothek*** ist eine Sammlung von Perl-Code, die in anderen Skripts wiederverwertet werden kann. Dies ist die Beschreibung von früheren Perl- Bibliotheken, die mittels des Operators `require` in andere Perl-Skripts importiert werden. In jüngster Zeit wird der Begriff »Bibliothek« immer häufiger gleichgesetzt mit einem Perl-Modul, was dazu führt, dass die bisherigen Bibliotheken und ihr Operator `require` veraltet sind. Das Importieren von Code mit `require` werde ich in einem der hinteren Abschnitte dieses Kapitels noch eingehender beschreiben.

Ein Perl-***Modul*** ist eine Sammlung von wiederverwendbarem Perl-Code. Perl-Module definieren ihre eigenen Pakete, wobei jedes Paket seinen eigenen Satz an Variablen definiert. Sie nutzen die Module, indem Sie sie mit dem `use`-Operator in Ihre Skripts importieren. Anschließend können Sie (normalerweise) die Subroutinen (und manchmal auch die Variablen) in diesem Modul wie alle anderen beliebigen Subroutinen (oder Variablen) referenzieren. Manche Module sind objektorientiert programmiert. Als Folge werden sie etwas anders behandelt, die grundlegende Vorgehensweise ist jedoch die gleiche.

Zusätzlich gibt es noch die ***Pragmas***. Dabei handelt es sich um eine besondere Art von Perl-Modulen, die darauf Einfluß nehmen, wie Perl ein Skript kompiliert und ausführt (die meisten normalen Perl-Module haben nur Einfluß auf die eigentliche Ausführung). Ansonsten verhalten sich beide gleich. `use strict` ist ein Beispiel für die Verwendung eines Pragmas. Auf Pragmas werde ich noch weiter hinten in diesem Kapitel eingehen.

Wo findet man passende Module?

Wo aber befinden sich diese Module? Wenn Sie über Perl verfügen, sind Sie bereits im Besitz einer Reihe von Modulen, mit denen Sie experimentieren können, ohne weitere Schritte unternehmen zu müssen. Die Standard-Perl-Bibliothek (standard Perl library) ist eine Sammlung von Modulen, Pragmas und Skripts, die mit der Standardversion von Perl ausgeliefert wird. Verschiedene Versionen von Perl für unterschiedliche Plattformen können unterschiedliche Standardbibliotheken aufweisen - so hat die Windows-Version von Perl einige Module, um auf Windows-spezifische Fähigkeiten zuzugreifen. Diese Module müssen Sie zu ihrer Verwendung lediglich importieren, und es entfällt die Notwendigkeit, sie herunterzuladen oder zu installieren.

Der »offizielle« Satz an Bibliothekmodulen ist in der Hilfsdokumentation ***perlmod*** vollständig beschrieben und umfaßt Module zu folgenden Themen:

- Schnittstellen zu Datenbanken
- Einfache Netzwerke
- Spracherweiterungen, modul- und plattformspezifische Unterstützung zur Software-Entwicklung, dynamisches Laden von Modulen und Funktionen
- Textverarbeitung
- Objektorientierte Programmierung
- Höhere Mathematik
- Programmieren mit Dateien, Verzeichnissen und Befehlszeilenargumenten
- Fehlerbehandlung
- Datum und Uhrzeit
- Lokale (zur Erstellung internationaler Skripts)

In Anhang B, »Überblick über die Perl-Module«, finden Sie darüber hinaus weitere Informationen zu vielen der Module in der Standardbibliothek.

Zur Windows-Version von Perl gehören die ***Standard-Win32***-Module für Windows- Erweiterungen. Zu diesen gehören unter anderem:

- `Win32::Process`: Erzeugung und Einsatz von Windows-Prozessen
- `Win32::OLE`: Für OLE-Automation
- `Win32::Registry`: Zugriff auf die Windows-Registrierdatenbank
- `Win32::Service`: Verwaltung der Windows-NT-Dienste
- `Win32::NetAdmin`: Einrichten von Benutzern und Gruppen im Netz

MacPerl umfaßt `Mac`-Module für den Zugriff auf die Mac-Toolbox, einschließlich AppleEvents, Dialoge, Dateien, Schriftarten, Videos, Internet-Konfigurationen, QuickDraw und Spracherkennung (wow!). Einige der `Mac`- und `Win32`-Module werde ich Ihnen in Kapitel 18, »Perl und das Betriebssystem«, vorstellen.

Zusätzlich zu der Standard-Perl-Bibliothek gibt es noch das ***Comprehensive Perl Archive Network***, auch als CPAN bekannt. CPAN ist eine Sammlung von Perl- Modulen, Skripts, Dokumentationen und anderen Werkzeugen zu Perl. Perl- Programmierer in der ganzen Welt schreiben Module und schicken sie dann an das CPAN. Um die Module von CPAN zu nutzen, müssen Sie sie herunterladen und in Ihre Perl-Version installieren. Manchmal müssen diese Module auch mit einem C- Compiler kompiliert werden. Ich möchte diesen Abschnitt mit den Modulen beginnen, die bereits installiert vorliegen. Zu CPAN werden wir später in diesem Kapitel kommen.

Module importieren

Um Zugriff auf den Code aus einem beliebigen Modul Ihres Skripts zu erhalten, müssen Sie dieses Modul mit dem `use`-Operator und dem Namen des Moduls ***importieren***:

```
use CGI;
use Math::BigInt;
use strict;
```

Der `use`-Operator importiert die Subroutine und die Variablennamen, die von diesem Modul definiert und exportiert wurden, in das aktuelle Paket, so dass Sie sie verwenden können - so als hätten Sie sie selbst definiert (mit anderen Worten, das Modul enthält ein Paket, das eine Symboltabelle definiert; durch das Importieren dieses Moduls wird die Symboltabelle in die aktuelle Symboltabelle Ihres Skripts geladen).

Modulnamen können in vielen Formen auftreten: Ein einzelner Name benennt ein einfaches Modul, zum Beispiel `CGI`, `strict`, `POSIX` oder `Env`. Ein Name, der aus zwei oder mehr Teilen besteht, die durch zwei Doppelpunkte getrennt sind, bezieht sich auf Teile von größeren Modulen (sie beziehen sich, um genau zu sein, auf Pakete, die innerhalb anderer Pakete definiert sind). So bezieht sich zum Beispiel der Name `Math::BigInt` auf den Teil `BigInt` des `Math`-Moduls oder `Win32::Process` auf den Teil `Process` des `Win32`-Moduls. Modulnamen beginnen in der Regel mit einem Großbuchstaben.

Wenn Sie den Code eines Moduls mit dem `use`-Operator in Ihr Skript importieren, sucht Perl in einem besonderen Satz von Verzeichnissen, dem sogenannten `@INC`- Array, nach der Datei dieses Moduls. `@INC` ist eine spezielle Perl-Variable, die alle Verzeichnisse enthält, die in der Perl-Befehlszeile mit der Option `-I` aufgeführt wurden, gefolgt von den Verzeichnissen für die Standard-Perl-Bibliothek ***(/usr/lib/perl5*** und verschiedenen Unterverzeichnissen unter Unix, ***perl\lib*** unter Windows, ***MacPerl:lib*** unter Macintosh) und einem Punkt (`.`), der für das aktuelle Verzeichnis steht. Der endgültige Inhalt von `@INC` variiert von System zu System, und verschiedene Versionen von Perl können unter Umständen standardmäßig verschiedene Werte für `@INC` aufweisen. Auf meinem System, einer Linux-Maschine mit Perl 5.005_02 lautet der Inhalt von `@INC`

```
/usr/lib/perl5/5.00502/i486-linux
/usr/lib/perl5/5.00502
/usr/lib/perl5/site_perl/5.005/i486-linux
/usr/lib/perl5/site_perl/5.005
.
```

Wenn Sie ein Modul importieren wollen, das in einem anderen Verzeichnis abgelegt ist, verwenden Sie das Pragma `lib` zu Beginn Ihres Skripts und geben das Verzeichnis an.

```
use lib '/home/meineDaten/perl/lib/'
```

```
use Meinmodul;
```

Die Dateien der Perl-Module haben den gleichen Namen wie die Module selbst und verwenden die Extension `.pm`. Viele Module enthalten ganz normalen Perl-Code mit ein wenig Extracode, damit sie sich wie Module verhalten. Wenn Sie also neugierig sind und wissen möchten, wie sie funktionieren, sollten Sie sich den Code einmal anschauen. Andere Module hingegen enthalten oder nutzen plattformspezifischen kompilierten Code, der nicht gerade besonders aussagekräftig ist.

Die letztgenannten Module verwenden die sogenannten Perl-Extensionen, manchmal auch XSUBS genannt, mit denen Sie kompilierte C-Bibliotheken in Perl-Module einbinden können. Die Arbeit mit Extensionen geht weit über den Rahmen dieses Buchs hinaus; in Kapitel 20 werde ich Ihnen aber zumindest einige Tips geben, wo Sie weiterführende Informationen finden.

Module verwenden

Mit `use` können Sie also ein Modul importieren. Und dann? Nun, nach dem Import können Sie den Code aus dem Modul nutzen. Wie Sie dabei vorgehen, hängt davon ab, ob das Modul einfacher Art oder objektorientiert ist (in der Dokumentation für das jeweilige Modul läßt sich feststellen, ob das Modul objektorientiert ist oder nicht).

Betrachten wir zum Beispiel das Modul `Carp`, das Teil der Standard-Perl-Bibliothek ist. Das `Carp`-Modul umfaßt die Subroutinen `carp`, `croak` und `confess` zur Erzeugung von Fehlermeldungen - ähnlich den vordefinierten Funktionen `warn` und `die`. Durch das Importieren des `Carp`-Moduls werden auch die drei Subroutinennamen in das aktuelle Paket importiert. Damit erhalten Sie Zugriff auf diese Subroutinen, so als ob es vordefinierte Funktionen wären oder Subroutinen, die Sie selbst definiert hätten:

```
use Carp;
open(OUT, ">ausgabedatei"
          || croak "ausgabedatei kann nicht geöffnet werden\n";
```

Nebenbei erwähnt, entsprechen die Subroutinen `carp` und `croak` auch insofern den Funktionen `warn` und `die`, als sie zur Ausgabe von Fehlermeldungen verwendet werden (und im Falle von `die` und `croak` danach die Ausführung abbrechen). Die `Carp`-Subroutinen sind aber besser für die Verwendung in Modulen geeignet, da sie angeben, wo ein Fehler aufgetreten ist. Für den Fall, dass ein Skript ein Modul importiert, das eine Subroutine mit einem Aufruf von `carp` enthält, wird `carp` das Paket und die Zeilennummer des umschließenden Skripts angeben und nicht die Zeilennummer innerhalb des Moduls selbst (was für das Debuggen nicht besonders sinnvoll wäre). Auf `Carp` gehe ich im Zusammenhang mit CGI-Skripts in Kapitel 16 noch näher ein.

Einige Module sind objektorientiert. In der objektorientierten Programmierung spricht man statt von Funktionen und Subroutinen von »Methoden«, die dann allerdings anders ausgeführt werden als normale Funktionen. Für importierte objektorientierte Module müssn Sie eine besondere Syntax verwenden, um auf den Code zuzugreifen. (Ihr Skript muss allerdings nicht notwendigerweise objektorientiert sein; das sollte Ihnen kein Kopfzerbrechen bereiten. Sie können objektorientiertes Perl mit normalem Perl problemlos mischen.) Im folgenden sehen Sie ein Beispiel für ein CGI-Modul (das wir in Kapitel 16 noch eingehender untersuchen werden):

```
use CGI;
my $x = new CGI;
my $name = $x->param("meinname");
print $x->header();
print $x->start_html("Hallo!");
print "<H2>Hallo $name!\n";
print $x->end_html();
```

Sieht seltsam aus, nicht wahr? Wenn Sie mit der objektorientierten Programmierung vertraut sind, ist Ihnen klar, dass Sie hier ein neues CGI-Objekt erzeugen, die Referenz auf das Objekt in der Variablen `$x` speichern und dann

die Methoden mit der ->- Syntax aufrufen.

Ist Ihnen die objektorientierte Programmierung fremd, scheint alles nur seltsam. Hier eine Erläuterung in Kürze:

- Die Zeile `my $x = new CGI;` erzeugt ein neues CGI-Objekt und speichert eine Referenz darauf in der Variablen `$x`. `$x` ist kein normaler Skalar, wie ein Array oder ein String. In diesem Fall ist es eine Referenz auf ein spezielles CGI-Objekt.
- Um Subroutinen (genauer gesagt Methoden) aufzurufen, die in dem Modul definiert sind, bedienen Sie sich der Variablen, die das Objekt enthält, des `->-` Operators und des Namens der Subroutine. Demzufolge ruft die Zeile `$x- >header()` die Subroutine `header()` auf, die in dem in `$x` gespeicherten Objekt definiert ist.

Verwenden Sie die gleiche Notation für die anderen Subroutinen des Moduls, und Sie werden keine Schwierigkeiten haben. Auf Referenzen und Objektorientierung komme ich noch in Kapitel 19, »Mit Referenzen arbeiten«, zu sprechen.

Symbole manuell importieren

Das Importieren eines Moduls mittels `use` bindet die Variablen und Subroutinennamen ein, die von dem besagten Modul definiert und exportiert wurden. Das Hauptaugenmerk in dem vorigen Satz liegt auf den Wörtern ***und exportiert*** - manche Module exportieren alle ihre Variablen, manche nur einige davon, und andere wiederum exportieren überhaupt keine. Mit `use` erhalten Sie Zugriff auf den ganzen Code des Moduls, jedoch nicht notwendigerweise so, als ob Sie ihn selbst geschrieben hätten. Deshalb ist es manchmal erforderlich, einige Extraschritte zu unternehmen, um auf ganz bestimmte Teile des Moduls Zugriff zu bekommen.

Wenn ein Modul, das Sie verwenden wollen, keine Variablen oder Subroutinennamen exportiert, werden Sie das früh genug herausfinden - spätestens wenn Sie versuchen, diese Namen zu verwenden, überschüttet Sie Perl mit Fehlermeldungen über nicht definierte Elemente. Es gibt jedoch zwei Möglichkeiten, um auf die von Ihnen gewünschten Elemente des Moduls zuzugreifen:

- Sie können die gewünschten Variablen oder Subroutinen referenzieren, indem Sie den vollständigen Paketnamen angeben.
- Sie können diese Symbole (Variablen oder Subroutinennamen) manuell in die `use`-Anweisung importieren.

Bei der ersten Möglichkeit müssen Sie für den Zugriff lediglich den Paketnamen vor die gewünschte Variable oder Subroutine stellen. Dieser Weg ist besonders dann zu empfehlen, wenn Sie in Ihrem Code gleichlautende Variablen oder Subroutinen haben und einen Namenskonflikt vermeiden wollen:

```
# Aufruf von aufsummieren, definiert im Modul Meinmodule
$ergebnis = &Meinmodule::aufsummieren(@werte);
# aendert Wert der Variable $total (definiert in Meinmodule)
$Meinmodule::total = $ergebnis;
```

Dabei gilt es zu beachten, dass ein Paketname, der bereits zwei Doppelpunkte enthält, in voller Länge vor den Variablennamen gesetzt wird:

```
$Text::Wrap::columns = 5;
```

Die zweite Möglichkeit, alle benötigten Symbole zu importieren, ist dann geboten, wenn Sie beabsichtigen, die Subroutinen eines Moduls recht häufig in Ihrem Code aufzurufen. Durch das Importieren erreichen Sie, dass Sie den Paketnamen nicht jedesmal mit angeben müssen. Sie importieren einen Namen von einem Modul in das aktuelle Paket, indem Sie ihn an das Ende des `use`-Befehls anhängen. In der Regel geschieht dies mit Hilfe der Funktion `qw`, mit der Sie Anführungszeichen weglassen und neue Symbole schnell übernehmen können:

```
use MeinModule qw(ersteSub, zweiteSub, dritteSub);
```

Wichtig ist, dass es sich hierbei um Symbolnamen und nicht um Variablennamen handelt. Es gibt keine Sonderzeichen vor den Namen, und alle Variablen in dem Modul, die diesen Namen tragen, werden importiert (das Symbol `foo` importiert zum Beispiel `$foo`, `@foo` und `&foo` und so weiter). Spezielle Variablen lassen sich mit Verwendung des jeweiligen Präfix importieren:

```
use MeinModule qw($zaehlen);
```

Einige Module sind so definiert, dass sie einen Satz an Variablen aufweisen, die standardmäßig importiert werden, und einen Satz, der nur bei Bedarf importiert wird (ein Blick auf den Code wird Ihnen zeigen, dass das Hash `%EXPORT` in der Regel die standardmäßig exportierten Symbole enthält; `%EXPORT_OK` enthält die optionalen Symbole). Der einfachste Weg, beide zu importieren, besteht darin, `use` zweimal aufzurufen: einmal für die standardmäßigen Symbole und noch einmal für alle optionalen Symbole:

```
use Meinmodule;    # importiert alle Standardnamen
use Meinmodule qw(dies, das);   # importiert auch dies und das
```

Import-Tags

Einige der größeren Module ermöglichen es Ihnen aus Effizienzgründen, nur einen Teilbereich ihrer Elemente zu importieren. Diese Module verwenden spezielle ***Import- Tags***. Wenn Sie mit einem Modul arbeiten, das Import-Tags verwendet, so können Sie anhand der Dokumentation des Moduls herausfinden, welche Tags das Modul unterstützt. Sie können aber auch anhand des Hash `%EXPORT_TAGS` im Quelltext erkennen, welche Tags Sie verwenden können (Tags werden vom Modul exportiert und in Ihren Code importiert).

Um einen Teilbereich eines Moduls zu importieren, fügen Sie das Tag an das Ende der `use`-Anweisung an:

```
use CGI qw(:standard);
```

Obwohl Sie das Import-Tag auch direkt in Anführungszeichen anhängen könnten, ist obige Form gebräuchlicher und vereinfacht überdies das Hinzufügen weiterer Tags.

Wie sich ein Modul verhält - ob es Import-Tags verwendet oder Variablen und Subroutinen enthält, die mit explizit importiert werden müssen -, wird vom Modul selbst definiert und ist (hoffentlich) dokumentiert. Vielleicht versuchen Sie einmal, `perldoc` auf dem Modul auszuführen, um alle vom Autor des Moduls bereitgestellten Online-Dokumentationen herauszufiltern, oder Sie gehen die `readme`-Dateien zu dem Modul durch, um sicherzustellen, dass Sie das Modul korrekt verwenden.

Pragmas verwenden

Die Zeile `use strict`, mit der globale Variablen auf das aktuelle Skript beschränkt werden, ist ein Beispiel für eine besondere Art von Modul namens Pragma. Ein ***Pragma*** ist ein Modul, das darauf Einfluß hat, wie Perl sich zur Kompilier- und zur Laufzeit verhält (im Vergleich zu normalen Modulen, die nur zur Laufzeit Code für Perl bereitstellen). Insbesondere das `strict`-Pragma teilt Perl mit, beim Parsen Ihres Codes genau vorzugehen und verschiedene unsichere Konstrukte abzulehnen.

Wenn Sie sich an Kapitel 1, »Eine Einführung in Perl«, erinnern, wo ich gesagt habe, dass es sich bei Perl nicht um eine kompilierte Sprache wie bei C oder Java handelt, werden Sie die Begriffe ***Kompilierzeit*** und ***Laufzeit*** vielleicht etwas irritieren. In den genannten Sprachen bedienen Sie sich eines Compilers, um Ihren Quelltext in Bytecode oder eine ausführbare Datei zu überführen. Anschließend rufen Sie die neue Datei auf, um das Programm auszuführen. Bei Perl ist das Skript bereits eine ausführbare Datei. Es gibt keinen Kompilierschritt dazwischen.

In Wahrheit habe ich in Kapitel 1 etwas geschummelt, denn natürlich kompiliert auch Perl ebenso wie C und Java seinen Quelltext. Aber dann wird sofort das Ergebnis ausgeführt. Es gibt keine ausführbare Ergebnisdatei, die irgendwo gespeichert wird.

Das bedeutet aber, dass es Aufgaben gibt, die Perl zur Kompilierzeit (während das Skript kompiliert wird) erledigt, und Aufgaben, die während der Laufzeit (während das Ergebnis ausgeführt wird) bearbeitet werden. Beim Kompilieren überprüft Perl die Syntax und stellt sicher, dass alles, was für die Ausführung des Skripts benötigt wird, auch verfügbar ist. Zur Laufzeit wird das Skript dann ausgeführt, und es werden die Daten des Anwenders verarbeitet. Mit zunehmender Praxis werden Sie verschiedene Operationen kennenlernen, bei denen der ***Zeitpunkt*** der Ausführung genauso wichtig ist wie die Tatsache, dass sie überhaupt ausgeführt werden.

Jedoch zurück zu den Pragmas. Wie ich bereits gesagt habe, ist ein Pragma ein kurzer importierter Code, der darauf Einfluß nimmt, wie Perl während der Kompilier- und der Laufzeit arbeitet. Im Gegensatz zu den meisten

importierten Codes in Modulen und Bibliotheken, die nur auf das Laufzeitverhalten eines Skripts einwirken, können Pragmas die Gesamtheit Ihres Codes und die Art, wie Perl ihn sieht, ändern.

Perl stellt Ihnen in seiner Standardbibliothek nur wenige Pragmas zur Verfügung (im Gegensatz zu Modulen, von denen es Dutzende gibt). Es ist Usus, Pragmas nur in Kleinbuchstaben zu schreiben, um sie von den Modulen zu unterscheiden. Aktiviert werden Pragmas genauso wie Module mit dem `use`-Operator:

```
#!/usr/bin/perl -w
use strict;
use diagnostics;
```

Jedes Pragma kann in Ihrem Skript ganz oben angegeben werden, damit das ganze Skript davon betroffen ist. Es kann aber auch innerhalb eines Blocks verwendet werden, so dass nur das Verhalten dieses Blocks geändert wird. Am Ende des Blocks geht das Skript wieder zu seinem normalen Verhalten über.

Zu den nützlicheren Perl-Pragmas gehören `strict` und `diagnostics`. Eine etwas vollständigere Liste der verfügbaren Pragmas finden Sie in der ***perlmod***-Manpage im Abschnitt »Pragmatic Modules«.

strict

Das Pragma `strict`, das Sie bereits kennengelernt haben, beschränkt die Verwendung einiger unsicherer Konstrukte in Ihren Skripts. So hält das `strict`- Pragma Ausschau nach fehlplazierten globalen Variablen, reinen Wörtern (Strings ohne Anführungszeichen, zu denen es in der Sprache keine Definitionen gibt) und symbolischen Referenzen (auf die wir in Kapitel 19 noch näher eingehen). Sie können auch nur ausgewählte dieser unsicheren Konstrukte abfangen, indem Sie die Strings `'vars'`, `'subs'` oder `'refs'` nach der Angabe von `use strict` mit aufnehmen:

```
use strict 'vars';
```

Mit dem Befehl `no strict` (und, wenn notwendig, den optionalen Anhängen `'vars'`, `'subs'` und `'refs'`) können Sie die genaue Kontrolle für bestimmte Blöcke ausschalten. Dieser Befehl erstreckt sich nur bis zum Ende des umschließenden Blocks (Subroutine, Bedingung, Schleife oder freier Block). Danach kehrt Perl zu den vorherigen Einstellungen zurück.

diagnostics

Das Pragma `diagnostics` dient dazu, die expliziten Warnungen von Perl einzuschalten. Seine Funktionsweise entspricht in etwa der des `-w`-Schalters. Es kann jedoch dazu verwendet werden, die Diagnose-Meldungen und Warnungen auf spezielle Teile Ihres Skripts (Blöcke) zu beschränken. Sie können `diagnostics` nicht zur Kompilierzeit Ihres Skripts ausschalten (wie das bei `strict` mit Hilfe von `no strict` der Fall ist). Sie können die Laufzeitwarnungen allerdings mit den Direktiven `enable` und `disable` steuern:

```
use diagnostics;
# etwas Code
disable diagnostics;
# Code der normalerweise eine Laufzeit-Warnung ausgibt
enable diagnostics;
# weiter wie gehabt...
```

Das Modul *English*

Das Modul `English` ist deshalb erwähnenswert, weil es, vergleichbar den Pragmas, eine Möglichkeit bietet, darauf Einfluß zu nehmen, wie Perl Ihr Skript interpretiert. Im Gegensatz zu den Pragmas wirkt es jedoch zur Laufzeit und kann hinsichtlich des Gültigkeitsbereichs nicht auf einen Block beschränkt werden. Das `English`-Modul wird verwendet, um die vordefinierten speziellen Variablennamen etwas weniger kryptisch erscheinen zu lassen. Während die wahren Perl-Junkies fröhlich ihre Skripts mit Variablen wie `$_`, `$"`, `$\` und so weiter übersäen, haben wir einfachen Sterblichen meist schon genug Mühe damit, auch nur die wichtigsten speziellen Variablen auseinanderzuhalten. In genau diesen Fällen ist `use English` eine große Hilfe, da es für die kryptischen Variablennamen verschiedene längere Aliase zur Verfügung stellt.

So ist zum Beispiel die Variable `$,` (ein Dollarzeichen und ein Komma) bekannt als das Trennzeichen für das Ausgabefeld, und es wird in `print`-Anweisungen verwendet, um die einzelnen Elemente voneinander zu trennen. Mit dem Befehl `use English` können Sie sich auf die Variable wie gehabt mit `$,` beziehen, aber genauso gut auch mit den Namen `$OUTPUT_FIELD_SEPARATOR` oder `$OFS`. Alle drei lassen sich gleichermaßen gut verwenden.

Eine Liste der speziellen Variablen von Perl und ihrer Namen (und Aliase) finden Sie in der ***perlvar***-Manpage.

Ein Beispiel: Das Modul Text::Wrap

Im folgenden möchte ich Ihnen ein kleines Beispiel präsentieren, in dem ein Modul der Standardbibliothek verwendet wird: das Modul `Text::Wrap`. Dieses Modul wird einen sehr langen String in mehrere Zeilen einer gegebenen Länge aufspalten und für jede Zeile ein optionales Einrückungszeichen aufnehmen.

Dieses spezielle Beispiel formatiert eine Eingabedatei auf eine Zeilenlänge von 80 Zeichen und rückt den Text in E-Mail-Manier ein: An den Anfang jeder Zeile wird das Zeichen > gestellt. Die betreffende Datei wird dabei in mehrere Abschnitte aufgeteilt, die jeweils durch eine Leerzeile getrennt werden. Wenn also die Eingabedatei wie folgt aussieht (ein einziger langer String; die Zeilenenden in dem untenstehenden Beispiel sind keine eigentlichen Zeilenenden bei der Eingabe):

```
The event on which this fiction is founded has been supposed, by Dr. Darwin, and some of the ph
```

wird die Ausgabe folgendermaßen aussehen:

```
> The event on which this fiction is founded has been supposed, by Dr.
> Darwin, and some of the physiological writers of Germany, as not of
> impossible occurrence. I shall not be supposed as according the remotest
> degree of serious faith to such an imagination; yet, in assuming it as
> the basis of a work of fancy, I have not considered myself as merely
> weaving a series of supernatural terrors. The event on which the interest
> of the story depends is exempt from the disadvantages of a mere tale of
> spectres or enchantment. It was recommended by the novelty of the
> situations which it develops; and, however impossible as a physical
> fact, affords a point of view to the imagination for the delineating of
> human passions more comprehensive and commanding than any which the
> ordinary relations of existing events can yield.
```

In Listing 13.1 finden Sie den Code für das Skript, mit dem Sie diese Aufgabe bewältigen.

Listing 13.1: Das Skript umbrechen.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  use Text::Wrap;                 # importiert das Modul
5:  my $indent = "> ";              # Einrückungszeichen
6:
7:  while (<>) {
8:      print wrap($indent, $indent, $_);
9 : }
```

Wie Sie sehen können, ist der Code nicht besonders umfangreich, und es ist wesentlich einfacher, das Modul zu importieren, als die gleiche Prozedur mit rohem Perl-Code zu schreiben. Die wichtigen Teile des Skripts sind:

- Zeile 4: Hier wird das Modul `Text::Wrap` importiert.
- Zeile 5: Diese Zeile definiert das Einrückungszeichen (hier »> «, es können jedoch auch beliebige andere Einrückungszeichen sein).
- Zeile 8: Hier wird die `wrap`-Funktion aufgerufen, die den Text letztendlich umbricht. Die `wrap`-Funktion, die in dem Modul `Text::Wrap` definiert ist, übernimmt drei Argumente: das Zeichen, mit dem die erste Zeile eingerückt wird, die Zeichen, mit denen alle folgenden Zeilen eingerückt werden und den umzubrechenden String. In diesem Falle wollen wir alle Zeilen mit dem gleichen Zeichen einrükken, so dass der Aufruf von `wrap` zweimal `$indent` spezifiziert.

Standardmäßig wird mit der Funktion `Text::Wrap` eine Zeile nach 76 Zeichen umbrochen. Dieser Wert läßt sich mit der Variablen `$columns` ändern, auch wenn diese Variable nicht automatisch vom Modul importiert wird. Sie müssen diese Variable explizit importieren oder den vollständigen Paketnamen angeben, wenn Sie davon Gebrauch machen wollen:

```
use Text::Wrap qw($columns);   # importiert $columns
$columns = 50;                 # setzt die Variable
```

Module von CPAN (Comprehensive Perl Archive Network) verwenden

Wenn die Module der Standard-Perl-Bibliothek nicht ausreichen - und oft ist dies der Fall -, gibt es immer noch das CPAN. Das ***Comprehensive Perl Archive Network*** ist, wie ich bereits erwähnt habe, eine riesige Sammlung von öffentlich verfügbaren Perl- Modulen, die fast alle erdenklichen Bereiche abdecken. Sie benötigen ein Modul für die Handhabung verschlüsselter Daten? CPAN hat es. Sie wollen eine E-Mail senden. In CPAN finden Sie das passende Modul dafür. Sie wollen eine komplette HTML- Datei einlesen und verarbeiten? Kein Problem. Egal, was Sie vorhaben, es ist auf alle Fälle ratsam, erst einmal in CPAN nachzuforschen, ob Ihnen bereits jemand die Arbeit abgenommen hat. Es gibt keinen Grund, das Rad neu zu erfinden, wenn Sie den Code eines anderen verwenden können. Es ist schon von Vorteil, wenn man eine Perl- Gemeinde hat, auf die man sich verlassen kann.

Ein warnender Hinweis

Auf eines möchte ich Sie allerdings aufmerksam machen: CPAN-Module haben zwei Nachteile. Der erste besteht darin, dass Sie Module erst herunterladen, einrichten und installieren müssen, bevor Sie sie verwenden können. Das bedeutet etwas mehr Arbeit als nur das Einfügen einer `use module`-Zeile in Ihr Skript. Bei einigen Modulen ist eventuell eine Kompilierung erforderlich, was wiederum einen funktionierenden C- Compiler auf Ihrem PC bedingt.

Das zweite Problem der Module im CPAN besteht darin, dass die meisten von ihnen für Unix-Perl entwickelt wurden. Wenn Sie Windows oder MacPerl verwenden, ist das von Ihnen benötigte Modul unter Umständen nicht für Ihre Plattform verfügbar. Diese Situation hat sich zum Glück in letzter Zeit etwas gewandelt, und zunehmend werden die Module plattformübergreifend programmiert. Vor allem Windows-Unterstützung wird immer populärer. Es gibt sogar ein spezielles Tool namens PPM zum Installieren und Verwalten von Windows-spezifischen CPAN-Modulen für die ActiveState-Version von Perl für Windows (PPM wird uns später noch, im Abschnitt »CPAN-Module unter Windows installieren«, beschäftigen). Wenn Sie sich nicht sicher sind, ob ein bestimmtes Modul für Ihre Plattform vorhanden ist, müssen Sie in der Dokumentation zu dem Modul nachschauen - und sich darauf einstellen, es eventuell selbst zu portieren.

Wie man Module von CPAN herunterlädt

Die CPAN-Module sind auf der CPAN-Website oder einer gespiegelten Site (Mirror) abgelegt. Wenn Sie bei `http://www.perl.com/CPAN/CPAN.html` starten, können Sie von dort aus feststellen, was CPAN enthält, wie man an die Dateien gelangt und wie man herausfindet, ob ein bestimmtes Modul für Ihre Plattform verfügbar ist. Es gibt auch eine Suchmaschine für Module, so dass Sie ziemlich schnell herausfinden können, ob es zu Ihrem Problem ein Modul gibt. Anhang B enthält ebenfalls eine umfangreiche Liste der Module, die zum Zeitpunkt der Drucklegung dieses Buches zur Verfügung standen (da jedoch ständig neue Module hinzukommen, sollten Sie trotzdem noch die Online-Version konsultieren).

Einige Module sind gebündelt, um die Abhängigkeiten zwischen den verschiedenen Modulen zu reduzieren (es gibt nichts Ärgerlicheres, als ein Skript auszuführen, das ein Modul benötigt, und dann nach dem Herunterladen festzustellen, das dieses Modul ein weiteres Modul erfordert, das dann wiederum ein drittes Modul benötigt und so weiter). Modulbündel beginnen in der Regel mit dem Wort `lib` - zum Beispiel die Modulgruppe `libwww`, die eine ganze Reihe von Modulen zur Handhabung von allem, was das World Wide Web betrifft, enthält. Wenn das von Ihnen benötigte Modul als Teil eines Bündels im CPAN erhältlich ist, sind Sie gut beraten, wenn Sie das ganze Bündel herunterladen und nicht nur das einzelne Modul.

Angenommen Sie haben ein Modul gefunden, das Sie verwenden wollen. Grundsätzlich gehen Sie so vor, dass Sie das Modul herunterladen und mit den Tools für Ihre Plattform dekomprimieren oder dearchivieren (normalerweise `gzip` und `tar` für Unix, `WinZip` für Windows und `Stuffit` für Mac). Unter Unix gehört zu vielen Modulen eine `make`-Datei, die dafür Sorge trägt, das alles an der richtigen Stelle installiert wird (mit dem Befehl `make` starten Sie

den Prozeß). Wenn Sie mit einem Windows- oder Mac-PC arbeiten und es keine besonderen Anweisungen für die Installation der Dateien auf Ihrer Plattform gibt, genügt es unter Umständen, die `.pm`-Moduldateien einfach an die entsprechenden Stellen in der Perl-Hierarchie zu kopieren. In der ***perlmodinstall***-Manpage der Perl- Version 5.005 finden Sie viele Vorschläge speziell zum Dekomprimieren und Installieren der Module. Das Verfahren kann jedoch für jedes Modul variieren. Deshalb sollten Sie sich eng an die Vorgabe der `README`-Dateien halten, um sicherzustellen, dass alles korrekt installiert wird.

Einige Module enthalten Teile, die in C geschrieben wurden. Dafür kann unter Umständen ein C-Compiler auf Ihrer Maschine erforderlich werden. Haben Sie keinen derartigen Compiler zur Verfügung, können Sie Teile des Moduls unter Umständen auch ohne C-Compiler verwenden. Auch hier gilt es, die Dokumentation zu dem Modul zu Rate zu ziehen.

Wenn Sie unter Unix arbeiten und Module mit kompilierten Teilen installieren, sollten Sie sicherstellen, dass der C-Compiler zum Kompilieren der Module der gleiche ist wie der, mit dem Perl ursprünglich kompiliert wurde. Bei Modulen, die nicht in der gleichen Umgebung kompiliert wurden wie Perl, können nur schwer zu lösende Inkompatibilitäten auftreten.

Nachdem Sie die Dearchivierung, Einrichtung, Kompilierung und Installation abgeschlossen haben, sollte eine Reihe von Dateien an den richtigen Positionen in der Perl-Hierarchie installiert worden sein. Vielleicht wollen Sie die verschiedenen Verzeichnisse in Ihrem `@INC`-Array durchgehen, um sicherzustellen, dass es sie gibt.

CPAN-Module unter Windows installieren

Der Perl Package Manager, auch PPM genannt, ist ein Zusatzprogramm, das mit der ActiveState-Version von Perl für Windows ausgeliefert wird. Es erleichtert ganz enorm die Installation und Verwaltung von CPAN-Modulen unter Windows. Bei Verwendung von PPM müssen Sie sich keine Gedanken darum machen, ob ein bestimmtes Modul von Windows unterstützt wird, oder langwierig austüfteln, wie es mit Hilfe irgendwelcher mysteriöser Unix-Tools kompiliert oder installiert wird. Mit PPM können Sie von einem Programm aus einzelne, bereits erstellte Module installieren, aktualisieren und entfernen.

> *Ein **Paket** im Sinne des Perl Package Manager ist eine Sammlung von einem oder mehreren Modulen und unterstützenden Dateien. Verwechseln Sie es nicht mit den Paketen für die Perl-Namensbereiche.*

Um PPM nutzen zu können, müssen Sie mit dem Internet verbunden sein. Das PPM- Skript erhält seine Pakete von einer Sammlung auf der ActiveState-Website. Gestartet wird PPM einfach durch die Eingabe von `ppm` in eine Befehls-Shell:

```
c:\> ppm
PPM interactive shell (0.9.5) - type 'help' for available commands
PPM>
```

Nach der PPM-Eingabeaufforderung haben Sie mehrere Möglichkeiten einschließlich

- `help` - um eine Liste von Optionen auszugeben
- `search` - um anzuzeigen, welche Pakete für eine Installation zur Verfügung stehen
- `query` - um anzuzeigen, welche Pakete Sie bereits installiert haben
- `install` - um ein spezielles Paket zu installieren
- `verify` - um zu überprüfen, dass alle Pakete aktuell sind
- `remove` - um ein spezielles Paket zu entfernen

Mehr über PPM erfahren Sie auf der PPM-Webseite (Teil der ActiveState-Site für die Installation von Perl unter Windows) oder unter `http://www.activestate.com/ activeperl/docs/ppm.html`.

Module von CPAN nutzen

Nachdem Sie ein CPAN-Modul installiert haben - entweder durch eigenes Herunterladen und Installieren, mit Hilfe

von PPM oder durch manuelles Kopieren der Dateien in Ihre Perl-Installation - steht das Modul Ihren Skripts zur Verfügung. Sie können es anschließend mit `use` importieren und sein Leistungsspektrum wie bei jedem anderen Modul nutzen. Anhand der Modul-Dokumentation können Sie sich darüber informieren, ob es ein normales oder objektorientiertes Modul ist, oder feststellen, ob es Import-Tags verwendet. Aber abgesehen davon, dass sie erst einmal installiert werden müssen, unterscheiden sich CPAN-Module in nichts von denen der Standardbibliothek.

Vertiefung

In diesem Kapitel habe ich etliche Themen angerissen: Pakete, Module, Importieren derselben und die Art und Weise, wie Perl den Code zu verschiedenen Zeiten sieht. Vieles, was in diesem Kapitel angesprochen wurde, ist nur die Spitze des Eisbergs. Allein über Pakete und Module könnte man ganze Bücher füllen, und die paar Absätze, die ich dem objektorientierten Design gewidmet habe, reichen kaum aus, um in das Thema einzusteigen.

Auf einige dieser Themen werde ich weiter hinten in diesem Buch noch näher eingehen. Andere Themen jedoch, einschließlich der Entwicklung eigener Pakete und praktisch alles, was mit der Erstellung von Modulen zusammenhängt, dürfen den erfahreneren Perl-Programmierern vorbehalten bleiben und werden deshalb nicht im Rahmen dieses Buches behandelt. Nachdem Sie die 21 Lektionen dieses Buchs durchgearbeitet haben, können Sie Ihre Kenntnisse über Module und Pakete mit Hilfe der Online-Hilfsdokumentationen und der FAQs erweitern.

Einige Themen mit direktem Bezug zu den bisher in diesem Kapitel behandelten Themen kann ich jedoch noch in diesem Abschnitt behandeln.

Typeglobs (Typenplatzhalter)

Typeglobs ist eine seltsame Bezeichnung dafür, dass man auf die verschiedenen Typen von Variablen mit einem einzigen Namen Bezug nehmen kann (der Begriff ***Typeglob*** stammt von Unix, wo der Bezug auf mehrere Dateien mit `file.*` oder einem ähnlichen Zeichen auch als Datei-***Globbing*** bezeichnet wird). Typeglobbing bezieht sich auf die Einträge einer Symboltabelle eines gegebenen Pakets.

Der Typglob-Ausdruck `*foo` bezeichnet alle Variablen mit dem Namen `foo`: `$foo, @foo`, `&foo` und so weiter. Normalerweise unterscheiden sich diese Variablen, mittels Typeglobbing werden Sie jedoch zusammengefaßt.

In früheren Versionen von Perl wurde Typeglobbing dazu verwendet, um Arrays als Referenz an Subroutinen zu übergeben. Sie konnten einen Typeglob verwenden, um eine lokale Variable als Alias für ein globales Array zu verwenden; Änderungen am lokalen Array sollten dann an das globale Array übertragen werden:

```
@foo = (1,2,3);
&dreienentfernen(*foo);
sub dreienentfernen {
   my *liste = @_;
   foreach my $x @liste {
      if ($x == 3) { undef $x }
   }
   return @liste;
}
```

In diesem Beispiel finden sich alle Änderungen an der lokalen Liste, in `@liste` auch in der Liste `@foo`, da `@liste` als Alias für `@foo` eingerichtet wurde.

Statt Typeglobs verwendet man für die Übergabe von Arrays an Subroutinen heute meist die neue Referenzsyntax. Mit Referenzen können Sie nicht nur einzelne Arrays als Referenz an Subroutinen übergeben, sondern auch die Integrität mehrerer Arrays wahren. Verwenden Sie deshalb für die Übergabe von Arrays an Subroutinen Referenzen statt Typeglobs (weitere Informationen zu Referenzen in Kapitel 19).

Ein weiterer Unterschied zwischen local und my

Der auffälligste Unterschied zwischen Variablen, die mit `my` definiert wurden, und solchen, die mit `local` definiert wurden, sind, wie ich weiter vorn schon bemerkt habe, der lexikalische und der dynamische Gültigkeitsbereich. Ein zweiter Unterschied besteht darin, wie Perl diese Variablen verwaltet:

- `local`-Variablen sind eigentlich versteckte globale Variablen. Wenn Sie eine lokale Variable erzeugen, während eine globale Variable gleichen Namens existiert, sichert Perl den Wert dieser globalen Variablen und reinitialisiert die gleiche Variable (und die gleiche Position in der Symboltabelle) mit dem neuen lokalen Wert. Nach dem Verlassen des lokalen Gültigkeitsbereichs ist der Wert der globalen Variablen wieder gültig.

- `my`-Variablen sind völlig neue Variablen, die nicht in der Symboltabelle gespeichert werden. Sie sind absolut privat zu dem Block oder der Subroutine, in der sie gespeichert sind. Damit sind sie schneller in der Anwendung als die lokalen `local`- Variablen, da kein Durchsuchen der Symboltabelle erforderlich ist.

Keiner der Unterschiede wird einen Einfluß darauf haben, wie Sie `local` oder `my` in Ihren eigenen Skripts verwenden (lokale `local`-Variablen sind für das Typeglobbing relevant, doch das ist nicht mehr Thema dieses Buches). In den meisten Fällen sollten Sie `my` verwenden, wenn Sie eine lokale Variable erzeugen wollen, und es wird keine Probleme geben.

Ein Beispiel für local im Vergleich zu my

In dem Abschnitt zu den lokalen Variablen, die entweder mit `local` oder mit `my` definiert wurden, hatte ich Ihnen noch ein Beispiel versprochen. Das Skript in Listing 13.2 soll etwas Licht ins Dunkel der Unterscheidung zwischen `local` und `my` bringen (und Sie hoffentlich nicht noch mehr verwirren).

Listing 13.2: Ein Skript zum Gültigkeitsbereich

```
1:  #!/usr/bin/perl -w
2:
3:  $global = " globale Variable hier verfügbar\n";
4:
5:  &subA();
6:  print "Hauptskript:\n";
7:  foreach $var ($global, $mylocal, $locallocal) {
8:      if (defined $var) {
9:          print $var;
10:     }
11: }
12:
13: sub subA {
14:     my $mylocal = " mylocal-Variable hier verfügbar\n";
15:     local $locallocal = " lokale local-Variable hier verfügbar\n";
16:     print "SubA:\n";
17:     foreach $var ($global, $mylocal, $locallocal) {
18:         if (defined $var) {
19:             print $var;
20:         }
21:     }
22:     &subB();
23: }
24:
25: sub subB {
26:     print "SubB: \n";
27:     foreach $var ($global, $mylocal, $locallocal) {
28:         if (defined $var) {
29:             print $var;
30:         }
31:     }
32: }
```

Dieses Skript verwendet drei Variablen, `$global`, `$mylocal` und `$locallocal`, und deklariert sie ordnungsgemäß. Für jede Subroutine sowie am Ende des Skripts werden die Werte dieser Variablen ausgegeben, soweit sie existieren und einen definierten Wert haben. Versuchen Sie, dem Programmfluß zu folgen und vorherzusagen, was wann ausgedruckt wird.

Und so lautet die Ausgabe:

```
SubA:
 globale Variable hier verfügbar
 mylocal-Variable hier verfügbar
```

```
 lokale local-Variable hier verfügbar
SubB:
 globale Variable hier verfügbar
 lokale local-Variable hier verfügbar
Hauptskript:
 globale Variable hier verfügbar
```

Hätten Sie das erwartet? Lassen Sie uns das Programm gemeinsam durchgehen. Das Skript beginnt oben und ruft `&SubA()` auf, eine Subroutine, die wiederum `&SubB()` aufruft und einige Variablen ausgibt. In der Subroutine `&SubA()` deklarieren wir in Zeile 14 und 15 `$mylocal` und `$locallocal` mit den beiden Modifizierern `my` und `local`. Beide Variablen plus der Variablen `$global` sind damit innerhalb der Grenzen dieser Subroutine verfügbar, so dass alle drei Werte ausgegeben werden.

In Zeile 22 ruft `&subA()` die Subroutine `&subB()` auf. Hier geben wir nur die Variablen aus, die verfügbar sind. Das wäre zum einen die globale Variable, da globale Variablen allen Teilen des Skripts zur Verfügung stehen. Die Variable `$mylocal` gibt es allerdings nicht mehr, denn der `my`-Modifizierer legt fest, dass die lokale Variable nur in der Subroutine verfügbar ist, in der sie definiert wurde, und nicht in anderen Teilen des Skripts. Die Variable `$locallocal` ist dagegen auch in den Subroutinen verfügbar, deren Definitionen von der Subroutine mit der Variablendefinition umschlossen werden.

Nachdem `&subB()` abgearbeitet ist, kehrt die Ausführung zurück zu `&subA()` und damit zum Hauptteil des Skripts, wo wir ebenfalls versuchen, diese Werte auszugeben. Da hier nur die globale Variable verfügbar ist, wird auch nur deren Wert ausgegeben.

Paketinitialisierung und -beendigung mit BEGIN und END

Bevor wir fortfahren, möchte ich Ihnen noch einen Aspekt der Anwendung von Paketen vorstellen: die Subroutinen `BEGIN` und `END`, mit denen ein Skript vor seiner Ausführung initialisiert und nach seiner Ausführung beendet wird. Diese Subroutinen werden am häufigsten in komplexen Perl-Bibliotheken und -Modulen und als Konstruktoren und Destruktoren für objektorientierte Klassen verwendet.

Die `BEGIN`-Subroutine wir ausgeführt, sobald das Programm darauf stößt, und zwar während der Kompilierzeit, bevor der Rest des Skripts geparst ist. Sie sollten `BEGIN` für jeden Code verwenden, der zur Kompilierzeit ausgeführt werden soll, zum Beispiel um Symbole für Moduldefinitionen zu importieren, die später in anderen Code exportiert werden, oder um Code mit aufzunehmen, der von dem Modul benötigt wird.

`END` hingegen wird ausgeführt, nachdem die Ausführung des Perl-Skripts abgeschlossen ist. Das gilt sowohl für Skripts, die korrekt ausgeführt wurden, als auch für frühzeitig abgebrochene Skripts, bei denen ein Fehler auftrat (einschließlich `die`). Mit `END` sollten Sie nach der Ausführung Ihres Skripts aufräumen. Sie können mit `END` zum Beispiel den Statuswert ändern, der an die Unix-Shell zurückgegeben wird, nachdem Ihr Skript beendet ist.

Mehr zu `BEGIN` und `END` finden Sie in der ***perlmod***-Manpage.

Code mit *require* importieren

Bisher habe ich Ihnen gezeigt, wie Sie mit der `use`-Funktion Code aus Modulen importieren. Kurz gesagt: `Use` importiert zur Kompilierzeit Code und Symbole aus anderen Quellen in das aktuelle Paket. Die Funktion `require` hingegen ermöglicht es, Code von anderen Quellen zur Laufzeit mit aufzunehmen (um genau zu sein, entspricht `use` dem Aufruf von `require` innerhalb einer `BEGIN`-Subroutine und dem Import der Variablen dieser Datei in den aktuellen Namensbereich).

In früheren Versionen von Perl wurde `require` als allgemeiner Import-Mechanismus verwendet. Subroutinen und globale Variablendefinitionen wurden in einer separaten Datei gespeichert, und diese Datei wurde dann mit `require` in das Skript mit aufgenommen:

```
require 'foo.pl';
```

Perl sucht nach der gegebenen Datei, die importiert werden soll, in den in `@INC` gespeicherten Verzeichnissen. Gleichzeitig merkt sich Perl, welche Dateien bereits importiert wurden, so dass bereits geladener Code nicht noch einmal importiert wird. Wenn die aufgenommene Datei jedoch ihr eigenes Paket definiert, dann werden mit

`require` die Variablen dieses Pakets nicht in das aktuelle Paket importiert (nicht einmal von `main`), und Sie müssen diese Variablen mit dem kompletten Paketnamen ansprechen. Außerdem gilt es bei der Verwendung von `require` Sorgfalt walten zu lassen, denn `require` wird zur Laufzeit ausgeführt. Das bedeutet, Sie müssen sicherstellen, dass `require` ausgeführt wird, bevor irgend etwas aufgerufen oder verwendet wird, was in dieser importierten Datei definiert ist.

Dieser Mechanismus zum Importieren von Code von einer Datei in eine andere funktioniert auch noch anstandslos in den neueren Versionen von Perl, und Sie können ihn verwenden, um Ihre eigenen einfachen Bibliotheken mit Subroutinendefinitionen aufzubauen. Die neuen Mechanismen der Pakete und Module sind jedoch leistungsfähiger und erlauben eine genauere Kontrolle über das, was und vor allem wann importiert wird. Wenn Sie ernsthaft in die Entwicklung von Bibliotheken einsteigen wollen, sollten Sie sich intensiver mit den Entwicklungspaketen, Modulen und `use` vertraut machen.

Ein weiteres nützliches Einsatzgebiet von `require` ist seine Verwendung zusammen mit der Perl-Versionsnummer. Falls Sie diese Einsatzmöglichkeit nutzen und das Skript mit einer früheren Version von Perl ausführen, wird das Skript sofort mit einer Fehlermeldung abbrechen. Damit können Sie sicherstellen, dass die zur Ausführung verwendete Perl-Version auch über die gleichen Merkmale verfügt, die Sie in Ihrem Skript verwenden - zum Beispiel Merkmale, die es nur in Perl 5 gibt, oder noch fortschrittlichere Merkmale, die nur in 5.005 oder höher existieren:

```
require 5.005;
```

Weitere Informationen zu `require` finden Sie in der ***perlfunc-***Manpage.

Zusammenfassung

Einige der heute behandelten Themen mögen Ihnen vielleicht etwas abgehoben vorkommen, aber sie werden mit jedem weiteren Kapitel dieses Buches an Bedeutung gewinnen. Die erste Hälfte dieses Kapitels war den Variablen und ihren Gültigkeitsbereichen gewidmet. Ich habe Ihnen die globalen Variablen vorgestellt und gezeigt, was es bedeutet, wenn sie durch die Verwendung von Paketen nicht mehr im eigentlichen Sinne global sind, und wie man sicherstellt, dass globale Variablen durch die Definition von `my` lokal zu einem Skript sind.

Dann sind wir zu den lokalen Variablen übergegangen, und Sie haben mehr über die Verwendung von `my`-Variablen innerhalb von Blöcken und Subroutinen erfahren und über die Definition von lokalen Variablen mit `local`.

In Anbetracht all dieser Möglichkeiten zum Deklarieren und Verwenden der verschiedenen Variablen ist es schwer, die richtige Wahl zu treffen. Perl- Programmierer haben allgemeine Regeln und Praktiken entwickelt, die gewöhnlich als Leitfaden für die Anwendung von Variablen und Gültigkeitsbereichen dienen:

- Deklarieren Sie keine reinen globalen Variablen. Deklarieren Sie alle globalen Variablen mit `my`, und verwenden Sie `use strict`, um sicherzustellen, dass alles seine Ordnung hat.
- Deklarieren Sie lokale Variablen vorzugsweise mit `my` und nicht mit `local`, es sei denn, es gibt besondere Gründe dafür. Lokale `local`-Variablen weisen viele der gleichen Probleme hinsichtlich Wiederverwertung und Debuggen auf wie globale Variablen und beeinträchtigen die ansonsten klare Trennung zwischen den Begriffen *lokal* und *global*.
- Versuchen Sie, gleichlautende Namen unter lokalen und globalen Variablen zu vermeiden, es sei denn, es gibt besondere Gründe dafür.
- In der zweiten Hälfte des Kapitels bin ich auf die `use`-Funktion eingegangen. Sie haben gelernt, wie Sie `use` einsetzen, um Pragmas einzuschalten - Hinweise an Perl, wie Ihre Skripts zu kompilieren und auszuführen sind - und um Code zu importieren und zu verwenden, der in Modulen der Standardbibliothek oder der CPAN enthalten ist. Zu den wichtigsten Punkten, die Sie heute gelernt haben, gehören die Module. Module werden uns im Rest des Buches immer wieder begegnen.

Die weiteren Funktionen und Befehle, die wir heute besprochen haben, umfassen:

- `package` - um zwischen den verschiedenen Paketen hin- und herzuschalten
- `my` - um eine lokale Variable zu definieren oder eine globale Variable, die nicht Teil eines Pakets ist
- `use strict` - um sicherzustellen, dass Sie nicht irgendwelche wilden globalen Variablen verwenden
- `local` - um auf eine andere Art lokale Variablen zu definieren (`my` ist vorzuziehen)

- `use` - um allgemein ein Pragma oder ein Modul zu importieren

Fragen und Antworten

Frage:
Worin liegt der Vorteil von globalen Variablen, die mit `my` deklariert wurden, gegenüber normalen globalen Variablen, wenn ich in sich geschlossene Skripts schreibe, die nur sehr einfache Aufgaben ausführen?

Antwort:
Wenn Sie nur kleinere, abgeschlossene Skripts schreiben, müssen Sie nicht unbedingt `use strict` verwenden oder sicherstellen, dass Ihre Variablen mit `my` deklariert wurden. Globale `my`-Variablen werden dann verwendet, wenn davon auszugehen ist, dass Ihr Code, und sei es auch nur zufällig, in den Code eines anderen Skripts inkorporiert wird. Da es häufig vorkommt, dass Code - auch gegen den ursprünglichen Zweck - wiederverwertet wird, ist die Verwendung von `my` und `use strict` bei globalen Variablen nur eine vorsorgliche Absicherung und guter Programmierstil. Erforderlich ist es jedoch nicht immer.

Frage:
Ich verwende die ActiveState-Version von Perl für Win32. Mein Skript weist `use strict`-Definitionen auf und beschwert sich aber über die Variable innerhalb der `foreach`-Definition. Deklariere ich die Variable mit `my`, erhalte ich ebenfalls einen Fehler. Was mache ich falsch?

Antwort:
Nichts. In der ActiveState-Version von Perl für Win32 können Sie die `my`- Deklaration nicht innerhalb der `foreach`-Schleife verwenden. Deklarieren Sie die `my`-Variable einfach irgendwo anders, und verwenden Sie sie dann innerhalb der `foreach`-Schleife. Damit sind Sie aus dem Schneider.

Frage:
Die meisten Sprachen verwenden entweder den lexikalischen oder den dynamischen Gültigkeitsbereich, aber nicht beide. Warum schafft Perl da Verwirrung, indem es mehrere Arten von lokalen Gültigkeitsbereichen bietet?

Antwort:
Das hat größtenteils historische Gründe. Frühere Versionen von Perl boten einen dynamischen Gültigkeitsbereich für Variablen mit dem `local`-Operator. Der `my`-Operator kam später hinzu, um einen lexikalischen lokalen Gültigkeitsbereich zu schaffen, der eindeutiger ist. Aus Gründen der Rückwärtskompatibilität mit früheren Skripts gibt es `local` immer noch. Wenn Ihnen die Unterscheidung zuviel Kopfzerbrechen bereitet, verwenden Sie einfach ausschließlich `my`-Variablen, und gehen Sie davon aus, dass sie privat zu allen gegebenen Subroutinen sind. Das dürfte reichen.

Frage:
Ich habe ein älteres Skript von jemand anderem, das mit einer Reihe von `require dieseBibliothek.pl`-Zeilen beginnt. Soll ich diese in `use`-Befehle ändern?

Antwort:
Besser nicht. Mit dem `require`-Operator wurde früher Bibliothekscode in Perl- Skripts eingebunden, und es ist wahrscheinlich, dass die Bibliothek, die Sie importieren (`dieseBibliothek.pl` in Ihrem Beispiel), nicht als Modul geschrieben und deshalb auch nicht gut mit `use` anzusprechen ist. Solange Sie nicht vorhaben, das gesamte Skript neu zu schreiben - einschließlich aller Bibliotheken -, sollten Sie ruhig weiterhin `require` verwenden.

Frage:
Ich habe hier ein Modul namens `Mail`, das mir Zugriff auf Subroutinen zum Senden und Empfangen von Mails verschaffen soll: `send_mail()` und `rec_mail()`. Ich habe das Modul mit `use` importiert, erhalte aber Fehlermeldungen, dass die beiden Subroutinen nicht definiert seien.

Antwort:
Das klingt, als wenn das Modul diese zwei Subroutinen nicht explizit importiert. In diesem Falle haben Sie zwei Möglichkeiten: Entweder importieren Sie die Subroutinen selbst, oder Sie rufen die Subroutinen mit ihrem vollen

Paketnamen auf:

```
    use Mail;  # importiert Standardroutinen, falls vorhanden
use Mail qw(send_mail rec_mail);  # importiert Subroutinen
send_mail();
# ODER
&Mail::send_mail();  # ruft Subroutine mit vollem Paketnamen auf
```

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen, Ihr Wissen zu festigen, und Übungen, die Sie anregen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist ein Paket? Warum sind Pakete nützlich?
2. Wie rufen Sie eine Variable mit ihrem vollen Paketnamen auf?
3. Können Sie `my` mit globalen Variablen verwenden? Warum sollten Sie das machen?
4. Was bewirkt die Zeile `use strict`? Warum sollten Sie diesen Befehl verwenden?
5. Worin liegen die Unterschiede zwischen Bibliotheken, Modulen, Paketen und Pragmas?
6. Was ist das CPAN? Wo ist es zu finden?
7. Wozu wird das `@INC`-Array benötigt?
8. Wie importieren Sie ein Modul in Ihr Skript? Was bringt Ihnen der Import?
9. Was versteht man unter einem Import-Tag, und wann sollte man es verwenden?
10. Wie rufen Sie eine Subroutine auf, die Sie von einem Modul importiert haben? Wie rufen Sie eine Subroutine von einem objektorientierten Modul auf?

Übungen

1. Definieren Sie eine Subroutine, die als einziges Argument einen String übernimmt und einen neuen String daraus macht, wobei die Zeichen durch einen global definierten Wert getrennt werden (wie zum Beispiel »:«). Geben Sie den neuen String zurück. Der Haken dabei: Verwenden Sie den gleichen Variablennamen für die lokale Variable, die den neuen String, und die globale Variable, die das Trennzeichen enthält. HINWEIS: Verwenden Sie nicht `use strict` oder globale `my`-Variablen.
2. FEHLERSUCHE: Was ist falsch an diesem Skript?

```
    use Dies:Das;                # importiert Module
while (<>) {
    print dasandere($_);
}
```

3. Modifizieren Sie das Skript ***umbrechen.pl***, so dass es Sie auffordert, eine Spaltenbreite einzugeben. Brechen Sie dann den Eingabetext nach dieser Größenangabe um.
4. Betrachten Sie das Modul `Config`, das Teil der Standard-Perl-Bibliothek ist. Dieses Modul wird verwendet, um Konfigurationsinformationen über die aktuelle Perl- Version zu speichern. Bedienen Sie sich der Dokumentation zu `Config` (die Sie über das Programm ***perldoc***, die ***Shuck***-Anwendung in MacPerl oder über die ***perlmod-***Manpage erhalten). Schreiben Sie ein Skript, das die verschiedenen Werte, die in `Config` verfügbar sind, ausgibt. HINWEIS: Das `Config`-Modul importiert nicht automatisch alle Subroutinennamen.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Pakete werden verwendet, um Sätze von globalen Variablen in einer einzigen Einheit zu bündeln. Damit können diese Einheiten kombiniert werden, ohne dass es zu Überschneidungen zwischen den Variablen der verschiedenen Einheiten kommt. Pakete sind am nützlichsten für Skripts, die aus mehreren Teilen bestehen,

die harmonisch aufeinander abgestimmt werden müssen, oder für Module mit wiederverwendbarem Code.

2. Der vollständige Paketname einer jeden Variablen oder Subroutine besteht aus dem Variablensymbol (`$` für Skalare, `@` für Arrays, `%` für Hashes, `&` für Subroutinen und so weiter), dem Paketnamen, zwei Doppelpunkten und dem Variablennamen. Ein Beispiel: `$Einmodule::einevariable` oder `&Einmodule::einesubroutine()`.
3. Die Deklaration von globalen Variablen mit `my` verhindert, dass sie in dem `main`- Paket deklariert werden. Dadurch wird die Wertesuche und -zuordnung etwas effizienter, und Ihr Skript zeigt ein besseres Verhalten, sollte es je in ein anderes Paket inkorporiert oder mit anderen Skripts kombiniert werden.
4. `use strict` ist eine Perl-spezifische Direktive, die sicherstellt, dass alle Variablen in Ihrem Skript lokal oder einem bestimmten Paket zugeordnet sind. Auf Ihrem derzeitigen Perl-Wissensstand ist es sinnvoll, freie globale Variablen abzufangen und Ihre Skripts hinsichtlich der Variablendeklarationen so abgeschlossen wie möglich zu machen.
5. Bibliotheken sind Sammlungen von Perl-Code, der wiederverwendet werden kann. Bibliotheken können Pakete verwenden, um Variablennamen innerhalb der Bibliothek zu verwalten. Ein Modul ist eine Bibliothek, die ein Paket verwendet und den gleichen Dateinamen trägt wie das Paket. Module umfassen Code, der dafür sorgt, dass das Modul ohne Schwierigkeiten in anderen Skripts wiederverwendet werden kann. Ein Pragma ist eine besondere Art von Modul, das Einfluß auf die Arbeitsweise von Perl zur Kompilier- und Laufzeit nimmt.
6. CPAN steht für Comprehensive Perl Archive Network. Es handelt sich dabei um eine Sammlung von Beiträgen von Perl-Nutzern: Module, Skripts, Dokumentationen und Hilfsprogramme, die jeder, der in Perl programmiert, nutzen kann. CPAN ist auf diversen Sites rund um den Globus verfügbar. Ihre nächstgelegene Site finden Sie ausgehend von `http://www.perl.com/CPAN/`.
7. Das `@INC`-Array definiert die Verzeichnisse, in denen Perl nach den Modulen und dem Code schaut, der in Ihre Skripts mit `use` importiert werden soll.
8. Importieren Sie ein Modul in Ihr Skript durch Verwendung von `use`, gefolgt von dem Namen des Moduls und einer optionalen Liste von Variablen oder Import- Tags. Der Import eines Moduls verschafft Ihnen Zugriff auf die Variablen und Subroutinen, die von dem Modul definiert sind.
9. Ein Import-Tag definiert einen Teilbereich von Variablen und Subroutinen in dem Modul, das Sie in Ihr Skript importieren wollen. Import-Tags werden von dem Modul-Entwickler definiert und sind in den jeweiligen Dokumentationen zu den Modulen nachzulesen.
10. Subroutinen, die von Modulen importiert wurden, können wie reguläre Subroutinen aufgerufen werden. Verwenden Sie dazu den Namen der Subroutine mit Klammern um eventuelle Argumente.

1. In objektorientierten Modulen müssen Sie ein neues Objekt erzeugen, bevor Sie Subroutinen aufrufen können. Mit einem in einer Skalarvariablen gespeicherten Objekt, würden Sie dann eine Subroutine mit der Syntax `$var->sub()` aufrufen, wobei `$var` der Name der Variablen ist, die das Objekt enthält, und sub der Name der Subroutine. Subroutinen, die innerhalb von Objekten definiert sind, werden auch als Methoden bezeichnet.

Lösungen zu den Übungen

1. Das Geheimnis liegt darin, richtige globale Variablen zu verwenden, die in dem Paket `main` gespeichert sind, und dann im Rumpf der Subroutine über den vollen Paketnamen darauf Bezug zu nehmen. Beachten Sie, dass dies mit globalen Variablen, die mit `my` definiert wurden, nicht funktioniert, da sie zu keinem Paket gehören.

```
    #!/usr/bin/perl -w
$x = ":";               # Trennzeichen (global)
print &auftrennen("defenestration"), "\n";
sub auftrennen {
    my $string = $_[0];
    my $x = '';                 # neuer String (global);
    $x = join $main::x, (split //,$string);
    return $x;
}
```

2. In dem Modulnamen steht nur ein Doppelpunkt. Module mit zwei Namen haben immer zwei Doppelpunkte.
3. Die Änderungen an ***umbrechen.pl*** bestehen lediglich darin, die Variable `$columns` zu modifizieren. Diese Variable wird nicht automatisch in Ihr Skript importiert. Das müssen Sie selbst übernehmen.

```
    #!/usr/bin/perl -w
use strict;
use Text::Wrap;                 # importiert Standardmodule
```

```
use Text::Wrap qw($columns);    # importiert auch $columns
$\ = "";                        # Absatzmodus
my $indent = "> ";              # Einrückungszeichen
print 'Geben Sie eine Spaltenbreite an: ';
chomp($columns = <STDIN>);
while (<>) {
    print wrap($indent, $indent, $_);
}
```

4. Die Subroutine `myconfig()` übernimmt das für Sie. Da diese Subroutine nicht automatisch importiert wird, müssen Sie sie durch Angabe des vollen Paketnamens aufrufen (oder explizit importieren):

```
    #!/usr/bin/perl -w
use strict;
use Config;
print Config::myconfig();
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Ein paar längere Beispiele

Widmen wir den letzten Tag der Woche ein paar weiteren Beispielen. Sie werden in diesem Kapitel nicht viel Neues lernen, auch Übungen und Quiz gibt es heute nicht. Betrachten Sie es als kurze Verschnaufpause, in der Sie sich vollständige Perl-Skripts ganz in Ruhe ansehen können. Insgesamt hat das Buch drei dieser Beispiellektionen (immer am Ende der Wochen), die Ihr neues Wissen festigen.

Heute analysieren wir zwei längere Perl-Skripts:

- Ein Adreßbuch-Skript, das die Namen und Adressen in dem Format einer einfachen Textdatenbank speichert. Mit unserem Perl-Skript können Sie in der Datenbank mit Hilfe einfacher logischer UND- und ODER-Tests nach Adressen suchen.
- Einen Analysierer für die Log-Dateien von Websites. Dieses Skript verarbeitet das Standardformat für Webserver-Log-Dateien (auch ***common log format*** genannt) und erzeugt Statistiken aller Art über die Nutzung der Website.

Ein Adreßbuch zum Durchsuchen (adressen.pl)

Unser erstes Skript heute besteht aus zwei Teilen:

- Einer einfachen Adreßbuch-Datei, die die Namen, Adressen und Telefonnummern enthält
- Einem Perl-Skript, das Sie auffordert, Suchbegriffe einzugeben, und dann alle gefundenen Adressen ausgibt

Dieses Skript nutzt so ziemlich alles, was Sie diese Woche gelernt haben: Skalar- und Hash-Daten, Bedingungen, Schleifen, Ein- und Ausgabe, Subroutinen, lokale Variablen und Mustervergleich. Es gibt sogar ab und zu einen Funktionsaufruf, um die Sache interessanter zu machen. Aber jetzt lassen Sie uns ohne weitere Umschweife beginnen.

Wie es funktioniert

Das Skript ***adressen.pl*** wird mit einem einzigen Argument aufgerufen: der Adreßdatei namens ***adressen.txt***. Geben Sie den Aufruf in der Befehlszeile ein, wie Sie es bereits bei anderen Perl-Skripten gemacht haben:

```
% adressen.pl adressen.txt
```

Wenn Sie MacPerl verwenden, speichern Sie das Skript ***adressen.pl*** als Droplet und ziehen (Drag) Sie dann die Datei ***adressen.txt*** auf das Symbol ***adressen.pl***, wo Sie diese ablegen (Drop).

Als erstes fragt das Adreßbuch-Skript Sie nach dem gesuchten Begriff:

```
Wonach soll gesucht werden? Johnson
```

Das Suchmuster, das Sie an `adressen.pl` übergeben, kann verschiedene Formen haben:

- einzelne Wörter, wie zum Beispiel `Johnson` im obigen Beispiel
- mehrere Wörter (`John Maggie Alice`). Alle Adressen, die einem dieser Wörter entsprechen, werden ausgegeben (gleicht einer ODER-Suche).
- mehrere Wörter getrennt durch `AND` oder `OR` (in Groß- oder Kleinbuchstaben). Boolesche Suchläufe verhalten sich wie logische Operatoren in Perl und werden von links nach rechts getestet. (Denken Sie daran, dass ein AND-Suchlauf nur sinnvoll ist, wenn der Vergleich innerhalb einer Adresse erfolgt; es erfolgt kein Vergleich

über mehrere Adressen, wie das bei OR-Suchläufen der Fall ist).
- mehrere Wörter in Anführungszeichen (»`dies das`«) werden als ein einziges Suchmuster betrachtet. In diesem Fall sind Leerzeichen von Bedeutung.
- Pattern-Matching-Zeichen werden akzeptiert und als reguläre Ausdrücke verarbeitet (lassen Sie die `//` um die Muster fort).

So liefert zum Beispiel die Suche nach dem Namen `Johnson` in meiner Beispieldatei ***adressen.txt*** folgende Ausgabe zurück:

```
**********
Paul Johnson
212 345 9492
234 33rd St Apt 12C, NY, NY 10023
http://www.foo.org/users/don/paul.html
**********
Alice Johnson
(502) 348 2387
(502) 348 2341
**********
Mary Johnson
(408) 342 0999
(408) 323 2342
mj@asd.net
http://www.mjproductions.com
**********
```

Alle Namen, Adressen, Telefonnummern und Webseiten dieser Beispiel-Adreßdatei sind natürlich frei erfunden. Irgendwelche Übereinstimmungen dieser Daten mit lebenden oder toten Personen sind reiner Zufall.

Die Adreßdatei

Das Herzstück eines Adreßbuches (das Sie selbst erstellen müssen, wenn Sie dieses Skript nutzen wollen) ist eine Datei von Adressen in einem speziellen Format, das von Perl verstanden wird. Sie können es auch als einfache textbasierte Datenbank betrachten und dann Perl-Skripts schreiben, die diese Datenbank um Datensätze (Adressen) ergänzen oder auch Datensätze löschen.

Das Format der Adreßbuch-Datei in seiner generischen Form sieht folgendermaßen aus:

```
Name: Name
Telefon: Nummer
Fax: Nummer
Adresse: Adresse
EMail: email-Adresse
URL: Web URL
---
```

zum Beispiel:

```
Name: Paul Johnson
Telefon: 212 345 9492
Adresse: 234 33rd St Apt 12C, NY, NY 10023
URL: http://www.foo.org/users/don/paul.html
---
```

Jeder Datensatz besteht aus einer Reihe von Feldern (Name, Telefon, Fax, Adresse, E-Mail und URL, die jedoch nicht alle erforderlich sind) und endet mit drei Gedankenstrichen. Die Feldnamen (`Name`, `Telefon`, `URL` usw.) sind von ihren Werten durch einen Doppelpunkt und ein Leerzeichen getrennt. Die Werte müssen kein spezielles Format aufweisen. Sie können weitere Feldnamen in die Datenbank aufnehmen, und die Suchschlüssel werden diese zusätzlichen Felder auch durchsuchen, aber in der Ausgabe werden diese Felder übergangen. (Wenn Sie wirklich ein zusätzliches Feld benötigen, zum Beispiel für eine Handy-Nummer, können Sie das Skript jederzeit ändern. Perl

macht es Ihnen in dieser Hinsicht leicht.)

Sie können beliebig viele Adressen in die Datei ***adressen.txt*** aufnehmen. Doch je größer das Adreßbuch, um so länger die Zeitdauer, bis übereinstimmende Datensätze gefunden werden, da für jede Adresse die Datei von vorn bis hinten durchgegangen wird. Solange Sie jedoch keine vier bis fünf Millionen Freunde haben, werden Sie von Perls Anstrengungen nichts mitbekommen.

Das Skript

Das Skript ***adressen.pl*** liest die Datei ***adressen.txt*** Adresse für Adresse ein und gleicht dann das Suchmuster mit jeder Adresse ab. Der oberste Teil des Skripts ist eine `while`-Schleife, die diese Aufgabe übernimmt, wobei wiederum fünf weitere Subroutinen abgearbeitet werden, um die komplexeren Teile des Skripts zu bewältigen.

Lassen Sie uns ganz oben im Skript beginnen. Auf oberster Ebene definieren wir drei globale Variablen:

- `%rec` - enthält den aktuellen Adreßdatensatz, der nach Feldnamen indiziert ist.
- `$search` - nimmt das Suchmuster auf, das Sie bei der Eingabeaufforderung eingeben.
- `$bigmatch` - gibt an, ob in der Adreßdatei ein Datensatz gefunden wurde, der dem Suchmuster entspricht (es gibt auch lokale Variablen für den Fall, dass der aktuelle Datensatz übereinstimmt, doch dazu gleich mehr).

Der erste Schritt in dem äußeren Teil des Skripts besteht darin, zur Eingabe des Suchbegriffs aufzufordern und diesen dann in `$search` abzulegen:

```
$search = &getpattern();        # fragt nach dem Suchmuster
```

Die Subroutine `&getpattern()` besteht aus den grundlegenden Schritten »Eingabe lesen / zurechtschneiden / Ergebnis zurückliefern«, die Ihnen schon viel zu oft in diesem Buch begegnet sind:

```
sub getpattern {
    my $in = '';  # Eingabe
    print 'Wonach soll gesucht werden? ';
    chomp($in = <STDIN>);
    return $in;
}
```

Schritt zwei im äußeren Teil des Skripts ist eine endlose `while`-Schleife, die einen Datensatz einliest, das Suchmuster verarbeitet und bei Übereinstimmung den Datensatz ausgibt:

```
while () {              # durchsucht die Adressdatei
    %rec = &read_addr();
    if (%rec) {         # Datensatz gefunden
        &perform_search($search, %rec);
    } else {            # Ende der Adressdatei, Aufräumarbeiten
        if (!$bigmatch) {
            print "Nichts gefunden.\n";
        } else { print "**********\n"; }
        last;           # verlassen, wir sind fertig
    }
}
```

Innerhalb der `while`-Schleife rufen wir `&read_addr()` auf, um einen Datensatz einzulesen, und wenn ein Datensatz gefunden wurde, durchsuchen wir ihn mit Hilfe der Subroutine `&perform_search()`. Sind wir am Ende der Adreßdatei angekommen und die Variable `$bigmatch` ist gleich `0`, heißt das, dass keine Übereinstimmungen gefunden wurden, und wir informieren den Anwender darüber. Am Ende der Adreßdatei rufen wir jedoch auf alle Fälle `last` auf, um aus der Schleife auszusteigen und das Skript zu beenden.

Adressen einlesen

Mit der Subroutine `&read_addr()` wird ein Adreßdatensatz eingelesen. In Listing 14.1 sehen Sie den Inhalt von `&read_addr()`.

Listing 14.1: Die Subroutine &read_addr()

```
1:  sub read_addr {
2:      my %curr = ();              # aktueller Datensatz
3:      my $key = '';               # temp. Schlüssel
4:      my $val = '';               # temp. Wert
5:
6:      while (<>) {
7:          chomp;
8:          if ($_ ne '---') {      # Datensatz-Trennzeichen
9:              ($key, $val) = split(/: /,$_,2);
10:             $curr{$key} = $val;
11:         }
12:         else { last; }
13:     }
14:     return %curr;
15: }
```

In früheren Beispielen mit `while`-Schleife, die <> verwendeten, haben wir die gesamte Datei auf einmal eingelesen und verarbeitet. Bei dieser `while`-Schleife ist das etwas anders; sie liest nur Teile der Datei ein und hört auf, wenn Sie auf ein Datensatz- Trennzeichen stößt (in diesem Fall der String '`---`'). Das nächste Mal, wenn die Subroutine `&read_addr()` aufgerufen wird, fährt die `while`-Schleife dort fort, wo sie in der Adreßdatei gestoppt hat. Perl hat keine Schwierigkeiten mit diesem Stop-and-Go der Eingabe und ist damit besonders geeignet zum Einlesen und Verarbeiten von Teilabschnitten einer Datei, wie sie hier vorliegen.

Konkret ausgedrückt, liest diese Subroutine eine Zeile ein. Lautet die Zeile nicht '`---`', so befindet sich die Zeile mitten in einem Datensatz und besteht aus dem Feldnamen (`Name:`, `Telefon:`, usw.) und dem Wert. Der Aufruf der `split`-Funktion erfolgt in Zeile 9. Beachten Sie das zweite Argument am Ende von `split`; damit wird angezeigt, dass es in jeder Datensatzzeile nur zwei Teile gibt. Mit dem Feldnamen (`$key`) und dem Wert (`$val`) können Sie beginnen, den Hash für diese Adresse einzurichten.

Handelt es sich bei der eingelesenen Zeile um die Datensatzende-Marke, springt die `if`-Anweisung in Zeile 8 direkt zum `else`-Teil in Zeile 12, wo der `last`-Befehl die Schleife verläßt. Das Ergebnis dieser Subroutine ist ein Hash, der alle Zeilen der Adresse nach Feldnamen indiziert enthält.

Die Suche durchführen

Inzwischen sind Sie in der Ausführung des Skripts soweit fortgeschritten, dass Sie einen Suchbegriff in der Variablen `$search` und eine Adresse in der Variablen `$rec` gespeichert haben. Der nächste Schritt besteht jetzt darin, zum nächsten Teil unser großen `while`-Schleife zu Beginn des Skripts überzugehen. Das heißt, wenn `%rec` definiert ist (also eine Adresse existiert), rufen wir die Subroutine `&perform_search()` auf, um konkret festzustellen, ob der Suchbegriff in `$search` auch mit der Adresse in `&rec` übereinstimmt.

Die Subroutine `&perform_search()` wird in Listing 14.2 gegeben:

Listing 14.2: Die Subroutine &perform_search()

```
1:  sub perform_search {
2:      my ($str, %rec) = @_;
3:      my $matched = 0;            # Übereinstimmung
4:      my $i = 0;                  # Position innerhalb des Suchmusters
5:      my $thing = '';             # temporäres Wort
6:
7:      my @things = $str =~ /("[^"]+"|\S+)/g;  # in Suchelemente aufspalten
8:
9:      while ($i <= $#things) {
10:         $thing = $things[$i];   # Suchelement, UND oder ODER
11:         if ($thing eq 'ODER' || $thing eq 'oder') { # OR Fall
12:             if (!$matched) {    # noch keine Übereinstimmung,
                                    # nächstes Element
13:                 $matched = &isitthere($things[$i+1], %rec);
14:             }
15:             $i += 2;            # ODER überspringen und nächstes Element
16:         }
17:         elsif ($thing eq 'UND' || $thing eq 'und') { # UND-Fall
```

```
18:             if ($matched) {
                # Übereinstimmung gefunden, andere Seite prüfen
19:                 $matched = &isitthere($things[$i+1], %rec);
20:             }
21:             $i += 2;            # UND überspringen und nächstes Element
22:         }
23:         elsif (!$matched) {     # noch keine Übereinstimmung
24:             $matched = &isitthere($thing, %rec);
25:             $i++;               # weiter!
26:         }
27:         else { $i++; }          # $match wurde gefunden, weiter zum
                                    # nächsten Element
28:     }
29:
30:     if ($matched) {             # alle Schlüssel durchgearbeitet, gab es
                                    # eine Übereinstimmung?
31:         $bigmatch = 1;          # Ja, wir haben etwas gefunden
32:         print_addr(%rec);       # Datensatz ausgeben
33:     }
34: }
```

Diese Subroutine ist sehr lang und etwas komplexer. Aber keine Bange, sie ist nicht so kompliziert, wie es den Anschein hat. Fangen wir oben an; dort übernimmt die Subroutine zwei Argumente: den Suchbegriff und den Adressen-Hash: Beachten Sie, dass diese Werte in globalen Variablen gespeichert sind. Deshalb gibt es an sich auch keinen Grund, sie als Argumente an die Subroutine zu übergeben. Wir hätten auf diese globalen Variablen einfach im Rumpf der Subroutine zugreifen können. Unsere Strategie, die Daten als Argumente zu übergeben, führt allerdings dazu, dass die Subroutine in sich abgeschlossener ist, da nur die Daten bearbeitet werden, die explizit übergeben werden. Sie könnten zum Beispiel diese Subroutine kopieren und in ein anderes Suchskript einfügen, ohne einen Gedanken an die Umbenennung von Variablen verschwenden zu müssen.

Die erste richtige Operation in dieser Subroutine erfolgt in Zeile 7, in der wir den Suchbegriff in seine Bestandteile aufsplitten. Denken Sie daran, dass der Suchbegriff in vielen Formen auftreten kann, einschließlich verschachtelter Strings in Anführungszeichen, UNDs und ODERs oder als eine Liste von Schlüsselwörtern. In Zeile 7 werden die einzelnen Elemente aus dem Suchbegriff extrahiert, und anschließend werden diese »Suchelemente« gemeinsam in dem Array `@things` gespeichert. Dabei sollten Sie beachten, dass der reguläre Ausdruck mit der Option `g` endet und in einem Listenkontext ausgewertet wird - das soll heißen, dass die `@things`-Liste alle möglichen von den Klammern eingefangenen Übereinstimmungen enthält. Womit stimmt dieses besondere Muster überein? Es gibt zwei Gruppen von Mustern, getrennt durch das Alternationszeichen (`|`). Die erste Gruppe lautet:

```
"[^"]+"
```

Dabei handelt es sich, wenn Sie an Ihre Muster zurückdenken, um ein doppeltes Anführungszeichen gefolgt von einem oder mehreren Zeichen, die kein doppeltes Anführungszeichen sind, und einem abschließenden Anführungszeichen. Dieses Muster vergleicht mehrteilige Strings im Suchbegriff (in Anführungszeichen), wie zum Beispiel »`John Smith`« oder »`San Francisco`«, und behandelt sie als einen Suchbegriff.

Der zweite Teil des Musters besteht lediglich aus einem oder mehreren Zeichen, die keine Whitespace-Zeichen sind (`\S`). Dieser Teil des Musters vergleicht alle einfachen Wörter, wie zum Beispiel UND oder ODER, oder einzelne Schlüsselwörter. Von diesen beiden Mustern wird ein langer, komplexer Suchbegriff wie »`San Jose`« `ODER` »`San Francisco`« `UND John` in folgende Liste aufgesplittet (»`San Jose`«, `ODER`, »`San Francisco`«, `UND`, `John`).

Nachdem nun alle unsere Suchteile in einer Liste stehen, besteht die eigentliche Arbeit darin, diese Liste durchzugehen, wenn nötig, nach der Adresse zu suchen und die logischen Ausdrücke abzuarbeiten. All dies wird in der großen `while`-Schleife ausgeführt, die in Zeile 9 beginnt. Die Schleife verwendet eine Platzhaltervariable `$i` zum Festhalten der aktuellen Position in dem Muster und geht das Muster bis zum Ende durch. Innerhalb der `while`-Schleife prüft die Variable `$matched` ständig, ob ein bestimmter Teil des Musters mit dem Datensatz übereinstimmt. Begonnen wird mit einer `0`, ***falsch***, für keine Übereinstimmung.

Innerhalb der `while`-Schleife starten wir in Zeile 10, wo wir der Variablen `$things` auf den aktuellen Teil des zu untersuchenden Musters zuweisen, einfach um uns bei weiteren Zugriffen Tipparbeit zu ersparen. Anschließend folgen vier größere Tests:

- Ist der aktuelle Teil ein ODER, befinden wir uns mitten zwischen zwei Tests, von denen einer bereits

abgeschlossen wurde und entweder ***wahr*** oder ***falsch*** (je nach Wert von `$matched`) zurückgeliefert hat. Ist `$matched` ***wahr***, wurde für die linke Seite eine Übereinstimmung gefunden, und es gibt keinen Grund, die Untersuchung auf die rechte Seite auszudehnen (genau, es handelt sich um ein Ausschluß-ODER). Gibt es für die linke Seite keine Übereinstimmung, ist der Wert der `$matched`-Variablen 0, und wir müssen die rechte Seite ebenfalls überprüfen. Dieser Schritt erfolgt in Zeile 13. Dort wird die Subroutine `&isitthere()` aufgerufen, die die eigentliche Suche nach dem Suchmuster durchführt und der als Argumente die rechte Seite des ODER (der nächste Teil in dem `@things`-Array) und der Datensatz selbst (`%rec`) übergeben werden.
Unabhängig davon, ob es eine Übereinstimmung gegeben hat oder nicht, verarbeitet dieser Test das ODER und das Muster rechts davon. Deshalb können wir in dem `@things`-Array zwei Elemente vorrücken. Diese Aufgabe übernimmt Zeile 15, die den Zähler `$i` um zwei inkrementiert.

- Ist der aktuelle Teil ein UND, begeben wir uns zum Test in Zeile 17. Dieser Abschnitt ähnelt bis auf eine Ausnahme dem Abschnitt mit ODER: Der Ausschluß erfolgt auf anderem Wege. Erinnern wir uns: Gegeben sei ein Test für `x UND y`. Wenn `x` ***falsch*** ist, ist der gesamte Ausdruck falsch. Wenn aber `x` ***wahr*** ist, muss immer noch `y` überprüft werden, um zu sehen, ob diese Variable ebenfalls ***wahr*** ist. Und so funktioniert auch dieser Test. Wenn die Variable `$matched` ***wahr*** ergibt, dann ist damit die linke Seite des UND-Ausdrucks ***wahr***, und wir rufen `&isitthere()` auf, um die rechte Seite zu testen. Andernfalls machen wir einfach gar nichts und überspringen die nächsten zwei Elemente, sowohl das UND als auch die rechte Seite des UND (`$i+=2`, Zeile 21), um danach fortzufahren.
- In Zeile 23 haben wir UND und ODER bereits abgehandelt, deshalb muss das gesuchte Teil ein konkreter Suchbegriff sein. Dabei kann es sich um einen einzelnen Suchbegriff oder um mehrere Suchbegriffe vereint zu einem String handeln. Was es ist, spielt letztlich keine Rolle, da beide Fälle einzeln untersucht werden können. Wir müssen jedoch nur dann nach dem Suchbegriff suchen, wenn wir noch keine Übereinstimmung gefunden haben. (Zur Erinnerung: Mehrere Suchläufe werden als ODER-Tests behandelt. Wenn also eine Übereinstimmung gefunden wurde, war es das schon.) In Zeile 23 rufen wir die Suchroutine `&isitthere()` daher nur dann auf, wenn wir bisher keine Übereinstimmung gefunden haben.
- Der letzte Fall tritt ein, wenn in `$thing` ein richtiger Suchschlüssel steht und `$matched` ***wahr*** ist. Wir müssen hier nichts machen, da bereits eine Übereinstimmung gefunden wurde. Deshalb muss nur die Position in dem Muster um eins inkrementiert und die Schleife erneut gestartet werden.

Wenn Sie mir soweit folgen konnten, haben Sie den schwierigsten Teil des Skripts bereits hinter sich. Sollten Sie noch Schwierigkeiten haben, versuchen Sie es einfach mal selbst mit verschiedenen Suchmustern: mit einzelnen Suchelementen, Elementen, die durch UND oder ODER getrennt sind, und Mustern mit mehreren Suchschlüsseln. Kontrollieren Sie die Werte von `$i` und `$matched` beim Schleifendurchlauf (wenn Sie mit dem Perl-Debugger bereits umgehen können, ist dies recht einfach, aber Sie können es auch auf Papier von Hand machen).

Was also passiert in der mysteriösen `&isitthere()`-Subroutine, die in der großen `while`-Schleife immer wieder aufgerufen wird? Nachdem Suchbegriff und Datensatz gegeben sind, findet hier die eigentliche Suche statt. Ich werde darauf verzichten, Ihnen hier den Inhalt von `&isitthere()` vorzustellen, Sie finden diese Subroutine in voller Länge als Teil des Gesamtcodes in Listing 14.3. Ich möchte Sie jedoch darauf hinweisen, dass die Subroutine lediglich den Inhalt des Adressen-Hash durchläuft und mit Hilfe eines regulären Ausdrucks den Suchbegriff mit jeder Zeile vergleicht. Bei einer Übereinstimmung liefert die Subroutine 1 zurück, bei keiner Übereinstimmung 0.

Im letzten Teil der Subroutine sind alle Teile des Suchbegriffs verarbeitet, einige Suchläufe wurden durchgeführt, und wir wissen, ob der Suchbegriff mit dem Datensatz übereinstimmt oder nicht. Die Zeilen 30 bis 33 testen, ob eine Übereinstimmung gefunden wurde. Wenn ja, setzen wir die Variable `$bigmatch` (mindestens eine Adresse wurde als Übereinstimmung gefunden) und rufen `&print_addr()` auf, um diese Adresse auszudrucken.

Den Datensatz ausdrucken

Von hier an wird es einfach. Die letzte Subroutine in der Datei wird nur aufgerufen, wenn eine Übereinstimmung gefunden wurde. Die Subroutine `&print_addr()` durchläuft einfach den Datensatz-Hash und gibt die Werte aus, um den Adreßdatensatz anzuzeigen.

```
sub print_addr {
    my %record = @_;
    print "**********************\n";
    foreach my $key (qw(Name Telefon Fax Adresse EMail URL)) {
        if (defined($record{$key})) {
            print "$record{$key}\n";
        }
```

```
    }
}
```

Interessant an dieser Subroutine ist allein die Liste der Schlüssel in der `foreach`- Schleife. Ich habe die Schlüssel in dieser Reihenfolge aufgeführt (und mit der Funktion `qw` in Anführungszeichen gesetzt), so dass die Ausgabe in einer bestimmten Reihenfolge erfolgt. Da Hashes in einer intern festgelegten Reihenfolge gespeichert werden, können wir nur so dafür sorgen, dass die Daten in der korrekten Reihenfolge ausgegeben werden. Nebenbei wird dadurch auch erreicht, dass nur die Zeilen ausgegeben werden, die tatsächlich verfügbar waren - der Aufruf von `defined` innerhalb der `foreach`-Schleife stellt sicher, dass nur die Felder ausgegeben werden, die auch tatsächlich im Datensatz existieren.

Der Code

Alles OK? Nein? Manchmal ist es eine Hilfe, wenn man den ganzen Code auf einmal sieht. Listing 14.3 enthält den vollständigen Code für ***adressen.pl***. Wenn Sie den Quelltext von der Website zu diesem Buch (unter `http://www.typerl.com`) heruntergeladen haben, werden Sie feststellen, dass der Code wesentlich mehr Kommentare enthält, um deutlich zu machen, was gerade passiert.

Wie ich bereits gestern in dem Abschnitt zu `my`-Variablen angedeutet habe, können einige Perl-Versionen Schwierigkeiten mit der Verwendung der `my`-Variablen im Skript und den `foreach`-Schleifen haben. Sie können das Problem umgehen, indem Sie die `foreach`-Variable einfach wie folgt vor ihrem Gebrauch deklarieren:

```
my $key = 0;

foreach $key (qw(Name Telefon Fax Adresse EMail URL)) { ...
```

Listing 14.3: Der Code für adressen.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my $bigmatch = 0;               # wurde etwas gefunden?
5:  my %rec = ();                   # zu durchsuchender Datensatz
6:  my $search = '';                # Suchmuster
7:
8:  $search = &getpattern();        # Eingabeaufforderung für das Muster
9:
10: while () {                      # durchsucht die Adressdatei
11:     %rec = &read_addr();
12:     if (%rec) {         # Datensatz gefunden
13:         &perform_search($search, %rec);
14:     } else {            # Ende der Adressdatei, Aufräumarbeiten
15:         if (!$bigmatch) {
16:             print "Nichts gefunden.\n";
17:         } else { print "**********************\n"; }
18:         last;           # Verlassen, wir sind fertig
19:     }
20: }
21:
22: sub getpattern {
23:     my $in = '';  # Eingabe
24:     print 'Wonach soll gesucht werden? ';
25:     chomp($in = <STDIN>);
26:     return $in;
27: }
28:
29: sub read_addr {
30:     my %curr = ();              # aktueller Datensatz
31:     my $key = '';               # temp. Schlüssel
32:     my $val = '';               # temp. Wert
33:
34:     while (<>) {                # bricht bei EOF ab
```

```
35:         chomp;
36:         if ($_ ne '---') {      # Datensatz-Trennzeichen
37:             ($key, $val) = split(/: /,$_,2);
38:             $curr{$key} = $val;
39:         }
40:         else { last; }
41:     }
42:     return %curr;
43: }
44:
45: sub perform_search {
46:     my ($str, %rec) = @_;
47:     my $matched = 0;            # gesamte Übereinstimmung
48:     my $i = 0;                  # Position innerhalb des Suchmusters
49:     my $thing = '';             # temporäres Wort
50:
51:     my @things = $str =~ /("[^"]+"|\S+)/g;  # in Suchelemente aufspalten
52:
53:     while ($i <= $#things) {
54:         $thing = $things[$i];   # Suchelement, UND oder ODER
55:         if ($thing eq 'ODER' || $thing eq 'oder') { # ODER-Fall
56:             if (!$matched) {    # noch keine Übereinstimmung, nächstes
                                    # Element
57:                 $matched = &isitthere($things[$i+1], %rec);
58:             }
59:             $i += 2;            # ODER überspringen und nächstes Element
60:         }
61:         elsif ($thing eq 'UND' || $thing eq 'und') { # UND-Fall
62:             if ($matched) {     # Übereinstimmung gefunden, andere Seite
                                    # prüfen
63:                 $matched = &isitthere($things[$i+1], %rec);
64:             }
65:             $i += 2;            # UND überspringen und nächstes Element
66:         }
67:         elsif (!$matched) {     # noch keine Übereinstimmung
68:             $matched = &isitthere($thing, %rec);
69:             $i++;               # weiter!
70:         }
71:         else { $i++; }          # $match wurde gefunden, weiter zum
                                    # nächsten Element
72:     }
73:
74:     if ($matched) {             # alle Schlüssel durchgearbeitet, gab es
                                    # eine Übereinstimmung?
75:         $bigmatch = 1;  # Ja, wir haben etwas gefunden
76:         print_addr(%rec);       # Datensatz ausgeben
77:     }
78: }
79:
80: sub isitthere {                 # einfacher Test
81:     my ($pat, %rec) = @_;
82:     foreach my $line (values %rec) {
83:         if ($line =~ /$pat/) {
84:             return 1;
85:         }
86:     }
87:     return 0;
88: }
89:
90: sub print_addr {
91:     my %record = @_;
92:     print "**********************\n";
93:     foreach my $key (qw(Name Telefon Fax Adresse EMail URL)) {
94:         if (defined($record{$key})) {
95:             print "$record{$key}\n";
96:         }
97:     }
98: }
```

Ein Prozessor für Log-Dateien von Websites (weblog.pl)

Das zweite Beispielskript übernimmt eine Protokolldatei, wie Sie von Webservern erzeugt wird, und erstellt aus den

darin enthaltenen Daten eine Statistik. Die meisten Webserver legen Dateien dieser Art an, mit denen sie unter anderem verfolgen, wie viele Besucher (»Hits«) es für eine Website gab, welche Dateien angefordert wurden und von welchen Sites die Anfragen kamen.

Im Web gibt es bereits viele Programme zur Analyse von Protokolldateien (einschließlich der Programme, die üblicherweise für Ihren Webserver zur Verfügung stehen). Deshalb ist dieses Beispiel auch nicht gerade neu. Die Statistiken, die damit erstellt werden, sind relativ einfach. Sie können dieses Skript jedoch ohne weiteres umschreiben, so dass es jede beliebige von Ihnen gewünschte Information mit aufnimmt. Es ist eine gute Ausgangsbasis zum Verarbeiten von Webprotokollen beziehungsweise ein gutes Muster, wenn Protokolldateien anderer Programme zu verarbeiten sind.

Funktionsweise

Das Skript ***weblog.pl*** wird mit einem Argument, der Protokolldatei, aufgerufen. Auf vielen Webservern werden diese Dateien `access_log` genannt und folgen dem sogenannten ***common log-Format***. Das Skript arbeitet erst einmal eine Weile vor sich hin. Dabei gibt es die Datumsstempel der bearbeiteten Protokolle aus, damit Sie erkennen, dass das Skript noch arbeitet. Anschließend werden einige Ergebnisse ausgegeben. Sehen Sie im folgenden, wie eine Ausgabe aussehen kann (das untenstehende Beispiel ist von den Protokollen meines eigenen Webservers `www.lne.com`):

```
% weblog.pl access_log
Log-Dateien verarbeiten....
Verarbeite 09/Apr/1998
Verarbeite 10/Apr/1998
Verarbeite 11/Apr/1998
Verarbeite 12/Apr/1998
Auswertung der Log-Datei:
Gesamtzahl der Treffer: 55789
Gesamtzahl der fehlgeschlagenen Treffer: 1803 (3.23%)
(erfolgreiche) HTML-Dateien: 18264 (33.83%)
Anzahl der Hosts: 5911
Anzahl der Domänen: 2121
Die beliebtesten Dateien:
  /Web/index.html (2456 Treffer)
  /lemay/index.html (1711 Treffer)
  /Web/Title.gif (1685 Treffer)
  /Web/HTML3.2/3.2thm.gif (1669 Treffer)
  /Web/JavaProf/javaprof_thm.gif (1662 Treffer)
Die beliebtesten Hosts:
  202.185.174.4 (487 Treffer)
  vader.integrinautics.com (440 Treffer)
  linea15.secsa.podernet.com.mx (437 Treffer)
  lobby.itmin.com (284 Treffer)
  pyx.net (256 Treffer)
Die beliebtesten Domänen:
  mindspring.com (3160 Treffer)
  aol.com (1808 Treffer)
  uu.net (792 Treffer)
  grid.net (684 Treffer)
  compuserve.com (565 Treffer)
```

Die Ausgabe hier zeigt, um Platz zu sparen, nur die ersten fünf Dateien, Hosts und Domänen. Sie können das Skript allerdings so konfigurieren, dass es eine beliebige Anzahl dieser Statistiken ausgibt.

Der Unterschied zwischen einem Host und einer Domäne mag vielleicht nicht direkt ersichtlich sein. Ein Host ist der vollständige Hostname des Systems, das auf den Webserver zugegriffen hat, und kann dynamisch zugewiesene Adressen und Proxy- Server mit umfassen. So unterscheidet sich der Host `dialup124.servers.foo.com` vom Host `dialup567.servers.foo.com`. Die Domäne andererseits bezeichnet eine größere Gruppe von mehreren Hosts und besteht in der Regel aus zwei oder drei Teilen. So ist `foo.com` ein Domäne, ebenso wie `aol.com` oder `demon.co.uk`. Die Domänen-Listings neigen dazu, einzelne Host-Einträge in größere Gruppen zusammenzufassen - alle Hosts im Wirkungsbereich von `aol.com` erscheinen in der Domänenliste als Hit von `aol.com`.

Beachten Sie, dass es sich bei einem einfachen Hit um eine HTML-Seite, ein Bild, ein eingereichtes Formular oder eine andere beliebige Datei handeln kann. Es gibt normalerweise wesentlich mehr Hits als tatsächliche Zugriffe auf eine Seite (page views). Dieses Skript macht den Unterschied deutlich, indem HTML-Anfragen getrennt von der

Gesamtzahl der Hits gezählt werden.

Wie sieht ein Webprotokoll aus

Da das Skript ***weblog.pl*** Webprotokolle verarbeitet, ist es von Vorteil, wenn man weiß, wie diese Protokolldateien aussehen. Webprotokolle speichern einen Treffer pro Zeile und jede Zeile in dem sogenannten allgemeinen Protokollformat (allgemein, da das Format mehreren Webservern gemeinsam ist). Die meisten Webserver erzeugen ihre Protokolldateien in diesem Format beziehungsweise können entsprechend konfiguriert werden (viele Server verwenden eine erweiterte Form des allgemeinen Protokollformats, das mehr Informationen enthält). Eine Zeile einer Protokolldatei im allgemeinen Protokollformat kann folgendermaßen aussehen (diese Zeile steht aus drucktechnischen Gründen in zwei Zeilen, in Wirklichkeit steht sie jedoch nur in einer Zeile):

```
proxy2bh.powerup.com.au - - [03/Apr/1998:00:09:02 -0800]
"GET /lemay/ HTTP/1.0" 200 4621
```

Die einzelnen Elemente jeder Zeile der Protokolldatei sind:

- Der Hostname, der auf den Server zugreift (hier `proxy2bh.powerup.com.au`).
- Der Benutzername der Person, die auf die Seite zugreift, ermittelt durch `ident` (ein Unix-Programm zum Identifizieren von Benutzern) oder durch den Benutzer Ihrer Site selbst bekanntgemacht. Diese zwei Teile erscheinen normalerweise als zwei Bindestriche (`- -`), wenn der Benutzername nicht ermittelt werden kann.
- Datum und Zeit des Abrufs in eckigen Klammern.
- Innerhalb von Anführungszeichen die Aktion, die den Hit ausgelöst hat (in der Regel eine Webserver-Aktion): `GET` steht für das Anfordern einer Datei oder das Abrufen eines Formulars, `POST` für das Einreichen eines Formulars auf anderem Wege und `HEAD` für das Auslesen von Header-Informationen zu einer Datei.
- Nach der Aktion folgt der angeforderte Dateiname (oder das Verzeichnis), hier `/lemay/`.
- Die Versionsnummer des Protokolls, hier `HTTP/1.0`.
- Der Rückgabecode für den Treffer: 200 steht für erfolgreich, 404 für »nicht gefunden« und so weiter.
- Die Anzahl der übertragenen Bytes.

Natürlich sind nicht all diese Elemente einer Protokolldatei von Interesse für ein Skript zum Aufbau einer Statistik, und viele werden Sie erst verstehen, wenn Sie wissen, wie Webserver funktionieren. Ein paar jedoch, wie der Host, das Datum, der Dateiname und der Rückgabecode können extrahiert und für jede Zeile der Datei verarbeitet werden.

Das Skript erstellen

Der logische Ablauf dieses Skripts ist leichter zu verfolgen als für das Skript ***adressen.pl***. Es gibt eigentlich nur zwei größere Schritte - die Verarbeitung des Protokolls und die Erzeugung der Statistik. Dabei werden wir von einer Reihe von Subroutinen unterstützt.

Um genau zu sein, der gesamte Code für dieses Skript ist in Subroutinen untergebracht. Der Rumpf des Codes besteht aus einer Reihe von globalen Variablen und dem Aufruf von zwei Subroutinen: `&process_log()` und `&print_results()`.

Die globalen Variablen dienen dazu, die verschiedenen statistischen Daten und sonstigen Informationen über Teile der Protokolldatei zu speichern. Da viele dieser statistischen Daten Hashes sind, wäre die Verwendung von lokalen Variablen und das Weiterreichen der Daten zu kompliziert. In diesem Fall ist die Verwaltung der Variablen einfacher, wenn sie globaler Art sind. Zu den globalen Daten, die wir verfolgen, gehören:

- Die Anzahl der Treffer, Anzahl der mißlungenen Treffer und Anzahl der Treffer für HTML-Seiten.
- Ein Hash, um die verschiedenen Hostnamen und die Häufigkeit der Hosts in dem Protokoll zu speichern.
- Ein Hash, der dasselbe für die verschiedenen Dateien in dem Protokoll macht.

Darüber hinaus gibt es noch zwei weitere globale Variablen:

- Die Variable `$topthings` speichert eine Zahl, die angibt, wie viele Einträge Sie für die »beliebtesten« Teile der Statistik ausgeben wollen. In meinem Beispiel habe ich den Wert für `$topthings` auf 5 gesetzt und damit eine nette kurze Ausgabe erzeugt. Wenn Sie den Wert auf 20 setzen, werden die ersten 20 Dateien,

Hosts und Domänen ausgegeben.
- Die Variable `$default` sollte auf die Standard-HTML-Seite Ihres Webservers gesetzt werden. Meistens heißt diese `index.html` oder `Home.html`. Dies ist die Datei, die als Hauptdatei für ein Verzeichnis dient, wenn der Benutzer keine bestimmte Datei anfordert. Normalerweise lautet der Name `index.html`.

Diese zwei Variablen legen fest, wie sich das Skript selbst verhält. Wir hätten diese Variablen auch tief im Innern des Programms verbergen können. Doch dadurch, dass wir sie hier direkt am Anfang aufgeführt haben, können Sie oder alle anderen, die das Skript nutzen, das Gesamtverhalten des Skripts ändern, ohne nach den zu ändernden Variablen lange suchen zu müssen. Diese Verfahrensweise gehört zum »guten Programmierstil«, unabhängig davon, welche Programmiersprache Sie verwenden.

Das Protokoll verarbeiten

Der erste Teil des ***weblog.pl***-Skripts besteht aus der Subroutine `&process_log()`, die die einzelnen Zeilen der Protokolldatei durchläuft und die statistischen Daten aus der Zeile speichert. Ich werde Ihnen nicht jede Zeile dieser Subroutine erläutern, aber ich zeige Ihnen die wichtigsten Teile. Den vollständigen Code können Sie in Listing 14.7 am Ende dieses Kapitels einsehen.

Das Kernstück der Subroutine `&process_log()` ist eine weitere `while (<>)`-Schleife, um die Zeilen der Eingabe einzeln einzulesen. Im Gegensatz zu ***adressen.pl*** liest dieses Skript die Datei von Anfang bis Ende ein.

Für die Verarbeitung der Zeile splitten wir sie zuerst in ihre Bestandteile und speichern diese Teile in einem Hash, dessen Schlüssel der Teilename ist (`'site'`, `'file'` und so weiter). Für das Aufsplitten gibt es eine separate Subroutine namens `&splitline()`, die in Listing 14.4 zu sehen ist.

Listing 14.4: Die Subroutine &splitline()

```
1:  sub splitline {
2:      my $in = $_[0];
3:      my %line = ();
4:      if ($in =~ /^([^\s]+)\s           # Site
5:                 ([\w-]+\s[\w-]+)\s     # Benutzer
6:                 \[([^\]]+)\]\s         # Datum
7:                  \"(\w+)\s             # Protokoll
8:                 (\/[^\s]*)\s           # Datei
9:                 ([^"]+)\"\s            # HTTP-Version
10:                (\d{3})\s              # Rückgabe-Code
11:                ([\d-]+)               # übertragene Bytes
12:     /x) {
13:        $line{'site'} = $1;
14:        $line{'date'} = $3;
15:        $line{'file'} = $5;
16:        $line{'code'} = $7;
17:        return %line;
18:     } else { return (); }
19: }
```

Das erste, was Ihnen bei dieser Subroutine wahrscheinlich ins Auge fällt, ist der nicht enden wollende reguläre Ausdruck in der Mitte von Zeile 4 bis Zeile 11. Er ist so häßlich, dass er sechs Zeilen belegt! Und Kommentare erforderlich macht! Dieser reguläre Ausdruck hat die Form erweiterter regulärer Ausdrücke. Ich habe diese Ausdrücke bereits in dem Abschnitt »Vertiefung« in Kapitel 5, »Mit Hashes arbeiten«, eingehend beschrieben. Hier eine kurze Zusammenfassung: Angenommen Sie haben einen besonders ekligen regulären Ausdruck wie den in diesem Beispiel (aus drucktechnischen Gründen steht er auf zwei Zeilen, da er nicht in eine Zeile paßt!):

```
if ($in =~ /^([^\s]+)\s([\w-]+\s[\w-]+)\s\[([^\]]+)\]\s\"(\w+)
\(\/[^\s]*)\s([^"]+)\"\s(\d{3})\s([\d-]+)/)
```

Sehr wahrscheinlich brauchen Sie entweder eine unendliche Geduld oder sehr starke Beruhigungsmittel, um diesen Ausdruck aufzuschlüsseln und zu verstehen. Und das Debuggen dieses Ausdrucks ist auch nicht sehr lustig. Wenn Sie jedoch an das Ende des Ausdrucks die Option `/x` setzen (wie hier in Zeile 12), können Sie den regulären Ausdruck aufsplitten und auf verschiedene Zeilen verteilen, die Sie außerdem noch mit Kommentaren versehen können. Alle Leerzeichen darin werden ignoriert. Wenn Sie im Text eine Übereinstimmung auf Leerzeichen suchen wollen, müssen Sie `\s` verwenden. Die Option `/x` erleichtert lediglich das Lesen und Debuggen des regulären

Ausdrucks.

Unser regulärer Ausdruck geht von dem allgemeinen Protokollformat (common log format) aus, das ich bereits oben beschrieben habe:

- Zeile 4 ermittelt den Namen der Site (Host). Die Site steht immer am Anfang der Zeile und besteht aus einer Reihe von Zeichen (keine Whitespace-Zeichen) gefolgt von einem Leerzeichen (in diesem Fall `\s`, damit sich das erweiterte Muster anwenden läßt).
- Zeile 5 prüft die Benutzerfelder (zwei). Die Benutzer bestehen aus einem oder mehreren alphanumerischen Zeichen oder einem Gedankenstrich, getrennt durch und gefolgt von einem Whitespace-Zeichen. Beachten Sie die Gedankenstriche innerhalb der Zeichenklassen; in der `\w`-Klasse sind sie nicht mit eingeschlossen.
- Zeile 6 ermittelt das Datum. Dabei handelt es sich um ein oder mehrere Zeichen oder Whitespaces zwischen eckigen Klammern (`[]`).
- Zeile 7 ermittelt das Protokoll (`GET`, `HEAD` usw.). Der String beginnt mit einem Anführungszeichen gefolgt von einem oder mehreren Zeichen (das schließende Anführungszeichen steht nach der HTTP-Version in Zeile 9).
- Zeile 8 ermittelt die Datei. Diese Information beginnt immer mit einem Slash (`/`) gefolgt von einer 0 oder mehreren anderen Zeichen und endet mit einem Whitespace-Zeichen.
- Zeile 9 ermittelt die HTTP-Version, die sich in den übriggebliebenen Zeichen vor dem abschließenden Anführungszeichen verbirgt.
- Zeile 10 ermittelt den Rückgabecode, der immer drei Ziffern umfaßt, gefolgt von einem Whitespace-Zeichen (es wäre weniger speziell, hier nur `\d+` zu verwenden, aber so habe ich eine Gelegenheit, Ihnen die Verwendung des Musters `{3}` zu zeigen).
- Zeile 11 beendet die Prüfung mit der Anzahl der übertragenen Bytes. Dabei kann es sich um eine beliebige Anzahl von Ziffern handeln. Wurden keine Bytes übertragen, weil zum Beispiel der Treffer ein Fehler war, besteht dieses Feld aus einem Gedankenstrich. Auch diesen findet das Muster.

Jedes Element dieses regulären Ausdrucks wird in einem geklammerten Ausdruck (und einer Übereinstimmungsvariablen) gespeichert, wobei die zusätzlichen Klammern oder Anführungszeichen entfernt werden. Sobald das Pattern Matching beendet ist, können wir die verschiedenen übereinstimmenden Teile in einem Hash ablegen. Beachten Sie, dass wir nur die Hälfte der Übereinstimmungen in dem Hash ablegen. Wir müssen nur das abspeichern, was wir am Ende auch nutzen wollen. Wenn Sie aber dieses Beispiel erweitern wollen, um Statistiken über weitere Teile der Treffer zu erstellen, müssen Sie lediglich Zeilen einfügen, die diese Übereinstimmungen dem Hash hinzufügen. Sie brauchen den regulären Ausdruck nicht zu verändern, um mehr Informationen zu erhalten.

Nachdem die Zeile jetzt in ihre einzelnen Elemente zerlegt ist, kehren wir von der Subroutine `&splitline()` zurück zu der Hauptroutine `&process_log()`. Diese Routine überprüft als nächstes alle fehlgeschlagenen Treffer. Wenn eine Zeile im Webprotokoll nicht dem Muster entspricht - was bei einigen der Fall ist -, liefert die Subroutine `&splitline()` Null zurück. Dieses Ergebnis wird als fehlgeschlagener Treffer interpretiert, der dann zu der Zahl der fehlgeschlagenen Treffer addiert wird. Anschließend wird der Rest der Schleife übersprungen, um mit der nächsten Zeile fortzufahren:

```
if (!%hit) {  # mißgestaltete Zeile im Webprotokoll
   $failhits++;
    next;
}
```

Der nächste Schritt im Skript ist ein Entgegenkommen an all diejenigen, die das Skript ausführen. Die Verarbeitung einer beliebig großen Protokolldatei kann lange dauern, und manchmal ist es schwer zu sagen, ob Perl noch die Protokolldatei bearbeitet oder ob sich das System aufgehängt hat und keinen Wert mehr liefern wird. Dieser Teil des Skripts gibt eine Nachricht mit dem Datum der Zeilen aus, die gerade verarbeitet werden. Jedesmal, wenn die Treffer eines Tages vollständig bearbeitet worden sind, erscheint eine neue Nachricht, die das Fortschreiten von Perl in der Datei anzeigt:

```
$dateshort = &getday($hit{'date'});
if ($currdate ne $dateshort) {
    print "Verarbeite $dateshort\n";
    $currdate = $dateshort;
}
```

In diesem Fragment ist `&getday()` eine kurze Subroutine, die den Monat und den Tag aus dem Datumsfeld

ausliest. Dabei wird ein Muster verwendet, so dass Monat und Tag mit dem gerade verarbeiteten Datum verglichen werden können (auf den Ausdruck des Codes für `&getday()` verzichte ich, da Sie ihn in dem vollständigen Listing am Ende des Kapitels finden). Sind sie unterschiedlich, wird eine Nachricht ausgegeben und die Variable `$currdate` aktualisiert.

Zusätzlich zu den Zeilen in der Protokolldatei, die nicht dem Protokollformat entsprechen, werden auch jene Zeilen als fehlgeschlagene Treffer bezeichnet, die zwar dem Muster entsprechen, aber nicht dazu führten, dass wirklich eine Datei zurückgeliefert wurde (falsche URL-Angaben oder Dateien, die verschoben wurden, lösen diese Art von Treffer aus). Diese Treffer werden in der Protokolldatei mit einem Fehlercode aufgezeichnet, der mit 4 beginnt (vielleicht ist Ihnen bereits der Error 404 im Web aufgefallen). Der Rückgabecode gehört mit zu den Elementen der Zeile, die wir gespeichert hatten. Deshalb ist die Überprüfung ein einfacher Musterabgleich:

```
if ($hit{'code'} =~ /^4/) { # 404, 403, etc. (Fehler)
    $failhits++;
```

Der `else`-Teil dieser `if`-Anweisung betrifft alle anderen Treffer - gemeint sind damit alle erfolgreichen Treffer, die tatsächlich HTML-Dateien oder Grafiken zurückgeliefert haben. Diese Treffer haben einen Rückgabecode von 200 oder 304.

