Inhaltsverzeichnis

Lizenz

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Vorwort

Kapitel 1

Vorwort

Tja das Vorwort wird geschrieben wenn das Buch fertig ist.....

Die Projektdefinition dieses Wikibuchs

Kapitel 2

Projektdefinition

Zusammenfassung des Projekts

Zielgruppe

Neueinsteiger sollen hier einen einfachen Einstieg in die Programmiersprache Perl finden. Fachleute haben andere Quellen als Nachschlagewerk (z.B. Perldoc, CPAN, Perlwiki, etc.)

Lernziele

Dieser Kurs ist dafür geschrieben, die ganze Welt der Perl-Programmierung in Theorie und Praxis vorzustellen.

Ein Motto von Perl lautet: ||There is more than one way to do it|| . Nach der Lektüre sollte der Leser für jede denkbare Teilaufgabe mindestens einen Weg kennen, sie durchzuführen.

Buchpatenschaft / Ansprechperson

- Verantwortliche Autoren: – Noch niemand

- Co-Autoren:

  - Giftnuss, Sebastian Knapp
  - Glauschwuffel

- Lichtkind
- MGla
- Turelion
- Ap0calypse 11:18, 16. Jul. 2007 (CEST)

Sind Co-Autoren gegenwärtig erwünscht?

Das Projekt ist bereits fortgeschritten, und verfügt bereits über Co-Autoren. Weitere lassen sich problemlos einfügen, und sind ausdrücklich erwünscht. Auf der Diskussionsseite wird eine Todo-Liste eingerichtet, so daß diese auch sofort loslegen können.

Richtlinien für Co-Autoren:

Da es in Perl durch seine Möglichkeiten der Syntaxgestaltung sehr leicht möglich ist, extrem unleserlichen Code zu produzieren, sollte eine gewisse Einheitlichkeit an den Tag gelegt werden, um es dem Leser einfacher zu machen, den Quelltext zu verstehen. Dies gilt für die Art der Klammerung, Variablendeklarationen, usw. Viele dieser Richtlinien stammen aus dem Buch ||Perl Best Practices|| von Damian Conway und gelten als Standards für guten Perl-Code. Natürlich steht es jedem frei, sich diese Regeln anzueignen, aber wenigstens im Laufe dieses Buches sollten sie verwendet werden.

Klammerung Die Richtlinie zur Klammerung innerhalb des Buches ist der K& R-Stil. Er hat gegenüber anderen Klammerungsstilen den Vorteil, dass er eine Zeile spart, dabei die Lesbarkeit der Schleife oder Anweisung jedoch nicht beeinträchtigt.

Beispiel (K& R-Stil): `for (1 .. 10) { print $_, ||\n||; }`

Im Gegensatz zu diesem Beispiel, hier ein Beispiel mit dem BSD-Stil:

Beispiel (BSD-Stil): `for (1 .. 10) { print $_, ||\n||; }` Zwar ist der Code hier mindestens gleich gut zu lesen, aber er braucht eben eine Zeile mehr. Und Zeilen sind kostbar. :) Diese Richtlinie gilt auch bei der Deklaration von Listen und Hashes.

Beispiel-Deklaration (K& R-Stil): `my @list = ( 'blubb', 'blibb', 'blebb' );`

```
my %hash = ( 'id1' => 'name1', 'id2' => 'name2', 'id3' =>
'name3', 'id4' => 'name4' );
```

Bei kurzen Listen ist dies natürlich nicht erforderlich, aber sobald eine Liste die kritische Masse erreicht hat, in der sie nicht mehr in eine Zeile passt ($>$ 80 Zeichen) sollte sie in der oben angeführten Form geschrieben werden um es dem Leser verständlicher zu machen.

Einrückungen Generell sollte für jede Stufe eine Einrückung von 4 Leerzeichen verwendet werden. Das ist der Kompromiss zwischen den 8 und den 2 Leerzeichen, die sonst oft Anwendungen finden. Damit hat man genug Übersicht um sich zurechtzufinden und muss trotzdem kein Augentennis spielen wenn man zwischen den Zeilen herumspringt.

Beispiel: `if (1 == 1) { print "1 ist 1", "\n"; }`

Operatoren und Zuweisungen Um eine Zuweisung übersichtlich zu halten, ist es zu begrüßen, wenn sie den Elementen der Zuweisung ein wenig Platz lassen. So ist folgende Zuweisung `my $var=(1+3/5)*(6-3);`

erheblich schwerer zu lesen als

```
my $var = (1 + 3 / 5) * (6 - 3);
```

Zwar benötigt man so mehr Platz, aber um die Übersichtlichkeit zu gewähren, ist das ein verschmerzliches Übel.

use strict; use warnings; Dieses Buch richtet sich an Anfänger. Gerade diesen sollte man zeigen, wie man gefährliches Perl vermeidet. Dazu zählt der Einsatz von Warnungen und 'strictures'.

Die beiden Pragmas `use strict;` und `use warnings;` sollten am Anfang eines jeden Beispiels stehen, das eigenständig lauffähig ist. Bei nicht eigenständigen Code-Fragmenten sind sie nicht unbedingt nötig, aber diese sollten durch Verwendung von "..." oder explizit durch Perl-Kommentare als unvollständig gekennzeichnet werden. Darüber hinaus sollten Code-Fragmente nur selten verwendet werden, der Schwerpunkt sollte auf vollständigen Programmen liegen.

`perl -w` ist antik und soll vermieden werden, s. perllexwarn, *What's wrong with -w and $\hat{W}$*

Formatierungen

- Bei Codestücken, die code-zeilen-nummerierung mitnehmen, da es dann leichter wird sich zurechzufinden, wenn man sich auf etwas bezieht (nur wenns längere beispiele sind, bei einzeilern ist das sinnfrei ;))
- Falls im Text Funktionen/Schlüsselwörter angesprochen werden, diese so herauszuheben: `while` -> `while` um sie leichter erkennbar zu machen
- Variablen, die im Beispiel vorkommen und im Text erläutert werden fett zu halten: **$testvar** -> **$testvar**
- Im Text erwähnte CPAN-Module bitte als Links