```
} elsif ($hit{'code'} =~ /200|304/) {   # behandelt nur erfolgreiche Treffer
```

Webserver sind so eingerichtet, dass sie eine Standarddatei, in der Regel `index.html`, zurückliefern, wenn eine URL angefordert wird, die einem Verzeichnisnamen entspricht. Das bedeutet, dass eine Anfrage nach `/web/` und eine Anfrage nach `/web/ index.html` sich auf die gleiche Datei beziehen, jedoch in der Protokolldatei als unterschiedliche Einträge erscheinen. Um Verzeichnisse und Standarddateien als einen Eintrag zu behandeln, gibt es einige Zeilen, die prüfen, ob die angeforderte Datei mit einem Slash endet, und wenn ja, dafür sorgen, dass der Standarddateiname hinten angehängt wird. Die Standarddatei, wie ich bereits oben erwähnt habe, wird durch die Variable `$default` definiert:

```
if ($hit{'file'} =~ /\/$/) { # slashes werden zu $default
    $hit{'file'} .= $default;
}
```

Nachdem wir dies erledigt haben, können wir die Verarbeitung damit abschließen, dass wir die Variable `$htmlhits` inkrementieren, wenn es sich bei der Datei um eine HTML-Datei handelt, und die Hashes für die Site und für die Datei aktualisieren:

```
if ($hit{'file'} =~ /\.html?$/) { # .htm oder .html
    $htmlhits++;
}
$hosts{ $hit{'site'} }++;
$files{ $hit{'file'} }++;
```

Damit sind wir ans Ende der `while`-Schleife angelangt, die mit der nächsten Zeile in der Datei von vorne beginnt. Die Schleife wird so lange durchlaufen, bis alle Zeilen verarbeitet sind. Danach gehen wir zum Ausgeben der Ergebnisse dieses Skripts über.

Die Ergebnisse ausgeben

Die Subroutine `&process_log()` verarbeitet die Protokolldatei zeilenweise und ruft zu ihrer Unterstützung die Subroutinen `&splitline()` und `&getday()` auf. Der zweite Teil unseres ***weblog.pl***-Skripts besteht aus der Subroutine `&print_results()`, die ebenfalls auf einige weitere Subroutinen zur Unterstützung zurückgreift. Der größte Teil der Subroutine besteht jedoch aus einer Reihe von `print`-Anweisungen, um die verschiedenen Statistiken auszugeben.

Die ersten Zeilen geben die Gesamtzahl der Treffer, Gesamtzahl der fehlgeschlagenen Treffer und Gesamtzahl der HTML-Treffer aus. Die letzteren werden auch als Prozent der gesamten Treffer aufgeschlüsselt, wobei die HTML-Treffer sich nur auf die Gesamtsumme der erfolgreichen Treffer beziehen. Wir erhalten diese Werte mit ein wenig Mathematik und der Anweisung `printf`:

```
print "Auswertung der Log-Datei:\n";
```

```
print "Gesamtzahl der Treffer: $totalhits\n";
print "Gesamtzahl der fehlgeschlagenen Treffer: $failhits (";
printf('%.2f', $failhits / $totalhits * 100);
print "%)\n";
print "(erfolgreiche) HTML-Dateien: $htmlhits (";
printf('%.2f', $htmlhits / ($totalhits - $failhits) * 100);
print "%)\n";
```

Als nächstes kommt die Gesamtzahl der Hosts. Diesen Wert erhalten wir, indem wir die Schlüssel aus dem Hash `%hosts` herausziehen und in einer Liste ablegen. Anschließend werten wir diese Liste in einem skalaren Kontext aus (mit Hilfe der Funktion `scalar`).

```
print 'Anzahl der Hosts: ';
print scalar(keys %hosts);
print "\n";
```

Um die Anzahl der Domänen zu ermitteln, müssen wir den Hash `%hosts` verarbeiten, um die Hosts ihren Domänen zuzuordnen, und einen neuen Hash (`%domains`) einrichten, der die Anzahl der Treffer für die Domänen aufnimmt. Dazu verwenden wir eine Subroutine namens `&getdomains()`, die ich im nächsten Abschnitt besprechen werde. Gehen wir einfach davon aus, dass wir unseren Hash `%domains` bereits haben. Wir können auf die Schlüssel dieses Hash den gleichen Trick mit `scalar` anwenden, um die Anzahl der Domänen zu ermitteln:

```
my %domains = &getdomains(keys %hosts);
print 'Anzahl der Domänen: ';
print scalar(keys %domains);
print "\n";
```

Als letztes sind die beliebtesten Dateien, Hosts und Domänen auszudrucken. Für die Ermittlung dieser Werte gibt es die Subroutine `&gettop()`, die jeden Hash nach seinen Werten sortiert (die Häufigkeit, mit der jede Datei, Host oder Domäne in einem Treffer vorkam) und dann einen Array deskriptiver Strings einrichtet mit den Schlüsseln und Werten im Hash. Das Array enthält nur die 5 oder 10 (oder was auch immer Sie als Wert in `$topthings` ablegen) beliebtesten Dateien, Hosts oder Domänen. Doch gleich mehr zu der Subroutine `&gettops()`.

Jedes dieser Arrays wird zum Schluß ausgegeben. Hier sehen Sie den Code für die Ausgabe der Dateien:

```
print "Die beliebtesten Dateien: \n";
    foreach my $file (&gettop(%files)) {
        print "  $file\n";
    }
```

Die Subroutine &getdomains()

Noch sind wir nicht fertig. Es fehlen uns noch die Hilfsroutinen zur Ausgabe der Statistiken: `&getdomains()`, um die Domänen aus dem `%hosts`-Hash zu extrahieren und die Statistik neu zu berechnen, und `&gettop()`, um einen Hash von Schlüsseln und Frequenzwerten zu übernehmen und die beliebtesten Elemente zurückzuliefern. Die Subroutine `&getdomains()` finden Sie in Listing 14.5.

Listing 14.5: Die Subroutine &getdomains()

```
1:  sub getdomains {
2:      my %domains = ();
3:      my ($sd,$d,$tld);       # sekundäre Domäne, Domäne, oberste Domäne
4:      foreach my $host (@_) {
5:          my $dom = '';
6:          if($host =~ /(([^.]+)\.)?([^.]+)\.([^.]+)$/ ) {
7:              if (!defined($1)) { # nur zwei Domänen (i.e. aol.com)
8:                  ($d,$tld) = ($3, $4);
9:              } else {            # eine gewöhnliche Domäne x.y.com etc
10:                 ($sd, $d, $tld) = ($2, $3, $4);
11:             }
12:             if ($tld =~ /\D+/) { # ignoriert reine IP-Zahlen
13:                 if ($tld =~ /com|edu|net|gov|mil|org$/i) { # US TLDs
14:                     $dom = "$d.$tld";
15:                 } else { $dom = "$sd.$d.$tld"; }
16:                 $domains{$dom} += $hosts{$host};
```

```
17:             }
18:         } else { print "Fehlerhaft: $host\n"; }
19:     }
20:     return %domains;
21: }
```

Diese Subroutine ist nicht so kompliziert, wie sie aussieht. Ich gehe dabei von ein paar Grundvoraussetzungen für den Hostnamen aus: dass zum Beispiel jeder Hostname aus mehreren Teilen besteht, die durch Punkte getrennt sind, und dass die Domäne abhängig von ihrem Namen entweder aus den zwei oder drei ganz rechts stehenden Teilen besteht. In dieser Subroutine werden wir dann jeden Host auf seine eigentliche Domäne reduzieren und dann diesen Domänennamen als Index für einen neuen Hash nutzen, wobei wir alle ursprünglichen Treffer für den Hostnamen in dem neuen domänenbasierten Hash speichern.

Die Hauptarbeit dieser Subroutine wird in der `foreach`-Schleife, die in Zeile 4 startet, geleistet. Das Argument, das dieser Subroutine übergeben wird, ist ein Array mit den Hostnamen aus dem `%hosts`-Array. Dabei durchläuft die Schleife alle Hostnamen, um sicherzustellen, dass sie alle berücksichtigt werden.

Der erste Teil der `foreach`-Schleife ist der lange und abschreckende reguläre Ausdruck in Zeile 6. Dieser Ausdruck greift sich die letzten zwei Teile des Hostnamens und, wenn es kann, auch die letzten drei (einige Hostnamen bestehen nur aus zwei Teilen, die jedoch auch vom regulären Ausdruck erfaßt werden). Von Zeile 7 bis 11 wird geprüft, wie viele Teile wir haben (2 oder 3). Diese Teile werden den Variablen `$sd`, `$d` und `$tld` zugewiesen (`$sd` steht für sekundäre Domäne, `$d` für Domäne und `$tld` für Top-Level-Domäne, falls Sie sie auseinanderhalten wollen).

Der zweite Teil der Schleife legt fest, ob wir zwei oder drei Teile des Hostnamens als eigentliche Domäne verwenden wollen, und ignoriert in Zeile 12 alle Hosts, die aus IP-Nummern anstelle von eigentlichen Domänennamen bestehen. Die rein willkürliche Regel, nach der ich entschieden habe, ob eine Domäne aus zwei oder drei Teilen besteht, lautet: Handelt es sich bei der Top-Level-Domäne (den am weitesten rechts gelegenen Teil des Hostnamens) um eine US-Domäne wie `.com`, `.edu` etc. (die vollständige Liste sehen Sie in Zeile 13), dann hat die Domäne nur zwei Teile. Dazu gehören `aol.com`, `mit.edu`, `whitehouse.gov` etc. Lautet die Top-Level-Domäne anders, ist es mit großer Wahrscheinlichkeit eine landesspezifische Domäne wie `.us`, `.au`, `.mx` etc. Diese Domänen verwenden in der Regel drei Teile, um auf eine Site Bezug zu nehmen (zum Beispiel `citygate.co.uk` oder `monash.edu.au`). Zwei Teile wären in diesem Falle nicht genau genug (`edu.au` bezieht sich auf alle Universitäten in Australien und nicht auf eine spezielle namens `edu`).

Das ist also die Aufgabe der Zeilen 13 bis 15: einen Domänennamen aus zwei oder drei Teilen zusammenzusetzen und in dem String `$dom` zu speichern. Wenn wir den Domänennamen zusammengesetzt haben, können wir ihn als Schlüssel für den neuen Hash verwenden und die Treffer, die wir für den ursprünglichen Host ermittelt haben, übertragen (Zeile 16). Nachdem der Domänen-Hash eingerichtet ist, sollten alle Treffer in dem Host-Hash auch in dem Domänen-Hash berücksichtigt sein, so dass wir diesen Hash an die Subroutine `&print_results` zurückgeben können.

Noch eine Sache: In Zeile 18 prüft die Subroutine auf Fehler im Hostnamen. Wenn der Ausdruck des Mustervergleichs in Zeile 6 zu keiner Übereinstimmung führt, muss in der Tat ein sehr seltsamer Hostname vorliegen, und wir geben eine entsprechende Nachricht aus. Im allgemeinen sollte eine solche Nachricht allerdings nicht erscheinen, da ein abartiger Hostname in der Protokolldatei normalerweise bedeutet, dass ein abartiger Hostname auf dem Host selbst vorliegt, was eigentlich über das Internet nur schwer möglich sein dürfte.

Die Subroutine &gettop()

Noch eine Subroutine, und dann können wir den Lehrstoff dieser Woche zur Seite legen, ein Bierchen trinken und feiern, dass wir zwei Drittel dieses Buches bereits bewältigt haben. Die letzte Subroutine `&gettop()` übernimmt einen Hash, sortiert ihn nach Werten und schneidet dann die obersten X Elemente ab, wobei X durch die Variable `&topthings` gegeben ist. Die Subroutine liefert ein Array von Strings zurück, wobei jeder String den Schlüssel und den Wert für die obersten X Elemente in einer Form enthält, die leicht durch die Subroutine `&print_results()`, von der aus diese Subroutine aufgerufen wurde, ausgegeben werden kann. Sehen Sie dazu das Listing 14.6.

Listing 14.6: Die Subroutine &gettop()

```
1:  sub gettop {
2:      my %hash = @_;
```

```
3:      my $i = 1;
4:      my @topkeys = ();
5:      foreach my $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
6:          if ($i <= $topthings) {
7:              push @topkeys, "$key ($hash{$key} hits)";
8:              $i++;
9:          }
10:     }
11:     return @topkeys;
12: }
```

Der Code

Listing 14.7 enthält den vollständigen Code für das Skript ***weblog.pl***.

Je nach Perl-Version sollten Sie auch hier an die `my`-Variablen innerhalb der `foreach`-Schleifen denken. Details finden Sie in dem Hinweis direkt vor dem Listing 14.3.

Listing 14.7: Der Code für weblog.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my $default = 'index.html';     # Angabe Ihrer Standard-HTML-Datei
5:  my $topthings = 30;             # Anzahl der zu protokollierenden
                                    # Dateien, Sites etc.
6:  my $totalhits = 0;
7:  my $failhits = 0;
8:  my $htmlhits = 0;
9:  my %hosts= ();
10: my %files = ();
11:
12: &process_log();
13: &print_results();
14:
15: sub process_log {
16:     my %hit = ();
17:     my $currdate = '';
18:     my $dateshort = '';
19:     print "Log-Dateien verarbeiten....\n";
20:     while (<>) {
21:         chomp;
22:         %hit = splitline($_);
23:         $totalhits++;
24:
25:         # Prüfen auf fehlerhafte Zeilen
26:         if (!%hit) {  # fehlerhafte Zeilen im Webprotokoll
27:             $failhits++;
28:             next;
29:         }
30:
31:         $dateshort = &getday($hit{'date'});
32:         if ($currdate ne $dateshort) {
33:             print "Verarbeite $dateshort\n";
34:             $currdate = $dateshort;
35:         }
36:
37:         # nach 404ern suchen
38:         if ($hit{'code'} =~ /^4/) { # 404, 403, etc. (Fehler)
39:             $failhits++;
40:         # andere Dateien
41:         } elsif ($hit{'code'} =~ /200|304/) {
            # nur im Erfolgsfall bearbeiten
42:             if ($hit{'file'} =~ /\/$/) { # slashes werden zu $default
43:                 $hit{'file'} .= $default;
44:             }
```

```
45:
46:             if ($hit{'file'} =~ /\.html?$/) { # .htm oder .html
47:                 $htmlhits++;
48:             }
49:
50:             $hosts{ $hit{'site'} }++;
51:             $files{ $hit{'file'} }++;
52:         }
53:     }
54: }
55:
56: sub splitline {
57:     my $in = $_[0];
58:     my %line = ();
59:     if ($in =~ /^([^\s]+)\s           # Site
60:                ([\w-]+\s[\w-]+)\s     # Benutzer
61:                \[([^\]]+)\]\s         # Datum
62:                 \"(\w+)\s             # Protokoll
63:                (\/[^\s]*)\s           # Datei
64:                ([^"]+)\"\s            # HTTP-Version
65:                (\d{3})\s              # Rückgabe-Code
66:                ([\d-]+)               # übertragene Bytes
67:     /x) {
68:        # wir sind nur an bestimmten Daten interessiert
69:        # (zufällig jede 2. Information)
70:        $line{'site'} = $1;
71:        $line{'date'} = $3;
72:        $line{'file'} = $5;
73:        $line{'code'} = $7;
74:        return %line;
75:     } else { return (); }
76: }
77:
78: sub getday {
79:     my $date;
80:     if ($_[0] =~ /([^:]+):/) {
81:         $date = $1;
82:         return $date;
83:     } else {
84:         return $_[0];
85:     }
86: }
87:
88: sub print_results {
89:     print "Auswertung der Log-Datei:\n";
90:     print "Gesamtzahl der Treffer: $totalhits\n";
91:     print "Gesamtzahl der fehlgeschlagenen Treffer: $failhits (";
92:     printf('%.2f', $failhits / $totalhits * 100);
93:     print "%)\n";
94:
95:     print "(erfolgreiche) HTML-Dateien: $htmlhits (";
96:     printf('%.2f', $htmlhits / ($totalhits - $failhits) * 100);
97:     print "%)\n";
98:
99:     print 'Anzahl der Hosts: ';
101:    print scalar(keys %hosts);
102:    print "\n";
103:
104:    my %domains = &getdomains(keys %hosts);
105:    print 'Anzahl der Domänen: ';
106:    print scalar(keys %domains);
107:    print "\n";
108:
109:    print "Die beliebtesten Dateien: \n";
110:    foreach my $file (&gettop(%files)) {
111:       print "  $file\n";
112:    }
113:    print "Die beliebtesten Hosts: \n";
114:    foreach my $host (&gettop(%hosts)) {
115:       print "  $host\n";
116:    }
117:
118:    print "Die beliebtesten Domänen: \n";
119:    foreach my $dom (&gettop(%domains)) {
```

```
120:       print "  $dom\n";
121:    }
122: }
123:
124: sub getdomains {
125:     my %domains = ();
126:     my ($sd,$d,$tld);       # sekundäre Domäne, Domäne, oberste Domäne
127:     foreach my $host (@_) {
128:         my $dom = '';
129:         if($host =~ /(([^.]+)\.)?([^.]+)\.([^.]+)$/ ) {
130:             if (!defined($1)) { # nur zwei Domänen (i.e. aol.com)
131:                 ($d,$tld) = ($3, $4);
132:             } else {            # eine normale Domäne x.y.com etc
133:                 ($sd, $d, $tld) = ($2, $3, $4);
134:             }
135:             if ($tld =~ /\D+/) { # ignoriert reine IP-Zahlen
136:                 if ($tld =~ /com|edu|net|gov|mil|org$/i) { # US TLDs
137:                     $dom = "$d.$tld";
138:                 } else { $dom = "$sd.$d.$tld"; }
139:                 $domains{$dom} += $hosts{$host};
140:             }
141:         } else { print "Fehlerhaft: $host\n"; }
142:     }
143:     return %domains;
144: }
145:
146: sub gettop {
147:     my %hash = @_;
148:     my $i = 1;
149:     my @topkeys = ();
150:     foreach my $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {
151:         if ($i <= $topthings) {
152:             push @topkeys, "$key ($hash{$key} hits)";
153:             $i++;
154:         }
155:     }
156:     return @topkeys;
157: }
```

Zusammenfassung

Meistens wird Ihnen das Wissen in Programmierbüchern mit vielen Worten, aber zu wenigen praktischen Codebeispielen vermittelt, so dass Sie oft Schwierigkeiten haben, das Erlernte umzusetzen. Ich möchte zwar nicht behaupten, dass dieses Buch zu den wortkargen gehört, mit diesen Beispielkapiteln möchte ich Ihnen jedoch etwas längere Programme präsentieren, die richtige Probleme lösen und demonstrieren, wie ein reales Skript zusammengesetzt wird.

In der heutigen Lektion haben wir zwei längere Skripts unter die Lupe genommen: eine einfache Adreßdatei mit Suchfunktionen, die eine textbasierte Datenbank mit Namen und Adressen verwendet. Das Skript zur Verarbeitung dieser Datei ermöglicht es Ihnen, ein relativ komplexes Suchmuster zu verarbeiten, einschließlich dem Verschachteln logischer Ausdrücke, und das Zusammenstellen von Wörtern und Phrasen mit Hilfe von Anführungszeichen. Sie könnten dieses Beispiel so erweitern, dass Sie damit so ziemlich jede Situation meistern, in der eine komplexe Suche über Teile einer Datendatei ausgeführt werden soll: zum Beispiel um Mail-Nachrichten anhand von bestimmten Kriterien aus einem Mail-Ordner zu filtern oder nach besonderen Comic-Büchern aus einer Sammlung von Comics zu suchen. Jede Textdatei kann als einfache Datenbank dienen, und dieses Skript kann sie durchsuchen, solange es dahingehend modifiziert wurde, die Daten dieser Datenbank zu verarbeiten.

Das zweite Beispiel war ein Skript zur Auswertung von Log-Dateien, das Protokolle von Web-Servern verarbeitet und Statistiken ausgibt. Reine Protokolle sind häufig vom Äußeren ziemlich abschreckend. Dieses Skript liefert Ihnen einige grundlegende Informationen über das, was auf einer Website so alles abläuft. Dabei bediente es sich einiger komplexer regulärer Ausdrücke und einer Menge von Hashes, um die Rohdaten zu speichern. Sie könnten dieses Beispiel dahingehend erweitern, dass es auch andere Statistiken erstellt (zum Beispiel um Histogramme über die Anzahl der Treffer pro Tag oder pro Stunde anzulegen oder außer HTML-Dateien auch Bild- oder andere Dateien zu verfolgen). Oder Sie könnten Änderungen vornehmen, so dass andere Protokollarten (Mail-Protokolle, FTP-Protokolle oder was gerade anfällt) statistisch erschlossen werden.

Meinen Glückwunsch zum erfolgreichen Abschluß der zweiten Woche dieses dreiwöchigen Exkurses. Nach dieser

Woche haben Sie bereits ein Großteil der Skriptsprache aufgenommen, so dass Sie jetzt in der Lage sein sollten, bereits einige Aufgaben in Perl zu lösen. Ab jetzt werden wir auf das Erlernte aufbauen. Also auf zu Woche 3!

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl für Fortgeschrittene

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Dateien und E/ A

Die ersten zwei Wochen in diesem Buch haben das Thema Eingabe und Ausgabe (auch als E/A oder I/O für input/output bezeichnet) wiederholt am Rande gestreift. Sie kennen `<STDIN>` und `print` für die Standardein- und -ausgabe und wissen, dass man die Dateieingabe über die Befehlszeile, den <>-Operator und die `while`-Schleife vornimmt, damit jede Zeile automatisch der Variablen `$_` zugewiesen wird.

In dem heutigen Kapitel möchte ich Ihr Wissen über die Ein- und Ausgabe vertiefen, Ihnen noch ein bißchen mehr über die Argumentlisten für Skripts erzählen und Ihnen zeigen, wie Sie Daten und Optionen in Ihre Skripts einlesen. Heute erfahren Sie:

- alles über Datei-Handles: wie man sie erstellt, Eingaben ausliest und Ausgaben einliest
- wie man einfache Dateitests durchführt, um Informationen zu einer bestimmten Datei zu erhalten
- wie man mit Skriptargumenten und `@ARGV` arbeitet
- wie man das Modul `Getopt` verwendet, um Schalter zu verwalten

Ein- und Ausgabe mit Datei-Handles

Schon zu Beginn dieses Buches, in Kapitel 3, »Weitere Skalare und Operatoren«, habe ich Ihnen im Zusammenhang mit der Standardeingabe und -ausgabe ein wenig über Datei-Handles erzählt. Damals habe ich Ihnen erklärt, dass `STDIN` und `STDOUT` eine besondere Art von Datei-Handle darstellen, die sich auf Ein- und Ausgabestreams beziehen, die - statt mit einer Datei - mit der Tastatur beziehungsweise dem Bildschirm verbunden sind. Aber zum Glück läßt sich vieles von dem, was Sie bisher gelernt haben, auch auf Datei-Handles übertragen, die sich auf das Dateisystem beziehen.

In diesem Abschnitt lernen Sie, wie Sie die tückischen Datei-Handles in den Griff bekommen: wie man sie mit der `open`-Funktion erzeugt, Daten aus ihnen ausliest oder einliest beziehungsweise anhängt und wie man die Dateien wieder schließt, nachdem sie ihren Dienst getan haben. Außerdem werden wir nebenbei wiederholen, was wir bisher über Eingabe und Ausgabe gelernt haben.

Datei-Handles mit open erzeugen

Um Eingaben aus einer Quelle einzulesen oder Ausgaben an ein Ziel zu schreiben, benötigt man einen Datei-Handle. Ein Datei-Handle ist in der Regel mit einer bestimmten Datei auf der Festplatte verbunden, die zum Lesen oder Schreiben von Daten dient. Er kann sich aber auch auf eine Netzwerkverbindung (Socket), eine Pipe (eine Art Verbindung zwischen Standardeingabe und Standardausgabe, auf die wir noch am Tag 18, »Perl und das Betriebssystem«, eingehen) oder sogar bestimmte Hardware-Geräte beziehen. Das Konzept der Datei-Handles sorgt dafür, dass alle diese Operationen in gleicher Weise ausgeführt werden und Sie in gleicher Weise vorgehen können - egal woher die Daten kommen oder wohin sie gehen.

Perl stellt Ihnen drei Standard-Datei-Handles zur Verfügung, von denen Sie bereits zwei kennen: `STDIN`, `STDOUT` und `STDERR`. Die ersten zwei stehen für die Standardeingabe und Standardausgabe (in der Regel Tastatur und Bildschirm). `STDERR` ist die Standardfehlerausgabe und wird für Fehlermeldungen und andere Nachrichten genutzt, die nicht Teil der eigentlichen Skriptausgabe sind. Normalerweise werden `STDERR`-Nachrichten wie bei `STDOUT` auf dem Bildschirm ausgegeben. Nur Programmen, die speziell Gebrauch von der Standardausgabe machen (zum Beispiel Programme auf der anderen Seite von Unix-Pipes), wird hier ein Unterschied auffallen.

Sie brauchen diese Datei-Handles weder zu öffnen noch zu initialisieren. Sie können sie so, wie sie sind, verwenden (wie bereits in den Lektionen der letzten Woche geschehen).

Um aus einer Datei aus- oder einzulesen, müssen Sie zuerst mit Hilfe der Funktion `open` für die betreffende Operation einen Datei-Handle erzeugen. Die `open`-Funktion öffnet eine Datei zum Einlesen (Eingabe) oder zum Schreiben (Ausgabe) von Daten und verbindet diese Datei mit einem Datei-Handle (dessen Namen Sie frei vergeben können). Beachten Sie, dass es sich beim Einlesen aus einer Datei und beim Schreiben in eine Datei um getrennte Operationen handelt, die beide einen eigenen Datei-Handle benötigen.

Die `open`-Funktion übernimmt zwei Argumente: den Namen eines Datei-Handles und die zu öffnende Datei (dazu gehört auch ein besonderer Code, der anzeigt, ob die Datei zum Lesen oder Schreiben geöffnet wird). Hier einige Beispiele:

```
open(FILE, 'meinedatei');
open(CONFIG, '.scriptconfig');
open(LOG, '>/home/www/logfile');
```

Den Namen des Datei-Handles können Sie beliebig wählen. Die Konvention sieht jedoch vor, dass er aus Großbuchstaben besteht und Buchstaben, Zahlen oder Unterstriche enthält. Im Gegensatz zu den Variablen muss ein Datei-Handle mit einem Buchstaben beginnen.

Das zweite Argument ist der Name der Datei auf der Festplatte, die mit Ihrem Datei- Handle verbunden werden soll. Ein einfacher Dateiname ohne weitere Pfadangaben wird im aktuellen Verzeichnis gesucht (entweder das, in dem Ihr Skript ausgeführt wird, oder ein anderes Verzeichnis, sollten Sie es gewechselt haben). Alles weitere zum Navigieren von Verzeichnissen erfahren Sie in Kapitel 17, »Umgang mit Dateien und Verzeichnissen«.

Wollen Sie statt einfacher Dateiangaben Pfadnamen verwenden, sollten Sie Vorsicht walten lassen - die Pfadnotation ist von Plattform zu Plattform unterschiedlich. Unter Unix werden die einzelnen Pfade durch Schrägstriche (`/`) getrennt, wie das letzte unserer obigen Beispiele zeigt.

Auf Windows-Systemen läßt sich problemlos die Standard-DOS-Notation mit Backslash (`\`) zwischen den Verzeichnissen verwenden, solange Sie daran denken, den ganzen Pfadnamen in einfache Anführungszeichen zu setzen. Zur Erinnerung: Ein Backslash gehört zu den Sonderzeichen in Perl. Wenn Sie also die Anführungszeichen fortlassen, erhalten Sie unter Umständen einen seltsamen Pfad, der keinen Bezug zur Realität hat. Und wenn Sie den String in doppelte Anführungszeichen setzen, müssen Sie vor den Backslash ein Escape-Zeichen (ebenfalls ein Backslash) setzen, um ein korrektes Ergebnis zu erzielen:

```
open(FILE, 'c:\temp\nummern');  # korrekt
open(FILE, "c:\temp\nummern");
# ooh! enthält einen Tabulator und eine Neue Zeile (\t, \n)
open(FILE "c:\\temp\\nummern"); # korrekt
```

Da die meisten modernen Windows-Systeme auch Pfadnamen mit Schrägstrichen verstehen, sollten Sie vielleicht besser diese verwenden. Damit erhöhen Sie außerdem gleichzeitig die Portierbarkeit auf Unix-Systeme (falls Ihnen daran gelegen ist).

Beim Macintosh ist das Trennzeichen zwischen den Verzeichnissen ein Doppelpunkt, und der absolute Pfadname beginnt mit der Festplatte oder dem Laufwerksbezeichner (Festplatte, CD-ROM, Diskette und so weiter). Streben Sie die Portierung auf andere Systeme an, sollten Sie sich eine Notiz machen und Ihre Pfadnamen später konvertieren. Hier ein paar Beispiele für die Mac-Syntax:

```
open(FILE, "Meine Festplatte:Perl:config");
open(BOOKMARKS, "HD:System Folder:Preferences:Netscape:Bookmarks.html");
```

In jedem dieser Beispiele haben wir ein Datei-Handle geöffnet, um die Eingaben in das Skript einzulesen. Dies entspricht dem Standard. Wollen Sie die Ausgabe in eine Datei zurückschreiben, benötigen Sie dafür ebenfalls einen Datei-Handle, den Sie mit `open` öffnen - allerdings unter Verwendung eines besonderen Zeichens vor dem Dateinamen:

```
open(OUT, ">output");
```

Das Zeichen > zeigt an, dass dieser Datei-Handle zum Schreiben von Daten geöffnet wird. Die gegebene Datei wird geöffnet und der aktuelle Inhalt, sofern vorhanden, gelöscht. (Um das Überschreiben bestehender Dateien zu vermeiden, können Sie in einem Test abfragen, ob eine Datei existiert, bevor Sie sie zum Schreiben von Daten

öffnen - im Abschnitt »Datei-Tests« erfahren Sie mehr darüber.)

Wie verfahren Sie, wenn Sie Eingaben von einer Datei auslesen wollen, diese Daten dann bearbeiten und anschließend in dieselbe Datei zurückschreiben wollen? Dann müssen Sie zwei Datei-Handles öffnen: einen Handle zum Einlesen der Eingabe und später den zweiten Handle, um die Datei erneut zu öffnen und die Daten zurückzuschreiben. Lesen und Schreiben sind unterschiedliche Prozesse und bedürfen unterschiedlicher Datei-Handles.

Es sei angemerkt, dass es auch eine Syntax gibt, mit der man ein und dieselbe Datei für das Lesen und Schreiben öffnen kann: "+>filename". Diesen Code können Sie verwenden, wenn Sie die Datei als eine Datenbank verstehen, die Sie nicht als Ganzes einlesen, sondern auf der Festplatte speichern und dann auf diese Datei zum Lesen und Schreiben zugreifen, wenn Sie Daten lesen oder ändern. In diesem Buch beschränke ich mich darauf, einfache Textdateien zu lesen und zu schreiben, so dass die Verwaltung der Daten mit Hilfe von zwei getrennten Datei-Handles weniger verwirrend und leichter ist: ein Handle, um die Daten in den Speicher einzulesen, und ein zweiter Handle, um die Daten wieder in die Datei zu schreiben.

Sie können eine Datei auch zum Anhängen öffnen - dabei bleibt der aktuelle Inhalt der Datei erhalten, und alle Ausgaben über das Ausgabe-Datei-Handle werden an das Ende der Datei angehängt. Um Daten anzuhängen, müssen Sie die Sonderzeichen >> in Ihrem `open`-Aufruf verwenden:

```
open(FILE, ">>logdatei");
```

Die Funktion die

Die Funktion `open` wird fast immer zusammen mit einem logischen `or` und einem Aufruf von `die` auf der anderen Seite aufgerufen:

```
open(FILE, "dieDatei") or die "dieDatei wurde nicht gefunden\n";
```

Der Aufruf von `die` (»sterben«) ist nicht erforderlich, erfolgt aber so häufig in Perl, dass die Kombination fast unzertrennlich wirkt. Wenn Sie die Kombination nicht verwenden, ist es sehr wahrscheinlich, dass Sie irgend jemand, der Ihren Code sieht, eines Tages darauf anspricht und fragt, warum Sie den Befehl fortgelassen haben.

»Öffne diese Datei oder stirb« ist die implizierte Drohung dieser Anweisung, und das ist normalerweise genau das, was Sie auch wollen. Der `open`-Befehl könnte fehlschlagen - sei es, dass Sie eine Datei zum Lesen öffnen, die nicht existiert, oder dass Ihre Festplatte sich seltsam verhält und die Datei nicht öffnen kann oder dass sonstige unvorhersehbaren Ereignisse eintreten. Normalerweise wollen Sie nicht, dass Ihr Skript einfach weiter ausgeführt wird, wenn etwas so absolut falsch läuft und das Skript nichts zum Einlesen finden kann. Zum Glück liefert die `open`-Funktion `undef` zurück, wenn sie die Datei nicht öffnen kann (und 1 wenn es ihr gelungen ist), so dass Sie dieses Ergebnis abfragen können und damit eine Entscheidungshilfe haben, was zu tun ist.

Manchmal kann die Art, in der Sie auf den Fehler reagieren wollen, vom Skript abhängen. In der Regel jedoch wollen Sie einfach nur das Skript mit einer Fehlermeldung verlassen. Und genau dafür sorgt die `die`-Funktion: Sie beendet automatisch das gesamte Perl-Skript und gibt ihr Argument (eine String-Nachricht) über den Datei-Handle `STDERR` (normalerweise der Bildschirm) aus.

Wenn Sie ein Neue-Zeile-Zeichen an das Ende der Nachricht setzen, wird Perl diese Nachricht beim Beenden des Skripts ausgeben. Wenn Sie das Neue-Zeile-Zeichen weglassen, gibt Perl als zusätzliche Information die aktuelle Zeilennummer aus: `at script.pl line nn`. Dabei ist `script.pl` der Name Ihres Skripts und `line nn` die Zeile, in der `die` ausgelöst wurde. Diese Information kann beim späteren Debuggen Ihres Skripts sehr nützlich sein.

Der Befehl `die` kann aber noch mehr: Die spezielle Perl-Variable `$!` enthält den letzten Fehler des Betriebssystems (falls Ihr Betriebssystem einen solchen erzeugt hat). Indem Sie die Variable `$!` in den String für `die` einbinden, kann Ihre Fehlermeldung unter Umständen noch mehr Informationen liefern, als nur mitzuteilen, dass die Datei nicht geöffnet werden konnte. So kann zum Beispiel die folgende Version von `die`:

```
die "Datei konnte nicht geoeffnet werden: $!\n";
```

in Kombination mit der Variablen folgende Meldung ausgeben: »`Datei konnte nicht geoeffnet werden: Zugriff verweigert.`« Hier erfährt der Benutzer gleich, dass die Datei nicht geöffnet wurde, weil die Zugriffsberechtigung für die Datei fehlte. Die Verwendung von `$!` ist immer dann zu empfehlen, wenn Sie `die` als Antwort auf ein Art von Systemfehler aufrufen.

Obwohl `die` meist zusammen mit `open`-Aufrufen verwendet wird, sollte man daraus nicht schließen, dass `die` nur in diesem Kontext sinnvoll einzusetzen ist. Sie können `die` (und seine weniger radikale Entsprechung `warn`) überall dort in Ihrem Skript einsetzen, wo Sie die Ausführung des Skripts abbrechen wollen (oder eine Warnung ausgeben wollen). Weitere Informationen zu `die` und `warn` finden Sie in der ***perlfunc***- Manpage.

Eingaben über einen Datei-Handle einlesen

Angenommen Sie haben einen Datei-Handle, und er ist mit einer Datei verbunden, die Sie zum Lesen geöffnet haben. Um Daten über den Datei-Handle einzulesen, verwenden Sie den (Eingabe-)Operator <> zusammen mit dem Namen des Datei- Handles:

```
$line = <FILE>;
```

Kommt Ihnen sicherlich bekannt vor, oder? Bei `STDIN` sind Sie genauso vorgegangen, um eine über die Tastatur eingegebene Zeile einzulesen. Das ist das Tolle an den Datei-Handles - Sie können genau die gleichen Prozeduren für das Lesen aus einer Datei wie für das Lesen von der Tastatur oder von einer Netzwerkverbindung zu einem Server verwenden. Perl macht da keinen Unterschied. Das Verfahren ist immer dasselbe, und alles was Sie bisher über E/A gelernt haben, können Sie auch für die Arbeit mit Dateien verwenden.

In einem skalaren Kontext liest der Eingabeoperator eine einzige Zeile bis zum Neue- Zeile-Zeichen ein:

```
$line = <STDIN>;
if (<FILE>) { print "weitere Eingaben..." };
```

Eine besondere Möglichkeit für die Verwendung des Eingabeoperators in einem skalaren Kontext besteht darin, den Operator innerhalb des Tests einer `while`-Schleife zu verwenden. Damit erreichen Sie, dass bei jedem Schleifendurchlauf jeweils eine Zeile eingelesen und diese Zeile dann der Variablen `$_` zugewiesen wird. Die Schleife hört erst auf, wenn das Ende der Eingabe erreicht ist:

```
while (<FILE>) {
   # ... die Zeilen der Datei (in $_) verarbeiten
}
```

Die gleiche Notation ist Ihnen bereits häufiger begegnet, allerdings mit leeren Eingabeoperatoren. Die leeren Eingabeoperatoren <> stellen in Perl einen Sonderfall dar. Wie Sie gelernt haben, verwendet man die leeren Eingabeoperatoren, um den Inhalt von Dateien einzulesen, die in der Befehlszeile des Skripts angegeben werden. In einem solchen Fall öffnet Perl die Dateien für Sie und sendet Ihnen deren Inhalt Datei für Datei über den Datei-Handle `STDIN`. Sie selbst brauchen nichts weiter zu tun. Natürlich könnten Sie auch jede Datei selbst öffnen und einlesen, aber die <>- Operatoren in der `while`-Schleife stellen eine wirklich praktische Hilfe dar, die den Prozeß erheblich verkürzen.

In einem Listenkontext erreicht man mit den Eingabeoperatoren, dass die gesamte Eingabe auf einmal eingelesen wird, wobei jede Zeile der Eingabe einem Element der Liste zugeordnet wird. Seien Sie vorsichtig, wenn Sie den Eingabeoperator in einem Listenkontext verwenden, da er sich nicht immer so verhält, wie Sie es erwarten. Hierzu einige Beispiele:

```
@input = <FILE>;   # liest die gesamte Datei nach @input ein;
$input = <FILE>;   # liest die erste Zeile der Datei nach $input ein
($input) = <FILE>; # liest die erste Zeile der Datei nach $input ein und
                   # verwirft den Rest der Datei!
print <FILE>;      # gibt den gesamten Inhalt von <FILE> auf dem Bildschirm
                   # aus
```

Ausgaben in einen Datei-Handle schreiben

Um Ausgaben in einen Datei-Handle zu schreiben, verwendet man meist die Funktionen `print` oder `printf`. (Es gibt zwar noch die `write`-Funktion, doch wird diese meist in Kombination mit Ausgabeformaten verwendet - ein Thema, das wir erst in Kapitel 20, »Was noch bleibt«, ansprechen werden).

Die Funktion `print` (wie auch `printf`) geben ihre Daten standardmäßig an den Datei- Handle `STDOUT` aus. Um einen anderen Datei-Handle anzusprechen, zum Beispiel um eine Zeile in eine Datei zu schreiben, müssen Sie erst den Datei-Handle zum Schreiben öffnen:

```
open(FILE, ">$meinedatei") or
                          die "Kann die Datei $meinedatei nicht finden\n";
```

Und dann verwenden Sie `print` mit dem Datei-Handles als Argument, um Daten in dieser Datei abzulegen:

```
print FILE "$zeile\n";
```

Die Funktionen `printf` und `sprintf` sind in ihrer Funktionsweise ähnlich; denken Sie nur daran, vor dem Formatierstring und den auszugebenden Werten den Datei-Handle anzugeben, in den die Ausgabe erfolgen soll:

```
printf(FILE "%d Antworten wurden erfaßt\n", $total / $count);
```

Einen Punkt dürfen Sie bei der Verwendung von `print` und `printf` nicht vergessen: Es steht kein Komma zwischen dem Datei-Handle und der Liste der Dinge, die ausgegeben werden sollen. Dies ist einer der häufigsten Perl-Fehler (der allerdings aufgefangen wird, wenn Sie die Perl-Warnungen eingeschaltet haben). Das Argument des Datei-Handles ist vollkommen getrennt zu sehen von dem zweiten Argument, bei dem es sich um eine Liste von Elementen handelt, die durch Kommata getrennt sind.

Binäre Dateien lesen und schreiben

Bisher haben wir in diesem Buch nur Textdaten gelesen und geschrieben. Aber nicht alle Dateien, mit denen Sie in Perl konfrontiert werden, liegen im Textformat vor. Häufig hat man es mit binären Dateien zu tun. Wenn Sie Perl unter Unix oder auf einem Mac verwenden, macht das keinen großen Unterschied, denn Unix und MacPerl verarbeiten Text- und Binärdateien ohne Probleme. Arbeiten Sie hingegen unter Windows, erhalten Sie unleserliche Ergebnisse, wenn Sie versuchen, eine binäre Datei in einem normalen Perl-Skript zu verarbeiten.

Glücklicherweise kann man dem leicht abhelfen: Die Funktion `binmode` übernimmt als Argument einen einzelnen Datei-Handle und verarbeitet ihn (aus ihm lesen und in ihn schreiben) in binärer Form:

```
open(FILE, "meineDatei.exe") or
                 die " Datei meineDatei konnte nicht geoeffnet werden: $!\n";
binmode FILE;
while (<FILE>) { # im binären Modus lesen...
```

Einen Datei-Handle schließen

Wenn Sie mit dem Lesen oder Schreiben der Daten über den Datei-Handle fertig sind, sollte er geschlossen werden. Meist wird Ihnen diese Aufgabe abgenommen, denn wenn die Ausführung Ihres Skripts beendet ist, schließt Perl alle Ihre Datei-Handles für Sie. Und wenn Sie mit `open` den gleichen Datei-Handle mehrmals hintereinander öffnen (zum Beispiel um einen Datei-Handle, aus dem zuvor gelesen wurde, zum Schreiben zu öffnen), trägt Perl dafür Sorge, dass der Datei-Handle automatisch geschlossen wird, bevor Sie ihn erneut öffnen. Es gehört jedoch zum guten Programmierstil, seine Datei-Handles zu schließen, nachdem sie ihren Dienst erfüllt haben. Dann belegen Sie in Ihrem Skript auch keinen unnötigen Speicher mehr.

Einen Datei-Handle schließt man mit `close`:

```
close FILE;
```

Ein Beispiel: Betreffzeilen extrahieren und sichern

Mailboxen für E-Mails gehören zu den Formaten, die Perl wirklich gut beherrscht. Jede Nachricht folgt einem

speziellen Format (auf der Basis des Protokolls RFC822), und alle Nachrichten zusammen werden in einer Mailbox mit ebenfalls speziellem Format gesammelt. Wenn Sie also eine Mailbox sichten und Nachrichten, die bestimmte Kriterien erfüllen, in irgendeiner Weise verarbeiten wollen, dann ist Perl Ihre Sprache. Wie man E-Mails filtert, werden wir uns in Kapitel 21, »Ein paar längere Beispiele«, noch genauer anschauen.

Das Beispiel in diesem Kapitel ist noch sehr einfach: Das Skript übernimmt eine Mailbox als Argument aus der Befehlszeile, liest die Nachrichten einzeln ein, extrahiert alle Zeilen, die mit ***subject*** beginnen (englisches Wort, mit dem die Betreff-Zeile der E- Mail beginnt) und legt in dem gleichen Verzeichnis eine Datei namens `Betreff` an, die eine Liste der Betreffzeilen enthält. Existiert in dem Verzeichnis bereits eine Datei dieses Namens, wird sie überschrieben (im Anschluß an diesen Abschnitt zeige ich Ihnen, wie Sie testen können, ob eine gleichlautende Datei existiert, und dann eine Warnung ausgeben lassen).

Listing 15.1 enthält den (sehr einfachen) Code.

Listing 15.1: Das Skript subject.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  open(OUTFILE, ">Betreff") or
                  die " Datei Betreff konnte nicht geoeffnet werden: $!\n";
5:
6:  while (<>) {
7:      if (/^Subject:/) {
8:          print OUTFILE $_;
9:      }
10: }
11: close OUTFILE;
```

Ein kurzes Skript, werden Sie denken, das kaum als Beispiel taugt. Aber genau das ist der springende Punkt: Zum Auslesen von Daten aus Dateien und zum Schreiben in Dateien verwenden Sie die gleichen Techniken wie für die Standardeingabe und - ausgabe. Auf zwei Dinge möchte ich Ihre Aufmerksamkeit lenken: Erstens, Zeile 4 öffnet die Datei ***Betreff*** zum Schreiben (beachten Sie dabei das Zeichen > am Anfang des Dateinamens) und zweitens, Zeile 8 übergibt unsere Ausgabe dem gleichen Datei- Handle `OUTFILE` und nicht der Standardausgabe.

Dieses Skript erzeugt zwar keine sichtbare Ausgabe, aber wenn Sie es auf einer Datei mit abgespeicherten E-Mails ausführen und dann die Datei `Betreff` betrachten, finden Sie dort Zeilen wie die folgenden (mein Beispiel hier ist das Ergebnis einer Analyse meiner »Business-Mail«):

```
Subject: FREE SOFTWARE TURN$ COMPUTER$ INTO CA$H MACHINE$!!
Subject: IBM 33.6 PCMCIA Modem $89.00
Subject: 48 MILLION Email Leads $195 + BONUSES
Subject: Re: E-ALERT: URGENT BUY RECOMMENDATION
Subject: Make $2,000 - $5,000 per week -NOT MLM
Subject: Email your AD to 57 MILLION People for ONLY $99
Subject: SHY?.....................................
Subject: You Could Earn $100 Every Time the Phone Rings!!
```

Dateitests

Das Öffnen von Dateien zum Einlesen oder Hineinschreiben ist gut und schön, wenn Sie die Dateien, mit denen Sie arbeiten, kennen - zum Beispiel wissen, dass sie alle vorhanden sind oder dass Sie nicht Gefahr laufen, wichtige Daten zu überschreiben. Manchmal jedoch wollen Sie in Perl die Eigenschaften einer Datei prüfen, bevor Sie sie öffnen, oder eine Datei in Abhängigkeit von ihren diversen Eigenschaften in der einen oder anderen Weise verarbeiten.

Perl kennt eine (ziemlich umfangreiche) Reihe von Tests für die verschiedenen Eigenschaften von Dateien. Mit Hilfe dieser Tests läßt sich leicht feststellen, ob eine Datei bereits existiert, ob Sie Daten enthält, zu welcher Art von Datei sie gehört oder wie alt sie ist (»1996 war ein gutes Jahr für binäre Dateien, nicht wahr?«). Diese Tests erinnern alle an Schalter (`-e`, `-R`, `-o` und so weiter), sie sollten aber nicht mit diesen verwechselt werden (die Ähnlichkeit ist eine Folge davon, dass sie alle ihren Ursprung in der Unix-Shell-Skriptsprache haben).

Tabelle 15.1 gibt einen Überblick über einige der nützlicheren Dateitests. In der ***perlfunc***-Manpage sind unter dem Eintrag `-X` alle Tests aufgeführt (allerdings sind nicht alle Optionen für alle Plattformen verfügbar).

Jeder dieser Tests kann als Argument einen Dateinamen oder einen Datei-Handle übernehmen; beides ist möglich (wenn Sie jedoch überprüfen wollen, ob es eine bestimmte Datei gibt oder nicht, werden Sie mit Sicherheit den Dateinamen übergeben, um den Test vor dem Aufruf von `open` durchzuführen).

Test	Was er bewirkt
`-d`	Handelt es sich bei der Datei um ein Verzeichnis?
`-e`	Existiert die Datei?
`-f`	Ist die Datei eine einfache Datei (und kein Verzeichnis, Link oder Netzwerkverbindung)?
`-l`	Ist die Datei ein Link (nur Unix)?
`-r`	Kann die Datei gelesen werden (von Benutzer oder Gruppe unter Unix)?
`-s`	Wieviel Byte umfaßt die Datei?
`-t`	Ist das Datei-Handle offen für STDIN (oder ein anderes tty-Gerät unter Unix)?
`-w`	Kann in die Datei geschrieben werden (von Benutzer oder Gruppe unter Unix)?
`-x`	Handelt es sich bei der Datei um eine ausführbare Datei?
`-z`	Ist die Datei vorhanden, jedoch leer?
`-A`	Wieviel Zeit ist seit dem letzten Zugriff verstrichen (in Sekunden)?
`-B`	Handelt es sich um eine binäre Datei (keine Textdatei)?
`-M`	Wieviel Zeit ist seit der letzten Änderung verstrichen (in Sekunden)?
`-T`	Handelt es sich um eine Textdatei (keine binäre Datei)?

Tabelle 15.1: ***Dateitests***

Fast alle Tests liefern entweder ***wahr*** (`1`) oder ***falsch*** (»«) zurück. Nur `-e`, `-s`, `-M` und `-A` liefern andere Werte. Der Test `-e` liefert `undef` zurück, wenn die Datei nicht existiert, `-s` liefert die Anzahl der Bytes (Zeichen) in der Datei, und die Zeitoperatoren `-M` und `-A` liefern die Anzahl der Sekunden, die seit der letzten Änderung beziehungsweise dem letzten Zugriff verstrichen sind.

Angenommen Sie wollten das Skript ***subject.pl*** dahingehend ändern, dass für den Fall, dass eine Datei `Betreff` bereits existiert, der Benutzer mit einer Eingabeaufforderung selber entscheiden kann, ob die Datei überschrieben werden soll oder nicht. Anstelle des bisherigen einfachen Aufrufs von `open` könnten Sie einen Test durchführen, um sicherzugehen, dass die Datei überhaupt existiert, und wenn ja, den Benutzer entscheiden lassen, ob diese überschrieben werden soll oder nicht. Schauen Sie sich dazu den folgenden Code an:

```
if (-e 'Betreff') {
    print 'Datei existiert bereits.  Überschreiben (J/N)? ';
    chomp ($_ = <STDIN>);
    while (/[^jn]/i) {
        print 'J oder N, bitte: ';
        chomp ($_ = <STDIN>);
    }
    if (/n/i) { die "Die Datei Betreff existiert bereits; Abbrechen.\n"; }
}
```

In diesem Beispiel sehen Sie eine andere Anwendungsmöglichkeit für die Funktion `die` - diesmal ohne die Funktion `open`. Wenn der Benutzer die Frage zum Überschreiben der Datei mit `N` beantwortet, könnten Sie das Skript einfach verlassen. Die `die`-Funktion beendet das Skript ausdrücklich und gibt eine entsprechende Nachricht aus.

Mit @ARGV und Skriptargumenten arbeiten

Einen Aspekt bei der Ausführung von Perl-Skripten, den ich in den letzten Tagen etwas vernachlässigt habe, betrifft den Umgang mit Befehlszeilenargumenten. Sie haben schon ein wenig über Perls eigene Schalter (`-e`, `-w` und so weiter) erfahren, wie aber gehen Sie vor, wenn Sie solche Schalter oder Argumente Ihren eigenen Skripten übergeben wollen - wie kann man diese verarbeiten? Dieser Abschnitt behandelt im besonderen folgende Themen: Skriptargumente im allgemeinen und den Einsatz von Skriptschaltern.

Die Anatomie von @ARGV

Wenn Sie ein Perl-Skript nicht nur mit dem Namen des Skripts, sondern mit weiteren Argumenten aufrufen, werden diese Argumente in einer besonderen globalen Liste, der `@ARGV`-Liste, gespeichert (für Mac-Droplets enthält `@ARGV` die Namen der Dateien, die auf das Droplet gezogen wurden). Sie können dieses Array genauso verarbeiten wie jede andere Liste in Ihrem Perl-Skript. Sehen Sie im folgenden ein Codefragment, das lediglich die Argumente, mit denen das Skript aufgerufen wurde, ausgibt, und zwar ein Argument pro Zeile:

```
foreach my $arg (@ARGV) {
   print "$arg\n";
}
```

Wenn Ihr Skript ein Konstrukt wie `while (<>)` verwendet, zieht Perl den Inhalt der `@ARGV`-Liste als Dateinamen zum Öffnen und Lesen heran (stehen keine Dateien in `@ARGV`, versucht Perl, von der Standardeingabe zu lesen). Mehrere Dateien werden hintereinander geöffnet und gelesen, als ob es eine einzige große Datei wäre.

Wenn Sie mehr Kontrolle über den Inhalt der Dateien, die Sie in Ihr Skript einlesen, haben wollen, können Sie die Namen der zu öffnenden und zu lesenden Dateien aus der `@ARGV`-Liste auslesen. Das Auswerten von `@ARGV` bietet sich auch dann an, wenn Sie nach bestimmten Argumenten suchen - zum Beispiel einer Konfigurationsdatei und einer Datendatei. Wollen Sie hingegen den Inhalt einer beliebigen Anzahl von Dateien bearbeiten, ist es praktischer, die Abkürzung <> zu verwenden. Und wenn Sie einen bestimmten Satz von Argumenten erwarten und kontrollieren wollen, wie diese verarbeitet werden sollen, lesen Sie die Dateien von `@ARGV` aus, und bearbeiten Sie sie einzeln.

Im Gegensatz zu `argv`, wie es C und Unix kennen, enthält die `@ARGV`-Liste von Perl nur die Argumente und nicht den Namen des Skripts selber (`$ARGV[0]` enthält das erste Argument). Um den Namen des Skripts zu ermitteln, können Sie die spezielle Variable `$0` verwenden.

Skriptschalter und Spaß mit Getopt

Ein typischer Anwendungsbereich für die Skript-Befehlszeile ist die Übergabe von Schaltern an ein Skript. Schalter sind Argumente, die mit einem Gedankenstrich beginnen (`-a`, `-b`, `-c`) und in der Regel dazu dienen, das Verhalten des Skripts zu steuern. Manchmal bestehen Sie nur aus einem Buchstaben (`-s`), manchmal sind sie zu Gruppen zusammengefaßt (`-abc`), und manchmal ist ihnen ein Wert oder Argument zugeordnet (`-o ausgabe.txt`).

Sie können ein Skript mit jedem beliebigen Schalter aufrufen. Die Schalter werden, wie alle anderen Argumente auch, als Elemente im `@ARGV`-Array aufgenommen. Wenn Sie das Array `@ARGV` mit <> bearbeiten, müssen Sie die Schalter im Array erst loswerden, bevor Sie irgendwelche Daten auslesen - sonst würde Perl davon ausgehen, dass es sich bei `-s` um einen Dateinamen handelt. Um alle Schalter, die über das ganze `@ARGV`-Array verstreut sind, zu bearbeiten und zu entfernen, könnten Sie mühselig das Array durchgehen und versuchen herauszufinden, welche Elemente davon Optionen und welche Optionen mit dazugehörigen Argumenten sind. Als Endergebnis bliebe dann eine Liste der eigentlichen Dateinamen. Sie könnten sich aber auch des Moduls `Getopt` bedienen und sich damit die Arbeit abnehmen lassen.

Das Modul `Getopt`, das als Teil der Standard-Modul-Bibliothek zusammen mit Perl ausgeliefert wird, dient der Verwaltung der Skriptschalter. Eigentlich handelt es sich um zwei Module: `Getopt::Std` für die Bearbeitung von Schaltern, die aus einem Buchstaben bestehen (`-a`, `-d`, `-odatei` und so weiter), und `Getopt::Long`, das fast alle erdenklichen Optionen akzeptiert, einschließlich Optionen, die aus mehreren Buchstaben bestehen (`-sde`), oder Optionen im GNU-Stil mit doppeltem Bindestrich (`--help`, `--size` und so weiter).

In diesem Abschnitt bespreche ich das Modul `Getopt::Std` für die einfachen Optionen. Wenn Sie für komplexere Optionen das Modul `Getopt::Long` einsetzen möchten, sollten Sie auf die Dokumentation zu diesem Modul zurückgreifen (Details finden Sie in der ***perlmod***-Manpage).

Für eine erfolgreiche Arbeit müssen Sie das Modul `Getopt::Std`, wie jedes andere Modul auch, in Ihr Skript importieren:

```
use Getopt::Std;
```

Durch den Import von `Getopt::Std` erhalten Sie zwei Funktionen: `getopt` und `getopts`. Diese Funktionen werden benötigt, um die Schalter aus Ihrem `@ARGV`-Array herauszuziehen und für jeden dieser Schalter in Ihrem Skript eine Skalarvariable zu setzen.

Das Modul `Getopt` funktioniert in der Anwendungsversion von MacPerl nicht ordnungsgemäß, da es in MacPerl keine einfache Möglichkeit gibt, Befehlszeilenschalter einzulesen. Im Abschnitt »Skriptschalter auf dem Macintosh« finden Sie Hinweise, wie Sie dieses Problem in MacPerl umgehen.

getopts

Beginnen wir mit der Funktion `getopts`, die einbuchstabige Schalter mit oder ohne Werte definiert und bearbeitet. `getopts` übernimmt ein einziges String-Argument, das die Zeichen für die Schalter enthält, die von Ihrem Skript akzeptiert werden sollen. Argumente, die Werte übernehmen, müssen von einem Doppelpunkt (`:`) gefolgt werden. Groß- und Kleinbuchstaben machen einen Unterschied und definieren verschiedene Schalter. Schauen wir uns ein Beispiel an:

```
getopts('abc');
```

Das in diesem Beispiel verwendete Argument `'abc'` bearbeitet die Schalter `-a`, `-b` oder `-c` in einer beliebigen Reihenfolge und ohne zugeordnete Werte. Die Schalter können auch zusammengefaßt werden: `-ab` oder `-abc` lassen sich genauso verwenden wie die einzelnen Schalter. Noch ein Beispiel:

```
getopts('ab:c');
```

Hier kann der Schalter `-b` einen Wert übernehmen, der auf der Perl-Befehlszeile direkt nach dem Schalter folgen muss:

```
% meinskript.pl -b 10
```

Das Leerzeichen hinter dem Schalter ist nicht erforderlich. `-b10` läßt sich genauso schreiben wie `-b 10`. Sie können die Schalter sogar verschachteln, solange der Wert nur nach dem korrekten Schalter erscheint:

```
% meinskript.pl -acb10 # OK
% meinskript.pl -abc10 # falsch, b und 10 gehören zusammen
```

Für jeden in `getopts` definierten Schalter erzeugt `getopts` einen Skalarvariablen- Schalter mit dem Namen `$opt_x`, wobei `x` der Buchstabe des Schalters ist (in unserem Beispiel würde `getopts` drei Variablen `$opt_a`, `$opt_b` und `$opt_c` erzeugen). Der Anfangswert jeder Skalarvariablen ist `0`. Wenn der Schalter in den Argumenten zu dem Skript enthalten ist (in `@ARGV` steht), setzt `getopts` den Wert der zugeordneten Variable auf `1`. Erwartet ein Schalter einen Wert, weist `getopts` den Wert aus `@ARGV` der Skalarvariablen für die Option zu. Danach werden der Schalter und sein dazugehöriger Wert aus dem Array `@ARGV` gelöscht. Nachdem `getopts` seine Bearbeitung abgeschlossen hat, ist Ihr `@ARGV` entweder leer oder enthält die restlichen Argumente in Form von Dateinamen, die Sie dann mit den Datei-Handles oder mit `<>` verarbeiten können.

Nachdem `getopts` seine Aufgabe erledigt hat, steht Ihnen für jeden Schalter eine Variable zur Verfügung, die entweder den Wert `0` hat (für einen unbenutzten Schalter), `1` (für einen benutzten Schalter) oder einen bestimmten Wert (für einen Schalter, der einen Wert erfordert). Sie können diese Werte abfragen und Ihr Skript, je nachdem

mit welchem Schalter es aufgerufen wurde, verschiedene Operationen ausführen lassen:

```
if ($opt_a) {  # -a wurde verwendet
   ...
}
if ($opt_b) {  # -b wurde verwendet
 ...
}
```

Wenn beispielsweise der Aufruf Ihres Skripts wie folgt aussieht:

```
% skript.pl -a
```

wird `getopts('abc')` die Variable `$opt_a` auf `1` setzen. Würde das Skript folgendermaßen aufgerufen:

```
% skript.pl -a -R
```

wird `$opt_a` auf `1` gesetzt und der Schalter `-R` einfach gelöscht, ohne dass eine Variable gesetzt wurde. Bei folgendem Aufruf des Skripts:

```
% skript.pl -ab10
```

und gleichzeitigem Aufruf von `getopts` mit

```
getopts('ab:c');
```

wird `$opt_a` auf `1` gesetzt und `$opt_b` auf `10`.

Denken Sie daran, dass Perl bei Verwendung des Befehls `use strict` die plötzlich auftauchenden Variablen `$opt_` monieren wird. Das läßt sich jedoch vermeiden, indem Sie diese Variablen im voraus mit `use vars` wie folgt deklarieren:

```
use vars qw($opt_a $opt_b $opt_c);
```

Fehlerbehandlung mit getopts

Es gilt zu beachten, dass `getopts` das Array `@ARGV` der Reihe nach ausliest und die Bearbeitung unterbricht, wenn es auf ein Element stößt, das nicht mit einem Gedankenstrich (`-`) beginnt oder das keinen Wert für eine vorausgehende Option darstellt. Das bedeutet, dass Sie beim Aufruf eines Perl-Skripts zuerst die Optionen und dann die anderen Argumente aufführen sollten. Andernfalls droht Ihnen, dass Optionen unbearbeitet bleiben und Sie Fehler erhalten, weil versucht wird, Dateien zu lesen, die gar keine Dateien sind. Sie können Ihr Skript natürlich auch so schreiben, dass sichergestellt ist, dass `@ARGV` leer ist, nachdem `getopts` abgearbeitet ist, oder dass die übriggebliebenen Argumente nicht mit einem Gedankenstrich beginnen.

Grundsätzlich sind die von `getopts` definierten Schalter die ***einzigen*** Schalter, die Ihr Skript akzeptiert. Wenn Sie ein Skript mit einem Schalter aufrufen, der nicht in dem Argument zu `getopts` definiert ist, gibt `getopts` einen `Unknown option`-Fehler aus (»unbekannte Option«), löscht die Option aus `@ARGV` und liefert ***falsch*** zurück. Dieses Verhalten können Sie sich zunutze machen, um sicherzustellen, dass Ihr Skript korrekt aufgerufen wird, und um im anderen Falle mit einer Meldung das Skript zu beenden. Dazu muss der Aufruf von `getopts` lediglich innerhalb eine `if`-Anweisung erfolgen:

```
if (! getopts('ab:c')) {
   die "Aufruf: meinskript -a -b:c datei\n";
}
```

Bedenken Sie auch, dass im Falle, dass `getopts` die Bearbeitung Ihrer Schalter aufgrund eines Fehlers mittendrin abbricht, alle Schaltervariablen, die zuvor gesetzt wurden, ihre Werte beibehalten, auch die inkorrekten Werte. Je nachdem wie robust Ihre Argumentenprüfung sein soll, können Sie diese Werte auch für den Fall überprüfen, dass `getopts` ***falsch*** zurückliefert (oder ganz abbricht).

getopt

Die Funktion `getopt` gleicht der Funktion `getopts` insofern, als beide ein Stringargument übernehmen, in dem die Schalter definiert sind, jedem dieser Argumente eine Variable `$opt_` zuweisen und diese dann aus `@ARGV` entfernen. Zwischen `getopt` und `getopts` gibt es jedoch drei wesentliche Unterschiede:

- Das Argument für `getopt` ist ein String mit Schaltern, denen ein Wert zugeordnet sein muss.
- Mit `getopt` erübrigt sich die Definition von Argumenten ohne Werte. Jede einbuchstabige Option ist erlaubt und für jede wird eine `$opt_`-Variable erzeugt.
- `getopt` liefert keinen (nützlichen) Wert zurück und gibt auch keine Fehlermeldung für unerwartete Optionen aus.

Angenommen Ihr Aufruf an `getopt` lautet wie folgt:

```
getopt('abc');
```

Diese Funktion geht davon aus, dass Ihr Skript mit einer beliebigen Kombination der drei Schalter `-a`, `-b` oder `-c` aufgerufen wird, wobei jedem Schalter ein Wert zugewiesen wird. Wird das Skript mit Schaltern aufgerufen, die keinen Wert haben, sollten Sie von `getopt` keine Warnung erwarten - statt dessen weist es der Variablen für den Schalter freudig das nächste Element in `@ARGV` zu, auch wenn das nächste Element ein anderer Schalter oder ein Dateiname ist, der als Datei ausgelesen werden sollte. Es obliegt Ihnen, herauszufinden, ob die Werte korrekt sind oder ob das Skript mit dem falschen Satz an Argumenten aufgerufen wurde.

Im Grunde genommen liegt der Hauptunterschied zwischen `getopt` und `getopts` darin, dass Sie bei `getopt` Ihre Optionen nicht deklarieren müssen, was jedoch die Fehlerbehandlung wesentlich erschwert. Ich ziehe in den meisten Fällen `getopts` vor, da ich so unnötiges Wertetesten vermeiden kann.

Skriptschalter auf dem Macintosh

Auf dem Mac gibt es keine Skript-Befehlszeile und deshalb können MacPerl-Skripts auch keine Befehlszeilenschalter annehmen, wie das bei Unix oder bei Windows- Skripten möglich ist (über Droplets können sie jedoch Dateinamenargumente entgegennehmen). Das Problem läßt sich auf mehreren Wegen umgehen: So könnten Sie zum Beispiel am Anfang Ihres Skripts zur Eingabe von Schalter auffordern und die Eingabe in `@ARGV` speichern (so dass Sie `getopt` verwenden können, um die Schalter innerhalb Ihres Skripts zu verarbeiten). Das folgende Codefragment zeigt hierzu ein Beispiel:

```
print "Schalter eingeben: ";
chomp($in = <STDIN>);
@ARGV = split(' ', $in);
```

Eine etwas ausgefeiltere Version, die das MacPerl-Modul und ein Dialogfeld verwendet, finden Sie als Teil von MacPerl FAQ unter `http://www.perl.com/CPAN- local/doc/FAQs/mac/MacPerlFAQ.html`:

```
if( $MacPerl::Version =~ /Application$/ ) {
        # Ausführung als Anwendung
        local( $cmdLine, @args );
        $cmdLine = &MacPerl::Ask( "Befehlszeilenargumente eingeben:" );
        require "shellwords.pl";
        @args = &shellwords( $cmdLine );
        unshift( @ARGV, @args );
    }
```

Der wahre Mac-Benutzer umgeht die Befehlszeilenschalter, indem er MacPerl-Module verwendet, um eigene Dialoge zu erstellen, in denen die Schalter als echte Benutzerschnittstellenelemente implementiert werden. Ein paar einfache Dialoge werden wir in Kapitel 18 besprechen. Weitere Informationen zu den Dialogen finden Sie in der MacPerl-Dokumentation.

Wenn Sie die MPW-Version von MacPerl verwenden, können Sie das bisher Gesagte vergessen, denn

hier haben Sie eine Befehlszeile. Also nutzen Sie sie!

Ein weiteres Beispiel

Im folgenden sehen Sie ein einfaches Beispiel (Listing 15.2), das eine Datei auf unterschiedliche Arten bearbeitet. Wie die Datei konkret verarbeitet wird, hängt von den verwendeten Schaltern ab.

Listing 15.2: Das Skript schalter.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use Getopt::Std;
4:  use vars qw($opt_r $opt_l $opt_s $opt_n);
5:
6:  if (! getopts('rlsn')) {
7:      die "Aufruf: schalter.pl -rlsn\n";
8:  }
9:
10: my @file = <>;
11:
12: if ($opt_s) {
13:     @file = sort @file;
14: }
15:
16: if ($opt_n) {
17:     @file = sort {$a <=> $b} @file;
18: }
19:
20: if ($opt_r) {
21:     @file = reverse @file;
22: }
23:
24: my $i = 1;
25: foreach my $line (@file) {
26:     if ($opt_l) {
27:         print "$i: $line";
28:         $i++;
29:     } else {
30:         print $line;
31:     }
32: }
```

Dieses Skript verwendet nur einfache Schalter ohne Werte (beachten Sie den Aufruf von `getopts` in Zeile 6; es stehen keine Doppelpunkte nach den Optionen). Die Schalter lauten `-r`, um den Inhalt der Datei umzudrehen, `-s`, um die Zeilen der Datei alphabetisch zu sortieren, `-n`, um die Zeilen numerisch zu sortieren, und `-l`, um die Zeilennummer auszugeben. Sie können die Optionen in der Befehlszeile kombinieren, auch wenn manche Kombinationen nicht besonders sinnvoll sind (`-sn` sortiert die Datei und sortiert sie dann erneut numerisch).

In Zeile 4 werden die Variablen vorab deklariert, so dass Sie nicht plötzlich aus dem Nichts auftauchen, wenn sie mit `getopts` erzeugt werden (und Beschwerden durch `use strict` auslösen).

Der Test in den Zeilen 6 bis 8 stellt sicher, dass das Skript mit den richtigen Optionen aufgerufen wird. Rutscht dabei eine undefinierte Option mit durch (zum Beispiel `-a` oder `-x`), dann bricht das Skript ab und gibt eine entsprechende Meldung aus.

Anschließend wird in verschiedenen `if`-Anweisungen geprüft, ob die Variablen `$opt_r`, `$opt_l`, `$opt_s` und `$opt_n` existieren, und dann werden je nach aufgerufener Option auf der Befehlszeile verschiedene Operationen durchgeführt. Alle Argumente, die keine Schalter sind, verbleiben nach dem Aufruf von `getopts` in `@ARGV` und werden mit dem Operator <> in Zeile 10 in das Skript eingelesen.

Vertiefung

In dieser Lektion habe ich Ihnen die Grundlagen der Ein- und Ausgabe vermittelt und gezeigt, wie man Dateisysteme verwaltet. Das, was Sie hier gelernt haben, sollte sich auf alle Ihre Perl-Programme, die Dateien und

Befehlszeilenargumente verwenden, übertragen lassen. In Kapitel 17 werden wir einen eingehenderen Blick auf das Dateisystem selbst werfen. In dem Vertiefungsabschnitt dieser Lektion möchte Ihnen aufzeigen, wo Sie weitere Informationen und Details zu fortgeschritteneren Aspekten der Ein- und Ausgabe und dem Umgang mit dem Dateisystem finden.

Alle vordefinierten Perl-Funktionen sind, wie ich bereits gesagt habe, in der ***perlfunc***- Manpage dokumentiert. Von Nutzen können aber auch die FAQs zu Dateien und Formaten in der Hilfsdokumentation `perlfaq` sein.

Mehr zu open und den Datei-Handles

Hier noch einige weitere Kurzschreibweisen und Eigenheiten der `open`-Funktion:

Sie können den Dateinamen beim Aufruf der `open`-Funktion weglassen, wenn - und nur in diesem Fall - einer Skalarvariablen, die den gleichen Namen wie das Datei- Handle trägt, bereits der Name der zu öffnenden Datei zugewiesen wurde. Zum Beispiel:

```
$FILE = "meinedatei.txt";
open(FILE) or die "Datei $FILE kann nicht geöffnet werden: $!\n";
```

Dies kann für Dateinamen, die geöffnet oder erneut geöffnet werden müssen, nützlich sein. Sie können die Variable zu Beginn Ihres Skripts setzen und dann den Dateinamen immer wieder verwenden.

Im Gegensatz zu dem, was ich Ihnen zu Beginn dieses Kapitels gesagt habe, können Sie eine Datei auch gleichzeitig zum Lesen und Schreiben öffnen. Verwenden Sie dazu das Sonderzeichen +> vor den Dateinamen:

```
open(FILE, "+>diedatei") or die "Datei kann nicht geöffnet werden: $!\n";
```

Da dies jedoch oft nur Verwirrung stiftet, ziehe ich es vor, getrennte Datei-Handles zu verwenden und das Lesen und Schreiben als getrennte Operationen zu betrachten.

Dateinamen, die mit einem Pipe-Zeichen (`|`) beginnen, fungieren als Befehl, und die Ausgabe wird über die Befehls-Shell Ihres Systems an diesen Befehl geleitet.

Die Fülle an Möglichkeiten, die Ihnen `open` bietet, finden Sie detailliert in der Hilfsdokumentation ***perlfunc***-Manpage beschrieben.

Weitere Dateifunktionen

Tabelle 15.2 enthält weitere vordefinierte dateibezogene Funktionen, die ich in dieser Lektion noch nicht beschrieben habe.

Funktion	Was sie bewirkt
`eof`	Liefert *wahr* zurück, wenn die nächste Zeileneingabe am Ende der Zeile erfolgt
`eof()`	Unterscheidet sich von `eof`; diese Version findet das Ende der letzten Datei für die Datei-Eingabe mit <>
`lstat`	Zeigt Informationen zu Links an
`pack`	Datenausgabe erfolgt in einer binären Struktur
`select`	Ändert das Standard-Datei-Handle für die Ausgabe (wird normalerweise für Formate verwendet, siehe Kapitel 20)
`stat`	Gibt verschiedene Informationen über eine Datei oder einen Datei-Handle aus
`truncate`	Löscht den Inhalt einer Datei oder eines Datei-Handle
`unpack`	Dateneingabe erfolgt von einer binären Struktur

Tabelle 15.2: Weitere E/ A-Funktionen

Eingabe/ Ausgabe für Experten

Die bisher in diesem Kapitel beschriebenen Eingabe- und Ausgabetechniken umfassen die einfache, zeilenorientierte gepufferte Ein- und Ausgabe über Datei-Handles, der Standard-Ein-/-Ausgabe oder der Standardfehlerausgabe. Wenn Sie an den fortgeschrittenen Möglichkeiten der Ein- und Ausgabe interessiert sind, sollten Sie die Dokumentation der verschiedenen anderen E/A-Funktionen, die Perl für Sie bereithält, lesen. Eine Übersicht finden Sie in Tabelle 15.3.

Funktion	Was sie bewirkt
`fcntl`	Datei-Steuerung (Unix-Funktion `fctrl(2)`)
`flock`	Datei sperren (Unix-Funktion `flock(2)`)
`getc`	Liest nächstes Byte ein
`ioctl`	TTY-Steuerung (Unix-Systemaufruf `ioctl(2)`)
`read`	Einlesen einer bestimmten Anzahl von Bytes (`fread(2)`)
`rewinddir`	Setzt die aktuelle (Eingabe-)Position an den Anfang des Verzeichnis-Handles
`seek`	Positioniert den Dateizeiger auf eine bestimmte Stelle in einer Datei (entspricht `fseek()` in C)
`seekdir`	Entspricht `seek` für Verzeichnis-Handles
`select`	Bereitet die Dateideskriptoren zum Einlesen vor (entspricht dem Unix-Befehl `select(2)`; nicht zu verwechseln mit `select` zum Setzen des Standard-Datei-Handles)
`syscall`	Aufruf eines Unix-Systemaufrufs (`syscall(2)`)
`sysopen`	Öffnet einen Datei-Handle mit Modus und Zugriffsberechtigungen
`sysread`	Liest mit Hilfe von `read(2)` eine bestimmte Anzahl von Bytes ein
`syswrite`	Schreibt mit Hilfe von `write(2)` eine bestimmte Anzahl von Bytes in den Datei-Handle
`tell`	Liefert die aktuelle Position des Dateizeigers
`telldir`	Entspricht `tell` für Verzeichnis-Handles
`write`	Schreibt einen formatierten Datensatz in einen Ausgabedatei-Handle (siehe Tag 20); `write` ist nicht das Gegenstück zu `read`

Tabelle 15.3: Weitere E/ A-Funktionen

Außerdem gibt es noch das Modul `POSIX`, das weitere Möglichkeiten für fortgeschrittene E/A-Operationen bietet (leider läßt sich dieses Modul nur unter Unix einsetzen). Weitere Informationen zu `POSIX` finden Sie in der ***perlmod***-Manpage.

DBM-Dateien

Perl bietet auch Unterstützung für Berkeley-Unix-DBM-Dateien (Datenbankdateien). Diese Dateien sind in der Regel kleiner und schneller anzusprechen als reine textbasierte Datenbanken. Weitere Informationen zu DBM finden Sie unter dem `DB_File`-Modul, der `tie`-Funktion und den diversen `Tie`-Modulen (`Tie::Hash`, `Tie::Scalar` und so weiter).

CPAN enthält eine Reihe von Modulen für die Arbeit mit Datenbanken - sei es, dass Sie selbst Datenbanken erstellen wollen, sei es, dass Sie Schnittstellen und Treiber für den Zugriff auf kommerzielle Datenbanken wie Oracle und Sybase benötigen. Für letztgenannte seien die Pakete DBD (Datenbanktreiber) und DBI (Datenbankschnittstelle) der Perl Database Initiative wärmstens empfohlen.

Zeitmarkierungen

Dateien und Verzeichnisse können mit sogenannten Zeitmarkierungen versehen werden. Dabei handelt es sich um Angaben, wann die Datei erzeugt, geändert oder zuletzt darauf zugegriffen wurde. Mit Hilfe von Dateitests (`-M` für Änderungen, `-A` für Zugriff und `-C` für Änderungen an den ***inode***-Informationen[1]) können Sie die Zeitmarkierungen überprüfen, mit Hilfe der `stat`-Funktion erhalten Sie detailliertere Informationen über die Zeitmarkierungen, und mit der `utime`-Funktion läßt sich die Zeitmarkierung einer Datei ändern. Das Verhalten der Tests und Funktionen kann dabei von Plattform zu Plattform variieren.

Alle Zeitangaben erfolgen in Sekunden, gemessen ab einem bestimmten Zeitpunkt; für Unix und Windows ist das der 1. Januar 1970, für den Macintosh der 1. Januar 1904. Zum Decodieren und Ändern von Zeitmarkierungen können auch die Funktionen `time`, `gmtime`, `localtime` und die Module `Time::Local` nützlich sein.

Zusammenfassung

In diesem Kapitel haben wir das, was Sie bisher über die Ein- und Ausgabe gelernt haben, vertieft und ergänzt. Dabei haben wir die Techniken, die beim Lesen aus der Standardeingabe und beim Schreiben in die Standardausgabe zur Anwendung kamen, auf das Einlesen und Schreiben von Dateien übertragen.

Zu Beginn standen Dateien und Datei-Handles im Zentrum unserer Betrachtung. Ich habe Ihnen gezeigt, wie Sie die Funktion `open` verwenden, um eine Datei zu öffnen und ein Datei-Handle zu erzeugen, über das Sie aus dieser Datei lesen oder in die Datei schreiben können. In Zusammenhang mit `open` haben Sie die Funktion `die` kennengelernt, die das Skript beendet und dabei eine Fehlernachricht an die Standardfehlerausgabe schickt.

Im weiteren Verlauf der Lektion sprachen wir über Skript-Argumente und Schalter: was passiert, wenn Sie ein Skript mit Argumenten aufrufen (sie werden in `@ARGV` abgelegt), und wie Sie vorgehen, um diese Argumente zu bearbeiten. Enthalten diese Argumente Schalter, bearbeiten Sie sie am besten mit dem Modul `Getopt::Std`. Damit können Sie Schalter für Ihr Skript definieren und bearbeiten und dann mit Hilfe spezieller Variablen prüfen, ob diese Schalter überhaupt existieren.

Unter anderem haben Sie folgende Funktionen in diesem Kapitel kennengelernt:

- `open` - erzeugt Datei-Handles
- `die` - beendet das Skript mit einer Fehlernachricht
- `binmode` - setzt einen Datei-Handle in den Binärmodus
- `close` - schließt einen Datei-Handle
- `getopts` - Teil des Moduls `Getopt`; deklariert und bearbeitet Argumente
- `getopt` - ebenfalls Teil des Moduls `Getopt`; behandelt Argumente

Fragen und Antworten

Frage:
Ich versuche, eine Datei zum Schreiben zu öffnen, aber es wird immer wieder die Funktion `die` ausgelöst, und ich kann mir nicht erklären, warum. Das Verzeichnis erlaubt den Lesezugriff, die Datei existiert noch nicht - es gibt keinen Grund, warum hier etwas schieflaufen sollte.

Antwort:
Haben Sie daran gedacht, dass >-Zeichen vor den Dateinamen zu setzen? Sie brauchen dieses Zeichen, um Perl mitzuteilen, dass in die Datei geschrieben werden soll. Andernfalls geht Perl davon aus, dass aus dieser Datei ausgelesen wird, und wenn es diese Datei dann nicht findet, ist es auch nicht in der Lage, sie zu öffnen.

Frage:
Ich möchte meine Datei mit einer Subroutine öffnen und dann den Datei-Handle an andere Subroutinen weiterreichen. Aber wenn ich das versuche, funktioniert es nicht. Warum?

Antwort:
Weil es so einfach nicht geht. Man kann zwar mit Typeglobs einige trickreiche Kniffe anwenden, um Symbolnamen zwischen Subroutinen weiterzureichen, doch ist dies ein Gebiet, das wieder seine ganz eigenen Probleme birgt. Am besten übergibt man Datei-Handles, indem man das Modul `FileHandle` nutzt, den Datei-Handle als Objekt erzeugt und dieses Objekt dann zwischen den Subroutinen weiterreicht.

Da wir jedoch noch nicht allzuviel über Objekte gelernt haben, könnten Sie (erst einmal) Ihr Datei-Handle global erzeugen und darauf in den Subroutinen Bezug nehmen. (Auf die Objekte werde ich in Kapitel 20 kurz zu sprechen kommen.)

Frage:
Ich versuche eine einfache textbasierte Datenbank in Perl zu lesen. Ich kenne das Format der Datei und weiß, wie man es dekodieren muss, um etwas damit anfangen zu können. Trotzdem ist die Eingabe, die ich erhalte, absolut unverständlich. Wo liegt der Fehler?

Arbeiten Sie unter Windows? Liegt die Datenbank-Datei im binären Format vor? Verwenden Sie die Funktion `binmode`, um sicherzustellen, dass Perl Ihren Datei- Handle in binärem Format liest.

Frage:
Ich arbeite mit MacPerl. Jetzt habe ich eine Datei mit Zahlen von einem Unix- System vor mir liegen, und wenn ich sie in mein Skript einlese, erhalte ich als Ergebnis einen einzigen großen String anstatt eines Arrays von einzelnen Strings. MacPerl ignoriert anscheinend die Neue-Zeile-Zeichen. Wie gehe ich weiter vor?

Antwort:
Das haben Sie ganz richtig diagnostiziert: MacPerl ignoriert die Neue-Zeile- Zeichen. Unix- und Macintosh-Systeme handhaben Zeilenendezeichen unterschiedlich. Unter Unix wird das Zeichen `\n` (Zeilenvorschub, ASCII 10) verwendet und unter Macintosh das Zeichen `\r` (Wagenrücklauf, ASCII 13). MacPerl ist nur zum Lesen von Macintosh-Dateien ausgelegt, so dass es ein Wagenrücklaufzeichen anstelle eines Zeilenvorschubs erwartet. (Falls es Sie interessiert, Windows/DOS verwendet beide).

Um dieses Problem zu umgehen, haben Sie zwei Möglichkeiten. Zum einen konvertieren FTP und die meisten Dateitransferprogramme heutzutage die Zeilenvorschubzeichen in Wagenrücklaufzeichen automatisch, wenn Sie eine Unix-Datei auf einen Mac verschieben. Und wenn nicht, gibt es viele Editoren, die das können. Nachdem Sie eine Unix-Datei in eine Mac-Datei konvertiert haben, dürfte das Problem nicht mehr auftauchen.

Antwort:
Die andere Lösung besteht darin, ganz oben in Ihrem Skript das Eingabedatensatz-Trennzeichen von MacPerl in ein Zeilenvorschubzeichen zu ändern:

```
$/ = "\n";
```

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist ein Datei-Handle? Wozu dienen die Datei-Handle `STDIN`, `STDOUT` und `STDERR`?
2. Worin liegt der Unterschied zwischen dem Erzeugen eines Datei-Handles zum Lesen, Schreiben oder Anhängen?
3. Wozu benötigt man die Funktion `die`? Warum sollten man sie zusammen mit `open` verwenden?
4. Wie lesen Sie Eingaben von einem Datei-Handle? Beschreiben Sie, wie sich Eingabe in skalarem Kontext, im skalaren Kontext einer `while`-Schleife und in einem Listenkontext verhalten.
5. Wie schicken Sie Ausgaben an ein Datei-Handle?
6. Worauf prüfen die folgenden Datei-Tests:

1. `-e`
1. `-x`
1. `-f`
1. `-M`
1. `-Z`

7. Wozu wird `@ARGV` verwendet? Was enthält die Variable?
8. Was ist der Unterschied zwischen `getopt` und `getopts`?
9. Welche Schalter erlauben die folgenden Aufrufe von `getopts`:
1. `getopts('xyz:');`
1. `getopts('x:y:z');`
1. `getopts('xXy');`
1. `getopts('xyz');`
1. `getopt('xyz');`

Übungen

1. Schreiben Sie ein Skript, das zwei Dateien (angegeben in der Befehlszeile) zeilenweise miteinander vermischt (eine Zeile von Datei 1, eine Zeile von Datei 2, eine Zeile von Datei 1 und so weiter). Schreiben Sie das Ergebnis in eine Datei namens `gemischt`.
2. Schreiben Sie ein Skript, das zwei nur dann Dateien mischt, wenn die Extensionen der Dateinamen identisch sind. Verwenden Sie die gleiche Extension auch für die Ergebnisdatei `gemischt`. Sind die Extensionen unterschiedlich, verlassen Sie das Skript mit einer Fehlermeldung. (HINWEIS: Verwenden Sie reguläre Ausdrükke, um die Extensionen der Dateinamen zu ermitteln).
3. Schreiben Sie ein Skript, das wie in der zweiten Übung zwei Dateien mischt. Das Skript soll eine einzige Option akzeptieren: `-o`. Ist die Option gesetzt, sollen auch Dateien mit unterschiedlichen Extensionen vermischt werden (die Extension der Ergebnisdatei können Sie frei wählen), und es wird keine Fehlermeldung ausgegeben.
4. Schreiben Sie ein Skript, das ein einziges String-Argument und eine beliebige Kombination der folgenden vier Schalter übernimmt: `-u`, `-s`, `-r` und `-c`. `-u` liefert den String in Großbuchstaben zurück, `-s` entfernt Interpunktionszeichen und Whitespace, `-r` dreht den String um, und `-c` zählt die Anzahl der Zeichen. Stellen Sie sicher, dass die Optionen in verschiedenen Kombinationen verwendet werden können, um unterschiedliche Effekte zu erzielen.
5. FEHLERSUCHE: Ein Skript wird folgendermaßen aufgerufen:

```
meinskript.pl -sz eingabedatei
```

1. Was ist falsch an diesem Skript? (HINWEIS: Die Prüfung von `$opt_z` liefert nicht ***wahr*** zurück).

```
    use strict;
use Getopt::Std;
use vars qw($opt_s $opt_z);
getopt('sz');
if ($opt_z) {
   # ...
}
```

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. In Perl verwendet man Datei-Handles, um Daten aus einer Quelle auszulesen oder Daten in ein Ziel zu schreiben (dabei kann es sich um eine Datei, die Tastatur, den Bildschirm, eine Netzwerkverbindung oder ein anderes Skript handeln). Die Datei- Handles `STDIN`, `SDTOUT` und `STDERR` beziehen sich auf die Standardeingabe, die Standardausgabe und die Standardfehlerausgabe. Datei-Handles zu allen anderen Dateien erzeugen Sie mit der Funktion `open`.
2. Durch spezielle Zeichen vor dem Namen der zu öffnenden Datei zeigt man an, ob die Datei zum Lesen, Schreiben oder Anhängen geöffnet werden soll. Standardmäßig werden Dateien zum Lesen geöffnet.
3. Die `die`-Funktion bricht das Skript behutsam ab und gibt dabei eine (hoffentlich) hilfreiche Fehlernachricht aus. Meistens wird die Funktion zusammen mit `open` verwendet, da man in dem Fall, dass eine Datei aus irgendwelchen Gründen nicht geöffnet werden kann, das Skript üblicherweise nicht weiter ausführen möchte. Es gehört daher zum guten Programmierstil, stets die Rückgabewerte von `open` zu überprüfen und `die` aufzurufen, falls beim Aufruf von `open` etwas schieflief.
4. Verwenden Sie zum Lesen der Eingabe aus einem Datei-Handle den Eingabeoperator <> und den Namen

des Datei-Handles. In einem skalaren Kontext liest der Eingabeoperator immer nur eine Zeile. Innerhalb einer `while`- Schleife weist er die einzelnen Zeilen nacheinander der Variablen `$_` zu. In einem Listenkontext wird die gesamte Eingabe bis zum Ende der Datei eingelesen.

5. Um eine Ausgabe an einen Datei-Handle zu schicken, braucht man die `print`- Funktion und den Namen des Datei-Handles. Beachten Sie, dass kein Komma zwischen dem Datei-Handle und dem, was ausgegeben werden soll, steht.
1. `-e` testet, ob die Datei existiert.
1. `-x` testet, ob es sich bei der Datei um eine ausführbare Datei handelt (normalerweise nur für Unix relevant).
1. `-f` testet, ob es sich bei der Datei um eine einfache Datei (und nicht ein Verzeichnis, einen Link oder etwas anderes) handelt.
1. `-M` prüft das Bearbeitungsdatum der Datei.
1. `-Z` testet, ob die Datei existiert und leer ist.
7. Die Array-Variable `@ARGV` speichert alle Argumente und Schalter, mit denen das Skript aufgerufen wurde.
8. Die Funktion `getopt` definiert Schalter mit Werten, akzeptiert aber jede beliebige Option. Die Funktion `getopts` deklariert die möglichen Optionen für das Skript und legt fest, ob diese Werte haben oder nicht. Ein weiterer Unterschied besteht darin, dass `getopts` den Wert ***falsch*** zurückliefert, wenn Fehler bei der Bearbeitung der Befehlszeilenschalter aufgetreten sind. `getopt` liefert keinen praktischen Wert zurück.
1. `getopts('xyz:')` definiert die Schalter `-x`, `-y` und `-z` mit einem Wert
1. `getopts('x:y:z')` definiert `-x` und `-y` mit Wert und `-z` ohne Wert
1. `getopts('xXy')` definiert `-x` und `-X` (beides sind separate Schalter) sowie `-y`. Keiner der Schalter hat einen Wert.
1. `getopts('xyz')` definiert `-x`, `-y` und `-z` alle ohne Werte.
1. `getopt('xyz')` definiert `-x`, `-y` und `-z` mit Werten sowie beliebige weitere einbuchstabige Schalter.

Antworten zu den Übungen

1. Hier ist eine Antwort:

```
    #!/usr/bin/perl -w
use strict;
my ($file1, $file2) = @ARGV;
open(FILE1, $file1) or
            die "Datei $file1 konnte nicht geoeffnet werden: $!\n";
open(FILE2, $file2) or
            die "Datei $file2 konnte nicht geoeffnet werden: $!\n";
open(MERGE, ">gemischt") or
           die "Die gemischte Datei konnte nicht geoeffnet werden: $!\n";
my $line1 = <FILE1>;
my $line2 = <FILE2>;
while (defined($line1) || defined($line2)) {
    if (defined($line1)) {
        print MERGE $line1;
        $line1 = <FILE1>;
    }
    if (defined($line2)) {
        print MERGE $line2;
        $line2 = <FILE2>;
    }
}
```

2. Hier ist eine Antwort:

```
    #!/usr/bin/perl -w
use strict;
my ($file1, $file2) = @ARGV;
my $ext;
if ($file1 =~ /\.(\w+)$/) {
    $ext = $1;
    if ($file2 !~ /\.$ext$/) {
  die "Extensionen sind nicht identisch.\n";
    }
}
open(FILE1, $file1) or
            die "Datei $file1 konnte nicht geoeffnet werden: $!\n";
open(FILE2, $file2) or
            die "Datei $file2 konnte nicht geoeffnet werden: $!\n";
```

```
open(MERGE, ">gemischt") or
           die "Die gemischte Datei konnte nicht geoeffnet werden: $!\n";
my $line1 = <FILE1>;
my $line2 = <FILE2>;
while (defined($line1) || defined($line2)) {
    if (defined($line1)) {
  print MERGE $line1;
  $line1 = <FILE1>;
    }
    if (defined($line2)) {
  print MERGE $line2;
  $line2 = <FILE2>;
    }
}
```

3. Hier ist eine Antwort:

```
    #!/usr/bin/perl -w
use strict;
use Getopt::Std;
use vars qw($opt_o);
getopts('o');
my ($file1, $file2) = @ARGV;
my $ext;
if ($file1 =~ /\.(\w+)$/) {
    $ext = $1;
    if ($file2 !~ /\.$ext$/) {
  if (!$opt_o) {
     die "Extensionen sind nicht identisch.\n";
  }
    }
}
open(FILE1, $file1) or
            die "Datei $file1 konnte nicht geoeffnet werden: $!\n";
open(FILE2, $file2) or
            die "Datei $file2 konnte nicht geoeffnet werden: $!\n";
open(MERGE, ">gemischt") or
           die "Die gemischte Datei konnte nicht geoeffnet werden: $!\n";
my $line1 = <FILE1>;
my $line2 = <FILE2>;
while (defined($line1) || defined($line2)) {
    if (defined($line1)) {
  print MERGE $line1;
  $line1 = <FILE1>;
    }
    if (defined($line2)) {
  print MERGE $line2;
  $line2 = <FILE2>;
    }
}
```

4. Hier eine mögliche Lösung:

```
    #!/usr/bin/perl -w
use strict;
use Getopt::Std;
use vars qw($opt_s $opt_r $opt_u $opt_c);
getopts('sruc');
my $str = $ARGV[0];
if ($opt_s) {
    $str =~ s/[\s.,;:!?'"]//g;
}
if ($opt_r) {
   $str = reverse $str;
}
if ($opt_u) {
   $str = uc $str;
}
if ($opt_c) {
   $str = length $str;
}
print "$str\n";
```

5. Das Skriptfragment verwendet die Funktion `getopt`. Im Gegensatz zu `getopts` geht diese Funktion davon aus, dass alle Schalter mit Werten versehen sind. Wenn also das Skript mit dem Schalter `-sz` aufgerufen wird, geht `getopt` davon aus, dass Sie den Schalter `-s` verwenden, der den Wert `z` hat. Der Schalter `-z` wird von `getopt` nie registriert. Verwenden Sie die Funktion `getopts`, um sicherzustellen, dass sowohl `$opt_s` und `$opt_z` gesetzt werden.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl für CGI-Skripts

In den letzten Jahren hat sich die Erstellung von CGI-Skripten zu einem der Hauptanwendungsbereiche von Perl entwickelt. CGI ist das Akronym für »Common Gateway Interface« und bezieht sich auf Programme und Skripts, die auf einem Webserver stehen und ausgeführt werden als Antwort auf entsprechende Eingaben eines Webbrowsers. Diese Eingaben können die Form von Formularen, komplexen Links oder bestimmten Arten von Image-Maps annehmen - ja eigentlich bedarf alles, was als Antwort nicht gerade eine ganz gewöhnliche Datei ist, einer Art von CGI- Skript.

Aufgrund der Popularität von Perl als CGI-Sprache möchte ich nicht darauf verzichten, Ihnen im Zusammenhang mit Perl eine kurze Einführung in CGI zu geben. Heute werden wird das bisher Erlernte anwenden, um CGI-Skripts für das Web zu erstellen. Im einzelnen erfahren Sie

- einiges Wissenswertes über CGI,
- wie die CGI-Kommunikation von Browser zu Server und wieder zurück abläuft,
- wie man ein komplettes CGI-Skript erstellt,
- etwas über das Modul `CGI.pm`,
- wie man mit Daten aus HTML-Formularen verarbeitet,
- wie man Daten ausgibt,
- wie man das Skript debuggt.

Bevor Sie starten

Um in Perl ein CGI-Skript zu schreiben, benötigen Sie drei Dinge:

- Einen Webserver, der Perl unterstützt, und Kenntnisse darüber, wie man CGI- Skripts auf diesem Server installiert
- Das Modul `CGI.pm`, das Teil Ihrer Perl-Version sein sollte (mehr dazu später)
- Grundkenntnisse über HTML

Sie benötigen nicht unbedingt das Modul `CGI.pm`, um CGI-Skripts mit Perl zu erstellen. Es gibt auch andere Hilfsskripte, mit denen Sie CGI programmieren können, oder Sie schreiben den ganzen zugrundeliegenden Code gleich selbst. Dies würde jedoch einen erheblich größeren Programmieraufwand bedeuten; es ist viel, viel einfacher, statt dessen `CGI.pm` einzusetzen. Wir tun es alle. Warum schließen Sie sich uns nicht an?

Aufgrund der großen Bandbreite an Webservern für verschiedene Plattformen und den Unterschieden zwischen ihnen, möchte ich der Diskussion, wie Sie Ihren Webserver für CGI fit machen, nicht allzuviel Platz einräumen. Umfangreiche Hilfe finden Sie in den zahlreichen Hilfedateien im Web:

- Die grundlegende CGI-Dokumentation, die sich größtenteils auf Unix-basierte Server bezieht (unter `http://hoohoo.ncsa.uiuc.edu./cgi/`).
- Die am häufigsten gestellten Fragen im Zusammenhang mit Perl für CGI (unter `http://www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html`).
- »The Idiot's Guide to Solving CGI Problems« - Der Idiotenführer für CGI-Probleme (unter `http://www.perl.com/CPAN-local/doc/FAQs/cgi/idiots-guide.html`).
- Für Windows: Die am häufigsten gestellten Fragen im Zusammenhang mit Perl für Win32 (unter `http:/www.activestate.com/support/faqs/win32/`). Hier finden Sie eine Fülle von Informationen zur

Programmierung, dem Aufsetzen und dem Einsatz von Perl für CGI. Aber auch zu Ihrem Webserver kann es spezielle Webserver-bezogene Informationen zur Erstellung und Konfiguration von CGI- Skripten mit Perl geben.

- Für MacPerl: In der Datei ***Readme.cgi***, die im Verzeichnis ***MacPerl CGI*** von MacPerl steht, finden Sie reichlich Informationen, wie man CGI auf Mac-basierten Servern zum Laufen bringt.

Sie können auch jederzeit die Dokumentation, die mit Ihrem Webserver ausgeliefert wurde, zu Rate ziehen. Und wenn Sie vorhaben, wirklich viel mit CGI zu arbeiten, sollten Sie vielleicht die Anschaffung von ***Sams Teach Yourself CGI Programming in a Week*** von Rafe Colburn in Erwägung ziehen. Das Buch wird Ihnen eine größere Hilfe sein und mehr Beispiele bieten, als ich es im Rahmen dieses Buches kann.

Um mit dem Rest dieser Lektion nicht allzu viele Schwierigkeiten zu haben, sollten Sie zumindest einige flüchtige Kenntnisse in HTML haben. Auch hier möchte ich Ihnen empfehlen, sich durch die Vielzahl der HTML-Tutorien im Web zu kämpfen oder das Buch ***Sams Teach Yourself Web Publishing with HTML 4 in 21 Days*** von Laura Lemay (das bin übrigens ich) durchzuarbeiten.

Allgemeines zu CGI

Beginnen wir diesen Abschnitt damit, dass ich Ihnen etwas Hintergrundwissen zu CGI vermittle und Ihnen erkläre, wo CGI in der Beziehung zwischen Webbrowser und Webserver einzuordnen ist.

CGI steht, wie bereits erwähnt, für Common Gateway Interface. Common (»gemeinsam, allgemein«) bedeutet, dass der gleiche Prozeß für viele verschiedene Arten von Webservern verwendet wird, und Gateway (»Tor«) resultiert daher, dass die Skripts früher einmal in der Regel als Tor zwischen dem Webserver und einem größeren Programm fungierten - zum Beispiel einer Datenbank oder einer Suchmaschine. Heutzutage hat CGI viel von seiner ursprünglichen Bedeutung verloren und bezieht sich einfach auf ein Skript oder Programm, das als Antwort auf eine Eingabe eines Webbrowsers ausgeführt wird.

Es gibt viele Probleme, die sich mit einem CGI-Skript lösen lassen, und es gibt für jedes Problem unzählige Möglichkeiten, das zugehörige Skript zu schreiben. Im Rahmen dieser Lektion konzentrieren wir uns auf einen typischen Anwendungsbereich von CGI-Skripten: die Bearbeitung von Daten, die als Teil eines HTML-Formulars empfangen wurden. In Abbildung 16.1 sehen Sie ein typisches Ablaufdiagramm, das zeigt, was passiert, wenn ein Benutzer über seinen Webbrowser ein Formular anfordert, es ausfüllt und wieder zurückschickt.

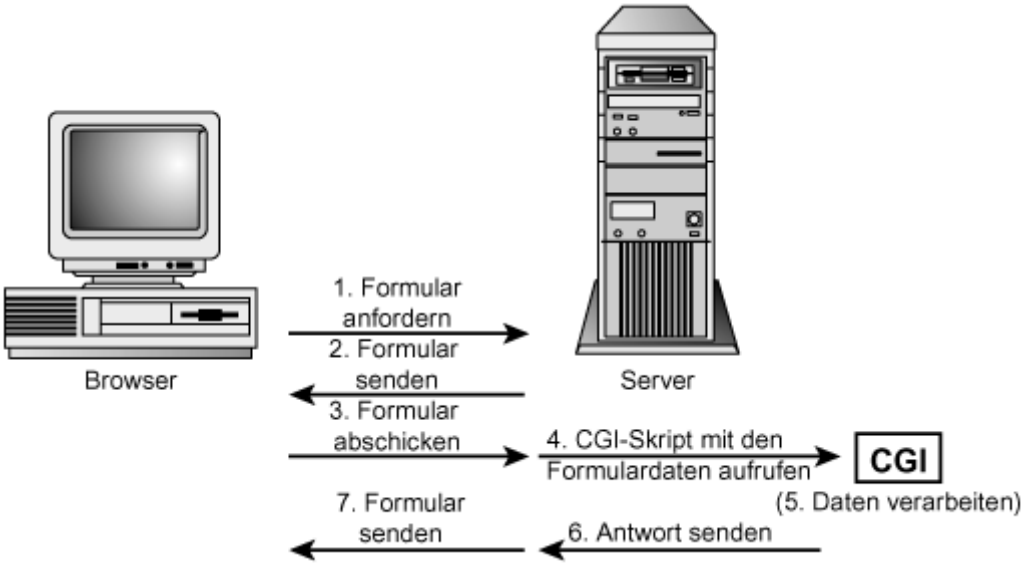

Abbildung 16.1: Der CGI-Prozeß

Und so sehen die Schritte im einzelnen aus:

- Der Benutzer fordert über seinen Webbrowser eine Seite mit einem Formular an.
- Der Server sendet das Formular. Das Formular selbst kann ein CGI-Skript sein, das den HTML-Code für das Formular erzeugt, oder einfach nur eine normale HTML-Datei.

- Der Benutzer füllt das Formular aus und klickt auf den Schalter **Abschicken**.
- Der Browser teilt die Formulardaten in Pakete auf und sendet sie an den Webserver.
- Der Webserver übergibt die Daten an das CGI-Skript.

- Das CGI-Skript dekodiert die Daten, verarbeitet sie in irgendeiner Form und liefert eine Antwort zurück (normalerweise eine andere HTML-Datei).
- Der Server sendet die Antwort an den Browser.

Das scheint nicht allzu kompliziert. Wichtig ist, dass Sie verstanden haben, wo das CGI-Skript in dem Datenfluß eingreift, woher die Daten für das CGI-Skript kommen und wohin sie gehen. Später wird dies noch von Bedeutung sein.

Ein CGI-Skript erstellen, vom Formular bis zur Antwort

Am besten erlernt man die CGI-Programmierung, indem man einfach loslegt - also, auf geht's! In diesem Abschnitt erstellen wir ein ganz einfaches CGI-Skript - das Gegenstück zu »Hallo Welt«. Dazu benötigen wir ein einfaches HTML-Formular, Perl für das CGI-Skript und das Modul `CGI.pm`, um alles zusammenzubinden. Das HTML- Formular gebe ich Ihnen vor, das Skript werden wir Zeile für Zeile erarbeiten.

Das Formular

In Listing 16.1 finden Sie den HTML-Code für ein einfaches HTML-Formular, das Sie nach Ihrem Namen fragt. In Abbildung 16.2 können Sie dann sehen, wie das Ganze auf dem Webbrowser ausgegeben wird.

Listing 16.1: Die Datei name.html

```
1:  <HTML>
2:  <HEAD>
3:  <TITLE>Tell Me Your Name</TITLE>
4:  </HEAD>
5:  <BODY>
6:  <FORM ACTION="/cgi-bin/name.pl">
7:  <P>Geben Sie Ihren Namen ein: <INPUT NAME="name">
8:  <P><INPUT TYPE="SUBMIT" VALUE="Abschicken!">
9:  </FORM>
10: </BODY>
11: </HTML>
```

Zwei wichtige Dinge möchte ich zu diesem HTML-Code anmerken:

- In Zeile 6 verweist das Attribut `ACTION` auf das CGI-Skript, das dieses Formular verarbeiten wird, wenn es an den Server geschickt wird. In unserem Beispiel heißt das Skript ***name.pl*** und steht auf dem Server im Verzeichnis ***cgi-bin*** (dem regulären Speicherort für CGI-Skripts; das Verzeichnis muss jedoch nicht bei allen Servern gleich lauten; es kann auch erforderlich sein, dass Sie erst die Erlaubnis einholen müssen, bevor Sie Ihre Skripts dort ablegen). Passen Sie den Pfad gegebenenfalls an.

Abbildung 16.2: Hallo!

Einige Webserver verlangen, dass Sie Ihre Skripts mit der Extension `.cgi` versehen, damit sie als CGI-Skripts erkannt werden. Da dies von Server zu Server unterschiedlich gehandhabt wird, sollten Sie vorsichtshalber Ihre Server-Dokumentation konsultieren.

- Zeile 7 definiert ein Textfeld als Formularelement (`<INPUT>`-Tag). Mit Hilfe des Attributs `NAME` geben Sie diesem Element einen Namen. Dies wird von Bedeutung sein, wenn Sie das CGI-Skript für das Formular erstellen.

Das Skript

Kommen wir jetzt zu dem Perl-Skript, das die Daten verarbeitet. Ein CGI-Skript wird in Perl grundsätzlich in der gleichen Weise aufgesetzt wie ein normales Perl-Skript, das von der Befehlszeile aus ausgeführt wird. Es gibt jedoch einige wesentliche Unterschiede, die darauf beruhen, dass Ihr Skript vom Webserver aufgerufen wird und nicht von Ihnen. So erhalten Sie zum Beispiel keine Optionen oder Dateinamen- Argumente über die Befehlszeile. Alle Daten, die Sie im Skript erhalten, kommen vom Webserver (oder werden von Ihnen selbst aus Dateien auf der Festplatte eingelesen). Die Ausgabe Ihres Skripts muss in einem bestimmten Format vorliegen - normalerweise HTML.

Zum Glück gibt es das Modul `CGI.pm`, geschrieben von Lincoln Stein, das mit der aktuellen Perl-Version ausgeliefert wird, Ihnen die CGI-Skripterstellung leichter macht und einen großen Teil der Probleme abfängt.

Lassen Sie uns mit den obersten Zeilen unseres CGI-Skripts anfangen:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
```

Die shebang-Zeile und `use strict` kennen Sie bereits, und die dritte Zeile `use CGI` sollte Sie auch nicht allzusehr überraschen. Das Tag `:standard` importiert, wie Sie sich vielleicht aus Kapitel 13, »Gültigkeitsbereiche, Module und das Importieren von Code«, erinnern, nur einen Teilbereich des CGI-Moduls und nicht das ganze Modul. Dieses Tag werden Sie wahrscheinlich im Zusammenhang mit `CGI.pm` häufig verwenden.

Die meisten CGI-Skripts bestehen aus zwei Teilen: Der erste Teil liest die Daten ein, die Sie von einem Formular oder dem Webbrowser erhalten haben, und verarbeitet sie. Der zweite Teil gibt eine Antwort aus, die in der Regel

im HTML-Format erzeugt wird. Da es in unserem Beispiel kaum etwas zu verarbeiten gibt, gehen wir gleich zum Ausgabeteil über.

Das erste, was wir ausgeben müssen, ist ein besonderer Header an den Webserver, der ihm mitteilt, welche Art von Datei Sie zurücksenden. Senden Sie eine HTML- Datei zurück, lautet der Typ `text/html`. Ist es eine einfache Textdatei, lautet der Typ `text/plain`. Für eine Grafik ist es `image/gif`. Diese Dateitypen sind alle als Teil der MIME-Spezifikation standardisiert, und wenn Sie tiefer in die CGI-Skripterstellung einsteigen, werden Sie sich mit zumindest einigen dieser Formate näher vertraut machen müssen. Das geläufigste Format ist zweifelsohne `text/html`. `CGI.pm` stellt Ihnen eine grundlegende Subroutine namens `print_header()` zur Verfügung, die den entprechenden Header für diesen Formattyp ausgibt.

```
print header();
```

Alle weiteren Ausgaben, die auf den Header folgen, müssen jetzt im HTML-Format sein. Sie können entweder reine HTML-Tags ausgeben, die Perl-Subroutinen von `CGI.pm` verwenden, um den HTML-Code zu erzeugen, oder beide Methoden miteinander kombinieren. Da ich mit HTML vertraut bin, ziehe ich die Subroutinen von `CGI.pm` nur dort vor, wo es mir Tipparbeit erspart, und verwende ansonsten normale `print`-Anweisungen. Sehen Sie hier den Rest unseres einfachen CGI-Skripts:

```
print start_html('Hallo!');
print "<H1>Hallo, ", param('name'), "!</H1>\n";
print end_html;
```

Die erste Zeile ruft die `CGI.pm`-Subroutine `start_html()` auf, die den ersten Teil einer jeden HTML-Datei ausgibt (die Tags `<HTML>`, `<HEAD>`, `<TITLE>` und `<BODY>`). Das String-Argument für `start_html()` ist der Titel der Seite. Sie können dieser Subroutine aber auch andere Argumente übergeben, um zum Beispiel Hintergrundfarbe, Schlüsselwörter und andere Besonderheiten des Headers zu setzen (mehr dazu im nächsten Abschnitt).

Die zweite Zeile ist eine reguläre `print`-Anweisung, die eine HTML-Überschrift (`<H1>`- Tag) zum Hallo-Sagen ausgibt. Zwischen dem öffnenden und dem schließenden Tag befindet sich der interessante Teil. Die Subroutine `param()` ist ebenfalls Teil von `CGI.pm` und dient dazu, an die Informationen zu gelangen, die Ihr Benutzer in das Formular eingegeben hat. Rufen Sie dazu `param()` mit dem Namen des Formularelements (aus dem `NAME`-Attribut des HTML-Tags des Formularelements) auf, und die Routine liefert den Wert zurück, den der Benutzer für das Element eingegeben hat. Mit dem Aufruf von `param('name')` erhalten wir den String, den der Benutzer in das Textfeld unseres Formulars eingegeben hat. Mit diesem Wert können wir dann die Antwort erzeugen.

Die dritte Zeile ist eine weitere Subroutine von `CGI.pm`, die lediglich die abschließenden HTML-Elemente (`</BODY>` und `</HTML>`) ausgibt und damit die Antwort komplettiert. Sehen Sie im folgenden das komplette Skript:

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
print header;
print start_html('Hallo!');
print "<H1>Hallo, ", param('name'), "!</H1>\n";
print end_html;
```

Vielleicht ist Ihnen aufgefallen, dass wir für die Ausgabe eigentlich keine besonderen Schritte unternommen haben. Wir haben lediglich einfache `print`-Anweisungen verwendet. Und doch geht die Ausgabe nicht an den Bildschirm, sondern zurück an den Browser. Damit ist ein CGI-Skript ein Paradebeispiel dafür, dass die Standardeingabe und -ausgabe nicht unbedingt die Tastatur oder der Bildschirm sein müssen. CGI-Skripts lesen Ihre Eingaben von der Standardeingabe und schreiben ihre Ausgabe in die Standardausgabe, nur das in diesem Falle die Standardeingabe und - ausgabe der Webserver ist. Sie müssen keine besonderen Vorkehrungen treffen, um mit dem Server zu kommunizieren; über die Standardwege klappt das reibungslos.

Was aber ist unsere Ausgabe? Wenn der Benutzer in das Formular `Fred` eingegeben hätte, würde die Ausgabe des CGI-Skripts samt Header und HTML-Daten wie folgt aussehen:

```
Content-type: text/html
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Hallo!</TITLE>
</HEAD><BODY><H1>Hallo, Fred!</H1>
```

```
</BODY></HTML>
```

Diese Daten werden über die Standardausgabe an den Webserver übergeben, der sie wiederum an den Webbrowser von Fred zurückgibt.

Das Skript testen

Bevor Sie ein Skript auf Ihren Webserver installieren, ist es angebracht, das Skript zu testen, um sicherzustellen, dass Sie keine groben Fehler gemacht haben. Mit Hilfe von `CGI.pm` können Sie das CGI-Skript zum Test von der Befehlszeile aus starten:

```
% name.pl
(offline mode: enter name=value pairs on standard input)
```

Nach dieser Zeile können Sie die Formulareingabe in Form von Name/Werte-Paaren simulieren. So müßten Sie zum Beispiel für das Hallo-Formular als Name `name` angeben, und der Wert wäre dann irgendein Name (zum Beispiel `Fred`). Ihre Eingabe sähe dann wie folgt aus:

```
name=Fred
```

Solange Ihre Eingabe keine Leerzeichen enthält, müssen Sie keine Anführungszeichen setzen. Nachdem Sie Ihre Name/Werte-Paare eingegeben haben, drücken Sie `[Strg]+[D]` (`[Strg]+[Z]` für Windows), um die Standardeingabe zu beenden. Das Skript wird dann ausgeführt, als ob es die Eingabe vom Formular erhalten hätte, und das Ergebnis wird auf dem Bildschirm ausgegeben.

Alternativ können Sie die Namen und Werte auch als Argumente in der Befehlszeile des Skripts eingeben:

```
% name.pl name=Fred
```

Das Skript wird dann diese Name/Werte-Paare als Eingabe betrachten und Sie nicht nach weiteren Eingaben fragen.

Nachdem Sie sich davon überzeugt haben, dass Ihr CGI-Skript so funktioniert, wie Sie es erwarten, besteht der letzte Schritt darin, das Skript auf dem Webserver zu installieren. Installation kann bedeuten, dass Sie Ihr Skript in einem besonderen Verzeichnis namens `cgi-bin` ablegen, die Skriptextension in `.cgi` umbenennen oder auf anderweitige Art und Weise dem Webserver signalisieren, dass es sich bei Ihrem Skript um ein CGI-Skript handelt. (Auch hier gilt, dass sich diese Anforderungen von Server zu Server unterscheiden können, so dass Sie auf alle Fälle in Ihrer Server- Dokumentation über Einzelheiten informieren sollten). Eventuell müssen Sie auch sicherstellen, dass Ihr Skript eine ausführbare Datei ist, oder spezielle Zugriffsberechtigungen vergeben. Und abschließend werden Sie noch die Original- HTML-Datei ändern müssen, so dass sie auf die aktuelle Position des Skripts zeigt.

Haben Sie all diese Probleme gelöst, sollte es Ihnen möglich sein, in die HTML-Datei mit dem Formular einen Namen einzugeben, den **Abschicken**-Schalter anzuklicken und vom CGI-Skript eine Antwort zu erhalten. In Abbildung 16.3 sehen Sie die Antwort als Webseite.

Abbildung 16.3: Hallo, die Antwort

CGI-Skripts mit CGI.pm entwickeln

Das Kernstück eines jeden CGI-Skripts ist das Modul `CGI.pm`. Sie können CGI-Skripts zwar auch in reinem Perl schreiben oder andere im Web verfügbare CGI-Programme verwenden, doch `CGI.pm` wird als Teil von Perl mit ausgeliefert, findet breite Unterstützung, ist robust, läuft plattformübergreifend und bietet Ihnen alles, was Sie bei Ihrer Arbeit mit CGI gebrauchen könnten. Auf `CGI.pm` zu verzichten, heißt sich das Leben als Entwickler von CGI-Skripten unnötig schwer zu machen.

Im vorangehenden Abschnitt habe ich Ihnen ein wirklich einfaches Beispiel für den Einsatz von `CGI.pm` gezeigt. In diesem Abschnitt möchte ich Sie mit dem Inhalt dieses Moduls näher vertraut machen, so dass Sie hinterher wissen, was Sie alles damit anfangen können.

Woher bekommt man das CGI.pm-Modul?

Wenn Sie eine aktuelle Version von Perl haben, stehen die Chancen gut, dass `CGI.pm` dabei ist. Sind Sie sich nicht sicher, können Sie in dem *`lib`*-Verzeichnis Ihrer Perl- Installation nachschauen, ob das Modul dort abgelegt wurde. Ist es nicht vorhanden, können Sie es von der Website unter `http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html` herunterladen; es ist aber auch als Teil von CPAN verfügbar. Auf der `CGI.pm`-Seite finden Sie genau erklärt, wie Sie bei der Installation dieser Datei vorzugehen haben, aber kurz gesagt, müssen Sie dazu lediglich die Datei `CGI.pm` in Ihr Perl-Verzeichnis `lib` kopieren.

Das `CGI.pm`-Paket enthält neben `CGI.pm` noch ein weiteres Modul namens `Carp`. `Carp` gehört in das CGI-Verzeichnis unter dem ***lib***-Verzeichnis von Perl (falls es noch nicht dort abgelegt wurde). `Carp` wird verwendet, um hilfreiche Fehlermeldungen in Ihren Webserver-Protokollen oder dem Webbrowser auszugeben. Für das Debuggen von CGI-Skripten ist das Modul von unschätzbarem Wert, was Sie schnell feststellen werden, wenn Sie später einmal intensiver in die Arbeit mit CGI einsteigen (weiter unten erfahren Sie gleich noch etwas mehr über `Carp`).

Sowohl zu `CGI.pm` als auch zu `Carp.pm` gibt es eine Dokumentation im POD-Format. Um sich diese anzuschauen, brauchen Sie nur `perldoc CGI` einzutippen (MacPerl- Benutzer können die ***shuck***-Anwendung nutzen). Sie finden die aktuelle Dokumentation aber auch im Web unter `http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html`.

CGI.pm verwenden

Um CGI.pm in Ihren Perl-Skripten zu verwenden, importieren Sie es wie jedes andere Modul. CGI.pm weist mehrere Import-Tags auf, die Sie verwenden können, unter anderem:

- :cgi - importiert Unterstützung für das CGI-Protokolls selbst, beispielsweise param().
- :html2 - importiert Unterstützung für die Erzeugung von HTML2-Tags, beispielsweise start_html() und end_html()
- :form - importiert Unterstützung zur Erzeugung von Formularelementen
- :standard - importiert alle Elemente von :cgi, :html2 and :form
- :html3 - importiert Unterstützung für HTML 3.0
- :netscape - importiert Unterstützung für die Netscape-Version von HTML
- :html - importiert alle Elemente von :html2, :html3 und :netscape
- :all - importiert alles

CGI.pm ist so implementiert, dass Sie es sowohl objektorientiert verwenden können (unter Verwendung eines CGI-Objekts dessen Methoden man aufruft) als auch durch Aufruf einfacher Subroutinen. Wenn Sie einen der oben angeführten Import-Tags verwenden, stehen Ihnen die Subroutinennamen in Ihrem Skript zur Verfügung. Wenn Sie keine Import-Tags verwenden, wird davon ausgegangen, dass Sie die objektorientierte Version von CGI.pm nutzen möchten.

Formulareingaben verarbeiten

Der vielleicht größte Nutzen, den Ihnen CGI.pm bietet, besteht darin, dass dieses Modul Formulareingaben verarbeitet, die der Webbrowser an den Webserver schickt und der Server an das Skript weiterleitet. Diese Eingaben kommen vom Browser in einer ganz besonderen codierten Form, manchmal über die Standardeingabe und manchmal als Schlüsselargumente mit nichtalphanumerischen Zeichen im Hex-Code. Wenn Sie Ihren eigenen CGI-Prozessor von Grund auf selbst schreiben wollten, müßten Sie sich um die Decodierung selbst kümmern (und, glauben Sie mir, das macht keinen allzu großen Spaß). Mit CGI.pm bleibt Ihnen all das erspart, und Sie müssen sich lediglich um die tatsächlichen Eingabewerte kümmern, um die es ja auch eigentlich geht.

Die Eingabe, die Sie von einem Formular erhalten, besteht aus Schlüssel/Wert- Paaren. Der Schlüssel ist der Name des Formularelements (im HTML-Code durch das NAME-Attribut definiert), der Wert ist das, was der Benutzer letztendlich eingegeben, ausgewählt oder im Formular markiert hat. Der Wert, den Sie erhalten, hängt vom Typ des Formularelements ab. Einige Formularelemente - wie zum Beispiel Textfelder - liefern einfache Strings, andere - zum Beispiel Markierungsfelder - liefern nur *Ja* oder *Nein*. Popup-Menüs und scrollbare Listen, die mit dem HTML-Tag <SELECT> erzeugt wurden, können sogar mehrere Werte enthalten.

Das CGI.pm-Modul speichert diese Schlüssel und Werte in einem Parameter-Array. Mit der Subroutine param() können Sie auf die Elemente im Array zugreifen. Ohne irgendwelche Argumente liefert param() nur eine Liste der Schlüssel im Parameter- Array zurück (die Namen der Formularelemente). Wenn das CGI-Skript sowohl die ursprüngliche HTML-Seite mit dem Formular als auch das Ergebnis erzeugt, kann man anhand dieser Liste feststellen, ob das Formular vollständig ausgefüllt wurde. Auch beim Debuggen lohnt es sich param() ohne Argumente aufzurufen, um die Schlüssel und Werte des Formulars auszugeben. Der Code dazu würde folgendermaßen lauten:

```
foreach $key (param()) {
   print "$key hat den Wert ", param($key), "\n";
}
```

Beachten Sie, dass die Parameter im Array in der gleichen Reihenfolge stehen, wie sie vom Browser ursprünglich gesendet wurden. In den meisten Fällen entspricht das der Reihenfolge, in der sie auf der Seite erscheinen. Da es jedoch dafür keine Garantie gibt, sind Sie am besten beraten, wenn Sie auf jedes Formularelement explizit Bezug nehmen - falls Ihnen die Reihenfolge wichtig ist.

Wird die Subroutine param() mit dem Namen eines Formularelements als Argument aufgerufen, liefert Sie entweder den Wert dieses Formularelements zurück oder undef, wenn zu diesem Formularelement kein Wert geschickt wurde. Auf diese Art und Weise werden Sie param() wahrscheinlich am häufigsten in Ihren CGI-Skripten aufrufen. Der Schlüssel, den Sie als Argument für param() verwenden, muss mit dem Namen des Formularelements in der HTML-Datei genau übereinstimmen. Um zum Beispiel den Wert eines Textfeldes zu erhalten, das als <INPUT NAME="foozle"> definiert wurde, müssen Sie param('foozle') eingeben. Meistens erhalten Sie einen einfachen skalaren Wert als Antwort. Manche Formularelemente, bei denen Mehrfachauswahl

möglich ist, geben jedoch eine Liste der gewählten Optionen zurück. Es ist Ihre Aufgabe, die verschiedenen Werte, die Sie aus einem Formular zurückerhalten, in dem CGI-Skript zu verarbeiten.

HTML-Code erzeugen

Ein Großteil des CGI-Skripts besteht in der Regel aus dem HTML-Code, der für die Antwort erzeugt wird. Die Skripts, die wir für CGI schreiben, enthalten wahrscheinlich mehr `print`-Anweisungen als alle anderen Skripts, die wir bisher geschrieben haben.

Um eine HTML-Ausgabe zu erzeugen, geben Sie einfach, wie bei jedem anderen Skript, die auszugebenden Zeilen weiter an die Standardausgabe. Sie können dazu mehrere Wege einschlagen:

- einzelne `print`-Anweisungen
- »hier«-Dokumente
- Subroutinen aus `CGI.pm`

Ausgabe mit print

Für den ersten Weg rufen Sie `print` einfach mit dem auszugebenden HTML-Code auf - so wie Sie es bereits die ganze Zeit gemacht haben:

```
print "<HTML><HEAD><TITLE>Dies ist eine Webseite</TITLE></HEAD>\n";
print "<BODY BGCOLOR=\"white\">\n";
# und so weiter
```

Ausgabe mit Hier-Dokumenten

`print`-Anweisungen bereiten zwar keine Probleme, können aber manchmal recht unhandlich sein, besonders wenn die Ausgabe recht umfangreich ist oder Sie es mit verschachtelten Anführungszeichen zu tun haben (wie bei dem Wert »`white`« im obigen Beispiel). Wenn Sie einen größeren HTML-Block auszugeben haben, können Sie von einer speziellen Perl-Option, den sogenannten »Hier-Dokumenten«, Gebrauch machen. Der Name mutet seltsam an, besagt aber einfach: »Gib alles bis ***hier*** aus.« Ein Hier-Dokument könnte zum Beispiel folgendermaßen aussehen:

```
print <<EOF;
Diese Zeilen werden so ausgegeben
wie sie hier erscheinen; Sie benötigen keine zusätzlichen
print-Anweisungen, "Anführungszeichen mit Escape-Zeichen"
oder besonderen Neue-Zeile-Zeichen. So wie Sie es hier sehen.
EOF
```

Dieser kurze Perl-Code würde folgendes Ergebnis erzeugen:

```
Diese Zeilen werden so ausgegeben
wie sie hier erscheinen; Sie benötigen keine zusätzlichen
print-Anweisungen, "Anführungszeichen mit Escape-Zeichen"
oder besonderen Neue-Zeile-Zeichen. So wie Sie es hier sehen.
```

Mit anderen Worten, der ausgegebene Text entspricht fast genau dem Text in dem Skript. Die am Anfang stehende `print`-Anweisung im Hier-Dokument legt fest, wie weit gelesen und ausgegeben werden soll. Zur Kennzeichnung des Endes des Hier- Dokuments wird ein Endemarker definiert, bei dem es sich entweder um ein Wort, das in der Sprache keine anderweitige Bedeutung hat, oder um einen String in Anführungszeichen handeln kann. Ich habe in obigem Beispiel `EOF` (End of File für »Ende der Datei«) gewählt, da dies eine nette, kurze und allgemein übliche Abkürzung ist, die sich gut vom Rest des Skripts abhebt.

Wenn Sie den Endemarker in Anführungszeichen setzen, legt die Art der Anführungszeichen fest, wie der Text innerhalb des Hier-Dokuments bearbeitet wird. Ein einfaches Wort, wie das von mir oben verwendete `EOF`, erlaubt Variableninterpolation - so als ob der Text innerhalb des Hier-Dokuments in doppelten Anführungszeichen stehen würde. Den gleichen Effekt erzielen Sie, wenn Sie das Wort in doppelte Anführungszeichen setzen. Ein in einfache Anführungszeichen geklammerter Endemarker unterdrückt die Variableninterpolation wie bei allen Strings in einfachen Anführungszeichen.

Das Ende eines Hier-Dokuments wird von dem Endemarker (Wort oder String) angezeigt, mit dem das Hier-Dokument gestartet wurde, allerdings ohne die Anführungszeichen. Der Endemarker steht allein auf einer Zeile ohne irgendwelche führenden oder angehängten Zeichen oder Whitespaces. Nach dem Endemarkierer können Sie ein weiteres Hier-Dokument starten, wieder auf `print`-Anweisungen zurückgreifen oder beliebige anderen Zeilen in Perl-Code schreiben.

Weitere Informationen zu den Hier-Dokumenten finden Sie in der ***perldata***-Manpage.

Ausgabe mit CGI.pm-Subroutinen

Die dritte Möglichkeit, HTML-Code in einem CGI-Skript zu erzeugen, ist die Verwendung der dafür vorgesehenen `CGI.pm`-Subroutinen. Für die meisten HTML- Tags gibt es in `CGI.pm` äquivalente Perl-Subroutinen. Diese `CGI.pm`-Subroutinen haben den Vorteil, dass Sie auf diese Weise Variablenreferenzen einfügen (mittels Strings in doppelten Anführungszeichen) und schnell bestimmte HTML-Elemente, wie zum Beispiel Formularelemente, erzeugen können. Außerdem verfügt `CGI.pm` über die Subroutinen `start_html()` und `end_html()`, mit denen Anfang und Ende einer HTML-Datei ausgegeben werden. Alle Subroutinen zur Erzeugung von HTML-Code liefern Strings zurück. Um diese Strings tatsächlich auszugeben, müssen Sie sie einer `print`-Anweisung übergeben.

Manche Subroutinen erzeugen einzelne Tags, die keine Argumente übernehmen (zum Beispiel `p()` und `hr()`, die `<P>` und `<HR>` erzeugen). Andere wiederum erzeugen paarweise öffnende und schließende Tags. Tags dieser Art übernehmen ein oder mehrere String-Argumente für den Text zwischen das öffnende und das schließende Tag.

Subroutinen können ineinander verschachtelt werden:

```
h1('Dies ist eine Überschrift');    # <H1>Dies ist eine Überschrift</H1>
b('Bold');                          # <B>Bold</B>
b('mal fett mal ', i('kursiv'));        # <B>mal fett mal <I>kursiv</I></B>
ol(
  li('erstes Element'),
  li('zweites Element'),
  li('drittes Element'),
);
```

Wenn das auszugebende HTML-Tag selbst Attribute enthält, stellen Sie diese in geschweifte Klammern `{}` und trennen Sie Name und Wert des Attributs durch die Zeichen `=>` (wie bei einem Hash):

```
a({href=>"index.html", name=>"foo"}, "Homepage");
# <A HREF="index.html" NAME="foo">Homepage</A>
```

Fast alle HTML-Tags, die Sie in einer HTML-Datei finden können, sind als Subroutine verfügbar. Für welche Tags Ihnen Subroutinen zur Verfügung stehen, hängt allerdings davon ab, welche Gruppe von Subroutinen Sie in die `use CGI`-Zeile importieren. `CGI.pm` enthält außerdem einen besonders robusten Satz an Subroutinen zur Erzeugung von weiteren Formularelementen. Ich kann hier leider nicht auf alle einzeln eingehen, deshalb möchte ich Sie auf die `CGI.pm`-Dokumentation verweisen.

Das Ergebnis debuggen

Im Verlauf unseres Hallo-Beispiels haben Sie bereits eine Möglichkeit kennengelernt, wie man Skripts vor der Installation auf dem Server debuggt. Dabei haben wir die CGI-Eingabe als Name/Wert-Paar entweder über die Skript-Befehlszeile oder über die Standardeingabe eingegeben. Dieser Mechanismus ist unschätzbar, wenn es darum geht, kleinere Fehler aufzudecken, die sich beim Schreiben der CGI-Skripts eingeschlichen haben. Wenn Sie das Skript von der Befehlszeile ausführen, können Sie auch den Perl-Debugger verwenden, um sicherzustellen, dass Ihr Skript ordnungsgemäß läuft, bevor es installiert wird.

Irgendwann ist es jedoch soweit, dass Sie das CGI-Skript installieren und vor Ort ausführen müssen, um seine ordnungsgemäße Funktionsweise sicherzustellen. Wenn das Skript erst einmal installiert ist, kann das Debuggen allerdings Schwierigkeiten bereiten, da Fehler normalerweise in wenig hilfreichen Meldungen (wie `Server Error 500`) dem Browser mitgeteilt werden oder in Fehlerprotokollen landen, die weder Identifizierer noch Zeitmarkierungen haben, anhand derer man sehen könnte, welcher Fehler auf Sie zurückgeht.

An dieser Stelle kommt das Modul `CGI::Carp` ins Spiel. `CGI::Carp` ist Bestandteil von `CGI.pm` und sollte deshalb

wie letzteres Modul auch Teil Ihrer Standard-Perl-Version sein. (Überprüfen Sie das, indem Sie in dem CGI-Unterverzeichnis des Perl- Verzeichnisses ***lib*** danach suchen. Es gibt auch noch ein reguläres `Carp`-Modul, das verwandt, aber nicht dasselbe ist.) `Carp` wird verwendet, um Fehlermeldungen für CGI- Skripts zu erzeugen, die Ihnen dann beim Debuggen dieser Skripts sehr dienlich sein können. Vor allem das Schlüsselwort `fatalsToBrowser` kann für das Debuggen sehr nützlich sein, da es alle Perl-Fehler in dem CGI-Skript als HTML-Code ausgibt, der als Antwort auf das abgeschickte Formular in dem Browser angezeigt wird, der wiederum das Formular angefordert hat. Um diese Fehler als Echo an den Browser zu schicken, fügen Sie folgende `use`-Zeile in den Anfang Ihres Skripts auf:

```
use CGI::Carp qw(fatalsToBrowser);
```

Abgesehen von `fatalsToBrowser` enthält das `CGI::Carp`-Modul neue Definitionen für die Funktionen `warn()` und `die()` (und fügt die Subroutinen `croak()`, `carp()` und `confess()` hinzu), so dass Fehler mit vernünftigen Identifizierern ausgegeben werden und im dem Fehlerprotokoll Ihres Webservers erscheinen. (Näheres zu `croak()`, `carp()` und `confess()` finden Sie in der Dokumentation zum Standard-`Carp`-Modul.) Zusammengefaßt läßt sich sagen, dass `CGI::Carp` besonders nützlich zum Debuggen Ihrer CGI-Skripts auf dem Server ist.

Ein Beispiel: Umfrage

Unser »Hallo Welt«-Beispiel zu Beginn hat Ihnen vielleicht einen Vorgeschmack darauf gegeben, wie sich CGI-Skripts einsetzen lassen. Um wirklich von Nutzen zu sein, ist es aber zu einfach. Deshalb wollen wir im folgenden ein wesentlich komplexeres Skript betrachten, das eine webbasierte Umfrage verarbeitet. Das Skript zeichnet alle aus der Umfrage stammenden Daten auf, indem es die Eingabe des aktuellen Formulars verarbeitet und daraus Ergebnistabellen erzeugt, die auf allen bis dato eingesandten Daten basieren. In Abbildung 16.4 sehen Sie das Formular unserer kleinen Umfrage.

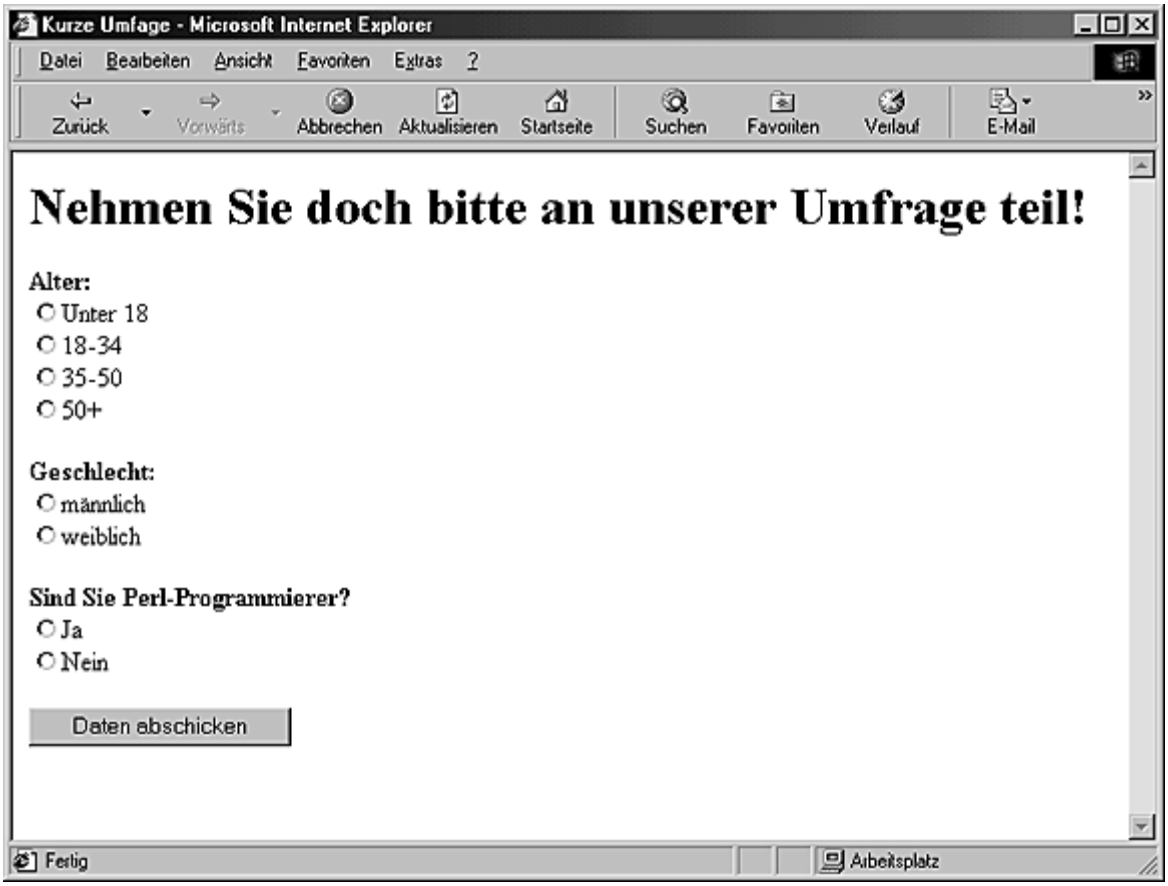

Abbildung 16.4: Eine einfache Web-Umfrage

Nachdem alle Fragen der Umfrage beantwortet wurden, verarbeitet das CGI-Skript diese Eingaben, fügt sie zu den bereits erhaltenen Daten hinzu und erzeugt eine Reihe von Tabellen, wie in Abbildung 16.5 zu sehen.

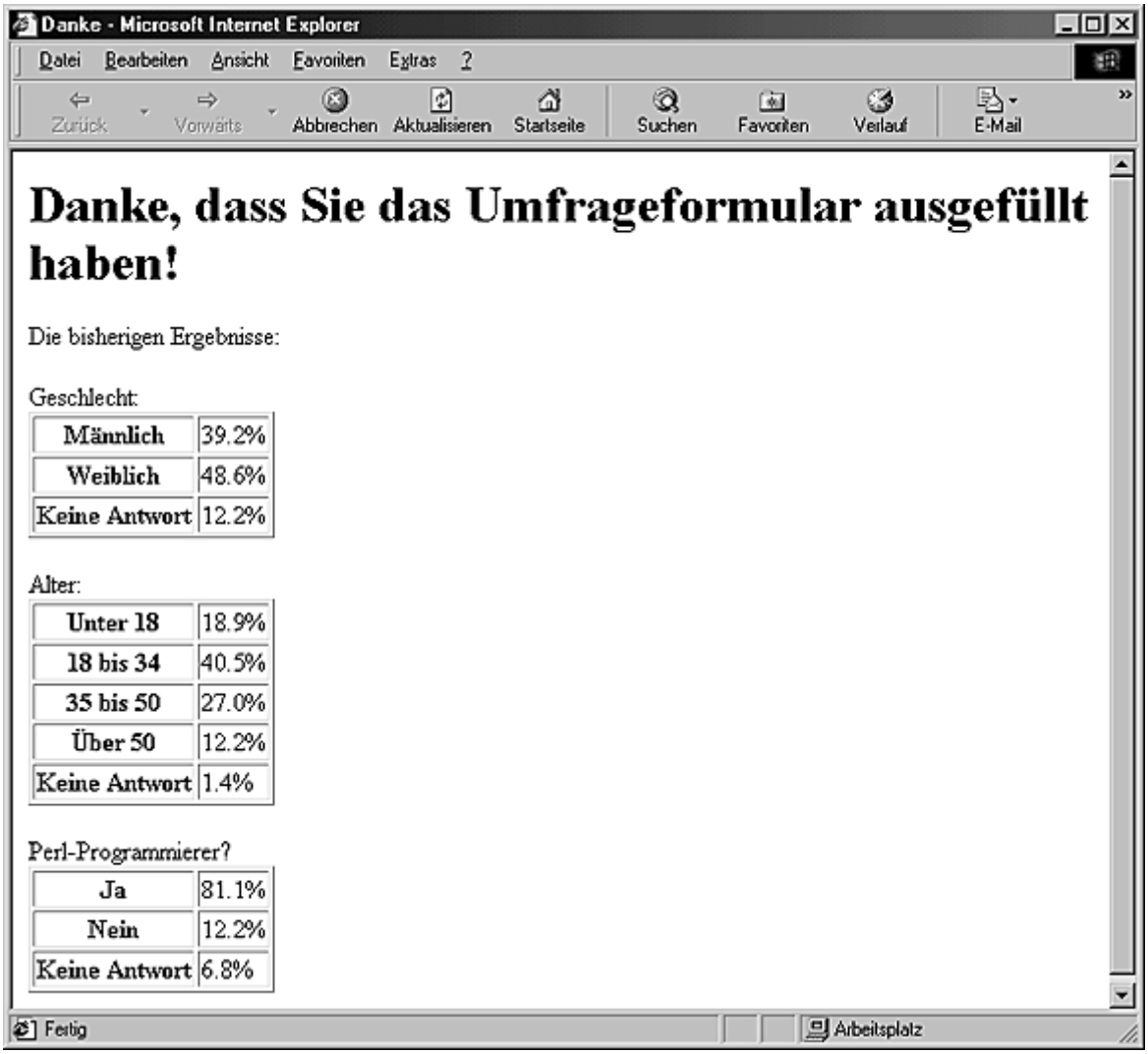

Abbildung 16.5: Die Ergebnisse der Web-Umfrage

Alle Daten der Umfrage werden in einer separaten Datei auf dem Webserver gespeichert. Über das CGI-Skript werden Sie diese Datendatei öffnen, lesen und in die Datei schreiben.

Das Formular

Beginnen wir mit dem HTML-Code für das Formular, damit Sie lernen, mit welchen Werten Sie es in dem CGI-Skript zu tun bekommen. In Listing 16.2 sehen Sie diesen HTML-Code:

Listing 16.2: survey.html

```
<HTML>
<HEAD>
<TITLE>Kurze Umfage</TITLE>
</HEAD>
<BODY>
<H1>Nehmen Sie doch bitte an unserer Umfrage teil!</H1>
<FORM ACTION="/cgi-bin/survey.pl">
<P><STRONG>Alter: </STRONG><BR>
<INPUT TYPE="radio" NAME="age" VALUE="under18">Unter 18<BR>
<INPUT TYPE="radio" NAME="age" VALUE="18to34">18-34<BR>
<INPUT TYPE="radio" NAME="age" VALUE="35to50">35-50<BR>
<INPUT TYPE="radio" NAME="age" VALUE="50plus">50+
<P><STRONG>Geschlecht: </STRONG><BR>
<INPUT TYPE="radio" NAME="sex" VALUE="male">männlich<BR>
<INPUT TYPE="radio" NAME="sex" VALUE="female">weiblich
<P><STRONG>Sind Sie Perl-Programmierer? </STRONG><BR>
<INPUT TYPE="radio" NAME="perl" VALUE="yes">Ja<BR>
<INPUT TYPE="radio" NAME="perl" VALUE="no">Nein
```

```
<P><INPUT TYPE="submit" VALUE="Daten abschicken">
</FORM>
</BODY>
</HTML>
```

Einige wenige, aber wichtige Punkte möchte ich zu diesem HTML-Code anmerken. Im Gegensatz zum vorangehenden Beispiel finden Sie hier keine Textfelder, sondern Gruppen von Optionsfeldern (englisch: »radio buttons«). Beachten Sie, dass alle Optionsfelder einer Gruppe den gleichen Namen haben (so heißen zum Beispiel alle vier Optionsfelder in der Gruppe Alter »`age`«). Damit wird verhindert, dass mehr als ein Schalter in einer Gruppe zur Zeit ausgewählt werden kann. Das bedeutet wiederum, dass in der Eingabe für Ihr Skript nur ein Wert pro Gruppe erscheint. Sie müssen die Eingabe mit allen möglichen Werten vergleichen, um herauszufinden, welche Option gewählt wurde.

Das Skript

Das CGI-Skript, das dieses Formular verarbeitet, hat vier Hauptaufgaben zu erledigen:

- Es öffnet die Datei mit den Umfragedaten und liest diese Daten in einen Hash ein.
- Es verarbeitet die Eingaben aus dem Formular und fügt die neuen Daten zu den alten hinzu.
- Es schreibt die neuen Daten in die Datei.
- Es erzeugt HTML-Code für die tabellarische Ausgabe, die aus den aktuellen Umfragedaten erzeugt wurde.

Listing 16.3 zeigt den Code für unser CGI-Skript mit Namen `umfrage.pl`.

Listing 16.3: Das Skript umfrage.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use CGI qw(:standard);
4:
5:  my $results = 'umfrage_ergebnisse.txt';
6:  my %data = ();
7:  my $thing = '';
8:  my $val = 0;
9:
10: open(RESULTS, $results) or
                      die "Ergebnisdatei konnte nicht geoeffnet werden: $!";
11: while (<RESULTS>) {
12:     ($thing, $val) = split(' ');
13:     $data{$thing} = $val;
14: }
15: close(RESULTS);
16:
17: # Gesamtsumme
18: $data{total}++;
19:
20: # Alter
21: if (!param('age')) { $data{age_na}++ }
22: else {
23:     if (param('age') eq 'under18') { $data{age_under18}++; }
24:     elsif (param('age') eq '18to34') { $data{age_18to34}++; }
25:     elsif (param('age') eq '35to50') { $data{age_35to50}++; }
26:     elsif (param('age') eq '50plus') { $data{age_50plus}++; }
27: }
28:
29: # Geschlecht
30: if (!param('sex')) { $data{sex_na}++ }
31: else {
32:     if (param('sex') eq 'male') { $data{sex_m}++; }
33:     elsif (param('sex') eq 'female') { $data{sex_f}++; }
34: }
35:
36: # Perl
37: if (!param('perl')) { $data{perl_na}++ }
38: else {
39:     if (param('perl') eq 'yes') { $data{perl_y}++; }
40:     elsif (param('perl') eq 'no') { $data{perl_n}++; }
41: }
```

```
42:
43: open(RESULTS, ">$results") or
                  die "In Ergebnisdatei kann nicht geschrieben werden: $!";
44: foreach $thing (keys %data) {
45:     print RESULTS "$thing $data{$thing}\n";
46: }
47: close(RESULTS);
48:
49: print header;
50: print start_html('Danke');
51: print <<EOF;
52: <H1>Danke, dass Sie das Umfrageformular ausgefüllt haben!</H1>
53: <P>Die bisherigen Ergebnisse:
54: <P>Geschlecht:
55: <TABLE BORDER><TR><TH>Männlich</TH><TD>
56: EOF
57:
58: print &percent('sex_m'), "</TD></TR>\n";
59: print "<TR><TH>Weiblich</TH><TD>\n";
60: print &percent('sex_f'), "</TD></TR>\n";
61: print "<TR><TH>Keine Antwort</TH><TD>\n";
62: print &percent('sex_na'), "</TD></TR>\n";
63: print "</TABLE>\n";
64:
65: print "<P>Alter:\n";
66: print "<TABLE BORDER><TR><TH>Unter 18</TH><TD>\n";
67: print &percent('age_under18'), "</TD></TR>\n";
68: print "<TR><TH>18 bis 34</TH><TD>\n";
69: print &percent('age_18to34'), "</TD></TR>\n";
70: print "<TR><TH>35 bis 50</TH><TD>\n";
71: print &percent('age_35to50'), "</TD></TR>\n";
72: print "<TR><TH>Über 50</TH><TD>\n";
73: print &percent('age_50plus'), "</TD></TR>\n";
74: print "<TR><TH>Keine Antwort</TH><TD>\n";
75: print &percent('age_na'), "</TD></TR>\n";
76: print "</TABLE>\n";
77:
78: print "<P>Perl-Programmierer?\n";
79: print "<TABLE BORDER><TR><TH>Ja</TH><TD>\n";
80: print &percent('perl_y'), "</TD></TR>\n";
81: print "<TR><TH>Nein</TH><TD>\n";
82: print &percent('perl_n'), "</TD></TR>\n";
83: print "<TR><TH>Keine Antwort</TH><TD>\n";
84: print &percent('perl_na'), "</TD></TR>\n";
85: print "</TABLE>\n";
86:
87: print end_html;
88:
89: sub percent {
90:     if (defined $data{$_[0]}) {
91:         return sprintf("%.1f%%", $data{$_[0]} / $data{total} * 100);
92:     }
93:     else { return '0%'; }
94: }
```

Ich möchte dieses Skript nicht Zeile für Zeile durchgehen, da Sie einen großen Teil davon schon in irgendeiner Form gesehen haben (und viele Zeilen nur aus `print`- Anweisungen bestehen). Statt dessen möchte ich Sie auf einige der wichtigeren Teile dieses Skripts aufmerksam machen:

Zeile 5 speichert den Namen der Datei mit den Umfragedaten in der Variablen `$results`. Dabei wird davon ausgegangen, dass sich die Datei in dem gleichen Verzeichnis befindet wie das CGI-Skript. Gegebenenfalls müssen Sie den Pfadnamen in Ihrem Skript ändern. Die Datendatei besteht aus einem Satz von Datenschlüsseln für die einzelnen Optionen in der Umfrage und »na«-Schlüsseln für den Fall, dass in einer Gruppe keine Option ausgewählt wurde. Jeder Schlüssel hat einen Wert für die Anzahl der »Stimmen«, die eingereicht wurden. In den Zeilen 10 bis 15 wird die Datendatei geöffnet und die Daten in den Hash `%data` eingelesen. Anschließend wird die Datei direkt wieder geschlossen.

Die Datendatei - die hier im gleichen Verzeichnis steht wie das CGI-Skript - muss dem Webserver den Schreibzugriff erlauben. Für Unix-Systeme bedeutet dies, dass die Datei mit den entsprechenden Zugriffsberechtigungen versehen sein muss, so dass der Webserver mit seiner Benutzer- oder Gruppen-ID (in der Regel `nobody`*) darauf zugreifen kann. Aber auch unter Windows müssen Sie Ihre Sicherheitseinstellungen entsprechend setzen. Wenn Sie das Skript auf einem Webserver ausführen und dabei Fehler erhalten, die besagen, dass in die Datei nicht geschrieben werden konnte (und die bei der Ausführung von der Befehlszeile aus nicht aufgetreten sind), sollten Sie die Zugriffsberechtigungen für die Datei prüfen.*

Die Zeilen 17 bis 41 verarbeiten die Eingabe des Formulars in Gruppen, die mit den Gruppen der Optionsschalter in der HTML-Datei (Alter, Geschlecht und Perl) übereinstimmen. Beachten Sie, dass Sie für jede Gruppe testen müssen, ob eventuell keine Antwort gegeben wurde (in diesem Falle wird nämlich für diese Gruppe kein Schlüssel in dem `param()`-Array, das Sie von `CGI.pm` erhalten, eingerichtet.) Wurde eine Auswahl getroffen, inkrementieren wir den Wert dieses Schlüssels in dem Daten- Hash. Dabei verfolgen wir die Gesamtsumme (Zeile 17), die wir benötigen, um die Prozentwerte für die Ausgabe zu berechnen.

Nachdem wir die Daten verarbeitet haben, können wir die neue Datendatei in die gleiche Datei schreiben, je einen Datenschlüssel und -wert pro Zeile, getrennt durch ein Leerzeichen. Die eigentliche Reihenfolge der Schlüssel ist nicht von Belang, da sie vom Skript nur wieder aus der Datei in einen Hash gelesen werden. Eine einfache `foreach`-Schleife in den Zeilen 44 und 45 schreibt die neuen Werte in die Datendatei (die wir in Zeile 43 zum Schreiben neu geöffnet haben).

Die zweite Hälfte des Skripts erzeugt als Ausgabe das aktuelle Ergebnis der Umfrage, die der Benutzer als Antwort erhält. Ab Zeile 49 erfolgt die Ausgabe, die mit dem Header beginnt (Zeile 49). Anschließend werden der Titel (Zeile 50) und mit Hilfe eines Hier-Dokuments (Zeile 51 bis 56) die ersten Zeilen HTML-Text ausgegeben.

Ab Zeile 58 wird das Ganze dann erst richtig interessant. Die restliche HTML-Datei besteht aus Tabellen, in denen die aktuellen Ergebnisse der Umfrage präsentiert werden. Ein Großteil der folgenden `print`-Ausgabe wird für den HTML-Code für die Tabellen benötigt. Die endgültigen Prozentzahlen werden mit einer von uns definierten Hilfsroutine namens `&percent()` berechnet. Die in den Zeilen 89 bis 94 definierte `&percent()`-Routine erzeugt einen Prozent-String, der sich auf der Basis des ihr übergebenen Werts geteilt durch die Gesamtzahl der Antworten errechnet. Diese Routine stellt auch sicher, dass der gegebene Datenschlüssel tatsächlich einen Wert hat (ist er gleich Null, erscheint er nicht im Hash `%data`). Und zum Schluß formatiert sie mit Hilfe der Funktion `sprintf` die Prozentangaben mit einem Dezimalpunkt und einem Prozentzeichen (beachten Sie, dass Sie aufgrund des `sprintf`-Formatiercodes nicht ohne weiteres ein einfaches Prozentzeichen ausgeben können; Sie müssen es als `%%` eingeben).

Vertiefung

Wie ich bereits zu Beginn dieser Lektion erwähnt habe, könnte ich problemlos Seite um Seite und Kapitel um Kapitel mit all den verschiedenen Aspekten von CGI füllen. Eine genaue Beschreibung von `CGI.pm` allein wäre schon wesentlich umfangreicher als mein bescheidenes Kapitel hier. Dennoch gibt es einige Besonderheiten von `CGI.pm`, auf die ich noch näher eingehen möchte. Alle diese Besonderheiten finden Sie natürlich in der Dokumentation zu `CGI.pm` (`perldoc CGI` weist Ihnen den Weg) und auf der Webseite `http://www.genome.wi.mit.edu/ftp/pub/software/WWW/ cgi_docs.html`.

Wenn Sie an weitergehenden Informationen zu CGI selbst interessiert sind, scheuen Sie sich nicht, die Webseiten zu besuchen, die ich eingangs dieser Lektion erwähnt habe, oder lesen Sie das Buch, das ich Ihnen dort empfohlen habe. Wenn Sie Unterstützung brauchen, um Ihre CGI-Skripts auf verschiedenen Plattformen zum Laufen zu bringen, schauen Sie doch mal in der Usenet-Newsgroup `comp.infosystems.www.authoring.cgi` nach.

CGI-Variablen verwenden

Wenn Ihr CGI-Skript mit den Daten vom Browser aufgerufen wird, hält der Webserver in seiner Umgebung einige interessante Werte bereit, die sich auf das Skript selbst, auf den Webserver und auf das System, auf dem der Browser läuft, der das Formular abgeschickt hat, beziehen. Auf Unix-Systemen werden diese Werte auch Umgebungsvariablen genannt, auf die man innerhalb des Perl-Skripts mit dem Hash %ENV zugreifen kann. Auf anderen Webservern erfolgt die Übergabe dieser Variablen unter Umständen anders. In `CGI.pm` jedoch finden Sie Subroutinen, um auf diese Variablen in einer Art und Weise zuzugreifen, die plattform- und Webserver- unabhängig

ist. Sie müssen diese Subroutinen nicht unbedingt in Ihren CGI-Skripten verwenden, aber vielleicht finden Sie die Daten ganz nützlich.

Tabelle 16.1 enthält die `CGI.pm`-Subroutinen zu den CGI-Variablen.

Subroutine	Was Sie dadurch erhalten
`accept()`	Eine Liste der MIME-Typen, die der Browser akzeptiert
`auth_type()`	Den Authentifizierungstyp (normalerweise »`basic`«)
`path_info()`	Pfadinformationen kodiert im Skript-URL (falls verwendet)
`path_translated()`	Das gleiche wie `path_info()`, erweitert zu einem vollen Pfadnamen
`query_string()`	Argumente für das CGI-Skript, die an den URL angehängt wurden
`raw_cookie()`	Liefert reine Cookie-Informationen. Verwalten Sie diese Informationen mit den `cookie()`-Subroutinen
`referer()`	Den URL der Seite, die dieses Skript aufgerufen hat
`remote_addr()`	Die IP-Adresse des Hosts, der dieses Skript aufgerufen hat (der Host des Browsers)
`remote_ident()`	Die User-ID, aber nur wenn das System `ident` ausführt (nicht üblich)
`remote_host()`	Der Name des Hosts, der dieses Skript aufgerufen hat
`remote_user()`	Der Name des Benutzers, der dieses Skript aufgerufen hat (normalerweise nur gesetzt, wenn der Benutzer sich mit Authentifizierung eingeloggt hat)
`request_method()`	Die Webserver-Methode, mit der das Skript aufgerufen wurde (zum Beispiel `GET` oder `POST`)
`script_name()`	Der Name des Skripts
`server_name()`	Der Hostname des Webservers, der dieses Skript aufgerufen hat
`server_software()`	Der Name und die Version der Webserver-Software
`virtual_host()`	Für Server, die virtuelle Hosts unterstützen, der Name des virtuellen Hosts, der dieses Skript ausführt
`server_port()`	Der Netzwerk-Port, den der Server verwendet (normalerweise 80)
`user_agent()`	Name und Version des Browsers, der dieses Skript aufgerufen hat, zum Beispiel `Mozilla/4.x (Win95)`
`user_name()`	Der Name des Benutzers, der dieses Skript aufgerufen hat (fast nie gesetzt)

Tabelle 16.1: ***Die Subroutinen zu den CGI-Variablen***

POST und GET

Ein Browser kann CGI-Skripts auf zwei Wegen aufrufen: via `POST` oder via `GET`. `GET` kodiert die Formularelemente direkt in den URL, während `POST` die Formularelemente über die Standardeingabe sendet. Darüber hinaus kann `GET` dazu verwendet werden, um ein Formular abzuschicken, das eigentlich nicht vorhanden ist - Sie könnten zum Beispiel einen Link haben, bei dem der URL die hartkodierten Formularelemente zum Abschicken enthält.

`CGI.pm` verarbeitet beide Wege und speichert sie in dem Parameter-Array Ihres CGI- Skripts, so dass Sie sich keine Gedanken darüber machen müssen, welche Methode zum Abschicken des Skripts verwendet wurde. Wenn Sie wirklich daran interessiert sind, die Parameter aus dem URL herauszulesen, verwenden Sie statt `param()` die Funktion `url_param()`.

Redirektion

Manchmal ist das Ergebnis eines CGI-Skripts keine HTML-Datei, sondern statt dessen ein Verweis auf eine

existierende HTML-Datei auf dem eigenen oder einem beliebigen Server. `CGI.pm` unterstützt diese Art von Ergebnis durch die Subroutine `redirect()`:

```
print redirect('http://www.andererserver.com/anderedatei.html');
```

Mit `redirect` teilen Sie dem Browser des Benutzers mit, dass er nicht irgendeinen HTML-Code auf dem Bildschirm anzeigen, sondern eine bestimmte Datei im Web suchen soll. Da ein CGI-Skript, das `redirect` aufruft, keine neue Webseite erzeugt, sollten Sie in der Ausgabe eines CGI-Skripts `redirect` nicht mit irgendeinem HTML- Code kombinieren.

Cookies und das Hochladen von Dateien

Mit `CGI.pm` können Sie sogar Cookie-Werte verwalten und Dateien bearbeiten, die über den Upload-Mechanismus der HTML-Formulare hochgeladen werden. Zur Verwaltung von Cookies dient die Subroutine `cookie()` aus `CGI.pm`. Das Hochladen von Dateien verläuft ähnlich wie die Bearbeitung normaler Formularelemente. Die Subroutine `param()` wird verwendet, um den Dateinamen zurückzuliefern, der in dem Formular eingegeben wurde. Dieser Dateiname ist gleichzeitig auch ein Datei-Handle, der offen ist und aus dem Sie Zeilen mit den Standard-Perl-Mechanismen auslesen können.

Zu beiden Themen finden Sie weitere Informationen in der `CGI.pm`-Dokumentation.

CGI-Skripts und Sicherheit

Jedes CGI-Skript stellt ein potentielles Sicherheitsloch auf Ihrem Webserver dar. Jede beliebige Person, die durch das Web streift, kann CGI-Skripts auf Ihrem Server mit irgendwelchen Eingaben ausführen lassen. Je nachdem wie sorglos Sie Ihre Skripts schreiben und wie entschlossen die Gegenpartei ist, kann Ihr CGI-Skript für böswillige Nutzer Schlupflöcher aufweisen, die diese nutzen können, um in Ihr System einzubrechen und schlimmstenfalls unwiderruflichen Schaden anzurichten.

Der beste Weg, ein Problem zu beseitigen, besteht darin, es zu erkennen und Schritte zu seiner Vermeidung zu unternehmen. Ein empfehlenswerter Ansatzpunkt sind die häufig gestellten Fragen zur Sicherheit im WWW unter `http://www.genome.wi.mit.edu/ WWW/faqs/www-security-faq.html`. Perl verfügt außerdem über eine besondere Option, den sogenannten ***Taint-Modus***, der Sie daran hindert, unsichere Daten so zu verwenden, dass Ihr System Schaden erleiden könnte. Im Kapitel 20, »Was noch bleibt«, erfahren Sie mehr über den Taint-Modus.

Perl in den Webserver einbetten

Jedesmal, wenn ein in Perl geschriebenes CGI-Skript von einem Webserver aufgerufen wird, erfolgt ein Aufruf an Perl, um das Skript auszuführen. Für sehr stark frequentierte Webserver, auf denen eine Unmenge von CGI-Skripten ausgeführt wird, kann das bedeuten, dass viele Kopien von Perl gleichzeitig aufgerufen werden, was eine beträchtliche Last für die Maschine darstellt, die als Webserver fungiert. Um ihre Leistungsfähigkeit zu verbessern, verfügen viele Webserver über einen Mechanismus, mit dem der Perl-Interpreter in dem Webserver selbst eingebettet wird, so dass Skripts, die auf dem Webserver aufgerufen werden, nicht länger als CGI-Skripts ausgeführt werden. Statt dessen werden sie als Teile der Webserver-Bibliothek ausgeführt, wodurch der Overhead und die Anlaufzeit für die Skripts reduziert werden. Oftmals muss man die CGI-Skripts nicht einmal anpassen, um sie auf diese Weise ausführen zu lassen.

Unterschiedliche Webserver auf unterschiedlichen Plattformen verwenden unterschiedliche Mechanismen zur Einbettung von Perl. Sie müssen in der Dokumentation zu Ihrem Webserver nachlesen, ob mit Perl verfaßte CGI-Skripts eingebettet werden können und welche Tools und Module man dafür benötigt.

Wenn Sie einen ISAPI-Webserver unter Windows (beispielsweise den IIS) verwenden, benötigen Sie das »Perl für ISAPI«-Paket (manchmal auch PerlIIS genannt). Dieses Paket ist Teil der ActiveState-Version von Perl für Windows und wird automatisch zusammen mit diesem Paket installiert. Sie können es aber auch separat von der ActiveState-Web-Site unter `http://www.activestate.com` herunterladen.

Wenn Sie einen Open-Source Apache-Webserver verwenden, finden Sie in `mod_perl` ein Apache-Modul, das den Perl-Interpreter in den Apache-Webserver einbettet. Damit erreichen Sie, dass Ihre in Perl geschriebenen CGI-Skripts schneller und effizienter ausgeführt werden. Doch obwohl dies sicherlich das wichtigste Ergebnis ist, ist es

nicht das einzige. Darüber hinaus gestattet Ihnen die Einbettung fortan, mit Perl auf alle internationalen Erweiterungs-APIs von Apache zuzugreifen, so dass Sie Ihren Webserver mit Hilfe von Perl in nahezu unbegrenzter Weise anpassen können. Weitere Informationen zum Apache-Webserver finden Sie unter `http://www.apache.org` und beim Apache/Perl-Integrationsprojekt, den Entwicklern von `mod_perl`, und zwar unter `http://perl.apache.org`.

Zusammenfassung

Nicht alles in Perl ist gleichbedeutend mit langen, komplizierten Skripten und vielen, vielen Subroutinen. Manchmal sind es gerade die nützlichsten und interessantesten Dinge, die man mit Hilfe von Modulen und ein wenig eigenem Code, der das Skript zusammenhält, implementieren kann. CGI ist dafür ein hervorragendes Beispiel. Das Modul `CGI.pm` übernimmt den Großteil der harten Arbeit bei der CGI-Programmierung und macht es leicht, die Werte aus einem Formular oder einer anderen CGI-Quelle auszulesen und eine HTML-Datei als Ergebnis zurückzusenden.

Heute haben Sie ein wenig darüber gelernt, wie Perl für CGI eingesetzt werden kann. Sie wissen jetzt, wie CGI abläuft - vom Browser zum Server zum Skript und wieder zurück -, wie `CGI.pm` in Ihr Skript importiert und dort genutzt werden kann und wie die zahlreichen Möglichkeiten dieses Moduls Ihr Leben mit CGI angenehmer machen. Wir haben ein umfangreicheres CGI-Beispiel gesehen: ein Umfrageformular, das die Umfragedaten in einer externen Datei abspeichert. Ich kann zwar nicht garantieren, dass diese Lektion Ihnen genügend Wissen vermittelt hat, um CGI-Skripts für beliebige Aufgaben zu erstellen (weil ich dazu zuviel in dieser Lektion auslassen mußte), aber ich hoffe, dass sich die Lektion als eine gute Ausgangsbasis erweist, auf der Sie aufbauen können.

Die in dieser Lektion vorgestellten Subroutinen sind alle Teil der `CGI.pm`-Moduls und umfassen:

- `param()` - liest die Parameter, die dem CGI-Skript, normalerweise als Teil des übersendeten Formulars, übergeben wurden. `param()` ohne Argumente liefert eine Liste der verfügbaren Schlüssel (Namen der Formularelemente) zurück. `param()` mit einem Schlüsselargument liefert den Wert für diesen Schlüssel zurück.
- `print_header()` - gibt den CGI-Header für die Ausgabe aus. `print_header()` ohne Argumente geht davon aus, dass die Ausgabe im HTML-Format erfolgt.
- `start_html()` - erzeugt den obersten Teil einer HTML-Seite einschließlich der Tags `<HTML>`, `<HEAD>`, `<TITLE>` und `<BODY>`. Unterschiedliche Argumente für `start_html()` erzeugen unterschiedliche Werte für den HTML-Code (so setzt zum Beispiel ein einzelnes String-Argument den Titel der Seite).
- `end_html()` - erzeugt die schließenden Tags (`</BODY>` und `</HTML>`) für die Ausgabe.

Fragen und Antworten

Frage:
Ich habe eine Reihe von CGI-Skripten gesehen, die in Perl geschrieben waren und trotzdem keinen Gebrauch von `CGI.pm` gemacht haben. Statt dessen verwenden diese Skripts andere Bibliotheken wie z.B. `cgi-lib.pl`. Sind diese Skripts schlecht und sollten korrigiert werden?

Antwort:
Nicht unbedingt. Es gibt im Web etliche Bibliotheken zur Erstellung von CGI- Skripten mit Perl, und `cgi-lib.pl` ist eine der bekanntesten. Sie können bedenkenlos diese anderen Bibliotheken verwenden. Ich habe mich in diesem Kapitel für `CGI.pm` entschieden, da es inzwischen der Standard für Perl im Zusammenspiel mit CGI geworden ist. Es ist ein unproblematisches Modul, das den Standard-Modul-Konventionen folgt, bei Bedarf objektorientiert genutzt werden kann und, was am wichtigsten ist, Teil der Standard-Perl- Version ist. Damit hat es einen unschlagbaren Vorteil gegenüber anderen Bibliotheken.

Frage:
Meine CGI-Skripts lassen sich von der Befehlszeile aus ausführen, nicht jedoch von einer Webseite. Was mache ich falsch?

Antwort:
In Anbetracht der vielen verschiedenen Plattformen und Webserver, die es gibt, läßt sich diese Frage nicht so einfach beantworten. Ist das Skript eine ausführbare Datei? Ist das Skript in einem »offizielles« CGI-Verzeichnis Ihres Servers abgelegt (normalerweise ein spezielles `cgi-bin`-Verzeichnis oder etwas Ähnliches)? Sind die

Zugriffsberechtigungen für Ihr Skript korrekt gesetzt (Webserver führen CGI-Skripts als einen besonderen Benutzer mit begrenzten Zugriffsrechten aus)? Sind Sie sicher, dass auf Ihrer Maschine überhaupt ein Webserver installiert ist? Auf den Web-Sites, die ich Ihnen im Laufe dieser Lektion genannt habe, finden Sie Tipps und Vorschläge zum Debuggen Ihrer CGI-Skripts.

Frage:
Meine Website befindet sich auf einer Unix-Maschine. Dort kann ich CGI-Skripts ausführen. Aber ich möchte meine CGI-Skripts lieber auf meinem Windows-NT- PC zuhause schreiben und debuggen, so dass ich nicht die ganze Zeit eingeloggt sein muss. Ist dies möglich?

Antwort:
Dank des Wunders der Plattformunabhängigkeit von Perl und des Moduls `CGI.pm` ist das selbstverständlich möglich. Sie müssen dazu einen Webserver auf Ihrem NT-System installieren - am besten einer der ähnlich oder gleich dem auf der Unix-Maschine ist. Danach können Sie lokal CGI-Skripts installieren, debuggen und ausführen. Sie sollten allerdings auf Unterschiede in den Pfadangaben der Dateien achten und auf die Art, in der verschiedene Server mit CGI umgehen. Halten Sie Ihr Skript so einfach wie möglich und Sie sollten wenig Arbeit damit haben, es auf Unix zu übertragen.

Frage:
Ich habe die Umfragedatei modifiziert, so dass sie jetzt mit einem Gästebuch- ähnlichen Skript verwendet wird und es den Besuchern erlaubt, Kommentare zu einer Webseite abzugeben. Allerdings habe ich Probleme damit, wenn mehrere Besucher zur selben Zeit ihren Kommentar an die Webseite schicken. Manchmal wird von mehreren Besuchern gleichzeitig in die Datei geschrieben, und Kommentare gehen dadurch verloren, oder seltsame Dinge passieren. Was mache ich falsch?

Antwort:
Das Umfragebeispiel in diesem Kapitel zeigt nur die Grundlagen der CGI- Skripterstellung. Um kommerzielle Web-Sites zu unterstützen, müssen Sie schon etwas tiefer einsteigen (ich empfehle, die von mir bereits genannten Websites zu konsultieren und ein weiterführendes Buch anzuschaffen). Besonders, wenn Sie mit einer externen Datei arbeiten, die potentiell von mehreren Benutzern gleichzeitig beschrieben werden kann - wie das mit jedem CGI-Skript möglich ist -, werden Sie diese Datei »sperren« müssen, bevor in die Datei geschrieben wird, und anschließend die Sperre wieder aufheben. Das könnte so einfach sein, wie eine weitere temporäre Datei namens `umfrage.lock` anzulegen. In Ihrem Skript würden Sie also, bevor Sie in die Datei mit den Umfrageergebnissen schreiben, Folgendes tun:

- Prüfen, ob die Sperrdatei existiert. Wenn ja, schreibt gerade jemand anders in die Datei. Ein wenig warten (bedienen Sie sich dazu der `sleep`- Funktion) und erneut versuchen.
- Wenn die Datei frei ist, die Sperrdatei erzeugen
- Die Daten schreiben
- Die Datei wieder freigeben

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Wofür steht CGI? Wozu wird es verwendet?
2. Welche Unterstützung bietet Ihnen das Modul `CGI.pm`? Wo finden Sie es, und wie nutzen Sie es?
3. Warum sollten Sie ein CGI-Skript erst über die Befehlszeile ausführen, bevor Sie es über eine Webseite ausführen lassen?
4. Wie setzen Sie die Subroutine `param()` ein?
5. Nennen Sie drei Möglichkeiten, HTML-Code als Ausgabe Ihres CGI-Skripts zu erzeugen.
6. Warum ist ein Hier-Dokument manchmal sinnvoller als `print`-Anweisungen?

Übungen

1. Schreiben Sie ein CGI-Skript, das die Formularelemente, die ihm übergeben wurden, als Liste mit Aufzählungszeichen ausgibt. HINWEIS: Listen mit Aufzählungszeichen haben in HTML folgendes Format:

```
    <UL>
<LI>Ein Element
<LI>Zwei Elemente
<LI>Drei Elemente
</UL>
```

1. Ein weiterer Hinweis: `param()` ohne Argumente gibt Ihnen eine Liste der Namen aller Formularelemente aus, die an das Skript geschickt wurden.
2. FEHLERSUCHE: Was ist falsch an folgendem Skriptfragment:

```
    if (!param()) { $data{sex_na}++ }
else {
    if (param() eq 'male') { $data{sex_m}++; }
    elsif (param() eq 'female') { $data{sex_f}++; }
}
```

3. Schreiben Sie ein CGI-Skript, das eine einfache HTML-Seite ausgibt (»Hallo Welt« reicht), gleichzeitig registriert, wie oft auf die Seite zugegriffen wurde, und diesen Wert oben auf der gleichen Seite anzeigt.
4. Schreiben Sie ein CGI-Skript, um ein sehr einfaches »Gästebuch« zu implementieren, das es den Benutzern erlaubt, einzeilige Kommentare an eine Webseite zu schicken. Registrieren Sie alle vergangenen Kommentare. Gehen Sie dabei von der Existenz eines HTML-Formulars mit zwei Elementen aus: einem Textfeld namens **Mail** für die E-Mail-Adresse des Absenders und einem Textfeld namens **Kommentar** für die Kommentare.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. CGI steht für Common Gateway Interface (Allgemeine Zugangsschnittstelle) und bezieht sich auf Programme, die auf einem Webserver als Antwort auf Anfragen von einem Webbrowser (meist von einem Formular auf einer HTML-Seite) ausgeführt werden.
2. Das Modul `CGI.pm` hilft beim Schreiben, Ausführen und Debuggen von CGI- Skripten. Es definiert eine Reihe von Subroutinen, mit denen die Ein- und Ausgaben der CGI-Skripts bearbeitet werden können, sowie etliche Hilfsroutinen zu fast jedem erdenklichen Aspekt der CGI-Skripterstellung.
1. Das Modul `CGI.pm` ist Teil der meisten aktuellen Perl-Versionen. Wenn Sie nicht schon bereits darüber verfügen, können Sie es von dem weiter vorn in dieser Lektion genannten URL herunterladen und auf Ihrem Computer installieren.
1. Um das `CGI.pm`-Modul zu nutzen, müssen Sie es, wie jedes andere Modul auch, erst einmal importieren. Dazu verwenden Sie `use` und ein Import-Tag wie `:standard` oder `:all`.
3. Ein CGI-Skript von der Befehlszeile auszuführen, macht es in der Regel leichter, Syntax- oder sonstige Fehler zu finden (so brauchen Sie das Skript nicht erst auf dem Webserver zu installieren, Sie müssen keine Verbindung zum Internet herstellen und so weiter). Indem Sie dem Skript bei Ausführung von der Befehlszeile Musterdaten übergeben, können Sie es testen, bevor Sie es online verfügbar machen.
4. Die Subroutine `param()` dient dazu, die Werte aller Formularelemente auf der Webseite, die das Skript aufgerufen hat, zu ermitteln. Ohne Argumente aufzurufen, liefert `param()` eine Liste der Schlüssel zurück, die sich auf die Namen der Formularelemente beziehen. Mit einem Schlüsselargument (einem String) aufgerufen, liefert `param()` den Wert zurück, der mit diesem Formularelement verbunden ist.
5. HTML-Code können Sie auf eine der folgenden Arten ausgeben:

- Mit regulären `print`-Anweisungen; beachten Sie eingebettete Anführungszeichen!
- Mit Hier-Dokumenten
- Mit verschiedenen Subroutinen aus dem `CGI.pm`-Modul

6. Hier-Dokumente stellen eine Möglichkeit dar, einen Block von Text wörtlich auszugeben. Alle Neue-Zeile-Zeichen und Anführungszeichen werden unverändert ausgegeben. Hier-Dokumente helfen Ihnen, eine Menge von sich wiederholenden `print`-Anweisungen sowie Anführungszeichen mit Escape-Zeichen zu

vermeiden.

Antworten zu den Übungen

1. Hier ein Lösungsvorschlag:

```
    #!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
my @keys = param();
print header;
print start_html('Hallo!');
print "<H1>Schlüssel/Wert-Paare</H1>\n";
print "<UL>\n";
foreach my $name (@keys) {
    print "<LI>$name = ", param($name), "\n";
}
print "</UL>\n";
print end_html;
```

2. Die Subroutine `param()` wird ohne irgendwelche Argumente aufgerufen. In einigen Fällen - wie zum Beispiel in Übung 1 - entspricht dies genau Ihren Wünschen. In diesem Fall jedoch, in dem die Tests einen String-Wert erwarten, sollte `param()` besser mit einer Art von String-Argument aufgerufen werden, um einen Wert aus den Formularparametern zu erhalten.
3. Hier ein Lösungsvorschlag (vergessen Sie nicht vorher ***zaehlert.txt*** zu erzeugen):

```
    #!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
my $countfile = 'zaehler.txt';
my $count = '';
open(COUNT, $countfile) or
                    die "Zählerdatei konnte nicht geöffnet werden: $!";
while (<COUNT>) {
    $count = $_;
}
close(COUNT);
$count++;
open(COUNT, ">$countfile") or
                  die "In Zählerdatei kann nicht geschrieben werden: $!";
print COUNT $count;
close(COUNT);
print header;
print start_html('Hallo!');
print "<H1>Hallo Welt</H1>\n";
print "<P>Diese Seite wurde bereits $count Mal besucht.\n";
print end_html;
```

4. Hier ein (besonders einfacher) Lösungsvorschlag. Dieses Skript geht von einer Datei mit bereits abgegebenen Kommentaren aus, pro Zeile einer, mit der E-Mail- Adresse vorneweg:

```
    #!/usr/bin/perl -w
use strict;
use CGI qw(:standard);
my $guestbook = 'gaeste.txt';
my $mail = '';                  # EMail-Adresse
my $comment = '';               # Kommentare
open(GUEST, $guestbook) or
                   die "Gästebuch konnte nicht geöffnet werden: $!";
print header;
print start_html('Kommentare');
print "<H1>Kommentare</H1>\n";
print "<P>Kommentare zu dieser Website!\n";
print "<HR>\n";
while (<GUEST>) {
    ($mail, $comment) = split(' ',$_,2);
    if ($mail) {
        if (!$comment) { $comment = "nichts\n"; }
        else {  print "<P><B>$mail sagt:</B> $comment"; }
```

```
        }
    }
    $mail = param('mail');
    $comment = param('comment');
    print "<P><B>$mail sagt:</B> $comment\n";
    open(GUEST, ">>$guestbook") or
                        die "Gästebuch konnte nicht geöffnet werden: $!";
    print GUEST "$mail $comment\n";
    print end_html;
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Dateien und Verzeichnisse verwalten

In Kapitel 15, »Dateien und E/A«, haben wir uns damit beschäftigt, wie man in Perl aus Dateien liest oder in Dateien schreibt. Heute wollen wir uns weiteren Aspekten der Dateiverwaltung zuwenden - insbesondere den verschiedenen Möglichkeiten, mit denen Sie über Ihr Skript das Dateisystem beeinflussen können. Heute lernen Sie,

- wie man Dateien verwaltet, das heißt, wie man sie entfernt, umbenennt, kopiert oder verknüpft und wie man Zugriffsberechtigungen ändert,
- wie man mit Verzeichnissen arbeitet, das heißt, wie man durch das Verzeichnissystem navigiert, wie man Verzeichnisse erzeugt und wieder entfernt und
- wie man auf Gruppen von Dateien zugreift und wie man Verzeichnis-Handles verwendet,

Dateien verwalten

Mit Perl können Sie nicht nur aus Dateien lesen und in Dateien schreiben, sondern Sie können die Dateien auch verwalten - so wie Sie es üblicherweise von der Befehlszeile oder einem Datei-Manager aus tun: Sie können Dateien umbenennen, löschen, Zugriffsrechte ändern oder Verknüpfungen zu Dateien erstellen. Für jede dieser Aufgaben greifen Sie auf verschiedene vordefinierte Perl-Funktionen zurück, die ich Ihnen in diesem Abschnitt vorstellen möchte.

Dateien umbenennen

Um einen Dateinamen durch einen anderen zu ersetzen, ohne dabei den Inhalt der Datei zu beschädigen, rufen Sie die Funktion `rename` mit zwei Argumenten auf: dem alten Namen der Datei und dem neuen Namen:

```
rename meinedatei, meinedatei.bak;
```

Existiert bereits eine Datei mit dem gleichen Namen wie die neue Datei, wird diese von Perl überschrieben. Sind Sie an Unix gewöhnt, stellt dies kein Problem für Sie dar. Arbeiten Sie jedoch auf einem Windows- oder Mac-Rechner, gilt es vorsichtig zu sein! Es gibt keine Sicherheitsabfragen, wenn Sie eine bestehende Datei überschreiben. Deshalb sollten Sie vielleicht besser sicherstellen, dass es für den neuen Namen nicht bereits eine gleichnamige Datei gibt (mit einem der Dateitests wie `-e`), bevor Sie Ihre Datei umbenennen.

Die `rename`-Funktion liefert `1` oder `0` zurück (***wahr*** oder ***falsch***), je nachdem ob Sie Ihre Datei erfolgreich umbenennen konnten oder nicht. Um eine Datei umbenennen zu können, müssen Ihnen (oder dem Benutzer Ihres Skripts) die entsprechenden Zugriffsrechte eingeräumt sein.

Verknüpfungen erstellen und Verknüpfungen nachfolgen

In der Unix-Version von Perl gibt es zwei Funktionen zum Erstellen von Verknüpfungen zwischen Dateien: `link` und `symlink`, mit denen harte beziehungsweise symbolische Verknüpfungen (entsprechend dem Unix-Befehl `ln`) erstellt werden. Harte Verknüpfungen dienen dazu, eine Datei mit mehr als einem Namen zu erzeugen. Wenn Sie einen beliebigen dieser Namen (einschließlich der Originaldatei) löschen, existiert die Datei weiter - solange wie es noch Namen für die Datei gibt. Symbolische Verknüpfungen sind Verweise auf andere Dateien. Wird die Verknüpfung gelöscht, bleibt die Datei bestehen, löschen Sie hingegen die Originaldatei, zeigt die symbolische Verknüpfung ins Leere. Nicht alle Unix-Systeme unterstützen symbolische Verknüpfungen.

Beide Funktionen, `link` und `symlink`, erfordern zwei Argumente: den Name der Originaldatei, für die eine Verknüpfung erstellt werden soll, und den Namen der neuen Verknüpfung. Sehen Sie dazu folgendes Beispiel:

```
link datei1, datei2;
```

Beide Funktionen liefern im Erfolgsfall `1` (***wahr***) und ansonsten `0` (***falsch***) zurück.

Weder hart noch symbolisch verknüpfte Dateien stellen für Perl ein Problem dar. Wenn Sie `open` mit einem Dateinamen verwenden, der eine Verknüpfung darstellt, wird statt der Verknüpfung die Originaldatei geöffnet. Mit Hilfe des Dateitests `-l` können Sie prüfen, ob eine bestimmte Datei eine symbolische Verknüpfung ist oder nicht. Anschließend können Sie die Verknüpfung mit Hilfe der Funktion `readlink` zu der ursprünglichen Position der Datei zurückverfolgen:

```
if (-l $dat) {                        # dat ist eine Verknüpfung
   $ursprungsdatei = readlink $dat;   # ermittelt die echte Datei
}
```

Beachten Sie, dass Sie mit `readlink` nur den relativen Pfadnamen der Originaldatei zu ihrer symbolischen Verknüpfung erhalten. Das bedeutet, dass Sie entweder diesen Pfad zu einer vollen Pfadangabe erweitern oder in das Verzeichnis mit der symbolischen Verknüpfung wechseln müssen, um den Pfadnamen der Originaldatei zu verwenden. Wie Sie Verzeichnisse wechseln, erfahren Sie weiter hinten in diesem Kapitel.

Verwenden Sie Perl für Windows? Dann können Sie weder mit harten noch mit symbolischen Verknüpfungen arbeiten. Sie können aber das Modul `Win32::Shortcut` verwenden, um Windows-Explorer-Verknüpfungen zu erzeugen.

In MacPerl gibt es zwar keine Entsprechung zu den harten Verknüpfungen, doch funktionieren die symbolischen Verknüpfungen ganz analog zu den Mac-Aliasen. Die Funktion `symlink` erzeugt einen Alias, der Dateitest `-l` liefert ***wahr*** zurück, wenn die Datei ein Alias ist und `readlink` ermittelt die Position der Datei, mit der der Alias verknüpft ist.

Dateien und Verknüpfungen entfernen

Sie sind mit einer Datei fertig und sind sich sicher, dass Sie sie nicht weiter benötigen. Wie werden Sie diese Datei jetzt wieder los? Dazu steht Ihnen die Funktion `unlink` zur Verfügung (trotz ihres Namens entfernt sie sowohl Verknüpfungen als auch Dateien). Sie entspricht in ihrer Funktionsweise dem Unix-Befehl `rm` oder dem DOS-Befehl `del`. Allerdings verschiebt diese Funktion die Datei nicht in einen Papierkorb wie unter Windows oder beim Mac. Wird eine Datei gelöscht, wird sie unwiederbringlich entfernt. Deshalb sollten Sie Vorsicht walten lassen, um nicht aus Versehen Dateien zu löschen, die Sie eigentlich noch benötigen.

Die Funktion `unlink` übernimmt ein Listenargument, bei dem es sich entweder um einen einzigen Dateinamen oder um eine ganze Liste von Dateinamen handeln kann. Als Rückgabe liefert Sie die Anzahl der gelöschten Dateien zurück.

```
unlink temp, config, foo;
```

Beachten Sie, dass Perl auf manchen Systemen - allen voran Unix - bedenkenlos auch Nur-Lesen-Dateien löscht. Vorsicht ist also geboten, denn der Nur-Lesen-Status einer Datei bezieht sich lediglich darauf, dass der ***Inhalt*** dieser Datei nicht geändert werden darf, und nicht darauf, ob der Dateiname verschoben oder gelöscht wird.

Wird die Funktion `unlink` auf eine harte Verknüpfung angewendet, wird diese Verknüpfung entfernt. Andere harte Verknüpfungen zu der Datei bleiben bestehen. Auch im Falle einer symbolischen Verknüpfung wird die Verknüpfung entfernt, die Originaldatei bleibt bestehen. Mit `readlink` können Sie symbolischen Verknüpfungen nachfolgen. Beachten Sie, dass nach dem Löschen einer Datei, zu der symbolische Verknüpfungen bestehen, alle symbolischen Verknüpfungen zu der Datei ins Leere zielen.

Sie können mit `unlink` keine Verzeichnisse entfernen. Wie Sie Verzeichnisse löschen, erfahren Sie im Abschnitt »Verzeichnisse verwalten und wechseln«.

Weitere Operationen

Es gibt noch eine Reihe weiterer dateibezogener Funktionen, die ich Ihnen kurz vorstellen möchte. Eine ausführlichere erspare ich mir, da sich die betreffenden Operationen von Plattform zu Plattform unterscheiden (oder unter Umständen unter Windows oder Mac gar nicht existieren). In Tabelle 17.1 sind weitere

Dateiverwaltungsfunktionen aufgeführt, die auf Unix-Systemen verfügbar sind und mit Unix-Befehlen korrspondieren. Wenn Sie eine der Funktionen aus Tabelle 17.1 verwenden möchten, schauen Sie in der Dokumentation zu Ihrer Perl-Version nach, ob diese Funktion unterstützt wird oder nicht (oder ob es für die betreffende Operation anderweitig Unterstützung gibt - zum Beispiel durch ein plattformspezifisches Modul).

Weitere Informationen zu diesen Funktionen finden Sie in der ***perlfunc***-Manpage.

Funktion	Was sie bedeutet
`chmod`	Ändert die Zugriffsrechte für die Datei (`Lesen`, `Schreiben`, `Ausführen`, `für Welt`, `Gruppe` oder `Besitzer`). Windows und Mac haben eingeschränkte Versionen dieses Befehls, der nur Zugriffe von 0666 (`Lesen` und `Schreiben`) und 0444 (`Nur Lesen` oder `gesperrt`) akzeptiert.
`chown`	Ändert den Besitzer dieser Datei (nur Unix)
`fileno`	Liefert den Dateideskriptor für die Datei zurück
`utime`	Ändert die Zeitangaben für eine Datei

Tabelle 17.1: *Weitere Dateiverwaltungsfunktionen*

Um unter Windows und Windows NT Dateizugriffsberechtigungen und -attribute zu verwalten, können Sie auch die Module `win32::File` und `win32::FileSecurity` verwenden. Diese ermöglichen es Ihnen, Dateiattribute und NT-Nutzerrechte zu prüfen und zu ändern.

Verzeichnisse verwalten und wechseln

So wie Sie in Ihren Perl-Skripten mit Dateien in der gleichen Weise arbeiten, wie Sie es von der Shell oder der Befehlszeile aus tun, so können Sie in Ihren Perl-Skripten auch durch das Dateisystem navigieren und Verzeichnisse verwalten (im Sprachgebrauch mancher Betriebssysteme spricht man auch von Ordnern statt von Verzeichnissen). Sie können das aktuelle Verzeichnis ermitteln, das aktuelle Verzeichnis wechseln, den ganzen Inhalt des Verzeichnisses oder einen Teilbereich davon auflisten, und Sie können Verzeichnisse selbst anlegen oder löschen. In diesem Abschnitt möchte ich auf einige Aspekte bei der Ausführung dieser Operationen mit Perl eingehen.

Verzeichnisse wechseln

Jedes Perl-Skript kennt sein aktuelles Arbeitsverzeichnis, das heißt das Verzeichnis aus dem das Skript aufgerufen wurde (wenn Sie gewohnt sind, mit einem befehlszeilenorientierten System zu arbeiten, wird Sie dies nicht überraschen). Auf dem Mac, der mit Droplet-Skripten arbeitet, ist das aktuelle Arbeitsverzeichnis das Verzeichnis, in dem sich das Droplet selbst befindet.

Um das aktuelle Verzeichnis von einem Perl-Skript aus zu ändern, verwenden Sie die Funktion `chdir`:

```
chdir "bilder";
```

Wie schon bei der Verwendung von Verzeichnisnamen im Aufruf der `open`-Funktion sollten Sie darauf achten, dass Sie die Pfadangaben korrekt angeben und die Eigenheiten der verschiedenen Systeme berücksichtigen. Relative Pfadangaben, wie in diesem Beispiel, sind in Ordnung. Obiges Beispiel ändert das aktuelle Arbeitsverzeichnis in das Verzeichnis `bilder`, das unter dem gleichen Verzeichnis angeordnet ist, aus dem das Skript ursprünglich aufgerufen wurde (puh!).

Ist es möglich, herauszufinden, in welchem Verzeichnis Sie sich momentan befinden? Ja, aber wie Sie dazu vorgehen, hängt von dem System ab, auf dem Sie gerade arbeiten. Unter Unix stellen Sie das aktuelle Verzeichnis fest, indem Sie die schrägen Anführungsstrich-Operatoren verwenden, um den Befehl `pwd` aufzurufen (mehr zu den schrägen Anführungszeichen in Kapitel 18, »Perl und das Betriebssystem«):

```
$curr =`pwd`;
```

Diese Methode läßt sich auch auf dem Mac anwenden. Unter Windows erhalten Sie das aktuelle Arbeitsverzeichnis mit der Funktion `Win32::GetCWD`. Der beste Weg, das aktuelle Arbeitsverzeichnis zu ermitteln - ein Weg, der übrigens auf allen Plattformen gleich gut funktioniert - führt allerdings über das Modul `Cwd`, das Teil der Standardbibliothek ist. Das `Cwd`-Modul stellt Ihnen die `cwd`-Funktion zur Verfügung, die das aktuelle Arbeitsverzeichnis zurückliefert (unter Unix führt `cwd` übrigens einfach `` `pwd` `` aus, so dass Sie auch hier bezüglich des Ergebnisses sicher sein können):

```
use Cwd;
$curr = cwd();
```

Dateien auflisten

In Perl gibt es zwei Wege, eine Liste der Dateien in einem Verzeichnis erstellen zu lassen: Die erste Methode verwendet Platzhalter, mit deren Hilfe Sie eine Liste von Dateien erzeugen können, die einem bestimmten Muster entsprechen. Die zweite Methode verwendet die Funktionen `opendir` und `readdir`, die beide eine komplette Liste aller Dateien in einem Verzeichnis ausgeben.

Platzhalter in Dateinamen

Diese Technik, die man im Englischen als Datei-»Globbing« bezeichnet, erlaubt es Ihnen, eine Liste aus Dateien des aktuellen Verzeichnisses zu erstellen, die einem bestimmten einfachen Muster entsprechen. Wenn Sie jemals mit einem befehlszeilenorientierten System gearbeitet und Dateien mit Mustern wie `*.txt` oder `*foo*` aufgelistet haben, dann haben Sie schon mit Datei-Globbing gearbeitet (auch wenn Sie vielleicht nicht wußten, dass man diese Technik so nennt).

> *Der Begriff »**glob**« ist - falls Sie es nicht schon geraten haben - von Unix entlehnt. Verwenden Sie niemals einen einfachen Begriff, wenn es auch kompliziert geht.*

Beim Datei-Globbing nutzen Sie das Platzhalterzeichen `*`, `um` einen bestimmten Satz von Dateinamen zu definieren. Verwechseln Sie das Datei-Globbing aber nicht mit den regulären Ausdrücken - beim Datei-Globbing steht das `*` für ein beliebiges Zeichen (keines oder mehrere), und es ist das einzige Sonderzeichen, das Sie verwenden können. Als Ergebnis erhalten Sie eine Auflistung aller Dateinamen in dem aktuellen Verzeichnis, die dem Muster entsprechen. So liefert zum Beispiel `*.html` alle Dateinamen zurück, die auf die Extension `.html` auslauten, und `*foo*` alle Dateien, die im Namen selbst die Zeichenfolge `foo` enthalten. Sie können diese Dateinamen dann verwenden, um auf den einzelnen Dateien bestimmte Operationen auszuführen.

Perl erzeugt Datei-Globs auf zwei Wegen: mit dem Operator <>, der zwar aussieht wie der Eingabeoperator, aber keiner ist, oder mit der `glob`-Funktion.

Wenn Sie den <>-Operator verwenden, müssen Sie das Muster innerhalb der spitzen Klammern angeben:

```
@dateien = <*.pl>;
```

Dieser Ausdruck, bei dem <> in einem Listenkontext verwendet wird, liefert eine Liste aller Dateien mit der Extension `.pl` zurück. Wenn Sie <> in einem skalaren Kontext verwenden, erhalten Sie jeden Dateinamen separat (wie bei der Eingabe), und wenn Sie den Operator innerhalb einer `while`-Schleife verwenden, werden die Dateinamen nacheinander der `$_`-Variablen zugewiesen, ebenfalls wie bei der Eingabe).

So gibt zum Beispiel das folgende Codebeispiel eine Liste aller Dateien in dem aktuellen Verzeichnis aus, die die Extension `.txt` haben:

```
while (<*.txt>) {
   print $_, "\n";
}
```

Verwechseln Sie den Globbing-Operator <> nicht mit dem <>-Operator für die Eingabe - sie sehen nur äußerlich

gleich aus, arbeiten aber in ganz unterschiedlicher Weise. Letzterer benötigt einen Datei-Handle als Argument und liest den Inhalt dieser Eingabedatei. Ersterer liefert Strings mit den Dateinamen aus dem aktuellen Verzeichnis, die dem Muster entsprechen.

Da man die beiden Operatoren für das Globbing und für die Eingabe leicht verwechseln kann, gibt es noch die `glob`-Funktion. Damit wird das Globbing für alle deutlich und die Verwechslungsgefahr gebannt. Bei der `glob`-Funktion müssen Sie das Filtermuster in Anführungszeichen setzen:

```
@dateien = glob "*.pl";
```

Das Ergebnis ist das gleiche. Auch hier erhalten Sie eine Liste der Dateien, die auf `.pl` enden (in einem skalaren Kontext erhalten Sie die Dateien eine nach der anderen). In modernen Perl-Skripten sollte man dem Datei-Globbing mit der `glob`-Funktion den Vorzug geben.

Verzeichnislisten

Es gibt noch einen zweiten Weg, um eine Liste der Dateien in einem Verzeichnis zu erhalten, und der führt über die Verzeichnis-Handles. Diese erinnern stark an die Datei-Handles und verhalten sich auch so. Sie beziehen sich aber ausschließlich auf Verzeichnisse. Wenn Sie ein Verzeichnis mit Hilfe eines Verzeichnis-Handles einlesen, erhalten Sie eine vollständige Liste aller Dateien in dem Verzeichnis, einschließlich der versteckten Dateien, die mit einem Punkt beginnen (die Sie durch Datei-Globs nicht erhalten), sowie `.` und `..` für das aktuelle und das übergeordnete Verzeichnis in Unix und Windows (der Mac erkennt diese Zeichen nicht als Elemente des aktuellen Verzeichnisses; Sie können statt dessen `:` für das aktuelle und `::` für das übergeordnete Verzeichnis verwenden).

Verzeichnis-Handles werden auf die gleiche Art und Weise geöffnet und geschlossen wie Datei-Handles, nur dass man die Funktionen `opendir` und `closedir` verwendet. Die Funktion `opendir` übernimmt zwei Argumente: den Namen eines Verzeichnis- Handles (von Ihnen gewählt) und das Verzeichnis, das ausgegeben werden soll.

```
opendir(DIR, ".") or die "Verzeichnis kann nicht geöffnet werden: $!\n";
```

Wie bei der `open`-Funktion sollten Sie stets das Ergebnis prüfen und das Skript gegebenenfalls mit Hilfe der `die`-Funktion in Würde sterben lassen (die Variable `$!` ist hier ebenfalls sehr hilfreich).

Bezüglich der Namensgebung gelten für die Verzeichnis-Handles (hier `DIR`) die gleichen Regeln wie für die Datei-Handles. Verzeichnis-Handles werden vollkommen unabhängig von den Datei-Handles betrachtet - Sie können einem Verzeichnis- Handle den gleichen Namen geben wie einem Datei-Handle, es wird keine Konflikte geben.

Sobald ein Verzeichnis-Handle geöffnet ist, können Sie mit `readdir` daraus lesen. Wie schon der Eingabeoperator `<>` liefert auch `readdir` in einem skalaren Kontext einen einzelnen Dateinamen zurück und in einem Listenkontext eine Liste aller Dateinamen. Das folgende Codefragment öffnet zum Beispiel das aktuelle Verzeichnis (das ».«-Verzeichnis unter Unix und Windows; Mac-Anwender schreiben statt dessen »:«) und gibt eine Liste des Verzeichnisses aus:

```
opendir(CURR, ".") or die "Verzeichnis kann nicht geöffnet werden: $!\n";
while (defined($file = readdir(CURR))) {
   print "$file\n";
}
```

Beachten Sie, dass `readdir` - im Gegensatz zum Eingabeoperator `<>` - keine automatischen Zuweisungen an die Variable `$_` vornimmt. Sie müssen den Code dazu selbst in die `while`-Schleife aufnehmen oder (wie ich) eine temporäre Variable verwenden.

Mit `readdir` erhalten Sie eine Liste aller Dateien und Unterverzeichnisse in dem Verzeichnis. Wenn Sie bestimmte Dateien (versteckte Dateien, `.` oder `..`) herausfiltern wollen, müssen Sie dazu einen Vergleich mit einem regulären Ausdruck anschließen.

Sobald Sie das Verzeichnis-Handle nicht mehr benötigen, können Sie es mit der `closedir`-Funktion schließen:

```
closedir(CURR);
```

Verzeichnisse anlegen und löschen

Sie können aus Ihren Perl-Skripten heraus sogar neue Verzeichnisse anlegen und nicht mehr benötigte Verzeichnisse löschen. Die Funktionen dazu lauten `mkdir` und `rmdir`.

Die Funktion `mkdir` übernimmt zwei Argumente: den Namen eines Verzeichnisses (oder Ordners) und einen Satz an Zugriffsberechtigungen. Unter Unix beziehen sich die Zugriffsberechtigungen auf die Standardzugriffsbits des `chmod`-Befehl im Oktalformat (0 plus drei Zahlen von 0 bis 7). Unter Windows und Mac ist das Argument der Zugriffsberechtigung zwar gefordert, aber nicht sinnvoll. Verwenden Sie einfach `0777` als Argument, und Sie dürften keine Probleme bekommen:

```
mkdir temp, 0777;
```

> *`0777` ist eine Oktalzahl, die sich auf die Unix-Zugriffsberechtigungsbits bezieht und Lese-, Schreib- und Ausführungsberechtigungen für die Welt, die Gruppe und den Besitzer bedeutet. Auf dem Mac und unter Windows werden andere Formen von Zugriffsberechtigungen für Dateien und Verzeichnisse verwendet, so dass Sie das Argument der Zugriffsberechtigung an `mkdir` hier eigentlich nicht brauchen (dennoch müssen Sie einen Wert übergeben, und deshalb ist `0777` eine gute, allgemein sinnvolle Wahl.)*

Die `mkdir`-Funktion erzeugt das neue Verzeichnis in dem aktuellen Arbeitsverzeichnis des Skripts. Sie können das Verzeichnis aber auch überall sonstwo anlegen, indem Sie den vollen Pfadnamen angeben, zuerst das aktuelle Verzeichnis wechseln oder - falls Sie Spaß an ausgefallenem Code haben - die Möglichkeiten des Moduls `File::Path` nutzen, das Teil der Standardbibliothek ist und Funktionen zum Erstellen und Löschen ganzer Verzeichnisunterbereiche enthält.

Zum Entfernen eines Verzeichnisses gibt es den Befehl `rmdir`.

```
rmdir temp;
```

Das Verzeichnis muss jedoch leer sein (das heißt es darf keine Dateien enthalten), damit Sie es löschen können.

Ein Beispiel: Verknüpfungen erstellen

Jetzt möchte ich Ihnen ein richtig nützliches Beispiel präsentieren, das ich selbst recht häufig verwende: Wenn Sie ein Verzeichnis voller GIF- oder JPEG-Grafiken haben (in dem Verzeichnis also Dateien mit den Extension `.gif`, `.jpeg` oder `.jpg` stehen), können Sie mit Hilfe dieses Skripts eine HTML-Datei namens `index.html` erzeugen, die Verknüpfungen zu diesen Dateien enthält. Auf diesem Wege können Sie schnell eine große Menge an Grafikdateien in das Web stellen, ohne erst viel Zeit mit dem Anlegen der HTML-Datei zu verschwenden (wenn Sie möchten, können Sie die Datei selbstverständlich danach noch nachbessern).

Das Skript, das ich für diese Aufgabe geschrieben habe, verwendet Datei-Handles, um die Datei `index.html` zu öffnen und in die Datei zu schreiben, Verzeichnis-Handles, um den Inhalt des aktuellen Verzeichnisses einzulesen, und Datei-Tests mit regulären Ausdrücken, um die Dateien herauszufiltern, nach denen wir suchen. In Listing 17.1 sehen Sie das Ergebnis:

Listing 17.1: Das Skript imagegen.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use Cwd;
4:
5:  open(OUT, ">index.html") or
                        die "Index-Datei kann nicht geöffnet werden: $!\n";
6:  &printhead();
7:  &processfiles();
8:  &printtail();
```

```
9:
10: sub printhead {
11:     my $curr = cwd();
12:     print OUT "<HTML>\n<HEAD>\n";
13:     print OUT "<TITLE>Bilddateien aus dem Verzeichnis $curr</TITLE>\n";
14:     print OUT "</HEAD>\n<BODY>\n";
15:     print OUT "<H1>Bilddateien</H1>\n";
16:     print OUT "<P>";
17: }
18:
19: sub processfiles {
20:     opendir(CURRDIR, '.') or
                        die "Verzeichnis kann nicht geöffnet werden ($!),
        exiting.\n";
21:     my $file = "";
22:
23:     while (defined($file = readdir(CURRDIR))) {
24:         if (-f $file and $file =~ /(\.gif|\.jpe?g)$/i) {
25:             print OUT  "<A HREF=\"$file\">$file</A><BR>\n";
26:         }
27:     }
28:     closedir(CURRDIR)
29: }
30:
31: sub printtail {
32:     print OUT "</BODY></HTML>\n";
33:     close(OUT);
34: }
```

Wir beginnen, indem wir die Ausgabedatei (`index.html`) in Zeile 5 öffnen. Achten Sie auf das >-Zeichen, es zeigt Ihnen an, dass es sich hier um ein Handle für eine Ausgabedatei handelt.

Die Subroutine `&printhead()` in den Zeilen 10 bis 17 gibt lediglich den oberen Teil der HTML-Datei aus. Die einzige Schwierigkeit dabei besteht darin, das aktuelle Verzeichnis in den Titel der Seite aufzunehmen. Zu diesem Zweck wird die Funktion `cwd()` aufgerufen (die in Zeile 3 aus dem `Cwd`-Modul importiert wurde). Ich habe hier auf die Verwendung eines Hier-Dokuments für die Ausgabe des HTML-Codes verzichtet, denn es wären dann mit Sicherheit nur noch mehr Zeilen geworden, als es ohnehin schon sind.

In der Subroutine `&processfiles()` (Zeile 19 bis 29) findet die eigentliche Arbeit statt. Zeile 20 öffnet das aktuelle Verzeichnis (Mac-Anwender müssen das »`.`« in »`:`« ändern). Daran schließt sich in den Zeilen 23 bis 26 eine `while`-Schleife an, die jeden Dateinamen im Verzeichnis durchläuft. Der Test in Zeile 24 prüft, ob es sich bei der Datei auch tatsächlich um eine Datei und nicht um ein Verzeichnis handelt (der Test `-f`), und filtert mit Hilfe eines regulären Ausdrücks die Grafikdateien heraus (jene mit der Extension `.gif`, `.jpeg` und `.jpg`). Handelt es sich bei der aktuellen Datei wirklich um eine Grafikdatei, geben wir in Zeile 25 eine Verknüpfung zu dieser Datei aus, wobei wir den Dateinamen als Verknüpfungstext verwenden.

Beachten Sie, dass Sie durch die Verwendung der Dateinamen als Verknüpfungstexte nicht unbedingt zu sehr aussagekräftigen Verknüpfungen kommen - wenn Sie Pech haben, lauten Ihre Verknüpfungen »bild1.gif«, »bild2.gif«, »bild3.gif« und so weiter. Nicht sehr aussagestark, aber zumindest ein Anfang. Nachdem das Skript ausgeführt wurde, können Sie ja die HTML-Datei bearbeiten und aussagekräftigere Verknüpfungen erstellen (»netter Sonnenaufgang«, »Osterumzug«, »Bill verhält sich dumm« und so weiter).

Als Abschluß des Skripts rufen wir die Subroutine `&printtail()` auf, die das Ende der HTML-Datei ausgibt und den Ausgabedatei-Handle schließt.

Vertiefung

Zusätzlich zu den Funktionen, die ich in dieser Lektion und in Kapitel 15 beschrieben habe und die allesamt in der ***perlfunc***-Manpage beschrieben sind, gibt es in der Standard-Modulbibliothek noch eine Reihe weiterer Module mit Funktionen zum Verwalten von Dateien und Datei-Handles.

Viele dieser Module sind einfach objektorientierte Hüllen zu den Standard- Dateioperationen, andere wiederum sind besonders nützlich, um plattformübergreifende Probleme zu lösen oder zu umgehen. Wieder andere stellen einfach Funktionen zur Verfügung, die die Verwaltung von Dateien und Verzeichnissen leichter machen.

Einige dieser Module habe ich bereits erwähnt - beispielsweise `Getopt` und `Cwd`. In Tabelle 17.2 finden Sie eine ausführlichere Liste. Details zu den einzelnen Modulen finden Sie in der ***perlmod***-Manpage.

Modulname	Beschreibung
`CWD`	Ermittelt das aktuelle Arbeitsverzeichnis auf sicherem, plattformübergreifendem Weg
`DirHandle`	Eine objektorientierte Version zum Manipulieren von Verzeichnis-Handles
`File::Basename`	Ermöglicht Ihnen, Datei- und Verzeichnis-Pfadnamen plattformübergreifend zu parsen
`File::CheckTree`	Führt mehrere Dateitests auf einer Gruppe von Dateien durch
`File::Copy`	Kopiert Dateien oder Datei-Handles
`File::Find`	Ähnlich dem Unix-Befehl `find`. Durchforstet einen Verzeichnisbaum nach einer speziellen Datei, die einem bestimmten Muster entspricht
`File::Path`	Erzeugt oder entfernt mehrere Verzeichnisse oder Verzeichnisbäume
`FileCache`	Auf einigen Systemen ist es nicht möglich, eine größere Anzahl von Dateien gleichzeitig geöffnet zu haben. Mit diesem Modul umgehen Sie dieses Problem
`FileHandle`	Eine objektorientierte Version zum Manipulieren von Datei-Handles
`Getopt::Long`	Verwaltet Skript-Argumente (komplizierte POSIXSyntax)
`Getopt::Std`	Verwaltet Skript-Argumente (einfachere Einzelzeichen-Syntax)
`SelectSaver`	Sichert Datei-Handles und stellt sie wieder her (wird zusammen mit der `select`-Funktion verwendet. Diese Funktion besprechen wir in Kapitel 20)

Tabelle 17.2: Dateibezogene Module

Zusammenfassung

Ein Perl-Skript ist keine Insel. Es kommt der Punkt, da ist es sehr wahrscheinlich, dass Ihr Skript sich mit der äußeren Welt des Dateisystems beschäftigen muss, besonders dann, wenn Ihr Skript extensiv Dateien liest und in Dateien schreibt. In Kapitel 15 haben Sie gelernt, wie man den Inhalt einer Datei liest und in eine Datei schreibt. Heute haben wir uns den globaleren Themen der Datei- und Verzeichnisverwaltung innerhalb von Perl-Skripten gewidmet.

Begonnen haben wir mit den Dateien: wie man sie umbenennt, wie man für sie Verknüpfungen erstellt und wie man sie verschiebt - ganz so wie man mit Dateien von der Befehlszeile aus oder in einem grafischen Dateimanager arbeiten würde.

Wenn man Dateien verwalten kann, sollte man auch in der Lage sein, Verzeichnisse zu verwalten. In der zweiten Hälfte dieser Lektion ging es daher darum, wie man Verzeichnisse anlegt und löscht, das aktuelle Arbeitsverzeichnis ausgibt oder wechselt und wie man Listen von Dateien aus einem Verzeichnis einliest (entweder durch Datei-Globbing oder durch Lesen aus einem Verzeichnis-Handle).

Fragen und Antworten

Frage:
Ich versuche, ein portierbares Skript zu schreiben, das eine `config`-Datei einliest. Ich kann in den Pfadnamen für Unix und Windows den Schrägstrich verwenden, aber der Mac mit seinen Doppelpunkten in den Pfadangaben macht die Sache doch sehr kompliziert. Wie kann ich das Problem lösen?

Antwort:
Sie können mit Hilfe eines Tests feststellen, ob Sie Ihr Skript auf einem Mac ausführen, und dann das Pfad-Trennzeichen ordnungsgemäß einrichten (und Ihren Dateipfad durch das Aneinanderhängen von Strings aufbauen). Verwenden Sie das `Config`-Modul, um herauszufinden, auf welcher Plattform Sie sich befinden:

```
    use Config;
if ( $Config{'osname'} =~ /^macos/i ) {
   # ... Macintosh-spezifische Anweisungen
}
```

Frage:
Ich arbeite auf einem Unix-Rechner. Ich habe ein Skript getestet, das den Inhalt einer Datei einliest und die Datei dann löscht. Da ich wollte, dass die Datei erst gelöscht wird, nachdem das Skript debuggt worden ist, habe ich für die Datei die Zugriffsberechtigung zum Schreiben gelöscht. Trotzdem hat Perl die Datei einfach gelöscht. Warum?

Antwort:
Die Zugriffsberechtigungen einer Datei legen lediglich fest, ob Sie den Inhalt der Datei lesen und schreiben können oder nicht. Ob die Datei umbenannt, verschoben oder geändert werden kann, hängt von den Zugriffsberechtigungen des betreffenden Verzeichnisses ab. Wenn Sie in Perl verhindern wollen, dass Dateien gelöscht werden, müssen Sie die Schreibberechtigung für das Verzeichnis löschen, in dem sich die Datei befindet. Oder - noch besser - kommentieren Sie einfach Ihren `unlink`*-Befehl aus, bis Sie den Rest Ihres Skripts debuggt haben. Unter* `http:// www.perl.com/CPAN-local/doc/FMTEYEWTK/file-dir-perms` *finden Sie eine Seite, auf der dieses Problem detailliert beschrieben ist.*

Frage:
Sie haben beschrieben, wie man eine Datei umbenennt, eine Verknüpfung zu einer Datei erstellt und Dateien entfernt. Wie jedoch kopiert man eine Datei?

Antwort:
Sie können dazu beide Dateien öffnen (die Datei, die kopiert werden soll, und die Datei, in die kopiert werden soll) und dann die Zeilen aus der einen Datei auslesen und in die andere ausgeben. Oder Sie nutzen die schrägen Anführungszeichen, um den Kopierbefehl des Systems aufzurufen (wie Sie noch in Kapitel 18 lernen werden). Oder Sie verwenden das Modul `File::Copy`*, das Ihnen eine Kopierfunktion zur Verfügung stellt, mit der Sie Dateien oder Datei-Handles von einer Quelle in ein Ziel kopieren können.*

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist der Unterschied zwischen einer harten und einer symbolischen Verknüpfung? Was passiert jeweils, wenn Sie die Datei löschen, für die die Verknüpfung erstellt wurde?
2. Was ist der Unterschied zwischen `` `pwd` `` und `cwd()`? Warum sollten Sie die eine Form der anderen vorziehen?
3. Was ist Datei-Globbing? Wozu ist es nützlich?
4. Was ist der Unterschied zwischen einem Datei-Globbing mit <*> und einer Liste von Dateien, die Sie mit `readdir()` erhalten?
5. Welche Unterschiede bestehen zwischen Datei-Handles und Verzeichnis-Handles?
6. Was bewirkt das Zugriffsberechtigungsargument für `mkdir`?

Übungen

1. Schreiben Sie ein Skript, das eine Liste von Dateien aus dem aktuellen Verzeichnis einliest und diese in alphabetischer Reihenfolge ausgibt. Sind darunter Verzeichnisse, so geben Sie diese Namen mit einem anschließenden Schrägstrich (`/`) aus.
2. FEHLERSUCHE: Das folgende Skript soll alle Dateien und Verzeichnisse aus dem aktuellen Verzeichnis entfernen. Doch das Skript versagt. Warum?

```
    while (<foo*>) {
   unlink $_;
}
```

3. FEHLERSUCHE: Wo liegt hier der Fehler?

```
    #!/usr/bin/perl -w
opendir(DIR, '.') or die "Verzeichnis kann nicht geöffnet werden ($!)\n";
while (readdir(DIR)) {
  print "$_\n";
}
closedir(DIR);
```

4. Schreiben Sie ein Skript, das Ihnen ein Menü mit fünf Optionen anzeigt: eine Datei erzeugen, eine Datei umbenennen, eine Datei löschen, alle Dateien auflisten und verlassen. Für die Option, eine Datei anzulegen, fordern Sie den Benutzer auf, einen Dateinamen anzugeben. Legen Sie die neue Datei (darf leer sein) in dem aktuellen Verzeichnis an. Für die Optionen zum Umbenennen oder Löschen fordern Sie zuerst den alten Namen der Datei an, die umbenannt oder gelöscht werden soll. Soll die Datei umbenannt werden, fordern Sie den Benutzer darüber hinaus auf, einen neuen Namen anzugeben. Für die Option, alle Dateien aufzulisten, sollten Sie es sich einfach machen. VORSCHLAG: Kümmern Sie sich nicht um Verzeichnisse. Zusatzaufgabe: Führen Sie für alle Dateinamen eine Fehlerprüfung durch, um sicherzustellen, dass Sie nicht eine Datei erzeugen, die bereits existiert, beziehungsweise eine Datei löschen oder umbenennen, die nicht exis-tiert.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Harte Verknüpfungen bieten die Möglichkeit, für eine Datei mehrere Dateinamen vorzusehen. Wenn Sie eine Instanz des Dateinamens entfernen - sei es das Original oder eine Verknüpfung -, hat dies keine Auswirkungen auf die anderen harten Verknüpfungen. Die Datei wird nur dann endgültig gelöscht, wenn auch tatsächlich alle Verknüpfungen dazu gelöscht sind. Symbolische Verknüpfungen bieten ebenfalls die Möglichkeit, eine Datei unter mehreren Dateinamen zu führen, jedoch ist diesmal die Originaldatei getrennt von den Verknüpfungen. Wenn Sie die Verknüpfung entfernen, hat das keine Auswirkung auf die Originaldatei, entfernen Sie hingegen die Originaldatei, weisen die Verknüpfungen ins Leere.
1. Beachten Sie, dass nur Unix-Systeme zwischen harten und symbolischen Verknüpfungen unterscheiden. Es gibt aber auch Unix-Systeme, die keine symbolische Verknüpfungen kennen. Die Aliase des Mac's und die Verknüpfungen unter Windows sind analog den symbolischen Verknüpfungen.
2. Der Befehl `` `pwd` `` verwendet schräge Anführungszeichen, um den Unix-Befehl `pwd` aufzurufen. Während manche Perl-Versionen auf anderen Systemen diesen Aufruf ebenfalls korrekt verarbeiten (namentlich MacPerl), gilt dies bei weitem nicht für alle Perl-Versionen. Soll Ihr Skript problemlos portierbar sein, verwenden Sie das Modul `Cwd` und die Funktion `cwd()`. Denken Sie daran, dass Sie `Cwd` erst importieren müssen (`use Cwd`), bevor Sie es verwenden können.
3. Datei-Globbing ist eine Methode, um eine Liste von Dateien zu erhalten, die einem bestimmten Muster entsprechen. `*.pl` zum Beispiel ist ein Glob, der alle Dateien in einem Verzeichnis zurückliefert, die die Extension `.pl` aufweisen. Globs sind besonders nützlich, wenn man einen Satz von Dateien aufgreifen und auslesen möchte, ohne explizit das Verzeichnis-Handle zu öffnen, die Liste zu lesen und die Dateien auszusortieren, die dem Muster entsprechen.
4. <*> liefert alle Dateien in einem Verzeichnis, abgesehen von den versteckten Dateien, den Dateien, die mit ».« starten, und den Verzeichnissen ».« und »..«. Verzeichnis-Handles liefern alle diese Dinge zurück.
5. Datei-Handles dienen dazu, den Inhalt von Dateien zu lesen und in Dateien zu schreiben (nicht zu vergessen das Schreiben auf den Bildschirm oder in eine Netzwerkverbindung oder ähnliches). Verzeichnis-Handles werden verwendet, um Listen von Dateien aus Verzeichnissen auszulesen. Datei-Handle und Verzeichnis-Handle verwenden ihre eigenen Variablen, so dass es keine Namenskonflikte zwischen ihnen gibt.
6. Das Argument zu `mkdir` legt fest, welche Zugriffsberechtigung das neue Verzeichnis haben soll. Unter Unix folgen diese Zugriffsberechtigungen den normalen Format für Lese-, Schreib- und Ausführungszugriff für Besitzer, Gruppe oder Welt. Für Mac oder Windows ist das Argument der Zugriffsberechtigung ohne Bedeutung, dennoch müssen Sie es mit angeben (verwenden Sie einfach `0777`, und Sie bekommen keine Schwierigkeiten).

Antworten zu den Übungen

1. Hier ist eine Antwort:

```
    #!/usr/bin/perl -w
use strict;
use Cwd;
my $curr = cwd();
opendir(CURRDIR, $curr) or
                    die "Verzeichnis kann nicht geöffnet werden ($!)\n";
my @files = readdir(CURRDIR);
closedir(CURRDIR);
foreach my $file (sort @files) {
    if (-d $file) {
        print "$file/\n";
    }
    else { print "$file\n"; }
}
```

2. Die Funktion `unlink` ist nur für das Löschen von Dateien gedacht. Verzeichnisse werden mit `rmdir` gelöscht. Geht es darum, alle Dateien und Verzeichnisse zu entfernen, müssen Sie die Dateinamen einzeln daraufhin prüfen, ob es sich um eine Datei oder ein Verzeichnis handelt, und dann die entsprechende Funktion aufrufen.
3. Der Fehler liegt in dem Test für die `while`-Schleife. Fälschlicherweise wurde angenommen, dass `readdir` in einer `while`-Schleife den nächsten Verzeichniseintrag der Variablen `$_` zuweist - genauso wie dies beim Einlesen einer Eingabezeile von einem Datei-Handle der Fall ist. Dies stimmt jedoch nicht. Für `readdir` müssen Sie explizit den nächsten Verzeichniseintrag einer Variablen zuweisen:

```
    while (defined($in = readdir(DIR))) { ... }
```

4. Hier sehen Sie ein Beispiel. Achten Sie besonders auf die Subroutine `&getfilename()`, die die Fehlerprüfung durchführt, um sicherzustellen, dass die Datei, die angegeben wurde, auch existiert (zum Umbenennen oder Löschen einer Datei) oder gerade nicht existiert (zum Erzeugen einer neuen Datei).

```
    #!/usr/bin/perl -w
use strict;
my $choice = '';
my @files = &getfiles();
while ($choice !~ /q/i) {
    $choice = &printmenu();
  SWITCH: {
      $choice =~ /^1/ && do { &createfile(); last SWITCH; };
      $choice =~ /^2/ && do { &renamefile(); last SWITCH; };
      $choice =~ /^3/ && do { &deletefile(); last SWITCH; };
      $choice =~ /^4/ && do { &listfiles(); last SWITCH; };
      $choice =~ /^5/ && do { &printhist(); last SWITCH; };
  }
}
sub printmenu {
    my $in = "";
    print "Wählen Sie einen Befehl (or Q to quit): \n";
    print "1. Datei erzeugen\n";
    print "2. Datei umbenennen\n";
    print "3. Datei loeschen\n";
    print "4. Alle Dateien auflisten\n";
    while () {
        print "\nIhre Wahl --> ";
        chomp($in = <STDIN>);
        if ($in =~ /^[1234]$/ || $in =~ /^q$/i) {
            return $in;
        } else {
            print "Keine gueltige Wahl.  1-4 oder Q, bitte,\n";
        }
    }
}
sub createfile {
    my $file = &getfilename(1);
    if (open(FILE, ">$file")) {
        print "Datei $file wurde erzeugt.\n\n";
        @files = &getfiles();
        close(FILE);
```

```
        } else {
            warn "$file konnte nicht geoeffnet werden ($!).\n";
        }
    }
    sub renamefile {
        my $file = &getfilename();
        my $in = '';
        print "Geben Sie den neuen Dateinamen ein: ";
        chomp($in = <STDIN>);
        if (rename $file,$in) {
            print "Datei $file wurde in $in umbenannt.\n";
            @files = &getfiles();
        } else {
            warn "$file konnte nicht umbenannt werden ($!).\n";
        }
    }
    sub deletefile {
        my $file = &getfilename();
        if (unlink $file) {
            print "Datei $file wurde geloescht.\n";
            @files = &getfiles();
        } else {
            warn "$file konnte nicht geloescht werden ($!).\n";
        }
    }
    sub getfilename {
        my $in = '';
        # ohne Argumente aufrufen, um sicherzustellen, dass die
        # Datei existiert
        my $new = 0;
        # mit Argument aufrufen, um sicherzustellen, dass die
        # Datei nicht existiert
        if (@_) {
           $new = 1;
        }
        while (!$in) {
            print "Geben Sie einen Dateinamen ein (? fuer Liste): ";
            chomp($in = <STDIN>);
            if ($in eq '?') {
                &listfiles();
                $in = '';
            } elsif ((grep /^$in$/, @files) && $new) {
                # Datei existiert, keine neue Datei, OK
                # Datei existiert, neue Datei: Fehler
                print "Datei ($in) existiert bereits.\n";
                $in = '';
            } elsif ((!grep /^$in$/, @files) && !$new) {
                # Datei existiert nicht, neue Datei, OK
                # Datei existiert nicht, keine neue Datei: Fehler
                print "Datei ($in) nicht gefunden.\n";
                $in = '';
            }
        }
        return $in;
    }
    sub getfiles {
        my $in = '';
        opendir(CURRDIR, '.') or
                          die "Verzeichnis konnte nicht geoeffnet werden ($!),
        exiting.\n";
        @files = readdir(CURRDIR);
        closedir(CURRDIR);
        return @files;
    }
    sub listfiles {
        foreach my $file (@files) {
            if (-f $file) { print "$file\n"; }
        }
        print "\n";
    }
```

1.
1.

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl und das Betriebssystem

Bisher habe ich mich auf die Perl-Elemente beschränkt, die sich überall gleich verhalten, unabhängig davon, ob Sie Ihr Skript auf einem Unix-System, unter Windows oder auf einem Mac ausführen (zumindest habe ich Sie auf Unterschiede, soweit vorhanden, hingewiesen). Zum Glück gibt es, soweit es den Kern der Sprache betrifft, nicht allzu viele Punkte, die eine plattformübergreifende Skripterstellung schwermachen: Ein Hash ist ein Hash ist ein Hash, egal in welcher Sprache Sie ihn betrachten.

Perl enthält aber auch Elemente, die nicht portierbar sind. Bei manchen Elementen hat die Bindung an bestimmte Plattformen einen geschichtlichen Hintergrund. Da Perl ursprünglich für Unix entwickelt wurde, gründen viele vordefinierten Elemente auf Besonderheiten von Unix, die es auf anderen Plattformen nicht gibt. Andere Elemente sind in plattformspezifischen Modulen untergebracht und beziehen sich dann natürlich ausschließlich auf diese Plattform (zum Beispiel die Windows-Registrierdatenbank oder die Mac Toolbox). Wenn Sie sicher sind, dass Ihre Skripts nur auf einer bestimmten Plattform ausgeführt werden, können Sie diese Module nutzen, um plattformspezifische Probleme zu lösen. Aber auch wenn Sie ein Skript erhalten haben, das von einer Plattform auf eine andere portiert werden soll, ist es hilfreich, wenn Sie wissen, welche Elemente welcher Plattform eigen sind.

Heute wollen wir uns einige der plattformspezifischen Elemente von Perl anschauen, die in die Sprache oder die Module Eingang gefunden haben. Insbesondere wollen wir heute folgendes untersuchen:

- Spezielle Unix-Features wie die schrägen Anführungszeichen und die Funktion `system`
- Wie man Unix-Prozesse mit `fork` und `exec` erzeugt
- Funktionen zur Handhabung von Unix-Systemdateien
- Die Windows- und Mac-Kompatibilität der Unix-Features
- Die Win32-Modulgruppe einschließlich `Win32::Process` und `Win32::Registry`
- Wie man Dialogfenster in MacPerl erzeugt

Unix-Features in Perl

Das Unix-Erbe zeigt sich in vielen der vordefinierten Elemente, die direkt von Unix- Tools wie zum Beispiel den Shells entlehnt sind oder sich speziell auf die Verwaltung bestimmter Unix-Dateien beziehen. Deshalb wollen wir in diesem Abschnitt die Elemente von Perl untersuchen, die auf Unix-Systemen nützlich sind:

- Arbeit mit Umgebungsvariablen
- Die Funktion `system`, um andere Programme auszuführen
- Andere Programme ausführen und deren Ausgabe mit schrägen Anführungszeichen auffangen
- Mit `fork`, `wait` und `exec` neue Prozesse erzeugen und verwalten
- Funktionen zum Verwalten von Unix-Benutzer- und Gruppeninformationen

Beachten Sie, daß mit Ausnahme der Prozesse viele dieser Elemente auch in Perl- Versionen für andere Systeme verfügbar sein können, unter Umständen aber mit etwas anderem oder begrenzterem Funktionsumfang. Also auch, wenn Sie unter Windows oder auf einem Mac arbeiten, empfehle ich Ihnen, diesen Abschnitt zumindest zu überfliegen, bevor Sie mit dem Teil fortfahren, der sich mit Ihrer Plattform beschäftigt.

Umgebungsvariablen

Perl-Skripts erben wie Shell-Skripts ihre Umgebung (das heißt den aktuellen Ausführungspfad, den Benutzernamen, die Shell und so weiter) von der Shell, in der sie gestartet wurden (oder von der Benutzer-ID, von der sie ausgeführt werden). Und wenn Sie von Ihrem Perl-Skript aus andere Programme ausführen oder Prozesse starten, erhalten diese ihre Umgebung wiederum von Ihrem Skript. Wenn Sie Perl- Skripts von der Befehlszeile aus ausführen, dürften diese Variablen nicht von großem Interesse für Sie sein. Aber Perl-Skripts, die in anderen

Umgebungen ausgeführt werden, können über zusätzliche Variablen verfügen, die sich auf diese Umgebung beziehen, oder weisen für diese Variablen andere Werte auf, als Sie es erwarten. CGI- Skripts verfügen zum Beispiel, wie Sie in Kapitel 16, »Perl für CGI-Skripts«, gelernt haben, über eine Reihe von Umgebungsvariablen, die mit verschiedenen CGI- spezifischen Konzepten verbunden sind.

Perl speichert alle seine Umgebungsvariablen in einem speziellen Hash namens `%ENV`, in dem die Namen der Variablen die Schlüssel und die Werte dieser Variablen die Werte zu den Schlüsseln sind. Umgebungsvariablen werden in der Regel in Großbuchstaben geschrieben. Um zum Beispiel den Ausführungspfad für Ihr Skript auszugeben, würden Sie folgende Zeile verwenden:

```
print "Pfad: $ENV{PATH}\n";
```

Sie können alle Umgebungsvariablen mit einer regulären `foreach`-Schleife ausgeben:

```
foreach $key (keys %ENV) {
   print "$key -> $ENV{$key}\n";
}
```

Unix-Programme ausführen

Wollen Sie von Ihrem Perl-Skript aus Unix-Befehle ausführen? Kein Problem. Rufen Sie dazu die Funktion `system` wie folgt auf:

```
system('ls');
```

In diesem Fall führt `system` einfach den Unix-Befehl `ls` aus, mit dem der Inhalt des aktuellen Verzeichnisses auf der Standardausgabe ausgegeben wird. Die auszuführenden Befehle lassen sich auch um Optionen ergänzen, die dann jedoch in dem String-Argument eingeschlossen sein müssen. Alles, was Sie zusammen mit einem Shell-Befehl eingeben können (und was für den aktuellen Ausführungspfad verfügbar ist), können Sie als Argument für `system` mit aufnehmen.

```
system("find t -name '*.t' -print | xargs chmod +x &");
system('ls -l *.pl');
```

Wenn Sie als Argument zu `system` einen String in doppelten Anführungszeichen verwenden, wird Perl Variablen interpolieren, bevor es den String an die Shell übergibt:

```
system("grep $thing $file | sort | uniq >neueDatei.txt");
```

*Seien Sie recht vorsichtig damit, Daten an die Shell zu übergeben, die Sie nicht persönlich verifiziert haben (zum Beispiel Daten, die irgendein Benutzer über die Tastatur eingegeben hat). Bösartige Benutzer könnten Ihnen so Daten zuspielen, die - bei ungeprüfter Übergabe an die Shell - Ihrem System großen Schaden zufügen oder irgend jemandem unberechtigten Zugriff auf Ihr System einräumen könnten. Deshalb sollten Sie zumindest die eingehenden Daten überprüfen, bevor Sie sie an die Shell übergeben. Alternativ gibt es in Perl einen Mechanismus, den sogenannten **Taint-Modus**, der es Ihnen erlaubt, potentiell unsichere (tainted = vergiftet, befleckt) Daten zu kontrollieren und zu verwalten. Weitere Informationen sind in der **perlsec**-Manpage enthalten.*

Der Rückgabewert der `system`-Funktion entspricht dem Rückgabewert des Shell- Befehls: im Erfolgsfall 0 und andernfalls 1 oder größer. Beachten Sie, daß dies genau umgekehrt ist wie bei den Standardwerten, die Perl für ***wahr*** und ***falsch*** verwendet. Wenn Sie also testen wollen, ob bei dem Aufruf von `system` Fehler aufgetreten sind, werden Sie ein logisches `and` anstatt `or` verwenden müssen:

```
system('who') and die "who konnte nicht ausgeführt werden\n";
```

Wenn der Befehl `system` ausgeführt wird, übergibt Perl das String-Argument einer Shell (in der Regel `/bin/sh`), wo die Shell-Metazeichen erweitert werden (zum Beispiel Variablen oder Dateinamen-Globs) und der Befehl

anschließend ausgeführt wird. Wenn Sie keine Shell-Metazeichen verwenden, können Sie den Prozeß dadurch beschleunigen, daß Sie `system` eine Liste von Argumenten anstelle eines einzelnen Strings übergeben. Das erste Element der Liste sollte der Name des auszuführenden Befehls sein und alle weiteren Elemente die verschiedenen Argumente zu diesem Befehl:

```
system("grep $thing $file");  # startet eine Shell
system("grep", "$thing", "$file");
# umgeht die Shell, ist etwas effizienter
```

Perl nimmt Ihnen diese Optimierung ab, wenn Ihr String-Argument einfach genug ist - das heißt, wenn es nicht irgendwelche Sonderzeichen enthält, die die Shell bearbeiten muß, bevor das Programm ausgeführt werden kann (zum Beispiel Shell- Variablen oder Dateinamen-Globs).

Unabhängig davon, ob Sie nun ein einfaches String-Argument oder eine Liste von Argumenten übergeben, löst die `system`-Funktion am Ende neue Unterprozesse für jeden der Befehle in ihrem Argument aus. Jeder neue Prozeß erbt seine aktuellen Umgebungsvariablen von den Werten in `%ENV` und teilt die Standardeingabe, -ausgabe und -fehlerausgabe mit dem Perl-Skript. Perl wartet, bis der Befehl zu Ende ausgeführt ist, bevor es mit dem Skript fortfährt (es sei denn, der Befehl endet auf ein `&`, so daß der Befehl im Hintergrund ausgeführt wird - ganz wie das auch in der Shell der Fall wäre).

Überlegen Sie genau, bevor Sie die `system`-Funktion verwenden. Da `system` für jeden auszuführenden Befehl einen eigenen Prozeß startet (und manchmal auch für die Shell, die diese Befehle ausführt), können diese ganzen zusätzlichen Prozesse Ihr Perl-Skript überlasten. Normalerweise ist es besser, eine Aufgabe mit etwas Code innerhalb Ihres Perl-Skripts zu lösen, als für die gleiche Aufgabe eine Unix-Shell zu starten. Und außerdem ist Perl-Code besser portierbar.

Eingaben mit schrägen Anführungszeichen

Sie haben bereits gelernt, wie Sie Eingaben über die Standardeingabe oder Datei- Handles in ein Perl-Skript einlesen. Die dritte Möglichkeit besteht in der Verwendung von schrägen Anführungszeichen ``, einem geläufigen Paradigma in Unix-Shells.

Schräge Anführungszeichen entsprechen in ihrer Funktionsweise in etwa dem `system`- Befehl - beide führen einen Unix-Befehl innerhalb eines Perl-Skripts aus. Der Unterschied liegt in der Ausgabe. Befehle, die mit `system` ausgeführt werden, leiten ihre Ausgabe an die Standardausgabe. Wenn Sie jedoch schräge Anführungszeichen verwenden, um einen Unix-Befehl auszuführen, wird die Ausgabe des Befehls je nach Kontext, in dem die schrägen Anführungszeichen verwendet wurden, entweder als String oder als eine Liste von Strings aufgefangen.

Betrachten wir zum Beispiel den Befehl `ls`, der eine Liste des aktuellen Verzeichnisses ausgibt:

```
$ls = `ls`;
```

Hier führen die schrägen Anführungszeichen den Befehl `ls` in einer Unix-Shell aus, und die Ausgabe des Befehls (die Standardausgabe) wird der Skalarvariablen `$ls` zugewiesen. In einem skalaren Kontext (wie bei diesem Beispiel) wird die resultierende Ausgabe in einem einzigen String gespeichert, in einem Listenkontext wird jede Zeile der Ausgabe ein eigenes Listenelement.

Wie schon bei `system`, können Sie jeden Befehl, den Sie in einer Unix-Shell ausführen können, auch in schräge Anführungszeichen setzen. Der Befehl wird dann in seinem eigenen Prozeß ausgeführt, erbt seine Umgebungsvariablen von `%ENV` und teilt auch Standardeingabe, -ausgabe und -fehlerausgabe. Der Inhalt des Strings in den schrägen Anführungszeichen wird wie die Strings in doppelten Anführungszeichen von Perl variableninterpoliert. Der Rückgabestatus des Befehls wird in der Sondervariablen `$?` gespeichert. Und wie bei `system` lautet der Rückgabestatus im Erfolgsfall 0 und andernfalls 1 oder größer.

Mit Prozessen arbeiten: fork, wait und exec

Wenn Sie ein Perl-Skript ausführen, läuft es als sein eigener Unix-Prozeß. Für viele einfache Skripts benötigen Sie unter Umständen nur einen einzigen Prozeß, vor allem wenn Ihr Skript vornehmlich linear, das heißt Schritt für Schritt vom Anfang bis zum Ende, abgearbeitet wird. Wenn Sie jedoch beginnen, komplexere Skripts zu erstellen, in denen verschiedene Teile des Skripts gleichzeitig verschiedene Aufgaben lösen müssen, werden Sie daran interessiert sein, weitere Prozesse zu erzeugen, die unabhängig vom Rest des Skripts ausgeführt werden. Hierfür wird die `fork`-Funktion verwendet. Nachdem Sie einen neuen Prozeß erzeugt haben, können Sie seine Prozeß-ID (PID) verfolgen, das Ende des Prozesses abwarten oder ein anderes Programm in diesem Prozeß ausführen. All dies möchte ich Ihnen in diesem Abschnitt zeigen.

Neue Prozesse zu erzeugen und deren Verhalten zu steuern, ist eines der Konzepte von Perl für Unix, das sich nur schlecht auf anderen Systemen nachvollziehen läßt. So können Sie zwar mit der Erzeugung neuer Prozesse die Leistung Ihrer Perl-Skripts beträchtlich steigern, aber wenn Ihre Skripts portierbar sein sollen, werden Sie diese Besonderheiten eher vermeiden oder sich überlegen, wie Sie das Problem auf anderen Plattformen umgehen können.

Interessant sind in dieser Hinsicht die sogenannten Threads, eine experimentelle Neueinführung in Perl 5.005, die Hilfe bei der Portierung prozeßbasierter Skripts auf andere Plattformen versprechen. Threads bieten fast den gleichen Leistungsumfang wie Unix-Prozesse, sind jedoch effizienter und lassen sich auf alle Plattformen portieren. Zum Zeitpunkt der Drucklegung dieses Buches sind die Threads jedoch noch absolut neues Terrain, auf dem noch viel experimentiert wird. Ich werde in Kapitel 21, »Ein paar längere Beispiele«, noch einmal kurz darauf eingehen.

Wie Prozesse funktionieren

Prozesse werden eingesetzt, um verschiedene Teile Ihres Skripts gleichzeitig auszuführen. Wenn Sie in einem Skript einen neuen Prozeß erzeugen, läuft dieser Prozeß unabhängig in einem eigenen Speicherbereich weiter, bis er zu Ende ist oder Sie den Prozeß abbrechen. Sie können von Ihrem Skript aus so viele Prozesse starten, wie Sie benötigen. Grenzen sind Ihnen nur durch Ihr System gesetzt.

Wozu benötigen Sie mehrere Prozesse? Zum Beispiel, wenn Sie verschiedene Teile Ihres Programms gleichzeitig ausführen wollen oder mehrere Kopien Ihres Programms gleichzeitig ausgeführt werden sollen. Häufig werden Prozesse zum Einrichten von netzwerkbasierten Servern eingesetzt, die darauf warten, daß ein Client eine Verbindung herstellt und diese Verbindung dann in irgendeiner Weise bearbeitet. Wenn ein solcher Server nur einen einzigen Prozeß verwendet und es wird eine Verbindung hergestellt, dann »erwacht« der Server und bearbeitet die Verbindung (parst die Eingabe, holt Werte aus einer Datenbank, liefert Dateien zurück - was auch immer). Solange der Server seine Verbindung bearbeitet, muss jede zweite Verbindung, die gerade ankommt, warten. Bei einem sehr stark frequentierten Server kann das bedeuten, daß eine ganze Menge von Verbindungen in der Warteschleife hängen und darauf warten, daß der Server bald fertig ist und zur Bearbeitung der nächsten Verbindung schreitet.

Wenn Sie Ihren Server so einrichten, daß er mit Prozessen arbeitet, kann Ihr Skript aus einem Hauptteil bestehen, der nur auf die Verbindungen wartet, und aus einem zweiten Teil, der diese Verbindungen bearbeitet. Erhält der Hauptserver dann eine Verbindung, löst er einen neuen Prozeß aus, reicht die Verbindung an diesen neuen Prozeß zur Bearbeitung weiter und ist dann wieder frei, um auf neue eingehende Verbindungen zu warten. Der zweite Prozeß verarbeitet die Eingaben aus der Verbindung und stirbt, wenn er seine Aufgabe erledigt hat. Diese Vorgehensweise kann für jede neue Verbindung wiederholt werden, so daß die Verbindungen parallel abgearbeitet werden und nicht seriell.

Am Beispiel der Netzwerk-Server läßt sich gut darlegen, warum Prozesse so nützlich sind. Sie brauchen aber nicht unbedingt ein Netzwerk für den Einsatz von Prozessen. Immer, wenn Sie verschiedene Teile Ihres Skripts parallel ausführen oder einen bearbeitungsintensiven Teil Ihres Skripts vom Hauptteil trennen wollen, bieten sich Prozesse als Lösung an.

Wenn Sie Threads bereits von Sprachen wie Java her kennen, könnten Sie dem Glauben erliegen, Sie verstünden auch schon, was Prozesse sind. Doch seien Sie vorsichtig. Im Gegensatz zu Threads sind laufende Prozesse völlig unabhängig voneinander. Die Eltern- und Kindprozesse laufen unabhängig voneinander ab. Es werden kein Speicherplatz und auch keine Variablen geteilt, und es ist ziemlich schwierig, Informationen zwischen den Prozessen auszutauschen. Für die Kommunikation zwischen den Prozessen müssen Sie einen besonderen

Mechanismus einrichten, den sogenannten IPC (Interprocess Communication). Im Rahmen dieses Buches ist kein Platz, um näher auf IPC einzugehen, aber im Vertiefungsabschnitt am Ende dieses Kapitels werde ich Ihnen einige Hinweise geben.

fork und exit

Um in einem Perl-Skript einen neuen Prozeß zu erzeugen, verwendet man die `fork`- Funktion. `fork`, das keine Argumente übernimmt, erzeugt einen neuen zweiten Prozeß zusätzlich zu dem Prozeß für das Originalskript. Jeder neue Prozeß ist ein Klon des ersten, mit den gleichen Werten für die gleichen Variablen (auch wenn er diese nicht mit dem Elternprozeß teilt, denn sie befinden sich in unterschiedlichen Speicherbereichen). Der Kindprozess führt parallel das gleiche Skript bis zum Ende aus und verwendet dabei die gleiche Umgebung sowie die gleiche Standardein- und - ausgabe wie der Elternprozeß. Ab der `fork`-Funktion ist es, als ob Sie zwei Kopien des gleichen Prozesses gestartet hätten.

Das gleiche Skript zweimal auszuführen, ist normalerweise jedoch nicht der Grund, einen neuen Prozeß zu erzeugen. In der Regel setzen Sie einen neuen Prozeß (auch Kind genannt) ein, um etwas anderes als den ersten Prozeß (auch Eltern genannt) auszuführen. Am häufigsten wird `fork` deshalb in `if`-Bedingungen verwendet, die den Rückgabewert von `fork` abfragen. Je nachdem ob der aktuelle Prozeß ein Eltern- oder ein Kindprozeß ist, liefert `fork` unterschiedliche Werte zurück. Für den Elternprozeß wird die Prozeß-ID (PID) des neuen Prozesses zurückgeliefert, und für den Kindprozeß lautet der Rückgabewert 0 (wenn, aus was für Gründen auch immer, `fork` nicht ausgeführt wird, lautet der Rückgabewert `undef`). Durch Testen dieses Rückgabewertes können Sie im Kindprozeß anderen Code ausführen als im Elternprozeß.

Das Codegerüst zum Erzeugen von Prozessen sieht meist folgendermaßen aus:

```
if (defined($pid = fork)) {  # fork funktionierte
   if ($pid) {               # pid ist eine Zahl, dies ist der Elternprozess
      &parent();
   } else {                  # pid ist 0, dies ist der Kindprozess.
      &child();
   }
} else {              # fork funktionierte nicht, neuer Versuch oder Abbruch
   die "Fork funktionierte nicht...\n";
}
```

In diesem Beispiel ruft die erste Zeile `fork` auf und speichert das Ergebnis in der Variablen `$pid` (die Variable für die Prozeß-IDs wird fast immer `$pid` genannt, Sie können aber auch einen beliebigen anderen Namen vergeben). Es gibt drei mögliche Ergebnisse: eine Prozeß-ID, 0 oder `undef`. Der Aufruf von `define` in der ersten Zeile prüft, ob die Funktion erfolgreich ausgeführt wurde. Sollte dies nicht der Fall sein, springt die Skriptausführung zu dem äußeren `else` und bricht mit einer Fehlermeldung ab.

Wenn `fork` aufgrund eines Fehlers nicht ausgeführt wird, wird die aktuelle Fehlermeldung (oder Fehlernummer, je nachdem, was Sie verwenden) in der globalen Systemvariablen `$!` gespeichert. Da viele `fork`-Fehler dazu neigen, vorübergehender Art zu sein (ein überladenes System hat unter Umständen im Moment keine neuen Prozesse zur Verfügung), testen einige Perl-Programmierer, ob `$!` den String »`No more Processes`« enthält, warten dann einen Augenblick und versuchen später erneut, mit `fork` zu verzweigen.

Ein erfolgreiches Ergebnis ist entweder eine 0 oder eine Zahl für die Prozeß-ID des neuen Prozesses. Das Ergebnis teilt dem Skript mit, welcher Prozeß es ist. In obigem Codefragment habe ich zwei fiktive Subroutinen, `&parent()` und `&child()`, aufgerufen, die unterschiedliche Teile des Skripts ausführen, je nachdem ob das Skript als Elternprozeß oder als Kindprozeß ausgeführt wird.

Sehen Sie im folgenden ein einfaches Beispiel für ein Skript, das in drei Kindprozesse verzweigt und im Elternprozeß sowie den drei Kindprozessen entsprechende Meldungen ausgibt. Am Ende des Skripts wird die Meldung »`Ende`« ausgegeben.

Listing 18.1: prozesse.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my $pid = undef;
5:
6:  foreach my $i (1..3) {
7:      if (defined($pid = fork)) {
8:          if ($pid) { #Eltern
9:              print "Eltern: Kind $i ($pid) gestartet \n";
10:         } else {    #Kind
11:             print "Kind $i: wird ausgeführt\n";
12:             last;
13:         }
14:     }
15: }
16:
17: print "Ende...\n";
```

Die Ausgabe dieses Skripts wird in etwa folgendermaßen aussehen (vielleicht variiert die Ausgabe auf Ihrem System etwas):

```
% prozesse.pl
Eltern: Kind 1 (8577) gestartet
Eltern: Kind 2 (8578) gestartet
Eltern: Kind 3 (8579) gestartet
Ende...
%
Kind 1: wird ausgeführt
Ende...
Kind 2: wird ausgeführt
Kind 3: wird ausgeführt
Ende...
Ende...
```

Diese Ausgabe ist vielleicht etwas überraschend. Die Ausgaben der Prozesse sind vermischt, und wo kommt der zusätzliche Prompt in der Mitte her? Warum gibt es vier `Ende...`-Meldungen?

Die Antwort auf all diese Fragen liegt darin, wie die Prozesse ausgeführt werden und was zu welcher Zeit ausgegeben wird. Beginnen wir unsere Betrachtungen mit dem, was in dem Elternprozeß passiert:

- Ein neuer Prozeß wird gestartet (Zeile 7). Im Elternprozeß wird in der Variablen `$pid` die Prozeß-ID des neuen Prozesses abgelegt.
- Es wird getestet, ob `$pid` einen Wert ungleich Null hat. Wenn ja, wird für jeden verzweigten Prozeß eine Meldung ausgegeben (Zeilen 8 und 9).
- Diese Schritte werden für jeden Durchlauf der `foreach`-Schleife (noch zweimal) wiederholt.
- Es wird `Ende...` ausgegeben.
- Das Skript wird verlassen (der System-Prompt wird ausgegeben).

All dies geschieht ziemlich schnell, so daß die Ausgabe des Elternprozesses sehr schnell erfolgt. Kommen wir jetzt zu den drei Kindprozessen, deren Ausführung direkt nach der `fork`-Funktion beginnt:

- Zeile 8 fragt den Wert von `$pid` ab. Da `$pid` für jeden der drei Kindprozesse 0 ist, ergibt der Test in Zeile 8 ***falsch,*** und die Ausführung wird in Zeile 10 fortgesetzt. Anschließend wird die Meldung in Zeile 11 ausgegeben.
- Die `foreach`-Schleife wird direkt mit `last` verlassen. Ohne ein `last` an dieser Stelle würde der Kindprozeß fortfahren, die Schleife zu wiederholen (denken Sie daran, daß der Kindprozeß genau dort einsetzt, wo `fork` aufhört. Er ist ein Klon des Elternprozesses).
- Es wird `Ende...` ausgegeben.
- Das Skript wird verlassen. Es wird kein System-Prompt ausgegeben, da der Kindprozeß durch den Elternprozeß ausgelöst wurde.

Die Ausgaben der Prozesse mischen sich beim Schreiben an die Standardausgabe. Da die Kindprozesse einige Zeit benötigen, bevor sie mit der Ausführung beginnen, ist der Elternprozeß bereits beendet, und seine Ausgaben sind auf dem Bildschirm, noch bevor überhaupt einer der Kindprozesse begonnen hat. Beachten Sie, daß die Zeile, die »`Ende...`« ausgibt, sowohl für den Eltern- als auch den Kindprozeß ausgeführt wird; da der Kindprozeß den

gleichen Code abarbeitet wie der Elternprozeß, wird er über seinen `else`-Block hinaus mit der Ausführung des Codes fortfahren.

Unter Umständen ist dieses Verhalten jedoch unerwünscht. Vielleicht soll der Elternprozeß erst nach dem Kindprozeß beendet werden, oder der Kindprozeß soll gestoppt werden, wenn ein spezieller Codeblock ausgeführt worden ist. Oder Sie wollen, daß der Elternprozeß wartet, bis der letzte Kindprozeß beendet ist, bevor der nächste Kindprozeß gestartet wird. Hierzu müssen Sie in die Prozeßverwaltung einsteigen, die ich im nächsten Abschnitt besprechen werde.

Prozeßverwaltung mit exit und wait (und manchmal kill)

Einen Kindprozeß mit `fork` zu starten und dann die Ausführung dieses Prozesses einfach laufen lassen, ist wie wenn man ein vierjähriges Kind frei in einer öffentlichen Anlage herumlaufen ließe. Sie erhalten zwar Ergebnisse, aber diese werden nicht unbedingt Ihren Erwartungen (oder denen anderer Anwender) entsprechen. Hier kommt die Prozeßsteuerung ins Spiel. Die zwei Funktionen, `exit` und `wait`, helfen Ihnen, Ihre Prozesse zu kontrollieren und zu steuern.

Beginnen wir mit der `exit`-Funktion. Die `exit`-Funktion bricht ganz einfach die Ausführung des aktuellen Skripts an der Stelle ab, wo die Funktion steht. Sie entspricht in dieser Hinsicht in etwa der Funktion `die`. Während die Funktion `die` das Skript jedoch mit einem Fehlerstatus (unter Unix) abbricht und eine Fehlermeldung ausgibt, beendet `exit` das Programm mit einem normalen Statusargument (0 für Erfolg, andernfalls 1).

`exit` wird meistens dazu verwendet, einen Kindprozeß daran zu hindern, mehr vom Code des Elternprozesses auszuführen, als erwünscht ist. Wenn Sie `exit` an das Ende des Codeblocks für den Kindprozeß setzen, wird der Kindprozeß nur bis zu diesem Punkt ausgeführt und bricht dort ab. Lassen Sie uns deshalb in dem kleinen Skript `prozesse.pl` aus dem letzten Abschnitt den Aufruf von `last` durch einen Aufruf von `exit` ersetzen:

```
# ...
} else {    #child
    print "Kind $i: wird ausgeführt\n";
    exit;
}
```

Mit dieser Änderung wird der Kindprozeß seine Nachricht ausgeben und dann abbrechen. Er wird die `foreach`-Schleife nicht erneut starten und auch nicht die »`Ende...`«-Anweisung ausgeben. Der Elternprozeß dagegen, der den anderen Teil der `if`-Anweisung ausführt, führt die »`Ende...`«-Anweisung aus, nachdem die Schleife beendet ist. Die Ausgabe dieser Version des Skripts sieht folgendermaßen aus:

```
% procexit.pl
Eltern: Kind 1 (11828) gestartet
Eltern: Kind 2 (11829) gestartet
Eltern: Kind 3 (11830) gestartet
Ende...
%
Kind 1: wird ausgeführt
Kind 2: wird ausgeführt
Kind 3: wird ausgeführt
```

Wie schon im vorigen Beispiel sind die Ausgaben der Eltern- und Kindprozesse vermischt, und der Elternprozeß wird vor den Kindprozessen abgeschlossen.

Wenn Sie noch mehr Einfluß darauf nehmen wollen, wann die Kindprozesse ausgeführt werden und wann der Elternprozeß beendet wird, verwenden Sie die Funktionen `wait` und `waitpid`. Beide Funktionen bewirken das gleiche: Die Ausführung des aktuellen Prozesses (meist der Elternprozeß) wird so lange angehalten, bis der Kindprozeß zu Ende ist. Damit wird der vermischten Ausgabe ein Ende bereitet und der Elternprozeß erst nach den Kindprozessen beendet. In komplizierteren Skripts als unserem Beispiel kann man auf diese Weise verhindern, daß das Skript sogenannte »Zombie«-Prozesse zurückläßt (das sind Kindprozesse, deren Ausführung beendet ist, die aber immer noch im System herumhängen und Ressourcen belegen).

Der Unterschied zwischen `wait` und `waitpid` liegt darin, daß `wait` keine Argumente übernimmt und darauf wartet, daß irgendeiner der Kindprozesse zurückkehrt. Wenn Sie fünf Prozesse auslösen und dann `wait` aufrufen, wird

`wait` ein erfolgreiches Ergebnis zurückliefern, wenn einer der fünf Kindprozesse beendet wird. `Waitpid` hingegen übernimmt als Argument eine bestimmte Prozeß-ID und wartet dann darauf, daß dieser bestimmte Kindprozeß beendet wird (denken Sie daran, daß der Elternprozeß die PID des Kindprozesses als Rückgabewert der `fork`-Funktion erhält).

Sowohl `wait` als auch `waitpid` liefern die PID des Kindprozesses zurück, der beendet wurde, oder `-1`, wenn im Moment keine Kindprozesse ausgeführt werden.

Kehren wir noch einmal zu unserem Prozeßbeispiel zurück, das drei Kindprozesse in einer `foreach`-Schleife auslöst. Mit `exit` haben wir das Verhalten der Kindprozesse modifiziert, jetzt wollen wir das Verhalten des Elternprozesses ändern. Dazu fügen wir innerhalb des Elternteils der Bedingung einen Aufruf an `wait` und eine weitere Meldung ein:

```
if ($pid) { #Eltern
  print "Eltern: Kind $i ($pid) gestartet \n";
  wait;
  print "Eltern: Kind $i ($pid) beendet \n";
} else { ....
```

In der letzten Version des Beispiels gab der Eltern-Code lediglich die erste Meldung aus. Dann wiederholte sich die `foreach`-Schleife und löste in rascher Folge drei Kindprozesse aus. In dieser Version wird der Kindprozeß erzeugt, der Elterncode gibt die erste Meldung aus und wartet dann, bis der Kindprozeß zu Ende ist. Dann gibt er die zweite Meldung aus. Der nächste Durchlauf der Schleife erfolgt erst, wenn der aktuelle Kindprozeß abgeschlossen und beendet wurde. Die Ausgabe dieser Skriptversion lautet:

```
% procwait.pl
Eltern: Kind 1 (11876) gestartet
Kind 1: wird ausgeführt
Eltern: Kind 1 (11876) beendet
Eltern: Kind 2 (11877) gestartet
Kind 2: wird ausgeführt
Eltern: Kind 2 (11877) beendet
Eltern: Kind 3 (11878) gestartet
Kind 3: wird ausgeführt
Eltern: Kind 3 (11878) beendet
Ende...
%
```

Auffällig hieran ist, daß die Ausführung sehr regelmäßig ist. Jeder Kindprozeß wird verzweigt, ausgeführt und beendet, bevor der nächste Prozeß startet. Und der Elternprozeß endet erst, nachdem der dritte Kindprozeß abgeschlossen ist.

Wenn aber wie in obigem Beispiel alle Kindprozesse hintereinander ausgeführt werden, stellt sich die Frage, wozu man überhaupt Prozesse einsetzt (zumal jeder Prozeß Zeit benötigt, um in Gang zu kommen, und zusätzliche Prozessorressourcen belegt). Nun, die `wait`-Funktion ist so flexibel, daß man ja nicht unbedingt warten muss, bis der zuletzt ausgelöste Kindprozeß beendet ist, bevor man einen neuen Prozeß startet - Sie können auch fünf Prozesse starten und dann später in Ihrem Skript `wait` fünfmal aufrufen, um alle Prozesse aufzuräumen. Ein Beispiel dazu werde ich Ihnen weiter hinten in dieser Lektion zeigen, wo wir uns an einem größeren Skript versuchen werden.

Eine letzte Funktion, die im Zusammenhang mit der Steuerung von Prozessen noch genannt werden sollte, ist die Funktion `kill`, die ein Kill-Signal an einen Prozeß sendet. Um `kill` zu verwenden, müssen Sie etwas über Signale verstehen. Aus Platzgründen möchte ich hier auf eine Beschreibung der Signale wissen. Sie finden aber einige Hinweise im Vertiefungsabschnitt und der ***perlfunc***-Manpage.

Mit exec andere Programme ausführen

Wenn Sie mit `fork` einen neuen Prozeß starten, erzeugt dieser Prozeß einen Klon des aktuellen Skripts und führt dieses von der aktuellen Position aus fort. Manchmal möchte man jedoch, daß der aktuelle Prozeß mit dem, was er gerade macht, aufhört und statt dessen ein anderes Programm ausführt. Hierfür gibt es die Funktion `exec`.

Die `exec`-Funktion bewirkt, daß der aktuelle Prozeß die Ausführung des aktuellen Skripts abbricht und statt dessen

etwas anderes ausführt. Dieses »etwas anderes« ist in der Regel ein anderes Programm oder Skript, das `exec` als Argument übergegeben wird:

```
exec("grep $who /etc/passwd");
```

Für die Argumente zu `exec` gelten die gleichen Regeln wie bei `system`. Wenn Sie ein einfaches String-Argument verwenden, übergibt Perl dieses Argument zuerst an die Shell. Mit einer Liste von Argumenten können Sie den Shell-Prozeß umgehen. Genau genommen, sind die Ähnlichkeiten zwischen `exec` und `system` kein Zufall - die Funktion `system` ist vielmehr eine Kombination aus `fork` und `exec`.

Sobald Perl auf ein `exec` trifft, bedeutet dies das Ende für das Skript. `exec` überträgt die Kontrolle an das neue auszuführende Programm; von dem alten Skript werden keine weiteren Zeilen ausgeführt.

Weitere Unix-bezogene Funktionen

Zusätzlich zu den bisher in diesem Kapitel beschriebenen Funktionen gibt es in Perl unter den vordefinierten Funktionen noch eine ganze Reihe weiterer prozeßbezogener Funktionen und kleinerer Hilfsfunktionen, die Informationen über verschiedene Teile des Systems liefern. Da sich diese Funktionen ausschließlich auf Unix-Systemdateien und -Elemente beziehen, sind die meisten von ihnen auf anderen Systemen nicht verfügbar (wenn auch die Entwickler, die Perl auf diese Systeme portieren, immer mal wieder versuchen, rudimentäre Entsprechungen für das Verhalten dieser Funktionen zu erzeugen).

Tabelle 18.1 gibt Ihnen einen Überblick über die meisten dieser Funktionen. Weitere Informationen zu den einzelnen Funktionen finden Sie in Anhang A, »Perl- Funktionen«, oder der ***perlfunc***-Manpage.

Funktion	Wirkung
`alarm`	Sendet ein SIGALRM-Signal an einen Prozeß
`chroot`	Ändert das Hauptverzeichnis für den aktuellen Prozeß
`getgrent,`	Sucht Werte von `/etc/groups`
`setgrent,`	Setzt Werte von `/etc/groups`
`endgrent`	Setzt Werte von `/etc/groups`
`getgrgid`	Sucht einen Gruppendateieintrag in `/etc/groups` (gesucht wird nach einer ID)
`getgrnam`	Sucht einen Gruppendateieintrag in `/etc/groups` (gesucht wird nach einem Namen)
`getpgrp`	Ermittelt den Prozeßgruppennamen für einen Prozeß
`getppid`	Ermittelt die Prozeß-ID des Elternprozesses (falls das aktuelle Skript in einem Kindprozeß ausgeführt wird)
`getpriority`	Liefert die aktuelle Priorität für einen Prozeß, eine Prozeßgruppe oder einen Benutzer
`getpwent, setpwent, endpwent`	Sucht/setzt Werte von `/etc/passwd`
`getpwnam`	Sucht einen Benutzer in `/etc/passwd` (gesucht wird nach einem Namen)
`getpwuid`	Sucht einen Benutzer in `/etc/passwd` (gesucht wird nach einer Benutzer-ID (UID))
`setpgrp`	Setzt die Prozeßgruppe für einen Prozeß

Tabelle 18.1: ***Unix-bezogene Funktionen***

Perl für Windows

Perl für Windows unterstützt die meisten der grundlegenden Unix-Features und enthält eine Reihe von

Erweiterungen für Win32. Wenn Sie die ActiveState-Version von Perl für Windows installiert haben, stehen Ihnen die Win32-Module als Teil des Pakets zur Verfügung. Wenn Sie Perl für Windows selbst kompilieren, benötigen Sie das Paket `libwin32` von CPAN (siehe `http://www.perl.com/CPAN-local/modules/ by-module/Win32/` wegen der neuesten Version). Ansonsten ist die Funktionalität die gleiche.

Frühere Versionen von Perl für Windows waren wesentlich unübersichtlicher hinsichtlich der Frage, welche Module und Elemente in welchem Paket zur Verfügung stehen. Ist Ihre Version von Perl für Windows älter als 5.005 (oder die ActiveState-Version von Perl älter als Build 500), sollten Sie Ihre Software aktualisieren, um sicherzustellen, daß Sie die neueste und beste Version haben.

Wenn Sie mit irgendeinem der Aspekte von Perl unter Win32 Probleme haben, gibt es eine Reihe von hilfreichen Quellen. Beginnen Sie mit den von mir bereits erwähnten »Häufig gestellten Fragen« (FAQs) für Perl für Win32 (zu finden unter `http://www.activestate.com/support/faqs/win32/`).

Kompatibilität mit Unix

Bis auf einige wenige Ausnahmen lassen sich die meisten Unix-bezogenen Perl- Features auf Windows übertragen, wobei die Anwendung allerdings meist etwas vom Unix-Original abweicht. Die größte nennenswerte Ausnahmen sind `fork` und seine verwandten Funktionen, die nicht unterstützt werden. Dennoch können Sie mit `system`, `exec`, schrägen Anführungszeichen oder einer der Win32-Erweiterungen ein anderes Programm von einem Perl-Skript aus ausführen (mehr dazu später).

Die `system`-Funktion, `exec` und die schrägen Anführungszeichen funktionieren auch in Perl für Windows. Die »Shell« für diese Befehle ist `cmd.exe` für Windows NT oder `command.com` für Windows 95. Befehle und Argumentlisten für diese Befehle müssen den Windows-Konventionen folgen, einschließlich der Pfadnamen und des Datei- Globbings.

Wenn Sie ein Unix-Skript, das mit Hilfe von `system` oder den schrägen Anführungszeichen Unix-Hilfsprogramme aufruft, nach Windows portieren wollen, genügt es nicht, daß Ihre Perl-Version `system` und schräge Anführungszeichen unterstützt. Sie müssen auch Windows-Entsprechungen für die augerufenen Unix-Hilfsprogramme finden.

Funktionen, die sich auf ganz spezielle Unix-Features beziehen (wie die in Tabelle 18.1) lassen sich mit großer Wahrscheinlichkeit mit Perl für Windows nicht ausführen. Es gibt noch eine Reihe spezialisierter Funktionen, auf deren Beschreibung ich in diesem Buch verzichtet habe und die ebenfalls unter Windows nicht ausgeführt werden können (beispielsweise Funktionen für die Kommunikation zwischen den Prozessen (interprocess communication) oder Low-Level-Netzwerkfunktionen). Allgemein kann man jedoch sagen, daß die meisten Perl-Funktionen, die Sie verwenden, auch in Perl für Windows verfügbar sind. Eine komplette Liste der nicht implementierten Funktionen findet sich in den häufig gestellten Fragen (FAQs) für Perl für Win32 unter `http://www.activestate.com/support/faqs/win32/` (schauen Sie auch im Abschnitt »Implementation Quirks« (Implementierungstricks) nach).

Vordefinierte Win32-Subroutinen

Die Perl-Erweiterungen für Windows bestehen aus zwei Teilen: einem Satz von vordefinierten Win32-Subroutinen und einer Reihe zusätzlicher Win32-Module für fortgeschrittene Techniken (zum Beispiel `Win32::Registry` oder `Win32::Process`). Die Win32-Subroutinen bestehen vornehmlich aus Routinen zur Abfrage von Systeminformationen sowie aus einer Reihe von kleineren Hilfsroutinen. Wenn Sie Perl für Windows laufen haben, müssen Sie zur Verwendung dieser Subroutinen keines der Win32-Module importieren. In Tabelle 18.2 sind einige der Win32- Subroutinen aufgelistet, die in Perl für Windows zur Verfügung stehen.

Weitere Informationen zu diesen Subroutinen finden Sie in den häufig gestellten Fragen zu Perl für Win32. Besonders hilfreich fand ich persönlich auch die Seiten von Philippe Le Berre unter `http://www.inforoute.cgs.fr/leberre1/main.htm`.

Subroutine	Wirkung
`Win32::DomainName`	Liefert die Microsoft-Netzwerk-Domäne zurück
`Win32::FormatMessage errorcode`	Übernimmt den Fehlerstring, der von `GetLastError` zurückgeliefert wurde, und wandelt ihn in einen informativen String um
`Win32::FsType`	Der Typ des Dateisystems (FAT oder NTFS)
`Win32::GetCwd`	Ermittelt das aktuelle Verzeichnis
`Win32::GetLastError`	Ist die letzte Win32-Subroutine fehlgeschlagen, erfahren Sie mit dieser Subroutine den Grund (formatieren Sie das Ergebnis mit `FormatMessage`)
`Win32::GetNextAvailDrive`	Ermittelt den Buchstaben des nächsten Laufwerks, zum Beispiel `E:`
`Win32::GetOSVersion`	Liefert eine Array zurück, das die BS-Version und -Nummer darstellt (`$string`, `$major`, `$minor`, `$build`, `$id`), wobei `$string` ein beliebiger String ist, `$major` und `$minor` sind Versionsnummern, `$build` gibt an, um die wie vielte Kompilation (Build) es sich handelt, und `$id` ist 0 für ein generisches Win32, 1 für Windows 95 oder 2 für Windows NT.
`Win32::GetShortPathName`	Liefert für lange Dateinamen (`diesisteinwirklichlangerdateiname.txt`) die zugehörige 8.3-Version (`diesi~1.txt`)
`Win32::GetTickCount`	Die Anzahl der Ticks (Millisekunden), die verstrichen sind, seitdem Windows gestartet wurde
`Win32::IsWin95`	*Wahr*, wenn Sie mit Windows 95 arbeiten
`Win32::IsWinNT`	*Wahr*, wenn Sie mit Windows NT arbeiten
`Win32::LoginName`	Der Benutzername desjenigen, der das Skript ausführt
`Win32::NodeName`	Der Knotenname der aktuellen Maschine im Microsoft Netzwerk
`Win32::SetCwd newdir`	Wechselt das aktuelle Verzeichnis
`Win32::Sleep milliseconds`	Schläft für die gegebene Anzahl an Millisekunden
`Win32::Spawn`	Löst einen neuen Prozeß aus (siehe auch den folgenden Abschnitt »Win32-Prozesse«)

Tabelle 18.2: Vordefinierte Win32-Subroutinen

Win32::MsgBox

Mit Hilfe der grundlegenden Win32-Subroutinen können Sie von Perl aus auf die grundlegenden Windows-Features und Systeminformationen zugreifen. Durch Installation des `libwin32`-Moduls (oder durch Verwendung der ActiveState-Version von Perl für Windows) erhalten Sie darüber hinaus Zugriff auf Win32-Module, die Ihnen eine Vielzahl fortgeschrittener Windows-Features erschließen[1]. Ein äußerst praktisches Element der Win32-Module ist die Subroutine `Win32::MsgBox`, mit der man einfache modale Dialogfenster von einem Perl-Skript aus aufspringen lassen kann. `Win32::MsgBox` übernimmt bis zu drei Argumente: den Text, der im Dialogfenster angezeigt werden soll, einen Code für das Symbol im Dialogfenster und für die Kombination von Schaltern sowie den Text für die Titelleiste des Dialogfensters. Der folgende Code wird Ihnen das Dialogfenster zur linken von Abbildung 18.1 liefern:

```
Win32::MsgBox("Ich kann das nicht tun!");
```

Abbildung 18.1: Dialoge

Der folgende Code erzeugt ein Dialogfenster mit zwei Schaltern, **OK** und **Abbrechen** und einem Fragezeichensymbol (zu sehen auf der rechten Seite von Abbildung 18.1):

```
Win32::MsgBox("Sind Sie sicher, daß Sie das Objekt löschen wollen?", 33);
```

Das zweite Argument ist der Code für die Anzahl der Schalter und den Typ des Symbols. In Tabelle 18.3 sind die verschiedenen Kombinationsmöglichkeiten für die Schalter zusammengestellt.

Code	Ergebnis
0	`OK`
1	`OK` und `Abbrechen`
2	`Beenden`, `Wiederholen`, `Ignorieren`
3	`Ja`, `Nein` und `Abbrechen`
4	`Ja` und `Nein`
5	`Wiederholen` und `Abbrechen`

Tabelle 18.3: Schaltercodes

Tabelle 18.4 enthält die Codes für die verschiedenen Symbole.

Code	Ergebnis
16	Hand
32	Fragezeichen (?)
48	Ausrufezeichen (!)
64	Sternchen (*)

Tabelle 18.4: Symbolcodes

Das zweite Argument für `Win32::MsgBox` erhalten Sie, indem Sie sich in beiden Tabellen für je eine Option entscheiden und die Codewerte aus beiden Tabellen addieren. So ergibt zum Beispiel das Symbol für Ausrufezeichen (48) zusammen mit den Schaltern *Ja* und *Nein* (4) die Codezahl 52.

Der Rückgabewert von `Win32::MsgBox` hängt von den Schaltern ab, die Sie im Dialogfenster verwendet haben und die der Benutzer angeklickt hat. Tabelle 18.5 zeigt die möglichen Rückgabewerte.

Code	Betätigter Schalter
1	`OK`
2	`Abbrechen`
3	`Beenden`
4	`Wiederholen`
5	`Ignorieren`
6	`Ja`
7	`Nein`

Tabelle 18.5: Schaltercodes

Win32-Prozesse

Perl für Windows enthält keine Unterstützung für `fork`. Diese Funktion basiert zu stark auf Unix-spezifischen Eigenheiten. Es ist jedoch möglich, neue Prozesse zu starten, die andere Programme ausführen (entspricht einem

`fork` gefolgt von einer `exec`- Funktion). Am einfachsten geschieht dies mit Hilfe von `system` oder den schrägen Anführungszeichen oder indem man das aktuelle Skript mit `exec` anhält. Alternativ kann aber auch eines der Module `Win32::Spawn` oder `Win32::Process` eingesetzt werden.

`Win32::Spawn` ist Teil der grundlegenden Subroutinen für Win32 und stellt eine wirklich einfache Möglichkeit dar, einen anderen Prozeß zu starten. Das Modul `Win32::Process` ist neueren Datums, robuster und hält sich an die Modulkonventionen, es ist jedoch auch schwieriger zu verstehen und einzusetzen.

Um mit `Win32::Spawn` einen neuen Prozeß zu erzeugen, benötigen Sie drei Argumente: den vollen Pfadnamen für den Befehl, der in dem neuen Prozeß ausgeführt werden soll, die Argumente für diesen Befehl (einschließlich des Befehlsnamens) und eine Variable, die die Prozeß-ID für den neuen Prozeß enthält. Im folgenden sehen Sie ein Beispiel, das Notepad auf Windows 95 mit einer temporären Datei in `C:\tempfile.txt` startet. Sie fängt auch Fehler ab:

```
my $command = "c:\\windows\\notepad.exe";
my $args = "notepad.exe c:\\tempfile";
my $pid = 0;
Win32::Spawn($command, $args, $pid) || &error();
print "Gestartet! Die neue PID lautet $pid.";
sub error {
    my $errmsg = Win32::FormatMessage(Win32::GetLastError());
    die "Fehler: $errmsg\n";
}
```

Ärgerlich an `Win32::Spawn` ist jedoch, daß der neue Prozeß - in diesem Fall ***Notepad*** - minimiert angezeigt wird, so daß es den Anschein hat, als ob nichts passiert. Außerdem fährt das Perl-Skript in der Ausführung fort, während der neue Prozeß läuft.

Das `Win32::Process`-Modul handhabt Prozesse wesentlich vernünftiger. Dafür ist es jedoch auch wesentlich komplexer und objektorientiert ausgelegt, was bedeutet, daß die Syntax etwas abweicht (da Sie die Grundlagen bereits in Kapitel 13, »Gültigkeitsbereiche, Module und das Importieren von Code«, kennengelernt haben, sollten Sie damit keine Schwierigkeiten haben).

Die Erzeugung eines `Win32::Process`-Objekts ähnelt in etwa der Verwendung von `Win32::Spawn`. Wie bei `Spawn` benötigen Sie einen Befehl und eine Liste von Argumenten, doch ist dies nicht alles. Um ein `Win32::Process`-Objekt zu erzeugen, müssen Sie zuerst sicherstellen, daß Sie `Win32::Process` importiert haben:

```
use Win32::Process;
```

Rufen Sie anschließend die Methode `Create` auf, um Ihr neues Prozeßobjekt zu erzeugen (Sie benötigen eine Variable, die das Objekt aufnimmt):

```
my $command = "c:\\windows\\notepad.exe";
my $args = "notepad.exe c:\\tempfile";
my $process; # Prozess-Objekt
Win32::Process::Create($process,
     $command,
     $args,
     0,
     DETACHED_PROCESS,
     '.') || &error();
```

(Ich habe hier die Definition für die Fehlersubroutine fortgelassen, um Platz zu sparen).

`Win32::Process` erwartet folgende Argumente:

- Eine Variable, die eine Referenz auf das neue Prozeßobjekt enthält
- Den auszuführenden Befehl
- Die Argumente für diesen Befehl
- Ob Handles geerbt werden (wenn Sie nicht wissen, was das bedeutet, geben Sie einfach 0 ein)
- Eine von mehreren Optionen; `DETACHED_PROCESS` ist die geläufigste, `CREATE_ NEW_CONSOLE` ist manchmal auch ganz hilfreich

- Das temporäre Verzeichnis für diesen Prozeß

Wenn Sie Ihren neuen Prozeß auf diese Weise starten, wird Notepad im Vollbildmodus zur Bearbeitung der Datei in `C:\tempfile` gestartet. Das ursprüngliche Perl-Skript wird immer noch ausgeführt.

Damit der Elternprozeß wartet, bis der Kindprozeß zu Ende ausgeführt ist, müssen Sie die `Wait`-Methode verwenden (beachten Sie das große W, denn es ist nicht die `wait`- Funktion von Perl gemeint). Sie müssen `Wait` als eine objektorientierte Methode des Prozeßobjekts aufrufen:

```
$process->Wait(INFINITE);
```

In diesem Fall wartet der Elternprozeß, bis der Kindprozeß beendet ist. Sie können `Wait` aber auch als Argument eine Anzahl an Millisekunden mitgeben, die der Elternprozeß warten soll, bis er mit der Ausführung fortfährt.

Das `Win32::Process`-Modul verfügt neben `Wait` auch noch über folgende Methoden:

- `Kill`, um den neuen Prozeß zu beenden
- `Suspend`, um den Prozeß vorübergehend anzuhalten
- `Resume`, um einen mit Suspend angehaltenen Prozeß wieder aufzunehmen
- `GetPriorityClass` und `SetPriorityClass`, um die Priorität eines Prozesses zu ermitteln und zu ändern
- `GetExitCode`, um herauszufinden, warum ein Prozeß abgebrochen wurde

Weitere Informationen über `Win32:Process` finden Sie in der Dokumentation, die den Win32-Modulen beiliegt, oder in der Online-Dokumentation unter `http:// www.activestate.com/activeperl/docs`.

Mit der Win32-Registrierdatenbank arbeiten

Die Windows-Registrierdatenbank ist ein großes Archiv mit Informationen über Ihr System, seine Konfiguration und die darauf installierten Programme. Mit dem objektorientierten Modul `Win32::Registry` können Sie von einem Perl-Skript aus in dieser Windows-Registrierdatenbank Werte auslesen, ändern und hinzufügen.

> *Wenn Sie nicht mit der Windows-Registrierdatenbank vertraut sind, sollten Sie von Ihren Perl-Skripts aus nicht damit herumspielen. Sie können Ihr System lahmlegen, wenn Sie in der Registrierdatenbank etwas ändern, was nicht geändert werden darf. Sie können auch das Windows-Programm* `regedit` *dazu verwenden, um die Windows-Registrierdatenbank einzusehen und zu ändern.*

Die Windows-Registrierdatenbank besteht aus einer baumartigen Hierarchie von Schlüsseln und Werten. Auf oberster Ebene enthält die Registrierdatenbank mehrere Teilbäume, beispielsweise `HKEY_LOCAL_MACHINE` für Informationen über die Konfiguration des lokalen Rechners oder `HKEY_CURRENT_USER` für Informationen über den gerade eingeloggten Benutzer. Welche Teilbäume im Detail vorhanden sind, hängt davon ab, ob Sie Windows NT oder Windows 95 verwenden. Unter jedem Teilbaum gibt es eine Vielzahl von Schlüssel/Werte-Sätzen - ähnlich wie bei einem Hash, nur daß diese auch verschachtelt sein können (ein Schlüssel kann auf einen ganzen anderen Hash-Baum verweisen).

Wenn Sie das Modul `Win32::Registry` importieren (mit `use Win32::Registry`), erhalten Sie für jeden obersten Teilbaum ein Schlüsselobjekt, zum Beispiel `$HKEY_LOCAL_MACHINE`. Mit Hilfe der verschiedenen Methoden von `Win32::Registry` können Sie jeden Teil der Windows-Registrierdatenbank öffnen, sichten und verwalten.

Um das `Win32::Registry`-Modul optimal nutzen und die verschachtelte Hash-Struktur der Registrierdatenbank-Schlüssel bearbeiten zu können, sind Kenntnisse über Referenzen unerläßlich. In Tabelle 18.6 finden sie einige der `Win32::Registry`- Methoden und deren Argumente. Wenn Sie das Kapitel zu den Referenzen gelesen haben, werden Sie diese Methoden besser verstehen.

Bevor Sie mit irgendeinem Teil der Registrierdatenbank arbeiten können, müssen Sie zuerst die Methode `Open` für eines der obersten Unterschlüsselobjekte aufrufen (vergessen Sie nicht, `use Win32::Registry` oben in Ihr Skript

aufzunehmen):

```
use Win32::Registry;
my $reg = "SOFTWARE";
my ($regobj, @keys);
$HKEY_LOCAL_MACHINE->Open($reg,$regobj)||
                          die "Registrierung kann nicht geöffnet werden\n";
```

Dann können Sie die verschiedenen Methoden der Registrierdatenbank für das neue `Registry`-Objekt aufrufen:

```
$regobj->GetKeys(\@keys);
$regobj->Close();
```

Methode	Wirkung
`Close`	Schließt den aktuell geöffneten Schlüssel
`Create schlüsselname, schlüsselref`	Erzeugt einen neuen Schlüssel mit dem Namen ***schlüsselname***. ***schlüsselref*** enthält eine Referenz auf den neuen Schlüssel
`DeleteKey schlüsselname`	Löscht den Schlüssel ***schlüsselname***
`DeleteValue wertname`	Löscht den Wert ***wertname***
`GetKeys listref`	Liefert eine Liste der Schlüssel in dem aktuellen Schlüssel. ***listref*** ist eine Referenz auf eine Liste
`GetValues hashref`	Liefert einen Hash der Schlüssel und Werte in dem aktuellen Schlüssel zurück. Die Schlüssel in diesem Hash sind verschachtelte Listen. ***hashref*** ist eine Referenz auf einen Hash
`Open obj, objref`	Öffnet einen Schlüssel, ***obj*** ist der zu öffnende Schlüssel, ***objref*** ist eine Referenz auf das Objekt, das diesen Schlüssel hält
`Save dateiname`	Speichert den aktuell offenen Schlüssel in ***dateiname***
`SetValue schlüsselname, REG_SZ, value`	Ändert den Wert von ***schlüsselname*** in ***value***. Das zweite Argument muss ***REG_SZ*** lauten
`Load schlüsselname, dateiname`	Importiert die Schlüssel und Werte aus ***dateiname*** in ***schlüsselname***

Tabelle 18.6: Win32:-Registrierdatenbank-Methoden

Für weitere Informationen über `Win32:Process` ist die Dokumentation, die den Win32-Modulen beiliegt, sehr hilfreich. Empfehlen kann ich auch die Seiten von Philippe Le Berre unter `http://www.inforoute.cgs.fr/leberre1/main.htm`. in denen viele Beispiele und Hinweise auf die Verwendung von `Win32::Registry` zu finden sind.

Weitere Win32-Module

Die Win32-Module, die mit der ActiveState-Version von Perl ausgeliefert werden und in `libwin32` zusammengefaßt sind, umfassen eine Unzahl von Modulen zur Bewältigung der verschiedenen Aspekte von Windows-Operationen. Mein kurzer Exkurs in `Win32::Process` und `Win32::Registry` hat dabei nur die Oberfläche gestreift. Und zusätzlich zu diesen »Standard«-Modulen für Win32 wurden und werden immer noch weitere Module geschrieben, die unter CPAN zur Verfügung gestellt werden. Wenn Sie viel mit Windows arbeiten und beabsichtigen, verstärkt Perl auf dieser Plattform zu nutzen, sollten Sie ein Blick in diese Module werfen und sich einen Überblick über die vielen dahinter verborgenen Möglichkeiten verschaffen. In Tabelle 18.7 finden Sie eine Zusammenfassung der Standard-Win32-Module. Weitere Module finden Sie im CPAN.

Modul	Wirkung
`Win32::ChangeNotify`	Bietet Zugriff auf die `ChangeNotify`-Objekte von Windows
`Win32::EventLog`	Bietet Zugriff auf die Windows-NT-Ereignisaufzeichnung (nur Windows NT)

Win32::File	Zur Verwaltung von Datei-Attributen
Win32::FileSecurity	Zur Verwaltung der NTFS-Dateisicherheitseinstellungen
Win32::IPC	Inter-Process Communication: ermöglicht die Synchronisierung und Kommunikation zwischen Prozeß, ChangeNotify, Semaphore und Mutex
Win32::Mutex	Bietet Zugriff auf die Mutex-Objekte von Windows
Win32::NetAdmin	Verwaltet Benutzer und Gruppen (nur Windows NT)
Win32::NetResource	Verwaltet Systemressourcen (Drucker, Server etc.) (nur Windows NT)
Win32::Process	Erzeugt und verwaltet Windows-Prozesse
Win32::OLE	Bietet Zugriff auf die OLE-Automation
Win32::Registry	Arbeitet mit der Windows-Registrierdatenbank
Win32::Semaphore	Bietet Zugriff auf die Semaphore-Objekte von Windows
Win32::Service	Ermöglicht Ihnen, Dienste auszuüben
Win32::WinError	Zur Behandlung von Windows-definierten (oder erzeugten) Fehlern

Tabelle 18.7: Win32-Module

MacPerl-Elemente

Viele Perl-Elemente sind auch in MacPerl verfügbar, vor allem wenn Sie die MPW (Macintosh Programmer's Workbench) installiert haben. Darüber hinaus bietet MacPerl Schnittstellen zur Macintosh Toolbox, so daß Sie über Perl Zugriff auf die MacOS-Features haben - vorausgesetzt Sie sind bereits ein wenig mit der sehr komplexen Macintosh Toolbox vertraut. Aber auch wenn Sie nicht allzu tief in die Mac-Programmierung mit Perl einsteigen wollen, bietet Ihnen MacPerl diverse, leicht zu erzeugende und einzusetzende Optionen (beispielsweise die Verwendung von Dialogen), mit denen Sie Ihren Skripts eine Mac-typische Benutzeroberfläche verleihen können.

Für Ihre tägliche Arbeit mit MacPerl finden Sie - neben den Tipps aus den häufig gestellten Fragen zu Standard-Perl - eine Fülle an hilfreichen Informationen in den »Häufig gestellten Fragen« zu MacPerl unter `http://www.perl.com/CPAN/doc/FAQs/ mac/MacPerlFAQ.html`. Außerdem gibt es eine Mailing-Liste für Perl-Benutzer und - Entwickler, die an noch mehr Informationen über MacPerl selbst interessiert sind. In den FAQs können Sie nachlesen, wie man sich auf die Mailing-Liste setzen läßt.

Kompatibilität mit Unix

Ebenso wie ich es für die Windows-Versionen von Perl getan habe, so habe ich mich im ganzen Buch auch bemüht, Sie auf Unterschiede im Verhalten von MacPerl gegenüber Unix-Perl hinzuweisen. Und wie bei Windows, gibt es in der Unix-Version von Perl eine Reihe von Elementen, die nur schwer auf dem Mac nachzuvollziehen sind, besonders in der Einzelrechnerversion von MacPerl. Folgender Kompatibilitätsprobleme sollten Sie sich bewußt sein:

- Perl-Elemente, die externe Programme ausführen (`system`, `exec`, schräge Anführungszeichen), funktionieren nicht in MacPerl. Wenn Sie den MPW ToolServer installiert haben, lassen sich Elemente wie schräge Anführungszeichen, Dateinamen-Globbing und die `system`-Funktion mit Mac- basierten Befehlen ausführen. Perl kann außerdem AppleScript aufrufen, das seinerseits externe Programme ausführen kann.
- Die Einzelrechner-Version von MacPerl unterstützt einige grundlegende Einsatzbereiche der schrägen Anführungszeichen, beispielsweise `` `pwd` `` oder `` `Directory` `` für das aktuelle Verzeichnis und `` `stty -raw` `` und `` `stty -cooked` `` für den Umgang mit reinen Eingaben von der Tastatur.
- MacPerl bietet keine Unterstützung für `fork` oder Prozesse à la Unix. Mit der MacToolbox haben Sie Zugriff auf Macintosh-Prozesse über das `Mac::Processes`- Modul. Doch dazu bedarf es einiger Kenntnisse der Low-Level-Macintosh- Prozesse.
- Das Konzept der Umgebungsvariablen ist nicht wirklich nützlich in MacPerl und wird nur aus Kompatibilitätsgründen unterstützt. Der Hash `%ENV` existiert und enthält standardmäßig einige wenige Basiswerte (wie in den MacPerl Preferences definiert) sowie Angaben über die Position von MacPerl und

seinen Bibliotheken.
- Alle Datums- und Zeitangaben beginnen mit dem 1.1.1904.
- Alle Pfadnamen verwenden den Doppelpunkt (`:`) als Verzeichnistrennzeichen sowie `:` und `::` als Äquivalent zu `.` und `...`.
- Funktionen, die sich ausschließlich auf Unix-Konzepte beziehen (siehe zum Beispiel Tabelle 18.1) lassen sich nicht in MacPerl anwenden. Eine Liste finden Sie in den häufig gestellten Fragen (FAQs).

Dialogfenster

MacPerl verfügt über eine Reihe von einfachen Subroutinen zum Erzeugen und Verwalten einfacher Dialogfenster von einem Perl-Skript aus. Mit diesen Subroutinen können Sie Dialogfenster mit bis zu drei Schaltflächen, ein Textfeld zur Aufnahme von Eingaben, ein Dialogfenster mit mehreren Optionen oder die Standarddialoge zum Öffnen und Speichern von Dateien erzeugen.

Um ein Dialogfenster mit einer oder mehreren Schaltflächen zu erzeugen, verwenden Sie `MacPerl::Answer` mit bis zu vier Argumenten (je nach der Anzahl der gewünschten Schaltflächen). Das erste Argument ist der Text im Dialogfenster, alle darauf folgenden Argumente geben die Titel der Schaltflächen an. Maximal sind drei Schaltflächen erlaubt.

```
MacPerl::Answer("Das ist keine Zahl.");              # Standardschalter (OK)
MacPerl::Answer("Das ist keine Zahl.", "Mist");              # ein  Schalter
MacPerl::Answer("Wollen Sie das Programm wirklich verlassen?",
                "OK", "Abbruch");                            # zwei Schalter
MacPerl::Answer("Wohin?", "Links", "Rechts", "Hoch");
```

In Abbildung 18.2 sehen Sie, wie die hier beschriebenen vier Dialogfenster aussehen. Die Rückgabewerte von `MacPerl::Answer` sind:

- ein leerer String für Dialogfenster mit einem einzigen OK-Schalter,
- 0 für Dialogfenster mit einem Schalter und
- 0, 1 oder 2 für Dialogfenster mit mehr als einem Schalter, je nachdem welcher Schalter ausgewählt wurde (die Schalter werden gemäß ihrer Reihenfolge in der Liste der Argumente durchnumeriert).

Um ein Dialogfenster mit einem Eingabefeld darin zu erzeugen, verwenden Sie `MacPerl::Ask` mit einer Eingabeaufforderung:

```
$age = MacPerl::Ask("Geben Sie bitte Ihr Alter an: ");
```

Sie können für das Eingabefeld auch einen Wert vorgeben:

```
$url = MacPerl::Ask("Anzuwählender URL: ", "http://");
```

Abbildung 18.2: Dialogfenster in MacPerl

`MacPerl::Ask` liefert den vom Benutzer eingegebenen Wert zurück oder `undef`, wenn der Dialog ohne Eingabe abgebrochen wurde. Abbildung 18.3 zeigt ein Beispiel:

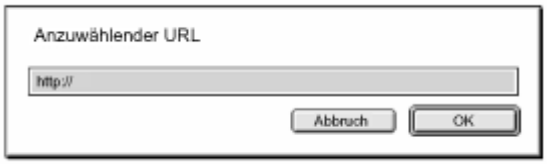

Abbildung 18.3: Eingabefelder in MacPerl

Wollen Sie Ihrem Benutzer mehrere Optionen zur Auswahl anbieten? Dann verwenden Sie `MacPerl::Pick` mit einer Eingabeaufforderung und einer Liste der Optionen:

```
$food = MacPerl::Pick("Was möchten Sie essen: ",
   "Pad Thai", "Burrito", "Pizza");
```

`MacPerl::Pick` liefert als Wert die gewählte Option zurück oder `undef`, wenn der Dialog abgebrochen wurde. In Abbildung 18.4 sehen Sie dieses Dialogfenster:

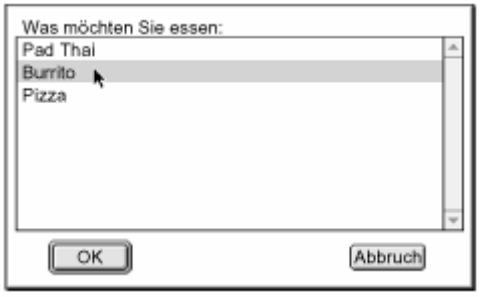

Abbildung 18.4: Dialogfelder mit Optionen in MacPerl

Wenn Sie die Standarddateidialoge nutzen wollen (die Dialogfenster, in denen Sie eine Datei aus dem Mac-Dateisystem auswählen können), sollten Sie die Datei `StandardFile.pl` in Ihre Skripts aufnehmen. Diese Datei stellt Ihnen eine einfach zu verwendende Schnittstelle zu der Subroutine `MacPerl::Choose` zur Verfügung. Verwenden Sie `require` zur Einbindung des `StandardFile`-Codes:

```
require 'StandardFile.pl:
```

Mit dieser Datei erhalten Sie Zugriff auf die folgenden vier Subroutinen:

- `StandardFile::GetFile`, um den Pfad für eine existierende Datei auszuwählen
- `StandardFile::GetNewFile` und `StandardFile::PutFile`, um den Pfad und den Dateinamen einer neuen Datei zu erhalten
- `StandardFile::GetNewFolder`, um den Pfad zu einem Ordner zu erhalten

Die Subroutine `GetFile` übernimmt drei Argumente: eine Eingabeaufforderung, eine Liste von Dateitypen, die herausgefiltert werden sollen (verwenden Sie für Textdateien beispielsweise `'TEXT'`, `'ttxt'`, `'ttro'`) und einen Standarddateinamen. Die Eingabeaufforderung und der Standarddateiname sind optional (die Eingabeaufforderung wird in aktuellen MacPerl-Versionen überhaupt nicht angezeigt):

```
$filename = &StandardFile::GetFile("Wählen Sie eine Datei aus",
                                   'TEXT','ttxt','ttro', "eingabe");
```

`GetNewFile`, `PutFile` und `GetNewFolder` erwarten zwei Argumente: eine Eingabeaufforderung und einen Standarddateinamen. Wie bei `GetFile` wird die Eingabeaufforderung derzeit ignoriert, ist jedoch als Argument obligatorisch. Der Standarddateiname ist optional:

```
$filename = &StandardFile::PutFile("Wählen Sie eine Protokolldatei aus",
                                   "LOG_Datei");
```

All diese Subroutinen liefern den vollen Macintosh-Pfadnamen für die ausgewählte Datei, und zwar mit »:« zwischen den Ordnernamen (zum Beispiel `Hard Disk:MyPerlApp:Stored Files:logdatei.txt`). Anschließend müssen Sie die E/A- Routinen von Perl verwenden, um die Datei zu lesen oder in die Datei zu schreiben.

Beachten Sie, daß es allgemein erwartet wird, daß Sie zum Lesen einer Datei die Subroutine `GetFile` verwenden, während die Subroutinen `GetNewFile` und `PutFile` sich eher zum Auswählen von Dateien für die Ausgabe anbieten. Durch die Wahl des falschen Dialogfensters könnten Sie Ihre Benutzer verwirren, die bestimmte Erwartungen bezüglich des Ablaufs hegen.

Auf der Begleit-CD zu diesem Buch (wie auch unter `www.typerl.com`) finden Sie ein Perl-Skript, das Beispiele für alle diese Dialogfenster erzeugt. Es heißt `macperldialoge.pl`.

Weitere MacPerl-Elemente

Erfahrenen Mac-Programmierern stellt MacPerl eine Reihe von Schnittstellen zu verschiedenen Teilen des Mac-Betriebssystems zur Verfügung. Dazu gehören unter anderem:

- Zugriff auf die Mac Toolbox über die Mac-Toolbox-Module
- Die Möglichkeit HyperCard XCMDs und XFCNs zu erzeugen und Verknüpfungen zu diesen herzustellen
- Das Ausführen von AppleScript-Skripts

In den MacPerl-Hilfedateien finden Sie Informationen hierzu. Vergessen Sie auch nicht das CPAN, das einige Mac-spezifische Module enthält, und die bereits erwähnten häufig gestellten Fragen (FAQs), die Hilfestellung bei vielen allgemeinen Problemen mit MacPerl bieten.

Vertiefung

Ganze Bücher sind darüber geschrieben worden, wie man die verschiedenen plattformspezifischen Aspekte von Perl nutzt (oder vermeidet). Um dieses Kapitel so klein wie möglich zu halten, habe ich etliche Elemente ausgelassen. Einige dieser Elemente, von denen ich annehme, daß Sie sich mit ihnen nach Lektüre dieses Buches noch eingehender beschäftigen möchten, werde ich in diesem Abschnitt kurz vorstellen.

Pipes

Das wahrscheinlich wichtigste Element, das ich von der Besprechung ausgespart habe, sind die Pipes. Eine Pipe ist eine Art Kanal, aus dem Sie Daten auslesen und an den Sie Daten senden können. Die Pipe kann mit der Standardeingabe und -ausgabe Ihres Skripts und einem anderen Programm verbunden werden. Sie kann aber auch mit einem Gerät wie einem Drucker oder einer Netzwerkverbindung wie einem Socket verbunden werden.

Unter Unix können Sie eine Pipe wie ein Datei-Handle öffnen, und genauso können Sie auch daraus lesen oder dorthin schreiben. Eine Pipe kann eine Verbindung zu einem anderen Programm oder zu einem anderen Prozeß, der das gleiche Programm ausführt, herstellen. Sie können auch benannte Pipes verwenden, die auf Ihrem System existieren. Die ***perlipc***-Manpage enthält weitere Informationen zur Verwendung von Pipes.

Unter Windows können Sie mit `open` - wie unter Unix - reguläre Pipes zu anderen Prozessen, die auf dem System ausgeführt werden, herstellen. Für benannte Pipes bedienen Sie sich des Moduls `Win32::Pipe` aus dem CPAN.

Mac-Anwender, die den ToolServer installiert haben, können eine eingeschränkte Version der Pipes verwenden.

Signale

Signale sind ein Unix-Konzept, mit dem man verschiedene Fehler und Meldungen abfangen und einer ordnungsgemäßen Behandlung zuführen kann. Der Hash `%SIG` enthält Referenzen auf verschiedene Signal-Bearbeitungsroutinen. Signale funktionieren nur unter Unix. Weitere Informationen finden Sie in der ***perlipc***-Manpage.

Netzwerkprogrammierung

Die Unix-Version von Perl verfügt über eine Reihe von vordefinierten Funktionen zur Bearbeitung von Low-Level-Netzwerk-Befehlen und -Sockets. Die meisten dieser Funktionen finden auf anderen Plattformen keine Unterstützung, sondern sind vielmehr ersetzt durch diverse plattformspezifische Netzwerk-Module. Wenn Sie ein wahrer Netzwerk-Fan sind, dann möchte ich Ihnen die ***perlipc***-Manpage ans Herz legen. Dort finden Sie einen wahren Schatz an Informationen zu Sockets und Netzwerkprogrammierung.

Wenn Sie aber lediglich, sagen wir, eine Webseite vom Internet herunterladen wollen, gibt es Module dafür, die Ihnen diese Arbeit abnehmen, ohne daß Sie irgend etwas über TCP wissen müssen. Darauf werde ich in Kapitel 20 noch näher eingehen.

Benutzerschnittstellen mit Perl erzeugen

Eine grafische Benutzerschnittstelle (GUI) in Perl zu erzeugen, ist nicht leicht und trägt nicht gerade zur Plattformunabhängigkeit bei. Aber es ist möglich.

Einer der besten Wege zur Erzeugung von GUIs in Perl führt über das Tk-Paket. Tk ist ein einfacher Weg zum Erzeugen und Verwalten von Benutzerschnittstellen-Fenstern, ursprünglich verbunden mit der TCL-Sprache, jetzt aber auch für Perl verfügbar. Tk gibt es für Unix und Windows in jeweils plattformspezifischer Ausführung. Zu Perl/Tk gibt es eine Datei mit häufig gestellten Fragen unter `http://w4.lns.cornell.edu/~pvhp/ptk/ptkTOC.html`.

Das Modul `Win32::OLE` ermöglicht es Ihnen, Schnittstellen in Visual Basic zu erstellen und sie dann via OLE-Automation zu steuern. Siehe dazu `Win32::OLE`.

Auf dem Mac steht Ihnen so ziemlich alles, was Sie für die Erstellung von Mac- Schnittstellen unter Perl benötigen, zur Verfügung. Voraussetzung ist aber, daß Sie wissen, wie man die zur Verfügung stehenden Möglichkeiten nutzt, und das bedeutet, daß Sie zumindest rudimentäre Kenntnisse der mehr als zehn Bände von »Inside Macintosh« haben sollten.

Zusammenfassung

Eines Tages wird Perl eine plattformunabhängige Sprache aus einem Guß sein, in der Sie Skripts schreiben können, die jede beliebige Aufgabe meistern und die sich auf jeder Plattform ausführen lassen, für die es eine Perl-Umgebung gibt. Aber noch ist es nicht soweit. Und manche mögen der Meinung sein, daß das Ziel einer absoluten plattformübergreifenden Kompatibilität gar nicht so erstrebenswert ist, in Anbetracht all der Unterschiede zwischen den Plattformen und all der coolen plattformspezifischen Dinge, die man machen kann.

Heute haben wir aufgehört, Perl als eine Sprache zu betrachten, die auf allen Plattformen die gleiche ist, sondern ein paar der plattformspezifischen Unterschiede untersucht: die Umgebungsvariablen, das Ausführen von Programmen, das Starten von Prozessen unter Unix und Windows, die Arbeit mit der Windows-Registrierdatenbank sowie das Erzeugen von Dialogfenstern und Eingabeaufforderungen unter MacPerl.

Fragen und Antworten

Frage:
Ich habe ein Skript, das mit Hilfe des Befehls `system` mehrere Programme aufruft. Ich möchte dieses Skript über einen Unix-`cron`-Job ausführen. Wenn ich es über die Befehlszeile ausführe, gibt es keine Probleme, aber sobald ich es installiert habe, erhalte ich Fehlermeldungen, daß dieser und jener Befehl nicht gefunden werden kann. Was habe ich falsch gemacht?

Antwort:
Wahrscheinlich ein Problem des Ausführungspfades. Denken Sie daran, daß jedes Perl-Skript seinen Ausführungspfad (das heißt die Umgebungsvariable `PATH`) von der Shell oder der Benutzer-ID, die gerade das Skript ausführt, erbt. Wenn ein Skript von einem anderen Prozeß ausgeführt werden soll - zum Beispiel einem CGI-Skript oder einem `cron`-Job -, erhält es einen anderen Ausführungspfad. In diesem Fall hat die `cron`-Benutzer-ID wahrscheinlich einen sehr eingeschränkten Ausführungspfad, und die `system`- Funktion findet das von ihr

benötigte Programm nicht. Um dieses Problem zu vermeiden, können Sie für alle ausführbaren Programme, die von `system` *aufgerufen werden, immer volle Pfadnamen angeben oder aber die* `PATH`*- Variable auf dem Weg über den* `%ENV`*-Hash selbst innerhalb des Perl-Skripts setzen.*

Frage:
Ich habe ein Skript, das mit `fork` in mehrere Prozesse verzweigt. Ich habe eine einzige globale Variable und möchte diese Variable global in allen Kindprozessen inkrementieren. Leider funktioniert das nicht. Warum?

Antwort:
Alle Prozesse, die mit `fork` *erzeugt werden, sind völlig unabhängig von den anderen Prozessen. Dazu gehören auch alle Variablen, die von dem Elternprozess definiert wurden. Der Kindprozeß erhält eine Kopie all dieser Variablen und hat keinen Zugriff auf diejenigen des Elternprozesses. Um zwischen den Prozessen zu kommunizieren, müssen Sie einen speziellen IPC- Mechanismus (IPC steht für »Interprocess Communication«) einrichten. Im Vertiefungsabschnitt finden Sie ein paar Hinweise in Richtung Interprozeß- Kommunikation.*

Frage:
Wenn ich in Kindprozesse verzweige, wird die Ausgabe der Kindprozesse mit der Ausgabe des Elternprozesses vermischt. Wie kann ich die Ausgaben trennen?

Antwort:
Das ist nicht möglich. Es gibt nur eine Standardausgabe, und diese Ausgabe wird zwischen allen Kindprozessen geteilt. Wenn Sie wirklich die Ausgaben der einzelnen Prozesse trennen müssen, könnten Sie beispielsweise die Ausgabe der Prozesse in unterschiedliche temporäre Dateien ablegen und dann diese Dateien nacheinander ausgeben.

Frage:
Ich habe ein Perl-Skript, das unter Unix geschrieben wurde. Durch das ganze Skript hindurch werden die Funktion `system` und das Unix-Programm `sendmail` verwendet. Wie kann ich dieses Programm so anpassen, daß es andere Systeme berücksichtigt?

Antwort:
Post zu senden ist unter Unix sehr bequem. Sie müssen nur das Mail- Programm aufrufen und die Meldung fortschicken. Leider ist das auf anderen Plattformen nicht ganz so einfach. Wenn Sie unter Windows arbeiten, finden Sie in den häufig gestellten Fragen zu Perl für Win32 eine Liste der Alternativen zum Versenden von Post. Auch das CPAN-Paket `Net::SMTP` *kann bei der Implementierung plattformunabhängiger Mail-Operationen helfen.*

Frage:
Ich möchte herausfinden, auf welcher Plattform ich mich befinde, so daß ich sicher bin, daß mein Skript nur auf einer Plattform ausgeführt wird.

Antwort:
Hier kann Ihnen das Modul `Config` *helfen. Der Schlüssel* `'osname'` *aus dem Hash* `%Config` *enthält die Plattform, auf der Sie sich befinden. Um beispielsweise sicherzustellen, daß Sie unter Windows arbeiten, könnten Sie folgenden Code verwenden:*

```
    use Config;
if ( $Config{'osname'} !~ /Win/i ) {
   die "Vorsicht!  Dieses Skript läuft nur unter Windows.  Sie können es nicht von hier ausführ
```

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Wofür wird `%ENV` verwendet? Warum ist es nützlich?

2. Was ist der Unterschied zwischen `system` und den schrägen Anführungszeichen?
3. Warum könnte es sinnvoll sein, mehrere Prozesse in einem Perl-Skript zu verwenden?
4. Was macht `fork`?
5. Was ist der Unterschied zwischen `system` und `exec`?
6. Kann man `Win32::Process` im Austausch mit `fork` verwenden?

Übungen

1. (Nur Unix) Schreiben Sie ein Skript, das eine einzige Zahl als Argument übernimmt. Starten Sie mit `fork` die gleiche Zahl Kindprozesse, und stellen Sie sicher, daß der Elternprozeß wartet, bis alle Kindprozesse beendet sind. Jeder Prozeß sollte

- eine Zufallszahl zwischen 1 und 100.000 erzeugen,
- alle Zahlen zwischen 1 und der Zufallszahl addieren und
- das Ergebnis ausgeben.

2. (Nur Unix) Modifizieren Sie das Skript `img.pl` aus Kapitel 10, »Erweiterte Möglichkeiten regulärer Ausdrücke« (das Skript, das Informationen über die Grafiken in einer HTML-Datei ausgab), so daß die Ausgabe Ihnen per E-Mail zugesandt und nicht auf dem Bildschirm ausgegeben wird. HINWEIS: Mit dem folgenden Befehl können Sie eine Nachricht senden:

```
mail ihremail@ihresite.com < textderNachricht
```

3. (Nur Windows) Schreiben Sie ein Skript, das eine Verzeichnisliste übernimmt (mit dem Befehl `dir`) und nur die Dateinamen, einen Namen pro Zeile, ausgibt (Sie müssen die Namen nicht sortieren).
4. (Nur Mac) Schreiben Sie ein Skript, das einen Standarddateidialog verwendet, in dem Sie eine Textdatei auswählen können (es sollen überhaupt nur Textdateien zur Auswahl angeboten werden). Danach soll das Skript diese Datei öffnen und ihren Inhalt auf die Konsole ausgeben.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Der Hash `%ENV` enthält die Variablennamen und -werte aus der Umgebung des Skripts. Unter Unix und Windows können die Werte in dem Hash nützlich sein, um Informationen über die Systemumgebung einzuholen (oder zu ändern und an andere Prozesse zu übergeben). Auf dem Mac erfüllt `%ENV` keinen besonderen Zweck.
2. Der Befehl `system` führt ein anderes Programm oder Skript von innerhalb Ihres Perl-Skripts aus und sendet die Ausgabe an die Standardausgabe. Schräge Anführungszeichen führen ebenfalls externe Programme aus, fangen aber die Eingabe in einem Skalar oder in einer Liste ab (je nach Kontext).
3. Mehrere Prozesse sind nützlich, um Ihr Skript in mehrere Teile aufzuspalten, die eventuell gleichzeitig ausgeführt werden müssen, oder um die Arbeit, die Ihr Skript bewältigen muss, aufzuteilen.
4. Die `fork`-Funktion erzeugt einen neuen Prozeß, einen Klon des Originalprozesses. Beide Prozesse führen das Skript von dem Punkt, an dem `fork` auftrat, weiter aus.
5. `system` und `exec` sind eng miteinander verwandt. Beide dienen dazu, ein externes Programm auszuführen. Der Unterschied besteht darin, dass `system` zuerst die `fork`-Funktion ausführt, während `exec` die Ausführung des aktuellen Skripts im aktuellen Prozeß unterbricht und statt dessen das externe Programm ausführt.
6. `Win32::Process` und `fork` sind nicht austauschbar. Die `fork`-Funktion ist eng an die Unix-Vorstellung von einem Prozeß gekoppelt; `Win32::Process` entspricht eher einem `fork`, auf das direkt ein `exec` folgt.

Antworten zu den Übungen

1. Eine mögliche Antwort:

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

```
use strict;
if (@ARGV > 1) {
    die "Bitte nur ein Argument.\n";
}
 elsif ($ARGV[0] !~ /^\d+/) {
    die "Nur Zahlen als Argument zugelassen.\n";
}
my $pid;
my $procs = pop;
foreach my $i (1..$procs) {
    if (defined($pid = fork)) {
        if ($pid) { #Eltern
            print "Eltern: Kind $i gestartet\n";
        } else {    #Kind
            srand;
            my $top = int(rand 100000);
            my $sum;
            for (1..$top) {
                $sum += $_;
            }
            print "Kind $i beendet:  Summe der ersten $top Zahlen: $sum\n";
            exit;
        }
    }
}
while ($procs > 0) {
    wait;
    $procs--;
}
```

2. Drei Änderungen sind erforderlich. Zuerst erzeugen und öffnen Sie eine temporäre Datei:

```
    my $tmp = "tempfile.$$";                # temporäre Datei;
  open(TMP, ">$tmp") ||
                die "Temporäre Datei $tmp kann nicht geöffnet werden\n";
```

1. Zweitens, stellen Sie sicher, dass alle `print`-Anweisungen in diese Datei schreiben:

```
    if (exists($atts{$key})) {
   $atts{$key} =~ s/[\s]*\n/ /g;
   print TMP "   $key: $atts{$key}\n";
}
```

1. Und drittens, verwenden Sie `use`, um die temporäre Datei zu versenden und zu entfernen:

```
    my $me = "ihreEmail@ihreSite.com";
  system("mail $me <$tmp");
unlink $tmp
```

3. Hier ist eine mögliche Lösung (starten Sie die `substr`-Funktion mit dem Zeichen 44, wenn Sie mit Windows 95 arbeiten):

```
    #!/usr/bin/perl -w
use strict;
my @list = `dir`;
foreach (@list) {
   if (/^\w/) {
      print substr($_,39);
   }
}
```

4. Hier ist eine Möglichkeit:

```
    #!/usr/bin/perl -w
use strict;
require 'StandardFile.pl';
my $file = &StandardFile::GetFile("Wählen Sie eine Datei",
                                  'TEXT','ttxt','ttro', "input");
```

```
open(FILE, "$file") || die "Datei $file kann nicht geöffnet werden\n";
while (<FILE>) { print };
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Mit Referenzen arbeiten

In den letzten Kapiteln haben wir einige Aspekte von Perl betrachtet, die nicht unbedingt den Kern der Sprache selbst betreffen, sondern eher sekundärer Natur sind. So haben wir gelernt, wie man mit Hilfe bestimmter Funktionen der Standardbibliothek auf das Dateisystem zugreift, wie man Prozesse erzeugt, wie man Code aus Modulen importiert und mit diesen Modulen die verschiedensten Aufgaben löst oder wie man den Perl-Debugger nutzt. Heute wollen wir, da wir uns inzwischen dem Ende des Buches nähern, wieder dem Kern der Sprache zuwenden und auf die Referenzen zu sprechen kommen. Referenzen stellen eine Möglichkeit dar, indirekt auf andere Daten in Perl zu verweisen. Mit Referenzen kann man Daten oftmals besser und effizienter verwalten, als wenn man die Daten direkt manipulieren würde. In der heutigen Lektion widmen wir uns den folgenden Themen:

- Was sind Referenzen und welche Vorteile bieten sie Ihnen
- Wie erzeugt und verwendet man Referenzen auf Skalare, Arrays und Hashes
- Wie verwendet man Referenzen in Subroutinen für Argumente und Rückgabewerte
- Wie erzeugt man verschachtelte Datenstrukturen (mehrdimensionale Arrays, Arrays von Hashes und so weiter)

Was ist eine Referenz?

Eine ***Referenz*** ist eine Form von Daten in Perl, die auf andere Daten verweisen. Die Referenz selbst ist, wie eine Zahl oder ein String, ein Skalar. Man kann sie deshalb wie eine Zahl oder ein String einer Skalarvariablen zuweisen, ausgeben, etwas hinzufügen, auf ***wahr*** oder ***falsch*** testen, in einer Liste speichern oder einer Subroutine übergeben. Zusätzlich zu diesem skalartypischen Verhalten verweist oder zeigt eine Referenz auf die Position anderer Daten. Um herauszufinden, worauf die Referenz zeigt, ***dereferenziert*** man die Referenz - eine etwas hochgestochene Beschreibung dafür, dass man dem Verweis nachgeht, um zu sehen, worauf er verweist.

Falls Sie noch an weiteren terminologischen Pretiosen interessiert sind: das Objekt, auf das die Referenz zeigt, wird auch Referent genannt. Sie dereferenzieren also die Referenz, um den Referenten zu erhalten. Ich persönlich ziehe einfache Formulierungen wie »das Objekt, auf das verwiesen wird« vor.

Referenzen in Perl sind den Zeigern und Referenzen anderer Sprachen sehr ähnlich und bieten die gleichen Vorteile. Wenn Sie jedoch keine anderen Sprachen kennen, könnten Sie sich fragen: »Und wozu das Ganze? Warum sollte man sich mit einer Referenz herumschlagen, wenn man auch mit den Daten selbst arbeiten kann?« Nun, durch den indirekten Bezug auf die Daten können Sie fortschrittlichere Aufgaben mit diesen Daten erledigen - zum Beispiel große Datenmengen als Referenz an oder aus Subroutinen übergeben oder mehrdimensionale Arrays erzeugen. Außerdem eröffnen Referenzen neue fortgeschrittenere Techniken, beispielsweise die Erzeugung objektorientierter Strukturen in Perl. Im Verlauf dieser Lektion werden wir einige dieser Anwendungsmöglichkeiten besprechen.

Bevor ich jetzt zum eigentlichen Code komme, möchte ich noch einen Punkt klarstellen: Wenn ich in diesem Kapitel von Referenzen spreche, so versteht man in versierten Perl-Kreisen darunter die ***harten*** Referenzen. In Perl gibt es auch noch eine andere Form der Referenz, die sogenannte ***symbolische Referenz***. So gibt es zwar zweifellos gute Gründe für die Verwendung von symbolischen Referenzen, doch sind harte Referenzen im allgemeinen wesentlich nützlicher, weshalb wir auch den Großteil dieses Kapitels damit füllen. Auf symbolische Referenzen werde ich dann im Vertiefungsabschnitt am Ende dieser Lektion etwas näher eingehen.

Die Grundlagen: Ein allgemeiner Überblick über die

Verwendung von Referenzen

Betrachten wir erst einmal einige einfache Beispiele, wie man Referenzen erzeugt und verwendet, damit Sie einen Eindruck von der Technik bekommen. Dabei muss gesagt werden, dass es in Perl mehrere Möglichkeiten gibt, mit Referenzen zu arbeiten. Wir werden uns hier auf die leichtesten und am weitesten verbreiteten Mechanismen konzentrieren und die anderen in einem späteren Abschnitt (»Weitere Möglichkeiten zum Einsatz von Referenzen«) beleuchten.

Eine Referenz erzeugen

Beginnen wir mit etwas, womit Sie bereits unzählige Mal in diesem Buch zu tun hatten: eine einfache Skalarvariable, die einen String enthält.

```
$str = "Dies ist ein String.";
```

Dies ist eine ganz gewöhnliche Skalarvariable, die ganz gewöhnliche skalare Daten enthält. Es gibt eine Position im Speicher, die diesen String aufnimmt, an den Sie über den Variablennamen `$str` gelangen. Wenn Sie `$str` etwas anderes zuweisen, würde die Position im Speicher einen anderen Inhalt bekommen, und `$str` hätte einen anderen Wert. All dies ist allgemein bekannt, da Sie es bereits die ganze Zeit so gemacht haben.

Jetzt möchten wir eine Referenz auf diese Daten erzeugen. Um eine Referenz zu erzeugen, müssen Sie wissen, wo die fraglichen Daten (ein Skalar, ein Array, ein Hash oder eine Subroutine) im Speicher abgelegt sind. Um diese Speicherposition zu erhalten, verwenden Sie den Backslash-Operator (`\`) und den Variablennamen:

```
$strref = \$str;
```

Der Backslash-Operator ermittelt die Speicherposition der Daten, die in `$str` gespeichert sind, und erzeugt eine Referenz auf diese Position. Diese Referenz wird dann der Skalarvariablen `$strref` zugewiesen (zur Erinnerung: Referenzen sind eine Art von skalaren Daten).

Der Backslash-Operator ist dem Adreßoperator (&) von C sehr ähnlich. Beide werden verwendet, um auf die Speicherposition von Daten zuzugreifen.

Zu keiner Zeit kommt der eigentliche Inhalt von `$str` - der String »Dies ist ein String« ins Spiel. Die Referenz beschäftigt sich nicht mit dem Inhalt von `$str`, sondern nur mit seiner Position. Außerdem bleibt `$str` eine Skalarvariable, die einen String enthält. Die Existenz einer Referenz ändert daran nichts (siehe Abbildung 19.1).

Abbildung 19.1: Eine Variable, ein String und eine Referenz

Dieses Beispiel erzeugt eine Referenz auf einen String. Sie können jedoch auch Referenzen auf Arrays, Hashes oder Subroutinen erzeugen - sozusagen auf alles, was eine Speicherposition in Perl hat. Sehen Sie im folgenden das Beispiel für eine Referenz auf ein Array:

```
@array = (1..10);
$arrayref = \@array;
```

Und hier ist ein Beispiel für einen Hash:

```
%hash = (
   'rot' => '0 0 255',
   'gruen' => '0 255 0',
   'blau' => '255 0 0; );
```

```
$hashref = \%hash;
```

Wie schon bei der Referenz auf einen skalaren Wert bleiben das Array und der Hash in diesen Beispielen Arrays und Hashes, die in den Variablen `@array` und `%hash` gespeichert sind. Und die Array- und Hash-Referenzen in `$arrayref` und `$hashref` sind skalare Daten, unabhängig von den Daten, auf die sie zeigen. Das Wichtigste, was Sie sich hiervon merken sollten, ist die Tatsache, dass die Referenz selbst ein Skalar ist. Das, worauf die Referenz verweist, kann praktisch jede Art von Daten sein.

Sie können auch Referenzen auf Subroutinen erzeugen. Da dies jedoch ein eher fortgeschrittenes Thema ist und nicht so oft zur Anwendung kommt wie Referenzen auf Skalare, Arrays und Hashes, werde ich Referenzen auf Subroutinen und deren Einsatz erst im Vertiefungsabschnitt am Ende dieser Lektion beschreiben.

Referenzen ausgeben und verwenden

Sie haben inzwischen mit Hilfe des Backslash-Operators eine Referenz erzeugt und in einer Skalarvariablen gespeichert. Wie sehen diese Referenzen aus? Sie sind Skalare und können deshalb auch überall dort verwendet werden, wo Skalare erlaubt sind. Und wie bei Zahlen oder Strings ist ihr Verhalten kontextabhängig.

Wenn Sie eine Referenz als String verwenden, enthält der String die Information, auf welche Art von Daten die Referenz verweist (ein Skalar, ein Array, ein Hash und so weiter), und eine Hexadezimalzahl, die die interne Speicherposition angibt, auf die die Referenz zeigt. Wenn Sie also folgenden Code hätten:

```
print "$strref\n";
```

sähe die Ausgabe ungefähr folgendermaßen aus:

```
SCALAR(0x807f61c)
```

Die Ausgaben für `$arrayref` und `$hashref` wären entsprechend:

```
ARRAY(0x807f664)
HASH(0x807f645)
```

Wenn Sie eine Referenz in einem numerischen Kontext verwenden, erhalten Sie die gleiche Hexadezimalzahl wie bei Verwendung der Referenz in einem String-Kontext - eine Zahl, die die Speicherposition darstellt, auf die die Referenz zeigt. Die Zahlen, die sowohl in der String- als auch der Zahlendarstellungen der Referenz auftauchen, variieren in Abhängigkeit davon, wann das Skript ausgeführt wird und welcher Speicher gerade für Perl frei ist. Sie sollten sich nicht auf diese Zahlen verlassen; betrachten Sie sie lediglich als eine interne Darstellung des Ortes, auf den die Referenz zeigt.

Weitere Anwendungen von Referenzen? Sie können Referenzen Skalarvariablen zuweisen (wie in unserem Beispiel), sie als Listenelemente verwenden oder sie in Arrays oder Hashes speichern (Sie können sie jedoch nicht als Hash-Schlüssel verwenden - mehr dazu später). Eine Referenz, die als Test verwendet wird, liefert immer ***wahr*** zurück. Die mit Abstand gebräuchlichste Art, eine Referenz zu nutzen, besteht allerdings darin, sie zu dereferenzieren, um dadurch Zugriff auf die Daten zu erhalten, auf die die Referenz zeigt.

Referenzen dereferenzieren

Wenn Sie eine Referenz dereferenzieren, greifen Sie auf die Daten zu, auf die die Referenz zeigt. Sie können sich auch vorstellen, dass Sie der Referenz folgen oder auf die Position, auf die sich die Referenz bezieht, zugreifen. Genau dies bezeichnet man allgemein mit dem Begriff ***dereferenzieren***.

Es gibt mehrere Möglichkeiten, eine Referenz zu dereferenzieren. Der einfachste Weg besteht darin, die Skalarvariable der Referenz dort einzusetzen, wo sonst ein einfacher Variablenname erwartet würde:

```
$originalstr = $$strref;
```

Zwei Dollarzeichen? Ja, denn ein einfaches Dollarzeichen markiert lediglich die Skalarvariable `$strref`, die Ihnen die Referenz selbst liefert. Das zweite Dollarzeichen besagt: »Gib mir das, worauf `$strref` zeigt.« In diesem Fall wäre das, worauf `$strref` verweist, der Originalstring »`Dies ist ein String`«. Sie können sich die zwei Dollarzeichen auch so erklären, dass Sie den Namen der Variablen, auf die Sie zugreifen wollen, durch die Referenz - `$strref` - ersetzen.

Um einer Array-Referenz zu folgen und Zugriff auf das Array selbst zu erhalten, gehen Sie genau gleich vor, allerdings mit einem @-Zeichen, und `$arrayref` steht dort, wo der Name des Arrays stehen würde:

```
@ersteliste = @$arrayref;
```

Der Inhalt von `@ersteliste` ist jetzt das ursprüngliche Array, das wir erzeugt haben und auf das `$arrayref` gezeigt hat (genau genommen ist es eine Kopie der ganzen Elemente dieses Arrays).

Sie möchten über eine Referenz auf ein Array-Element zugreifen? Kein Problem. Die Regel ist die gleiche: Setzen Sie einfach die Variable, die die Referenz enthält, dorthin, wo der Name des Array stehen müßte:

```
$erstes = $$arrayref[0];
```

Und mit der gleichen Regel erhalten Sie auch die oberste Indexzahl eines Arrays:

```
$index = $#$arrayref;
```

Bei Hashes ist das nicht anders:

```
%neueshash = %$hashref;    # das kopieren, worauf $hashref zeigt
$wert = $$hashref{rot};    # ermittelt den Wert für den Schlüssel "rot"
@keys = keys %$hashref;    # extrahiert die Schlüssel
```

Referenzierte Daten ändern

Kommen wir jetzt zum schwierigen Teil: das Ändern der Daten, auf die die Referenz verweist. Angenommen Sie haben wie in den obigen Beispielen eine Referenz `$strref`. Nun wird aber der Wert von `$str` geändert:

```
$str = "Dies ist ein String."
$strref = \$str;
#...
$str = "Dies ist ein anderer String."
```

Was passiert mit der Referenz `$strref`? Sie bleibt weiter bestehen. Sie zeigt weiterhin auf die Speicherposition der Variablen namens `$str`. Wenn Sie sie dereferenzieren, erhalten Sie den neuen String:

```
print "$$strref\n";   # Ausgabe: " Dies ist ein anderer String."
```

Die Referenz selber schert sich nicht um den Inhalt des Speicherplatzes, auf den sie verweist, sondern nur um die Speicherposition. Sie können den Inhalt also beliebig ändern - die Referenz wird immer auf die gleiche Position zeigen. Jedesmal, wenn Sie die Referenz dereferenzieren, erhalten Sie das, was gerade zur Zeit an dieser Position im Speicher steht.

Beachten Sie, dass sich dies von der Zuweisung regulärer Variablen unterscheidet, denn dort wird der Inhalt einer Speicherposition an eine andere Position kopiert. Referenzen zeigen immer auf die gleiche Speicherposition, und der Inhalt kann ohne Beeinflussung der Referenz geändert werden. Nehmen Sie zum Beispiel folgende Anweisungen:

```
@array1 = qw(Achtung Fertig Los);
@array2 = @array1;
$arrayref = \@array1;
push @array1, "Stop";
$, = ' ';  # setzt den Begrenzer für die Array-Elemente
print "@array1\n";
print "@array2\n";
```

```
print "@$arrayref\n";
```

Können Sie erraten, was für jede der drei `print`-Anweisungen ausgegeben wird? Der Inhalt von `@array1` wird in der ersten Anweisung erzeugt und später in der vierten geändert, so dass die Ausgabe von `@array1` folgendermaßen lautet:

```
Achtung Fertig Los Stop
```

`@array2` wird in Zeile 2 der Inhalt von `@array1` zugewiesen. Durch die Listenzuweisung wird das rechte Array in seine Bestandteile erweitert, und dann werden diese Bestandteile dem Array auf der linken Seite zugewiesen. So erhält `@array2` eine Kopie des aktuellen Inhalts von `@array1`. Wird `@array2` ausgegeben, erzeugt es folgende Ausgabe:

```
Achtung Fertig Los
```

Die Änderungen an `@array2` haben keine Auswirkungen auf `@array1`. Beides sind getrennte Arrays mit voneinander unabhängigen Inhalten.

Die Referenz auf `@array1` in `$arrayref` wird jedoch die gleiche Ausgabe liefern wie der aktuelle Inhalt von `@array1`, da die Referenz auf die gleiche Speicherposition zeigt wie `@array1`. Die Ausgabe dieser Dereferenzierung lautet daher:

```
Achtung Fertig Los Stop
```

Referenzen als Argumente und Rückgabewerte von Subroutinen

Inzwischen dürften Sie eine Vorstellung davon haben, wie Referenzen funktionieren. Es gäbe noch eine Menge zum Erzeugen und Verwenden von Referenzen zu sagen, aber lassen Sie uns hier erst einmal pausieren und uns der Praxis zuwenden. Im Kapitel 11, »Subroutinen erzeugen und verwenden«, habe ich im Zusammenhang mit den Subroutinen erwähnt, dass Listen und Subroutinen dazu neigen, unhandlich zu werden, wenn man keine Referenzen verwendet. Ich möchte nun auf dieses Thema zurückkommen und untersuchen, wie man mit Referenzen die Listenargumente und Rückgabewerte von Subroutinen wesentlich einfacher verwalten kann.

Subroutinenargumente

Wie Sie bereits wissen, verfügt Perl nur über ganz elementare Möglichkeiten zur Übergabe und zur Rückgabe von Argumente an und aus Subroutinen. Alle Listenargumente, die an eine Subroutine übergeben werden, werden in einer einzigen Liste zusammengefaßt und in `@_` gespeichert. Rückgabewerte werden ebenfalls als einfacher Skalar oder als eine einfache Liste von Skalaren zurückgegeben. Die einfachen Argumente lassen sich dadurch zwar leicht bearbeiten, doch gilt dies nicht für Subroutinen, die mehrere Listen als Argumente übernehmen, da diese Listen auf ihrem Weg in die Subroutine ihre Identität verlieren. Wie ich bereits in Kapitel 11 angemerkt habe, können Sie diese Beschränkung auf vielfältige Weise umgehen. So können Sie zum Beispiel Listen in globalen Array- oder Hash-Variablen speichern (und damit die Übergabe von Argumenten überhaupt vermeiden) oder Informationen über die Listen selbst als Argument übergeben (wie die Länge), was Ihnen dann hilft, die Liste innerhalb der Subroutine selbst zu rekonstruieren.

Am geschicktesten - und oft auch am effizientesten - umgehen Sie Perls Hang, Listenargumente an Subroutinen zu einer Liste zusammenzufassen, indem Sie es überhaupt vermeiden, Listeninhalte an Subroutinen zu übergeben. Übergeben Sie statt dessen Referenzen, und dereferenzieren Sie diese Referenzen dann innerhalb der Subroutine, um an die Inhalte der Listen zu gelangen.

Betrachten wir ein Beispiel, das aus einer früheren Übung stammt: eine Subroutine, die zwei Arrays als Argumente übernimmt und eine Liste aller Elemente zurückliefert, die beiden gemeinsam sind (die Schnittmenge der beiden Arrays). Die Länge des ersten Arrays wird als erstes Argument übergeben, so dass wir die beiden Arrays innerhalb der Subroutine rekonstruieren können. Im folgenden Beispiel rufen wir dafür die Funktion `splice` auf (denken Sie daran, dass `shift` ohne Argumente innerhalb einer Subroutine `@_` verschiebt):

```
1:  sub inter {
```

```
2:    my @first = splice(@_,0,shift);
3:    my @final = ();
4:    my ($el, $el2);
5:
6:    foreach $el (@first) {
7:        foreach $el2 (@_) {
8:            if (defined $el2 && $el eq $el2) {
9:                push @final,$el2;
10:               undef $el2;
11:               last;
12:           }
13:       }
14:   }
15:   return @final;
16: }
```

Wir rufen diese Subroutine mit einer Längenangabe und zwei Arrays als Argumente auf:

```
@one = (1..10);
@two = (8..15);
@three = inter(scalar(@one),@one,@two);
```

Man könnte jetzt behaupten, dass dieses Beispiel gar nicht so schrecklich ist. Es sind schließlich nur zwei Arrays, und mit `splice` wird dafür Sorge getragen, dass die Elemente korrekt getrennt werden. Was aber, wenn Sie mehr als zwei Arrays als Argumente hätten? Da hätten Sie viel zu trennen. Und wenn dann noch eines der Arrays extrem groß wäre, müßten Sie erst einmal eine Menge Elemente kopieren, bevor Sie damit beginnen könnten, tatsächlich irgendwelche Elemente zu verarbeiten. Nicht gerade sehr effizient.

Wagen wir uns jetzt an eine Neufassung dieser Subroutine, die Referenzen verwendet. Anstatt der Subroutine die eigentlichen Arrays zu übergeben, übergeben wir Referenzen auf diese Arrays. Innerhalb der Subroutine weisen wir diese Referenzen dann Variablen zu und dereferenzieren sie, um den Inhalt zu erhalten. Unsere neue Subroutine könnte folgendermaßen aussehen:

```
1:  sub inter {
2:      my ($first, $second) = @_;
3:      my @final = ();
4:      my ($el, $el2);
5:
6:      foreach $el (@$first) {
7:          foreach $el2 (@$second) {
8:              if (defined $el2 && $el eq $el2) {
9:                  push @final,$el2;
10:                 undef $el2;
11:                 last;
12:             }
13:         }
14:     }
15:     return @final;
16: }
```

Diese Subroutine rufen wir nur mit zwei Argumenten auf: den Referenzen auf die Arrays:

```
@one = (1..10);
$oneref = \@one;
@two = (8..14);
$tworef = \@two;
@three =  inter($oneref,$tworef);
```

Es gibt nur zwei Unterschiede zwischen dieser Subroutine und der vorherigen: die erste Zeile für die Argumentliste (Zeile 2) und die Referenzen auf die Listen in den beiden verschachtelten `foreach`-Schleifen (Zeilen 6 und 7). In der Referenzversion brauchen wir die Elemente der Argumentliste nicht mit `splice` zu trennen. Die Argumentliste enthält nur zwei Elemente: die zwei skalaren Referenzen. Deshalb können wir `splice` durch eine gewöhnliche Skalarzuweisung an zwei lokale Variablen ersetzen.

Mit diesen Referenzen kommen wir zu den `foreach`-Schleifen, die die einzelnen Listenelemente überprüfen. Hier arbeiten wir nicht mehr mit lokalen Arrays, sondern nur noch mit Referenzen. Um an den Inhalt der Arrays zu

gelangen, dereferenzieren wir die Referenzen mit `@$first` und `@$second` und greifen auf diese Weise auf den Inhalt der Arrays zu.

Bei dieser Subroutine behält jedes Array seinen ursprünglichen Inhalt und Aufbau. Wenn Sie Referenzen auf Hashes übergeben, bleiben diese Hashes und werden nicht in Listen zusammengefaßt. Außerdem entfällt das lästige Kopieren der Listendaten von einem Ort zu einem anderen, wie das bei der Übergabe von regulären Listen erforderlich ist. Diese Vorgehensweise ist wesentlich effizienter und oftmals auch leichter durchzuführen und zu verstehen, vor allem bei Subroutinen mit komplexen Argumenten.

Referenzen aus Subroutinen zurückgeben

Das Gegenstück zum Übergeben von Listen an Subroutinen ist das Zurückgeben von Listen aus Subroutinen mittels des `return`-Operators. Standardmäßig liefert `return` entweder einen einfachen Skalar oder eine Liste zurück, wobei mehrere Listen in einer Liste zusammengefaßt werden.

Um mehrere Elemente aus einer Subroutine zurückzugeben und dabei die Integrität von Listen und Hashes zu wahren, liefern Sie einfach Referenzen auf diese Strukturen zurück. Gehen Sie dabei genauso vor wie bei der Übergabe von Listendaten an eine Subroutine:

```
sub foo {
   my @templist;
   my @temphash;
   #...
   my $tempref = \@templist;
   my $temphashref = \@temphash;
   return ($tempref, $temphashref);
}
```

Dies mag auf den ersten Blick wie ein Fehler erscheinen, denn die Variablen in diesem Beispiel, `@templist` und `%temphash`, sind lokale Variablen, die aufgelöst werden, nachdem die Subroutine ausgeführt worden ist. Wenn aber die Variablen ihren Gültigkeitsbereich verlieren, stellt sich die Frage, worauf die Referenzen überhaupt noch verweisen? Das Geheimnis hierbei ist, dass zwar der Variablen***name*** verschwindet, wenn die Variable ihren Gültigkeitsbereich verliert (das heißt, wenn die Ausführung der Subroutine beendet ist), dass aber die Daten noch existieren und die Referenz auch noch weiterhin darauf zeigt. Genau genommen ist die Dereferenzierung dieser Referenz jetzt der ***einzige*** Weg, um weiterhin Zugriff auf diese Daten zu haben. Dieses Verhalten beeinflußt auch die Art und Weise, wie Perl Speicher belegt und freigibt, während Ihr Skript ausgeführt wird. Doch zu diesem Thema mehr im Abschnitt »Einige Anmerkungen zu Speicher und Speicherbereinigung«.

Weitere Möglichkeiten zum Einsatz von Referenzen

Meist erzeugt man Referenzen mit Hilfe des Backslash-Operators und dereferenziert sie, indem man die Referenz dort einsetzt, wo an sich ein Name erwartet wird. Doch gibt es noch viele weitere Wege, Referenzen zu erzeugen und nutzen, von denen Ihnen einige neue und komplexe Möglichkeiten eröffnen und anderes in besser lesbarer Form bewirken. In diesem Abschnitt möchte ich Ihnen einige dieser Wege zum Erzeugen und Nutzen von Referenzen aufzeigen und einige weitere Themen rund um Referenzen ansprechen.

Über Listenreferenzen auf Listenelemente zugreifen

Angenommen Sie haben eine Referenz auf eine Liste. Dann werden Sie diese Referenz wahrscheinlich am häufigsten dazu nutzen, Zugriff auf die einzelnen Elemente in der Liste zu erhalten - um sie auszugeben, zu sortieren, aufzuteilen und so weiter. Das könnten Sie zum einen mit der Syntax machen, die wir zu Beginn dieser Lektion gelernt und im vorigen Beispiel verwendet haben:

```
print "Kleinste Zahl: $$ref[0]\n";
```

Diese spezielle Zeile gibt das erste Element des Arrays aus, auf das die Referenz `$ref` zeigt. Wenn Sie das gleiche bei einem Hash machen wollten, müßten Sie die Hash- Syntax verwenden und den Hash-Namen durch die Referenz ersetzen:

```
print "Johns Nachname: $$ref{john}\n";
```

Es gibt jedoch noch einen anderen Weg, Zugriff auf die referenzierten Listen- und Hash-Elemente zu erhalten - ein Weg, der in vielen Fällen (besonders bei komplexen Datenstrukturen und objektorientierten Objekten) etwas einfacher zu lesen ist. Verwenden Sie die Referenz, den Pfeiloperator (->) und einen Array-Index, um auf die referenzierten Listenelemente zuzugreifen:

```
$erstes = $listref->[0];
```

Beachten Sie, dass in diesem Ausdruck nur ein Dollarzeichen steht. Dieser Ausdruck dereferenziert die Listenreferenz in der Variablen `$listref` und liefert das `0`-te Element der Liste zurück. Damit entspricht dieser Ausdruck exakt dem Standardmechanismus zum Dereferenzieren:

```
$erstes = $$listref[0];
```

Diese Syntax läßt sich auch auf Hashes übertragen; verwenden Sie in diesem Fall die Hash-Referenz, den Pfeiloperator (->) und den Hash-Schlüssel in Klammern:

```
$wert = $hashref->{$key};
```

Diese Form ist identisch mit folgendem Ausdruck:

```
$wert = $$hashref{$key};
```

Verwechseln Sie den Pfeiloperator -> nicht mit dem Hash-Paar-Operator =>. Ersterer wird verwendet, um eine Referenz zu dereferenzieren, letzterer ist das gleiche wie ein Komma und wird verwendet, um die Syntax für die Initialisierung von Hash-Inhalten besser lesbar zu machen.

Wir werden auf diese Syntax noch einmal im Abschnitt »Verschachtelte Datenstrukturen mit Referenzen« zu sprechen kommen.

Referenzen mit Blöcken

Ein dritter Weg, Referenzen zu dereferenzieren, ist dem ersten Weg recht ähnlich. Anstatt jedoch den Namen der Referenzvariable an die Stelle eines regulären Variablennamens zu setzen, ersetzen Sie diesen durch einen Block (geschweifte Klammern), der bei seiner Auswertung eine Referenz zurückliefert. Angenommen Sie hätten zum Beispiel eine Referenz auf eine Liste:

```
$listref = \@list;
```

Mit normaler Dereferenzierung würden Sie auf das dritte Element in der Liste wie folgt zugreifen:

```
$third = $$listref[3];
```

Oder mit der Pfeilnotation:

```
$third = $listref->[3];
```

Oder mit einem Block:

```
$third = ${$listref}[3];
```

Um an den Inhalt oder den letzten Index einer Liste über eine Referenz zu gelangen, können Sie wie gewohnt schreiben:

```
@list = @$listref;
$index = $#$listref;
```

oder einen Block verwenden:

```
@list = @{$listref};
$index = $#{$listref};
```

Die obigen Blöcke sind nicht gerade besonders sinnvolle Beispiele in Anbetracht der Tatsache, dass sie lediglich die Variable `$listref` auswerten und Sie dies fast genauso leicht und mit weniger Zeichen durch eine ganz gewöhnliche Dereferenzierung erreichen könnten. Sie könnten jedoch in dem Block selbst eine Subroutine aufrufen, die dann eine Referenz zurückliefert, Sie könnten eine `if`-Bedingung verwenden, um zwischen Referenzen zu wählen, oder im Block einen weiteren Ausdruck aufnehmen. Es muss ja nicht immer nur eine einfache Variable sein.

Mit Hilfe der Blockdereferenzierung können Sie komplexe Dereferenzierungen für komplexe Strukturen, die diese Referenzen verwenden, ausführen. Dieses Thema werden wir noch ausführlicher im Abschnitt »Zugriff auf Elemente in verschachtelten Datenstrukturen« behandeln.

Die ref-Funktion

Angenommen Sie haben eine Referenz in einer Skalarvariablen gespeichert und möchten jetzt wissen, welcher Art die Daten sind, auf die die Referenz verweist, damit Sie nicht plötzlich versuchen, Listen zu multiplizieren oder Elemente aus einem String auszulesen. Dafür gibt es in Perl eine vordefinierte Funktion namens `ref`.

Die `ref`-Funktion übernimmte einen Skalar als Argument. Ist der Skalar keine Referenz, das heißt, ist er ein String oder eine Zahl, dann liefert `ref` einen Null-String zurück. Andernfalls liefert er einen String zurück, der angibt, welcher Art die referenzierten Daten sind. In Tabelle 19.1 finden Sie die möglichen Werte.

Rückgabewert	Darauf zeigt die Referenz
`REF`	Eine andere Referenz
`SCALAR`	Einen skalaren Wert
`ARRAY`	Ein Array
`HASH`	Ein Hash
`CODE`	Eine Subroutine
`GLOB`	Ein Typeglob (Typenplatzhalter)
`** (Null-String)`	Keine Referenz

Tabelle 19.1: ***Mögliche Rückgabewerte der ref-Funktion***

Im Vertiefungsabschnitt werden wir Referenzen auf Subroutinen und Typeglobs besprechen.

Die `ref`-Funktion wird meist dazu verwendet, den Typ einer Referenz zu ermitteln:

```
if (ref($ref) eq "ARRAY") {
   foreach $key (@$ref) {
     #...
   }
} elsif (ref($ref eq "HASH") {
   foreach $key (keys %$ref) {
     #...
   }
}
```

Einige Anmerkungen zu Speicher und Speicherbereinigung

Einer der Nebeneffekte beim Verwenden von Referenzen betrifft die Menge an Speicherplatz, die Perl besetzt, wenn es Ihr Skript ausführt und verschiedene Arten von Daten erzeugt. Normalerweise reserviert Perl bei der

Ausführung Ihres Skripts automatisch Speicherbereiche für Ihre Daten und fordert diese Bereiche wieder zurück, wenn Sie fertig sind. Der Prozeß der Rückforderung - auch ***Speicherbereinigung*** genannt - unterscheidet Perl von vielen anderen Sprachen wie zum Beispiel C, wo Sie selbst Speicher allokieren und freigeben müssen.

Perl verwendet etwas, was man einen Referenzen zählenden Speicherbereiniger nennt. Das bedeutet, dass Perl für alle Daten verfolgt, wie viele Referenzen - einschließlich des ursprünglichen Variablennamens - auf die Daten verweisen. Wenn Sie also eine Referenz auf bestimmte Daten erzeugen, inkrementiert Perl die Referenzzählung um 1. Wenn Sie die Referenz auf etwas anderes verschieben oder eine lokale Variable, die Daten enthält, am Ende eines Blocks oder einer Subroutine verschwindet, dekrementiert Perl die Referenzzählung. Und liegt die Referenzzählung bei 0 - das heißt, es gibt keine Variablen, die sich auf diese Daten beziehen, noch irgendwelche Referenzen, die auf die Daten verweisen -, fordert Perl den Speicherbereich, der von den Daten eingenommen wurde, wieder zurück.

In der Regel läuft dies alles automatisch ab, und Sie müssen sich in Ihrem Skript um gar nichts kümmern. Es gibt jedoch in Zusammenhang mit den Referenzen einen Fall, bei dem Sie vorsichtig sein müssen: wenn es zu Kreisschlüssen durch Referenzen kommt.

Betrachten wir die folgenden zwei Referenzen:

```
sub silly {
   my ($ref1, $ref2);
   $ref1 = \$ref2;
   $ref2 = \$ref1;
   # .. törichte Dinge!
}
```

In diesem Beispiel zeigt die Referenz in `$ref1` auf das, worauf auch `$ref2` zeigt, und `$ref2` zeigt auf das, worauf auch `$ref1` zeigt. Dieses Phänomen nennt man einen Kreisschluß. Die Schwierigkeit dabei ist, dass die lokalen Variablennamen `$ref1` und `$ref2` zwar verschwinden, wenn die Subroutine ihre Ausführung beendet hat, die in den lokalen Variablen enthaltenen Daten aber weiterhin zumindest einmal referenziert werden, so dass der Speicher für diese Referenzen nicht zurückgefordert werden kann. Und ohne die Variablennamen oder eine zurückgelieferte Referenz auf die eine oder andere Referenz können Sie nicht einmal auf die Daten aus der Subroutine zugreifen. Die Daten liegen dann einfach so im Speicher herum. Und jedesmal, wenn die Subroutine bei Ausführung Ihres Skripts aufgerufen wird, wächst der Speicherbereich, den Sie nicht mehr zurückfordern können, an, bis Perl irgendwann den ganzen Speicher auf Ihrem System belegt hat, wenn Ihr Skript nicht vorher die Ausführung abbricht.

Kreisschlüsse durch Referenzen sind demnach schädlich. Und sollte Ihnen dieses Beispiel hier etwas dumm und zu leicht zu durchschauen erscheinen, so möchte ich Sie darauf hinweisen, dass man bei komplexen Datenstrukturen, bei denen überall Referenzen irgendwo hinzeigen, durchaus Gefahr läuft, zufällig und unbeabsichtigt eine Kreisreferenz zu erzeugen. Denken Sie also daran, alle Referenzen, die Sie in Blöcken oder Subroutinen verwenden, »aufzuräumen« (verwenden Sie `undef`, oder weisen Sie ihnen eine `0` oder `''` zu), um sicherzustellen, dass der Speicher immer wieder zurückgefordert wird.

Verschachtelte Datenstrukturen mit Referenzen

Doch Referenzen sind nicht nur in Subroutinen nützlich. Sie dienen noch einem anderen wichtigen Zweck. Ohne sie gäbe es keine komplexen Datenstrukturen wie zum Beispiel mehrdimensionale Arrays. In diesem Abschnitt möchte ich Ihnen zeigen, wie man komplexe Datenstrukturen mit verschachtelten Arrays und Hashes auf der Basis von Referenzen und anonymen Daten aufbaut. Weiter hinten in dieser Lektion, im Abschnitt »Zugriff auf Elemente in verschachtelten Datenstrukturen«, erfahren Sie dann, wie Sie die Daten wieder aus den verschachtelten Datenstrukturen, die Sie gerade erzeugt haben, auslesen.

Was sind verschachtelte Datenstrukturen?

Normalerweise sind Listen, Arrays und Hashes in Perl flach und eindimensional und enthalten nichts außer Skalare. Wenn Sie mehrere Listen kombinieren, werden diese alle in einer Liste zusammengefaßt. Hashes sind im Grunde genommen das gleiche wie Listen, bei denen nur die Daten intern anders organisiert sind. Dadurch wird zwar das Erzeugen und Verwenden von Datensammlungen recht einfach, wenn Sie jedoch versuchen, größere oder komplexere Datensätze effizient zu repräsentieren, stoßen Sie bald an die Grenzen dieses Konzepts.

Angenommen Ihre Daten bestehen aus Informationen über Menschen: Vorname, Nachname, Alter, Größe und Namen der Kinder. Wie würden Sie diese Daten darstellen? Vor- und Nachname sind kein Problem: Sie erzeugen einen Hash mit dem Nachnamen als Schlüssel und weisen diesem Schlüssel dann die Vornamen als Werte zu. Jetzt die Größe - nun gut, Sie könnten für die Größen einen zweiten Hash erzeugen, der ebenfalls den Nachnamen als Schlüssel verwendet. Doch was machen Sie, wenn Sie die Namen der Kinder aufnehmen wollen? Vielleicht einen dritten Hash einrichten, der ebenfalls den Nachnamen als Schlüssel nutzt und dessen Werte Strings sind, die die durch Doppelpunkte getrennten Namen der Kinder enthalten und bei Bedarf wieder in die einzelnen Namen zerlegt werden? Sie sehen: Sobald die Daten komplex werden, versinkt man in einem Wust einfacher Listen, oder man erzeugt seltsame Gebilde mit Strings, weil man ansonsten keine Listen innerhalb von anderen Listen speichern kann.

Hier kommen nun die Referenzen ins Spiel. Eine Liste ist eine flache Sammlung von Skalaren - daran ändert sich auch nach Einführung der Referenzen nichts. Aber eine Referenz ist ein Skalar - und eine Referenz kann auf eine andere Liste verweisen. Und diese Liste kann selbst auch wieder Referenzen auf andere Listen enthalten. Kapiert? Mit Referenzen können Sie Listen in Listen, Arrays in Arrays und Hashes in Arrays verschachteln, aber auch Arrays als Werte für Hashes verwenden und so weiter. All diese Konstrukte - eine beliebige Kombination von Listen, Arrays und Hashes - nennen wir ***verschachtelte Datenstrukturen***.

Anonyme Daten verwenden

Auch wenn Referenzen für das Erzeugen von verschachtelten Datenstrukturen unentbehrlich sind, sind sie nicht das einzige Hilfsmittel, das das Erstellen von verschachtelten Datenstrukturen leichter macht. Neben den Referenzen sollte man sich auch mit anonymen Daten auskennen. Der Begriff ***anonym*** bedeutet »ohne einen Namen«. Anonyme Daten beziehen sich speziell in Perl auf Daten (normalerweise Arrays oder Hashes, aber auch Subroutinen), auf die nur über eine Referenz zugegriffen werden kann - also Daten, die mit keinem Variablennamen verbundenen sind.

> *Wie schon im letzten Abschnitt betrachten wir auch hier vornehmlich Arrays und Hashes. Anonyme Subroutinen und die Referenzen werde ich im Vertiefungsabschnitt am Ende dieser Lektion ansprechen.*

Anonymen Daten sind wir schon weiter vorne in diesem Kapitel begegnet, als wir Arrays in Subroutinen erzeugt und dann Referenzen auf diese Arrays zurückgeliefert haben. Nachdem die Subroutine abgelaufen ist, verschwindet die ursprüngliche lokale Variable, die die Daten enthält, und der einzige Weg, auf die Daten zuzugreifen, führt über eine Referenz. Diese Daten nennt man dann anonym.

Anonyme Daten sind für verschachtelte Datenstrukturen nützlich, weil Sie für die Erzeugung einer Liste von Listen dann nur einen einzigen Variablennamen für die äußere Liste benötigen (und selbst auf diesen könnte man verzichten). Sie brauchen keine einzelnen Variablen für alle Daten innerhalb der Liste, Referenzen reichen völlig aus.

Sie könnten die anonymen Daten für Ihre verschachtelten Datenstrukturen mit Hilfe lokaler Variablen von Subroutinen oder Blöcken erzeugen. In einigen Situationen, beispielsweise wenn Sie Strukturen mit Daten füllen, die Sie aus Dateien oder der Standardeingabe einlesen, ist dies sogar meist der einfachste Weg. Es gibt aber noch eine andere Möglichkeit, anonyme Daten zu erzeugen, und zwar mit Hilfe einer speziellen Perl-Syntax: eckige Klammern `[]` oder geschweifte Klammern `{}`.

Angenommen Sie wollten eine Referenz auf ein Array erzeugen. Normalerweise würden Sie dazu wie folgt vorgehen:

```
@array = ( 1..10 );
$arrayref = \@array;
```

Doch daran ist nichts anonym. Das Array wird in der Variablen `@array` gespeichert. Um das gleiche Array anonym zu erzeugen, müssen Sie die Liste in eckigen Klammern statt in runden Klammern initialisieren. Das Ergebnis wäre eine Referenz, die Sie dann speichern können:

```
$listref = [ 1..10 ];
```

Auf dieses Array kann über die Referenz zugegriffen werden. Sie können die Referenz genauso wie für jedes andere Array dereferenzieren. Aber Sie können nicht über einen Variablennamen auf das Array zugreifen - es handelt sich um anonyme Daten.

Mit anonymen Hashes können Sie genauso verfahren. Hier erzeugen die geschweiften Klammern eine Referenz auf einen anonymen Hash:

```
$hashref = {
   'Taylor' => 12,
   'Ashley' => 11,
   'Jason'  => 12.
   'Brendan' => 13,
}
```

Beachten Sie, dass die Elemente in dem Hash weiterhin als Paare vorliegen - eine Kombination aus Schlüssel und Werten, wie bei normalen Hashes.

> *Die hier verwendeten eckigen und geschweiften Klammern sollten nicht mit denen verwechselt werden, die bei den Array-Indizes `$array[0]` und den Hash-Zugriffen `$hash{key}` zum Einsatz kommen. Die Zeichen sind die gleichen, so dass man sich leichter merken kann, dass eckige Klammern zu den Arrays gehören und geschweifte Klammern zu den Hashes. Aber die Funktion der Klammern ist eine völlig andere.*

Array- und Hash-Klammern konstruieren ein Array oder ein Hash im Speicher und liefern dann eine Referenz auf diese Speicherposition zurück. Alles, was Sie in einem Array oder einem Hash ablegen können, können Sie auch in anonyme Array- oder Hash-Klammern setzen. Dazu gehören auch Array- und Hash-Variablen, obwohl die folgenden zwei Zeilen ***nicht*** das gleiche Ergebnis liefern:

```
$arrayref = \@array;
$arrayref = [ @array ];
```

Der Unterschied zwischen diesen zwei Zeilen ist der, dass die erste Referenz auf die eigentliche Speicherposition der Variablen `@array` zeigt, während die zweite ein neues Array anlegt, in dem alle Elemente von `@array` kopiert werden, und dann eine Referenz auf diese neue Speicherposition erzeugt. Sie könnten es auch als die Erzeugung einer Referenz auf eine Kopie von `@array` bezeichnen. Diesen Trick sollten Sie sich unbedingt für später merken, wenn wir Datenstrukturen in Schleifen erzeugen.

Datenstrukturen mit anonymen Daten erzeugen

Mit anonymen Daten und Referenzen ist das Erstellen von verschachtelten Datenstrukturen letztlich nur eine Frage des Zusammenfügens. In diesem Abschnitt untersuchen wir drei verschiedene Arten von verschachtelten Datenstrukturen: Arrays von Arrays, Hashes von Arrays und Hashes von Hashes.

Arrays von Arrays

Beginnen wir mit etwas einfachem: einem Array von Arrays oder auch einem mehrdimensionalen Array (siehe Abbildung 0.2). Sie könnten zum Beispiel ein Array von Arrays verwenden, um eine Art von zweidimensionalem Feld (wie ein Schachbrett) zu erzeugen. Dabei hat jedes Feld auf dem »Brett« eine Position irgendwo in einer Reihe, und die Reihen werden in dem größeren Array gespeichert.

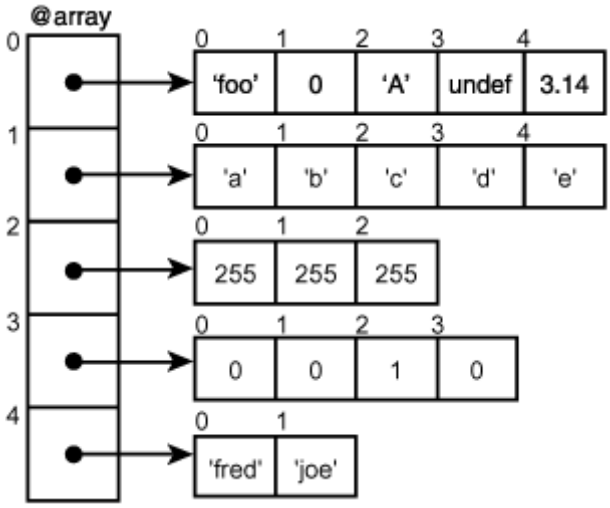

Abbildung 19.2: Ein Array von Arrays

Wenn Sie von C her an echte mehrdimensionale Arrays gewöhnt sind, sollten Sie sich vergegenwärtigen, dass die mehrdimensionalen Arrays in Perl eher Arrays von Zeigern sind und nicht im eigentlichen Sinne mehrdimensional.

Um ein Array von Arrays zu erzeugen, verwendet man für die inneren Arrays die Syntax der anonymen Arrays und für das äußere Array die reguläre Listensyntax. Das folgende Array von Arrays enthält die RGB-Werte für verschiedene Schattierungen von Grau.

```
@grauwerte = (
   [ 0, 0, 0 ],
   [ 63, 63, 63 ],
   [ 127, 127, 127 ],
   [ 191, 191, 191 ],
   [ 255, 255, 255],
);
```

Hier stehen die Arrays mit den Zahlen in eckigen Klammern, wodurch Referenzen auf diese Arrays erzeugt werden. Dann erzeugt das größere Array innerhalb der runden Klammern ein einfaches Array dieser Referenzen. Hüten Sie sich vor folgendem Fehler:

```
@grauwerte = [
   [ 0, 0, 0 ],
   [ 63, 63, 63 ],
   #...
];
```

Die eckigen Klammern um das äußere Array erzeugen eine Referenz auf ein Array und kein normales Array. Vielleicht ist es das, was Sie wollen, aber dann würden Sie das Ergebnis nicht einer Array-Variablen zuweisen. Statt dessen würden Sie die Referenz in einer Skalarvariablen unterbringen.

Hashes von Arrays

Ein Hash von Arrays ist eine verschachtelte Datenstruktur, in der ein Hash mit normalen Schlüsseln als Werte Referenzen auf Arrays erhält (siehe Abbildung 19.3). Sie könnten in einem Hash von Arrays zum Beispiel eine Liste von Personen und deren Kindern abspeichern. Die Schlüssel dieses Hash wären die Namen der Personen und die dazugehörigen Werte Listen der Namen der Kinder. Ein anderes Beispiel wäre ein Skript für ein Kino, das in einem Hash die Filme und deren Vorführungszeiten festhält.

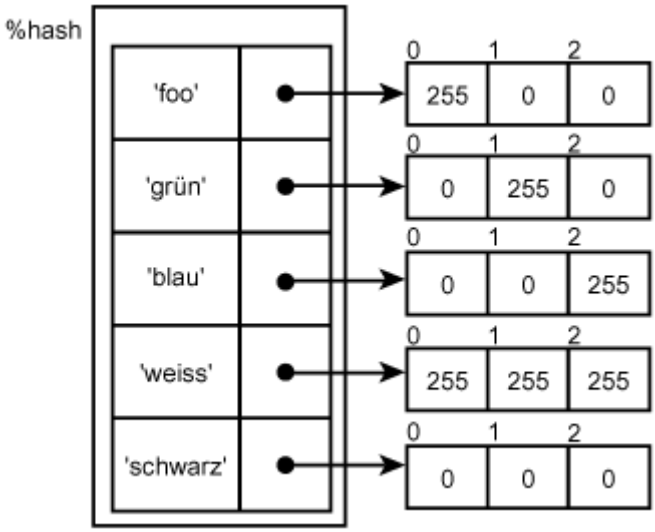

Abbildung 19.3: Hashes von Arrays

Um einen Hash von Arrays zu erzeugen, verwenden Sie für den äußeren Hash die Hash-Syntax, normale Strings als Schlüssel und anonyme Arrays als deren Werte (das folgende Beispiel enthält den Plan der sportlichen Aktivitäten in einem Sommerlager):

```
%plan = (
   'Montag' => [ 'Bogenschiessen', 'Fussball', 'Tanz' ],
   'Dienstag' => [ 'Korbflechten', 'Schwimmen', 'Kanufahren' ],
   'Mittwoch' => [ 'Exkursion', 'Fussball', 'Tanz' ],
   'Donnerstag' => [ 'Freizeit', 'Schwimmen', 'Kanufahren' ],
   'Freitag' => [ 'Bogenschiessen', 'Fussball', 'Wandern' ],
);
```

Auch hier gilt es sorgfältig zwischen eckigen und runden Klammern zu unterscheiden. Die inneren eckigen Klammern erzeugen die anonymen Arrays. Die äußeren runden Klammern erzeugen eine Liste, die dann in einen Hash umgewandelt wird, wenn sie der Variablen `%plan` zugewiesen wird. Mit geschweiften Klammern um die ganze Liste würden Sie eine Referenz auf einen anonymen Hash erzeugen.

Beachten Sie, dass in einem Hash von Arrays nur die Werte Arrays sein können. Die Hash-Schlüssel müssen Strings sein. Genau genommen, geht Perl davon aus, dass es Strings sind, und wandelt gnadenlos alles andere (Zahlen und Referenzen) in Strings um. Achten Sie darauf, dass Sie bei verschachtelten Hashes für Ihre Schlüssel Strings verwenden.

Hashes von Hashes

Wie komplex hätten Sie es gerne? Hashes von Hashes ermöglichen es Ihnen, sehr komplexe Datenstrukturen zu erzeugen. In einem Hash von Hashes enthält der äußere Hash normale Schlüssel und Werte, in denen wiederum Hashes gespeichert sind (siehe Abbildung 19.4). Sie könnten dann mit speziellen Schlüsseln und »Unterschlüsseln« in den einzelnen Hashes suchen. In einem Hash von Hashes könnte man beispielsweise die Kinder einer Schulklasse und die Noten der Kinder in den einzelnen Fächern erfassen. Dabei bilden die Nachnamen der Kinder den Schlüssel, und die Werte wären untergeordnete Hashes mit Einträgen für den Vornamen, das Geburtsdatum und die Noten für jedes Fach.

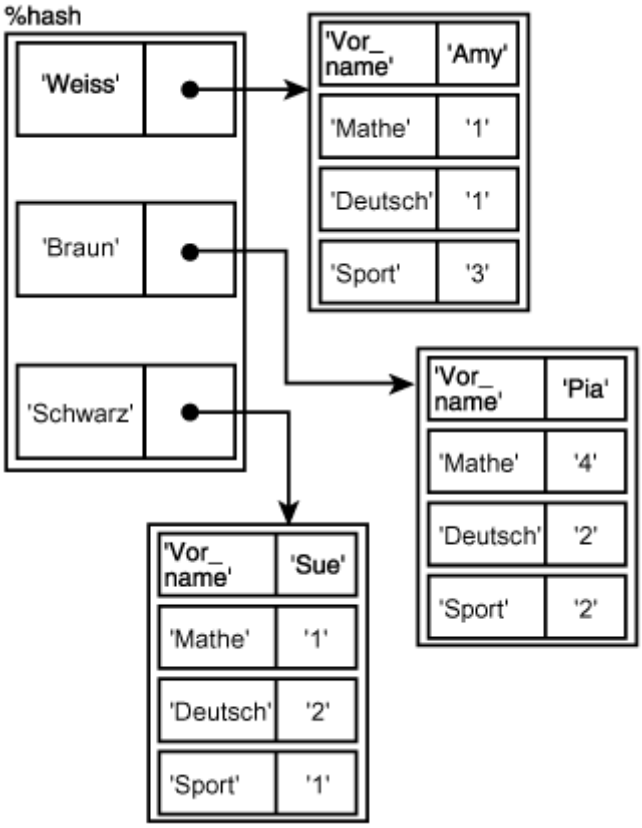

Abbildung 19.4: Hashes von Hashes

Hashes von Hashes verwenden für den inneren Teil die Syntax anonymer Hashes und für den äußeren Teil die Syntax regulärer Listen:

```
%leute = (
   'Jones' => {
      'name' => 'Alison',
      'alter' => 15,
      'tier' => 'dog',
   },
   'Smith' => {
      'name' => 'Tom',
      'alter' => 18,
      'tier' => 'fish',
   },
);
```

Andere Strukturen

Ich habe Ihnen in diesem Abschnitt drei einfache und häufig vorkommende Datenstrukturen vorgestellt: Arrays von Arrays, Hashes von Arrays und Hashes von Hashes. Sie können Arrays und Hashes jedoch fast beliebig mit Referenzen und anonymen Daten kombinieren, je nachdem, mit welchen Daten Sie arbeiten und wie sich die Daten am besten organisieren lassen. Sie können Ihre Daten auch noch tiefer verschachteln, als ich es hier getan habe. Denkbar wäre zum Beispiel auch ein Hash von einem Hash, in dem die Schlüssel Arrays und die Array-Elemente wiederum Hashes sind und so weiter. Es sind Ihnen hinsichtlich der Verschachtelungstiefe keine Grenzen gesetzt. Wenn es also die Situation erfordert, dann sollten Sie sich keine Beschränkungen auferlegen.

Datenstrukturen mit existierenden Daten aufbauen

Die obigen Beispiele für verschachtelte Datenstrukturen mit anonymen Daten lassen sich überall dort gut anwenden, wo Sie bereits im voraus genau wissen, welche Daten die Strukturen enthalten werden. In der Praxis sieht es jedoch so aus, dass solche komplexeren Datenstrukturen meistens aus Daten aufgebaut sind, die aus einer Datei gelesen oder über die Tastatur eingegeben wurden.

In solchen Fällen kann es passieren, dass Sie sowohl mit anonymen Daten als auch mit Referenzen auf reguläre Variablen arbeiten. So lange die Referenzen dabei an der richtigen Stelle stehen, ist es egal, welchen Mechanismus Sie zum Aufbau Ihrer Datenstruktur verwenden. Angenommen Sie haben eine Datei, die die folgende Matrix von Zahlen enthält:

```
3 4 2 4 2 3
5 3 2 4 5 4
7 6 3 2 8 3
3 4 7 8 3 4
```

Sie wollen diese Datei in ein Array von Arrays einlesen, wobei jede Reihe ein eigenes Array bildet und das äußere Array die einzelnen Reihen speichert. Sie könnten dies mit einer Schleife wie folgt lösen:

```
while (<>) {
   chomp;
   push @matrix, [ split ];
}
```

Diese Schleife würde jede Zeile lesen, das Neue-Zeile-Zeichen mit `chomp` entfernen, mit `split` die einzelnen Elemente voneinander trennen, mit Hilfe der eckigen Klammern ein anonymes Array dieser Elemente erzeugen und zum Schluß die Referenz auf dieses Array in das äußere Array schreiben (`push`).

Vielleicht denken Sie, dass dieses Beispiel besser lesbar wäre, wenn wir die Elemente zuerst in einer Liste aufsplitten und dann eine Referenz auf diese Liste speichern würden:

```
my @list = ();
while (<>) {
   chomp;
   @list = split;
   push @matrix, \@list;
}
```

Aber dieses Beispiel hat einen großen Haken (eine Falle, in die viele Programmierer tappen, die dies zum ersten Mal versuchen). Für eine vorgegebene Eingabe wie die obige Matrix wird diese Schleife ein Array von Arrays erzeugen, das folgendermaßen aussieht:

```
3 4 7 8 3 4
3 4 7 8 3 4
3 4 7 8 3 4
3 4 7 8 3 4
```

Können Sie sich denken, warum? Das Problem hat mit dem Variablennamen zu tun, auf den später die Referenz verweist. Bei jedem Durchlauf der Schleife ändert sich zwar der ***Inhalt*** der Variablen `@list`, aber die Speicherposition bleibt die gleiche. Jedesmal, wenn Sie eine Zeile lesen, legen Sie im äußeren Array eine Referenz auf die ***gleiche*** Speicherposition ab. Das Array, das Sie am Ende erhalten, ist ein Array von vier Referenzen, die alle auf genau die gleiche Speicherposition zeigen.

Eine Möglichkeit, diesen Fehler zu beheben - und er tritt häufig auf, also hören Sie gut zu -, besteht darin, eine Referenz auf eine Kopie der Array-Daten und nicht auf das Array selbst zu erzeugen. Auf diese Weise legen Sie im Speicher bei jedem Durchlauf der Schleife eine neue Position im Speicher an und erhalten Referenzen auf verschiedene Speicherplätze. Man erreicht dies, indem man einfach die eckigen Klammern für anonyme Arrays um die Array-Variable setzt:

```
my @list = ();
while (<>) {
   chomp;
   @list = split;
   push @matrix, [ @list ];
}
```

Achten Sie darauf, dass Sie für anonyme Hashes analog vorgehen (also anonyme Hashes in ein Array von Hashes ablegen):

```
push @arrayvonhashes, { %hash };
```

Das Problem läßt sich aber auch damit lösen, dass Sie in der Schleife eine `my`-Variable verwenden. Da die `my`-Variable bei jedem Schleifendurchlauf neu erzeugt wird, zeigt die Referenz jedesmal im Speicher woanders hin:

```
while (<>) {
   chomp;
   my @list = split;
   push @matrix, \@list ;
}
```

Zugriff auf Elemente in verschachtelten Datenstrukturen

Verschachtelte Datenstrukturen zu erstellen ist eine Sache, auf die Elemente in verschachtelten Datenstrukturen zuzugreifen eine andere. Über Referenzen in Arrays, auf die wieder andere Referenzen weisen, auf ein bestimmtes Element zuzugreifen, ist schon eine unangenehme Aufgabe, besonders bei komplexen Strukturen. Doch zum Glück unterstützt Sie Perl dabei mit einer speziellen Syntax.

Angenommen Sie haben eine Matrix (Array von Arrays) von Zahlen, wie wir sie schon im obigen Abschnitt gesehen haben:

```
@zahlen = (
   [ 3, 4, 2, 4, 2, 3 ],
   [ 5, 3, 2, 4, 5, 4 ],
   [ 7, 6, 3, 2, 8, 3 ],
   [ 3, 4, 7, 8, 3, 4 ],
);
```

Nehmen wir jetzt an, Sie wollten auf das vierte Element in der dritten Reihe zugreifen. Mit Hilfe der Standardsyntax für den Arrayzugriff können Sie auf die dritte Reihe zugreifen:

```
$zahlen[2];
```

Das Ergebnis wäre jedoch nur eine Referenz und nicht die Daten, auf die die Referenz verweist (erinnern Sie sich, Sie erhalten die Daten, auf die eine Referenz weist, nur durch explizites Dereferenzieren). Um die Referenz zu dereferenzieren und ein echtes Element zu erhalten, könnten Sie folgendermaßen vorgehen:

```
$zahlen[2]->[3];  # liefert das vierte Element des Arrays, auf das die
                  # Referenz $zahlen[2] verweist
```

oder so:

```
${ $zahlen[2] }[3];   # $zahlen[2] liefert Ihnen eine Referenz, die in dem
                      # Block dereferenziert wird
```

Beide Varianten sind möglich, doch keine ist besonders gut lesbar. Deshalb stellt Perl Ihnen eine verkürzte Syntax für mehrdimensionale Arrays zur Verfügung, die den Zugriff vereinfacht: Wenn Sie die standardmäßige Dereferenzierungssyntax mit dem Pfeil verwenden, können Sie die `->`-Zeichen fortlassen:

```
$zahlen[2][3];
```

Das ist wesentlich einfacher zu verstehen und entspricht dem Zugriff auf mehrdimensionalen Arrays in anderen Sprachen (wie C zum Beispiel).

Die Situation ist jedoch eine andere, wenn Sie anstatt eines echten Arrays in `@zahlen` nur eine Referenz auf ein Array von Arrays hätten. Dann wären nämlich zwei Referenzen zu dereferenzieren, und Sie würden folgende Syntax verwenden (hier ist `@zahlenref` die Referenz auf das Array von Arrays):

```
$zahlenref->[2][2];
```

Bei verschachtelten Hashes von Arrays und Hashes von Hashes verhält es sich analog, mit geschweiften Klammern für die Hash-Schlüssel und eckigen Klammern für die Array-Indizes:

```
$hash{joe}[5]; # Zugriff auf das sechste Element eines Arrays
               # des Hash %hash über den Schlüssel 'joe'
$hashref->{joe}[5];       # das gleiche, wenn $hashref eine Referenz enthält
$hash{Jones}{alter};      # Wert alter für den Jones-Datensatz in %hash
$hashref->{Jones}{alter}; # das gleiche, $hashref ist Referenz
```

Wenn Sie all diese verschachtelten Indizes und Schlüssel zu sehr verwirren, können Sie statt dessen von der Referenz auf das interne Array oder Hash, das Sie gerade interessiert, eine temporäre Kopie erzeugen und dann diese Referenz auf normalem Weg dereferenzieren:

```
my $tempref = $zahlen[0];  # Referenz auf die erste Reihe von Zahlen
print $$tempref[5];        # gibt fünftes Element aus
                           # das gleiche wie $zahlen[0][5]
```

Sie sind nur an einem Teilbereich eines verschachtelten Arrays interessiert? Normalerweise würden Sie dafür die übliche Syntax verwenden, mit den Referenzen an den entsprechenden Stellen. Des weiteren sind Sie in diesem Fall auf die Blockdereferenzierung angewiesen und am Ende erhalten Sie einen ziemlich häßlichen Ausdruck wie zum Beispiel:

```
@elemente = @{ $zahlen[1] }[2..5];  # extrahiert die Elemente 2 bis 5
                                    # im zweiten Array aus @zahlen):
```

Da diese Notation sehr schnell ziemlich unangenehm werden kann, ist es meist leichter, die Referenzen in temporären Variablen abzulegen und über diese die Teilbereiche zu entnehmen. Oder Sie setzen Schleifen auf, in denen die Elemente einzeln aus dem verschachtelten Array herausgezogen werden. Wenn Sie vertikale Bereiche herauslösen wollen (je ein Element aus verschiedenen der verschachtelten »Reihen«) oder rechteckige Bereiche (einige Elemente horizontal und mehrere Elemente vertikal), dann müssen Sie dazu eine Schleife verwenden.

Ein Beispiel: Eine Datenbank mit Künstlern und ihren Werken

Verschachtelte Datenstrukturen eignen sich bestens zum Verwahren komplexer Datensätze und ermöglichen es, diese Daten auf verschiedene Art und Weise zu manipulieren. In diesem Abschnitt untersuchen wir eine Datenbank, in der Künstler aufgeführt werden. Neben den Namen der Künstler enthält die Datenbank Informationen zu den einzelnen Künstlern und ihren Werken. Um Platz zu sparen, halte ich das Beispiel kurz und beschränke mich darauf,

- die Künstlerdaten aus einer Datei in eine komplexe verschachtelte Datenstruktur zu lesen,
- nach der Eingabe eines Suchstrings zu fragen und
- für einen gegebenen Suchstring die Daten des gesuchten Künstlers auszugeben.

Die Daten, die wir in diesem Beispiel betrachten, umfassen den Vor- und Nachnamen der Künstler, ihr Geburts- und Todesjahr sowie eine Liste der Titel ihrer Werke. Die Künstlerdaten sind in einer externen Datei gespeichert, die pro Künstler zwei Zeilen aufweist:

```
Monet,Claude,1840,1926
Herbst in Argenteuil:Pappeln:Camille:Wasserlilien
```

Die erste Zeile besteht aus den persönlichen Daten des Künstlers, die jeweils durch Kommata getrennt sind, die zweite Zeile enthält die Werke des Künstlers, getrennt durch Doppelpunkte. Die Datendatei, die ich `kuenstler.txt` genannt habe, enthält eine Reihe von Künstlern im gleichen Format.

Die Struktur, in die wir diese Informationen lesen, ist ein Hash von Hashes mit einem verschachtelten Array. Der oberste Hash verwendet als Schlüssel den Nachnamen des Künstlers. Die weiteren Künstlerdaten sind in einem verschachtelten Hash mit den Schlüsseln `FN`, `BD`, `DD` und `works`. Der Wert für den Schlüssel `works` ist wiederum ein Array, in dem die einzelnen Titel aufgeführt sind.

Listing 19.1 enthält den Code für dieses einfache Beispiel. Bevor Sie dazu übergehen, die Analyse dieses Codes zu lesen, studieren Sie die Zeilen in der `while`-Schleife, vor allem der Subroutine `&read_input()` (Zeilen 21 bis 35) und der Dereferenzierungen in der Subroutine `&process()` (Zeilen 53 und 55).

Listing 19.1: Das Skript kuenstler.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:
4:  my $artdb = "kuenstler.txt";           # Name der Künstlerdatenbank
5:  my %artists = ();                      # Hash der Künstler, Nachname
                                           # als Schlüssel
6:
7:  &read_input();
8:  &process();
9:
10: sub read_input {
11:     my $in = '';               # temp. Eingabezeile
12:     my ($fn,$ln,$bd,$dd);      # Nachname, Vorname
13:                                # Geburtsjahr, Todesjahr
14:     my %artist = ();           # temp. Künstler-Hash
15:
16:     open(FILE, $artdb) or
                die "Datenbank ($artdb) konnte nicht geöffnet werden: $!\n";
17:
18:     while () {
19:         # Name und Daten in erster Zeile
20:         chomp($in = <FILE>);
21:         if ($in) {
22:             ($ln,$fn,$bd,$dd) = split(',',$in);
23:             $artist{FN} = $fn;
24:             $artist{BD} = $bd;
25:             $artist{DD} = $dd;
26:
27:             chomp($in = <FILE>); # Liste der Werke in zweiter Zeile
28:             if ($in) {
29:                 my @works = split(':',$in);
30:                 $artist{works} = \@works;
31:             } else { print "no works";}
32:
33:             # die Referenz auf das artist-Hash in das äußere
34:             # artists-Hash eintragen
35:             $artists{$ln} = { %artist };
36:
37:         } else { last; }        # Ende von DB
38:     }
39:
40: }
41:
42: sub process {
43:     my $input = '';
44:     my $matched = 0;
45:
46:     print "Geben Sie einen Künstlernamen ein: ";
47:     chomp($input = <>);
48:
49:     foreach (keys %artists) {
50:         if (/$input/i  and !$matched) {
51:             $matched = 1;
52:             my $ref = $artists{$_};
53:             print "$_, $ref->{FN} $ref->{BD}-$ref->{DD}\n";
54:             my $work = '';
55:             foreach $work (@{$ref->{works}}) {
56:                 print "   $work\n";
57:             }
58:         }
59:     }
60:     if (!$matched) {
61:         print "Der Künstler $input wurde nicht gefunden.\n";
62:     }
63: }
```

Vielleicht ist Ihnen aufgefallen, dass ich in diesem Beispiel genau das Gegenteil von dem gemacht habe, was ich zuvor gepredigt habe: Anstatt alle Variablen lokal zu halten, verwende ich eine globale Variable für die globale Künstlerdatenbank. Wie Sie Ihre Daten und Variablen organisieren, bleibt Ihnen überlassen. Ich verwende in diesem Fall eine globale Variable, weil das Dereferenzieren ohnehin schon kompliziert genug ist und ich daher nicht noch eine weitere Ebene hinzufügen möchte.

Was Ihnen in diesem Beispiel vielleicht auch noch seltsam erscheinen mag, ist der hartcodierte Name der Künstlerdatenbank. Ich habe den Namen der Datei bewußt in den Code des Skripts mit aufgenommen, statt ihn über die Befehlszeile eingeben zu lassen. Aber auch dies ist eine Frage des Programmierstils und wie das Skript verwendet wird. Beide Wege sind gangbar (beachten Sie, dass ich den Dateinamen der Künstlerdatenbank ganz oben im Skript gesetzt habe, so dass er bei Bedarf schnell geändert werden kann).

Betrachten wir zuerst die Subroutine `&read_input()`, die die Künstlerdatenbank einliest und unsere verschachtelte Datenstruktur mit Daten füllt. Ich bin dabei so vorgegangen, dass ich einen temporären Hash für den aktuellen Künstler einrichte, diesen Hash mit Daten fülle und dann den temporären Hash über eine Referenz in den äußeren Hash eintrage.

Wir beginnen in Zeile 18 mit einer Schleife, die die Datenbankdatei der Künstler in Schritten von je zwei Zeilen einliest. Die Schleife wird verlassen, wenn es keine weiteren Daten mehr gibt (Test in Zeile 21). Zuerst wird die erste Zeile der Daten eingelesen, die den Namen des Künstlers sowie seine persönlichen Daten enthält:

```
Monet,Claude,1840,1926
```

Zeile 22 zerlegt diese Daten in ihre Einzelbestandteile, und die Zeilen 23 und 25 legen diese Daten dann in einem temporären Hash ab (namens `%artist`, der jedoch nicht mit dem äußeren Hash `%artists` verwechselt werden sollte).

Zeile 27 liest die jeweils zweite Zeile der Künstlerdaten ein, das heißt die einzelnen Werke:

```
Herbst in Argenteuil:Pappeln:Camille:Wasserlilien
```

In Zeile 29 teilen wir diese Zeile auf der Basis des Trennzeichens »:« in Listenelemente auf und speichern diese Liste in dem temporären Array `@works`. In Zeile 30 fügen wir eine Referenz auf dieses Array in unseren temporären Hash `%artist` mit dem Schlüssel »works« ein. Beachten Sie, dass wir bei jedem Durchlauf der `while`-Schleife einen neuen temporären `@works`-Array (deklariert mit `my`) erhalten, so dass wir das Problem, jedesmal die gleiche Speicherposition zu referenzieren, umgehen.

Nachdem wir auf diese Weise die Daten des Künstlers eingelesen haben, können wir diesen Datensatz endlich in den äußeren Künstler-Hash eintragen, wobei uns der Nachname des Künstlers als Schlüssel dient. Und genau das geschieht in Zeile 35. Beachten Sie hierbei, dass wir, da bei jedem Schleifendurchlauf der gleiche Hash `%artist` verwendet wird, einen anonymen Hash-Konstruktor und eine Kopie des Hash `%artist` verwenden, um sicherzustellen, dass die Referenz jedesmal auf eine andere Speicherposition weist.

Die Subroutine `&read_input()` legt die Daten in einem verschachtelten Hash ab, die Subroutine `&process()` holt die Daten wieder heraus. Dazu bedienen wir uns eines einfachen Algorithmus, um nach dem Nachnamen des Künstlers zu suchen und den gefundenen Datensatz auszugeben. Die Ausgabe dieser Subroutine sieht wie folgt aus:

```
Geben Sie einen Künstlernamen ein: Monet
Monet, Claude 1840-1926
   Herbst in Argenteuil
   Pappeln
   Camille
   Wasserlilien
```

Den wichtigsten Teil dieser Subroutine bilden die Zeilen 52 bis 56, in denen die Referenzen dereferenziert werden, um an die benötigten Daten zu gelangen. Lassen Sie uns aber weiter oben in Zeile 49 mit der `foreach`-Schleife beginnen. Da wir keine eigentliche Schleifenvariable haben, speichert Perl die Schlüssel (die Nachnamen der jeweiligen Künstler) in der Variablen `$_`.

Zeile 50 enthält den eigentlichen Test: Wir führen mit der Eingabe und dem aktuellen Schlüssel einen Mustervergleich durch, um festzustellen, ob es eine Übereinstimmung gibt. Und da wir in diesem Beispiel nur an der ersten Übereinstimmung interessiert sind, verfolgen wir außerdem die Variable `$matched`, um zu sehen, ob bereits eine Übereinstimmung gefunden wurde.

Angenommen es wurde ein übereinstimmender Datensatz gefunden, dann gehen wir weiter zu Zeile 52. Hier

erzeugen wir eine temporäre Variable, die die Referenz auf den Datensatz des Künstlers aufnimmt - dies ist, wie in unserem ***stats***-Beispiel, nicht unbedingt notwendig, erleichtert aber die Verwaltung der Referenzen. Da `$_` den übereinstimmenden Schlüssel enthält, können wir die Referenz in diesem Falle mit einer einfachen Hash-Suche ermitteln.

Haben wir erst einmal die Referenz, können wir sie dereferenzieren, um Zugriff auf den Inhalt des Hash zu erhalten. In Zeile 53 geben wir die persönlichen Daten aus: den Nachnamen (`$_`), den Vornamen (der Wert des Schlüssels `FN` im Hash), das Geburtsjahr (`BD`) und das Todesjahr (`DD`).

Die Zeilen 54 bis 56 dienen dazu, die Werke des Künstlers auf jeweils einer Zeile auszugeben. Merkwürdig ist allein die Referenz in der `foreach`-Schleife. Betrachten wir diese einmal näher:

```
@{$ref->{works}}
```

Zur Erinnerung: In `$ref` befindet sich eine Referenz auf einen Hash. Der Ausdruck `$ref->{works}` dereferenziert diese Referenz und liefert den Wert zurück, der durch den Schlüssel `works` gegeben ist. Dieser Wert ist wiederum eine Referenz, diesmal jedoch eine Referenz auf ein Array. Um diese Referenz zu dereferenzieren und ein tatsächliches Array zu erhalten, das man mit der `foreach`-Schleife durchlaufen kann, bedarf es der Blocksyntax zum Dereferenzieren: `@{}`.

Referenzen zu verstehen und herauszufinden, wie man an die tatsächlichen Daten herankommt, ist nicht immer ganz einfach. Meist hilft es, wenn man von außen nach innen vorgeht, wo nötig, Blöcke verwendet und wenn hilfreich, temporäre Variablen einsetzt. Auch die Analyse der Referenzausdrücke im Perl-Debugger oder mit `print`-Anweisungen kann helfen, die richtigen Dereferenzierungen zu erzeugen.

Vertiefung

Die Erzeugung und Verwendung von Referenzen ist sicherlich einer der komplexeren Aspekte von Perl (vermutlich nur von der objektorientierten Programmierung übertroffen, die wir morgen besprechen wollen). Die heutige Lektion hat Sie in die Grundlagen der Referenzen eingeführt und Ihnen die wichtigsten Einsatzbereiche aufgezeigt. Wie aber bei den meisten Themen in Perl gibt es auch im Zusammenhang mit Referenzen viele Bereiche, die ich nicht angesprochen habe, einschließlich der symbolischen Referenzen (eine gänzlich andere Form von Referenz) sowie der Referenzen auf Subroutinen, Typeglobs und Datei-Handles.

Weitere Informationen zu Referenzen finden Sie in der ***perlref***-Manpage. Wenn Sie intensiver mit verschachtelten Datenstrukturen arbeiten, finden Sie weitere Details und Beispiele in ***perldsc*** (Kochbuch der Datenstrukturen) und ***perllol*** (Liste der Listen).

Kurzformen für Referenzen auf Skalare

Müssen Sie mehrere skalare Referenzen auf einmal erstellen? So geht es ganz leicht:

```
@listevonrefs = \($ding1, $ding2, $ding3, $ding4);
```

Auf diese Weise erhalten Sie in `@listevonrefs` eine Liste von Referenzen. Es ist eine Kurzform von :

```
@listevonrefs = (\$ding1, \$ding2, \$ding3, \$ding4);
```

Symbolische Referenzen

Wie ich zu Beginn dieser Lektion schon am Rande bemerkt habe, kennt Perl eigentlich zwei Arten von Referenzen: harte Referenzen und symbolische Referenzen. Die von mir in dieser Lektion durchgehend verwendeten Referenzen waren harte Referenzen. Harte Referenzen sind eigentlich skalare Daten, die wie Skalare manipuliert werden können und die dereferenziert werden, um auf die referenzierten Daten zuzugreifen.

Symbolische Referenzen sind anders: Eine symbolische Referenz ist einfach ein String. Wenn Sie versuchen, diesen String zu dereferenzieren, wird der String als der Name einer Variablen interpretiert, und falls diese Variable existiert, erhalten Sie den Wert dieser Variablen. Sehen Sie dazu folgendes Beispiel:

```
$foo = 1;                                  # Variable $foo enthält Wert 1
$symref = "foo";                           # String
$$symref = "Ich bin eine Variable";        # setzt die Variable $foo
print "symbolische Referenz: $symref\n";   # Ergebnis ist "foo"
print "Foo: $foo\n";                  # Ergebnis ist "Ich bin eine Variable"
print "dereferenziert: $$symref\n";   # gibt $foo aus,
                                      # Ergebnis ist "Ich bin eine Variable"
```

Wie Sie sehen können, lassen sich symbolische Referenzen wie richtige Referenzen verwenden, es sind jedoch nur Strings, die Variablen benennen. Der Unterschied ist sehr fein und verwirrend, besonders wenn Sie harte und symbolische Referenzen kombinieren. So könnten Sie aus Versehen einen String dereferenzieren, wenn Sie eigentlich vorhatten, einen Skalar zu dereferenzieren, ein Fehler, der sich nur schwer beim Debuggen aufdekken läßt. Aus diesem Grund verfügt Perl über ein `strict-` Pragma, mit dem Sie die Verwendung von Referenzen auf harte Referenzen beschränken können.

```
use strict 'refs';
```

Wenn Sie dieses Pragma ganz oben in Ihr Skript mit aufnehmen, können Sie verhindern, dass symbolische Referenzen verwendet werden. Den gleichen Effekt erzielen Sie aber auch, wenn Sie oben in Ihrem Skript `use strict` allein verwenden.

Referenzen auf Typeglobs und Datei-Handles

Zwei Arten von Referenzen, die ich im Hauptteil dieser Lektion nicht besprochen habe, sind die Referenzen auf Typeglobs, die wiederum Referenzen auf Datei-Handles ermöglichen. Ein Typeglob ist, wie ich weiter vorn in diesem Buch kurz bemerkt habe, eine Möglichkeit, auf mehrere Typen von Variablen, die den gleichen Namen teilen (der in der Symboltabelle eingetragen ist), gleichzeitig Bezug zu nehmen. Typeglobs werden heutzutage in Perl nicht mehr so oft verwendet wie früher, als es noch keine Referenzen gab und man die Typeglobs dazu nutzte, Verweise auf Listen an Subroutinen zu übergeben. Typeglobs stellen aber auch einen Weg dar, um Referenzen auf Datei-Handles zu erzeugen, und geben uns damit eine Möglichkeit an die Hand, bei Bedarf Datei-Handles in und aus Subroutinen zu übergeben oder lokale Datei-Handles zu erzeugen.

Um eine Referenz auf einen Datei-Handle zu erzeugen, verwenden Sie einen Typeglob mit dem Namen des Datei-Handles und dem Backslash-Operator `\`:

```
$fh = \*MEINEDATEI;
```

Für den lokalen Datei-Handle verwenden Sie den Operator `local` (nicht `my`) und ein Datei-Handle-Typeglob:

```
local *MEINEDATEI;
```

Weitere Informationen finden Sie in der ***perldata***-Manpage (im Abschnitt zu Typeglobs und Datei-Handles).

Referenzen auf Subroutinen

Noch nützlicher als Referenzen auf Datei-Handles sind Referenzen auf Subroutinen. Da die Definitionen von Subroutinen wie Arrays oder Hashes im Speicher abgelegt sind, können Sie - wie bei anderen Daten auch - Referenzen auf Subroutinen erzeugen. Wenn Sie eine Referenz auf eine Subroutine dereferenzieren, rufen Sie die Subroutine auf.

Mit Referenzen auf Subroutinen können Sie die Definition einer Subroutine während der Ausführung ändern oder je nach Situation zwischen verschiedenen Subroutinen wählen. Referenzen auf Subroutinen öffnen zudem das Tor zu fortgeschrittenen Techniken in Perl, wie zum Beispiel die objektorientierte Programmierung und Closures (anonyme Subroutinen, deren lokale Variablen sich danach richten, in welchem Gültigkeitsbereich die Subroutine definiert wurde - auch wenn sie später in einem anderen Gültigkeitsbereich aufgerufen wurden).

Um eine Referenz auf eine Subroutine zu erzeugen, verwenden Sie den Backslash- Operator mit dem Namen einer bereits definierten Subroutine:

```
$subref = \&meinesub;
```

Sie können auch Referenzen auf anonyme Subroutinen erzeugen, indem Sie einfach den Namen der Subroutine bei der Definition fortlassen:

```
$subref = sub { reverse @_;};
```

Zum Dereferenzieren verwenden Sie die bekannte Referenzsyntax oder einen Block. Wenn Sie eine Subroutine dereferenzieren, rufen Sie sie auf; deshalb sollten Sie nicht vergessen, die Argumente mit einzuschließen:

```
@ergebnis = &$subref(1..10);
```

Weitere Details zu Referenzen auf Subroutinen und zu Closures finden Sie in der ***perlref***-Manpage.

Zusammenfassung

Der letzte große Teilbereich von Perl, der in diesem Buch noch zur Besprechung anstand, betraf die Referenzen. Mit der heutigen Lektion haben Sie eine gute Einführung in die Erzeugung und Verwendung von Referenzen in verschiedenen Kontexten erhalten.

Referenzen sind so etwas wie skalare Daten, die auf andere Daten (einen anderen Skalar, ein Array, einen Hash oder eine Subroutine) verweisen. Da Referenzen Skalare sind, können Sie sie an Subroutinen übergeben, in Variablen speichern, sie als String oder als Zahl behandeln, auf ihren Wahrheitswert testen oder sie in ein Array aufnehmen. Sie können Referenzen auf zwei Arten erzeugen:

- Verwenden Sie den Backslash-Operator `\` mit einer Variablen.
- Verwenden Sie einen der anonymen Konstruktoren, um eine Referenz auf ein Array, einen Hash oder eine Subroutine zu erzeugen.

Um Zugriff auf das zu erhalten, worauf die Referenz verweist, müssen Sie die Referenz dereferenzieren. Eine Referenz läßt sich auf drei Arten dereferenzieren:

- Verwenden Sie dort, wo sonst der normale Variablenname stehen würde, die Referenzvariable; zum Beispiel `$$ref`, `@$ref` oder `$$ref[0]`.
- Verwenden Sie dort, wo sonst der Variablenname stehen würde, einen Blockausdruck (der als Referenz ausgewertet wird); zum Beispiel `@{$ref[0]}`.
- Verwenden Sie besonders für Referenzen auf Listen die »Pfeilnotation« (`$ref->[0]` oder `$ref->{key}`. Bei verschachtelten Arrays und Hashes können Sie mehrere Indizes angeben, ohne dazwischen Pfeile setzen zu müssen (`$ref->[0][4]` oder `$ref[0]{key}`).

Neben den Grundtechniken zur Erzeugung und Verwendung von Referenzen habe ich Ihnen zwei der häufigsten Einsatzbereiche für Referenzen vorgestellt: als Argumente für Subroutinen (um die Struktur von Arrays und Hashes bei der Übergabe an Subroutinen zu erhalten) und zur Erzeugung von verschachtelten Datenstrukturen wie Arrays von Arrays und Arrays von Hashes. Schließlich haben Sie die `ref`-Funktion kennengelernt, die einen String zurückliefert, der Ihnen anzeigt, welche Art Daten die Referenz enthält.

Meinen Glückwunsch! Heute haben Sie die wirklich harte Arbeit mit diesem Buch abgeschlossen. Morgen untersuchen wir noch einige weitere Perl-Konzepte, die bisher noch nicht zur Sprache gekommen sind, und in Kapitel 21 beenden wir das Buch mit einigen Beispielen, die das bisher Gelernte praktisch umsetzen.

Fragen und Antworten

Frage:
Kann man Referenzen auf Referenzen erzeugen?

Antwort:
Aber selbstverständlich! Alles, was Sie dazu tun müssen, ist den Backslash- Operator zu verwenden, um die Speicherposition der Referenz zu erhalten. Denken Sie jedoch daran, dass Sie Referenzen auf Referenzen zweimal dereferenzieren müssen, um an die eigentlichen Daten zu gelangen.

Frage:
Ich versuche ein Array von Arrays mit Daten aus einer Datei zu füllen. Ich lese die Daten in ein einfaches Array und füge dieses Array dann in ein äußeres Array ein. Aber am Ende enthält das ganze Array nichts außer den zuletzt hinzugefügten Werten. Was mache ich falsch?

Antwort:
Das klingt, als wenn Sie ungefähr folgendes versuchen:

```
while (<>) {
    @input = split $_;
    @grossesarray = \@input;
}
```

Das Problem hierbei ist, dass alle Referenz auf `@input`, die Sie erzeugen, auf die gleiche Speicherposition zeigen. Der Inhalt von `@input` ändert sich also bei jedem Schleifendurchlauf, die Position jedoch bleibt die gleiche. Alle Referenzen zeigen deshalb auf die gleiche Stelle und haben als Wert die zuletzt dort abgelegten Dinge. Um das Problem zu lösen, können Sie

- Ihre temporäre `input`-Variable als `my`-Variable in der Schleife deklarieren. Damit erzeugen Sie jedesmal auch eine neue Speicherposition.
- für die Variable `@input` einen anonymen Array-Konstruktor verwenden (das heißt `[ @input ]`). Damit erzeugen Sie eine Referenz auf eine Kopie des Inhalts der Eingabe und somit auch jedesmal eine neue Speicherposition.

Frage:
Ich habe ein Array von Arrays erzeugt und mit der Anweisung `print "@meinarrary\n";` ausgegeben. Aber erhalten habe ich nur:

```
ARRAY(0x807f048) ARRAY(0x808a06c) ARRAY(0x808a0cc)
```

Was mache ich falsch?

Antwort:
Bei Arrays von Arrays ist eine Variableninterpolation nicht möglich. Ihr `print`-Befehl gibt nur die obere Ebene des Arrays aus - was im wesentlichen drei Referenzen sind. Die `ARRAY(...)`-Formulierungen sind lediglich die druckbaren Stringdarstellungen dieser Referenzen. Um ein verschachteltes Array (oder eine beliebige andere verschachtelte Datenstruktur) auszugeben, müssen Sie eine oder mehrere `foreach`-Schleifen verwenden und die Referenzen selbst dereferenzieren. Hier ein Beispiel dafür, wie sich dies realisieren ließe:

```
foreach (@meinarray) {
  print "( @$_ )\n";
}
```

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was sind Referenzen? Welche Vorteile bieten sie Ihnen?
2. Beschreiben Sie zwei Wege, wie man eine Referenz auf ein Array

Was noch bleibt

Allmählich nähert sich dieses Buch seinem Ende, und Sie haben sich mit den meisten Aspekten von Perl bereits vertraut gemacht oder zumindest hineingeschnuppert. Doch mit welchem Thema man sich auch beschäftigt, immer bietet Perl alternative Möglichkeiten und weitere Lösungswege, die ich aus Platzgründen nicht alle beschreiben konnte.

Betrachten Sie deshalb dieses Kapitel als Vertiefung zum ganzen Buch. Wir werden heute eine Reihe von Themen erörtern, die ich bisher nicht angesprochen habe, weil sie entweder zu komplex oder von zu peripherer Bedeutung für die einzelnen Kapitelinhalte waren. Diese Themen umfassen:

- Einzeilige Skripts, die von der Perl-Befehlszeile aufgerufen werden
- Eine Einführung in die objektorientierte Programmierung in Perl
- Die Formatierung der Ausgabe
- Sockets und einfache Netzwerkverbindungen
- POD-Dateien (***plain old documentation***)
- Quelltext zur Laufzeit auswerten
- Internationale Perl-Skripts
- Prüfen auf Sicherheitslecks mit dem Taint-Modus von Perl
- PerlScript für Windows
- Perl ausbauen
- Neue Elemente der Perl-Version 5.005

Einzeilige Perl-Skripts

Wenn Sie ein Perl-Skript schreiben, wird Ihre Vorgehensweise meistens so aussehen, wie bisher in diesem Buch beschrieben: Sie schreiben das Skript, speichern es in einer Datei und bedienen sich dann des Perl-Interpreters, um das Skript auszuführen. Manchmal jedoch haben Sie es mit einer wirklich einfachen Aufgabe zu tun oder aber mit einer einmaligen (oder zumindest sehr selten anfallenden) Aufgabe. Für Probleme dieser Art wäre es reine Zeitverschwendung, extra den Editor aufzurufen, nur um das eigentliche Skript zu schreiben. Abhilfe schaffen in diesen Fällen die einzeiligen Skripts von Perl, sogenannte Einzeiler.

Einzeiler sind Perl-Skripts, die Sie direkt an der Perl-Befehlszeile eingeben. Sie werden nirgends gespeichert. Haben Sie einen Fehler gemacht, müssen Sie sie neu eingeben.

Um einen Einzeiler in Perl zu erstellen, verwenden Sie die Option `-e` gefolgt von dem Skript in Anführungszeichen:

```
%  perl -e 'print "Dies ist ein Einzeiler\n";'
Dies ist ein Einzeiler
%
```

Unter Windows müssen Sie das ganze Skript in doppelte Anführungszeichen setzen und die doppelten Anführungszeichen im Skript selbst mit einem Backslash kennzeichnen:

```
C:\> perl -e "print \"Dies ist ein Einzeiler unter Windows\n\";"
```

Arbeiten Sie mit MacPerl, so haben Sie keine Befehlszeile. Doch keine Panik! Es gibt im Skriptmenü einen passenden Menübefehl, der die Befehlszeile simuliert und in den Sie Ihren Perl-Einzeiler eingeben können. Aber auch hier müssen Sie das Wort `perl`, die Option `-e` und das Skript in Anführungszeichen eingeben.

Wenn Ihr Skript mehrere Anweisungen enthält, so setzen Sie sie einfach alle in eine Zeile (in den meisten Unix-Shells können Sie einen Befehl auf mehrere Zeilen verteilen, indem Sie ein Backslash (\) an das Ende der Zeile setzen). Zur Erinnerung: Perl stört sich nicht an Zwischenraumzeichen (Whitespaces). Deshalb könnten Sie theoretisch einen unglaublich komplexen Einzeiler erzeugen, und Perl hätte trotzdem keine Probleme mit dessen Ausführung (vielleicht haben Sie auch schon den altbekannten Spruch vieler Perl-Programmierer gehört: »Ich erledige das alles in einer Zeile!« - was natürlich möglich ist, da man in Perl alles in einer Zeile machen kann, fragt sich nur, wie lang die Zeile dann wird).

Sehen Sie im folgenden einige Beispiele für einzeilige Perl-Skripts.

So drehen Sie alle Zeilen in einer Datei um:

```
% perl -e 'print reverse <>;' dateiname.txt
```

So geben Sie alle Zeilen einer Datei mit Zeilennummern aus:

```
% perl -e '$i=1;while(<>){print "$i: $_";$i++}' dateiname.txt
```

So entfernen Sie alle führenden Whitespace-Zeichen aus allen Zeilen einer Datei:

```
% perl -e 'while(<>){s/^\s+//g;print;}' dateiname.txt
```

So geben Sie eine Datei in Großbuchstaben aus:

```
% perl -e 'while(<>){print uc  $_;}' dateiname.txt
```

Da Skripts dieser Art häufig `while`-Schleifen mit `<>` und einer Form von `print` verwenden, gibt es in Perl dafür eine Kurzform. Die Option `-p` erlaubt Ihnen, den `while(<>)`-Teil wegzulassen und `$_` dennoch Zeile für Zeile auszugeben. Demzufolge könnte man das Beispiel zur Konvertierung in Großbuchstaben auch folgendermaßen schreiben (ob das wirklich besser ist, müssen Sie selbst entscheiden):

```
% perl -p -e '$_ = uc $_;' test.txt
```

Diese Zeile entspricht dem folgenden Code:

```
while (<>) {
   $_ = uc $_;
   print;
}
```

Sie können die beiden Befehlszeilenoptionen auch zusammenfassen, müssen aber die `p`-Option vor die `e`-Option stellen:

```
% perl -pe '$_ = uc $_;' test.txt
```

> *`-p` ist nicht die einzige Perl-Option, die Ihnen in Ihren Einzeilern Tipparbeit abnehmen soll. Die Option `-n` entspricht in etwa der Option `-p`, erzeugt aber keinen `print`-Teil (sie liefert Ihnen lediglich die `while(<>)`-Schleife). Die Option `-l` in Kombination mit `-p` oder `-n` trägt automatisch dafür Sorge, dass das Neue-Zeile-Zeichen am Ende jeder Zeile erst entfernt und dann beim Ausgeben wieder eingefügt wird. (Oder um genau zu sein, `-l` setzt den Wert der Variablen `$\`, dem Trennsymbol für die Ausgabedatensätze, auf `$/`, dem Trennsymbol für die Eingabedatensätze - in der Regel das Neue-Zeile-Zeichen). Alternativ können Sie `-l` einen oktalen Wert als zusätzliches Argument mitgeben, der das Zeichen repräsentiert, das Sie als Trennsymbol für die Ausgabedatensätze verwenden wollen.*

Einzeiler können extrem leistungsfähig sein, wenn sie zusammen mit der Option `-i` verwendet werden. Angenommen Sie haben einen Roman geschrieben und auf mehrere Dateien verteilt, die alle auf die Extension `.txt` enden. Jetzt wollen Sie alle Vorkommen des Namens »Stefan« durch den Namen »Fred« ersetzen.

Das einzeilige Perl-Skript, das dies leistet, alle Originaldateien ändert und von jeder Datei eine Sicherungsdatei anlegt, sieht wie folgt aus:

```
% perl -p -i.bak -e 's/Stefan/Fred/g' *.txt
```

Die Option `-i` schreibt direkt in Ihre Originaldateien, so dass die neuen Versionen die gleichen Dateinamen tragen wie die Originaldateien. Die alten Versionen dieser Dateien (in denen Ihr Held noch Stefan hieß) werden in Dateien mit der Extension `.txt.bak` gespeichert. Auf diese Weise haben Sie immer noch die Möglichkeit, auf die alten Dateien zuzugreifen, für den Fall, dass Sie vielleicht nicht Stefan, sondern Albert in Fred umbenennen möchten. Seien Sie vorsichtig mit diesem Perl-Befehl - testen Sie ihn zuerst einmal an einer einzigen Datei, ohne Änderungen vorzunehmen. So können Sie sicherstellen, dass Ihr Einzeiler auch korrekt funktioniert. Sonst kann es Ihnen passieren, dass Sie am Ende eine ganze Reihe von `.bak`-Dateien restaurieren müssen.

Objektorientierte Programmierung

Eines der größeren Themen, auf das ich in diesem Buch nicht näher eingegangen bin, betrifft die Verwendung von Perl für die objektorientierte Programmierung, kurz OOP (wäre der Titel dieses Buches ***Perl in 25*** *1/2,* hätten wir es vielleicht geschafft). Glücklicherweise fällt die Einarbeitung in die objektorientierte Programmierung unter Perl nicht allzu schwer, da Perl bekannte Elemente wie Pakete, Subroutinen und Referenzen verwendet, um eine objektorientierte Programmierumgebung zu schaffen. Auf diese Weise können Sie, wenn Sie sich schon etwas mit OOP auskennen, direkt - unter Beachtung bestimmter Regeln - mit der Programmierung beginnen. Wer in der objektorientierten Programmierung noch unerfahren ist, muss sich zuerst etwas Hintergrundwissen aneignen. Irgendwelche neuen größeren Perl-Features brauchen Sie nicht zu lernen, um Ihre Kenntnisse in objektorientierter Programmierung direkt umzusetzen.

Erste Schritte und ein wenig mehr

Wenn Sie mit objektorientierter Programmierung noch nicht vertraut und daran interessiert sind, objektorientiert zu programmieren, sollte Ihr erster Schritt darin bestehen, dass Sie sich die Grundkonzepte anschauen. Objektorientierte Programmierung bedeutet einfach, dass Sie das gleiche Programmierproblem aus einem anderen Blickwinkel betrachten. Alles, was Sie bisher in diesem Buch über Syntax und guten Programmierstil gelernt haben, behält seine Gültigkeit. Der Unterschied liegt darin, wie Ihr ganzes Skript organisiert ist und wie es sich verhält.

Der Grundgedanke, der der objektorientierten Programmierung zugrunde liegt, ist, dass Ihr Skript keine Sammlung von nacheinander ausgeführten Anweisungen und Subroutinen ist, sondern eine Sammlung von Objekten, die in einer vordefinierten Art und Weise miteinander interagieren. Jedes Objekt hat eine definierte Erscheinung oder Status (Variablen) und einen definierten Satz an Verhaltensweisen (Subroutinen, in OOP auch Methoden genannt). Objekte erhalten diese Verhaltens- und Statusschablonen von einer Klassendefinition. Diese Klassendefinition wiederum erbt (verwendet) oftmals die Merkmale von einer oder mehreren anderen Klassen. Wenn Sie ein objektorientiertes Skript erstellen, erzeugen Sie eine oder mehrere eigene Klassen, die Klassen und Objekte von anderen Quellen (in der Regel Module) importieren und verwenden. Wenn Ihr Perl-Skript ausgeführt wird, werden aus den verschiedenen Klassen Laufzeitobjekte erstellt, die untereinander ihre Variablen ändern und ihre jeweiligen Subroutinen aufrufen, um am Ende eine Art von Ergebnis zu liefern.

Hat Sie der obige Absatz zu Tode erschreckt, dann geraten Sie bitte nicht in Panik. In Perl können Sie sich langsam mit OOP anfreunden und bereits die ersten OOP- Konzepte nutzen, ohne gleich alle Konzepte und Techniken kennen zu müssen. Und außerdem gibt es im Netz eine Fülle von OOP-Tutorials, die Ihnen helfen, die Theorie und die verschiedenen Konzepte der objektorientierten Programmierung zu verstehen. Die folgenden Vorschläge sind ganz gute Ausgangspunkte:

- Die ***perltoot***-Manpage (objektorientiertes Tutorial) wird mit Perl ausgeliefert. Es enthält ein einführendes Tutorial zur objektorientierten Programmierung und vermittelt wichtiges Hintergrundwissen zu Perl. Danach kann man mit Hilfe der Manpages ***perlobj*** (Perl-Objekte) und ***perlbot*** (»back of tricks« -Trickkiste) die verbliebenen Wissenslücken schließen.
- Unter `http://www.progsoc.uts.edu.au/~geldridg/cpp/` finden Sie eine Reihe von Links zu verschiedenen Stellen mit OOP-bezogenen Informationen - nicht unbedingt Perl-spezifisch, aber mit einigen einführenden Tutorials.
- Viele Bücher, die anderen OOP-Sprachen wie Java oder C++ gewidmet sind, enthalten Kapitel mit allgemeinem Hintergrundwissen zu OOP. Falls Sie eines dieser Bücher schon irgendwo herumliegen haben oder sich eines von einem Freund borgen können, schlagen Sie diese Kapitel einmal nach. Das kann von

Nutzen sein kann (das entsprechende Kapitel finden Sie auch unter `www.typerl.com`). OOP-Konzepte lassen sich in der Regel von einer Sprache auf eine andere übertragen.

Sollten Sie immer noch verzagt sein, ist auch das noch kein Grund, in Panik zu verfallen. Zwar wird OOP von vielen als der Trend der Zukunft betrachtet, doch wenn Sie zufrieden damit sind, mit dem guten alten Perl weiterzuarbeiten, und niemand von Ihnen erwartet, sich in OOP einzuarbeiten, spricht nichts dagegen, sich an das Altbewährte zu halten. Denn schließlich führen viele Wege nach Rom.

Die Grundlagen (für Leser, die bereits über OOP-Kenntnisse verfügen)

Sie wissen also bereits, was ein Objekt ist und wie es sich zu einer Klasse verhält. Und Sie kennen die Begriffe Instanzvariable, Methode, Vererbung, Konstruktor, Destruktor und Kapselung. Dann lassen Sie uns diese Terminologie auf Perl übertragen.

Klassen, Objekte und Objektreferenzen

In Perl ist eine Klasse ein Paket, und der Namensbereich, der vom Paket definiert wird, definiert die Kapselung und den Gültigkeitsbereich für diese Klasse. Variablen, die in diesem Paket definiert werden, sind Klassenvariablen (statische Variablen). Instanzvariablen werden normalerweise dadurch erzeugt, dass man eine Referenz auf einen Hash erzeugt und als Schlüssel den Namen der Instanzvariablen verwendet. Klassen- und Instanzmethoden werden beide als Subroutinen definiert. Der einzige Unterschied zwischen einer Methode und einer regulären Subroutine ist der, dass eine Methode als erstes Argument einen Klassennamen (für Klassenmethoden) oder eine Objektreferenz (für Instanzmethoden) erwartet.

Eine Objektreferenz? Gut aufgepaßt. Das ist das einzig Neue, das Sie, soweit es Perl betrifft, lernen müssen, um in Perl objektorientiert programmieren zu können. In Perl ist ein Objekt eine Referenz auf eine Datenstruktur (in der Regel ein anonymer leerer Hash), die mit einer speziellen Markierung versehen wurde, so dass sie sich wie ein Objekt verhält und weiß, zu welcher Klasse sie gehört. Um etwas als ein Objekt zu markieren, verwenden Sie die vordefinierte `bless`-Funktion. Diese Funktion liefert eine Referenz auf ein Objekt zurück, die Sie dann wiederum, wie jede andere Referenz auch, einer Skalarvariablen zuweisen können.

Normalerweise verwenden Sie `bless` in dem Konstruktor Ihrer Klasse. Der Konstruktor ist eine ganz normale Subroutine, die per Konvention `new` genannt wird. Die `bless`-Funktion übernimmt zwei Argumente: das, wofür Sie eine Objektreferenz erstellen wollen, und den Namen einer Klasse. Eine einfache Klassendefinition könnte damit wie folgt aussehen:

```
package MeineKlasse;
sub new {
   my $klassenname = shift;
   my $selbst = {};
   return bless $selbst, $klassenname;
}
```

In diesem Beispiel wird davon ausgegangen, dass die Klassenmethode `new` mit einem Argument aufgerufen wird: einem Klassennamen. In `new` selbst erzeugen wir einen leeren, anonymen Hash, markieren ihn mittels `bless` mit dem aktuellen Klassennamen und liefern eine Referenz auf den Hash zurück. Beachten Sie, dass `new` eine Klassenmethode ist und dass Klassenmethoden immer als erstes Argument den Namen der Klasse erhalten.

Sie können `bless` auch mit nur einem Argument verwenden (dem Element, das mit `bless` markiert werden soll), und Perl wird den Namen der aktuellen Klasse automatisch als zweites Argument verwenden. Aufgrund der Art und Weise wie Perl jedoch mit vererbten Methoden umgeht, kann die Verwendung von nur einem Argument zu falschen Ergebnissen führen, wenn eine andere Klasse Ihren Konstruktor erbt. Allgemein möchte ich Ihnen nahelegen, sich daran zu gewöhnen, `bless` mit zwei Argumenten zu verwenden, und den Klassennamen der Argumentliste der Methode zu entnehmen.

Um ein Objekt zu erzeugen und zu verwenden (aus dieser Klasse oder aus irgendeiner anderen Klasse), rufen Sie

die Methode `new` zusammen mit einem Paket- (Klassen-) namen auf. Sie können dies in der gleichen Datei erledigen, in der auch Ihre Klassendefinition steht, solange Sie vorher in `package main` wechseln:

```
package main;
$obj = new MeineKlasse;
```

Die Skalarvariable `$obj` enthält dann eine Referenz auf ein Objekt, das durch die Klasse `MeineKlasse` definiert ist. Alternativ können Sie auch die Pfeilsyntax (mit `->`) verwenden, um den `new`-Konstruktor aufzurufen:

```
$obj = MeineKlasse->new();
```

Beide Möglichkeiten, `new` aufzurufen, führen zum gleichen Ergebnis. Wenn Ihr `new`- Konstruktor jedoch neben dem Klassennamen noch weitere Argumente benötigt, ist es oft leichter, das letztere Format anzuwenden:

```
$obj = MeineKlasse->new(12, 240, 15);
```

Um die Methoden aufzurufen, die in Ihrem neuen Objekt definiert sind, müssen Sie die Objektreferenz dereferenzieren. Dafür eignet sich die ->-Syntax besonders gut:

```
$obj->eineSubroutine('foo','bar');
```

Alternativ gibt es für Methoden eine Syntax, die stärker an die normale Funktionsaufruf-Syntax angelehnt ist. Dabei muss das erste Argument zu der Methode der Klassenname oder eine Objektreferenz sein:

```
eineSubroutine $obj, 'foo', 'bar';
```

Da das erste Argument hier eine Objektreferenz ist, wird Perl die Referenz für Sie dereferenzieren und die richtige Methode aufrufen.

Sie können eine Methode auch wie eine Funktion in einem Paket aufrufen (`(Myclass::eineSubroutine(...)`). Diese Vorgehensweise empfiehlt sich aber nur, wenn Sie genau wissen, was Sie machen. Damit untergraben Sie nämlich die OOP- Eigenschaften der Klasse und verlieren die Fähigkeit, an eine Methodendefinition zu gelangen, die Ihnen über Vererbung zur Verfügung steht. Im nächsten Abschnitt »Instanzvariablen« erzähle ich Ihnen noch mehr zum Definieren und Aufrufen von Referenzen.

Instanzvariablen

Wenn Sie wie oben beschrieben Objekte erzeugen, verwenden Sie einen anonymen Hash als das, was Sie mit `bless` als Objekt markieren. Warum gerade einen Hash? Weil ein Hash zum Speichern und Zugreifen auf Instanzvariablen genutzt werden kann und Sie jedesmal eine neue Version dieser Variablen erhalten. Auf diese Weise werden alle internen Objektstatusinformationen in Perl-Objekten gespeichert.

In manchen OOP-Sprachen wird zwischen Instanzvariablen und Klassenvariablen unterschieden (letztere werden manchmal auch als statische Daten bezeichnet). Perl kennt an sich keine Klassenvariablen. Sie können zwar jederzeit globale Paketvariablen erstellen, die die Funktion von Klassenvariablen haben, und auf diese dann über die normale Paketsyntax (`$MeineKlasse::meineKlassenVar`) zugreifen, allerdings werden diese Variablen nicht vererbt und können von jedem Benutzer Ihrer Klasse beliebig geändert werden. Versuchen Sie, die Verwendung von Klassenvariablen zu umgehen und statt dessen Instanzvariablen zu verwenden.

Im allgemeinen werden die Instanzvariablen einer Klasse in dem Konstruktor der Klasse definiert und initialisiert. Häufig werden Sie dabei Argumente an `new` übergeben, die die Anfangswerte dieser Klasse bilden sollen:

```
#!/usr/bin/perl -w
package Rectangle;
sub new {
   my ($classname, $w, $h) = @_;
   my $self = {};
```

```
   $self->{Width} = $w;
   $self->{Height} = $h;
   return bless $self, $classname
}
sub area {
   my $self = shift;
   return $self->{Width} * $self->{Height};
}
package main;
$sq = new Rectangle (12, 20);
print "Fläche: ", $sq->area(), "\n";
```

In diesem Beispiel haben wir einen Klassenkonstruktor verwendet, der zwei zusätzliche Argumente übernimmt - eine Breiten- und eine Höhenangabe. Auf der Grundlage dieser Argumente erzeugt der Konstruktor ein `Rectangle`-Objekt (Rechteck) und speichert die Werte in dem Hash des Objekts unter den Schlüsseln `Width` und `Height`. Des weiteren haben wir eine Methode namens `area` erzeugt, die die aktuellen Werte dieser Instanvariablen multipliziert und das Ergebnis zurückliefert.

Im allgemeinen sieht die objektorientierte Programmierung mit Instanzvariablen in Perl so aus, dass Sie zum Schreiben und Lesen dieser Variablen Methoden definieren und verwenden, anstatt die Werte der Variablen durch direkte Zuweisung zu ändern. Dies ist insbesondere bei der Vererbung von Vorteil und sorgt dafür, dass Ihre Klasse oder Ihr Objekt eine »reinere« objektorientierte Schnittstelle erhält. Sie können aber auch mit der normalen Dereferenzierungssyntax auf die Instanzvariablen zugreifen:

```
# kein objektorientierter Variablenzugriff
print "Width: $sq->{Width}\n";
print "Height: $sq->{Height}\n";
```

Vererbung

Wie jede ordentliche objektorientierte Programmiersprache erlaubt auch Perl die Vererbung von Klassen, die es Klassen erlaubt, die Definitionen anderer Klassen zu verwenden und zu erweitern. Wenn die Klasse B das Verhalten von Klasse A erbt, wird Klasse A als Basis- oder Superklasse und Klasse B als abgeleitete oder Subklasse bezeichnet.

Um anzuzeigen, dass eine Klasse von einer anderen Klasse erbt, verwenden Sie das spezielle Array `@ISA`. Das Array `@ISA` legt fest, in welchen Klassen nach Methodendefinitionen gesucht wird, wenn die Definition einer aufgerufenen Methode nicht in der aktuellen Klasse existiert. Wenn Sie zum Beispiel eine Superklasse `Katzen` hätten, würde die Subklasse `Tiger` wie folgt aussehen:

```
package Tiger;
@ISA = qw( Katzen );
...
```

Wenn eine Klasse nur von einer Superklasse erbt, ist der Aufruf von `qw` nicht unbedingt erforderlich, erleichtert aber das Hinzufügen weiterer Superklassen, falls man sich später für eine Mehrfachvererbung entscheidet:

```
package Hauskatze;
@ISA = qw( Katze Haustier );
...
```

Wenn für eine Methode, die über ein bestimmtes Objekt aufgerufen wird, in der aktuellen Klassendefinition keine Definition gefunden wird, geht Perl das `@ISA`-Array durch und sucht in jeder der aufgelisteten Superklassen nach der betreffenden Methodendefinition. Gibt es zu einer Superklasse wieder weitere Superklassen werden diese ebenfalls durchsucht, bevor die nächste Superklasse im `@ISA`-Array an die Reihe kommt. Nehmen wir an, die Methode `fressen` wird für ein Objekt der Klasse `Hauskatze` aufgerufen. Zuerst wird in `Hauskatze` selbst nach einer Definition gesucht, dann in `Katze` und dann in allen Superklassen von `Katze` (falls vorhanden) und in deren Superklassen, bevor die Suche nach der Definition in `Haustier` fortgeführt wird. Die Definition, die zuerst gefunden wird, wird auch verwendet.

Perl vererbt nur Methoden. Um Instanzvariablen zu »vererben«, können Sie geerbte Konstruktormethoden verwenden, um einen einzigen Hash von Instanzvariablen einzurichten, der die Instanzvariablen sämtlicher Superklassen aufnimmt - plus denen, die für die aktuelle Klasse definiert wurden. (Ein Beispiel hierfür finden Sie in

der Manpage ***perlobj***.)

Methoden definieren

Das Definieren von Methoden kann in drei allgemeine Kategorien unterteilt werden:

- Es werden neue Methoden definiert, die der aktuellen Klasse angehören.
- Es werden Methoden definiert, die Methoden überschreiben, welche in Superklassen definiert worden sind.
- Es werden Methoden definiert, die das Verhalten von Methoden einer Superklasse erweitern.

Methoden, die von der Vererbung keinen Gebrauch machen, werden als gewöhnliche Subroutinen in der Klassen-(Paket-)Definition aufgesetzt. Die einzige Besonderheit ist das erste Argument zu dieser Subroutine, das festlegt, ob die Methode eine Klassen- oder eine Instanzmethode ist.

Im Falle von Instanzmethoden ist das erste Argument eine Objektreferenz. Normalerweise wird bei Methoden dieser Art zuerst einmal die Referenz aus der Argumentliste extrahiert und in einer Skalarvariablen gespeichert (`$self` ist ein sehr geläufiger Variablenname, er kann aber auch beliebig anders lauten):

```
sub meineMethode {
   my $self = shift;
...
}
```

Bei Klassenmethoden ist das erste Argument einfach der Name der Klasse - nichts Besonderes, nur ein String. Auch hier werden Sie für gewöhnlich das Argument extrahieren und es in einer Skalarvariablen speichern.

Wie sieht es jetzt aber mit Methoden aus, die je nach Argument entweder eine Klassen- oder eine Instanzmethode sein können. Für solche Methoden brauchen Sie einen Weg, wie Sie im Rumpf der Methode feststellen können, ob es sich bei dem ersten Argument um ein Objekt oder eine Klasse handelt. Sehr hilfreich ist in diesem Zusammenhang die Funktion `ref`:

```
sub klasseOderInstanz {
  my $arg = shift;
  if (ref($arg)) {
     print "Argument ist ein Objekt\n";
  } else {
     print "Argument ist ein Klassenname\n";
  }
}
```

Beachten Sie, dass die `ref`-Funktion, wenn sie mit einer Objektreferenz als Argument aufgerufen wird, den Namen der Klasse zurückliefert, von der das Objekt eine Instanz darstellt.

Wenn Sie eine Methode aufrufen, die in der aktuellen Klasse definiert ist, wird diese Methode ausgeführt - auch wenn es weiter oben in der Vererbungskette eine gleichnamige Methode gibt. Dies ist der Weg, wie Sie Methoden definieren, die bestehende Methoden überschreiben - einfach definieren und fertig.

Wenn Sie das Verhalten einer Methode einer übergeordneten Klasse ergänzen wollen, anstatt es komplett zu überschreiben, verwenden Sie das Paket `SUPER`, das Perl anweist, in den in `@ISA` aufgeführten Klassen nach einer Methodendefinition zu suchen:

```
sub berechnen {
   my $self = shift;
   my $sum = $self->SUPER::berechnen(); # zuerst die Superklassen
   foreach (@_) {
      $sum += $_;
   }
   return $sum;
}
```

SUPER wird häufig in geerbten Konstruktoren (new-Methoden) verwendet. Mit SUPER können Sie einen Konstruktor erzeugen, der die Vererbungskette hinaufläuft, um sicherzustellen, dass das aktuelle Objekt über alle Instanzvariablen und Initialisierungen verfügt, die es benötigt.

Ich habe Ihnen bereits gezeigt, wie Sie Konstruktormethoden mit `bless` erzeugen. Sie können aber auch Destruktormethoden erzeugen, die ausgeführt werden, wenn alle Referenzen auf das Objekt gelöscht sind und das Objekt nur noch darauf wartet, von der Speicherbereinigung entfernt zu werden. Sie erzeugen eine Destruktormethode, indem Sie eine Subroutine namens `DESTROY` in Ihrer Klasse definieren:

```
sub DESTROY {
   print "Objekt löschen\n";
   ...
}
```

Selbstladende Methoden

In Verbindung mit den Methodenaufrufen gibt es in Perl eine nette Besonderheit, die ich Ihnen nicht vorenthalten möchte: die sogenannten selbstladenden Methoden. Selbstladende Methoden stellen eine Art letzte Zuflucht dar, wenn Sie für ein Objekt eine Methode aufrufen, für die Perl keine passende Definition finden kann. In diesem Fall versucht Perl, eine Methode namens `AUTOLOAD` aufzurufen. Die Paketvariable `$AUTOLOAD` enthält den Namen der Methode, die aufgerufen wurde (einschließlich des ursprünglichen Klassennamens, für den sie aufgerufen wurde). Sie können diese Informationen dann nutzen, um festzulegen, wie ansonsten unbekannte Methodenaufrufe verarbeitet werden sollen. Die Manpage ***perlobj*** enthält ein hervorragendes Beispiel dafür, wie man mit Hilfe einer selbstladenden Methode Zugriffsmethoden für Instanzvariablen simulieren kann, ohne dass man tatsächlich für jede Variable eine eigene Methode definieren müßte. Im folgenden finden Sie ein einfaches Beispiel für eine selbstladende Methode, die diese Art von Verhalten zeigt:

```
package Hauskatze;
@ISA = qw( Katze, Haustier );
sub new {
  my $classname = shift;
  return bless {}, $classname;
}
sub AUTOLOAD {
   my ($self,$arg) = @_;
   my $name = $AUTOLOAD;
   my $iv =~ s/.*://;  # Paket-Teil aus Namen entfernen
   if ($arg) {  # Wert setzen
      $self->{$iv} = $arg;
      return $arg;
   } else {     # kein Argument, Rückgabewert zurückliefern
      return $self->{$iv};
   }
}
package main;
my $cat = new Hauskatze;
$cat->color("Grau");
print "Meine Katze ist $cat->color()\n";
```

In diesem Beispiel hat die Klasse `Hauskatze` keine Methode namens `color` (und wir gehen davon aus, dass die übergeordneten Klassen ebenfalls keine Methode dieses Namens haben). Wenn die Methode `color` aufgerufen wird, ruft Perl daher `AUTOLOAD` auf. In der Definition von `AUTOLOAD` erhalten wir den Namen der aufgerufenen Methode über die Variable `$AUTOLOAD`, entfernen den Paketnamen zu Beginn und verwenden den übriggebliebenen Methodennamen dann als Namen der Instanzvariablen. Wurde die Methode mit einem Argument aufgerufen, weisen wir diesen der Instanzvariablen zu. Andernfalls geben wir einfach den aktuellen Wert aus (es obliegt dem Aufrufer, undefinierte Instanzvariablen zu handhaben). Sie können diese `AUTOLOAD`-Methode genau so einfach auch für alle anderen Instanzvariablen verwenden - `name`, `alter`, `temperament`, `lieblingsessen` und so weiter.

Technisch gesehen, stellen `AUTOLOAD`-Methoden nicht unbedingt die letzte Zuflucht dar. Zusätzlich zu `AUTOLOAD` gibt es noch die Klasse `UNIVERSAL`. `UNIVERSAL` ist eine Art globale Superklasse. In ihr können Sie Ihre Zufluchtsmethoden definieren.

Ein Beispiel: Objektorientierte Module in der Praxis

In Perl ist OOP kein Dogma, sondern eine optionale Möglichkeit. Sie können OOP nur sporadisch einsetzen oder die Sache gründlich angehen und OOP konsequent überall verwenden. Das hängt ganz davon ab, was am leichtesten ist und wie fanatisch Sie den Grundprinzipien der objektorientierten Programmierung anhängen.

In vielen Fällen nutzt man die Vorzüge der objektorientierten Programmierung am einfachsten dadurch, dass man die verschiedenen objektorientierten CPAN-Module in die eigenen Skripts einbindet - ohne dass man dazu notwendigerweise die eigenen Skripts als einen Satz von Objekten strukturieren müßte. Erinnern wir uns noch einmal an die CGI-Skripts und das Modul `CGI.pm` von Tag 16, »Perl für CGI-Skripts«. Dieses Modul ist so geschrieben, dass seine Subroutinen sowohl als einfache Subroutinen als auch als objektorientierte Methoden verwendet werden können. Betrachten wir eine der Übungen, die wir am Ende der Lektion gemacht haben - Übung 1, ein CGI-Skript, das nur die ihm übergebenen Schlüssel und Werte ausgibt - und setzen wir das CGI-Modul diesmal objektorientiert ein.

Das Originalskript sehen Sie noch einmal in Listing 20.1

Listing 20.1: paare1.pl

```
 1: #!/usr/bin/perl -w
 2: use strict;
 3: use CGI qw(:standard);
 4:
 5: my @keys = param();
 6:
 7: print header;
 8: print start_html('Hallo!');
 9: print "<H1>Schlüssel/Wert-Paare</H1>\n";
10: print "<UL>\n";
11:
12: foreach my $name (@keys) {
13:     print "<LI>$name = ", param($name), "\n";
14: }
15: print "</UL>\n";
16:
17: print end_html;
```

Wir verwenden in diesem Skript vier Subroutinen aus dem CGI-Moduls: `param` (Zeilen 5 und 13), die uns die gesamte Liste der verfügbaren Schlüssel und deren Werte liefert, `header` (Zeile7), um einen CGI-Header auszugeben, `start_html` (Zeile 8), um den oberen Teil einer HTML-Datei auszugeben, und `end_html` (Zeile 17), um den unteren Teil einer HTML-Datei auszugeben. Im obigen Beispiel haben wir diese Subroutinen als reguläre Subroutinen verwendet.

Das CGI-Modul kann aber auch als eine objektorientierte Klasse verwendet werden. Erzeugen Sie eine Instanz dieser Klasse, und Sie können die Subroutinen verwenden, als ob es sich dabei um Methoden handeln würde (was sie ja auch eigentlich sind). Alles was hierzu nötig ist, ist eine zusätzliche Codezeile und eine etwas abgeänderte Syntax für die Subroutinen (siehe Listing 20.2).

Listing 20.2: Die objektorientierte Version von paare1.pl

```
 1: #!/usr/bin/perl -w
 2: use strict;
 3: use CGI qw(:standard);
 4:
 5: my $obj = CGI->new();
 6: my @keys = $obj->param();
 7:
 8: print $obj->header();
 9: print $obj->start_html('Hallo!');
10: print "<H1>Schlüssel/Wert-Paare</H1>\n";
11: print "<UL>\n";
12:
13: foreach my $name (@keys) {
14:     print "<LI>$name = ", $obj->param($name), "\n";
15: }
```

```
16: print "</UL>\n";
17:
18: print $obj->end_html();
```

Sehen Sie die Unterschiede? Eigentlich sind es nur zwei:

- In Zeile 5 instantiieren wir unser CGI-Objekt und speichern eine Referenz darauf in der Variablen `$obj`. Entsprechend den Perl-Konventionen heißt der Objektkonstruktor für die CGI-Klasse `new`.
- Alle unsere Subroutinen - `param`, `start_html`, `header` und `end_html` - werden als Objektmethoden mittels der Dereferenzierungssyntax (das Objekt, `->` und der Name der Methode) aufgerufen.

Und das Endergebnis? Es gibt keinen Unterschied zu der Nicht-OOP-Version. Das CGI-Modul ist so geschrieben, dass es gleichermaßen gut als normale Sammlung von Subroutinen oder als objektorientierte Klasse verwendet werden kann.

Beachten Sie, dass es sich bei diesem Beispiel nicht um ein »reines« OOP-Programm handelt. Wir haben hier keine eigenen Klassen erzeugt. Ein »wahres« OOP-Skript würde den Großteil des Codes in einer eigenen Klasse unterbringen und dort das CGI- Modul nutzen. Im Hauptteil würde kaum mehr als die Instantiierung der Klasse und die Aufrufe von ein oder zwei Methoden stehen, um das Skript in Gang zu setzen. Das ist das Gute an OOP in Perl. Sie müssen nur so viel objektorientierte Programmierung einsetzen wie nötig. Im Gegensatz zu anderen strengeren OOP-Sprachen, müssen Sie Code, der für OOP ungeeignet erscheint, nicht unbedingt in Objekte pressen.

Formate

In Anbetracht der Tatsache, dass Perl ein Akronym für ***Practical Extraction and Report Language*** (Praktische Extraktions- und Reportsprache) ist, mag es Sie überraschen, dass bisher in diesem Buch zwar viel über das Extrahieren geschrieben wurde, aber kaum etwas über das Protokollieren. Für letzteres gibt es die Formate: zur Ausgabe von formatierten textbasierten Protokollen über Informationen, die Sie vorher gelesen, verarbeitet oder anderweitig malträtiert haben.

Warum wird dieses Thema erst jetzt gegen Ende des Buchs angesprochen? Ein großer Teil der Ausgabe von Perl-Skripts erfolgt heutzutage - als Ergebnis von CGI-Skripts - im HTML-Format und nicht mehr im Nur-Text-Format. Wenn Sie also davon ausgehen, dass Sie Perl hauptsächlich für CGI einsetzen werden, ist HTML für die Ausgabe besser geeignet als reiner Text. Wenn Sie aber vornehmlich mit einfacher Textausgabe arbeiten, sollten Sie sich unbedingt mit den Formaten befassen, vor allem auch unter dem Aspekt, dass `print`-Anweisungen ziemlich umständlich sind, um eine ordentliche und übersichtliche Präsentation Ihrer Ausgabe sicherzustellen.

Die Idee hinter den Formaten ist die, dass Sie mit Hilfe der `format`-Funktion eine spezielle Schablone (Template) deklarieren, die festlegt, wie Ihre Ausgabe auszusehen hat, und dann mit der `write`-Funktion und einem Datei-Handle diese Schablone auf die Daten anwenden. Als Ergebnis werden die Platzhalter in der Schablone durch einen Satz von Werten aus der gegebenen Datenmenge ersetzt. Die Schablone legt die Position einer jeden Datenspalte sowie ihre Breite und Ausrichtung fest. Zusätzlich steht Ihnen eine Reihe von speziellen Perl-Variablen für die Größe einer Seite (in Zeilen), den Header (Kopfbereich), der oben auf jeder Seite erscheinen soll, und die aktuelle Seitenzahl zur Verfügung. Insgesamt können Sie so für jede beliebige Art von Daten Protokolle mit Ganzseiten-Textformatierung erzeugen.

Schablonen werden mit Hilfe der Funktion `format` deklariert. Schablonennamen verhalten sich wie Subroutinen- oder andere Variablennamen, sie unterliegen den gleichen Regeln zur Namensgebung und geraten nicht mit den Namen anderen Arten von Variablen in Namenskonflikt. Eine `format`-Deklaration enthält einen Namen, eine Reihe von ***Bildzeilen***, die festlegen, wie das Format aussehen soll, und einen Satz an Variablen, die die Daten festlegen, die in die Schablone passen sollen. Das Format endet mit einem einzelnen Punkt in einer eigenen Zeile. Im folgenden sehen Sie ein Beispiel für ein einfaches Format, das eine Reihe von Firmen mit Börsenticker- Symbolen, Höchstpreisen, Niedrigstpreisen und Abschlußpreisen ausgibt:

```
format STDOUT =
@<<<<<<<<<<<<<<<<<<<@||||@|||||@|||||@|||||
$name,              $sym,$high,$low, $current
.
```

Bildzeilen enthalten Symbole für einfache, einzeilige Spalten (@) oder einfache, gefüllte Textspalten (^). Jede Zeile

kann auf vielfältige Weise mit verschiedenen Symbolen (<, |, >, #) ausgerichtet werden (links, zentriert, rechts, numerisch). Die Variablen geben an, welche Daten an den einzelnen Positionen im Format stehen sollen, wenn die Zeile ausgegeben wird. Wenn Sie `use strict` verwenden, müssen Sie diese Variablen vorher mit `my` deklarieren, bevor Sie sie in dem Format verwenden können.

Um die formatierten Daten an ein Datei-Handle auszugeben, verwenden Sie die `write`-Funktion:

```
write STDOUT;
```

Beachten Sie, dass der Name des Formats mit dem Namen des Datei-Handles, an das die Ausgabe erfolgt, übereinstimmen muss (obwohl Sie dies mit dem Modul `FileHandle` ändern können). In unserem Fall habe ich das Standard-Datei-Handle `STDOUT` verwendet.

Betrachten wir noch ein Beispiel. Unsere Daten sind eine einfache Aufgabenliste, wobei der Datensatz aus den folgenden Feldern besteht: Zeitpunkt der Fertigstellung, Priorität der Aufgabe (1 bis 5), ob erledigt oder nicht (`0` oder `1`) und einer Textbeschreibung der Aufgabe (ein String von ungefähr 40 Zeichen oder weniger). Diese Daten sind in einer Datei gespeichert und werden in ein Array von Hashes (`@data`) eingelesen (siehe gestrige Lektion). Ihr Skript soll unter anderem dazu in der Lage sein, die Aufgabenliste nach Prioritäten sortiert in einem ansprechenden Format auszugeben. Eine entsprechende Ausgabe könnte zum Beispiel wie folgt aussehen:

```
Fertig?   Fälligkeit    Priorität  Beschreibung
-----------------------------------------------------------
   Nein   23.10.1998    1          Kapitel 21 beenden
   Nein   31.10.1998    1          Kapitel 20 beenden, überarbeiten
   Nein   05.01.1999    2          Treffen mit Fred zum Essen
   Ja     05.12.1998    3          Punkt 3
   Nein   10.01.1999    5          Pyramide bauen
```

Um diese Daten ohne Formate auszugeben, würden Sie eine `foreach`-Schleife aufsetzen, die jeden Datensatz durchläuft, daraus eine Zeile aufbaut und diese dann an den Datei-Handle ausgibt. Da Sie keine Formate verwenden, müssen Sie selbst die Abstände zwischen den Daten kontrollieren, um sicherzustellen, dass die Daten korrekt ausgerichtet werden. Mit Formaten ist die ordentliche Ausrichtung der Daten wesentlich einfacher.

Um die Daten zu formatieren und auszugeben, überlegen wir zuerst, wie die Ausgabe mit der `format`-Deklaration aussehen soll. Genau genommen verwenden wir zwei `format`-Deklarationen: eine einfache Nur-Text-Deklaration für den Header und eine für die Datenzeilen. Außerdem deklarieren wir schon einmal unsere Formatvariablen, damit uns `use strict` keinen Ärger macht:

```
my ($done, $date, $prior, $desc); # Formatvariablen;
format STDOUT_TOP =
  Fertig?  Fälligkeit    Priorität  Beschreibung
-------------------------------------------------------------
.
format STDOUT =
   @||||     @<<<<<<<<<     @|      @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  $done,    $date,     $prior,   $desc
.
```

Die erste `format`-Deklaration (für `STDOUT_TOP`) ist eine besondere Art von Format, die einen Header für die Seite definiert. Header haben per definitionem den gleichen Namen wie das normale Format (hier `STDOUT`) plus dem angehängten `_TOP`. Sie könnten diesen Test als normale `print`-Anweisung ausgeben, aber formatierte Header haben den Vorteil, dass sie bei einem mehrseitigen Ausdruck auf jeder Seite automatisch gleich ausgegeben werden. Ich persönlich finde, dass sie das Formatieren stark vereinfachen, deshalb nehme ich sie meistens mit auf.

Kommen wir jetzt zu dem interessanteren, dem zweiten Format, das unsere eigentliche `format`-Deklaration enthält. Hier definieren wir die Breite und Ausrichtung all unserer Spalten: `done` und `prior` sind zentriert (beachten Sie das |-Zeichen), `date` und `desc` sind links ausgerichtet (<-Zeichen). Die Anzahl der Ausrichtungszeichen legt die Anzahl der Zeichen für die Spalte fest. Weisen die Daten mehr Zeichen auf, als die Spalte `Zeichen` aufnehmen kann, werden diese abgeschnitten.

Nachdem die Formate definiert sind, besteht der zweite Schritt darin, die Daten in einer Schleife zu durchlaufen und auszugeben. Doch zuerst sortieren wir die Datensätze nach ihrer Priorität:

```
my @sorteddata = sort { $a->{'prior'} <=> $b->{'prior'} } @data;
```

Anschließend durchlaufen wir die Daten und geben sie aus:

```
foreach (@sorteddata) {
    my %rec = %$_;
   if ($rec{'fertig'}) {
        $done = 'Ja';
   } else {
        $done = 'Nein';
    }
    $date = $rec{'datum'};
    $prior = $rec{'prior'};
    $desc = $rec{'desc'};
    write STDOUT;
}
```

Für jeden Teil jedes Datensatzes weisen Sie den temporären Variablen aus dem Format die auszugebenden Daten zu (die Variablen `$done`, `$date`, `$prior` und `$desc`). Die Daten für die Variable `$done` haben eigentlich die Werte `0` oder `1`. In unserem Beispiel ersetzen wir diese Werte durch `Ja` oder `Nein`, um das Ergebnis anschaulicher zu machen.

Nachdem alle Variablen gesetzt wurden, besteht der letzte Schritt aus einer `write`- Anweisung, die das entsprechende Format verwendet, um eine Zeile von Daten auszugeben, und dann zum nächsten Datensatz in der Zeile weitergeht.

Weitere Informationen zu den Formaten finden Sie in der ***perlform***-Manpage. Es lohnt sich auch, einen Blick auf die formatspezifischen Perl-Variablen in der ***perlvar***- Manpage zu werfen, wo Format-Header, Seitenzahlen und anderes beschrieben sind. Schließlich gibt es noch das `FileHandle`-Modul, das Ihnen eine objektorientierte Schnittstelle zu den Datei-Handles und den Formaten bietet und die Arbeit mit vielen der in Perl definierten Elemente zur Verwaltung mehrerer Formate und Datei-Handles erleichtert.

Sockets

Netzwerke waren schon seit jeher eine Stärke von Unix, und Perl wäre schlecht beraten, wenn es nicht Zugriff auf die Netzwerk-Features von Unix bieten würde - insbesondere die TCP- und UDP-Sockets. Perl stellt eine Reihe von eigenen Funktionen für die Arbeit mit Sockets bereit (siehe Tabelle 20.1), die das gleiche Verhalten aufweisen wie ihre C-Gegenstücke. Wenn Sie in socketbasierter Programmierung bereits Erfahrung sammeln konnten, werden Ihnen diese Funktion vertraut vorkommen. Darüber hinaus können Sie das Socket-Modul nutzen, das Ihnen Zugriff auf die allgemeinen Socket-Strukturdefinitionen in C bietet.

Die Verwendung von Sockets in Perl hat jedoch auch seine Einschränkungen. Zum einen stehen Ihnen die meisten der Socket-Features von Perl nur unter Unix zur Verfügung. Unter Windows oder auf Macintosh-Rechner müssen Sie auf die entsprechenden Elemente von Win32 oder MacPerl zurückgreifen, wodurch Ihr Skript nicht mehr so portabel ist. Außerdem gilt es zu bedenken, dass die Chancen meist sehr gut stehen, dass die Aufgaben, die Sie mit Hilfe der Sockets erledigen wollen, bereits als Modul vorhanden sind - es sei denn, Sie wollen irgendein Netzwerkprotokoll implementieren, das nicht dem Standard entspricht. So lohnt es sich kaum, irgendwelche Teile für einen Webserver oder Webbrowser aufzusetzen, da es hinreichend Module gibt, die Ihnen diese Arbeit bereits abgenommen haben. Alles, was Sie tun müssen, ist, diese Module zu nutzen. Morgen werden wir uns hierzu ein Beispiel anschauen, das von den besonders nützlichen Modulen der LWP (Bibliothek für WWW-Zugriff in Perl) Gebrauch macht.

Suchen Sie im CPAN, vor allem im Abschnitt zu den Netzwerken, nach Modulen, die für einen Großteil der Netzwerkaufgaben, die Sie erledigen müssen, fertige Lösungen parat haben. Sollten Sie hier nicht fündig werden, schlagen Sie in der ***perlipc***- Manpage nach, in der Sie weitere Informationen darüber finden, wie man Sockets verwendet (einschließlich einfacher Client- und Server-Beispiele).

In Tabelle 20.1 finden Sie die vordefinierten Funktionen für Sockets. Details zu diesen Funktionen befinden sich in der ***perlfunc***-Manpage.

Funktion	Beschreibung

`accept`	Für Netzwerk-Server: Akzeptiert eine Socket-Verbindung von einem Client. Entspricht `accept(2)`.
`bind`	Bindet eine Adresse an einen bereits geöffneten Socket-Datei-Handle. Entspricht `bind(2)`.
`connect`	Für Netzwerk-Clients: Stellt eine Verbindung zu einem Server her, der auf eine Verbindung wartet. Entspricht `connect(2)`.
`getpeername`	Liefert die Socket-Adresse vom anderen Ende einer Verbindung.
`getsockname`	Liefert die Socket-Adresse von diesem Ende einer Verbindung.
`getsockopt`	Liefert die Werte der Socket-Optionen.
`listen`	Für Server: Teilt dem System mit, auf Socket-Verbindungen zu diesem Socket zu lauschen und diese in eine Warteschleife zu stellen. Entspricht `listen(2)`.
`recv`	Empfängt eine Nachricht an einen Socket. Entspricht `recv(2)`.
`send`	Sendet eine Nachricht zu einem Socket. Entspricht `send(2)`.
`setsockopt`	Setzt die Socket-Optionen.
`shutdown`	Schließt einen Socket mit einigen Steuerungsoptionen. Sie können auch `close()` verwenden.
`socket`	Öffnet einen Socket und verbindet ihn mit einem Datei-Handle.
`socketpair`	Öffnet ein Paar von unbenannten Sockets. Entspricht `socketpair(2)`.

Tabelle 20.1: *Socket-Funktionen*

POD-Dateien

POD (ein Akronym für Plain Old Documentation, übersetzt mit »einfache, alte Dokumentation«) ist eine einfache Formatiersprache zum Erstellen von Dokumentationen zu Ihren Perl-Skripts oder -Modulen. In der Regel nehmen Sie diese Dokumentation direkt in Ihre Skriptdatei mit auf. Auf diese Art und Weise halten Sie das Skript und seine Dokumentation zusammen und müssen nicht beides getrennt warten. (Perl wird den POD-Inhalt einfach ignorieren, wenn Sie Ihre Skripts ausführen).

Um sich den POD-Inhalt Ihres Skripts (oder die POD-Dateien, die nur den POD-Inhalt enthalten) anzuschauen, können Sie das Skript `perldoc` verwenden (wird mit Perl ausgeliefert), mit dem sie den Text der Dokumentation auf Ihrem Bildschirm anzeigen lassen können. Sie können aber auch einen Übersetzer verwenden, der die POD-Dateien in ein anderes Format konvertiert - zum Beispiel `pod2text` für Textdateien, `pod2html` für HTML, `pod2man` für `nroff`-formatierte Manpages oder `pod2latex` für LaTeX-Formatierung. Bisher werden Sie wahrscheinlich `perldoc` verwendet haben, um die verschiedenen Teile der Perl-Dokumentation (im POD-Format) online einzusehen.

POD ist keine vollständige Textformatiersprache wie troff, LaTeX oder HTML. Wenn Sie also Ihre Skripts mit einer Unmenge an aufwendig formatierten Dokumentationen versehen wollen, sind Sie besser beraten, wenn Sie auf ein anderes Format zurückgreifen und Ihre Dokumentationsdateien getrennt von Ihren Skripts halten. POD-Dateien haben jedoch im allgemeinen den Vorteil, dass sie auf verschiedenen Plattformen und mit verschiedenen Systemen gelesen werden oder en passant in andere geläufigere Formate konvertiert werden können.

Beispiele für POD-Texte finden sich in fast jedem öffentlich verfügbaren Skript oder Modul, das mit Perl ausgeliefert wird oder aus dem CPAN heruntergeladen werden kann. Wenn Ihnen die nachfolgenden Ausführungen zu den POD-Dateien nicht genügen, finden Sie weitere Informationen in der ***perldoc***-Manpage.

POD-Dateien erzeugen

POD-formatierter Text besteht aus Befehlsabsätzen und normalen Absätzen. Befehlsabsätze enthalten einfache Formatierungsanweisungen und etwas Text. Normale Absätze enthalten den eigentlichen Text. Sie können aber auch Befehle zur Zeichenformatierung in die normalen Absätzen mit aufnehmen.

Befehlsabsätze stehen in eigenen Zeilen und beginnen mit einem Gleichheitszeichen. Einige Befehlsabsätze weisen

auch Text auf, der sich direkt an den Namen des Absatzes anschließt. Das Ende eines Befehlsabsatzes bildet eine leere Zeile.

Für Überschriften verwenden Sie die Befehlsabsätze `=head1` und `=head2`, denen sich direkt der Text für die Überschrift anschließt. Diese Überschriften entsprechen in etwa den Tags `<H1>` und `<H2>` in HTML oder dem `.SH`-Tag in `troff` oder `nroff`.

Für Listen und andere eingerückte Elemente gibt es die Befehle `=over`, `=item` und `=back`. Mit `=over` beginnen Sie eine Liste, wobei eine optionale Zahl angibt, um wie viele Leerzeichen die Liste eingerückt werden soll. Jedes Listenelement beginnt mit einem `=item`-Tag und einem optionalen Zeichen, das das Symbol oder die Zahl vor diesem Element darstellt (Sie müssen sich selbst um die Numerierung kümmern, da diese nicht automatisch erfolgt). Am Ende der Liste heben Sie die mit `=over` eingeleitete Einrückung mit `=back` wieder auf.

Normale Absätze werden als einfache Absätze eingegeben, ohne sie mit irgendwelchen Befehlen zu kennzeichnen. Absätze, die ohne Whitespace-Zeichen beginnen, werden in der Regel so formatiert, dass sie die Seitenbreite füllen (wie `<P>` in HTML). Absätze mit einer Einrückung am Anfang werden unverändert übernommen (wie `<PRE>` in HTML). Alle Absätze müssen mit einer leeren Zeile enden.

In den Absätzen können Sie Zeichenformatierungscodes und Links mit aufnehmen, um ein bestimmtes Wort hervorzuheben oder einen Link einzufügen (der zu einer anderen Perl-bezogenen Manpage wie ***perlfunc*** oder ähnlichem führt). Im folgenden sehen Sie einige der geläufigeren Zeichenformate:

- `I<text>` gibt das Wort `text` kursiv aus.
- `B<text>` gibt das Wort `text` fett aus.
- `C<text>` faßt `text` als Code auf.
- `&escape;` ersetzt den Escape-Code durch ein Sonderzeichen. Die Escape-Codes sind mit den HTML Escape-Codes für Akzente und andere Sonderzeichen weitgehend identisch.
- `E<escape>`; identisch mit `&escape`.
- `L<manpage>` erzeugt einen Link (oder einen textuellen Querverweis) auf eine Manpage; so erzeugt zum Beispiel `L<perlfunc>` eine Verbindung zu der ***perlfunc***- Manpage. Sie können auch einen Link zu einem speziellen Abschnitt in einer Manpage herstellen.

Um Text einzubetten, der in einer anderen Formatiersprache formatiert vorliegt - zum Beispiel HTML oder troff - gibt es ebenfalls Befehle: `=for`, `=begin` und `=end`. Text, der für bestimmte Formatiersprachen formatiert wurde, wird von dem speziellen Übersetzer unverarbeitet ausgegeben (der Übersetzer `pod2html` wird zum Beispiel formatiertes HTML direkt in die Ausgabe kopieren) und ansonsten einfach ignoriert. Verwenden Sie eingebettete Formatierungen, um in speziellen Fällen eine eingeschränkte Formatierung vorzunehmen.

POD in Skripts einbetten

Sie können POD-formatierten Text sowohl in einer eigenen Datei (in der Regel als Datei mit der Extension `.pod`) speichern oder ihn in Ihre Skript-Dateien integrieren, wo er schnell geändert und aktualisiert werden kann.

Um POD-Text in ein Skript aufzunehmen, markieren Sie den Anfang des POD- Textes mit einem beliebigen POD-Befehl (normalerweise `=head1`, obwohl auch `=pod` den Anfang eines POD-Textes anzeigen kann). Schließen Sie den POD-Text mit `=cut` ab. Auf diese Weise teilen Sie Perl mit, wann die Unterbrechung zu Ende ist und der Text weiter nach Code geparst werden kann.

POD-Text wird in der Regel am Anfang oder am Ende eines Skripts eingefügt. Sie können ihn aber auch irgendwo an einer beliebigen Stelle in Ihr Skript einfügen, zum Beispiel um das Verhalten einer Subroutine direkt über dem Code der Subroutine zu beschreiben. Solange Sie Ihren POD-Text mit einem Befehlsabsatz und `=cut` starten und enden, hat Perl keine Probleme damit.

Wenn Sie einen POD-Text an das Ende einer Datei setzen und eine `__END__`- Markierung verwenden (wie es zum Beispiel beim Erzeugen von Modulen vorkommt), müssen Sie darauf achten, dass eine Leerzeile nach dem `__END__` kommt.

Code en passant ausführen

Ein weiteres nützliches Feature von Perl, das anderen Sprachen abgeschaut wurde, ist die Fähigkeit, einen String zur Laufzeit des Skripts als Perl-Code aufzufassen und auszuführen. Bewerkstelligt wird dies durch die Funktion `eval`, die einen String oder Block als Argument übernimmt und den darin enthaltenen Perl-Code kompiliert und ausführt, so als wenn er in einer Datei stehen oder als Perl-Einzeiler ausgeführt würde - und das alles innerhalb eines gerade laufenden Perl-Skripts.

Warum ist diese Funktion so nützlich? Aus einer Vielzahl von Gründen. Einer davon ist, dass Sie damit anspruchsvolle Strukturen und Funktionsaufrufe aufbauen können, ohne extensiv in `if-else`-Anweisungen verzweigen zu müssen: Stellen Sie den aufzurufenden Code aus aneinandergehängten Strings zusammen, und rufen Sie dann `eval` auf, um den String auszuführen. Es ist auch möglich, andere Dateien, die Perl- Code enthalten, von einem Perl-Skript aus einzulesen und auszuführen - ähnlich der Funktionsweise von `use` und `require`, jedoch immer zur Laufzeit Ihres Skripts (genau genommen ist `require` semantisch identisch mit `eval`, plus einigen Extraoptionen zum Suchen und Neuladen von Dateien). Des weiteren ist es möglich, Code bei Bedarf en passant auszuführen - der Perl-Debugger macht zum Beispiel in seiner Option zur Codeausführung davon Gebrauch. Oder Sie könnten mit Hilfe von `eval` einen Interpreter für Ihre eigene Sprache schreiben.

Mit Hilfe der `eval`-Funktion von Perl kann man Code vor der eigentlichen Ausführung testen und anschließende Fehler und ungewöhnliche Bedingungen, die aus der Ausführung des Codes resultieren, abfangen. Diese Möglichkeit wird in anderen Sprachen oft als Ausnahmenbehandlung bezeichnet. Wenn Sie zum Beispiel testen möchten, ob eine bestimmte Perl-Funktion - zum Beispiel `fork` - auf dem aktuellen System zur Verfügung steht, könnten Sie innerhalb von `eval` ein einfaches Beispiel ausprobieren, schauen, ob es funktioniert, und wenn ja, mit dem Skript und `fork` fortfahren. Scheitert `eval`, führen Sie einen alternativen Code aus. Die Ausnahmebehandlung mit `eval` ermöglicht eine robustere Fehlerprüfung und - behandlung in Ihren Skripts.

Eine Beschreibung von `eval` finden Sie in der ***perlfunc***-Manpage.

Internationale Perl-Skripts erzeugen

Unter Internationalisierung, manchmal auch I18N genannt, versteht man die Verallgemeinerung von Skripts oder Programmen, so dass diese leicht in eine andere Sprache oder einen Dialekt übertragen oder übersetzt werden können. Von Lokalisierung, L10N, spricht man, wenn man eine internationalisierte Version eines Skripts nimmt und in einer speziellen Sprache zum Laufen bringt. Die Internationalisierung beginnt mit einfachen Maßnahmen - zum Beispiel alle Textstrings, die ein Skript verwendet, separat aufzubewahren, um diese ohne Änderungen am Perl-Code übersetzen zu können. Andere Dinge - die Arbeit mit anderen Zeichensätzen als dem Englischen ABC, die Verwendung anderer Sortier- und Vergleichsfunktionen für Texte, das Formatieren von Zahlen - können über die Verwendung des Lokale-Moduls von Perl gesteuert werden.

Weitere Informationen zur Anwendung und Verwaltung von Perl-Lokalen zur Erzeugung internationalisierter (oder lokalisierter) Skripts finden Sie in der ***perllocale***- Manpage.

Skriptsicherheit mit Taint

Angenommen Sie schreiben ein Perl-Skript, das von Menschen ausgeführt wird, die Sie nicht kennen und denen Sie auch nicht besonders trauen - zum Beispiel wenn Sie eine Multi-User-Unix-Maschine verwalten oder wenn Ihr Skript für CGI verwendet wird. Da Sie denjenigen, der Ihr Skript ausführt, nicht kennen, könnte diese Person theoretisch feindliche Absichten hegen und Ihr Skript nutzen, um auf irgendeinem Weg autorisierten Zugriff auf Ihr System zu erhalten oder es in irgendeiner Weise zu mißbrauchen oder zu schädigen.

Wie können Sie verhindern, dass ein bösartiger Benutzer Ihr Skript mißbraucht oder Schaden anrichtet? Sorgfältige Programmierung ist eine Möglichkeit - zum Beispiel vorher prüfen, ob eine Eingabe irgendwelche heiklen Daten enthält, bevor sie einem `system`-Funktionsaufruf oder schrägen Anführungszeichen übergeben wird. Doch manchmal ist es schwierig, zu entscheiden, welche Daten unsicher sind, manchmal vergißt man, diese Prüfungen durchzuführen. In all diesen Fällen ist der Taint-Modus sehr hilfreich.

Der Taint-Modus wird in Perl mit der Option `-T` aktiviert. (Er wird außerdem automatisch ausgeführt, wenn sich die Benutzer- oder Gruppen-ID des Skripts von der Benutzer- oder Gruppen-ID der Person unterscheidet, die das Skript ausführt - zum Beispiel ***setuid***-Skripts auf Unix-Systemen). Ist der Taint-Modus aktiviert, überwacht Perl die Daten, die in Ihr Skript gelangen - aus der Umgebung, als Befehlszeilenargumente oder als Daten von einem Datei-

Handle (einschließlich der Standardeingabe). Wenn Sie versuchen, diese Daten dazu zu nutzen, irgend etwas außerhalb Ihres Skripts zu manipulieren, bricht Perl sofort ab. Um diese Daten trotzdem zu nutzen, müssen Sie Ihr Skript so schreiben, dass diese Daten teilweise geändert oder extrahiert werden - wodurch verhindert wird, dass unerwartete Daten unbemerkt durch das Skript hindurchrutschen.

Mit anderen Worten, der Taint-Modus ist kein eigener Sicherheitsmechanismus, aber er zwingt Sie, beim Schreiben Ihres Codes die Sicherheit immer im Auge zu behalten. Wenn Ihre Skripts zum Beispiel in einer unsicheren Umgebung ausgeführt werden, kann der Taint-Modus Ihnen helfen, sicherzustellen, dass Ihre Skripts nicht zu einladenden Sicherheitslecks in Ihrem System werden.

Weitere Informationen zum Taint-Modus und zur Sicherheit in Perl finden Sie in der ***perlsec***-Manpage. Wenn Sie allgemeinere Auskünfte zum Thema Sicherheit und CGI benötigen, schauen Sie doch mal in den häufig gestellten Fragen (FAQs) zur Sicherheit im WWW unter `http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html` nach.

Wenn Ihnen speziell die Sicherheit Ihrer Skripts am Herzen liegt, sollten Sie Penguin aus dem CPAN ausprobieren. Penguin stellt eine Umgebung bereit, mit der es möglich ist, Codefragmente zu verschlüsseln und digital zu signieren, bevor sie an andere Sites verschickt werden. Am Zielort entschlüsselt Penguin diese Daten wieder und prüft die Zuverlässigkeit dieses Codes vor seiner Ausführung. Und selbst dann führt Penguin den Code noch innerhalb streng kontrollierter Grenzen aus. Betrachten Sie Penguin einfach als ein Gegenstück zu Javas Mechanismus zum Signieren von Applets und der anschließenden Ausführung innerhalb eines geschlossenen, sicheren »Sandkastens«. Im CPAN finden Sie ausführliche Informationen über Penguin.

PerlScript

PerlScript - Teil der ActiveState-Version von Perl für Windows - ist eine Skripting- Maschine für ActiveX-Komponenten. Mit PerlScript können Sie Perl als Skript- Sprache in jeden ActiveX-Skripting-Host verwenden - zum Beispiel im Internet Explorer, IIS oder in einem beliebigen Webserver oder sogar im Windows Scripting Host (WHS) von Microsoft selbst.

Microsofts Skripting-Maschinen für ActiveX-Komponenten unterstützen per Vorgabe VBScript- und JavaScript-Skripting-Maschinen. Diese Sprachen sind zwar für die meisten Zwecke vollkommen ausreichend, doch kann Ihnen Perl in vielen Fällen weitere und leistungsstärkere Optionen bieten. Außerdem ist es für Programmierer, die bereits mit Perl vertraut sind, ein unschätzbarer Vorteil, wenn Sie einfach mit Perl weiterarbeiten können, anstatt zu anderen Sprachen wechseln zu müssen.

Sie können mit PerlScript auch für das Web programmieren und Perl-Skripts in Webseiten einbetten, und das sowohl clientseitig (Webbrowser) als auch serverseitig (Webserver) - so wie man ansonsten JavaScript- und VBScript-Skripts oder ASP- Seiten (***Active Server Pages***) nutzt.

PerlScript läßt sich auch mit dem Windows Scripting Host verwenden. Dadurch haben Sie die Möglichkeit, verschiedene Aspekte des Windows-Systems über Skripts zu steuern (dient als Ersatz für die alten DOS-Batch-Dateien). Zur Zeit gibt es den Windows Scripting Host standardmäßig nur zusammen mit Windows 98. Er läßt sich aber für Windows 95 und Windows NT von `msdn.microsoft.com/scripting` herunterladen und installieren.

PerlScript wird automatisch mit der ActiveState-Version von Perl für Windows installiert. Sie können es aber auch als eigene Komponente von der ActiveState- Webseite herunterladen (Sie müssen allerdings eine Version von Perl für Windows installiert haben).

Weitere allgemeine Informationen zu ActiveX-Skripting finden Sie unter `http://msdn.microsoft.com/scripting`. Details zu PerlScript lassen sich in der Dokumentation zu PerlScript unter `http://www.activestate.com/activeperl/docs/ perlscript.html` oder in dem ausgezeichneten »Complete Guide to PerlScript« (vollständiger PerlScript-Führer) von Matt Sargeant unter `http://www.fastnetltd.ndirect.co.uk/Perl/Articles/PSIntro.html` nachlesen.

Perl-Erweiterungen

Eine Perl-Erweiterung ist ein Weg, eine externe Bibliothek, in der Regel in C geschrieben, in Perl-Skripts

einzubinden. Wenn Sie eine Perl-Erweiterung erzeugen, erzeugen Sie sozusagen einen speziellen Code, der es Ihnen ermöglicht, diese externe Bibliothek wie ein beliebiges Perl-Modul mit `use` in Ihre Perl-Skripts zu importieren. Wenn Sie genau hinschauen, werden Sie feststellen, dass viele der Perl-Module im CPAN sowohl Perl-Code als auch Perl-Erweiterungen verwenden.

Für die Erzeugung einer Perl-Erweiterung müssen Sie eine Sprache namens XS verwenden, die den »Kleber« zwischen Perl und C bereitstellt (Perl-Erweiterungen werden in Anlehnung an die `XSUB`-Funktion in der XS-Sprache manchmal auch XSUBs genannt). Sie können XS-Dateien selbst schreiben oder sie aus einer bestehenden C-Bibliothek erzeugen. Außerdem gibt es besondere Make-Dateien, die erstellt oder erzeugt werden müssen, damit am Ende alles zusammenpaßt (eine Make- Datei ist ein Skript, das verwendet wird, um ein Projekt zu kompilieren und die Abhängigkeiten zwischen mehreren Dateien eines Projekts zu verwalten).

In der Perl-Distribution findet sich eine Reihe von Tools und Modulen, die das Entwikkeln von Perl-Erweiterungen vereinfachen - einschließlich Tools, um durch Erzeugung von XS-Dateien bestehende C-Bibliotheken in Perl einzubinden oder um C-Code aus XS-Dateien zu erzeugen. Besonders erwähnenswert sind in diesem Zusammenhang die Module des `ExtUtils`-Pakets, besonders `ExtUtils::MakeMaker` (die Ihnen helfen, Make-Dateien zu erzeugen).

Bevor Sie damit beginnen, Perl-Erweiterungen zu erzeugen, sollten Sie über ausreichend Hintergrundwissen zu der Entwicklung von Perl-Modulen verfügen. Dieses Hintergrundwissen können Sie sich in der ***perlmod***-Manpage aneignen. Eine Einführung in die Erzeugung von Erweiterungen finden Sie in der ***perlxstut***-Manpage. Des weiteren sollten Sie einen Blick in ***perlxs*** (das XS-Referenzhandbuch), ***h2xs*** (ein Skript zum Konvertieren von C-Headerdateien in XS-Dateien) und ***perguts*** (für interne Perl-Funktionen) werfen. Die POD-Dateien in den verschiedenen `ExtUtils`- Modulen enthalten Erläuterungen, wie diese Module anzuwenden sind.

Neue und fortgeschrittene Elemente in Perl 5.005

Perl 5.005 wurde während der Arbeit an diesem Buch herausgebracht. Ich habe mich bemüht, auf alle Unterschiede und neuen Teile von Perl hinzuweisen, die für Ihre Arbeit mit der neuen Version von Bedeutung sein könnten. Für den Großteil von Perl sind allerdings nur geringfügige Unterschiede zwischen der Perl-Version 5.005 und der früheren Version 5 zu verzeichnen. Radikale Änderungen sollten Ihnen eigentlich nicht auffallen. Wenn Sie aber bereits über Perl 5.005 verfügen und gern etwas herumspielen wollen, finden Sie in dieser Version eine Reihe von neuen Optionen. Einige davon, die bisher im Buch noch keine Erwähnung fanden, möchte ich Ihnen kurz vorstellen.

Die wahrscheinlich bedeutendste Änderung in Perl 5.005 ist der Einsatz von Threading. Rufen Sie sich dazu noch einmal Kapitel 18, »Perl und das Betriebssystem«, in Erinnerung. Dort habe ich die `fork`-Funktion und den Einsatz von Prozessen behandelt und allgemein konstatiert, dass die Prozeßfähigkeiten der Unix- Version von Perl sich nur schwer auf andere Plattformen übertragen lassen. Threading soll dieses Problem jetzt lösen. Sie können damit mehrere gleichzeitig laufende Threads erzeugen, die unabhängig voneinander gestartet und gestoppt werden können. Wichtiger noch ist jedoch die Tatsache, dass Threading in der Perl- Sprache plattformübergreifend unterstützt wird, was bedeutet, dass ein für Unix geschriebenes komplexes Skript mit mehreren Threads sich genausogut unter Windows oder auf einem Macintosh-Rechner ausführen läßt. Perls Thread-System ist in dem Thread-Modul definiert, das man einbinden muss, um mit Threads arbeiten zu können. Beachten Sie, dass sich die Thread-Unterstützung noch im Beta-Stadium befindet - mit anderen Worten, sie kann Fehler aufweisen und Schwierigkeiten in der Anwendung bereiten. Außerdem muss man davon ausgehen, dass die Thread-Unterstützung in zukünftigen Versionen von Perl noch etliche Änderungen erfahren wird. Informieren Sie sich auf alle Fälle in der POD-Dokumentation, die bei dem Thread-Modul dabei ist.

Eine zweite bedeutsame Änderung ist die Integration eines Perl-Compilers. Der Compiler, mit Namen ***perlcc***, kann ein Perl-Modul in eine native Bibliothek konvertieren/kompilieren oder ein Perl-Skript in C-Code konvertieren und diesen Code in eine ausführbare Datei kompilieren. Genau genommen ist ***perlcc*** das Front- End zu einer generischen Kompilierendstufe in Perl (die Module B und O), mit der Sie Perl-Skripts in quasi alles konvertieren oder kompilieren können.

Perl 5.005 enthält außerdem eine neue Maschine für reguläre Ausdrücke, die selbsttätig etliche Fehler behebt und einige Neuheiten aufweist. Um die Performance zu verbessern, können Sie reguläre Ausdrücke außerdem mit dem neuen `C`-Operator vorkompilieren. Weitere Informationen hierzu finden Sie in den Manpages ***perlre*** und ***perlop***.

Vertiefung

Da dieses ganze Kapitel ein einzig großer Vertiefungsabschnitt ist, gibt es eigentlich nicht allzuviel in diesem Abschnitt zu sagen. Deshalb möchte ich Sie an dieser Stelle noch einmal auf folgendes aufmerksam machen: Wenn Sie Probleme, Fragen oder Schwierigkeiten haben oder einfach nur wissen wollen, wie sich ein bestimmter Perl-Teil verhält - von den elementaren Operatoren über die regulären Ausdrücken und die Referenzen bis zu den Modulen -, sollten Sie neben der Perl-Dokumentation (Manpages und POD-Dateien) auf alle Fälle die häufig gestellten Fragen (FAQs) zu Perl zu Rate ziehen. Das Kamelbuch (***Programmieren mit Perl***, das ich am Tag 1 im Vertiefungsabschnitt erwähnt habe) kann Ihnen helfen zu verstehen, wie sich Perl in verschiedenen Situationen verhält. Und wenn Sie trotzdem immer noch festhängen, nutzen Sie das Web - `www.perl.com`, `www.activestate.com` (für Windows) und viele der anderen Websites, die ich im Laufe dieses Buches erwähnt habe, können Ihnen dabei behilflich sein. Schließlich gibt es noch eine Reihe von Newsgroups (wie die `comp.perl`-Hierarchie - insbesondere `comp.perl.misc`) und Mailing-Listen, über die Sie andere Perl-Programmierer kontaktieren können.

Viel Glück!

Zusammenfassung

Heute ist der Tag, an dem wir alle losen Enden zusammenbinden. So haben wir heute eine Reihe von zusätzlichen Perl-Elementen betrachtet, die ich aus Zeit- und Platzgründen in keinem der anderen 19 Kapiteln unterbringen konnte. Dieses Mischmasch umfaßt:

- Perl-Einzeiler - einfache Skripts, die direkt von der Befehlszeile aus ausgeführt werden können, um einfache Aufgaben zu lösen, ohne ein ganzes Skript dafür schreiben zu müssen
- Objektorientierte Programmierung in Perl - eine Kombination aus Referenzen, Paketen, Modulen und Subroutinen sowie zusätzlichen Code, um das Ganze in Zusammenhang zu bringen
- Perl-Formate, mit denen Sie Schablonen für textformatierte Protokolle erzeugen können (der »Report«-Teil der »Praktischen Extraktions- und Reportsprache«)
- Einen Überblick über die Netzwerk-Socket-Programmierung in Perl
- POD-Dateien - Perls einfache Methode zum Erzeugen eingebetteter Dokumentationen, die auf verschiedene Art formatiert werden können (HTML, Text und so weiter)
- Erstellen und Auswerten von Code mit `eval`
- Internationalisierung und Lokalisierung mit dem Lokale-Modul
- Prüfen auf vergiftete Daten, um Sicherheitsfehler und -löcher in Ihren Skripts zu verhindern
- Der Einsatz der PerlScript-Maschine unter Windows, um Perl-Skripts in HTML- Webseiten einzubinden oder damit DOS- oder Windows-Batch-Dateien zu ersetzen
- XSUBs, Perl-Erweiterungen, die es Ihnen erlauben, Perl-Skripts mit nativen (C- Code-) Bibliotheken zu verbinden
- Ein kurzer Überblick über einige der interessanteren Neuheiten in der neuen Perl- Version 5.005

Meinen Glückwunsch! Jetzt bleibt uns nur noch ein Tag, und der ist ganz den Beispielen gewidmet. Sie haben Perl in ausreichendem Umfang kennengelernt, um einige aufregende Dinge zu machen. Also nur zu! Und vergessen Sie nicht - Perl ist ein Gemeinschaftsprodukt. Nutzen Sie die Module im CPAN. Und sollten Sie selbst etwas schreiben, von dem Sie meinen, dass auch andere es gebrauchen könnten, stellen Sie es doch auch in das CPAN.

Fragen und Antworten

Frage:
Meine Einzeiler funktionieren nicht in MacPerl.

Antwort:
Stellen Sie im Dialog `Edit->Preferences->Scripts` *sicher, dass Sie die Option* `Run Scripts Opened from Finder` *markiert haben. Wenn Sie hier* `Edit` *ausgewählt haben, lassen sich Ihre Einzeiler nicht ausführen.*

Frage:
Meine Einzeiler funktionieren nicht unter Windows.

Antwort:
Es gibt Unterschiede zwischen den Anführungszeichen in der DOS/Windows- Befehlszeile und in Unix. Unter Unix können Sie einfache und doppelte Anführungszeichen verwenden, unter Windows nur doppelte. Deshalb sollten Sie sich vergewissern, dass Ihre Einzeiler unter Windows vorn und hinten in doppelten Anführungszeichen stehen und alle Anführungszeichen im Skript selbst mit einem Backslash markiert sind.

Frage:
Meine Einzeiler funktionieren nicht in Unix.

Antwort:
Sie funktionieren nicht? Haben Sie das Skript auch wirklich in eine Zeile eingegeben, ohne Return-Zeichen? Haben Sie auch nicht vergessen, das gesamte Skript in einfache Anführungszeichen zu setzen? Haben Sie an die Perl-Option -e gedacht? Wenn Sie alle diese Punkte überprüft haben, stellen Sie sicher, dass Sie das gleiche Skript ausführen können, wenn Sie es in einer Datei abspeichern.

Frage:
Aus Ihrer Beschreibung entnehme ich, dass Perl die OOP-Konzepte der öffentlichen (public) und privaten (private) Daten nicht unterstützt. Was sollte dann jemand daran hindern, Methoden zu verwenden, die nicht als Teil der öffentlichen API ihrer Klasse gedacht sind?

Antwort:
Nichts. Das objektorientierte Modell von Perl unterscheidet nicht zwischen privater und öffentlicher API. Es geht davon aus, dass Sie und die Programmierer, die Ihre Klassen verwenden, sich anständig verhalten, was die API angeht. Als Entwickler einer Klasse sollten Sie Ihre API dokumentieren und angeben, welche Ihrer Methoden öffentlich sind (dazu eignet sich POD sehr gut), und dann davon ausgehen, dass alle, die Ihre Klasse verwenden, sich auch an diese Vorgaben halten. Umgekehrt sollten auch Sie, wenn Sie die Klassen anderer Programmierer verwenden, sich an deren dokumentierte öffentliche API halten und interne Methoden oder Daten nur verwenden, wenn Sie genau wissen, was Sie tun (und das Risiko zu tragen bereit sind).

Frage:
Mehrfachvererbung ist sehr verwirrend und fehleranfällig. Warum hat sich Perl nicht auf die einfache Vererbung beschränkt?

Antwort:
Um damit die Mächtigkeit der Sprache zu beschränken? Es sollte Ihnen an dieser Stelle im Buch bereits aufgefallen sein, dass Perl zu keiner Zeit wirklich versucht, den Programmierer in seiner Arbeit Grenzen zu setzen. Mehrfachvererbung mag zwar oft verwirrend und auch schwierig zu debuggen sein, aber sie ist auch unglaublich leistungsstark, wenn Sie zusammen mit einem guten Design verwendet wird (und vergessen Sie nicht, ein gutes Design bedingt nicht unbedingt Vererbung). Die Entwickler von Perl lassen Ihnen lieber genug Seil, um sich aufzuhängen, als dass sie Sie zu knapp halten, so dass Sie letztlich nicht genug Seil haben, um einen Knoten zu binden.

Frage:
Wo, zum Teufel, kommen die Begriffe I18N und L10N her?

Antwort:
Es stehen jeweils 18 Buchstaben zwischen dem I und dem N in dem Wort »Internationalization« (Internationalisierung) und 10 Buchstaben zwischen dem L und dem N in »Localization« (Lokalisierung). Internationalization und Localization sind wirklich sehr lange Wörter. Programmierer mögen das nicht.

Frage:
Ihre Beschreibung der Perl-Extensionen erläutert, wie man C-Code aus einem Perl-Skript heraus aufruft. Wie ruft man Perl-Code von einem C-Programm aus auf?

Antwort:
Schlagen Sie in der perlcall-Manpage nach. Außerdem müssen Sie sich in der XS-Sprache auskennen und wissen, wie man Erweiterungen konstruiert.

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was bewirken die folgenden Perl-Schalter (zusammen mit Einzeilern)?

```
  -e
-i
-p
```

2. Wie werden in Perl Klassen, Methoden und Instanzvariablen realisiert?
3. Zeigen Sie zwei Wege auf, wie man eine Methode aufrufen kann.
4. Wie wird nach mehrfachvererbten Superklassen gesucht, wenn eine Methode aufgerufen wird?
5. Was sind selbstladende Methoden?
6. Wozu werden Perl-Formate verwendet? Wie lauten die zwei Funktionen, die zur Verwendung von Formaten benötigt werden?
7. Was ist eine POD-Datei? Warum sollten Sie in Ihren Perl-Skripts POD anderen Formaten wie HTML vorziehen?
8. Wozu dient die Funktion `eval`? Wozu verwendet man sie?
9. Wozu wird der Taint-Modus verwendet? Wie aktivieren Sie ihn?
10. Was ist ein XSUB?
11. Perl 5.005 erweitert Perl um die Fähigkeit zum Multithreading. Wozu verwendet man Threads, und warum sind sie so nützlich für die Perl-Programmierer?

Übungen

1. Schreiben Sie einen Perl-Einzeiler, der alle Vorkommen des Buchstaben »t« in der Eingabe zählt und das Ergebnis ausgibt.
2. Schreiben Sie einen Perl-Einzeiler, der die Summe der Eingabe berechnet (unter der Annahme, dass die Eingabe numerisch ist).
3. Schreiben Sie einen Perl-Einzeiler, der alle Vorkommen von drei aufeinanderfolgenden Leerzeichen durch einem Tabulator ersetzt und das Ergebnis in einer Datei gleichen Namens speichert.
4. FEHLERSUCHE: Was ist an diesem Einzeiler falsch?

```
perl -p 's/t/T/q';
```

5. FEHLERSUCHE: Und was ist mit diesem? (HINWEIS: Es gibt mehr als ein Problem)

```
perl -ne 'print 'Zeile: ', reverse $_;';
```

6. Schreiben Sie eine Perl-Klasse namens `EinfacheKlasse`, die drei Instanzvariablen enthält: `a`, `b` und `c`. Nehmen Sie folgende Methoden mit auf:

- Einen `new`-Konstruktor, um die Variablen `a`, `b` und `c` zu erzeugen und zu initialisieren.
- Methoden, um die Werte von `a`, `b` und `c` auszugeben und zu ändern. Implementieren Sie diese, wie Sie wollen.
- Eine Methode, um die Summe von `a`, `b` und `c` auszugeben. Vergewissern Sie sich, dass `a`, `b` und `c` Zahlen enthalten; geben Sie andernfalls Fehlermeldungen aus.

Antworten

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. Die Option `-e` führt ein Perl-Skript (einen Einzeiler) von der Befehlszeile aus.
1. Die Option `-i` bearbeitet Dateien. Das bedeutet, das Ergebnis des Skripts wird in der Originaldatei

gespeichert. Wird der Option `-i` eine Dateiextension als Argument mitgegeben, wird diese verwendet, um eine Sicherungsdatei der Originalversion der Datei zu erstellen.

1. Die Option `-p` bettet den Perl-Einzeiler in eine `while`-Schleife (`<>`) mit einer `print`- Anweisung am Ende ein. Wenn Sie die Zeilen einer Datei einzeln bearbeiten und dann ausgeben wollen, sollten Sie diese Option verwenden, um sich Tipparbeit zu ersparen.
2. Klassen sind in Perl Pakete, Methoden sind Subroutinen, die entweder einen Klassennamen oder eine Objektreferenz als erstes Argument haben. Instanzvariablen werden in der Regel als Hash-Elemente implementiert, wobei die Namen der Variablen als Schlüssel dienen.
3. Sie können Methoden wie reguläre Funktionen aufrufen:

```
methode $obj,1,2,3;
```

1. oder die Dereferenzierungssyntax verwenden:

```
$obj->methode(1,2,3);
```

4. Mehrfachvererbte Superklassen werden vertikal durchsucht. Das heißt, zuerst werden die erste Superklasse in der Liste und alle ihre Superklassen durchsucht, bevor die nächste Superklasse in der Liste durchsucht wird.
5. Selbstladende Methoden (AUTOLOAD) werden aufgerufen, wenn in einem Skript eine Methode aufgerufen wird, für die es weder in der aktuellen Klasse noch in einer ihrer Superklassen eine zugehörige Methodendefinition gibt. Sie können selbstladende Methoden dazu nutzen, um Methoden (wie zum Beispiel die Lese- und Schreibmethoden für Variablen) zusammenzufassen oder um Aufrufe unbekannter Methodennamen aufzufangen.
6. Perl-Formate werden verwendet, um Daten tabellarisch für die Textausgabe aufzubereiten. Die Daten werden automatisch der eingestellten Breite angepaßt, wobei die einzelnen Spalten nach den von Ihnen definierten Formatvorgaben ausgerichtet oder ausgefüllt werden. Sie erzeugen ein Format, indem Sie mit `format` eine Formatschablone definieren und das Ergebnis mit `write` ausgeben.
7. POD steht für ***Plain Old Documentation***. Mit POD können Sie auf einfache Weise online-zugängliche Dokumentationen zu Ihren Perl-Skripts erzeugen. POD ist ein allgemein nützliches Dokumentationsformat, das leicht extrahiert und in andere Ausgabeformate wie HTML oder `troff` konvertiert werden kann.
8. Die Funktion `eval` wird verwendet, um Perl-Codefragmente en passant während der Ausführung eines Skripts auszuwerten. Sie können `eval` dazu nutzen, um andere Skripts in Ihr aktuelles Skript aufzunehmen oder Codeabschnitte zu testen, bevor Sie sie richtig ausführen lassen.
9. Der Taint-Modus von Perl dient dazu, Ihre Perl-Skripts gegen Fehler und Versäumnisse zu schützen, die zu unsicherem Code führen können. Unsicherer Code, der in einer unsicheren Umgebung ausgeführt wird, kann böswilligen Benutzern die Gelegenheit bieten, Ihr System zu manipulieren oder sogar zu schädigen. Der Taint-Modus legt alle externen Daten und Einstellungen in einer besonderen, kontrollierten Umgebung ab und verhindert so, dass Sie diese Daten zufällig zu Ihrem eigenen Schaden nutzen. Sie aktivieren den Taint-Modus mit der Perl-Option `-T`. Der Modus wird allerdings auch automatisch eingeschaltet, wenn sich die Benutzer- oder Gruppen-ID des Skripts von der Benutzer- oder Gruppen- ID der Person unterscheidet, die das Skript ausführt (ist beispielsweise für CGI- Skripts, die auf einem Webserver ausgeführt werden, der Fall).
10. XSUB ist der umgangssprachliche Name für eine Perl-Erweiterung: ein natives Codefragment, das von einem Perl-Skript aus ausgeführt werden soll. Die Bezeichnung XSUB leitet sich von einer Funktion in der XS-Sprache ab.
11. Multithreading ermöglicht die gleichzeitige Ausführung mehrerer »Threads« (zu Deutsch »Fäden«) in einem Perl-Skript. Perl-Threads sind nicht nur deshalb so interessant, weil Sie den Perl-Programmierern neue Möglichkeiten eröffnen und höhere Flexibilität bieten, sondern weil sie plattformübergreifend verwendet werden können. Vorher war diese Art von Verhalten nur durch die Unix-Funktion `fork` zu bewerkstelligen. Aber mit der Verwendung von `fork` waren Ihre Skripts nicht mehr auf andere Plattformen portierbar. Durch Threads wurde das Problem jetzt gelöst.

Lösungen zu den Übungen

1. Eine mögliche Lösung ist:

```
perl -e ' while (<>){while (/t/g){$c++;}};print $c;' datei.txt
```

2. Eine mögliche Lösung lautet:

```
perl -e 'while(<>){$sum+=$_;}print "$sum\n";'
```

3. Eine mögliche Lösung lautet:

```
perl -pe -i.bak 's/   /\t/g' ozy.txt
```

4. Die Option `-e` fehlt.
5. Dieses Beispiel weist zwei Probleme auf: eines syntaktischer und eines konzeptioneller Art. Der erste Fehler ist, dass man einfache Anführungszeichen nicht ineinander verschachteln kann. Ersetzen Sie die inneren einfachen Anführungszeichen durch doppelte Anführungszeichen.

1. Das zweite Problem betrifft die Funktion `reverse`. Dieser Einzeiler erweckt den Eindruck, dass die einzelnen Eingabezeilen zeichenweise in umgekehrter Reihenfolge mit dem Wort »Zeile« am Anfang ausgegeben werden. Sie sollten sich jedoch in Erinnerung rufen, dass die `reverse`-Funktion je nach Kontext (Skalar- oder Listenkontext) ein unterschiedliches Verhalten aufweist. In diesem Falle wird `reverse` in einem Listenkontext aufgerufen, da die Argumente zu `print` immer eine Liste oder eine Kombination von Listen sind. Doch damit wird nur die Reihenfolge der Zeilen in einem Array umgedreht. Das Argument `$_` wird dann in eine Liste konvertiert, diese Liste mit einer Zeile umgedreht und anschließend ausgegeben. Mit anderen Worten, äußerlich passiert nichts.

1. Damit der String wirklich umgedreht wird, müssen Sie die `reverse`-Funktion in einem skalaren Kontext aufrufen. Dieser Fehler kann schnell mit der `scalar`- Funktion behoben werden:

```
perl -ne 'print 'Zeile: ', scalar (reverse $_);'
```

1. Achten Sie auf die Neue-Zeile-Zeichen. Wenn Sie eine Zeile mit einem Neue- Zeile-Zeichen am Ende umdrehen, wird dieses Zeichen an den Anfang gesetzt. Die bessere Lösung ist, den String von dem Neue-Zeile-Zeichen zu befreien, ihn mit `reverse` umzudrehen und ihn dann mit dem frisch angehängten Neue-Zeile- Zeichen wieder auszugeben:

```
perl -ne 'chomp;print "Zeile: ", scalar(reverse $_), "\n" ; '
```

1. Zu unserer Bequemlichkeit gibt es eine Perl-Option, die diese Schritte vereinfacht: die Option `-l`. Damit wird das Neue-Zeile-Zeichen von der Eingabe entfernt, um es (oder irgendein anderes Zeilenende-Zeichen Ihrer Wahl) hinterher für Sie wieder anzuhängen:

```
perl -lne 'print "Zeile: ", scalar(reverse $_);'
```

6. Hier eine mögliche Lösung (mit Code, um das Ergebnis am Ende zu testen). Die folgende Version zeigt drei Wege auf, wie man auf Instanzvariablen zugreifen kann. Beachten Sie, dass keine dieser Methoden besonders robust ist. Keine von ihnen prüft, ob Sie versuchen, die Werte von nichtexistenten Variablen zu lesen oder zu setzen (und in der Tat werden die generischen Versionen mit Freude eine Instanzvariable hinzufügen, die nicht `a`, `b` oder `c` heißt).

```
    #!/usr/bin/perl -w
package EinfacheKlasse;
sub new {
   my ($classname, $a, $b, $c) = @_;
   my $self = {};
   $self->{a} = $a;
   $self->{b} = $b;
   $self->{c} = $c;
   return bless $self, $classname;
}
# der längste Weg
sub getA {
    my $self = shift;
    return $self->{a};
}
sub setA {
    my $self = shift;
    if (@_) {
        $self->{a} = shift;
    } else {
        warn "kein Argument; verwende undef\n";
        $self->{a} = undef;
```

```
        }
    }
    # ein etwas allgemeinerer Weg, der mehr Argumente benötigt.
    sub get {
        my ($self, $var) = @_;
        if (!defined $var) {
            print "Keine Variable!\n";
            return undef;
        } elsif (!defined $self->{$var}) {
            print "Variable nicht definiert oder ohne Wert.\n";
            return undef;
        } else {
            return $self->{$var};
        }
    }
    sub set {
        my ($self, $var, $val) = @_;
        if (!defined $var or !defined $val) {
            print "Brauche Variable und Wert als Argument!";
            return undef;
        } else {
            $self->{$var} = $val;
            return $val;
        }
    }
    # ein wirklich allgemeiner Weg
    sub AUTOLOAD {
        my $self = shift;
        my $var = $AUTOLOAD;
        $var =~ s/.*::/;
        $var =~ s/[gs]et//;
        $var = lc $var;
        if (@_) {
            $self->{$var} = shift;
            return $self->{$var};
        } else {
            return $self->{$var};
        }
    }
    sub sum {
       my $self = shift;
       my $sum = 0;
       foreach ('a','b','c') {
           if (!defined $self->{$_} or $self->{$_} !~ /^\d+/ ) {
               warn "Variable $_ enthaelt keine Zahl.\n";
           } else { $sum += $self->{$_}; }
       }
       return $sum;
    }
    package main;
    $obj = new EinfacheKlasse (10,20,30);
    print "A: ", $obj->getA(), "\n";
    $obj->setA("foo");
    print "A: ", $obj->getA(), "\n";
    print "B: ", $obj->get('b'), "\n";
    $obj->set('b', 'bar');
    print "B: ", $obj->get('b'), "\n";
    # es gibt keine getC-Methode; autoload
    print "C: ", $obj->getC(), "\n";
    $obj->setC('baz');              # ditto setC
    print "C: ", $obj->getC(), "\n";
    # zurücksetzen
    print "\nA: 10\n";
    $obj = new EinfacheKlasse (10);
    print "Summe: ", $obj->sum(), "\n";
    print "\nA: 10 B: 5\n";
    $obj = new EinfacheKlasse (10,5);
    print "Summe: ", $obj->sum(), "\n";
    print "\nA: 10 B: 5 C: 5\n";
    $obj = new EinfacheKlasse (10,5,5);
    print "Summe: ", $obj->sum(), "\n";
```

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Ein paar längere Beispiele

Ebenso wie die ersten beiden Wochen dieses Buches werden wir auch die dritte Woche mit einem Beispielkapitel abschließen. Die beiden Beispielskripts, die ich in diesem Kapitel besprechen werde, decken so ziemlich alle Aspekte von Perl ab, die Sie in den letzten 20 Tagen kennengelernt haben. Nachdem Sie diese beiden Beispielskripts durchgearbeitet und verstanden haben, sollten Sie in der Lage sein, Ihren eigenen Weg mit Perl zu gehen.

Heute untersuchen wir zwei Beispiele, beides CGI-Skripts:

- Einen Homepage-Generator, der auf der Grundlage einer Konfigurationsdatei und des HTML-Codes einer Vielzahl anderer Sites eine Webseite mit den von Ihnen gewünschten Informationen erzeugt - wirklich nur mit den von Ihnen gewünschten Informationen
- Eine webbasierte Aufgabenliste, die es erlaubt, Aufgaben hinzuzufügen, zu entfernen, als fertig zu markieren, Prioritäten festzulegen und die Liste auf vielfältige Art und Weise zu sortieren

Beide Beispiele sind länger als die bisherigen, das letzte sogar mehr als doppelt so lang. Zur Erinnerung: Alle Beispiele dieses Buches sind auch auf der Buch-CD und online auf der »Sams Teach Yourself Perl«-Website unter `http://www.typerl.com/` verfügbar. Damit Sie nicht meinen, Sie müßten 350 Zeilen Code von Hand abtippen. Nutzen Sie dieses Kapitel, um den Code sorgfältig zu studieren, und wenn Sie dann noch weiter damit experimentieren wollen, kopieren Sie das Skript von der CD oder aus dem Internet.

Beide Beispiele werden als CGI-Skripts ausgeführt. Wie schon bei den CGI-Skripts, die wir in Kapitel 16, »Perl für CGI-Skripts«, untersucht haben, sind die Installation und die Anwendung dieser Skripts von Plattform zu Plattform und von Webserver zu Webserver unterschiedlich. Einige Server verlangen, dass die Skripts in einem speziellen Verzeichnis (***cgi-bin***) installiert werden, dass die Dateinamen der Skripts besondere Extensionen (`.cgi` statt `.pl`) haben oder dass spezielle Zugriffsrechte gesetzt werden. Unter Umständen müssen Sie auch die Zugriffsrechte für die Konfigurations- und Datendateien der Skripts setzen. Weitere Informationen zum Installieren und Verwenden von CGI-Skripts finden Sie in der Dokumentation zu Ihrem Webserver, oder wenden Sie sich an Ihren Systemverwalter.

Ein Homepage-Generator (*meinehomepage.pl*)

Viele Websites verfügen heutzutage über Eintrittsseiten, deren Inhalt Sie anpassen und selbst zusammenstellen können (nach dem Motto »Mein Portal« oder ähnlich), so dass Sie beim Ansteuern dieser Websites schnell die Informationen finden, die Sie interessieren. Diese Seiten sind ein hervorragendes Mittel, um den Inhalt einer Website darzustellen. Ich dachte, es wäre ganz nett, wenn man jetzt noch einen Schritt weiter ginge und sich selbst eine eigene Seite anlegen würde, auf der man die verschiedensten Dinge von mehreren Websites kombinieren könnte - und damit meine ich nicht nur Links zu diesen Seiten, wie eine Textmarken- und Favoritenliste, sondern direkt den Inhalt der Seiten. Börsenkurse von einer Site, Wetterdaten oder Nachrichten von einer anderen, eine Sammlung von Informationen aus den verschiedensten Websites - und das alles auf einer Seite.

Und das ist genau das, was ***meinehomepage.pl*** macht. Das Skript nutzt das `LWP`- Modul, mit dem es Perl möglich ist, ausgewählte Webseiten von Websites über ein Netzwerk anzufordern (dies ist eines der Skriptbeispiele, in dem Sie eine ganze Reihe von Modulen einsetzen, um den Großteil der Arbeit zu erledigen, und ein wenig eigenen Perl-Code aufsetzen, um diese Module zusammenzubinden). Für jede Information, die auf der selbstgebauten Seite erscheinen soll, wird der HTML-Code der betreffenden Webseiten aus dem Internet herangezogen. Der relevante Code wird extrahiert und daraus die eigene Webseite mit den interessanten Teilen aufgebaut (plus einige weitere Links, die ich jedoch nicht jeden Tag besuchen möchte). Wie das Ergebnis von ***MeineHomepage*** aussehen kann, zeigt Ihnen Abbildung 21.1. Diese spezielle Version enthält die wichtigsten Börsendaten, das Wetter und den aktuellen Dilbert-Cartoon von United Media (Dilbert ist leider nicht zu sehen, da nicht genug Raum vorhanden ist

und es außerdem Probleme mit dem Copyright gäbe; doch glauben Sie mir, er ist vorhanden.)

Vorgehensweise

Ich hätte das Skript so schreiben können, dass ich alle Websites und deren Verarbeitung hart-kodiert hätte - man nehme eine Seite für Börsenkurse, extrahiere die Daten, schreibe den HTML-Code und wiederhole das gleich für das Wetter und den Comic-Strip. Und wenn ich mehr Daten aufnehmen oder weitere Links oder neue Inhalte einfügen wollte, müßte ich das Skript selbst modifizieren (siehe Abbildung 21.1).

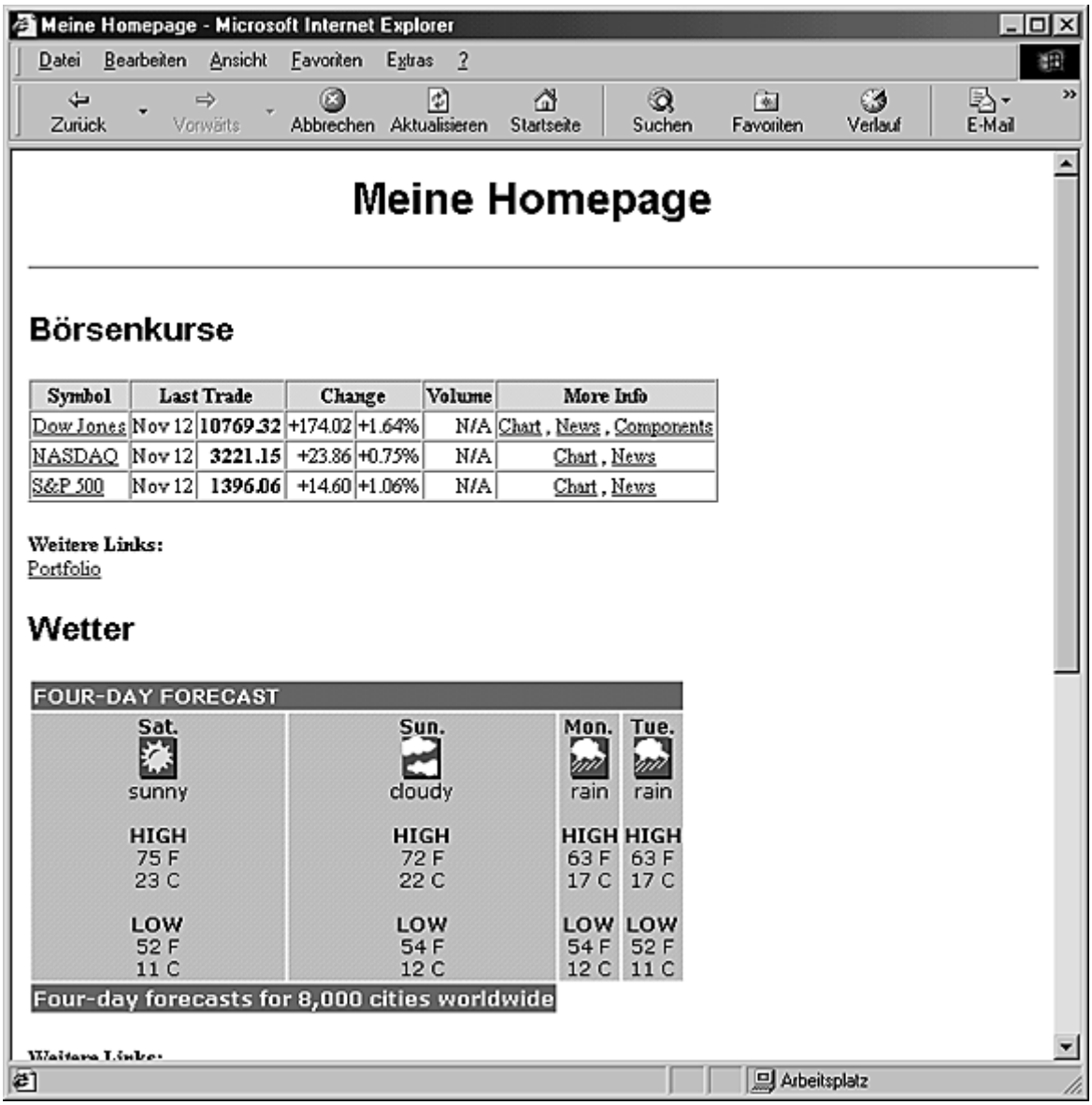

Abbildung 21.1: Meine Homepage, Endergebnis

Auf den ersten Blick mag das zwar praktisch erscheinen, aber letztlich ist es sinnvoller, die Daten - das heißt, das was auf der Homepage erscheinen soll - von dem Skript, das alles herunterlädt und verarbeitet, zu trennen. Deshalb besteht dieses Skript auch aus zwei Teilen: einer Konfigurationsdatei und einem Perl-Skript, das die Daten aus der Konfigurationsdatei einliest und verarbeitet.

Die Konfigurationsdatei entspricht dem Adreßbuch aus Kapitel 14: Es ist eine Datei mit mehreren »Datensätzen«, die jeweils durch drei Bindestriche getrennt sind (---). Jeder Datensatz entspricht einem Abschnitt der Homepage und enthält vier Felder:

- Einen Titel für den im Abschnitt angezeigten Inhalt
- Einen URL, der auf den Inhalt weist, den wir aus dem Internet holen wollen
- Einen Code-Abschnitt, der den Code enthält, mit dem die wichtigen Teile der Webseite extrahiert werden (in

der Regel eine Reihe regulärer Ausdrücke)
- Einen weiteren Abschnitt mit Code für einen Hash von weiteren Links, wobei die Schlüssel die Titel der Links sind und die Werte die URLs

Listing 21.1 zeigt einen Mustereintrag aus meiner ***config***-Datei.

Listing 21.1: Meine Homepage-Konfigurationsdatei (Muster)

```
title=Dilbert
url=http://www.unitedmedia.com/comics/dilbert/
code={
   if ($content =~ /src=\"(.*dilbert\d+.gif)\"/i) {
        print "<P><IMG SRC=\"";
        print url($1,$data{'url'})->abs->as_string, "\">\n";
    }
}
other=( 'United Media Comics' => 'http://www.unitedmedia.com/comics/',
       'San Jose Mercury news Comics' =>
           'http://comics.mercurycenter.com/comics/',
       'Tom Tommorow (mondays only)' =>
           'http://www.salonmagazine.com/comics/tomo/',
)
---
```

Das Skript zu ***Meine Homepage*** ist ein CGI-Skript, das Sie wie jedes andere CGI- Skript installieren müssen. Im Gegensatz zu den in Kapitel 14 betrachteten CGI- Skripts gibt es zu diesem Skript kein Formular, durch das es aufgerufen wird, und es müssen auch keine Formularparameter verarbeitet werden. Das Skript wird direkt wie ein URL (`http://www.ihresite.com/cgi-bin/meinehomepage.pl` oder so ähnlich) aufgerufen und liest dann die Konfigurationsdatei ein und verarbeitet sie (installieren Sie die Konfigurationsdatei dort, wo das Skript sie finden kann). Der Rückgabewert ist der für die Homepage generierte HTML-Code.

Das Skript

Kommen wir jetzt zum Perl-Teil. Gehen Sie den Code in Listing 21.2 einfach mal nur durch, und versuchen Sie einen Gesamteindruck von dem Ablauf des Skripts zu bekommen. Achten Sie dabei besonders auf die Subroutine `&procsection()`, denn hier fängt es an, interessant zu werden.

Listing 21.2: Der Code für meinehomepage.pl

```
1:  #!/usr/bin/perl -w
2:  use strict;
3:  use LWP::Simple;
4:  use URI::URL;
5:  use CGI qw(header start_html end_html);
6:
7:  my $config = 'config';
8:
9:  print header;
10: print start_html('Meine Homepage');
11: print "<H1 ALIGN=CENTER><FONT FACE=\"Helvetica,Arial\" SIZE+2>";
12: print "Meine Homepage</FONT></H1>\n";
13: print "<HR>\n";
14:
15: open(CONFIG, $config) or &quitnow();
16:
17: while () {
18:    my %rec = &readrec();
19:
20:     if (%rec) {
21:         &procsection(%rec);
22:     } else { last; }
23: }
24:
25: print end_html;
26:
27: sub readrec {
28:    my %curr = ();
```

```
29:     my $key = '';
30:     my $val = '';
31:
32:     while (<CONFIG>) {
33:        chomp;
34:         if ($_ ne '---' and $_ ne '') { # Trennsymbol für Datensätze
35:            if ($_ =~ /^\#/) { next;} # Kommentare
36:
37:             ($key, $val) = split(/=/,$_,2);
38:
39:             # mehrzeilige Einträge verarbeiten
40:             if ($key eq 'code' or $key eq 'other' ) {
41:                 my $in = '';
42:                 while () {
43:                     ($in = <CONFIG>);
44:                     $val .= $in;
45:                    if ($in =~ /^[})]/) { last; }
46:                 }
47:             }
48:
49:             $curr{$key} = $val;
50:         }
51:         else { last; }
52:    }
53:     return %curr;
54: }
55:
56: sub procsection {
57:     my %data = @_;
58:
59:     print "<H2><FONT FACE=\"Helvetica,Arial\">";
60:     print "$data{'title'}</FONT></H2>\n";
61:
62:     my $content = get($data{'url'});
63:     if (defined $content) {
64:         eval $data{'code'};
65:     } else {
66:        print "<P><B>Hoppla!</B> Kann nicht auf die Daten zugreifen\n ";
67:         exit;
68:     }
69:
70:     if (defined $data{'other'}) {
71:         my %others = eval $data{'other'};
72:
73:         print "<P><B>Weitere Links:</B> <BR>\n";
74:         foreach (keys %others) {
75:             print "<A HREF=\"$others{$_}\">$_</A><BR> ";
76:         }
77:        print "\n";
78:     }
79:
80: }
81:
82: sub quitnow {
83:     print "<H2>Hoppla! Fehler:</H2><P><TT> Konfigurationsdatei konnte nicht geöffnet werder
84:     print "<P>Überprüfen Sie Ihre Installation oder kontaktieren ";
85:     print "Sie Ihren Systemadministrator\n";
86:     print end_html;
87:     die "Konfigurationsdatei konnte nicht geöffnet werden: $!\n";
88: }
```

Das Skript ***meinehomepage.pl*** geht in vier Schritten vor:

- Es öffnet das Konfigurationsskript und liest einen Datensatz ein, wobei es einen Hash für die Felder in diesem Datensatz anlegt.
- Es besorgt sich den HTML-Quelltext für die Seite mit dem angegebenen URL.
- Es wendet den Code aus der Konfigurationsdatei auf den HTML-Quelltext an.
- Es gibt den resultierenden HTML-Code aus.

Das Skript wiederholt diese Schritte für jeden Eintrag in der Konfigurationsdatei.

Das meiste in diesem Skript sollte Ihnen bekannt vorkommen. Um ehrlich zu sein, ich habe das Skript ***adressen.pl*** aus Kapitel 14 als Grundlage für dieses Skript genommen, da sich die Konfigurationsdateien ähneln. Was Ihnen fremd erscheinen dürfte, ist die Verwendung der zwei Module `LWP::Simple` und `URI::URL` (und die damit verbundenen Subroutinen: `get`, `url`, `abs` und `as_string`) sowie die Verwendung der Funktion `eval`, um den Code der Konfigurationsdatei in den Rumpf des Skripts zu holen.

Die Module *LWP::Simple* und *URI::URL*

Wenn man sich nicht sicher ist, wie eine bestimmte Aufgabe zu lösen ist, verwendet man einfach den Code von anderen Programmierern. Oder noch besser: Anstatt viel Zeit damit zu verschwenden, Skripts für komplizierte Aufgaben zu schreiben (zum Beispiel Netzwerkcode zu schreiben, um eine Datei von einer Website anzufordern), schauen Sie erst einmal im CPAN nach. In diesem Fall waren die `LWP`-Module (Bibliothek für WWW-Zugriff in Perl) genau das, was ich benötigte. Da ich gerade im CPAN war, habe ich gleich das gesamte `libwww`-Paket heruntergeladen, das alle Arten von Modulen für die Handhabung von Webseiten und Webdaten enthält.

Um dieses Skript auszuführen, müssen Sie diese Dateien vom CPAN herunterladen (siehe `www.perl.com` für weitere Details) und dort auf ihrem System installieren, wo Sie das Skript ausführen. Lesen Sie im Kapitel 13, »Gültigkeitsbereiche, Module und das Installieren von Code«, nach, wie man Module in Perl installiert und verwendet, wenn Sie sich nicht mehr genau erinnern können, wie dies geht.

> *Die Bibliothek `libwww` allein wird Ihnen nicht ausreichen. Darüber hinaus gibt es noch eine ganze Reihe von weiteren Modulen, die Sie installieren müssen, einschließlich `libnet`, `HTML-Parser` und einige andere. All die Module, die wiederum von anderen Modulen benötigt werden, zu installieren, kann einem manchmal endlos vorkommen (Modul A benötigt Modul B, das wiederum Modul C benötigt und so weiter). Sie sollten sich aber die Zeit dafür nehmen, denn diese Module sind wichtig und sehr nützlich. Die **README**-Datei zu der Bibliothek `libwww` teilt Ihnen genau mit, welche Module Sie herunterladen und installieren müssen.*

Das `LWP`-Paket enthält eine Reihe von Modulen für den Umgang mit dem Web und seinen Webseiten. Die meisten dieser Module sind objektorientiert und bieten eine große Bandbreite an Möglichkeiten. Wenn Sie interessiert sind, empfehle ich Ihnen, die POD-Dokumentation zu diesen Modulen zu lesen. Für unser spezielles Skript benötigen wir allerdings nur eine einfache Lösung, um eine Webseite von einem Webserver in das Skript zu holen, und dafür reicht das Modul `LWP::Simple` fraglos aus.

`LWP::Simple` ist, wie der Name schon verrät, ein simpler Satz an Subroutinen zur Bewältigung einfacher Aufgaben. Aus diesem Satz haben wir die Subroutine `get` gewählt, die bei Übergabe eines URL als Argument die zugehörige Website über das Internet ansteuert, die Datei hinter dem URL abruft und als einen großen String zurückliefert. In Zeile 62 des Skripts ***meinehomepage.pl*** wird die Funktion wie folgt aufgerufen:

```
my $content = get($data{'url'});
```

Diese Zeile ist schnell erklärt: Der Hash `%data` enthält den Datensatz für diesen Abschnitt der Homepage, wobei der URL in dem URL-Schlüssel gespeichert ist. Den URL übergeben wir als Argument an die Subroutine `get` und speichern das Ergebnis in der Skalarvariable `$content`. Weiter hinten, im Abschnitt über `eval`, zeige ich Ihnen, wie Sie die wichtigen Teile aus dem Inhalt von `$content` herausziehen.

Das andere Modul, das wir in diesem Skript verwenden, lautet `URI::URL`. URI leitet sich von ***Uniform Resource Identifier*** ab und ist eine Art generische Version von URL (***Uniform Resource Locator***). Das URL-Modul ist ebenfalls objektorientiert und bietet verschiedene Möglichkeiten, um URLs aller Art zu erzeugen, zu ändern oder zu analysieren. Wir benötigen es in den Code-Abschnitten der `config`-Datei. Lassen Sie uns eine Zeile aus der `config`-Datei betrachten, die ich Ihnen oben bereits vorgestellt habe:

```
print url($1,$data{'url'})->abs->as_string, "\">\n";
```

Dieses kleine Codefragment verwendet drei Subroutinen (eigentlich Methoden, wenn wir uns in Erinnerung zurückrufen, dass dies ein objektorientiertes Modul ist): `url`, `abs` und `as_string`. Durch die Zusammenarbeit

dieser drei Methoden, kann man einen relativen URL - wie Sie ihn in Webseiten vorfinden, die Sie gerade aus ihrem Home- Verzeichnis ihres Webservers herausgelöst haben - in einen vollständigen, absoluten URL konvertieren. Mit diesem Code stellen wir sicher, dass alle Links in dem Code unserer Homepage auch tatsächlich auf die korrekten Websites verweisen. Wenn Sie an weiteren Informationen zu diesem Thema interessiert sind, lesen Sie die POD-Dokumentation zu der `URI::URL`-Manpage.

Um ehrlich zu sein: Ich habe diesen Code einem großartigen Dokument, dem `LWP cookbook`, entnommen, das mit dem `LWP`-Modul erhältlich ist. Es enthält eine Fülle von Beispielen für die Lösung von häufigen Aufgaben mit `LWP`. Wenn Sie dieses Skript ausbauen oder einfach nur mehr mit dem Web und Perl arbeiten wollen, konsultieren Sie erst einmal dieses Kochbuch. Vielleicht finden Sie dort ja eine Lösung zu Ihrem Problem.

Die Funktion *eval*

Hier kommt die Quizfrage: Wie kann man generischen Code aufsetzen, der wichtige Daten aus einer HTML-Datei extrahiert, wenn sich die wichtigen Teile von Datei zu Datei unterscheiden. Die Antwort lautet: Es geht nicht generisch, aber halten Sie den Code vom Hauptskript getrennt. In ***meinehomepage.pl*** finden Sie keinen Code zum Extrahieren der uns interessierenden Teile einer HTML-Datei. Dieser Code - normalerweise eine ganze Reihe von regulären Ausdrücken - befindet sich in der Konfigurationsdatei, zusammen mit den Titeln und den URLs, und unterscheidet sich je nach Seite.

Und nun die nächste Frage: Wie bekommen Sie den Code in Ihr Skript? Der erste Schritt besteht einfach darin, den Code aus der Konfigurationsdatei auszulesen. Der Teil des Skripts, der die Konfigurationsdatei einliest, legt den ganzen Codeabschnitt als ein einziges Element in einem Hash mit dem Schlüssel `'code'` ab.

Das letzte Teil in unserem Puzzle ist die `eval`-Funktion. Diese Funktion übernimmt, wie ich gestern erklärt habe, einen String und führt diesen String aus, als handele es sich um Perl-Code. In diesem Fall besteht unser String aus Perl-Code, so dass `eval` gleichbedeutend mit Kopieren und Einfügen an dieser Stelle ist. Der Code, der mit `eval` ausgeführt wird, hat Zugriff auf alle Variablen, die im Vorfeld definiert wurden, so dass wir den Hash `%data` und den String `$content` dazu nutzen können, den HTML-Code zu verarbeiten und auszugeben.

Sehen Sie hier noch einmal das Beispiel aus der `config`-Datei. Es extrahiert den URL der Comic-Strip-Grafik (bestehend aus dem Namen `dilbert`, einer Reihe von Zahlen und der Extension `.gif`) und erzeugt ein neues `IMG`-Tag in unserer Homepage (wobei der URL zu seinem absoluten Namen expandiert wird).

```
code={
   if ($content =~ /src=\"(.*dilbert\d+.gif)\"/i) {
        print "<P><IMG SRC=\"";
        print url($1,$data{'url'})->abs->as_string, "\">\n";
    }
}
```

Der HTML-Code, der durch dieses bißchen Code erzeugt wird, sieht ungefähr folgendermaßen aus:

```
<P><IMG SRC="http://www.unitedmedia.com/comics/dilbert/archive/images/
dilbert973012490104.gif">
```

Die `eval`-Funktion wird auch für den `'other'`-Teil der `config`-Datei benötigt, der einen Hash mit den zusätzlichen Links enthält. Ich hätte diese Schlüssel und URLs auch anders formatieren und in einem Hash des Skripts verarbeiten können, aber so war es einfacher. Die Zeilen 70 bis 78 in dem Skript ***meinehomepage.pl*** zeigen, wie die anderen Links ausgewertet und auf der HTML-Seite angezeigt werden.

Noch eine Konfiguration

Die Musterkonfigurationsdatei war ziemlich einfach. Deshalb möchte ich noch eine weitere, etwas komplexere besprechen, allerdings vom Standpunkt des Skript ***meinehomepage.pl*** aus. Konzentrieren wir uns auf die Subroutine `procsection` aus ***meinehomepage.pl***, wie sie in Listing 21.3 wiedergegeben wird:

Listing 21.3: Die Subroutine procsection

```
1:  sub procsection {
2:      my %data = @_;
3:
4:      print "<H2><FONT FACE=\"Helvetica,Arial\">";
5:      print "$data{'title'}</FONT></H2>\n";
6:
7:      my $content = get($data{'url'});
8:      if (defined $content) {
9:         eval $data{'code'};
10:     } else {
11:        print "<P><B> Hoppla!</B> Kann nicht auf die Daten zugreifen \n";
12:        exit;
13:     }
14:
15:     if (defined $data{'other'}) {
16:     my %others = eval $data{'other'};
17:
18:     print "<P><B>Weitere Links:</B> <BR>\n";
19:     foreach (keys %others) {
20:         print "<A HREF=\"$others{$_}\">$_</A><BR> ";
21:     }
22:     print "\n";
23:     }
24: }
```

Beginnen wir oben, wo wir das Datensatz-Hash-Argument in einer lokalen Variablen speichern und den obersten Teil des Abschnitts ausgeben (indem wir von dem Titel- Teil aus der Konfigurationsdatei in Zeile 5 Gebrauch machen).

In Zeile 7 wird mit Hilfe der LWP-Subroutine `get` die HTML-Datei angefordert. Im Erfolgsfall (Zeile 9) werten wir den Codeabschnitt der Konfigurationsdatei aus. Im anderen Fall geben wir einen Fehler aus und gehen zum nächsten Abschnitt über.

Lassen Sie uns einen Blick auf den Code aus der Konfigurationsdatei werfen (Listing 21.4). Dieser Code beschreibt den Wetterteil der Seite und zeichnet eine ganze Reihe von Tabellenzellen in die Mitte der Seite.

Listing 21.4: Konfigurationscode für den Wetterabschnitt

```
1:  {
2:      my $table = "<table>\n";
3:      while ($content =~ /(<tr.*?<\/tr>)/gis) {
4:          my $row = $1;
5:          if ($row =~ /FOUR-DAY/is) { # Überschriftenzeile
6:              $table .= $row;
7:          }
8:          if ($row =~ /<strong>(Fri|Sat|Sun|Mon|Tue|Wed|Thu)\./is) {
9:              $table .= $row;
10:         }
11:     }
12:     $table =~ s/src="([^\"]+)"/'src="' .
13:          url($1,$data{'url'})->abs->as_string . '"'/eisg;
14:     $table =~ s/"#FF[9C]+66"/"#8FCDFF"/g; # #FF9966 oder #FFCC66
15:
16:     $table .= "</table>";
17:     print "$table\n";
18: }
```

Der größte Teil dieses Codes besteht aus regulären Ausdrücken, doch wenn Sie eine Vorstellung davon haben, wonach wir suchen, ist das Entziffern des Codes einfacher. In Abbildung 21.2 sehen Sie das, was wir als Ergebnis anstreben: Dazu gehören eine oberste Zeile (»Four-Day Forecast«) und vier lange Zellen mit den Wettergrafiken und den Temperaturangaben.

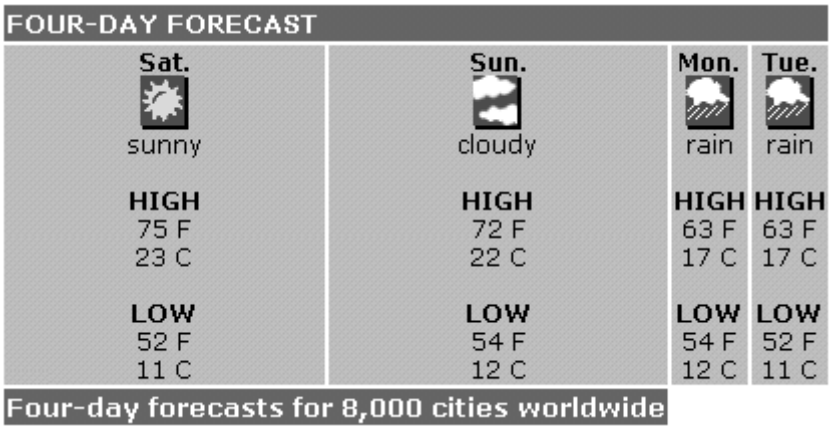

Abbildung 21.2: Der Wetterteil meiner Homepage

Jedes dieser Elemente wird in der Webseite durch Tabellenreihen repräsentiert. Deshalb sucht der erste reguläre Ausdruck in Zeile 3 nach Code für Tabellenreihen - das heißt Text, der von den Tags `<TR>` und `</TR>` umgeben ist. In einer Hinsicht ist dieser reguläre Ausdruck allerdings etwas tükkisch - er verwendet ein Muster, das Sie in den Kapiteln 9 und 10 nicht direkt gelernt haben (es wurde zwar im Vertiefungsabschnitt angesprochen, aber fand im Hauptteil keine Erwähnung). Vielleicht erinnern Sie sich, dass `.*` ein gieriger Quantifizierer ist, der mit Freude alles bis ans Ende der Reihe »aufsaugt«. Das ist zwar nicht gerade eine glückliche Lösung, aber leider hier nicht zu vermeiden, da wir eine Übereinstimmung bis `</TR>` und nicht bloß für ein einziges Zeichen benötigen. Aus diesem Grunde verwenden wir `.*?`, eine spezielle nichtgierige Version des Musters `.*` - es führt den Abgleich bis zum ersten Vorkommen von `</TR>` durch und hält dann an, anstatt alles abzugleichen und danach von hinten zu prüfen.

Das einzige, was an dem Ausdruck in Zeile 3 noch bemerkenswert ist, sind die Optionen: `g` für global, um der `while`-Schleife die Gelegenheit zu geben alles abzugleichen, `i` für eine Suche, die nicht zwischen Groß- und Kleinschreibung unterscheidet, und `s`, damit der Punkt (`.`) zeilenübergreifend abgleicht (zur Erinnerung: die gesamte HTML-Datei ist mitsamt seinen Neue-Zeile-Zeichen als ein String in `$content` gespeichert).

Sind wir auf Tabellenreihen gestoßen, besteht der nächste Schritt darin, nach den gewünschten Tabellenreihen Ausschau zu halten. Die erste ist die dunkle oberste Reihe in Zeile 5, die einfach nur Textzeichen enthält. Darauf folgen die eigentlichen Wetterreihen, die durch Abgleich mit den Tagesnamen (Zeile 8) ermittelt werden, wobei das Muster mit einem `<strong>`-Tag beginnt und mit einem Punkt endet. Bei jedem Abgleich hängen wir den übereinstimmenden Code an den String `$table` an und bauen so die HTML-Tabelle für unsere Webseite auf.

In den Zeilen 12 bis 14 polieren wir die so aufgebaute Tabelle noch etwas auf. Zeile 12 bedient sich des URL-Codes, den Sie im vorigen Abschnitt kennengelernt haben, und expandiert die URLs der Grafikdateien, so dass sie auch wirklich auf die zugehörigen Websites zeigen. Zeile 14 ist eher persönlicher Art: Hier habe ich die Hintergrundfarben der Wetterreihen, die verschiedene Orangetöne aufweisen, in Blau geändert (ich mag einfach kein Orange).

In den Zeilen 16 und 17 wird die Tabelle nur noch mit einem schließenden `</table>`- Tag abgeschlossen; anschließend wird das Ergebnis ausgegeben. Von hier aus springen wir zurück zu unserem Skript ***meinehomepage.pl***, geben die verschiedenen Links in dem `other`-Hash aus und erhalten als Ergebnis eine individuelle Webseite, die nur das enthält, was Sie sehen wollen.

Dieses System ist natürlich nicht unfehlbar. Werden die Quellseiten überarbeitet oder zu anderen URLs verschoben, müssen Sie Ihre Konfigurationen ändern. Wenn Sie Glück haben und Ihren Code so allgemein wie möglich halten, müssen Sie nicht allzu oft Änderungen vornehmen - dies ist einer der Nachteile, wenn man sich auf die Webseiten dritter verlassen muss.

Ein webbasierter Aufgabenlisten-Manager

Unser zweites - und letztes - Beispiel ist eine einfache Aufgabenlisten-Anwendung namens ***Aufgabenliste.pl***, die im Web ausgeführt werden kann. Sie können Listeneinträge hinzufügen, löschen und ändern, nach Datum, Priorität oder Beschreibungen sortieren und Einträge als fertig markieren. In Abbildung 21.3 sehen Sie ein Beispiel für eine

Aufgabenliste.

Die Aufgabenlisten-Anwendung besteht aus einer großen Tabelle, in der die einzelnen Aufgaben eingetragen sind. Jeder Eintrag besteht aus einem Markierungskästchen, das anzeigt, ob die Aufgabe erledigt ist oder nicht, sowie Feldern für die Priorität, die Fälligkeit und die Beschreibung. Die Daten dieser Tabelle können über die Formularelemente, aus denen die Tabelle aufgebaut ist, bearbeitet werden. Die Änderungen werden gültig, wenn der Benutzer den Schalter **Aktualisieren** drückt. Auch das Entfernen eines Elements (die Markierungskästchen in der ganz rechten Spalte der Tabelle), das Sortieren nach einem Feld (Auswahlfeld **Sortieren nach** unter der Tabelle) und die Entscheidung, ob erledigte Aufgaben angezeigt werden sollen oder nicht (das Markierungskästchen **Erledigte Einträge anzeigen** ebenfalls unter der Tabelle) werden erst nach Betätigen von **Aktualisieren** gültig.

Außer der Tabelle und den Anzeigeoptionen gibt es noch einen Bereich zum Einfügen neuer Einträge in die Liste. Nachdem Sie die Felder in diesem Teil der Anwendung ausgefüllt und **Neuer Eintrag** ausgewählt haben, werden diese Eingaben als neuer Eintrag mit aufgenommen (und alle anderen Änderungen werden ebenfalls gültig).

Wie schon bei ***Meine Homepage*** ist das der Aufgabenliste zugrundeliegende Skript ein CGI-Skript, das direkt als URL ausgeführt wird und kein Formular benötigt. Das Skript erzeugt seinen eigenen Inhalt, einschließlich der Formulare, mit denen Sie die Elemente in der Liste oder deren Anzeige ändern können. Alles geschieht in einem Skript. Der einzige zusätzliche Teil ist eine Datendatei - ***listdaten.txt*** -, die die Daten der Aufgaben speichert und von dem Aufgabenskript gelesen und beschrieben werden kann.

Abbildung 21.3: Die Web-Anwendung einer Aufgabenliste

Dieses Skript ist doppelt so lang wie alles, was wir bisher in diesem Buch behandelt haben. Deshalb werde ich, wie schon im Beispiel zuvor, nicht jede Zeile einzeln besprechen. Das vollständige Skript befindet sich am Ende dieses Abschnitts in Listing 21.5 und ist darüber hinaus auf der CD und der Website zu diesem Buch erhältlich. In diesem Abschnitt beschreibe ich den Fluß und die allgemeine Struktur dieses Skripts. Sollten Sie über das Besprochene hinaus noch Fragen dazu haben, gehen Sie den Code ruhig einmal selbst durch.

Die Datendatei

Wie die meisten Beispiele, die wir diese Woche betrachtet haben, hat auch dieses Skript eine Datendatei, aus der es liest und in die es schreibt, um die Daten auf dem aktuellen Stand zu halten. Die Datendatei zu diesem Skript, genannt ***listdata.txt***, speichert die Daten der einzelnen Aufgaben. Das Skript ***Aufgabenliste.pl*** liest diese Datei bei jeder Iteration und schreibt bei jeder Änderung die neuen Daten zurück. Diese Datei ähnelt stark den Datendateien, die Sie bereits in anderen Beispielen kennengelernt haben:

```
desc=Kapitel 21 beenden
date=23.10.1998
prior=1
done=0
---
desc=Punkt 3
date=05.12.1998
prior=3
done=1
---
desc=Pyramide bauen
date=10.01.1999
prior=5
done=0
---
desc=Kapitel 20 beenden, überarbeiten
date=31.10.1998
prior=1
done=0
---
desc=Treffen mit Fred zum Essen
date=05.01.1999
prior=2
done=0
---
```

Wie schon in der Datendatei für das ***Meine Homepage***-Skript werden die einzelnen Datensätze in der Datei durch drei Bindestriche (`---`) getrennt. Jedes Feld in dem Datensatz hat einen Schlüssel und einen Wert, die durch ein Gleichheitszeichen voneinander getrennt sind. Im Gegensatz zu den Daten für ***Meine Homepage*** gibt es hier keine mehrzeiligen Datensätze, sondern nur Schlüssel und Werte.

Wenn das CGI-Skript für die Aufgabenliste installiert wird, muss auch eine Datendatei angelegt werden, die der Webserver lesen und beschreiben kann. Diese Datendatei kann leer sein - das Skript wird dann eine Webseite ohne Einträge erzeugen -, muss aber vorhanden sein, damit das Skript funktioniert.

Wie das Skript funktioniert

Das Skript ***Aufgabenliste.pl*** ist zwar groß, aber nicht sonderlich kompliziert. Es gibt nur wenige verwirrende reguläre Ausdrücke, und der Fluß von Funktion zu Funktion ist auch ziemlich überschaubar. Genau genommen besteht ein Großteil des Skripts aus `print`-Anweisungen, um den HTML-Code für die Aufgabenliste und ihre verschiedenen Formularelemente zu erzeugen - und diese Elemente so anzupassen, dass sie sich in verschiedenen Situationen unterschiedlich verhalten.

Die ersten Subroutinen für das Skript ***Aufgabenliste.pl*** lauten `&init()` und `&process()`. `&init()` legt das aktuelle Datum fest, ruft die Subroutine `&read_data()` auf und gibt den obersten Teil der HTML-Datei aus, die von dem Skript erzeugt werden soll. An diesem Punkt wollen wir mit der Besprechung beginnen und uns von dort nach unten vorarbeiten.

Dateninitialisierung mit &init()

Die Hauptaufgabe der Initialisierungs-Subroutine besteht darin, dass `&read_data()` aufgerufen wird, so dass die Datendatei geöffnet und die einzelnen Einträge in eine Datenstruktur eingelesen werden. Bei dieser Datenstruktur handelt es sich um ein Array von Hashes, wobei die einzelnen Hashes die Daten für die verschiedenen Aufgaben der Liste enthalten. Die Schlüssel in dem Hash lauten:

- `desc` - die Beschreibung des Eintrags

- `date` - das Fälligkeitsdatum für das Element. Das Datum hat das Format `MM/TT/ JJJJ` (dieses Format wird vom Skript verlangt)
- `prior` - die Priorität des Elements von 1 (höchste Priorität) bis 5
- `done` - zeigt an, ob die Aufgabe erledigt ist oder nicht

Zusätzlich zu diesen Schlüsseln, die aus der Datendatei kommen, bekommt jeder Eintrag in der Liste noch eine ID-Nummer. Dieser ID werden während des Einlesens Werte von 0 bis »Anzahl der Einträge« zugewiesen. Die ID wird später benötigt, um die verschiedenen Formularelemente für die Listenelemente auseinanderzuhalten.

Das Formular und die Daten verarbeiten - *&process()*

Die Routine `&process()` erledigt den wesentlichen Teil der Arbeit in dem Skript. In dieser Subroutine gibt es zwei Hauptzweige, die auf der `param()`-Funktion aus dem Modul `CGI.pl` basieren. Am Tag 16 habe ich Ihnen `param()` vorgestellt, und Sie haben gelernt, wie man diese Funktion dazu nutzt, um Werte aus Formularelementen auszulesen. Die Funktion `param()` läßt sich allerdings auch ohne Argumente anwenden, in welchem Falle sie die Namen aller vorhandenen Formularelemente oder - falls das Skript nicht für ein Formular aufgerufen wurde - den undefinierten Wert (`undef`) zurückliefert. Dieses Verhalten machen wir uns in der Subroutine `&process()` zunutze, um zwei verschiedene Ergebnisse zu erhalten:

- Beim ersten Aufruf des Skripts gibt es keine Parameter. Deshalb wird lediglich die aktuelle Aufgabenliste (mit der Subroutine `&display_all()`) angezeigt.
- Wenn das Skript danach erneut aufgerufen wird, gilt es auf mögliche Änderungen zu reagieren. Das können entweder Änderungen an bestehenden Listeneinträgen oder neu hinzugefügte Einträge sein. Wird der Schalter **Aktualisieren** betätigt, werden alle zu löschenden Elemente gelöscht (mit der Subroutine `&remove_selected()`), alle Daten aktualisiert (mit der Subroutine `&update_data()`), die Daten zurück in die Datei geschrieben (mit der Subroutine `&write_data()`) und diese dann wieder angezeigt (mit der Subroutine `&display_all()`).
- Wurde der Schalter **Neuer Eintrag** gedrückt, gehen wir analog vor, außer dass zwischen dem Aktualisieren der Daten und dem Zurückschreiben in die Datei ein Zwischenschritt dafür Sorge trägt, dass mit dem Aufruf an `&add_item()` eine weitere Aufgabe der Aufgabenliste als neuer Eintrag hinzugefügt wird.

Während der ganzen Aktualisierungs- und Hinzufügeoperationen prüfen wir gleichzeitig auf Formatierungsfehler in den Daten. Doch mehr dazu später, wenn ich auf die Aktualisierung der Daten und das Hinzufügen von neuen Listenelementen zu sprechen komme.

Die Daten mit *&display_all()* und *&display_data()* anzeigen

Der größte Teil des Aufgabenlisten-Skripts steckt allerdings in den Subroutinen `&display_all()` und `&display_data()`. Diese Subroutinen erzeugen nicht nur den HTML-Code für die Daten; die Datentabelle ist auch ein Formular, und alle Elemente in diesem Formular müssen automatisch erzeugt werden. Außerdem ist ein Großteil des HTML-Codes einer Reihe von Bedingungen unterworfen, die sich aus den Zuständen der Tabelle ergeben. So werden zum Beispiel Aufgaben mit der Priorität 1 in Rot ausgegeben, und die Auswahlfelder für die Prioritäten müssen so initialisiert werden, dass sie die aktuellen Werte der Datensätze widerspiegeln. Anstatt also für diesen Teil des Skripts ein riesiges Hier-Dokument zu verwenden, müssen wir Zeile für Zeile durchgehen, um es zu erzeugen.

Die Subroutine `&display_all()` ist die Hauptroutine, die als erstes aufgerufen wird. Sie startet die Tabelle, gibt die Header aus, sortiert das Hauptarray nach der aktuell vorgegebenen Sortierreihenfolge und ruft dann innerhalb einer `for`-Schleife `&display_data()` auf, um die einzelnen Einträge in der Aufgabenliste auszugeben. Außerdem erzeugt es die Elemente unter den Daten: das Auswahlfeld **Sortieren nach**, das Markierungskästchen **Erledigte Einträge anzeigen**, die Formularelemente zum Hinzufügen eines neues Eintrags und die beiden Schalter zum Abschicken des Formulars. Nebenbei kontrolliert die Subroutine, ob ein Fehler bei der Verarbeitung der Datumsangaben aufgetreten ist, und gibt entsprechende Warnungen aus. Für all dieses benötigen Sie eine Menge von bedingten Befehlen und `print`- Anweisungen sowie eine Menge Zeilen HTML-Code.

Der Subroutine `&display_data()` obliegt die Aufgabe, die einzelnen Einträge in der Aufgabenliste zu verarbeiten. Jede Reihe der Tabelle besteht aus fünf Spalten, die jeweils ein Formularelement enthalten. Jedes Element bedarf eines eindeutigen Namens, und viele Formularelemente ändern ihre Erscheinung, je nachdem welche Daten sie widerspiegeln (so muss zum Beispiel das Markierungskästchen am Beginn der Reihe markiert sein, wenn eine Aufgabe erledigt ist). Die Subroutine `&display_data()` sorgt auch dafür, dass Einträge NICHT angezeigt werden -

wenn das Markierungskästchen **Erledigte Einträge anzeigen** nicht markiert ist, werden die Aufgaben, die als erledigt markiert wurden, nicht angezeigt. Es wird aber ein verborgenes Formularelement erzeugt, das einige der Daten der Aufgabe enthält, so dass die Aktualisierungen ordnungsgemäß durchgeführt werden können.

Wie schon bei `&display_all()` bedeutet auch dies eine Menge von `if`-Anweisungen und viel HTML-Code. Das Ergebnis ist ein gigantisches Formular, bei dem jedes Formularelement mit einer Aufgabe verbunden ist; es zeigt stets die aktuellen Daten der Aufgabenliste an. Wenn Sie irgend etwas an diesen Daten ändern, werden die Änderungen beim Abschicken des Formulars in die Originaldatensammlung übernommen.

Änderungen mit *&update_data()* übernehmen

Da wir inzwischen beim Eintragen der Änderungen angelangt sind, möchte ich auf die Subroutine `&update_data()` zu sprechen kommen. Diese Subroutine wird sowohl für den Schalter **Aktualisieren** als auch für den Schalter **Neuer Eintrag** aufgerufen, denn es soll sichergestellt werden, dass alle Änderungen am Skript in beiden Fällen übernommen werden. Die Aufgabe von `&update_data()` besteht darin, alle Formularelemente der Seite zu durchlaufen - die Element für die Listeneinträge und die Formularelemente **Sortieren nach** und **Erledigte Einträge anzeigen** - und die Daten oder globalen Einstellungen entsprechend den Änderungen, die auf einer Webseite vorgenommen wurden, zu aktualisieren.

Doch konzentrieren wir uns auf die Daten selbst. Jedes Element des HTML- Formulars, das durch `&display_data()` erzeugt wird, trägt einen eindeutigen Namen, der aus dem Namen des Feldes (Beschreibung, Priorität und so weiter) und der ID- Nummer dieses Elements erzeugt wird. Durch Teilen der Liste der Formularelementnamen, die zurückgeliefert werden, wenn das Formular abgeschickt wurde, können wir feststellen, welcher Name zu welchem Teil eines Datensatzes gehört, und die Werte vergleichen. Sollten diese nicht identisch sein, aktualisieren wir den Datensatz mit dem neuen Wert. Jedesmal, wenn das Formular abgeschickt wird, werden die Elemente einzeln geprüft. Das ist zwar nicht unbedingt der effizienteste Weg, Änderungen in den Daten zu verfolgen, aber wir können dadurch alles auf einer Seite belassen.

Eine andere Aufgabe, die die Subroutine `&update_data()` noch erfüllt, ist die Prüfung auf fehlerhafte Datumsangaben in den bestehenden Daten. Wenn Sie versuchen, ein Datum hinsichtlich seines normalen Formats zu ändern (`10/9/1998` oder so ähnlich), wird dies von `&update_data()` aufgefangen und ein Fehler generiert, der dann beim nächsten Aufruf von `&display_all()` für den Datensatz angezeigt wird.

Elemente mit *&add_item()* hinzufügen und mit *&remove_selected()* entfernen

Um Einträge aus der Liste zu entfernen, markieren Sie die Markierungskästchen in der **Entfernen**-Spalte für die betreffenden Daten und klicken auf **Aktualisieren**. Um ein Element der Liste hinzuzufügen, geben Sie diese Daten am Ende der Seite in das Formular ein und klicken auf **Neuer Eintrag**. In beiden Fällen wird `&remove_selected()` aufgerufen, im letzteren Fall darüber hinaus `&add_item()`.

Die Subroutine `&remove_selected()`aktualisiert die Daten, indem die vom Benutzer zum Löschen vorgesehenen Datensätze entfernt werden. In diesem Fall ist das Entfernen der Elemente einfach, da unsere ganzen Daten in einem Array von Referenzen gespeichert sind. Wir richten einfach ein weiteres Array von Referenzen ein, abzüglich der, die wir löschen wollen, und legen dieses neue Array in der Variablen des alten Arrays ab. Da es sich hierbei um ein Array von Referenzen handelt, bleiben alle Daten, auf die mit den Referenzen verwiesen wird, an ihrer Position und müssen nicht umkopiert werden.

Die Subroutine `&add_item()` ist ebenfalls einfach. Wir müssen die Daten aus den Formularelementen nur in einem Hash ablegen und in dem Datenarray eine Referenz auf diesen Hash einrichten. Wir weisen dem neuen Eintrag außerdem eine neue ID zu, die um eins höher liegt als die momentan größte ID. (Wenn dazwischenliegende Einträge gelöscht wurden, versuchen wir nicht, deren IDs neu zu vergeben - die Einträge erhalten ehedem neue IDs, wenn die Datei bei der nächsten Iteration des Skripts wieder eingelesen wird).

Andere Subroutinen: Daten fortschreiben und auf Fehler prüfen

Übriggeblieben sind jetzt nur noch ein paar kleine, aber wichtige Subroutinen: `&write_data()`, um die Daten zurück in die Datei ***listdaten.txt*** zu schreiben, und zwei Subroutinen für die Datenformate und die Vergleiche.

Die Subroutine `&write_data` ist einfach: Ihre einzige Aufgabe besteht darin, die Datei ***listdaten.txt*** zum

Schreiben zu öffnen und dann die Datensätze durchzugehen und in die Datei zu schreiben. Da diese Subroutine jedesmal, wenn das Skript ausgeführt wird und nachdem irgendwelche Änderungen an den Daten vorgenommen wurden, aufgerufen wird, können wir fast absolut sicher sein, dass die Daten nie korrumpiert werden oder Einträge verlorengehen. Beachten Sie, dass die IDs der Einträge nicht mit dem Rest der Daten in die Datendatei geschrieben werden. Die IDs werden erzeugt, wenn die Daten anfangs eingelesen werden, und nur dazu benötigt, um die Formularelemente zuordnen zu können. Darum müssen Sie nicht zwischen den Skriptaufrufen gespeichert werden.

Die letzten zwei Subroutinesätze beziehen sich auf die Datumsangaben. Datumsangaben haben, wie ich bereits erwähnt habe, das Format `MM/TT/JJJJ`. Es ist wichtig, ein konsistentes Format zu verwenden, da es nur dadurch möglich ist, die Liste der Elemente nach dem Datum zu sortieren - eine Art von numerischer Sortierung. Um das Datumsformat in eine Zahl zu verwandeln, die dann mit einer anderen Zahl verglichen werden kann, muss die Formatierung stimmen. Aus diesem Grund wird jede geänderte oder mit einem neuen Eintrag hinzukommende Datumsangabe mit der Subroutine `&check_date()` auf ihr Format hin geprüft. Liegen Abweichungen im Format vor, wird ein Fehler ausgegeben (eine große rote Meldung oben auf der Webseite und Sternchen in dem falschen Datum).

Die Sortierung der Liste nach dem Datum erfolgt in der Subroutine `&display_all()`. Dazu muss der Wert im Menü **Sortieren nach** auf **Datum** eingestellt worden sein. Um die Datumsangaben in etwas zu konvertieren, das mit etwas anderem verglichen werden kann, verwenden wir das Modul `Time::Local`. Dabei handelt es sich um ein vorgegebenes Modul, das dazu verwendet werden kann, verschiedene Teile einer Datums- oder Zeitangabe in ein `time`-Format zu konvertieren - das ist die Anzahl der Sekunden seit 1900 (das ist der Wert, der von der `time`-Funktion zurückgeliefert wird). Diesem Zwecke dient die Subroutine `&date2time()`, die die korrekt formatierte Datumsangabe in ihre Elemente aufsplittet und den `time`-Wert zurückliefert. Diese Routine hält gleichzeitig Ausschau nach falsch formatierten Datumsangaben - mit führenden Sternchen - und sortiert diese Werte ganz nach oben.

Beachten Sie, dass wir für die Jahresangabe in allen Daten unseres Skripts eine vierstellige Zahl verwenden. Damit stellt der Jahreswechsel 2000 kein Problem für uns dar. `Time::Local` ist eigentlich recht gut darin, festzustellen, ob eine zweiziffrige Jahresangabe in diese Epoche fällt oder in die nächste. Aber ich fand, dass es besser sei, sicherzugehen. Denn es kann durchaus sein, dass Sie dieses Buch im Jahr 2000 lesen.

Der Code

Listing 21.5 enthält den (ziemlich) vollständigen Code für das Skript ***Aufgabenliste.pl***. Gehen Sie es von vorn nach hinten durch. Kritisch sind nur die Teile, die mit der Vergabe der IDs an die Formularelemente und mit der Behandlung von Fehlern in den Datumsangaben zu tun haben (achten Sie auf die Subroutine `&check_date()`). Und wie bei allen CGI-Skripts ist es von Vorteil, wenn Sie HTML- Kenntnisse haben und wissen, wie Formulare und `CGI.pm` miteinander agieren.

Listing 21.5: Der Code für Aufgabenliste.pl

```
1:   #!/usr/bin/perl -w
2:   use strict;
3:   use CGI qw(:standard);
4:   use CGI::Carp qw(fatalsToBrowser);
5:   use Time::Local;
6:
7:   my $listdata = 'listdaten.txt';  # Datendatei
8:   my @data = ();                  # Array von Hashes
9:   my $id = 0;                     # oberste ID
10:
11:  # global Standardeinstellungen
12:  my $sortby = 'prior';       # Sortierreihenfolge
13:  my $showdone = 1;           # Bearbeitete Einträge anzeigen?  (1 == ja)
14:
15:  &init();
16:  &process();
17:
18:  sub init {
19:      # aktuelles Datum ermitteln und im Format MM/TT/YY ausgeben
20:      my ($day,$month,$year) = 0;
21:      (undef,undef,undef,$day,$month,$year) = localtime(time);
22:      $month++;             # Monate starten bei 0
```

```
23:      $year += 1900;        # Perl-Jahre sind die Jahre seit 1900;
24:
25:      my $date = "$month/$day/$year";
26:
27:      # Datendatei öffnen und lesen
28:      &read_data();
29:
30:      # HTML-Code
31:      print header;
32:      print start_html('Meine Aufgabenliste');
33:      print "<H1 ALIGN=CENTER><FONT FACE=\"Helvetica,Arial\" SIZE+2>";
34:      print "Aufgabenliste</FONT></H1>\n";
35:      print "<H2 ALIGN=CENTER><FONT FACE=\"Helvetica,Arial\" SIZE+2>";
36:      print "$date</FONT></H2>\n";
37:      print "<HR>\n";
38:      print "<FORM ACTION=\"/cgi-bin/Aufgabenliste.pl\" METHOD=POST>\n";
39:  }
40:
41:  sub process {
42:      my $dateerror = 0;      # Fehler im Datumsformat der alten Liste
43:      my $newerror = 0;       # Fehler im Datumsformat des neuen Eintrags
44:
45:      # Hauptverzweigung.  Es gibt zwei Möglichkeiten:
46:      # keine Parameter, alles anzeigen
47:      # irgendwelche Parameter: aktualisieren, Eintrag hinzufügen,
         # schreiben und anzeigen
48:      if (!param()) {                 # nur beim ersten Mal
49:          &display_all();
50:      } else { # Schalter behandeln
51:          &remove_selected();
52:          $dateerror = &update_data(); # vorhandene Änderungen
                                         # aktualisieren
53:
54:        # Elemente hinzufügen
55:          if (defined param('additems')) {
56:              $newerror = &check_date(param('newdate'));
57:              if (!$newerror) {
58:                  &add_item();
59:              }
60:          }
61:
62:        &write_data();
63:          &display_all($dateerror,$newerror);
64:      }
65:
66:    print end_html;
67:  }
68:
69:  # Datendatei in ein Array von Hashes einlesen
70:  sub read_data {
71:      open(DATA, $listdata) or
                  die " Datendatei kann nicht geöffnet werden: $!";
72:      my %rec = ();
73:      while (<DATA>) {
74:          chomp;
75:          if ($_ =~ /^\#/) { next;} # Kommentare
76:
77:          if ($_ ne '---' and $_ ne '') { # Datensatz erstellen
78:              my ($key, $val) = split(/=/,$_,2);
79:              $rec{$key} = $val;
80:          } else {                # Ende des Datensatzes
81:              $rec{'id'} = $id++;
82:              push @data, { %rec };
83:              %rec = ();
84:          }
85:      }
86:      close(DATA);
87:  }
88:
89:  # HTML-Ausgabe
90:  sub display_all {
91:      my $olderror = shift;              # ist ein Fehler aufgetreten?
92:      my $newerror = shift;
93:
```

```
94:
95:      print "<TABLE BORDER WIDTH=75% ALIGN=CENTER>\n";
96:
97:      if ($olderror or $newerror) {
98:          print "<P><FONT COLOR=\"red\">
                    <B>Fehler:  Daten, die mit *** markiert sind";
99:          print "haben falsches Format (verwende MM/TT/JJJJ)</B>
                    </FONT><P>\n";
100:     }
101:
102:     print "<TR BGCOLOR=\"silver\"><TH>Fertig?<TH>Priorität";
103:     print "<TH>Fälligkeitsdatum
                <TH ALIGN=LEFT>Beschreibung<TH>Entfernen?</TR>\n";
104:
105:     # Sortierreihenfolge festlegen (numerisch oder alphabetisch)
106:     my @sdata = ();
107:
108:     # Array von Hashes gemäß Wert von $sortby sortieren
109:     if ($sortby eq 'date') {   # nach Datum sortieren
110:         @sdata = sort {&date2time($a->{'date'}) <=>
111:                              &date2time($b->{'date'})} @data;
112:     } else { # Text/Priorität-Sortierung
113:         @sdata = sort {$a->{$sortby} cmp $b->{$sortby}} @data;
114:     }
115:
116:     # Einträge der Reihe nach ausgeben
117:     foreach (@sdata) {
118:         &display_data(%$_);     # Datensatz verarbeiten
119:     }
120:
121:     print "</TABLE>\n";
122:
123:     # Tabelle für Einstellungen
124:     print "<P><TABLE BORDER WIDTH=75% ALIGN=CENTER>\n";
125:     print "<TR><TD ALIGN=CENTER><B>Sortieren nach:</B> <SELECT
         NAME=\"sortby\">\n";
126:
127:     # Wert aus sortby abfragen, Auswahlfeld anzeigen
128:     print "<OPTION VALUE=\"prior\"  ";
129:     if ($sortby eq 'prior') { print "SELECTED>"}
130:     else { print ">"; }
131:     print "Priorität\n";
132:     print "<OPTION VALUE=\"date\" ";
133:     if ($sortby eq 'date') { print "SELECTED>"}
134:     else { print ">"; }
135:     print "Datum\n";
136:     print "<OPTION VALUE=\"desc\" ";
137:     if ($sortby eq 'desc') { print "SELECTED>"}
138:     else { print ">"; }
139:     print "Beschreibung\n";
140:     print "</SELECTED></TD>\n";
141:
142:
143:     # Wert von showdone abfragen, Markierungskästchen anzeigen
144:     print "<TD ALIGN=CENTER WIDTH=50%><B>
               Erledigte Einträge anzeigen?<B>\n";
145:     my $checked = '';
146:     if ($showdone == 1) { $checked = 'CHECKED'}
147:     print "<INPUT TYPE=\"checkbox\" NAME=\"showdone\"
          VALUE=\"showdone\"";
148:     print " $checked> </TD>\n";
149:
150:     # Aktualisieren-Schalter und Tabelle zum Hinzufügen von Einträgen
151:     print <<EOF;
152:     </TR></TABLE>
153:     <P><TABLE ALIGN=CENTER>
154:     <TR><TD ALIGN=CENTER VALIGN=CENTER>
155:     <INPUT TYPE="submit" VALUE="   Aktualisieren   "
          NAME="update"></TD></TR>
156:     </TABLE><HR>
157:     <TABLE ALIGN=CENTER>
158:     <TR><TH>Priorität<TH>Datum<TH ALIGN=LEFT>Beschreibung
159: EOF
160:     # Prioritäten-Auswahlfeld;
```

```
161:     print "<TR><TD><SELECT NAME=\"newprior\">\n";
162:     my $i;
163:     foreach $i (1..5) {         # Prioritäten von 1 bis 5
164:         if ($newerror and param('newprior') == $i) {
165:             $checked = 'SELECTED';
166:         }
167:         print "<OPTION $checked>$i\n";
168:     }
169:     print "</SELECT></TD>\n";
170:
171:     # Datum und Beschreibung ausgeben;
172:     # Fehlerfälle beachten
173:     my $newdate = '';
174:     my $newdesc = '';
175:     print "<TD ALIGN=CENTER><INPUT TYPE=\"text\" NAME=\"newdate\"";
176:     if ($newerror) {            # Fehler aufgetreten?
177:         $newdate = "***" . param('newdate');
178:        $newdesc = param('newdesc');
179:     }
180:     print "VALUE=\"$newdate\" SIZE=10></TD> \n";
181:
182:     # Beschreibung; im Fehlerfalle alten Wert beibehalten
183:     print "<TD><INPUT TYPE=\"text\" NAME=\"newdesc\"
          VALUE=\"$newdesc\"";
184:     print "SIZE=50></TD></TR></TABLE><TABLE ALIGN=CENTER>\n";
185:
186:    # fertigstellen
187:     print <<EOF;
188:     <TR><TD ALIGN=CENTER VALIGN=CENTER>
189:     <INPUT TYPE="submit" VALUE="Neuer Eintrag"
          NAME="additems"></TD></TR>
190:     </TABLE></FORM>
191: EOF
192: }
193:
194: # Daten zeilenweise ausgeben.  Daten sind bereits sortiert;
195: # einzelnen Datensatz ausgeben
196: sub display_data {
197:     my %rec = @_;               # auszugebender Datensatz
198:
199:     # Keine erledigten Einträge anzeigen, wenn die Option
200:     # Erledigte Einträge anzeigen nicht markiert ist.
201:     # Der Einfachheit halber die Einträge aber mit
202:     # verarbeiten
203:     if ($showdone == 0 and $rec{'done'}) {
204:         print "<INPUT TYPE=\"hidden\" NAME=\"Erledigt", $rec{'id'};
205:         print "\">\n";
206:         next;
207:     }
208:
209:     # Einträge mit Priorität 1 in Rot ausgeben
210:     my $bgcolor = '';
211:     if ($rec{'prior'} == 1) {
212:         $bgcolor = "BGCOLOR=\"red\"";
213:     }
214:
215:     # Erledigt oder nicht erledigt?
216:     my $checked = '';           # Erledigte Einträge
217:     if ($rec{'done'}) {
218:         $checked = 'CHECKED';
219:     }
220:
221:     print "<TR>\n";             # mit Reihe beginnen
222:
223:     # Erledigt-Kästchen
224:     print "<TD WIDTH=10% ALIGN=CENTER $bgcolor>";
225:     print "<INPUT TYPE=\"checkbox\" NAME=\"done", $rec{'id'};
226:     print "\" $checked></TD>\n";
227:
228:     # Priorität
229:     print "<TD WIDTH=10% ALIGN=CENTER $bgcolor>";
230:     print "<SELECT NAME=\"prior", $rec{'id'}, "\">\n";
231:     my $selected = '';
232:     my $i;
```

```
233:     foreach $i (1..5) {         # Prioritäten von 1 bis 5
234:        if ($rec{'prior'} == $i) {
235:            $selected = 'SELECTED';
236:         }
237:         print "<OPTION $selected>$i\n";
238:         $selected = '';
239:     }
240:     print "</SELECT></TD>\n";
241:
242:     # Datum
243:     print "<TD $bgcolor WIDTH=10% ALIGN=CENTER>";
244:     print "<INPUT TYPE=\"text\" SIZE=10 NAME=\"date", $rec{'id'}, "\" ";
245:     print "VALUE=\"", $rec{'date'}, "\"></TD>\n";
246:
247:     # Beschreibung
248:     print "<TD $bgcolor><INPUT TYPE=\"text\" NAME=\"desc", $rec{'id'};
249:     print "\" SIZE =50 VALUE=\"", $rec{'desc'}, "\"></TD>\n";
250:
251:     # Entfernen-Kästchen
252:     print "<TD $bgcolor ALIGN=CENTER>";
253:     print "<INPUT TYPE=\"checkbox\" NAME=r", $rec{'id'}, "></TD>";
254:
255:     # Ende der Reihe
256:     print "</TR>\n";
257: }
258:
259: # Alle Werte aktualisieren
260: sub update_data {
261:     my $error = 0;              # Fehlerbehandlung
262:
263:     # wurde showdone ausgewählt?
264:     if (defined param('showdone')) {
265:         $showdone = 1;
266:     } else { $showdone = 0; }
267:
268:     # Wert von sortby abfragen
269:     $sortby = param('sortby');
270:
271:     foreach (@data) {
272:         my $id = $_->{'id'};    # nicht das globale $id
273:
274:         # Einträge , die als erledigt markiert sind, können nicht
275:         # geändert werden. In so einem Fall können wir die
276:         # Überprüfung der weiteren Daten dieses Eintrags getrost
277:         # übergehen.
278:         if ($_->{'done'} == 1 && defined param('done' . $id)) { next; }
279:
280:         # Einträge, die als erledigt markiert wurden.
281:         if (defined param('done' . $id)) {
282:             $_->{'done'} = 1;
283:         } else { $_->{'done'} = 0; }
284:
285:         # Datum.  Auf fehlerhafte Datumsangaben prüfen
286:         if (param('date' . $id) ne $_->{'date'}) {
287:             $error = check_date(param('date' . $id));
288:             if ($error) {
289:                 $_->{'date'} = "***" . param('date' . $id);
290:             } else {
291:                 $_->{'date'} = param('date' . $id);
292:            }
293:         }
294:
295:         # Priorität, Beschreibung
296:         my $thing;
297:         foreach $thing ('prior', 'desc') {
298:             if (param($thing . $id) ne $_->{$thing}) {
299:                $_->{$thing} = param($thing . $id);
300:             }
301:         }
302:     }
303:     return $error;
304: }
305:
306: # Einträge löschen
```

```
307: sub remove_selected {
308:     my @newdata = ();
309:
310:     foreach (@data) {
311:         my $id = $_->{'id'};    # nicht das globale id
312:
313:         if (!defined param('r' . $id)) {
314:             push @newdata, $_;  # $_ ist die Referenz
315:        }
316:     }
317:     @data = @newdata;           # Eintrag löschen
318: }
319:
320: # neues Element hinzufügen. Wird nur aufgerufen, wenn check_date bereits OK ist
321: sub add_item {
322:     my %newrec = ();
323:
324:     $newrec{'desc'} = param('newdesc');
325:     $newrec{'date'} = param('newdate');
326:     $newrec{'prior'} = param('newprior');
327:     $newrec{'done'} = 0;
328:     $newrec{'id'} = $id++;      # global ID + 1
329:
330:     push @data, { %newrec };
331: }
332:
333: # Datumsangaben müssen das Format XX/XX/XX haben
334: sub check_date {
335:     my $date = shift;
336:
337:     # MM/TT/JJJJ, MM und TT können 0 oder 1 Zeichen haben, JJJJ müssen
338:     # 4 sein. Abschließende Leerzeichen sind okay
339:     if ($date !~ /^\d{1,2}\/\d{1,2}\/\d{4}\s*$/) {
340:         return 1;               # Fehler!
341:     }
342:    return 0;                   # OK
343: }
344:
345: # Datendatei überschreiben
346: sub write_data {
347:     open(DATA, ">$listdata") or
                    die "Listendatei konnte nicht geöffnet werden: $!.";
348:     foreach (@data) {
349:         my %rec = %$_;
350:
351:         foreach ('desc', 'date','prior','done') { # keine ids
352:             print DATA "$_=$rec{$_}\n";
353:         }
354:         print DATA "---\n";
355:     }
356:     close(DATA);
357: }
358:
359: # Ich verwende das Datumsformat MM/TT/JJ.  Um nach Datum zu sortieren,
360: # müssen Sie dieses Format in das Perl-Format (Sekunden-seit 1900)
361: # konvertieren. (Modul Time::Local, Funktion timelocal.)
362: sub date2time {
363:     my $date = shift;
364:     if ($date =~ /^\*\*\*/) {   # Formatierungsfehler
365:         return 0;
366:     } else {
367:         my ($m,$d,$y) = split(/\//,$date);
368:         $m--;           # Monate beginnen in Perls Zeitformat mit 0
369:         return timelocal(0,0,0,$d,$m,$y);
370:     }
371: }
```

Zusammenfassung

Wenn jemand, der über keinerlei Perl-Kenntnisse verfügt, dieses Buch aufschlägt und das Skript in Listing 21.5 betrachtet, stünden die Chancen nicht schlecht, dass er ziemliche Schwierigkeiten mit dem Entziffern des

Quelltextes hätte - auch wenn er schon einiges über Programmiersprachen wissen sollte (Perl ist in dieser Hinsicht etwas sonderbar). Nachdem Sie jetzt aber 21 Tage tief in die Sprache und ihre Idiosynkrasien eingetaucht sind, sollte Ihnen das Lesen dieses Quelltextes leichtfallen - oder zumindest nicht ganz ratlos zurücklassen.

Heute haben wir die Woche und das Buch mit den üblichen längeren Beispielen (einige länger als andere) beendet. Die beiden Skripts, die wir in diesem Kapitel untersucht haben, den Homepage-Generator und die Aufgabenliste, sind beides CGI- Skripts, die Daten von verschiedenen Quellen (oder mehreren Quellen) verarbeiten und HTML-Code als Ausgabe erzeugen. Das zweite Skript, die Aufgabenliste, erzeugte darüber hinaus sein eigenes Formular, mit dem es seine eigenen Daten verarbeiten kann. Das erste Skript ist ein gutes Beispiel dafür, wie man Code von verschiedenen Modulen aus dem CPAN mit eigenem Code zusammenklebt und allgemein gehaltene Anwendungen den eigenen Bedürfnissen anpaßt. Das letztere Skript demonstrierte den Aufbau eines HTML-Formulars und verwendete eine verschachtelte Datenstruktur zur Aufbewahrung seiner Daten. Mit diesen Skripts und der in den letzten 20 Kapiteln geleisteten Arbeit entlasse ich Sie jetzt, was Perl angeht, in die Welt ...

Nun gehen Sie schon!

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl-Funktionen

Dieser Anhang enthält eine kurze alphabetisch geordnete Referenz der vordefinierten Perl-Funktionen. Wenn Sie eine Funktion suchen, die es nicht gibt, verzagen Sie nicht. Ihre Chancen stehen trotzdem gut, dass irgend jemand ein Modul geschrieben hat, das eine Funktion, wie Sie sie benötigen, enthält. Eine Liste der wichtigsten verfügbaren Perl-Module finden Sie in Anhang B.

Beachten Sie, dass nicht alle Funktionen, die in diesem Anhang aufgeführt sind, von allen Perl-Versionen unterstützt werden. Dem Kapitel 18, »Perl und das Betriebssystem«, können Sie entnehmen, welche Funktionen für Windows- oder Macintosh-Rechner nicht zur Verfügung stehen. Als ultimative Referenz sollten Sie die Dokumentation zu Ihrem Perl-Port zu Rate ziehen.

Wo Sie weitere Informationen finden

Weitere Informationen zu den jeweiligen Funktionen finden Sie in der ***perlfunc***- Manpage. Wie alle Perl-Dokumentationen können Sie diese Manpage unter Unix mit dem Befehl `man`, unter Unix und Windows mit dem Befehl ***perldoc*** (`perldoc perlfunc` zeigt die ganze Liste und `perldoc -f` die einzelnen Funktionen) und auf dem Mac mit der ***Shuck***-Anwendung aufrufen.

Die gesamte Perl-Dokumentation ist auch über das Web unter `http://www.perl.com/ CPAN-local/doc/manual/html/pod/` abrufbar.

Neben den Perl-spezifischen Manpages verweist dieser Anhang auch häufig auf andere Unix-Manpages (vornehmlich deshalb, weil Perl oft von Unix-Features Gebrauch macht). Die Unix-Manpages sind in numerierten Kapiteln zusammengefaßt. Wenn Sie auf ein Element stoßen, hinter dem eine Zahl in Klammern steht, weist diese Zahl darauf, dass dieses Element in dem entsprechenden Kapitel der Manpages beschrieben ist. So finden Sie zum Beispiel `fseek(2)` in Kapitel 2 des Unix- Handbuches. Um die Funktion `fseek` in Kapitel 2 nachzuschlagen, würden Sie also folgenden Unix-Befehl eingeben:

```
man 2 fseek
```

Auf manchen Systemen lautet der Befehl ein wenig anders:

```
man -s 2 fseek
```

Der Befehl `man man` zeigt Ihnen Hilfe zu dem Befehl `man` an.

Die Perl-Funktionen in alphabetischer Reihenfolge

Hier finden Sie die Perl-Funktionen in alphabetischer Reihenfolge.

abs

```
abs WERT
```

Liefert den absoluten Wert von `WERT` zurück. `WERT` kann eine Zahl (wie `-7`) oder ein Ausdruck sein.

accept

```
accept NEUERSOCKET, GENERISCHERSOCKET
```

`accept` wird verwendet, um eine einkommende Socket-Verbindung anzunehmen. Wenn eine Verbindung hergestellt wurde, wird die gepackte Adresse zurückgeliefert. Andernfalls lautet der Rückgabewert ***falsch***. Entspricht dem Systemaufruf `accept(2)`.

alarm

```
alarm SEKUNDEN
```

Die `alarm`-Funktion sendet ein `SIGALRM` an das Programm, nachdem die in `SEKUNDEN` spezifizierten Sekunden verstrichen sind. Es kann nur ein Alarm aktiv sein. Wenn Sie diese Funktion aufrufen, während noch ein anderer Alarm läuft, wird der laufende Alarm durch den aktuelleren Aufruf ersetzt. Wenn Sie die Funktion mit dem Argument `0` aufrufen, wird der aktuelle Alarm unterbrochen, ohne dass ein neuer gestartet wird. Wird für `SEKUNDEN` keine Zahl angegeben, wird der Wert in `$_` verwendet.

atan2

```
atan2 Y, X
```

Liefert den Arkustangens von `X/Y` in dem Bereich von `-p` bis `p` zurück. Weitere Funktionen für die Tangensoperationen gibt es in den Modulen `POSIX` und `Math::Trig`.

bind

```
bind SOCKET, NAME
```

Die Funktion `bind` bindet eine Netzwerkadresse an einen Socket. `NAME` sollte dabei die gepackte Adresse für verwendete Art von Socket enthalten. Ist die Funktion `bind` erfolgreich, wird ***wahr*** zurückgeliefert, andernfalls ***falsch***. Entspricht dem Systemaufruf `bind`.

binmode

```
binmode DATEIHANDLE
```

`binmode` übernimmt als Argument ein Datei-Handle und gibt an, dass die Daten im Binärformat (und nicht im ASCII-Format) in das Datei-Handle geschrieben (beziehungsweise daraus gelesen) werden sollen. Unter Unix hat diese Funktion keine Auswirkung, aber für MS-DOS und andere archaische Plattformen ist sie wichtig. Die Funktion sollte nach dem Öffnen einer Datei und vor irgendwelchen Eingabe-/ Ausgabeoperationen auf diese Datei aufgerufen werden.

bless

```
bless REFERENZ, KLASSENNAME
```

`bless` wird in der objektorientierten Perl-Programmierung verwendet, um das, was mit `REFERENZ` referenziert wird, dem Paket `KLASSENNAME` zuzuweisen. Wird `KLASSENNAME` fortgelassen, wird `REFERENZ` dem aktuellen Paket zugewiesen. `bless` liefert die Referenz zurück, die mit `bless` zugewiesen wird. Detailliertere Informationen finden Sie in der ***perlobj***-Manpage.

caller

```
caller AUSDRUCK
caller
```

`caller` liefert den Kontext des aktuellen Subroutinenaufrufs zurück. In einem skalaren Kontext liefert `caller` den

Namen des Pakets, aus dem die Subroutine aufgerufen wurde, in einem Listenkontext liefert sie den Paketnamen, den Dateinamen des Programms und die Zeilennummer des Aufrufs. Wenn ein AUSDRUCK mit angegeben wird, liefert caller zusätzliche Informationen zur Verfolgung des Stacks. AUSDRUCK gibt an, um wie viele Aufruf-Frames der Stack, ausgehend vom aktuellen Frame, zurückgegangen werden muss. Wird AUSDRUCK angegeben, wird die folgende Liste an Werten zurückgegeben:

```
($package, $file, $line, $subname, $hasargs, $wantarray) = caller(bel_funk);
```

chdir

```
chdir AUSDRUCK
```

chdir übernimmt einen Ausdruck als Argument und versucht, das aktuelle Verzeichnis auf das im Ausdruck angegebene Verzeichnis zu setzen. Wird kein Argument mitgeliefert, versucht die Funktion, in das Home-Verzeichnis des aktuellen Benutzers zu wechseln.

chmod

```
chmod LISTE
```

chmod dient dazu, die Zugriffsberechtigungen für die im Argument LISTE aufgeführten Dateien zu ändern. Das erste Element in LISTE muss - in Oktalnotation - der numerische Modus für die Dateien sein. Er sollte auch das SUID-Bit mit einschließen. Sehen Sie hier ein Beispiel für die Anwendung von chmod:

```
chmod 0755, @dateien;
```

Beachten Sie, dass das erste Element der Liste nicht in Anführungszeichen steht. Es ist einfach nur eine Zahl. Weitere Informationen zu den Datei-Zugriffsrechten finden Sie in der ***chmod***-Manpage.

chomp

```
chomp VARIABLE
chomp LISTE
chomp
```

chomp ist die sichere Version von chop, die als nächstes beschrieben wird. Die Funktion entfernt alle Zeilenenden, die mit $/ übereinstimmen (die Variable, die das Eingabedatensatz-Trennzeichen, normalerweise ein Neue-Zeile-Zeichen, enthält). Wenn diese Funktion im Absatzmodus aufgerufen wird, werden alle angehängten Neue-Zeile-Zeichen aus dem String entfernt.

chop

```
chop VARIABLE
chop LISTE
chop
```

chop wird verwendet, um das letzte Zeichen in einem String zu entfernen. Ihre ursprüngliche Aufgabe bestand darin, das Zeilenvorschubzeichen am Ende einer Zeile möglichst problemlos zu entfernen (wenn Sie eine Datei Zeile für Zeile bearbeiten, ist es oft sinnvoll, das Zeilenvorschubzeichen aus den Zeilen zu entfernen, bevor Sie mit der Arbeit beginnen). Das Problem mit chop ist, dass die Funktion jedes letzte Zeichen vom String entfernt. Ist das letzte Zeichen ein Zeilenvorschubzeichen - wunderbar. Aber wenn nicht, verlieren Sie das letzte Zeichen, das durchaus eine Bedeutung haben kann.

```
while (<EINGABEDATEI>) {
  #    Beachten Sie, dass in diesem Beispiel $_ als Argument für die
  #    chop-Funktion verwendet wird.
  chop;
  push (@names);
}
```

Wird chop ohne Argument verwendet, wird das letzte Zeichen von $_ entfernt. Wird eine Liste übergeben, wird aus allen Elementen der Liste das jeweils letzte Zeichen entfernt.

chown

```
chown LISTE
```

chown wird verwendet, um die Besitzrechte der Dateien, die in LISTE stehen, für Benutzer und Gruppe zu setzen. Zurückgeliefert wird die Anzahl der Dateien, für die die Besitzrechte erfolgreich geändert wurden. Die ersten beiden Elemente von LISTE müssen die numerische UID und GID des Benutzers und der Gruppe sein, die Besitzer der Dateien werden. Normalerweise kann nur der ***root***-Benutzer die Besitzrechte von Dateien in einem System ändern.

chr

```
chr ZAHL
```

Die Funktion chr liefert das Zeichen aus der ASCII-Tabelle zurück, das dem Zahlencode entspricht, der der Funktion übergeben wurde. So liefert zum Beispiel chr(80) ein R zurück. Mit der Funktion pack können mehrere Zeichen gleichzeitig konvertiert werden.

chroot

```
chroot VERZEICHNISNAME
```

Die Funktion chroot ist identisch mit dem gleichnamigen Systemaufruf (Details sind der Manpage ***chroot(2)*** zu entnehmen). Im wesentlichen teilt chroot dem gerade ausführenden Programm sowie allen exec-Aufrufen und Subprozessen mit, dass das in VERZEICHNISNAME angegebene Verzeichnis als neues Root-Verzeichnis zu verwenden ist. Pfade, die mit / anfangen, beginnen danach mit dem Verzeichnis VERZEICHNISNAME und nicht mehr mit dem eigentlichen Root-Verzeichnis des Dateisystems. Nur der ***root***-Benutzer kann die Funktion chroot verwenden.

close

```
close DATEIHANDLE
```

Die Funktion close wird verwendet, um ein zuvor geöffnetes Datei-Handle zu schließen (unabhängig davon, ob es sich um eine Datei oder eine Pipe handelt). Sie führt die notwendigen Aufräumarbeiten auf der Systemebene durch und liefert ***wahr*** zurück, wenn alle Operationen erfolgreich waren. Beachten Sie, dass alle Datei- Handle automatisch geschlossen werden, wenn ein Perl-Programm beendet wird, so dass Sie oft davonkommen, ohne explizit alle Datei-Handle zu schließen, die Sie geöffnet haben.

closedir

```
closedir VERZEICHNISHANDLE
```

closedir schließt ein Verzeichnis, das mit der Funktion opendir geöffnet wurde.

connect

```
connect SOCKET, NAME
```

connect versucht, eine Verbindung zu einem entfernten Socket herzustellen. NAME sollte die gepackte Adresse entsprechend dem Typ des Sockets enthalten. Die Funktion liefert ***wahr*** zurück, wenn sie erfolgreich war, im anderen Fall ***falsch***. Diese Funktion ist identisch mit dem Systemaufruf connect.

cos

```
cos AUSDRUCK
```

Liefert den Kosinus von AUSDRUCK zurück. Für die inverse Kosinusoperation sollten Sie die Funktion `POSIX::acos()` oder das Modul `Math::Trig` verwenden.

crypt

```
crypt NORMALERTEXT, WERT
```

Die `crypt`-Funktion wird dazu verwendet, um Strings auf die gleiche Weise zu verschlüsseln, wie Paßwörter in einer Unix-Paßwortdatei gespeichert werden. Die Funktion übernimmt zwei Argumente: den String, der verschlüsselt werden soll, und einen Wert, mit dem der Verschlüsselungsalgorithmus initialisiert wird. Die `crypt`-Funktion ist eine Einbahnstraße: Es ist keine Methode bekannt, mit der man Text, der mit `crypt` verschlüsselt wurde, wieder entschlüsseln kann (Unix testet Paßwörter, indem `crypt` auf die vom Benutzer eingegebenen Paßwörter angewendet und das Ergebnis dann mit den verschlüsselten Paßwörtern in `/etc/passwd` verglichen wird).

dbmclose

```
dbmclose HASH
```

`dbmclose` löst die Bindung zwischen einem `HASH` und der DBM-Datei, mir der er verbunden ist. Diese Funktion wurde inzwischen durch die Funktion `untie` ersetzt.

dbmopen

```
dbmopen HASH, DBNAME, MODUS
```

`dbmopen` bindet eine `dbm`, `ndbm`, `sdbm`, `gdmb` oder Berkeley-DB-Datei an einen Hash. `HASH` ist der Name der Hash-Variablen, an die die Datenbank gebunden wird, und `DBNAME` der Name der Datenbankdatei ohne die Extension. Wenn `DBNAME` nicht angegeben wird, wird eine neue Datei erzeugt, deren Zugriffsberechtigungen durch `MODUS` spezifiziert werden.

Diese Funktion wurde inzwischen durch `tie` abgelöst.

defined

```
defined AUSDRUCK
```

`defined` wird verwendet, um Ausdrücke zu identifizieren, die den undefinierten Wert (im Gegensatz zu 0, `Neue Zeile` oder einen anderen leeren Rückgabewert) zurückliefern. Mit Hilfe der Funktion kann man festzustellen, ob eine Subroutine existiert oder eine Skalarvariable definiert wurde. Wenn kein AUSDRUCK übergeben wird, prüft `defined`, ob `$_` undefiniert ist.

delete

```
delete AUSDRUCK
```

Die `delete`-Funktion wird verwendet, um Elemente aus einem Hash zu entfernen. Dabei gehen Sie so vor, dass Sie der Funktion einfach den Namen des Hash und den Schlüssel, der entfernt werden soll, übergeben. Sehen Sie dazu folgendes Beispiel:

```
delete $hash{$key};
```

Da Sie nur ein einziges Element im Hash löschen wollen, referenzieren Sie die Hash- Variable im skalaren Kontext (mit `$`).

die

```
die LISTE
```

`die` übernimmt eine Liste als Argument. Wenn `die` aufgerufen wird, wird das Programm beendet, der Wert von `$!` zurückgeliefert und die als Argument an `die` übergebene Liste auf die Standardfehlerausgabe geschickt. Wenn die Liste nicht mit einem Neue-Zeile-Zeichen endet, werden der Name des Programms und die Zeilennummer der Zeile, bei der die Ausführung unterbrochen wurde, zusammen mit einem Neue-Zeile-Zeichen an die Ausgabe der Funktion angehängt.

Hier ein Beispiel:

```
open (FILE, $datei) or die "$datei kann nicht geöffnet werden";
```

Wenn `$datei` nicht geöffnet werden kann, erzeugt diese Zeile folgende Ausgabe:

```
/tmp/datei kann nicht geöffnet werden at test_program line 13.
```

do

```
do BLOCK
do SUBROUTINE(LISTE)
do AUSDRUCK
```

Wenn diese Funktion mit einem Codeblock in `BLOCK` aufgerufen wird, führt `do` die Anweisungen in dem Block aus und liefert den Wert der letzten Anweisung in dem Block zurück. Wenn `do` zusammen mit einem Schleifenausdruck verwendet wird, wird `BLOCK` ausgeführt, bevor die Schleifenbedingung das erste Mal getestet wird.

`do SUBROUTINE` ist eine veraltete Methode, um eine Subroutine aufzurufen, und `do AUSDRUCK` stellt eine Möglichkeit dar, Code in einer anderen Datei auszuführen. `AUSDRUCK` wird in diesem Falle als der Dateiname einer Perl-Datei interpretiert, deren Inhalt ausgeführt wird. Auch wenn sich `do` auf diese Weise einsetzen läßt, sollten Sie trotzdem auf `require` und `use` ausweichen, die robuster sind.

dump

```
dump LABEL
```

Mit `dump` bewirken Sie, dass Perl einen Speicherauszug (dump) des Programms ausgibt. Anschließend können Sie mit dem ***undump***-Programm eine binäre Datei erzeugen, die die Ausführung mit einem `goto LABEL`-Befehl beginnt.

each

```
each HASH
```

Die `each`-Funktion wird verwendet, um Werte für die Verarbeitung in einer Schleife aus einem Hash herauszuholen. Sie verhält sich je nach Kontext (skalar oder Liste) unterschiedlich.

Im skalaren Kontext liefert die `each`-Funktion den Schlüssel für das nächste Element im Hash zurück. Sie läßt sich daher folgendermaßen anwenden:

```
while ($key = each %hash) {
    $hash{$key}++;
}
```

Im anderen Falle, das heißt in einem Listenkontext, liefert die `each`-Funktion eine zweielementige Liste, die den Schlüssel und den Wert für das nächste Element im Hash enthält. Das sieht dann wie folgt aus:

```
while (($key, $value) = each %hash) {
    print "$key = $value\n";
}
```

eof

```
eof DATEIHANDLE
eof ()
eof
```

Die eof-Funktion liefert 1 zurück, wenn beim nächsten Lesen aus DATEIHANDLE die Dateiendemarke zurückgeliefert wird oder DATEIHANDLE nicht geöffnet ist. Ohne Argument wertet eof die zuletzt gelesene Datei aus. Erfolgt der Aufruf mit leeren Klammern als Argument, erkennt eof das Ende einer Pseudodatei, die aus all den Dateien besteht, die auf der Befehlszeile angegeben wurden. In der ***perlfunc***- Manpage wird klugerweise darauf hingewiesen, dass eof selten verwendet wird, da Perl automatisch den undefinierten Wert zurückliefert, wenn das Ende einer Datei erreicht ist - so dass das Dateiende auch bequem ohne diese Funktion entdeckt werden kann.

eval

```
eval AUSDRUCK
```

eval wird verwendet, um einen Ausdruck oder einen Codeblock auszuführen, als ob es sich um ein eigenständiges Perl-Programm handelte. Es wird im Kontext des gerade laufenden Perl-Programms ausgeführt, so dass alle Variablen und anderen persis-tenten Werte des übergeordneten Programms weiter definiert sind, wenn die Ausführung des Ausdrucks mit eval beendet ist.

Der von eval zurückgelieferte Wert ist der Wert des zuletzt ausgewerteten Ausdrucks. Um explizit einen bestimmten Wert zurückzuliefern, können Sie innerhalb von eval eine return-Anweisung verwenden. Wenn innerhalb der eval-Anweisung ein Syntax- oder ein Laufzeitfehler auftritt oder eine die-Anweisung ausgeführt wird, liefert die eval-Anweisung einen undefinierten Wert zurück, und die Variablen $@ enthält die Fehlermeldung.

Da fatale Fehler, die innerhalb von eval-Anweisungen auftreten, die Ausführung des übergeordneten Programms nicht anhalten, kann die Funktion genutzt werden, um Fehler abzufangen oder schwer kalkulierbaren Code auszutesten.

exec

```
exec LISTE
```

Die exec-Funktion führt einen Systembefehl aus und hat keinen Rückgabewert, es sei denn der Befehl existiert nicht. Wenn LISTE aus mehr als einem Element besteht, verwendet exec den Systemaufruf execvp(3) mit den Argumenten in LISTE. Wenn das Argument aus einem einzigen skalaren Wert besteht, wird das Argument auf Shell-Metazeichen geprüft. Wenn Shell-Metazeichen gefunden werden, wird das Argument durch /bin/sh -c ausgeführt; andernfalls wird das Argument in Wörter zerlegt und an execvp übergeben.

exists

```
exists AUSDRUCK
```

Die exists-Funktion wird verwendet, um zu prüfen, ob ein bestimmter Schlüssel in einem Hash definiert ist. exists prüft nicht, ob es zu dem Schlüssel auch einen Wert gibt; es wird ausschließlich zum Prüfen auf Schlüssel verwendet. Sehen Sie zur Veranschaulichung ein Beispiel:

```
if (exists $hash{$key}) { print "Ja."; }
else { print "Nein.\n"; }
```

exit

```
exit AUSDRUCK
```

Die exit-Funktion wertet AUSDRUCK aus und bricht das Programm sofort ab. Normalerweise ist die die sauberere Lösung, um die Ausführung eines Programms abzubrechen, da die zurückgelieferten Fehlerinformationen abgefangen werden können.

exp

```
exp AUSDRUCK
```

Liefert den Wert e hoch AUSDRUCK zurück. Wenn exp ohne AUSDRUCK aufgerufen wird, wird standardmäßig exp($). Für normale Exponenten verwenden Sie den Operator **.

fcntl

```
fcntl DATEIHANDLE, FUNKTION, SKALAR
```

Wird verwendet, um den Systemaufruf fcntl(2) zu emulieren. Mit use fcntl; erhalten Sie die Funktionsdefinitionen, die Sie benötigen, um diese Funktion zu nutzen. In der Manpage finden Sie weitere Informationen zu der Funktion. fcntl liefert einen fatalen Fehler zurück, wenn sie auf der Plattform, auf der sie ausgeführt wird, nicht implementiert ist.

fileno

```
fileno DATEIHANDLE
```

fileno liefert einen Dateideskriptor für einen gegebenen Datei-Handle zurück. Ein Dateideskriptor ist ein kleiner Integerwert, der die Datei identifiziert. Er kann für die Konstruktion von Bitmaps für die Verwendung mit select genutzt werden. Wenn DATEIHANDLE nicht geöffnet ist, wird undefined zurückgeliefert.

flock

```
flock DATEIHANDLE, OPERATION
```

Diese Funktion ruft den Systembefehl flock(2) für DATEIHANDLE auf. Weitere Informationen zu den verfügbaren Operationen finden Sie in der Manpage zu flock(2). Auf Systemen, die flock(2) oder Mechanismen zum Sperren von Dateien nicht unterstützen, erhalten Sie einen fatalen Fehler.

fork

```
fork
```

fork wird verwendet, um einen Systemaufruf in einen getrennten Prozeß zu verzweigen. fork liefert die Kind-PID an den übergeordneten Prozeß zurück. Diese Funktion ist nur auf Unix-ähnlichen Plattformen implementiert. Der gesamte Code in dem Block wird in einem neuen Prozeß ausgeführt.

format

```
format
```

Die format-Funktion wurde entworfen, um COBOL-Programmierern das Erlernen von Perl zu erleichtern. Genau genommen bietet es Ihnen eine Methode, um Schablonen für die formatierte Ausgabe zu erzeugen. In der ***perlform***-Manpage können Sie detailliert nachlesen, wie Sie mit format eine Ausgabe erzeugen.

formline

```
formline BILD, LISTE
```

Die Funktion formline wird intern von den Formaten verwendet. Sie wird verwendet, um das Argument LISTE analog zu BILD zu formatieren. Weitere Informationen entnehmen Sie bitte der Manpage ***perlform***.

getc

```
getc DATEIHANDLE
```

getc liefert das nächste Zeichen aus DATEIHANDLE zurück. Wird das Argument DATEIHANDLE weggelassen, liefert getc das nächste Zeichen aus STDIN. getc erlaubt keine ungepufferte Eingabe (mit anderen Worten, wenn STDIN die Konsole ist, dann erhält getc das Zeichen erst, wenn der Puffer mit einem Neue-Zeile-Zeichen geleert wurde).

getlogin

```
getlogin
```

Liefert die aktuelle Login-Information aus /etc/utmp zurück, falls vorhanden. Falls der Wert Null lautet, sollten Sie getpwuid() verwenden.

getpeername

```
getpeername SOCKET
```

getpeername liefert die gepackte sockaddr-Adresse des anderen Endes der SOCKET- Verbindung zurück.

getpgrp

```
getpgrp PID
```

getpgrp liefert die Prozeßgruppe für den spezifizierten Prozeß zurück. Wird als PID 0 angegeben, wird die Prozeßgruppe des aktuellen Prozesses zurückgeliefert.

getppid

```
getppid
```

getppid liefert die Prozess-ID für den Elternprozeß des aktuellen Prozesses zurück.

getpriority

```
getpriority WELCHE, WER
```

Unter der Voraussetzung, dass getpriority auf Ihrem System implementiert ist, liefert diese Funktion die Priorität für einen Prozeß, eine Prozeßgruppe oder einen Benutzer zurück.

getsockname

```
getsockname SOCKET
```

getsockname liefert die gepackte sockaddr-Adresse dieses Endes der SOCKET- Verbindung zurück.

getsockopt

```
getsockopt SOCKET, LEVEL, OPTNAME
```

getsockopt liefert die angeforderte Option zurück oder im Falle eines Fehlers undefined.

glob

```
glob AUSDRUCK
```

Die glob-Funktion liefert den Wert von AUSDRUCK mit Dateinamenerweiterungen zurück, wie sie auch unter einer Shell vorkommen könnten. Wird AUSDRUCK weggelassen, wird als Argument $_ verwendet.

gmtime

```
gmtime AUSDRUCK
```

`gmtime` konvertiert eine Zeitangabe in dem von der `time`-Funktion zurückgelieferten Zeitformat (Sekunden seit 1. Jan. 1970, 00:00) in die Greenwich-Standardzeit (auch bekannt als Greenwich Mean Time). Diese Zeitangabe wird als eine Liste mit neun Elementen zurückgeliefert. Die Inhalte der Elemente können Sie diesem Beispiel entnehmen:

```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime(time);
```

Beachten Sie, dass alle diese Elemente im numerischen Format zurückgegeben werden und die aufeinanderfolgenden Zahlen (wie Monate und Tage der Woche) statt mit einer 1 mit einer 0 beginnen. Das bedeutet, dass Monate von 0 bis 11 zählen. Das zurückgelieferte Jahr ist die Zahl des Jahres seit 1900 und nicht einfach nur die letzten zwei Stellen. Damit wird das so gefürchtete Jahr-2000-Problem vermieden. Wenn Sie `gmtime` in einem skalaren Kontext verwenden, liefert die Funktion die Zeit im `ctime(3)`-Format zurück:

```
Sat Jun  6 01:56:44 1998
```

goto

```
goto LABEL
goto AUSDRUCK
goto &NAME
```

Die `goto`-Funktion sucht die mit `LABEL` benannte Anweisung und fährt ab dort mit der Ausführung fort. Sie kann die Ausführung nicht mit Anweisungen fortsetzen, die sich in Blocks befinden und initialisiert werden müssen (zum Beispiel Subroutinen oder `foreach`-Schleifen). Es gibt noch zwei weitere Anwendungsbereiche für `goto`, die jedoch ziemlich veraltet sind. `goto AUSDRUCK` wird verwendet, um zu einer Marke (Label) zu springen, die mit `AUSDRUCK` angegeben wird, wobei der Gültigkeitsbereich dynamisch ist. `goto &NAME` ersetzt die aktuell laufende Subroutine durch einen Aufruf der benannten Subroutine, als ob diese zuerst aufgerufen worden wäre.

grep

```
grep AUSDRUCK, LISTE
grep BLOCK LISTE
```

Die `grep`-Funktion dient dazu, Listen zu durchsuchen und alle Elemente in der Liste zurückzuliefern, die einem bestimmten Muster entsprechen. `grep` übernimmt zwei Argumente: einen Ausdruck und eine Liste. Es liefert eine neue Liste zurück, die all die Elemente enthält, für den der Ausdruck ***wahr*** ist. Betrachten wir ein Beispiel:

```
@neuesarray = grep /rot/, @altesarray;
```

`@neuesarray` enthält in diesem Beispiel eine Liste der Elemente aus `@altesarray`, die den String `rot` enthalten. Wenn Sie `grep` in einem skalaren Kontext aufrufen, wird statt der Liste der Elemente die Anzahl der übereinstimmenden Elemente zurückgeliefert.

hex

```
hex AUSDRUCK
```

`hex` liest `AUSDRUCK` als einen hexadezimalen String und liefert den Dezimalwert zurück. Wird `AUSDRUCK` fortgelassen, liest die Funktion `$_`.

import

```
import KLASSENNAME LISTE
import KLASSENNAME
```

`import` ist keine vordefinierte Funktion. Sie wird vielmehr von Modulen implementiert, die Namen in andere Module exportieren wollen. Die `import`-Funktion wird von der `use`-Funktion aufgerufen, wenn ein Modul in ein Perl-Programm geladen wird.

index

```
index STR, SUBSTR, POSITION
index STR, SUBSTR
```

`index` wird verwendet, um einen Teilstring in einem größeren String zu lokalisieren. Die Funktion übernimmt drei Argumente, von denen eines optional ist. Bei den Argumenten handelt es sich um den String, der durchsucht werden soll, den gesuchten Teilstring und die Position, ab der die Suche beginnen soll (optional). `index` liefert die Position im String zurück, an der der Teilstring das erste Mal beginnt. Um zum Beispiel den String `bar` in dem größeren String `verwundbar` zu finden, könnten Sie den folgenden Code verwenden: `index ('verwundbar', 'bar');`. Um das zweite Vorkommen von `sein` in dem String `Sein oder Nichtsein` zu suchen, könnten Sie optional das dritte Argument verwenden, um die Suche bei Position 4 zu starten: `index ('Sein oder Nichtsein ', 'sein', 4);`.

int

```
int AUSDRUCK
```

`int` liefert den Integeranteil eines Strings zurück. Wenn beispielsweise ein String mit einem Integer beginnt, wie `60 kmh`, wird `int` diesen Integer, in unserem Beispiel `60`, zurückliefern. Strings, die nicht mit einem Integer beginnen, liefern 0 zurück. Wenn Sie die Perl-Warnungen aktiviert haben, erhalten Sie für jeden String, der nicht mit einer Zahl beginnt, eine Warnung.

ioctl

```
ioctl DATEIHANDLE, FUNKTION, SKALAR
```

`ioctl` wird dazu verwendet, den Systemaufruf `ioctl(2)` zu implementieren. Sie werden wahrscheinlich

```
require "ioctl.ph";
```

aufrufen müssen, bevor Sie die Funktionsdefinitionen für `ioctl` importieren können. Wenn die Funktion nicht existiert, werden Sie auf der Basis der Systemdatei `ioctl.h` Ihre eigenen Funktionsdefinitionen erzeugen müssen.

join

```
join AUSDRUCK, LISTE
```

Die `join`-Funktion ist das Gegenstück zu `split`. Sie wird benötigt, um die Elemente einer Liste in einem einzigen String zusammenzufassen. Sie übernimmt zwei Argumente: einen Ausdruck und eine Liste. Der Inhalt des Ausdrucks wird als Begrenzer zwischen den Elementen im zurückgelieferten String verwendet.

keys

```
keys HASH
```

Die `keys`-Funktion liefert ein Array zurück, das alle Schlüssel des in `HASH` angegebenen Hash enthält. Diese Funktion wird häufig verwendet, um die Schlüssel in einem Hash zu sortieren, bevor man eine Schleife darüber laufen läßt. Hier ein typisches Beispiel:

```
foreach $key (sort (keys %hash)) {
    print $key, " = ", $value, "\n";
}
```

kill

```
kill LISTE
```

`kill` dient eigentlich dazu, ein Signal an eine Liste von Prozessen zu senden, anstatt diese einfach abzuwürgen. Das erste Argument in `LISTE` muss das zu sendende Signal sein, der Rest sollten die Prozesse sein, die das Signal erhalten sollen. Um Prozesse zu beenden, würden Sie folgenden Code verwenden:

```
kill 1, 100, 102, 110;
```

Um die gleichen Prozesse mit größerer Entschlossenheit zu beenden, würden Sie schreiben:

```
kill 9, 100, 102, 110;
```

Sie können anstatt einer Signalzahl auch den Signalnamen angeben, der dann allerdings in Anführungszeichen stehen muss. Weitere Informationen zu den Signalen finden Sie in der Manpage `signal(5)`.

last

```
last LABEL
last
```

Der `last`-Befehl veranlaßt das direkte Verlassen der in `LABEL` angegebenen Schleife. Wird kein Wert für `LABEL` angegeben, wird die innerste Schleife verlassen.

lc

```
lc AUSDRUCK
```

Die `lc`-Funktion konvertiert alle alphabetischen Zeichen in einem String in Kleinbuchstaben. `lc 'ABC';` liefert demnach `abc` zurück. Wird kein `AUSDRUCK` angegeben, wird die `lc`-Funktion auf `$_` angewendet.

lcfirst

```
lcfirst AUSDRUCK
```

Liefert den Wert in `AUSDRUCK` zurück, nachdem das erste Zeichen in einen Kleinbuchstaben verwandelt wurde. Wird kein `AUSDRUCK` angegeben, wird die Funktion auf `$_` angewendet.

length

```
length AUSDRUCK
```

`length` übernimmt einen String als Argument und liefert einen Integer zurück, der die Länge des Strings in Bytes angibt. So liefert zum Beispiel `length("Hund")` den Wert `4` zurück. Wenn `AUSDRUCK` nicht angegeben wird, wird die Funktion `$_` angewendet.

link

```
link ALTEDATEI, NEUEDATEI
```

Erzeugt einen harten (anstelle eines symbolischen) Links von `ALTEDATEI` zu `NEUEDATEI`. Zur Erzeugung eines symbolischen Links müssen Sie sich der Funktion `symlink` bedienen.

listen

```
listen SOCKET, WARTESCHLANGENGROESSE
```

Die listen-Funktion in Perl erfüllt die gleiche Funktion wie der Systemaufruf listen. Sie liefert im Erfolgsfall ***wahr*** zurück und ansonsten ***falsch***.

local

```
local AUSDRUCK
```

local gibt an, dass die aufgelisteten Variablen lokal zu dem gerade ausgeführten Block, der Schleife, Subroutine, eval {} oder do sind. Wenn man local mehr als eine Variable übergeben möchte, sollte man diese in Klammern setzen. Um den Gültigkeitsbereich einer Variablen zu begrenzen, ist es meist besser, my zu verwenden.

localtime

```
localtime AUSDRUCK
```

Die Funktion localtime ist identisch mit der Funktion gmtime, nur dass sie die Zeit in die lokale Zeitzone und nicht in die Greenwich-Zeit konvertiert.

log

```
log AUSDRUCK
```

Liefert den Logarithmus (Basis e) von AUSDRUCK oder von $_, wenn AUSDRUCK nicht angegeben wird.

lstat

```
lstat DATEIHANDLE
lstat AUSDRUCK
lstat
```

lstat entspricht der Funktion stat, bis auf die Tatsache, dass sie statistische Informationen für einen symbolischen Link und nicht für die Datei, auf die der Link verweist, zurückliefert. Wenn AUSDRUCK weggelassen wird, wird lstat auf den Wert in $_ angewendet.

map

```
map BLOCK LISTE
map AUSDRUCK, LISTE
```

map stellt eine Alternative zu foreach dar, wenn es darum geht, eine Operation auf jedes Element in einer Liste auszuführen. Die Funktion kann in zwei Formen aufgerufen werden. Der folgende Code demonstriert, wie Sie alle Operationen in einem Codeblock auf eine Liste anwenden können:

```
@rueckwaerts_worte = map {
   lc;
   reverse;
} @worte;
```

Das obige Beispiel dreht alle Elemente in dem Array @worte um und konvertiert sie in Kleinbuchstaben. Die Ergebnisse von map werden in einem Listenkontext zurückgegeben, weshalb ich sie einem Array zuweise. Beachten Sie, dass jedes Element zur Bearbeitung der Variablen $_ zugewiesen wird. Das ist der Grund, warum ich die Funktionen im Codeblock ohne Argumente verwenden kann. Um eine einzige Operation auf jedes Element einer Liste durchzuführen, wird map folgendermaßen aufgerufen:

```
@neueliste = map(uc, @alteliste);
```

Auffällig ist, dass bei Anwendung einer einzelnen Operation die Operation mit Komma von der zu verarbeitenden Liste getrennt wird.

mkdir

```
mkdir DATEINAME, MODUS
```

`mkdir` wird verwendet, um ein neues Verzeichnis anzulegen, dessen Name in `DATEINAME` angegeben wird. Sie sollten die Zugriffsrechte für das Verzeichnis mit `MODUS` setzen. `MODUS` sollte im üblichen Oktalformat angegeben werden (eine reine Zahl, nicht von Anführungszeichen umschlossen) und das SUID-Bit enthalten.

msgctl

```
msgctl ID, CMD, ARG
```

`msgctl` ruft den Systemaufruf `msgctl(2)` auf. Diese Funktion ist nur auf Maschinen verfügbar, die das System V IPC unterstützen.

msgget

```
msgget SCHLÜSSEL, FLAGS
```

Ruft die System V IPC-Funktion `msgget` auf und liefert die ID der Nachrichtenwarteschlange oder im Falle eines Fehlers `undefined` zurück.

msgrcv

```
msgrcv ID, VAR, GROESSE, TYP, FLAGS
```

Ruft die System V IPC-Funktion `msgrcv` auf, um eine Nachricht von der Nachrichtenwarteschlange ID zu empfangen, die in der Variablen `VAR` gespeichert wird, mit einer maximalen Größe von `GROESSE`. Liefert im Erfolgsfall ***wahr*** zurück oder im Falle eines Fehlers ***falsch***.

msgsnd

```
msgsnd ID, MLD, FLAGS
```

Ruft die System V IPC-Funktion `msgsnd` auf, um `MLD` an die Nachrichtenwarteschlange, die durch `ID` spezifiziert ist, zu senden. Liefert im Erfolgsfall ***wahr*** zurück oder im Falle eines Fehlers ***falsch***.

my

```
my AUSDRUCK
```

`my` wird verwendet, um die aufgelisteten Variablen einem Gültigkeitsbereich zuzuordnen, der lokal zu dem aktuellen Block, zu `eval{}`, zur Subroutine oder zur importierten Datei ist. Wird mehr als eine Variable angegeben, müssen die Variablen in Klammern stehen.

next

```
next LABEL
next
```

Wenn das Programm in einer Schleife auf den Befehl `next` stößt, springt die Programmausführung direkt zum nächsten Schleifendurchlauf.

no

```
no MODULE LISTE
```

Das no-Modul ist das Gegenstück zu dem use-Operator. Ausführlichere Informationen dazu finden Sie in der ***perlobj***-Manpage.

oct

```
oct AUSDRUCK
```

oct liest AUSDRUCK als einen oktalen String ein und liefert den zugehörigen Dezimalwert zurück, es sei denn der String beginnt mit 0x, was bedeutet, dass der String als Hexadezimalwert interpretiert wird. Wird AUSDRUCK weggelassen, liest die Funktion $_ ein.

open

```
open (DATEIHANDLE, AUSDRUCK)
```

Die open-Funktion öffnet die Datei, die in AUSDRUCK spezifiziert wird, und weist sie DATEIHANDLE zu. Wenn AUSDRUCK weggelassen wird, wird angenommen, dass eine Variable mit dem gleichen Namen wie DATEIHANDLE den Namen der zu öffnenden Datei enthält.

Indem Sie < vor den Dateinamen setzen, können Sie die Datei für die Eingabe öffnen, und indem Sie ein > davorsetzen, öffnen Sie die Datei für die Ausgabe. Um Daten an die Ausgabedatei anzuhängen, anstatt sie zu überschreiben, müssen Sie zwei >>- Zeichen vor den Dateinamen setzen. Um einen Datei-Handle mit einer Pipe anstatt der Standardeingabe oder -ausgabe zu öffnen, können Sie das Pipe-Zeichen (|) verwenden. Wenn Sie das |-Zeichen vor den Programmnamen setzen, öffnen Sie eine Pipe zu dem Programm, und wenn Sie das | nach dem Dateinamen setzen, öffnen Sie eine Pipe von dem Programm zu Ihrem Datei-Handle.

Weitere Informationen zu der open-Funktion finden Sie in Kapitel 15, »Dateien und E/A«.

opendir

```
opendir (VERZEICHNISHANDLE, AUSDRUCK)
```

Die Funktion opendir öffnet das in AUSDRUCK spezifizierte Verzeichnis für die Eingabe und weist es VERZEICHNISHANDLE zu. Aus dem Verzeichnis-Handle kann dann die Liste der Einträge in dem Verzeichnis gelesen werden. Beachten Sie, dass sich der Namensbereich für Verzeichnis-Handles nicht mit dem von Datei-Handles überschneidet.

ord

```
ord AUSDRUCK
```

Liefert den numerischen ASCII-Wert des ersten Zeichens von AUSDRUCK zurück. Wenn AUSDRUCK nicht angegeben wird, wird $_ verwendet.

pack

```
pack SCHABLONE, LISTE
```

pack übernimmt eine Liste von Werten, packt sie in eine binäre Struktur und liefert den String zurück, der diese Struktur enthält. SCHABLONE ist eine Liste von Zeichen, die die Reihenfolge und den Typ der Werte vorgibt.

Zeichen	Bedeutung
A	Ein ASCII-String, der mit Leerzeichen aufgefüllt wird.
a	Ein ASCII-String, der mit Nullen aufgefüllt wird.
b	Ein Bit-String (aufsteigende Bitreihenfolge wie vec()).

B	Ein Bit-String (absteigende Bitreihenfolge).
h	Ein Hexadezimal-String (niedriger Nibble zuerst).
H	Ein Hexadezimal-String (hoher Nibble zuerst).
c	Ein vorzeichenbehafteter `char`-Wert (signed).
C	Ein vorzeichenloser `char`-Wert (unsigned).
s	Ein vorzeichenbehafteter `short`-Wert.
S	Ein vorzeichenloser `short`-Wert. (Hier bedeutet »short« genau 16 Bit, was sich von dem »short« eines lokalen C-Compilers unterscheiden kann.)
i	Ein vorzeichenbehafteter Integer-Wert.
I	Ein vorzeichenloser Integer-Wert. (Hier ist »Integer« mindestens 32 Bit lang. Seine genaue Länge hängt davon ab, was ein lokaler C-Compiler unter »int« versteht, und kann sogar noch länger sein als das »long« in der nächsten Tabellenreihe.)
l	Ein vorzeichenbehafteter `long`-Wert. (Dieses »long« ist genau 32 Bit, was sich von dem »long« eines lokalen C-Compilers unterscheiden kann.)
L	Ein vorzeichenloser Integer-Wert.
n	Ein `short`-Wert in »Netzwerk«-Reihenfolge (big-endian)
N	Ein `long`-Wert in »Netzwerk«-Reihenfolge (big-endian)
v	Ein `short`-Wert in »VAX«-Reihenfolge (little-endian)
V	Ein long-Wert in »VAX«-Reihenfolge (little-endian). (Diese »shorts« und »longs« sind genau 16 Bit bzw. 32 Bit groß).
f	Fließkommazahl mit einfacher Genauigkeit im nativen Format.
d	Fließkommazahl mit doppelter Genauigkeit im nativen Format.
p	Ein Zeiger auf einen nullterminierten String.
P	Ein Zeiger auf eine Struktur (String fester Länge).
u	Ein uu-codierter String.
w	Ein BER-komprimierter Integer. Seine Bytes repräsentieren einen vorzeichenlosen Integer zur Basis 128, die signifkanteste Ziffer zuerst, mit so wenig Ziffern wie möglich. Das achte Bit (das High-Bit) ist in jedem Byte außer dem letzten gesetzt.
x	Ein Null-Byte.
X	Ein Byte nach vorne rücken (letztes Byte wird überschrieben)
@	Null-Auffüllung bis zur absoluten Position.

Tabelle A.1: ***Schablonensymbole für pack***

Jeder Buchstabe kann von einer Zahl gefolgt sein, die angibt, wie oft der Buchstabe wiederholt werden soll. Mit der Funktion `unpack` können Elemente wieder aus einer binären Struktur extrahiert werden.

package

```
package NAMENSBEREICH
```

Die `package`-Funktion legt fest, dass alle Variablen in dem innersten Block, Subroutine, `eval` oder Datei, zu `NAMENSBEREICH` gehören. Weitere Informationen finden Sie in der ***perlmod***-Manpage.

pipe

```
pipe LESEHANDLE, SCHREIBHANDLE
```

pipe öffnet eine Pipe von LESEHANDLE zu SCHREIBHANDLE, wie der Systemaufruf des gleichen Namens.

pop

```
pop ARRAY
```

Die pop-Funktion entfernt das letzte Element in einem Array (verkürzt es um ein Element) und liefert es als skalaren Wert zurück. push (wird gleich besprochen) und pop sind sogenannte Stack- oder Stapel-Funktionen. Wenn Sie sich ein Array als einen Stapel Tabletts in einer Cafeteria vorstellen, wird mit pop das oberste Tablett vom Stapel entfernt.

pos

```
pos SKALAR
```

Liefert die Position in SKALAR zurück, an der die letzte m//g-Suche abgebrochen wurde. Wenn SKALAR nicht angegeben wird, wird $_ verwendet.

print

```
print DATEIHANDLE LISTE
print LISTE
print
```

Die print-Funktion wird dazu verwendet, die ihr im Listenkontext übergebenen Daten der Standardausgabe oder, wenn ein Datei-Handle spezifiziert wurde, diesem Datei- Handle zu übergeben. Wird keine Liste von auszugebenden Daten angegeben, wird standardmäßig der Inhalt von $_ ausgegeben. Beachten Sie, dass zwischen dem Datei- Handle und der Liste der auszugebenden Daten kein Komma stehen sollte. Um zum Beispiel einige Daten an das Datei-Handle DATEI auszugeben, würden Sie schreiben:

```
print DATEI $daten
```

oder:

```
print DATEI $daten, ' ', $mehr_daten, '\n';
```

printf

```
printf DATEIHANDLE LISTE
printf LISTE
```

printf wird verwendet, um die Ausgabe nach den Konventionen der sprintf- Funktion zu formatieren. Im wesentlichen ist

```
printf DATEIHANDLE FORMAT, LISTE;
```

identisch mit

```
print DATEIHANDLE sprintf(FORMAT, LISTE);
```

push

```
push ARRAY, LISTE
```

push wird verwendet, um ein Element an das Ende eines Arrays anzuhängen. Wenn Sie einen skalaren Wert in das Array »pushen«, wird das Array dadurch um ein Element länger, und der Wert wird dem letzten Element im Array zugewiesen. Stellen Sie sich den gleichen Stapel Tabletts aus unserem Beispiel zur pop-Funktion vor: In diesem Fall bewirkt die push-Funktion, dass ein Tablett oben auf den Stapel draufgelegt wird. Sie können auch mehrere Werte an ein Array anhängen, Sie müssen nur eine Liste als Argument für die push-Funktion angeben.

quotemeta

```
quotemeta AUSDRUCK
quotemeta
```

quotemeta liefert den Wert von AUSDRUCK zurück, wobei alle nichtalphanumerischen Zeichen mit Escape-Zeichen (Backslash) versehen sind. Verwendet $_, wenn AUSDRUCK weggelassen wurde.

rand

```
rand AUSDRUCK
rand
```

Die rand-Funktion liefert eine Zufallszahl zwischen 0 und AUSDRUCK. Wenn AUSDRUCK weggelassen wird, liefert die Funktion einen Wert zwischen 0 und 1 (1 nicht eingeschlossen). Unter srand erfahren Sie, wie Sie den Zufallszahlengenerator initialisieren.

read

```
read DATEIHANDLE, SKALAR, LAENGE, OFFSET
read DATEIHANDLE, SKALAR, LAENGE
```

Die read-Funktion wird verwendet, um eine beliebige Anzahl an Daten-Bytes von einem Datei-Handle in einen skalaren Wert zu lesen. Sie übernimmt vier Argumente: Datei-Handle, Skalar, Länge und Offset (Offset ist optional). Das Argument DATEIHANDLE gibt an, von welchem Datei-Handle die Daten zu lesen sind. Das Argument SKALAR definiert die Variable, der die Daten zugewiesen werden. LAENGE gibt an, wie viele Daten-Bytes gelesen werden, und OFFSET wird verwendet, wenn Sie die Daten nicht von Beginn des Strings an einlesen wollen, sondern erst ab einer späteren Position. Sehen Sie im folgenden ein Beispiel, das 1024 Byte vom 2048ten Byte an aus dem Datei-Handle DATEI einliest und der Variablen $chunk zuweist:

```
read DATEI, $chunk, 1024, 2048;
```

readdir

```
readdir VERZEICHNISHANDLE
```

readdir wird verwendet, um Einträge aus einem Verzeichnis zu lesen, das mit opendir geöffnet wurde. Wenn diese Funktion in einem skalaren Kontext verwendet wird, liefert sie den nächsten Eintrag in dem Verzeichnis zurück. In einem Listenkontext liefert sie alle übriggebliebenen Einträge im Verzeichnis. Wenn bereits alle Einträge in dem Verzeichnis gelesen wurden, lautet der Rückgabewert undefined.

readlink

```
readlink AUSDRUCK
```

Die readlink-Funktion liest den Wert eines symbolischen Links. Wenn auf der Plattform symbolische Links nicht implementiert sind, wird ein fataler Fehler zurückgeliefert. Wenn AUSDRUCK fortgelassen wird, wird der Wert in $_ verwendet.

recv

```
recv SOCKET, SKALAR, LAENGE, FLAGS
```

recv wird verwendet, um eine Nachricht auf einem Socket zu empfangen, wobei man sich der C-Funktion recvfrom bedient. Die Funktion empfängt LAENGE Bytes in der Variablen SKALAR von SOCKET. Dabei wird die Adresse des Absenders zurückgeliefert, es sei denn es liegt ein Fehler vor, in welchem Falle undefined zurückgeliefert wird. recv übernimmt die gleichen Flags wie der gleichlautende Systemaufruf.

redo

```
redo LABEL
redo
```

redo startet den aktuellen Schleifenblock neu, ohne die Testbedingung der Schleife neu zu bewerten. Wenn LABEL nicht angegeben wird, wirkt redo auf den innersten Block.

ref

```
ref AUSDRUCK
```

ref liefert ***wahr*** zurück, wenn AUSDRUCK eine Referenz ist, im anderen Falle ***falsch***. Wird AUSDRUCK nicht angegeben, wird $_ verwendet.

rename

```
rename ALTERNAME, NEUERNAME
```

Die rename-Funktion ändert den Namen der Datei ALTERNAME in NEUERNAME.

require

```
require AUSDRUCK
```

Die Funktion require wird meist dazu verwendet, eine externe Perl-Datei in das aktuelle Programm zu laden. Allgemein wird sie verwendet, um eine Art von Abhängigkeit von ihrem Argument zu schaffen. Wenn AUSDRUCK numerisch ist, wird die entsprechende Perl-Version für die Ausführung des Programms benötigt. Wenn kein Argument angegeben wird, wird $_ verwendet.

Um eine Datei zu laden, sollten Sie den Dateinamen als Argument zu require übergeben. Erfolgt die Übergabe des Dateinamens als einfaches Wort, wird automatisch .pl angehängt und :: durch / ersetzt, damit Standardmodule möglichst problemlos geladen werden können. Die benötigte Datei muss mit einer Anweisung enden, die als ***wahr*** ausgewertet wird. In der Regel enden Dateien, die für die Verwendung mit require erstellt werden, daher mit einer 1;-Anweisung.

reset

```
reset AUSDRUCK
reset
```

reset wird verwendet, um globale Variablen oder ??-Suchläufe zurückzusetzen, und steht häufig am Anfang einer Schleife oder in dem continue-Block am Ende einer Schleife. reset löscht die Werte aller Variablen, die mit den Zeichen aus AUSDRUCK beginnen. Wenn kein Argument angegeben wird, löscht reset alle ??-Suchläufe.

return

```
return AUSDRUCK
```

Die return-Funktion unterbricht die Ausführung von eval, einer Subroutine oder einer do-Datei und liefert den Wert von AUSDRUCK zurück. Wenn keine return- Anweisung vorgesehen wird, wird der Wert des zuletzt ausgewerteten Ausdrucks zurückgeliefert.

reverse

```
reverse LISTE
```

Die reverse-Funktion übernimmt einen skalaren Wert oder eine Liste als Argument. Im Falle eines skalaren Werts wird die Reihenfolge der Zeichen in dem Skalar umgedreht. So erzeugt zum Beispiel der Code `reverse "rot";` den Rückgabewert `tor`. Wenn `reverse` eine Liste übergeben wird, so wird die Reihenfolge der Elemente in der Liste umgedreht. `reverse ("rot", "gruen", "blau");` hat den Rückgabewert `("blau", "gruen", "rot")`.

rewinddir

```
rewinddir VERZEICHNISHANDLE
```

`rewinddir` setzt das Verzeichnis-Handle für ein mit `readdir` geöffnetes Verzeichnis zurück auf den ersten Eintrag in diesem Verzeichnis.

rmdir

```
rmdir DATEINAME
```

`rmdir` entfernt das in `DATEINAME` spezifizierte Verzeichnis, wenn es leer ist. Ist das Verzeichnis nicht leer oder die Funktion versagt aus einem anderen Grund, liefert sie `1` zurück. Im Erfolgsfall lautet der Rückgabewert `0`. Wenn `DATEINAME` nicht angegeben wird, wird der Wert in `$_` verwendet.

scalar

```
scalar AUSDRUCK
```

Erzwingt, dass der Wert in `AUSDRUCK` in einem skalaren Kontext ausgewertet wird, und liefert den Wert von `AUSDRUCK` zurück.

seek

```
seek DATEIHANDLE, OFFSET, VONWOAUS
```

`seek` wird verwendet, um die Position von `DATEIHANDLE` zu setzen. `VONWOAUS` kann einen der folgenden Werte annehmen: 0, um die Position auf `POSITION` zu setzen, 1, um `POSITION` zu der aktuellen Position hinzuzuaddieren, und 2, um die Position auf EOF plus `POSITION` zu setzen (aus offensichtlichen Gründen üblicherweise eine negative Zahl).

seekdir

```
seekdir VERZEICHNISHANDLE, POS
```

`seekdir` setzt die Position von `VERZEICHNISHANDLE` für die `readdir`-Funktion. `POS` muss ein Wert sein, der von `telldir` zurückgegeben wurde.

select

```
select DATEIHANDLE
select
```

Wenn der Aufruf ohne Argumente erfolgt, liefert `select` das gerade ausgewählte Datei-Handle zurück. Wenn Sie `select` ein Datei-Handle übergeben (oder einen Ausdruck, der einen Datei-Handle zurückliefert), wird dieser Datei-Handle zum Standard-Handle, an den alle Ausgaben gesendet werden. Mit anderen Worten: der Datei-Handle wird zur Standardausgabe. Wenn Sie also eine Reihe von Elementen in einem bestimmten Datei-Handle ausgeben wollen, ist es vielleicht einfacher, das Datei-Handle mit `select` einzurichten und dafür die Datei-Handle in den `print`- Anweisungen einzusparen.

semctl

```
semctl ID, SEMNUM, CMD, ARG
```

`semctl` ruft den System V IPC-Systemaufruf `semctl(2)` auf.

semget

```
semget SCHLÜSSEL, NSEMS, GROESSE, FLAGS
```

`semget` ruft den System-V-IPC-Systemaufruf `semget(2)` auf und liefert die Semaphore- ID zurück oder im Falle eines Fehlers `undefined`.

semop

```
semop KEY, OPSTRING
```

Diese Funktion ruft den System-V-IPC-Systemaufruf `semop(2)` auf, der Semaphore- Operationen wie Signalisieren und Warten ausführt.

send

```
send SOCKET, MLDNG, FLAGS, ZU
send SOCKET, MLDNG, FLAGS
```

Die `send`-Funktion sendet eine Nachricht über ein Socket. Wenn das Socket nicht verbunden ist, müssen Sie eine Adresse angeben, an die gesendet werden soll. Die Funktion übernimmt die gleichen Flags wie der Systemaufruf `send` und liefert die Anzahl der gesendeten Zeichen zurück, wenn sie erfolgreich ist (oder `undefined`, wenn ein Fehler aufgetreten ist).

setpgrp

```
setpgrp PID, PGRP
```

`setpgrp` setzt die Prozeßgruppe für die spezifizierte PID. Wenn als PID 0 angegeben wird, wird die Prozeßgruppe auf den aktuellen Prozeß gesetzt. Wenn `setpgrp(2)` nicht vom System unterstützt wird, wird ein fataler Fehler produziert.

setpriority

```
setpriority WELCHE, WER, PRIORITÄT
```

Setzt die Priorität für einen Prozeß, eine Prozeßgruppe oder einen Benutzer. Wenn `setpriority(2)` nicht unterstützt wird, führt dies zu einem fatalen Fehler.

setsockopt

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTWERT
```

`setsockopt` wird verwendet, um die spezifizierte Option für ein Socket zu setzen. Bei Auftreten eines Fehlers wird `undefined` zurückgeliefert. Verwenden Sie `undef` für `OPTWERT`, um eine Option zu setzen, ohne einen Wert für diese Option anzugeben.

shift

```
shift ARRAY
shift
```

Die `unshift`-Funktion ist das Gegenstück zu der `shift`-Funktion. Sie entfernt das erste Element aus einem Array und liefert es als skalaren Wert zurück. Die Indizes der anderen Elemente im Array werden um eins dekrementiert,

und das Array ist anschließend um ein Element kürzer als vorher. `unshift` wird in der Regel dazu verwendet, Argumente zu verarbeiten, die einer benutzerdefinierten Funktion übergeben wurden. Wie Sie wissen, werden Argumente mittels des Arrays `@_` an Funktionen übergeben. Mit Befehlen wie `$arg = unshift @_;` können sie Funktionsargumente verarbeiten, ohne Gedanken an ihre Indizes verschwenden zu müssen.

shmctl

```
shmctl ID, CMD, ARG
```

`shmctl` ruft den System-V-Systemaufruf `shmctl(2)` auf.

shmget

```
shmget SCHLUESSEL, GROESSE, FLAGS
```

`shmget` ruft den System-V-Systemaufruf `shmget(2)` auf.

shmread

```
shmread ID, VAR, POS, GROESSE
```

`shmread` ruft den System-V-Systemaufruf `shmread(2)` auf.

shmwrite

```
shmwrite ID, STRING, POS, GROESSE
```

`shmwrite` ruft den System-V-Systemaufruf `shmwrite(2)` auf.

shutdown

```
shutdown SOCKET, WIE
```

`shutdown` schließt eine Socket-Verbindung. Dabei wird die Art des Schließens in `WIE` vorgegeben, dass der Syntax des Systemaufrufs `shutdown` entspricht.

sin

```
sin AUSDRUCK
```

Liefert den Sinus von `AUSDRUCK` oder von `$_`, wenn kein Argument übergeben wird.

sleep

```
sleep AUSDRUCK
sleep
```

`sleep` versetzt das Programm in einen Schlaf - für die in `AUSDRUCK` definierten Sekunden oder, falls kein Argument übergeben wird, für unbegrenzte Zeit. `sleep` kann mit dem `SIGALRM`-Signal unterbrochen werden. Die Funktion liefert die Anzahl der tatsächlich geschlafenen Sekunden zurück.

socket

```
socket SOCKET, DOMÄNE, TYP, PROTOKOL
```

Die `socket`-Funktion wird dazu verwendet, das Socket zu öffnen, das mit dem Datei- Handle `SOCKET` verbunden ist. `DOMÄNE`, `TYP` und `PROTOKOL` werden auf die gleiche Weise spezifiziert wie im `socket`-Systemaufruf. Sie sollten

mit `use Socket;` das gleichnamige Modul importieren, bevor Sie die `socket`-Funktion aufrufen, um sicherzustellen, dass die korrekten Definitionen importiert sind.

socketpair

```
socketpair SOCKET1, SOCKET2, DOMÄNE, TYP, PAAR
```

Die Funktion `socketpair` erzeugt ein Paar unbenannter Sockets in der spezifizierten Domäne und mit dem spezifizierten Typ. Wenn die Funktion nicht implementiert ist, erfolgt ein fataler Fehler. Im Erfolgsfall liefert die Funktion ***wahr*** zurück.

sort

```
sort SUBNAME LISTE
sort BLOCK LISTE
sort LISTE
```

Die `sort`-Routine wird verwendet, um die Einträge einer Liste zu sortieren. Die Elemente der Liste werden sortiert zurückgegeben. Es gibt drei verschiedene Möglichkeiten, `sort` anzuwenden. Die einfachste davon ist der Aufruf von `sort` mit der zu sortierenden Liste als einzigem Argument. Der Rückgabewert ist eine Liste, die nach den Standardregeln für Stringvergleiche sortiert wurde.

Eine andere Möglichkeit besteht darin, eine Subroutine anzugeben, die die Elemente in der Liste vergleicht. Diese Subroutine liefert einen Integer kleiner als, gleich oder größer als Null zurück, je nachdem wie die Elemente in der Liste angeordnet werden sollen (in diesen Subroutinen werden besonders oft der `<=>`-Operator, der numerische Vergleiche durchführt, und der `cmp`-Operator, der für Stringvergleiche zuständig ist, verwendet).

Mit der im vorigen Absatz beschriebenen Subroutinen-Methode können Sie Listen nach anderen Kriterien als den standardmäßig vorgegebenen sortieren. Häufiger jedoch fügt man einfach nur einen Codeblock als erstes Argument an den Funktionsaufruf ein. Bestimmt ist Ihnen die `sort`-Funktion schon einmal in der folgenden Form begegnet:

```
@sortierteliste = sort { $a <=> $b } @liste;
```

Das obige Beispiel sortiert `@liste` in aufsteigender numerischer Reihenfolge und weist die zurückgelieferte Liste dem Array `@sortierteliste` zu. Die von der `sort`-Routine verglichenen Elemente werden zum Sortieren als `$a` und `$b` an den Codeblock (oder die Subroutine) gesendet. Der obige Codeblock vergleicht also die Elemente paarweise mit Hilfe des `<=>`-Operators. Betrachten wir noch ein paar weitere häufige Codeblocks, die zusammen mit der `sort`-Funktion verwendet werden:

```
# Sortiert in alphabetischer Reihenfolge (entspricht dem Standardverfahren)
@sortierteliste = sort {$a cmp $b } @liste;
# Sortiert in absteigender alphabetischer Reihenfolge
@sortierteliste = sort { $b cmp $a } @liste;
# Sortiert in numerischer Reihenfolge
@sortierteliste = sort { $a <=> $b } @liste;
# Sortiert in absteigender numerischer Reihenfolge
@ ortierteliste = sort { $b <=> $a } @liste;
```

splice

```
splice ARRAY, OFFSET, LAENGE, LISTE
splice ARRAY, OFFSET, LAENGE
splice ARRAY, OFFSET
```

`splice` ist das Schweizer Armeemesser unter den Array-Funktionen. Sie stellt eine Universalfunktion dar, um Elemente in ein Array einzufügen, daraus zu entfernen oder Elemente darin durch neue Werte zu ersetzen. `splice` kann mit bis zu vier Argumenten aufgerufen werden, von denen die letzten zwei optional sind. Das erste Argument sollte das Array sein, das mit `splice` bearbeitet werden soll. Das zweite Argument ist der Offset, der die Position im Array angibt, an der die Aktion stattfindet (wenn Sie vom hinteren Ende des Arrays abzählen wollen, geben Sie eine negative Zahl an). Das dritte - optionale - Argument ist die Anzahl der Elemente, die entfernt werden sollen

(wenn Sie dieses Argument fortlassen, werden alle Elemente von dem Offset bis zum Ende des Arrays entfernt). Der Rest der Argumente ist eine Liste von Elementen, die am Offset eingefügt wird. Das alles ist etwas verwirrend, deshalb möchte ich es anhand eines Beispiels veranschaulichen. Um alle Elemente eines Arrays ab dem zweiten Element zu löschen (zur Erinnerung: Array-Indizes beginnen mit einer 0), können Sie folgenden Code eingeben:

```
splice(@array, 2);
```

Um einen neuen skalaren Wert zwischen dem zweiten und dritten Element in ein Array einzufügen, ohne etwas zu entfernen, würden sie schreiben:

```
splice(@array, 2, 0, "neuer wert");
```

Um das zweite und dritte Element in einem Array durch drei neue Elemente zu ersetzen, würden Sie folgendes eingeben:

```
splice(@array, 2, 2, "rot", "gruen", "blau");
```

Sie sollten sich darüber im klaren sein, dass nach dem Aufruf von `splice` für ein Array die Elemente in dem Array neu indiziert werden, um die Änderungen in der Struktur widerzuspiegeln. Demzufolge sind in unserem Beispiel oben alle Indizes für die Elemente nach dem von uns eingefügten Element um eins inkrementiert, da wir zwei Elemente durch drei ersetzt haben.

split

```
split /MUSTER/, AUSDRUCK, LIMIT
split /MUSTER/, AUSDRUCK
split /MUSTER/
split
```

Die `split`-Funktion dient dazu, einen String in mehrere Teilstrings zu zerlegen und diese Teile als Liste zurückzugeben. Sie übernimmt bis zu drei Argumente: ein Muster, gemäß dem zerlegt wird, den String, der zerlegt werden soll, und eine Obergrenze für die Anzahl an Listenelementen, die zurückgegeben werden sollen (optional). Wenn Sie keinen zu zerlegenden String als Argument übergeben, wird der in `$_` gespeicherte Wert verwendet. Sie müssen auch nicht unbedingt ein Muster für die Zerlegung angeben. Wenn es fehlt, verwendet Perl die verschiedenen Whitespace-Zeichen als Begrenzer. Bei dem Musterargument handelt es sich immer um einen regulären Ausdruck der zwischen `//` steht. Um beispielsweise einen String an seinen Kommastellen zu zerlegen, sähe Ihr Muster so aus: `/,/`. Betrachten wir einige Beispiele:

```
# Leeres Muster zerlegt einen String in einzelne Zeichen
@buchstaben = split //, "wort";
# Ein Leerzeichen im Muster zerlegt einen Satz in seine
# einzelnen Wörter
@worte = split / /, "Dies ist ein Satz";
# Dieses Muster zerlegt den Satz bei jedem Whitespace-Zeichen und
# nicht nur bei Leerzeichen (entspricht dem Standard)
@worte = split /\s/, "Dies ist ein Satz";
# Das dritte Argument stellt sicher, dass nur die ersten zwei Elemente, die
# aus dem String extrahiert wurden, in der Liste zurückgeliefert werden.
($erstes, $zweites) = split /\s/, "Dies ist ein Satz", 2;
```

sprintf

```
sprintf FORMAT, LISTE
```

Die Perl-Funktion `sprintf` wird verwendet, um Strings nach den Konventionen für die C-Funktion `sprintf` zu formatieren. Nachstehend sehen Sie eine Tabelle, die eine Liste der Konvertierungen für `sprintf` enthält:

Format	Wofür es steht
%%	Ein Prozentzeichen
%c	Ein Zeichen

%s	Einen String
%d	Einen vorzeichenbehafteten Integer (in Dezimalnotation)
%u	Einen vorzeichenlosen Integer (in Dezimalnotation)
%o	Einen vorzeichenlosen Integer (in Oktalnotation)
%x	Einen vorzeichenlosen Integer (in Hexadezimalnotation)
%e	Eine Fließkommazahl (in wissenschaftlicher Notation)
%f	Eine Fließkommazahl (in fixer Dezimalnotation)
%g	Eine Fließkommazahl (in %e- oder %f-Notation)
%X	Entspricht %x, verwendet jedoch Großbuchstaben für die Hexadezimalnotation
%E	Entspricht %e, verwendet jedoch ein großes E
%G	Entspricht %g, verwendet jedoch ein großes G (falls anwendbar)
%p	Einen Zeiger, gibt die Speicherposition des Perl-Wertes in hexadezimaler Notation aus
%n	Speichert die Anzahl der bisher ausgegebenen Zeichen in der nächsten Variablen der Parameterliste

Tabelle A.2: sprintf-Formate

Weitere Informationen zu den Konventionen für `sprintf` finden Sie in der Manpage zu `printf(3)`.

sqrt

```
sqrt AUSDRUCK
```

`sqrt` liefert die Quadratwurzel von `AUSDRUCK` oder, wenn `AUSDRUCK` fehlt, von `$_` zurück.

srand

```
srand AUSDRUCK
```

`srand` initialisiert den Zufallszahlengenerator von Perl. Wenn Sie `AUSDRUCK` fortlassen, wird `srand(time)` verwendet. Diese Funktion sollten Sie nur einmal in Ihrem Programm verwenden.

stat

```
stat DATEIHANDLE
```

Die `stat`-Funktion trägt Informationen über die in `DATEIHANDLE` angegebene Datei zusammen und liefert eine Liste dieser Informationen zurück. Diese Funktion kann auch einen Ausdruck, der einen Dateinamen enthält, statt eines geöffneten Datei- Handles übernehmen. Wird kein Argument übergeben, verwendet `stat` den Wert von `$_` als sein Argument. Die von `stat` zurückgelieferten Daten stehen in einer Liste und umfassen:

- Die Gerätenummer des Dateisystems
- Die Inode-Informationen der Datei
- Den Dateimodus (Typ und Zugriffsrechte)
- Die Anzahl der harten Links auf die Datei
- Die UID und GID des Besitzers der Datei
- Den Geräte-Identifizierer (für besondere Dateien)
- Die Größe der Datei in Bytes
- Die Zeiten seit dem letzten Zugriff auf die Datei, seit der letzten Änderung und seit Änderung der Inode-Struktur
- Die Blockgröße der Datei
- Die Anzahl der verwendeten Blöcke

Werfen wir einen kurzen Blick auf die von stat zurückgelieferten Werte. Folgendermaßen würde man eine Liste, die von stat zurückgegeben wurde, einer Gruppe von Variablen zuweisen:

```
($dev,$inode,$mode,$uid,$gid,$rdev,
$size,$atime,$mtime,$ctime,$blksize,$blocks) = stat $filename;
```

study

```
study SKALAR
study
```

study nimmt sich Zeit, um SKALAR (oder $_, wenn SKALAR nicht angegeben wurde) zu studieren - mit dem Ziel, zukünftige Musterabgleiche auf dem Wert effizienter zu machen. Ob dies letzlich Zeit spart oder nicht, hängt davon ab, wie viele Musterabgleiche Sie machen wollen und welcher Art diese Abgleiche sind.

substr

```
substr AUSDRUCK, OFFSET, LAENGE
substr AUSDRUCK, OFFSET
```

substr wird verwendet, um eine Folge von Zeichen aus einem String zu extrahieren. Diese Funktion übernimmt drei Argumente, von denen das letzte optional ist. Die Argumente sind der Ausdruck, aus dem die Zeichen extrahiert werden sollen (kann ein skalarer Wert, eine Variable oder ein Aufruf an eine andere Funktion sein), die Position, an der mit dem Extrahieren der Zeichen begonnen werden soll, und optional die Anzahl der zu extrahierenden Zeichen. So liefert zum Beispiel der Aufruf `substr("foobar", 3, 2);` die Zeichenfolge ba zurück. Wenn Sie die Längenangabe (LAENGE) fortlassen, also `substr("foobar", 3);`, lautet der Rückgabewert bar. Sie können auch einen negativen Offsetwert verwenden. Dann wird die Position durch Rückwärtszählen vom Stringende an ermittelt. Das Beispiel `substr("foobar", -4, 2);` liefert ob zurück.

symlink

```
symlink ALTEDATEI, NEUEDATEI
```

Die Funktion symlink dient dazu, einen symbolischen Link von ALTEDATEI zu NEUEDATEI herzustellen. symlink löst einen fatalen Fehler aus, wenn das System symlink nicht unterstützt.

syscall

```
syscall LISTE
```

syscall ruft den Systembefehl auf, der als erstes Argument in LISTE spezifziert wurde. Die restlichen Elemente in LISTE werden dem Systembefehl als Argumente übergeben.

sysopen

```
sysopen DATEIHANDLE, DATEINAME, MODUS
sysopen DATEIHANDLE, DATEINAME, MODUS, PERMS
```

Öffnet die in DATEINAME spezifizierte Datei und verbindet sie mit DATEIHANDLE. Wenn die Datei noch nicht existiert, wird sie erzeugt.

sysread

```
sysread DATEIHANDLE, SKALAR, LAENGE, OFFSET
sysread DATEIHANDLE, SKALAR, LAENGE
```

Liest mit Hilfe des Systembefehls read(2) LAENGE-Bytes aus DATEIHANDLE in SKALAR ein. Zurückgeliefert wird die Anzahl der eingelesenen Bytes oder undefined, wenn ein Fehler auftrat. Verwenden Sie OFFSET, wenn Sie die eingelesenen Bytes nicht an den Anfang des Strings, sondern eine um OFFSET-Bytes verschobene Position

schreiben wollen.

sysseek

```
sysseek DATEIHANDLE, POSITION, VONWOAUS
```

Ähnlich der `seek`-Funkton. Allerdings wird der Systemaufruf `lseek(2)` anstelle von `fseek(2)` verwendet.

system

```
system LISTE
```

Die `system`-Funktion entspricht dem Aufruf `exec LISTE`, mit der Ausnahme, dass sie in einen neuen Prozeß verzweigt und in diesem Prozeß die Befehle in `LISTE` ausführt und dann zurückkehrt.

syswrite

```
syswrite DATEIHANDLE, SKALAR, LAENGE, OFFSET
syswrite DATEIHANDLE, SKALAR, LAENGE
```

`syswrite` versucht mit Hilfe des Systemaufrufs `write(2)` `LAENGE`-Bytes der Daten aus der Variablen `SKALAR` in `DATEIHANDLE` zu schreiben. Zurückgeliefert wird die Anzahl der geschriebenen Bytes oder, im Falle eines Fehlers, `undefined`.

tell

```
tell DATEIHANDLE
```

`tell` liefert die aktuelle Position für das spezifizierte Datei-Handle zurück. Wenn kein Datei-Handle angegeben wird, enthält der Rückgabewert die Position der zuletzt gelesenen Datei.

telldir

```
telldir VERZEICHNISHANDLE
```

`telldir` liefert die aktuelle Position in dem angegeben Verzeichnis-Handle.

tie

```
tie VARIABLE, KLASSENNAME, LISTE
```

`tie` bindet eine Variable an eine Paketklasse, die eine Implementierung für die Variable bereitstellt. `VARIABLE` ist der Name der zu bindenden Variablen und `KLASSENNAME` der Name der Klasse, die Objekte des korrekten Typs implementiert. Alle zusätzlichen Argumente werden der `new`-Methode der Klasse übergeben.

tied

```
tied VARIABLE
```

Falls `VARIABLE` an ein Paket gebunden ist, liefert `tied` eine Referenz auf das der `VARIABLE` zugrundeliegende Objekt zurück. Ist die Variable nicht gebunden, lautet der Rückgabewert `undefined`.

time

```
time
```

Die `time`-Funktion liefert die Anzahl an Sekunden zurück, die seit Beginn des systemspezifischen Referenzdatums

verstrichen sind. Dieser Zeitpunkt liegt auf den meisten Systemen bei 00:00:00 UTC, 1. Januar, 1970 und auf dem Macintosh- Betriebssystem bei 00:00:00 1. Januar 1904. Der zurückgelieferte Wert wird meistens `localtime` oder `gmtime` zur Formatierung übergeben.

times

```
times
```

`times` liefert ein Array mit vier Elementen zurück, das den Benutzer und die Systemzeiten für den aktuellen Prozeß und dessen Kinder enthält. Hier ein Beispiel:

```
($user, $system, $cuser, $csystem) = times;
```

truncate

```
truncate DATEIHANDLE, LAENGE
truncate AUSDRUCK, LAENGE
```

Verkürzt die Datei, die durch `DATEIHANDLE` oder `AUSDRUCK` spezifiziert wurde, auf die in `LAENGE` definierte Länge. Wenn `truncate` auf dem System nicht implementiert ist, wird ein fataler Fehler ausgelöst.

uc

```
uc AUSDRUCK
```

Im Gegensatz zu `lc`, das alle Buchstaben in einem String in Kleinbuchstaben konvertiert, konvertiert `uc` alle Buchstaben eines Strings in Großbuchstaben.

ucfirst

```
ucfirst AUSDRUCK
```

Liefert `AUSDRUCK` zurück, nachdem das erste Zeichen in einen Großbuchstaben konvertiert wurde.

umask

```
umask AUSDRUCK
```

`umask` wird verwendet, um die Standard-`umask`-Maske für den Prozeß zu setzen. Die Funktion übernimmt eine Oktalzahl (keinen String von Ziffern). Die Funktion `umask` ist nützlich, wenn Ihr Programm eine Reihe von Dateien erzeugt. Wenn `AUSDRUCK` fortgelassen wird, liefert `umask` die aktuelle `umask`-Maske zurück.

undef

```
undef AUSDRUCK
```

`undef` wird verwendet, um den Wert einer Variablen zu löschen. Die Funktion kann auf einer Skalarvariablen, einem ganzen Array oder einem ganzen Hash angewendet werden.

unlink

```
unlink (LISTE)
```

`unlink` löscht die Dateien, die ihr via `LISTE` übergeben wurden. Rückgabewert ist die Anzahl der Dateien, die erfolgreich gelöscht wurden. Wurde `unlink` keine Liste übergeben, wird `$_` als Argument verwendet.

unpack

```
unpack SCHABLONE, AUSDRUCK
```

`unpack` ist das Gegenstück zu `pack`. Sie übernimmt eine Datenstruktur und übersetzt sie in eine Liste, die auf einer `SCHABLONE` basiert. Das `SCHABLONEN`-Format ist das gleiche wie für `pack`.

unshift

```
unshift ARRAY, LISTE
```

Die `unshift`-Funktion fügt einen skalaren Wert als das erste Element in ein Array ein und verschiebt die Indizes aller folgenden Elemente im Array um eins.

utime

```
utime LISTE
```

`utime` ist das Perl-Äquivalent zu dem Unix-Befehl `touch`. Die Funktion setzt die Zugriffs- und Änderungszeiten für eine Liste von Dateien. Die ersten zwei Argumente müssen die numerischen Zugriffs- und Änderungszeiten für die Dateien enthalten. Bei allen folgenden Argumenten wird davon ausgegangen, dass es sich um Dateien handelt, deren Zugriffs- und Änderungszeiten geändert werden sollen. Die Funktion liefert die Zahl der Dateien zurück, die erfolgreich bearbeitet wurden.

values

```
values HASH
```

Diese Funktion liefert ein Array zurück, das die Werte (`values`) für die Elemente aus einem Hash enthält. Die Funktion ist damit das Pendant der Funktion `keys`, die ein Array der Schlüssel aus einem Hash zurückliefert.

vec

```
vec AUSDRUCK, OFFSET, BITS
```

`vec` behandelt einen String (spezifiziert in `AUSDRUCK`) als einen Vektor von vorzeichenlosen Integerwerten und liefert den Wert des von `OFFSET` spezifizierten Bitfeldes zurück.

wait

```
wait
```

`wait` wartet einfach auf das Ende eines Kindprozesses und liefert dann die PID dieses Prozesses zurück.

waitpid

```
waitpid PID, FLAGS
```

Die Funktion `waitpid` wartet darauf, dass ein bestimmter Kindprozeß (spezifiziert in `PID`) beendet wird, und liefert dann die Prozeß-ID für den toten Prozeß zurück.

wantarray

```
wantarray
```

`wantarray` liefert ***wahr*** zurück, wenn der Kontext der gerade ausgeführten Subroutine einen Listenwert benötigt. Wenn die Funktion in einem skalaren oder leeren Kontext aufgerufen wird, liefert sie ***falsch*** zurück. Um zu vermeiden, dass die ganze Subroutine ausgeführt wird, können Sie folgende Anweisung verwenden, die sicherstellt, dass die Subroutine in einem Listenkontext aufgerufen wurde:

```
return unless defined wantarray;
```

warn

```
warn LISTE
```

`warn` wird verwendet, um eine Nachricht an die Standardfehlerausgabe zu schicken, ohne dass dabei das Programm beendet wird. Abgesehen von der Tatsache, dass das Programm die Ausführung nicht unterbricht, entspricht diese Funktion der Funktion `die`.

write

```
write DATEIHANDLE
```

Die `write`-Funktion wird verwendet, um Daten mit Hilfe einer Schablone auszugeben, die mit der `format`-Funktion definiert wurde. Weitere Informationen hierzu entnehmen Sie bitte der ***perlform***-Manpage.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Überblick über die Perl-Module

Es gibt praktisch Hunderte von Modulen, die für die Arbeit mit Perl entwickelt wurden. Viele davon sind Teil der Standard-Perl-Bibliothek und werden mit Perl selbst ausgeliefert. Andere Module sind mit verschiedenen Perl-Ports zusammengefaßt, um Perl-Programmierer, die Perl auf einer bestimmten Plattform einsetzen, die speziellen Features dieser Plattformen verfügbar zu machen. Und dann gibt es noch das CPAN, das ***Comprehensive Perl Archive Network***, das als Ablage für benutzerdefinierte Module dient, die auf diesem Wege einer größeren Entwicklergemeinde zur Verfügung gestellt werden sollen. Dem CPAN werden ständig neue Module hinzugefügt, so dass jede Referenz ausgewählter Perl-Module bereits zum Zeitpunkt ihrer Drucklegung veraltet ist. Wenn Sie sich einen aktuellen Überblick über den Inhalt von CPAN verschaffen wollen, schauen Sie nach unter `http://www.perl.com/CPAN/CPAN.html` und seiner Begleitliste `http://www.perl.com/CPAN/modules/00modlist.long.html`.

Dieser Anhang enthält deshalb nur einige der bekanntesten und nützlichsten Perl- Module. Aus praktischen Erwägungen sind die Module dieses Anhangs genauso wie im CPAN kategorisiert. Wenn ein bestimmtes Modul nicht als Teil der Standardbibliothek gekennzeichnet ist, steht es im CPAN und muss heruntergeladen und installiert werden, bevor es genutzt werden kann. Nachdem ein Modul auf Ihrem System installiert wurde, können Sie mit `perldoc MODULE` oder `man MODULE` die Dokumentation zu diesem Modul lesen.

Bitte bedenken Sie, dass dieser Anhang keinesfalls als vollständig zu betrachten ist. Ich habe bei der Zusammenstellung nur die Module berücksichtigt, die nützlich sind, und häufig verwendet werden oder die meines Erachtens anderweitig interessant sind. Es gibt jedoch noch jede Menge weiterer Module zu Ihrer Auswahl. In der ***perlmod***- Manpage finden Sie eine komplette Liste der Module aus der Standard-Perl- Bibliothek. Die aktuelle Liste der CPAN-Module finden Sie unter den oben genannten URLs.

Beachten Sie außerdem, dass nicht alle Module auf allen Plattformen verfügbar sind. Sie können der Dokumentation, die zu Ihrem Modul gehört, entnehmen, ob es sich auf Ihrer Plattform ausführen läßt.

Pragmas

Pragmas sind Module, die Anweisungen enthalten, mit denen das Verhalten von Perl zur Kompilier- und zur Laufzeit gesteuert werden kann. Sie werden wie alle Module mit Hilfe der Funktion `use` importiert. Diese Pragmas sind Teil der Standardbibliothek von Perl. Zu allen Pragmas gibt es ausführliche Dokumentationen in den Manpages (oder dem ***perldoc***-System).

constant

Mit dem `constant`-Pragma können Sie zur Kompilierzeit konstante Variablen erzeugen.

diagnostics

Das `diagnostics`-Pragma erzeugt weitere Auskünfte in den Warnungen von Perl. Es entspricht dem Perl-Schalter `-w`, hat aber den Vorteil, das man es bei Bedarf für verschiedene Teile des Skriptes ein- und ausschalten kann. Die Manpage ***perldiag*** enthält ausführlichere Informationen zu den Warnungen, die vom `diagnostics`- Pragma ausgegeben werden.

integer

Das `integer`-Pragma teilt Perl mit, nur mit Integerarithmetik zu rechnen, was schneller ist als die Alternative, Fließkommazahl-Arithmetik.

lib

Das `lib`-Pragma bietet robustere Methoden zur Aufnahme weiterer Verzeichnisse in `@INC` (das Array, das die Liste der Verzeichnisse enthält, die nach Bibliotheken und Modulen durchsucht werden).

overload

Das `overload`-Pragma ermöglicht es Ihnen, Perl mitzuteilen, wie Operatoren für Ihre Objekte arbeiten sollen.

sigtrap

Mit dem `sigtrap`-Pragma können Sie das Drücken der Tastenkombination `[Strg]-[C]` während der Ausführung eines Perl-Programms abfangen und behandeln.

strict

Das `strict`-Pragma erlaubt es Ihnen, symbolische Referenzen, globale Variablen und reine Wörter (ohne Anführungsstriche und ohne Bedeutung) zu verbieten.

subs

Mit dem Pragma `subs` können Sie Subroutinennamen im voraus deklarieren

vars

Mit dem Pragma `vars` können Sie globale Variablen im voraus deklarieren, so dass sie unter dem `strict`-Pragma akzeptiert werden.

Elementare Perl-Module

Diese Module, die zum größten Teil in der Standardbibliothek von Perl zu finden sind, stellen die grundlegende Funktionalität zur Verfügung, um Perl zu erweitern oder das Verhalten von Perl zu steuern.

Autoloader

`Autoloader` ist ein Modul für Modulentwickler. Sie können damit Module nach Bedarf laden und nicht erst, wenn sie ausgeführt werden. `Autoloader` ist Teil der Standardbibliothek.

B

`B` ist eine Sammlung von Modulen, mit denen Sie auf den zugrundeliegenden Perl- Compiler zugreifen können. Diese Module ermöglichen die Erzeugung neuer Compiler-Schnittstellen, sogenannter ***Backends***, die Ihren Perl-Code in C-Code übersetzen, in plattformunabhängigen Byte-Code kompilieren oder in eine ausführbare Datei kompilieren können.

Carp

Das `Carp`-Modul ist vornehmlich für Modul-Entwickler interessant. Es bietet zusätzliche Funktionen zur Fehlerbehandlung, ähnlich `warn` oder `die`, aber mit dem Unterschied, dass sie berichten, wo der Aufruf der Subroutine, von der sie aufgerufen wurden, erfolgte und nicht, wo sie sich innerhalb der Subroutine selbst befinden. Normalerweise geben `warn` und `die` die Zeilennummer an, von der sie aufgerufen wurden. Wenn ein Skript ein Modul importiert und dieses Modul dann so verwendet, dass es ein `warn` oder `die` auslöst, liegt der Fehler irgendwo im Modul und nicht im Skript. Die ausgegebenen Zeilennummern von `warn` und `die` sind in so einem Fall beim Debuggen keine große Hilfe. `Carp`-Ersatzroutinen für `warn` und `die` liefern dagegen die Stelle in dem umgebenden Skript, wo der Fehler oder die Warnung auftrat.

Config

Das Config-Modul erlaubt es Ihnen, auf einen Hash namens %Config zuzugreifen, der die Einstellungen für die Perl-Installation enthält. Hier sind Informationen wie das Installationsverzeichnis, der Name Ihres Betriebssystems und die Unterstützung für bestimmte Funktionen gespeichert. Dieses Modul ist Teil der Standardbibliothek.

English

Mit dem English-Modul können Sie alternative Namen für die Perl-Variablen mit den kryptischen Namen, wie $_, $!, $| und so weiter, verwenden. Das Modul English ist Teil der Standardbibliothek.

Exporter

Exporter ermöglicht es Ihnen, Variablen von einem Paket zum anderen zu exportieren. Es ist Teil der Standardbibliothek.

Opcode

Opcode ist ein Modul für High-Level-Benutzer, die damit Perls interne Opcodes manipulieren können. Es ist Teil der Standardbibliothek.

PodParser

Hinter PodParser verbirgt sich ein Bündel an Modulen, die es Ihnen erlauben, Dokumentationen, die im POD-Format (Plain Old Documentation) geschrieben sind, zu manipulieren. So können Sie zum Beispiel mit Pod::Text (eines der Module im Bündel) die POD-Dokumenation in Nur-Text umformatieren.

Symbol

Symbol dient dazu, Perl-Symbole und ihre Namen zu handhaben. Das Modul Symbol ist Teil der Standardbibliothek.

Tie-Module

Die verschiedenen Tie-Module (Tie::Hash, Tie::Scalar, Tie::StdHash, Tie::StdScalar und Tie::StubstrHash) bieten Basisklassen für gebundene Daten. Sie sind alle Teil der Standardbibliothek.

Module zur Entwicklungsunterstützung

Diese Module sollen Ihnen helfen, bessere Skripts zu schreiben. Sie umfassen ein Modul zum Testen der Leistungsfähigkeit Ihrer Programme (Benchmarking) sowie weitere Tools zur Analyse Ihrer Programme.

Benchmark

Das Benchmark-Modul dient dazu, die Leistungsfähigkeit Ihrer Perl-Programme zu messen. Zu diesem Zweck wird die CPU-Zeit angezeigt, die in einem bestimmten Code-Abschnitt verbraucht wird. Das Modul ist Teil der Standardbibliothek.

Devel::Dprof

Devel::Dprof ist ein sogenannter Profiler für Perl, der Informationen über die Ausführungszeiten eines Programms und seiner einzelnen Subroutinen sammelt.

ExtUtils

Hinter ExtUtils verbirgt sich eine Reihe von Modulen, die Ihnen helfen, Perl- Erweiterungen (XSUBs) zu erstellen und zu paketieren. Viele von ihnen sind Teil der Standardbibliothek.

Usage

Mit dem Usage-Modul können Sie die Argumente von Subroutinen auf ihre Gültigkeit hin prüfen. Mit Hilfe dieses Moduls läßt sich leicht sicherstellen, dass die Argumente einer Subroutine einen bestimmten Wert enthalten oder von einem bestimmten Datentyp sind.

Betriebssystem-Schnittstellen

Während Perl mehr oder weniger plattformunabhängig ist, stellen Module ganz spezielle plattformspezifische Funktionalität zur Verfügung, die Sie nutzen können, wenn Sie mit einer ganz bestimmten Plattform arbeiten.

AppleII

Das Modul AppleII bietet Ihnen Zugriff auf AppleII-Grafiken auf Blockebene.

BSD::Resource

Wenn Ihr System die BSD-Funktionen getrusage(), getrlimit(), setrlimit(), getpriority() und/oder setpriority() unterstützt, können Sie mit dem Modul BSD::Resource über Perl darauf zugreifen.

BSD::Time

Wenn Ihr Betriebssystem die BSD-Funktionen gettimeofday() und settimeofday() unterstützt, stehen sie Ihnen mit diesem Modul auch in Ihren Perl-Programmen zur Verfügung

Env

Das Env-Modul macht Umgebungsvariablen über den %ENV-Hash Ihrem Perl- Programm verfügbar. Es ist Teil der Standardbibliothek.

Fcntl

Das Modul Fcntl, »Fcntl« steht für *File Control* (Kontrolle über Dateien), bietet Ihnen Zugriff auf Low-Level-Systemaufrufe, mit denen Sie Dateideskriptoren manipulieren können. Es ist Teil der Standardbibliothek.

Mac-Module

MacPerl enthält eine Reihe von Modulen, die Unterstützung für die Verwendung von Macintosh-Features in Perl-Skripten bieten. Daneben finden Sie auch im CPAN Module für den Zugriff auf weitere Macintosh-Features, beispielswese Mac::Types für den Zugriff auf Macintosh-Dateityp-Informationen, Mac::Apps::Launch zum Starten oder Verlassen von Anwendungen oder Mac::Apps:MacPGP, mit der Sie eine Schnittstelle zu der Mac-Version von PGP (***Pretty Good Privacy***) herstellen können.

OS2

Das OS2-Modul enthält Binärdateien und Dienstprogramme für die OS2-Version von Perl.

POSIX

Das POSIX-Modul bietet Zugriff auf die Funktionen, die ein Betriebssystem implementieren muss, um POSIX-kompatibel zu sein. Es ist Teil der Standardbibliothek, aber nur auf POSIX-kompatiblen Plattformen.

Netzwerkmodule

Netzwerkmodule bieten Schnittstellen zu geläufigen (und nicht so geläufigen) Netzwerkprotokollen und -diensten. Siehe auch »Module für HTML, HTTP, WWW und CGI«.

IPC::Signal

IPC::Signal bietet Ihnen eine Methode, um Signale des Betriebssystems zu behandeln.

Net::Bind

Net::Bind bietet Ihnen eine Schnittstelle, um Dämondateien zu binden.

Net::Cmd

Net::Cmd ist ein Modul, das eine Reihe von Netzwerkbefehlen für Protokolle wie SMTP und FTP enthält. Es ist Teil des libnet-Bündels.

Net::Country

Net::Country bildet die aus zwei Buchstaben bestehenden Ländercodes im Internet auf die vollen Ländernamen ab. Es weiß zum Beispiel, dass TM für Turkmenistan steht.

Net::DNS

Net::DNS bietet eine Schnittstelle für DNS-Server. Mit diesem Modul kann jede Art von DNS-Anfrage ausgeführt werden.

Net::Domain

Net::Domain erlaubt es Ihnen, den Hostnamen und die Domäne des aktuellen Hosts auszuwerten.

Net::FTP

Net::FTP ist eine Schnittstelle zu FTP, dem Dateitransfer-Protokoll (File Transfer Protocol). Es ist Teil des libnet-Bündels.

Net::Gen

Net::Gen ist eine allgemeine Socket-Schnittstelle für Perl. Es gehört zu dem Net-ext- Bündel.

Net::Ident

Net::Ident wird verwendet, um den Benutzernamen am anderen Ende einer TCP- Verbindung zu ermitteln. Dabei wird vorausgesetzt, dass auf der Remote-Maschine IDENTD ausführt wird.

Net::Inet

Net::Inet stellt die grundlegenden Dienste für Socket-basierte Kommunikation bereit und ist Teil des Net-ext-Bündels.

Net::Netrc

Net::Netrc stellt eine Schnittstelle zu .netrc-Dateien bereit. Es ist Teil des libnet- Bündels.

Net::NIS, Net::NISPlus

Net::NIS stellt eine Schnittstelle zu dem NIS (Network Information Service) von Sun bereit und ermöglicht es mehreren Computern, die Benutzerkonto-Informationen zu teilen. Net::NISPlus stellt eine Schnittstelle zu der zweiten Generation von Sun-NIS (NIS+) bereit.

Net::NNTP

Net::NNTP ermöglicht Ihnen die Kommunikation über das Network News Transfer Protocol (wird von Usenet verwendet). Es ist Teil des libnet-Bündels.

Net::Ping

Net::Ping erlaubt Ihnen, von einem Host die Informationen einzuholen, ob er an ist und wie lange es dauert, bis Ihre Pakete dort eingehen.

Net::POP3

Net::POP3 stellt eine Schnittstelle zu dem POP3-Protokoll bereit (POP steht für Post Office Protocol). In diesem Zusammenhang möchte ich auch auf das Modul Mail::POP3Client verweisen. Net::POP3 ist Teil des libnet-Bündels.

Net::SMTP

Net::SMTP stellt eine Schnittstelle zu dem Simple Mail Transfer Protocol bereit. Dieses Protokoll wird von dem meisten Computern im Internet für den Mail-Verkehr genutzt. Wenn Sie von einem Perl-Skript Mail senden wollen, sollten Sie dieses Modul nutzen. Es ist Teil des libnet-Bündels.

Net::SNPP

Net::SNPP ist eine Schnittstelle zu dem Simple Network Pager Protocol. Es ist Teil des libnet-Bündels.

Net::SSLeay

Net::SSLeay bietet eine Implementierung von Netscapes SSL-Protokoll (Secure Sokkets Layer), die in Perl-Programmen genutzt werden kann.

Net::TCP

Mit Net::TCP können Sie via TCP über Sockets kommunizieren. Es ist Teil des Net- ext-Bündels.

Net::Telnet

Mit Net::Telnet können Sie eine Telnet-Verbindung zu einem anderen Computer aufbauen und auf diesem Befehle ausführen.

Net::Time

Net::Time erlaubt Ihnen, die aktuelle Zeit von einem anderen Computer im Internet festzustellen. Dieses Modul ist Teil des libnet-Bündels.

Net::UDP

Net::UDP ermöglicht die Kommunikation über das UDP-Protocol. Es ist Teil des Net- ext-Bündels.

SNMP

SNMP ist eine Schnittstelle zu dem Simple Network Management Protocol.

Socket

Socket bietet zusätzliche Strukturen und Konstanten, um die Funktionen von Perl für die Socket-Kommunikation zu erweitern. Es ist Teil der Standardbibliothek.

Unterstützung für Datentypen

Mit den Datentypmodulen können Sie Daten manipulieren, die zu bestimmten nicht von Perl unterstützten Datentypen gehören.

Date::DateCalc

Date::DateCalc wird verwendet, um die Differenz zwischen zwei Datumsangaben oder die Zeit bis zu einem gegebenen Datum zu berechnen.

Date::Format

Date::Format enthält einige Datumsformatierungs-Routinen, die die Datumswerte in ASCII-Code konvertieren. Es ist Teil des TimeDate-Bündels.

Date::Language

Date::Language wird benötigt, um Datumsangaben in verschiedene Sprachen zu konvertieren. Es ist Teil des TimeDate-Bündels.

Date::Manip

Date::Manip wird verwendet, um Datumswerte zu manipulieren. Es kann Zeitdifferenzen und Offsets berechnen, und es kann Datumsangaben in das Unix- Zeitformat parsen. Außerdem akzeptiert es die Datumsangaben in einer Vielzahl von Formaten.

Date::Parse

Date::Parse konvertiert Datumsangaben in das Unix-Zeitformat. Es ist Teil des TimeDate-Bündels.

Math::BigFloat, Math::BigInt

Mit Math::Bigfloat und Math::BigInt können Sie Integer und Fließkommazahlen beliebiger Länge in Perl verwenden. Es ist Teil der Standardbibliothek.

Math::Complex

Mit Math::Complex können Sie komplexe Zahlen verarbeiten. Teil der Standardbibliothek.

Math::Fraction

Math::Fraction ermöglicht es Ihnen, in Perl Brüche zu verarbeiten.

Math::Matrix

Math::Matrix ermöglicht es Ihnen, in Perl mathematische Operationen auf Matrizen auszuführen.

Math::PRSG

Math::PRSG bietet Ihnen einen viel leistungsfähigeren Generator von Zufallszahlen als der von Perl (srand und rand). PRSG steht für Pseudo Random Sequence Generator, wird jedoch von Math::TrulyRandom noch übertroffen.

Math::Trig

Math::Trig bietet Ihnen Umkehrfunktionen und hyperbolische trigonometrische Funktionen. Diese sind auch in dem (gebündelten) POSIX-Modul zu finden.

Math::TrulyRandom

Math::TrulyRandom erzeugt Zufallszahlen aus Interrupt-Timing-Diskrepanzen.

Ref

Mit dem Ref-Modul können Sie Perl-Referenzen vergleichen und kopieren.

Sort::Versions

Sort::Versions erlaubt Ihnen, Versionsnummern zu sortieren, so dass Nummern wie 4.1a, 1.1.1 und 4.003_02 in der richtigen Reihenfolge stehen.

Statistics::Descriptive

Mit Statistics::Descriptive können Sie einfache statistische Operationen durchführen (zum Beispiel Mittel-, Median- und Modalwert sowie Standardabweichungen berechnen).

Time-Modules

In dem Bündel Time-Modules sind folgende Module zum Behandeln und Ändern von Zeit- und Datumswerten enthalten: Time::CTime, Time::JulianDay, Time::ParseDate, Time::Timezone und Time::DaysInMonth.

TimeDate-Bündel

Mit TimeDate verfügen Sie über ein weiteres Bündel von Modulen zum Konvertieren von Zeit- und Datumsangaben sowie zum Ändern von Zeit- und Datumswerten. Es enthält die folgenden Module: Date::Format, Date:.:Language, Date::Parse und Time::Zone.

Datenbankspezifische Module

Diese Module bieten Schnittstellen zu relationalen Datenbankdateien sowie zu DBM- Datenbankdateien.

AnyDBM_File

AnyDBM_File wird verwendet, um Dateien aller Datenbank-Manager-Formate zu handhaben, einschließlich DBM, GDBM, SDMB und so weiter. Es ist Teil der Standardbibliothek.

DBD

DBD steht für Datenbanktreiber (Database Driver). Es handelt sich dabei um ein Paket, das einen Perl-Treiber für viele verschiedene relationale Datenbanken bereitstellt. Als Entwicklung der Perl-DBI-Initiative bietet es eine Standardschnittstelle zu allen Datenbanken, die es unterstützt. Zur Zeit gibt es DBD-Treiber für Oracle, Sybase, Informix, DB2, solid, MSQL, MySQL, ODBC und andere Systeme. DBD funktioniert so, dass es den Backend-Treiber für das DBI-Modul bereitstellt, das dann die Standard- Datenbankschnittstelle implementiert.

DBI

DBI (Database Interface) implementiert eine Standardschnittstelle für den Zugriff auf relationale Datenbankdateien. Durch die Kombination des DBI-Moduls mit dem entsprechenden DBD-Treiber können Sie eine Verbindung zu den populärsten relationalen Datenbank-Engines im Handel herstellen.

DB_File

DB_File stellt eine Schnittstelle zu Berkeley DB bereit, das frei verfügbar ist. Diese Funktionalität finden Sie auch in AnyDBMF_File.

Msql

Das Modul Msql ist eine Schnittstelle zwischen Perl und Mini-SQL, einer relationalen Datenbank-Engine für Unix-Systeme. Diese Funktionalität erhalten Sie auch über DBD/ DBI.

Oraperl

`Oraperl` ist eine Perl4-Schnittstelle zu Oracle-Datenbanken. Es wurde durch `DBI` und dem `DBD`-Treiber für Oracle ersetzt. Ich erwähne es hier nur, da es noch häufig zu finden ist.

Pg

`Pg` ist eine Schnittstelle zu der frei verfügbaren relationalen Postgres-Datenbank- Engine.

SDBM_File

`SDBM_File` ist eine Schnittstelle zu SDBM, dem Simple Database Manager, der Teil der Standardbibliothek ist. Da SDBM gebündelt ist, können Sie mit Hilfe dieses Moduls portierbare Skripts schreiben.

Sybperl

`Sybperl` ist eine inzwischen veraltete Schnittstelle zu Sybase für Perl. Es wurde durch die Funktionalität in `DBI` und `DBD` ersetzt.

Benutzerschnittstellen

Die Module für die Benutzerschnittstellen bieten Programmierschnittstellen von Perl zu Benutzerschnittstellen-Toolkits wie ***X Windows***, ***curses*** und ***Tk***.

Curses

Das Modul `Curses` ermöglicht es Ihnen, ASCII-Benutzerschnittstellen mit ***curses*** zu erstellen.

Qt

Das Modul `Qt` ermöglicht es Ihnen, das Toolkit Qt GUI für ***X Windows*** zu benutzen, das von Troll Tech geschrieben wurde.

Term::AnsiColor

Mit `Term::AnsiColor` können Sie die Escape-Codes für die ANSI-Farbsequenzen verwenden.

Term::Gnuplot

`Term::Gnuplot` ermöglicht es Ihnen, die Low-Level-Zeichenroutinen des Gnuplot- Systems von Perl aus zu verwenden.

Tk

`Tk` ist eine Perl-Schnittstelle zu dem Tk-Grafik-Toolkit. Tk macht es einfach, Anwendungen in Skriptprachen (wie Perl) zu erstellen. Die häufig gestellten Fragen (FAQs) zu Perl/Tk finden Sie unter `http://www.perl.com/CPAN-local/doc/FAQs/tk/` .

X11::FVWM

`X11::FVWM` ist eine Schnittstelle zu der `Fvwm2`-API (ein Windows-Manager für ***X Windows***).

X11::Protocol

`X11::Protocol` stellt eine Schnittstelle zu dem grundlegenden ***X-Windows***-Protokoll bereit.

Dateisystem-Module

Diese Module bieten Ihnen Zugang zu Optionen des Dateisystems, die sonst nicht von Perl unterstützt werden.

Cwd

Cwd stellt drei Funktionen bereit, um das aktuelle Arbeitsverzeichnis von aus Perl einzustellen. Es ist Teil der Standardbibliothek.

File::Df

Eine Implementierung des Unix-Befehls df in Perl. Mit diesem Befehl können Sie feststellen, wie voll Ihre Festplatte ist.

File::Flock

File::Flock ist eine Hüllfunktion für den Systemaufruf flock. Der Systemaufruf flock() ermöglicht es Ihnen, Dateien zu sperren, damit nicht zwei Prozesse gleichzeitig in die gleiche Datei schreiben können. Auf diese Weise wird unnötiger Datenverlust verhindert. File::Lock ist eine weitere Hüllfunktion für flock.

File::Copy

Mit File::Copy können Sie Dateien über den Namen oder den Datei-Handle von Perl aus kopieren. Es ist Teil des File::Tools-Bündels.

File::Lock

File::Lock ist, analog zu File::Flock, eine Hüllfunktion für den Systemaufruf flock.

File::Lockf

File::Lockf ist ebenfalls ein Modul zum Sperren von Dateien. Es unterscheidet sich von den anderen Modulen, da es statt flock den Systemaufruf lockf verwendet. Das hat den Vorteil, dass auch Dateien auf Festplatten, die dem Netzwerk angehängt sind, gesperrt werden können.

File::Recurse

File::Recurse ermöglicht es Ihnen, eine Operation auf einem ganzen Verzeichnisbaum auszuführen. Es ist Teil des Moduls File::Tools.

File::Tools

File::Tools enthält die Module File::Copy und File::Recurse.

Module zur Stringverarbeitung

Im CPAN gibt es eine ganze Reihe von Modulen, die Ihnen Funktionen zur Bearbeitung von Strings zur Verfügung stellen. Bevor Sie selbst irgendwelchen Code zur Stringbearbeitung schreiben, sollten Sie auf alle Fälle prüfen, ob Ihnen nicht bereits jemand die Arbeit abgenommen hat.

Optionen-/ Argumentenverarbeitung

Diese Module dienen dazu, Optionen zu parsen, die Ihren Programmen in normalen Formaten übergeben wurden, einschließlich Befehlszeilenargumente und Windows ini-Dateien.

Getopt::Long

Getopt::Long ermöglicht es Ihnen, POSIX-Befehlszeilenoptionen, inklusive Gnu- Erweiterungen, einzuparsen. Es ist Teil der Standardbibliothek.

Getopt::Mixed

`Getopt::Mixed` ermöglicht es Ihnen, Befehlszeilenoptionen im Gnu-Stil zu verarbeiten. Es enthält Elemente von `Getopt::Long` und `Getopt::Std`.

Getopt::Std

Mit `Getopt::Std` können Sie Befehlszeilenoptionen in Ihren Perl-Programmen verarbeiten. Details dazu finden Sie in den Beispielen aus Kapitel 15, »Dateien und E/ A«.

IniConf

Mit `IniConf` können Sie die `ini`-Dateien von Windows lesen und schreiben.

Internationalisierung und Lokalisierung

Mit diesen Modulen ist es leichter, Programme zu schreiben, die für die Verwendung in anderen Lokalen (landesspezifischen Umgebungen) entworfen werden.

I18N::Collate

Mit `I18N::Collate` können Sie skalare Werte, die aus 8-Bit-Zeichen bestehen, auf der Basis der aktuellen Lokale vergleichen.

Locale::Codes

`Locale::Codes` bietet Zugriff auf die komplette Liste der zweibuchstabigen Länder- Codes.

Unicode

Das `Unicode`-Modul bietet Unterstützung für Unicode-Zeichensätze in Perl. Dieses Modul wird bald Teil der Perl-Kernsprache sein.

Verschlüsselung, Authentifizierung und Sicherheit

Hierunter sind Module zusammengefaßt, die Zugriff auf Algorithmen zur allgemeinen Sicherheit, Verschlüsselung und Authentifizierung bieten. Einige dieser Module lassen sich nur anwenden, wenn bestimmte Systemebenen-Bibliotheken vorhanden sind.

Authen::Radius

`Authen::Radius` bietet eine Schnittstelle zum Radius-Network-Authentication- Protokoll.

Crypt::Des

`Crypt::Des` ermöglicht es Ihnen, den DES-Blockverschlüsselungs-Algorithmus zu verwenden.

Crypt::Idea

`Crypt::Idea` ermöglicht es Ihnen, den IDEA-Blockverschlüsselungs-Algorithmus zu verwenden.

MD5

`MD5` ist ein Nachrichtenverarbeitungs-Algorithmus, der es Ihnen ermöglicht, mit RSA einen Einwege-»Daumenabdruck« zu einer Nachricht zu erzeugen, der dazu dient, deren Integrität zu gewährleisten,. Mit dem MD5-Modul können Sie MD5- Daumenabdrücke erzeugen.

PGP

Das PGP-Modul ist eine Schnittstelle zu der Verschlüsselungssoftware ***Pretty Good Privacy***.

Module für HTML, HTTP, WWW und CGI

Module, die für webspezifische Aufgaben benötigt werden, einschließlich der CGI- und Webclient-Programmierung.

Apache

Die Apache Perl-Module ermöglichen Ihnen, in Perl Module für den Apache- Webserver zu schreiben. Diese werden dann von einem Perl-Interpreter ausgeführt, der in dem Apache httpd-Server eingebettet ist. Der Vorteil liegt darin, dass Sie dann nicht immer den Perl-Interpreter starten müssen, wenn eines dieser Programme aufgerufen wird.

CGI

Das CGI-Modul, das ab Version 5.004 Teil der Standardbibliothek ist, läßt Sie schnell und einfach CGI-Programme aufsetzen, die Formulare erzeugen, verarbeiten und auf diese antworten. Es ist ein fast unentbehrliches Tool für alle CGI-Programmierer. Weitere Informationen dazu finden Sie in Kapitel 16, »Perl für CGI-Skripts«, oder in ***Sams Teach Yourself CGI-Programming in a Week***.

libwww

libwww ist ein großes Bündel von acht separaten Modulgruppen, mit denen Sie HTTP-Benutzeragenten in Perl erzeugen können. Des weiteren können Sie HTML parsen und übersetzen, MIME-codierten Inhalt bearbeiten etc. Die darin enthaltenen Bündel lauten File, Font, HTML, HTTP, LWP, MIME, URI und WWW.

Archivierung und Komprimierung

Dieser Abschnitt behandelt die Schnittstellen zu allgemeinen Komprimierungsalgorithmen.

Compress

Compress bietet eine Schnittstelle zu der Unix-Bibliothek zlib, mit der Sie Dateien mit der Standardkomprimierung von Unix (.Z) komprimieren können.

Convert::BinHex

Mit Convert::BinHex können Sie Header, Daten- und Ressourcenverzweigungen aus Macintosh-Dateien extrahieren.

Convert::UU

Convert::UU ermöglicht es Ihnen, Dateien mit uu zu codieren und zu decodieren.

Grafik-/ Bitmap-Manipulation

Dieser Abschnitt enthält Module zum Erstellen und Bearbeiten von Grafiken.

GD

GD ist eine Perl-Version der beliebten ***gd***-Bibliothek für C. Damit können Sie in Perl GIF-Grafiken erzeugen und bearbeiten.

Image::Size

Mit Image::Size können Sie die Größe einer Grafik bestimmen.

Image::Magick

PerlMagick ist eine Schnittstelle zu dem Grafik-Tool ***ImageMagick*** auf der Basis von ***X Windows***.

Mail und Usenet

Diese Module bieten Schnittstellen zu den allgemeinen Mail-Diensten und Usenet- News.

Mail::POP3Client

`Mail::POP3Client` ist eine clientseitige Schnittstelle für POP3-Mailserver.

News::Newsrc

`News::Newsrc` ermöglicht es Ihnen, mit Perl `.newsrc`-Dateien zu lesen und zu manipulieren. Die `.newsrc`-Dateien dienen dazu, die von einem Benutzer gelesenen Newsgroups zu verfolgen.

Programmsteuerung

Mit diesen Modulen können Sie größeren Einfluß darauf nehmen, wie sich ihr Programm zur Laufzeit verhält. Sehen Sie dazu auch das Pragma `sigtrap`.

AtExit

Das Modul `AtExit` ermöglicht Ihnen, Code zu schreiben, der ausgeführt wird, wenn Ihr Programm beendet ist. Eine ähnliche Funktionalität ist durch die `END`-Blocks gegeben (die am Tag 13 besprochen wurden). Auch das `POSIX`-Modul implementiert eine `atexit`-Funktion.

Religion

Das Modul mit dem netten Namen `Religion` ermöglicht es Ihnen, anzugeben, wo Ihre Perl-Programme hingehen, wenn sie sterben (`die`). Zu diesem Zweck erleichtert Ihnen das Modul den Zugriff auf die Funktionen `$SIG{__DIE__}` und `$SIG{__WARN__}`.

Datei-Handles und Eingabe/ Ausgabe

Diese Module stellen zusätzliche Dienste für die Dateieingabe und -ausgabe in Ihren Programmen bereit.

DirHandle

`DirHandle` ermöglicht es Ihnen, Verzeichnis-Handles als Objekte zu verwenden. Dieses Modul ist Teil der Standardbibliothek.

FileCache

Das Modul `FileCache` erlaubt es Ihnen, die Beschränkungen bezüglich der Anzahl der zu einem Zeitpunkt gleichzeitig geöffneten Dateideskriptoren zu umgehen. Das Modul ist Teil der Standardbibliothek.

FileHandle

Mit `FileHandle` können Sie Datei-Handles als Objekte verwenden. Es ist Teil der Standardbibliothek.

IO

Mit dem Modul `IO` können Sie die Module `IO::Handle`, `IO::Seekable`, `IO::File`, `IO::Pipe` und `IO::Socket` gleichzeitig laden. Alle diese Module sind Teil der Standardbibliothek.

Windows-Module

Windows-Module stellen Ihnen Funktionalität zur Verfügung, die speziell auf Windows- Plattformen zugeschnitten ist. Sie bieten Zugriff auf Nur-Windows-Dienste oder stellen die Funktionalität anderer Plattformen für Windows-Benutzer bereit.

libwin32

Das `libwin32`-Bündel ist eine Familie von Modulen, die Zugriff auf eine Vielzahl von Optionen der Windows-Plattform bieten. Sie sind Teil der ActiveState-Version von Perl für Windows. Viele von ihnen sind in Kapitel 18, »Perl und das Betriebssystem«, beschrieben.

Andere Module

Dieser Abschnitt beschreibt Module, die sich nicht unter den anderen Kategorien des CPAN einordnen lassen.

Archie

Archie ist ein Dienst, der den Inhalt von FTP-Sites katalogisiert. Mit dem `Archie`- Modul können Sie Archie-Anfragen von Perl aus erzeugen.

Business::CreditCard

Mit dem Modul `Business::CreditCard` können Sie die Gültigkeit von Kreditkarten überprüfen und den Kartentyp aus der Kartennummer ablesen.

CPAN

Das `CPAN`-Modul ermöglicht Ihnen, Perl-Module aus dem CPAN automatisch herunterzuladen und zu installieren. Dafür benötigen Sie das Modul `Net::FTP` und eine Internet-Verbindung.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl auf einem Unix-System installieren

In diesem Anhang möchte ich Ihnen zeigen, wie Sie Perl für Unix herunterladen und auf Ihrem Unix-Computer installieren. Im Gegensatz zu Windows und Macintosh gehört zur Installation von Perl auf einem Unix-Computer, dass Perl aus dem Quelltext neu kompiliert werden muss. Dank Tools wie dem ***Configure***-Programm und ***make*** ist dieser Prozeß jedoch glücklicherweise recht unkompliziert. Es setzt aber voraus, dass bestimmte Tools auf Ihrem Rechner vorhanden sein müssen, bevor Sie überhaupt beginnen können. Dieser Anhang erläutert Ihnen, welche Tools für die Perl-Installation erforderlich sind und wo man sie erhält. Außerdem erhalten Sie eine schrittweise Anleitung, wie die Installation durchzuführen ist.

Müssen Sie Perl installieren?

Normalerweise stehen Ihre Chancen recht gut, dass Sie Perl überhaupt nicht installieren müssen, es sei denn Sie arbeiten auf einem Ein-Platz-Unix-System mit Ihnen selbst als einzigem Benutzer. Perl ist für die Systemverwaltung unter Unix praktisch unverzichtbar, und ein Systemverwalter, der Perl noch nicht installiert hat, ist wirklich ein seltsamer Unix-Systemverwalter. Deshalb sollten Sie zuerst von Ihrem Unix-Systemprompt aus folgende Eingabe versuchen:

```
% perl -v
```

Wenn Sie eine Meldung wie `This ist perl, version 5.005_02` oder ähnlich erhalten, ist die Sache schon so gut wie geritzt. Dann können Sie diesen Anhang überspringen und in Kapitel 1 gleich voll in die Arbeit mit Perl einsteigen.

Erhalten Sie jedoch eine Meldung wie `perl: command not found` oder `This is perl, version 4` dann stehen Ihre Karten schon wesentlich schlechter. Entweder ist Perl noch gar nicht auf Ihrem System installiert oder die installierte Version ist älter (für dieses Buch benötigen Sie die Perl-Version 5 oder höher; ältere Versionen funktionieren nicht mit allen Beispielen des Buches). Oder es bedeutet, dass Perl zwar installiert ist, aber in Ihrem Suchpfad fehlt. Um das festzustellen, sollten Sie sich mal in den Verzeichnissen ***/usr/bin*** oder ***/usr/local/bin*** danach umschauen.

Wenn Sie Perl auch auf diesem Weg nicht finden können, sich aber auf einem System befinden, das nicht von Ihnen selbst verwaltet wird - etwa das System Ihrer Firma oder ein öffentliches ISP -, sollten Sie sich als nächsten Schritt an Ihren Systemverwalter oder die Support-Gruppe für Ihr System wenden, um in Erfahrung zu bringen, ob Perl bereits installiert wurde (und wenn ja, wo es sich befindet); wenn es sich um eine ältere Version handelt, sollten Sie darum bitten, Perl zu aktualisieren. Sie können zwar Perl auch auf einem System installieren, auf dem Sie keinen Administratorzugriff haben, aber im allgemeinen empfiehlt es sich, diese Aufgabe dem Systemverwalter zu überlassen.

Wenn Sie jedoch unter Ihrem eigenen Unix-System arbeiten - zum Beispiel einem Linux-System auf einer Partition Ihres Windows-Rechners - und Ihre Suche nach einer Perl-Installation ergebnislos verlaufen ist, dann obliegt Ihnen die Aufgabe des Systemverwalters, und Sie müssen Perl selbst installieren.

Perl herunterladen

Für die Installation von Perl auf Ihrem System können Sie zwei Wege einschlagen:

- Eine vorgegebene binäre Version herunterladen und installieren
- Die Quellcode-Version herunterladen, kompilieren und installieren

Der Weg zur binären Version

Die Installation der binären Version ist der einfachere Weg zu Perl - Sie müssen nichts kompilieren, sondern können nach dem Entpacken direkt loslegen. Binäre Versionen sind allerdings nur für ein paar ganz bestimmte Unix-Plattformen verfügbar und meistens nicht so aktuell wie der Perl-Quellcode. Außerdem besteht immer die Gefahr von Viren oder anderen unangenehmen Überraschungen, die in den binären Versionen verborgen sind, wenn Sie Ihren Code nicht aus einer zuverlässigen Quelle kopieren. Das Kompilieren des Perl-Quellcodes hingegen ist nicht besonders schwierig und, wenn Sie Unix für längere Zeit nutzen, werden Sie wahrscheinlich häufig kompilieren müssen. Schließlich hat die Quellcode-Installation den Vorteil, dass Sie immer über die neueste und vollständige Version verfügen. Deshalb sollten Sie stets dafür optieren, für Ihre Perl-Installation den Quellcode zu kompilieren.

Falls Sie es trotzdem vorziehen, nicht den Quellcode zu verwenden, sollten Sie zuerst beim Verkäufer Ihres Unix-Systems nach den binären Versionen von Perl nachfragen. Wenn Sie zum Beispiel unter Linux arbeiten, können Sie davon profitieren, dass die größeren Unix-Supportfirmen, wie Red Hat, kompilierte Versionen von Perl bereits auf ihren FTP-Sites anbieten. Oder wenn Ihre Unix-Software von einer Begleit-CD zu einem Buch oder von einer gekauften CD stammt, könnten Sie diese CD durchsuchen, um festzustellen, ob Perl zusammen mit dieser Software ausgegeben wurde (obwohl die Wahrscheinlichkeit in diesem Falle hoch ist, dass die Perl-Version veraltet ist).

Ansonsten können Sie noch, besonders für Solaris und Linux-Software, in dem Archiv unter `http://metalab.unc.edu/pub/`, früher auch Sunsite genannt, nachsehen. Für Solaris lautet die genaue Adresse `http://metalab.unc.edu/pub/packages/solaris/ sparc` und für Linux `http://metalab.unc.edu/pub/Linux`.

Die meisten Firmen und Sites haben eigene Methoden, Ihre binären Dateien zu paketieren. Sie werden nicht umhin können, den Anweisungen der jeweiligen Site zu folgen, um herauszufinden, wie die betreffende Binärversion von Perl auf Ihrem System installiert wird. Und vielleicht werden Sie am Ende feststellen, dass Sie besser dran gewesen wären, gleich den Weg über den Quellcode einzuschlagen, wenn Sie bedenken, wie lange sie nach den Binärdateien gesucht und ausgeknobelt haben, wie sie zu installieren sind.

Der Weg zum Quellcode (und den benötigten Tools)

Um Perl aus dem Quellcode zu kompilieren und zu installieren, benötigen Sie zusätzlich zu dem Perl-Code ein Reihe von weiteren Tools: ***tar*** und ***gzip***, um das Quell-Archiv zu entzippen, und einen C-Compiler, um die Quellcode zu kompilieren. Die Perl-Installation wird reibungslos durchzuführen sein, wenn Sie Superuser-Zugriff, das heißt ***root***-Zugriff, auf den Computer haben, auf den Perl installiert werden soll. Wenn Sie nicht der Systemverwalter für den PC sind, auf dem Sie Perl ausführen wollen, sind Sie wahrscheinlich besser beraten, wenn Sie sich zuerst an die betreffende Person wenden und sie bitten, die Installation für Sie vorzunehmen.

Um das Perl-Archiv zu dekomprimieren, benötigen Sie ***tar*** und ***gzip***. Das Dienstprogramm ***tar*** befindet sich auf fast allen Unix-Maschinen, und viele Unix- Computer haben auch ***gzip*** installiert, vor allem, wenn Sie eine freie Unix-Version verwenden, wie FreeBSD oder Linux. Fehlen Ihnen diese Dienstprogramme, müssen Sie sie im Internet suchen, herunterladen und bei sich installieren. Wie schon bei den Binärdateien von Perl ist es auch hier sehr wahrscheinlich, dass Sie diese Tools über Ihren Unix-Händler beziehen können. Und wenn Ihr Unix-System von CD ist, schauen Sie mal auf der CD nach, ob diese Tools Teil des Bündels sind. Solaris- Benutzer können vorkompilierte ***GNUtar***- und ***gzip***-Pakete von SunSite unter `http:/ /metalab.unc.edu/pub/packages/solaris/sparc` herunterladen.

Zweitens benötigen Sie einen C-Compiler. Es besteht eine gute Chance, dass auf Ihrem Computer bereits ***cc*** oder ***gcc*** installiert ist (geben Sie beim Eingabeprompt ***cc*** oder ***gcc*** ein, und schauen Sie, was passiert). Wenn Sie keinen C-Compiler auf Ihrem System finden, wenden Sie sich an Ihren Systemverwalter, um zu erfahren, wo Sie suchen sollen oder wie Sie einen Compiler installieren. Auch hier gilt, dass C- Compiler fast immer automatisch mit freien Unix-Systemen mitinstalliert werden. Aber auch viele handelsübliche und weit verbreitete Unix-Varianten installieren C- Compiler. Solaris-Benutzer können die vorkompilierten Versionen von ***gcc*** von dem oben erwähnten SunSite-URL herunterladen.

Nachdem Sie sichergestellt haben, dass Sie über alle notwendigen Tools für eine erfolgreiche Perl-Installation verfügen, können Sie den Perl-Quellcode herunterladen. Am einfachsten geht das, indem Sie mit Ihrem Webbrowser einfach `www.perl.com` ansteuern, wo sich unter `http://www.perl.com/CPAN/src/stable.tar.gz` die letzte gültige Version von Perl verbirgt.

Das Paket `stable.tar.gz` enthält den C-Quellcode, der sich auf praktisch jeder Plattform problemlos kompilieren

lassen sollte. Zum Zeitpunkt der Drucklegung dieses Buches lautet die aktuelle Versionsnummer `5.005_02`, die deshalb auch diesem Buch zugrunde liegt. Die ganz Mutigen unter Ihnen finden die neueste Entwicklerversion von Perl unter `http://www.perl.com/CPAN/src/devel.tar.gz`. Die Entwicklerversion trägt die Nummer `5.005_53` (was sich aber in der Zwischenzeit, wenn Sie dieses Buch lesen, geändert haben dürfte). Sie sollten diese Perl-Version nur verwenden, wenn Sie bereits einige Kenntnisse in Perl haben, wissen, was Sie tun, einige der neueren Features austesten wollen und bereit sind, ab und zu seltsames Verhalten in Kauf zu nehmen.

Alle `.gz`-Pakete weisen ein binäres Format auf. Deshalb sollten Sie sicherstellen, dass Sie sie als binäre Dateien herunterladen. Wenn Sie sie im Textformat herunterladen, werden sie nicht dekomprimiert (wenn Sie sie mit einem Browser herunterladen, brauchen Sie sich darüber nicht den Kopf zu zerbrechen).

Perl extrahieren und kompilieren

Nachdem es Ihnen geglückt ist, den Perl-Quellcode erfolgreich herunterzuladen, extrahieren Sie ihn aus seinem Archiv. Dazu sind zwei Schritte erforderlich: Erst wird mit ***gzip*** dekomprimiert und dann das Archiv mit ***tar*** expandiert. Entzippen Sie die Datei mit dem folgenden Befehl

```
gzip -d latest.tar.gz
```

Dann expandieren Sie die Datei mit:

```
tar xvf latest.tar
```

Während das `tar`-Programm ausgeführt wird, erscheint auf dem Bildschirm eine Liste der aus dem Archiv extrahierten Dateien. Um die Ausgabe müssen Sie sich nicht kümmern. Es wird ein Verzeichnis namens `perl5.005_02` oder ähnlich erzeugt, und alle Perl-Dateien werden dort hineinkopiert.

Ausführliche Installationsanweisungen finden Sie in den ***README***- und ***INSTALL***- Dateien. Ich werde den Vorgang in den folgenden Abschnitten zusammenfassen.

Das Konfigurationsprogramm ausführen

Bevor Sie Perl installieren, sollten Sie das Konfigurationsprogramm ausführen, um die Kompilierzeit-Optionen einzurichten. Wechseln Sie dazu in das Perl-Verzeichnis, das beim Expandieren des Archivs angelegt wurde (normalerweise `perl` plus einer Versionsnummer; zum Beispiel ***perl5.005_02***). Entfernen Sie dann alle bestehenden Konfigurationsdateien mit

```
rm -f config.sh
```

(Lassen Sie sich nicht beunruhigen, wenn es keine Konfigurationsdatei gibt. Auch das kann vorkommen.) Anschließend führen Sie das ***Configure***-Programm wie folgt aus:

```
sh Configure
```

Configure wird Ihnen eine Menge Fragen über Ihr System und über das Zielverzeichnis der Installation stellen. Sie können einen Großteil dieser Fragen übergehen wollen, indem Sie statt dessen den Befehl `sh Configure -d` eingeben. Dadurch wird ***Configure*** angewiesen, so viele Standardwerte wie möglich automatisch zu übernehmen und Perl in verschiedenen Standardverzeichnissen, je nach Plattform, zu installieren. Diese Anweisungen gehen davon aus, dass ***Configure*** vollständig ausgeführt wird.

*Bevor wir starten, möchte ich Sie darauf hinweisen, dass **Configure** geschrieben wurde, um Software unter alle möglichen komplexen Gegebenheiten der verschiedensten Plattformen zu konfigurieren. Für die meisten von Ihnen - außer denen, die sich gut mit Unix-Systemen und C auskennen - sind viele der Fragen etwas verwirrend oder scheinen keinen Sinn zu ergeben. Zu fast jeder Frage gibt es einen Standardwert, den Sie durch Betätigen der Eingabetaste bestätigen können. Im allgemeinen richten Sie keinen Schaden an, wenn Sie die Standardwerte von **Configure***

akzeptieren. Wenn Sie also nicht genau wissen, worauf die Frage hinausläuft, drücken Sie einfache die Eingabetaste.

Auch möchte ich anmerken, dass je nach Perl-Version, die installiert werden soll, die nachfolgend beschriebenen Fragen nicht oder in etwas anderer Reihenfolge erscheinen können. Wenn es anfängt, zu verwirrend zu werden, akzeptieren Sie einfach die Standardwerte. Damit sollten Sie aus dem Schneider sein.

Allgemeine Einstellungen

Zuerst stellt ***Configure*** sicher, dass Sie alles haben, was für die Perl-Installation nötig ist, dann versorgt es Sie mit einigen Anweisungen zum Installationsprozeß (die Sie lesen können, aber nicht müssen), und schließlich sucht es nach einigen der Dienstprogramme, die für den Installationsprozeß benötigt werden.

Um den Installationsprozeß zu beschleunigen, rät ***Configure***, auf welchem System Sie arbeiten, so dass einige Standardwerte vorgegeben werden können. Es ist sehr wahrscheinlich, dass diese Vermutung korrekt ist, so dass Sie lediglich die Eingabetaste betätigen und den Standardwert akzeptieren müssen.

Anschließend wird geraten, welches Betriebssystem Sie verwenden. Wenn die Vorgaben korrekt sind, drücken Sie für den Namen und die Version die Eingabetaste. Je nach Perl-Version werden Sie gefragt, ob Sie eine Threading-Version von Perl erstellen wollen. Nur wenn Sie genau wissen, was Sie tun, sollten Sie bereits zu diesem Zeitpunkt dafür optieren. Threads sind noch nicht voll ausgetestet. Wenn Sie später mehr über Perl wissen, können Sie Perl mit aktivierten Threads neu kompilieren.

Verzeichnisse und Grundkonfiguration

Im nächsten Schritt, der besonders wichtig ist, geben Sie die Verzeichnisstruktur an, unter der Perl installiert werden soll. In der Regel lautet die Vorgabe hierfür ***/usr/local***, wobei Binärdateien in ***/usr/local/bin***, Manpages in ***/usr/local/man*** und Bibliotheken in ***/usr/local/lib*** abgelegt werden. Sie können, wenn Sie wollen, eine andere Verzeichnisstruktur für die Installation von Perl wählen. So können Sie Perl zum Beispiel zusammen mit den Systemdateien direkt unter ***/usr*** installieren (***/usr/bin/*** , ***/usr/ lib***, ***/usr/man***). Geben Sie hier einfach das übergeordnete Verzeichnis an. Wenn Sie im Detail festlegen möchten, wo Sie die einzelnen Teile von Perl ablegen wollen, wird Ihnen dazu später noch Gelegenheit geboten.

Als nächstes müssen das Verzeichnis für site-spezifische und architekturabhängige Perl-Bibliotheken angeben. Wenn Sie bereits bei den anderen Verzeichnissen die Standardwerte akzeptiert haben, dürfte es damit auch hier keine Probleme geben.

Je nach der zu installierenden Perl-Version lautet die nächste Frage, ob Perl binär- kompatibel mit früheren Perl-Versionen sein soll. Wenn auf der Maschine, auf der Sie gerade Perl installieren, bereits Perl 5.001 installiert war, müssen Sie angeben, wohin die alten architekturabhängigen Bibliotheksdateien verschoben werden sollen. Beide Fragen lassen sich mit den Vorgabewerten beantworten, es sei denn Sie haben Grund, Änderungen vorzunehmen.

Configure prüft dann auf sichere `setuid`-Skripts, und Sie können entscheiden, ob Sie eine `setuid`-Emulation vornehmen wollen. Dabei ist es sehr wahrscheinlich, dass all diejenigen, die mit dieser Option nichts anfangen können, sie auch nicht benötigen. Akzeptieren Sie also die Vorgabe.

Die nächste Frage will wissen, welche Speichermodelle auf der Maschine unterstützt werden. Für die meisten lautet die Antwort **none**. Akzeptieren Sie die Vorgabe.

Compiler und Bibliotheken

Configure fragt Sie, welcher Compiler verwendet wird. Es stellt fest, welcher Compiler es am liebsten verwenden würde, und schlägt diesen als Vorgabewert vor.

Configure stellt außerdem fest, in welchen Verzeichnissen nach Bibliotheken zu suchen ist. Wenn Sie weitere Verzeichnisse kennen, die nach gemeinsamen Bibliotheken durchsucht werden sollen, fügen Sie diese der Liste hinzu und entfernen alle Verzeichnisse, die nicht durchsucht werden sollen. Meist kommt man auch hier mit dem Standardwert aus

Configure fragt nach der Dateiextension für gemeinsame Bibliotheken. Wenn Sie keine gemeinsamen Bibliotheken verwenden wollen, ändern Sie die Vorgabe in **None**. Aber wahrscheinlich werden Sie sie nicht ändern wollen.

Als nächstes prüft ***Configure***, ob einige bestimmte Bibliotheken auf Ihrem System vorhanden sind. Sie erhalten eine Liste der gemeinsam verwendeten Bibliotheken, die nach der Installation genutzt werden. Sie können aus der Liste Bibliotheken entfernen oder Bibliotheken hinzufügen. Aber die vorgegebene Liste wird wahrscheinlich ausreichen.

Danach stellt ***Configure*** einige Fragen zu Ihrem Compiler. Ändern Sie die Werte nur, wenn Sie ganz besondere Gründe dafür haben.

Eine der Compiler-Fragen, die ***Configure*** Ihnen stellen wird, betrifft den Speicherort der Perl-Binärdateien. Vorgegeben ist das übergeordnete Verzeichnis, das Sie zuvor angegeben haben, plus dem `bin`-Verzeichnis, zum Beispiel ***/usr/local/bin***. Hier haben Sie die Möglichkeit, korrigierend darauf einzuwirken.

Dokumentation und Netzwerke

Nachdem die compilerspezifischen Fragen abgehandelt sind, fragt ***Configure*** Sie, wo Sie die Man-Dateien von Perl ablegen wollen. Analog zu den Binärdateien besteht die Vorgabe aus dem übergeordneten Verzeichnis plus `man`. Diesen Wert müssen Sie nur ändern, wenn Sie die Manpages irgendwo anders auf Ihrem System unterbringen wollen. Sie werden auch nach den Erweiterungen für Ihre Manpages gefragt. Sie sollten die Vorgabe übernehmen.

Configure versucht jetzt, Ihren Host- und Domänennamen zu ermitteln. Liegt ***Configure*** richtig, können Sie die Vorgaben akzeptieren. Im andern Fall bearbeiten Sie die Vorschläge und geben die korrekten Werte ein. Der Hostname ist der vollständige Internet-Name für Ihr spezielles System (zum Beispiel. `www.typerl.com`), und der Domänenname ist der letzte Teil dieser Adresse (`typerl.com`).

Die nächste Frage betrifft Ihre E-Mail-Adresse. Perl wird versuchen, die richtige E-Mail-Adresse zu ermitteln, doch wird diese davon abhängen, auf welcher Maschine Sie Perl installieren. Unter Umständen müssen Sie die Adresse in Ihre allgemeine E-Mail-Adresse ändern. (So lautet die gefundene Adresse unter Umständen `person@einemaschine.einedomäne.com`, wenn Sie eigentlich die Adresse `person@einedomäne.com` benötigen.)

Perl erfragt auch die E-Mail-Adresse für die Person, die als Systemverwalter für Perl verantwortlich ist. Wenn Sie das nicht selbst sind, geben Sie hier die E-Mail-Adresse der verantwortlichen Person oder Gruppe ein (seien Sie so nett).

Sonstige Einstellungen

Als nächstes möchte Perl den Wert wissen, der in der Shebang-Zeile der Skripts gesetzt werden soll. Die Vorgabe ist hier mit größter Wahrscheinlichkeit korrekt, da Sie ***Configure*** bereits mitgeteilt haben, wo die Perl-Binärdateien installiert werden sollen. (Sorgen Sie sich nicht, wenn Sie nicht wissen, was ein Shebang ist, Sie werden es bald kennenlernen. Akzeptieren Sie die Vorgabe.)

Danach müssen Sie Perl mitteilen, wo die öffentlich ausführbaren Skripts abgelegt werden sollen. Gründe, warum Sie die Vorgabe überarbeiten sollten, werden von ***Configure*** gleich mitgeliefert.

Je nach Ihrer Perl-Version werden Sie nach weiteren Verzeichnispfadangaben gefragt, diesmal für die Bibliotheksdateien. Auch hier dürften die Vorgaben OK sein.

Die nächste Frage lautet, ob Sie statt `<stdio.h>` die experimentelle PerlIO- Abstraktionsebene verwenden wollen. Die Antwort ist mit Sicherheit nein.

Dann prüft ***Configure*** auf das Vorhandensein bestimmter Systembefehle und noch einige andere systemspezifische Dinge. Die Fragen zu diesen Punkten können Sie wahrscheinlich alle mit der Vorgabe bantworten. Auch wenn ***Configure*** Bedenken äußert (mein Linux-System gab Meldungen wie »WHOA THERE« und »Are you sure« aus), sollten Sie sich darüber hinwegsetzen und die Standardwerte akzeptieren.

Die letzte Frage von ***Configure*** betrifft die Art und Weise, in der die Perl- Erweiterungen zu laden sind (dynamisch oder statisch). Auch hier können Sie wahrscheinlich die Vorgabe übernehmen, der zufolge alle Erweiterungen

dynamisch zu laden sind.

Anschließend erhalten Sie die Gelegenheit, die Konfigurationsdatei ***config.sh***, die gerade erstellt wurde, manuell zu bearbeiten. Wahrscheinlich ist dies nicht nötig. Drü- cken Sie daher einfach die Eingabetaste.

Schließlich gibt Ihnen ***Configure*** die Chance, ***make depend*** auszuführen (nur zu), und wird dann beendet.

Detaillierte Anweisungen zu den meisten Optionen des ***Configure***-Programms finden Sie in der ***INSTALL***-Datei im Perl-Verzeichnis.

`make` ausführen

Der nächste Schritt nach der Erstellung der ***config.sh***-Datei durch das ***Configure***- Programm besteht darin, ***make*** im Perl-Verzeichnis aufzurufen. ***make*** kompiliert alle Binärdateien von Perl und bereitet sie für die Installation vor.

Auf einigen Systemen benötigen Sie das ***ar***-Programm, um Perl zu erstellen. Leider haben die meisten Benutzer ***ar*** nicht in ihrem Pfad. Wenn ***make*** nicht funktioniert, weil ***ar*** nicht gefunden werden kann, sollten Sie `man ar` eingeben, feststellen, wo ***ar*** gespeichert ist, und dieses Verzeichnis Ihrer `PATH`-Umgebungsvariablen hinzufügen (auf Solaris-Systemen befindet sich ***ar*** in ***/usr/ccs/bin***). Jetzt sollten Sie in der Lage sein, ***make*** erneut auszuführen und Perl erfolgreich zu erstellen. Es dauert einige Zeit, bis der ***make***-Prozeß beendet ist; Sie können also in der Zwischenzeit ruhig etwas anderes machen.

Bevor Sie Perl installieren, sollten Sie im Perl-Verzeichnis `make test` aufrufen, um sicherzustellen, dass alles korrekt erstellt wurde. Sobald das erledigt ist, können Sie mit `make install` alle Perl-Dateien in die Zielverzeichnisse verschieben, die Sie in ***Configure*** angegeben haben.

Eine letzte Frage, die Ihnen vielleicht gestellt wird, möchte Anwort darauf, ob Sie ***/usr/bin/perl*** mit dem Ort verknüpfen wollen, an dem Perl tatsächlich installiert ist. Viele Skripts gehen davon aus, dass Perl sich in ***/usr/bin/perl*** befindet. Wenn Sie Ihre Perl- Binärdateien mit ***/usr/bin/perl*** verknüpfen, ersparen Sie sich im weiteren Verlauf einiges an Arbeit.

Sobald `make install` beendet ist, sollte Perl zum direkten Einsatz installiert sein. Wenn Sie Perl in einem Standardverzeichnis in Ihrem `PATH` installiert haben, sollten Sie bei der Eingabe von

```
% perl -v
```

eine Versionsmeldung erhalten (`This ist perl, version 5`....). Jetzt sind Sie fertig und können damit beginnen, Perl zu lernen.

Weitere Informationen

Da Sie sich auf Unix befinden, wurde Perl speziell für Sie geschrieben. Alle wichtigsten Perl-Manpages, FAQs, Dienstprogramme, Module und Dokumentationen wurden ursprünglich für Unix geschrieben, so dass Sie eigentlich gleich loslegen können. Die zentrale Ablage für alle Dinge in Perl befindet sich unter `www.perl.com`. Starten Sie hier, und arbeiten Sie sich von dort aus weiter durch; berücksichtigen Sie auch die vielen Verweise, die immer mal wieder in diesem Buch auftauchen.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl für Windows installieren

Dieser Anhang beschreibt, wie Sie Perl für Windows-Plattformen, auch als »Win32 Perl« bekannt, herunterladen und installieren. Perl für Windows läuft auf Windows NT und Windows 95/98, wenn es auch auf Windows NT etwas robuster ist. (Ich persönlich konnte nur sehr geringe Unterschiede zwischen den beiden Plattformen feststellen.)

Mit dem Release der Perl-Version 5.005 wurde die Windows-Unterstützung in das Herz des Perl-Quellcodes mit aufgenommen und ist damit auf dem gleichen Stand wie die Unix-Version. Vorher gab es mehrere Versionen von Perl für Windows, die jeweils verschiedene Optionen unterstützten und der Unix-Version in verschiedener Hinsicht nachstanden. Die neu auf den Markt gebrachte Version 5.005 unterscheidet sich stark in Stabilität und Unterstützung für die Windows-Plattform von seinen Vorgängern. Wenn Sie noch mit einer alten Version von Perl für Windows arbeiten, möchte ich Ihnen ***sehr*** ans Herz legen, diese auf den neuesten Stand zu bringen, bevor Sie mit Perl in medias res gehen.

Um Perl für Windows zu installieren, können Sie zwei Wege einschlagen. Sie können

- den Perl-Quellcode herunterladen und selbst kompilieren und erstellen
- oder die ausführbare Version von Perl für Windows, manchmal auch ActivePerl genannt, von ActiveState herunterladen.

Wenn Sie den Weg über den Quellcode einschlagen, sind Sie immer auf dem neuesten Stand, was Bug-Fixes, experimentelle Neuerungen und Änderungen angeht. Sie müssen jedoch über einen modernen C++-Compiler verfügen (Visual C++ von Microsoft, Borland C++ und so weiter) und wissen, wie man große C-Projekte kompiliert und erstellt. Windows NT ist deshalb mit Sicherheit die bessere Plattform, um Perl selbst zu erstellen. Außerdem müssen Sie die Win32-Module (`libwin32`) selbst herunterladen, um Zugriff auf verschiedene Windows-Optionen, wie OLE und Prozesse, zu erhalten.

Die andere Alternative besteht darin, die binäre Version von Perl von ActiveState herunterzuladen. ActivePerl, so der Name dieser Version, enthält Perl für Windows, einige nette Installationsskripte, die Win32-Perl-Module, PerlScript (eine ActiveX- Skripting-Engine als Ersatz für JavaScript oder VBScript im Internet Explorer), Perl von ISAPI für Perl-CGI-Skripts und den PPM (Perl Package Manager), der die nachträgliche Installation von zusätzlichen Perl-Modulen wesentlich vereinfacht.

Da die ausführbare ActiveState-Version von Perl wahrscheinlich für die meisten Benutzer von Windows die bessere Wahl ist, beschäftigt sich dieser Anhang vornehmlich mit dem Herunterladen und Installieren dieser Version. Wenn Sie es dennoch vorziehen, Perl selbst zu erstellen, finden Sie dazu einen Abschnitt am Ende dieses Anhangs (»Den Perl-Quellcode herunterladen«), der Ihnen die ersten Schritte zeigt. Die dem Quelltext beigefügten ***README***-Dateien werden Ihnen dann weiterhelfen.

Perl für Windows herunterladen

Der erste Schritt bei der Installation von Perl für Windows auf Ihrem PC (entweder Windows NT oder Windows 95) besteht darin, das Installationspaket von der ActiveState-Website herunterzuladen.

ActiveState ist eine Firma, die es sich zur Aufgabe gemacht hat, Perl und Perl-Tools für die Windows-Plattform zu erstellen und zu unterstützen. Bei ActiveState finden Sie auch ein Perl Developer Kit, einen Perl-Debugger mit grafischer Benutzerschnittstelle und ein Plug-in für NT-basierte Server, das die CGI-Performance verbessert (keines dieser Pakete ist jedoch Teil des Hauptpakets Perl für

Windows).

Die neueste Version von Perl für Windows finden Sie unter `http://www.activestate.com/ActivePerl/download.html`. Die Links ganz unten auf der Seite (unter »Download the complete package«) erlauben es Ihnen, die Plattform (Intel oder Alpha) sowie die Methode zum Herunterladen des Installationspakets auszuwählen. Wenn Sie einen Webbrowser zum Herunterladen verwenden, wählen Sie HTTP.

Wenn Sie unter Win95 arbeiten, benötigen Sie zusätzlich das DCOM-Paket von Microsoft, um Perl für Windows nutzen zu können (für Windows NT oder Windows 98 ist dieses Paket nicht nötig). Sie finden dieses Paket unter `http:// www.microsoft.com/com/dcom/dcom1_2/default.asp`. Stellen Sie sicher, dass es installiert ist, bevor Sie Perl für Windows installieren.

Perl für Windows installieren

Die Datei, die Sie von ActiveState herunterladen, ist ein selbstextrahierendes zip- Archiv. Nachdem Sie die Datei auf die Festplatte heruntergeladen haben, doppelklicken Sie die Datei an, um die Installation zu starten und den Installations- Assistenten aufzurufen. Sie müssen zuerst den Perl-Lizenzbestimmungen zustimmen und die Installationshinweise lesen (lassen Sie sich nicht beunruhigen, wenn Sie nicht genau wissen, was gemeint ist). Der nächste Bildschirm zeigt an, wo Perl installiert wird. In der Regel ist dies `C:\Perl`. Sie können hier ein anderes Verzeichnis eingeben und mit **Next** zum nächsten Bildschirm gehen.

In diesem Bildschirm können Sie entscheiden, welche der Komponenten Sie bei sich installieren wollen. Zur Auswahl stehen:

- Perl: Das Kernstück der Perl-Installation.
- Perl für ISAPI: Wird nur benötigt, wenn Sie einen IIS-Webserver haben und mit Perl CGI-Skripts für den Server entwickeln wollen.
- PerlScript: Wird nur benötigt, wenn Sie das Plug-In PerlScript ActiveX verwenden wollen.
- Online-Hilfe und Dokumentationen: immer sinnvoll.
- Beispieldateien: ebenfalls sinnvoll.

Der nächste Bildschirm enthält die Optionen, die Sie bereits in den Installationshinweisen kennengelernt haben. Davon gibt es vier:

- Perl Ihrem Pfad hinzufügen.
- Die `.pl`-Dateien mit den ausführbaren Dateien von Perl verbinden.
- Die `.pl`-Dateien mit Ihrem IIS- oder WebSite-Webserver verbinden.
- Die `.plx`-Dateien mit IIS und Perl für ISAPI verbinden.

Sie sollten alle vier Optionen zulassen, es sei denn, Sie haben zwingende Gründe, auf diese Optionen zu verzichten.

Im nächsten Bildschirm richten Sie die Symbole für den **Programme**-Ordner ein. Erzeugen Sie einen neuen ActivePerl-Ordner, oder hängen Sie ihn an einen bereits bestehenden Ordner an.

Der letzte Bildschirm faßt nochmals alle von Ihnen eingestellten Optionen zusammen. Hier haben Sie noch einmal die Gelegenheit zurückzugehen und Ihre Entscheidungen zu widerrufen. Wählen Sie **Next**, um die eigentliche Installation und den Konfigurationsprozeß zu starten.

Wenn Sie sich unter Windows95 befinden, werden Sie im nächsten Bildschirm gefragt, ob Sie Ihre `autoexec.bat`-Datei ändern wollen, um Perl Ihrem Pfad hinzuzufügen. Damit können Sie dann Perl von der Befehlszeile ausführen. Nur wenn Sie genau wissen, wie man die `PATH`-Variable in der `autoexec.bat` ändert, sollten Sie diese Änderung allein vornehmen, ansonsten empfehle ich Ihnen, dem Perl- Installationsassis-tent diese Aufgabe zu überlassen (auf Windows NT wird dieser Bildschirm nicht angezeigt).

Nachdem der Installationsassistent die Dateien installiert hat und Perl für Windows konfiguriert wurde, können Sie die Release-Informationen einsehen oder das Programm verlassen (die Release-Informationen enthalten viele Hinweise zu dem, was sich seit dem letzten Release geändert hat; wenn Sie jedoch Perl zum ersten Mal installieren, dürften diese Informationen weniger interessant für Sie sein). Unter Windows 95 müssen Sie jetzt Ihren PC neu starten.

Jetzt können Sie Ihre Arbeit mit Perl beginnen. Wenn Sie einen Blick in das Verzeichnis `C:\Perl` (oder das von Ihnen gewählten Verzeichnis für Perl) auf Ihrem Computer werfen, werden Sie dort diese Unterverzeichnisse finden:

- `bin`: Enthält die ausführbare Datei für Perl und alle unterstützende Tools.
- `eg`: Beispiele. In den Dateien dieses Verzeichnisses finden Sie Beispiel-Skripts und verschiedene Perl-Codefragmente. (Die wenigsten dieser Beispiele sind dokumentiert, weshalb sie nicht immer besonders hilfreich sind.)
- `html`: Online-Dokumentation im HTML-Format. Sie können diese Dateien mit Ihrem bevorzugten Browser lesen. Beginnen Sie dabei mit der Datei ***index.html***.
- `lib`: Die wichtigsten Bibliotheksdateien.
- `site`: Zusätzliche Bibliotheksdateien von ActiveState.

Perl für Windows ausführen

Um Perl für Windows auszuführen, müssen Sie ein Befehlseingabefenster starten (eine DOS-Eingabeaufforderung unter Windows 95). Von der Befehlszeile aus können Sie mit der Option `-v` sicherstellen, dass Perl ordnungsgemäß installiert ist.

```
c:\> perl -v
```

Daraufhin erhalten Sie eine Nachricht, die Ihnen die Perl-Version sowie weitere Informationen zu Ihrer Perl-Installation mitteilt. Diese Nachricht signalisiert, dass Perl korrekt installiert worden ist und dass Ihr System die ausführbare Datei von Perl findet. Von hier aus können Sie direkt zu Tag 1 schreiten und mit dem Schreiben von Perl-Skripten beginnen.

Den Perl-Quellcode herunterladen

Wenn Sie beabsichtigen, Perl für Windows kommerziell zu nutzen, sollten Sie statt der binären Version den Perl-Quellcode herunterladen. Außerdem ist es manchmal recht hilfreich, wenn man den Quellcode vorliegen hat (vorausgesetzt Sie verstehen etwas von C). So können Sie zum Beispiel leichter herausfinden, wo etwas im Argen liegt, wenn sich Ihre Perl-Skripts nicht so verhalten, wie Sie es beabsichtigen. In beiden Fällen werden Sie dafür optieren, den Perl-Quellcode zusätzlich zu der ActiveState- Version oder an ihrer Stelle von Perl herunterzuladen.

Der Quellcode von Perl ist unter `http://language.perl.com/CPAN/src/` zu finden. In diesem Verzeichnis können Sie unter mehreren Perl-Versionen wählen. Die letzte stabile Version befindet sich immer unter `stable.tar.gz` (während der Entstehung dieses Buches lautete die Versionsnummer 5.005_02). Gleichzeitig steht die experimentelle Entwicklerversion unter `devel.tar.gz` zur Verfügung (nach meiner Recherche lautete diese Version 5.005_53, was sich aber in der Zwischenzeit schon geändert haben dürfte). Diesem Buch liegt die Version aus `stable.tar.gz` zugrunde.

Der Perl-Quellcode ist im Unix-Format in ***tar***-Archiven gespeichert und mit GNU Zip komprimiert. Es sind binäre Dateien, deshalb sollten sie im binären Format heruntergeladen werden. Haben Sie dieses Archiv auf Ihrer Festplatte gespeichert, können Sie die Perl-Quellcodedateien mit ***WinZip*** problemlos dekomprimieren und dearchivieren.

In der Datei ***README.win32*** finden Sie ausreichend Informationen darüber, wie man den Quellcode in eine funktionierende Perl-Installation kompiliert.

Weitere Informationen

Unabhängig davon, ob Sie die ActiveState-Version von Perl für Windows verwenden oder nicht, ist die Website von ActiveState unter `www.activestate.com` eine gute Ausgangsbasis, um Hilfe bei den ersten Schritten mit Perl für Windows zu finden. Von hier aus gelangen Sie zu den häufig gestellten Fragen zu Perl für Windows unter `http://www.activestate.com/support/faqs/win32` und den diversen Mailing-Listen zu diesem Thema (`http://www.activestate.com/support/mailing_lists.htm`.).

ActiveState bietet auch Unterstützung zu ihrer Version von Perl an. Details finden Sie unter `http://www.activestate.com/support/`.

Und dann gibt es immer noch die Perl-Ressourcen unter `http://www.perl.com` sowie die von mir über das Buch verteilten Hinweise auf Informationsquellen.

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen nächstes Kapitel

Perl für Macintosh installieren

Dieser Anhang beschreibt, wie Sie die Perl-Software auf einen Macintosh-Rechner herunterladen und dort installieren. Primär wurde Perl zwar für die Unix-Plattform entwickelt, aber inzwischen gibt es mit MacPerl eine Portierung auf den Macintosh, das die meisten Features der neuesten Perl-Versionen für Unix und Windows unterstützt. Darüber hinaus verfügt MacPerl über eine Reihe Macintosh-spezifischer Komponenten, die es dem Programm erlauben, viele der Mac-Betriebssystem- Optionen zu nutzen.

MacPerl benötigt das Betriebssystem System 7 oder höher, um erfolgreich ausgeführt zu werden. Sie können es entweder auf einem PowerPC oder einem 68K-Macintosh laufen lassen - vorausgesetzt, dass zumindest 4 Mbyte RAM und ein 68020-Prozessor oder höher zur Verfügung stehen.

MacPerl herunterladen

Der erste Schritt bei der Installation von MacPerl besteht darin, des Installationspakets habhaft zu werden. Sie finden das MacPerl-Binärpaket fast auf jeder bekannteren Macintosh-Softwaresite oder unter `www.perl.com`:

```
http://www.perl.com/CPAN/ports/mac/
```

In diesem Verzeichnis finden Sie eine Reihe von Dateien zu Ihrer Auswahl mit Namen wie ***Mac_Perl_520r4_appl.bin***, ***Mac_Perl_520r4_bigappl.bin*** oder ***Mac_Perl_520r4_src.bin***. Die Zahlen beziehen sich auf die aktuelle Version (die sich, wenn Sie dieses Buch lesen, bereits geändert haben kann). Die Extension `.bin` weist darauf hin, dass es sich um eine MacBinary- Datei handelt. Der mittlere Teil (`appl`, `bigappl`, `src`, `tool` und so weiter) bezieht sich auf das Installationspaket.

> *In diesem Verzeichnis können unter Umständen auch Pakete für Perl 4 enthalten sein - **Mac_Perl_418_appl.bin** oder ähnlich. Diese sollten Sie nicht wählen. Entscheiden Sie sich immer für die höchste Versionsnummer, die verfügbar ist. Die Beispiele in diesem Buch basieren auf Perl5 und lassen sich mit früheren Versionen vielleicht nicht ausführen.*

Zuerst müssen Sie die Paketversion herunterladen, die der Name »appl« aufweist (zum Beispiel ***Mac_Perl_520r4_appl.bin***). Wenn Sie auf einem Macintosh-PowerPC arbeiten, erhalten Sie damit alles, was Sie für die Anwendungsversion von MacPerl benötigen: die Anwendung, alle Bibliotheken und die Dokumentationen.

Je nachdem ob Sie eine 68K-Mac oder einen MPW verwenden, benötigen Sie noch einige zusätzliche Pakete.

Wenn Sie auf einem 68K-Macintosh arbeiten, ist die Hauptanwendung eigentlich ausreichend. Sie haben damit allerdings keinen Zugriff auf die Mac-Toolbox- Bibliotheken oder die GD-Bibliothek (GIF-Grafiken). Für den Zugriff auf diese Extrabibliotheken benötigen Sie zusätzlich zu dem Hauptpaket eine besondere Version des Anwendungspakets: Das gesuchte Paket trägt im Dateinamen die Bezeichnung ***bigapple*** oder ***appl_cfm68K***. Was ist der Unterschied? Bei ***bigappl*** sind die Bibliotheken in der Anwendung selbst kompiliert. Dadurch entsteht eine große Anwendung (big application), die mehr Speicher benötigt. Die letztere Version, ***appl_cfm68K,*** ermöglicht es Ihnen, die zusätzlichen Bibliotheken dynamisch, das heißt nach Bedarf, zu laden. Damit Sie diese Version nutzen können, müssen die folgenden Dateien als Teil Ihres Systems installiert sein (prüfen Sie dazu Ihren Systemordner):

- CFM-68K Runtime Enabler, Version 4.0 oder höher
- AppleScript Lib, Version 1.2.2 oder höher
- ObjectSupportLib, Version 1.2 oder höher

Wenn Sie unter MacOS 8 oder höher arbeiten, benötigen Sie diese Dateien nicht. Sie können problemlos mit der `appl_cfm68K`-Version arbeiten. Die einfache Wahl: Laden Sie beide, ***appl*** und ***bigappl***, herunter.

Wenn Sie MPW (Macintosh Programmer's Workshop) verwenden, steht Ihnen auch ein MPW-Tool für MacPerl zur Verfügung. Zusätzlich zu dem ***appl***-Paket benötigen Sie dann das ***tool***-Paket (***Mac_Perl_520r4_tool.bin***). Und wenn Sie mit MPW auf einem 68K-Mac arbeiten, benötigen Sie ***appl***, ***tool*** und entweder ***bigappl*** oder ***appl_cfm68K*** sowie ***bigtoolI***; außerdem sollten Sie sicherstellen, dass Sie alle benötigten Bibliotheken haben.

Daneben gibt es noch zwei weitere Pakete, die Ihr Interesse finden könnten:

- ***src***, das den Quellcode für MacPerl enthält
- ***appl_only***, das dem Paket ***appl*** entspricht, aber auf Bibliotheken und Dokumentationen verzichtet

Alle Installationspakete von MacPerl sind im MacBinary-Format in selbstextrahierenden Archiven unter `www.perl.com` gespeichert. Nachdem Sie beispielsweise die Datei ***Mac_perl_520r4_appl.bin*** auf Ihren Computer heruntergeladen haben, sollten Sie dort eine Anwendung mit Namen ***Mac_Perl_520r4_appl*** (ohne die Extension ***.bin***) wiederfinden. Trägt die Datei noch ihre Extension ***.bin***, wird ein Tool wie Stuffit oder MacBinary II benötigt, um die Datei in ein Archiv zu konvertieren, das dann durch Doppelklicken gestartet werden kann. Diese Hilfssoftware finden Sie auf jeder bekannteren Mac-Software-Site (versuchen Sie es mal mit `www.shareware.com` oder `www.download.com`).

MacPerl installieren

Sobald Sie alle Installationsdateien, die Sie benötigen, auf Ihren Macintosh heruntergeladen haben, können Sie jede einzeln installieren. Um das Installationsprogramm zu starten, müssen Sie lediglich auf das betreffende Symbol doppelklicken.

Beginnen Sie mit der Hauptanwendung von MacPerl - unabhängig davon, ob sie sich auf einem PowerPC oder einem 68K-PC befinden oder ob Sie MPW nutzen oder nicht. Lesen und akzeptieren Sie zuerst die MacPerl-Readme-Datei. Danach erscheint ein typischer Macintosh-Installationsdialog, in dem Sie wählen können, auf welcher Festplatte und in welchem Ordner MacPerl installiert werden soll.

Mit »Easy Install« wählen Sie die einfachste Form der MacPerl-Installation. Diese Option installiert MacPerl zusammen mit den gemeinsamen Bibliotheken auf Ihrer Plattform. Wenn Sie »Custom Install« wählen, wird eine sogenannte »fette« Binärdatei installiert, die sowohl auf dem 68K-Macintosh als auch auf dem PowerPC ausführbar ist. Sie benötigen diese Variante nur, wenn Sie Perl auf einer Festplatte installieren, auf die von verschiedenen PCs zugegriffen werden kann. Sie können darüber hinaus wählen, ob Sie die gemeinsamen Bibliotheken für MacPerl installieren wollen oder nicht.

Der Installationsprozeß installiert alles in einen Ordner mit dem überraschenden Namen MacPerl. In dem Ordner befinden sich:

- Die MacPerl-Anwendung
- Die Ordner ***lib*** und ***ext***, in denen all Ihre Perl-Bibliotheken und -Erweiterungen installiert sind.
- Ein Ordner namens ***pod***, in dem die MacPerl-Hilfsdateien gespeichert sind
- Eine Anwendung namens ***Shuck***, mir der Sie die `pod`-Dateien lesen können
- Viele Skripts, Beispiele und README-Dateien

Wenn Sie mit der Hauptanwendung fertig sind, können Sie sich den anderen Paketen zuwenden, die Sie vielleicht heruntergeladen haben. Wenn Sie mit einem 68K-Mac arbeiten, installieren Sie am besten entweder ***bigappl*** oder ***app_cfm68K***. Wenn Sie mit *MPW* arbeiten, installieren Sie das Paket ***tool***. Und wenn Sie mit einem 68K-Mac arbeiten, der *MPW* verwendet, installieren Sie zuerst ***appl***, dann ***bigappl*** oder ***appl_cfm68K***, gefolgt von ***tool*** und dann ***bigtool*** (in genau dieser Reihenfolge).

Die MacPerl-Anwendung starten

Wenn Sie die Installation beendet haben, können Sie MacPerl starten und mit dem Schreiben von Perl-Skripts beginnen. Um MacPerl zu starten, klicken Sie einfach auf das MacPerl-Symbol.

Wenn Sie neben der Arbeit in MacPerl die Hilfe aufrufen wollen, sollten Sie die ***Shuck***-Anwendung starten. ***Shuck*** bietet leichten Zugriff auf alle MacPerl- und Perl- Dokumentationen und ermöglicht es Ihnen, Perl-Dokumentationsdateien zu lesen, die in den Perl-Modulen eingeschlossen sind. Nutzen Sie auch die Optionen im **Go**-Menü, um auf die normale MacPerl-Dokumentation zuzugreifen.

MacPerl von MPW aus ausführen

Wenn Sie das Perl-Tool MPW verwenden, werden Sie sich wohl auch bereits mit MPW auskennen. Mit dem Perl-Tool MPW könne Sie Perl-Skripts von MPW aus bearbeiten und Perl von der MPW-Befehlszeile ausführen. In dieser Hinsicht ist es näher an der originalen Unix-Version von Perl als die eigentliche Macintosh- Anwendung.

Dieses Buch geht allerdings davon aus, dass die meisten Leser die Anwendungsversion von MacPerl verwenden; wir konzentrieren uns deshalb auf dieses Paket.

Weitere Informationen

Während ich an diesem Buch schreibe, lautet die aktuelle Version von MacPerl 5.2.0r4, die der Perl-Version 5.004 für Unix entspricht. Die 5 in der MacPerl-Version bezieht sich auf die Hauptziffer der Versionsnummer für die Haupt-Perl-Anwendung und der Teil r4 auf die untergeordnete Ziffer. Dagegen beschreibt der Teil .2.0 die untergeordnete Versionsnummer von ***MacPerl*** und nicht von Perl selbst. Einige der neueren Elemente von Perl sind für MacPerl unter Umständen nicht verfügbar, doch prinzipiell dürften die Unterschiede unerheblich sein (Optionen, die in diesem Buch mit 5.005 gekennzeichnet wurden, sind vielleicht nicht für MacPerl verfügbar).

Die offizielle MacPerl-Homepage lautet `http:/www.iis.ee.etz.ch/~neeri/ macintosh/perl.html`. Ausführlichere und aktuellere Informationen zu MacPerl stehen jedoch in den häufig gestellten Fragen zu MacPerl unter `http:// www.perl.com/CPAN/doc/FAQs/mac/MacPerlFAQ.html` zur Verfügung. Auch die MacPerl-Seiten bei Prime Time Freeware unter `http://www.ptf.com/macperl/` enthalten eine Fülle an Informationen.

Wenn Sie vorhaben, viel mit MacPerl zu arbeiten, sind Sie vielleicht an einer Mailing- Liste interessiert. Unter `http://www.ptf.com/macperl/depts/mlist.html` erfahren Sie, wie Sie sich anmelden und das Archiv der Mails einsehen.

Und dann gibt es immer noch die Standard-Perl-Ressourcen unter `www.perl.com` sowie die von mir über das Buch verteilten Hinweise auf Informationsquellen.

Inhaltsverzeichnis Stichwortverzeichnis Suchen

Stichwortverzeichnis

B

MD5-Modul

- Überblick über die Perl-Module

Methoden, Definition

- Was noch bleibt

GET

- Perl für CGI-Skripts

POST

- Perl für CGI-Skripts

selbstladende

- Was noch bleibt

mkdir-Funktion

- Dateien und Verzeichnisse verwalten
- Perl-Funktionen

Modifikatoren

- Bedingungen und Schleifen

lokale

- Gültigkeitsbereiche , Module und das Importieren von Code
- Gültigkeitsbereiche , Module und das Importieren von Code

my

- Subroutinen erstellen und verwenden

Module

- Eine Einführung in Perl

AnyDBM

- Überblick über die Perl-Module

Apache

- Überblick über die Perl-Module

AppleII

- Überblick über die Perl-Module

Archie

- Überblick über die Perl-Module

AtExit

- Überblick über die Perl-Module

Authen::Radius

- Überblick über die Perl-Module

Autoloader

- Überblick über die Perl-Module

B

- Überblick über die Perl-Module

Benchmark

- Überblick über die Perl-Module

BSD::Resource

- Überblick über die Perl-Module

BSD::Time

- Überblick über die Perl-Module

Business::CreditCard

- Überblick über die Perl-Module

Carp

- Überblick über die Perl-Module

CGI

- Überblick über die Perl-Module

CGI.pm

- Perl für CGI-Skripts

Compress

- Überblick über die Perl-Module

Config

- Überblick über die Perl-Module

Convert::BinHex

- Überblick über die Perl-Module

Convert::UU

- Überblick über die Perl-Module

CPAN

- Überblick über die Perl-Module

CPAN (Comprehensive Perl Archive Network), herunterladen

- Gültigkeitsbereiche , Module und das Importieren von Code

Crypt::Des

- Überblick über die Perl-Module

Crypt::Idea

- Überblick über die Perl-Module

Curses

- Überblick über die Perl-Module

Cwd

- Überblick über die Perl-Module

Date::DateCalc

- Überblick über die Perl-Module

Date::Format

- Überblick über die Perl-Module

Date::Language

- Überblick über die Perl-Module

Date::Manip

- Überblick über die Perl-Module

Date::Parse

- Überblick über die Perl-Module

Math::Complex
- Überblick über die Perl-Module

Math::Fraction
- Überblick über die Perl-Module

Math::Matrix
- Überblick über die Perl-Module

Math::PRSG
- Überblick über die Perl-Module

Math::Trig
- Überblick über die Perl-Module

Math::TrulyRandom
- Überblick über die Perl-Module

MD5
- Überblick über die Perl-Module

Msql
- Überblick über die Perl-Module

Net::Bind
- Überblick über die Perl-Module

Net::Cmd
- Überblick über die Perl-Module

Net::Country
- Überblick über die Perl-Module

Net::DNS
- Überblick über die Perl-Module

Net::Domain
- Überblick über die Perl-Module

Net::FTP
- Überblick über die Perl-Module

Net::Gen
- Überblick über die Perl-Module

Net::Ident
- Überblick über die Perl-Module

Net::Inet
- Überblick über die Perl-Module

Net::Netrc
- Überblick über die Perl-Module

Net::NIS
- Überblick über die Perl-Module

Net::NISPlus
- Überblick über die Perl-Module

Net::NNTP
-

Qt
- Überblick über die Perl-Module

Ref
- Überblick über die Perl-Module

Religion
- Überblick über die Perl-Module

SDBM_File
- Überblick über die Perl-Module

SNMP
- Überblick über die Perl-Module

Socket
- Überblick über die Perl-Module

Sort::Versions
- Überblick über die Perl-Module

Statistics/Descriptive
- Überblick über die Perl-Module

Sybperl
- Überblick über die Perl-Module

Symbol
- Überblick über die Perl-Module

Term::AnsiColor
- Überblick über die Perl-Module

Term::Gnuplot
- Überblick über die Perl-Module

Text::Wrap
- Gültigkeitsbereiche , Module und das Importieren von Code

Tie
- Überblick über die Perl-Module

TimeDate
- Überblick über die Perl-Module

Time-Modules
- Überblick über die Perl-Module

Tk
- Überblick über die Perl-Module

Unicode
- Überblick über die Perl-Module

Usage
- Überblick über die Perl-Module

Win32

X11::FVWM
- Überblick über die Perl-Module

X11::Protocol

N

P

Q

S

T

Inhaltsverzeichnis Stichwortverzeichnis Suchen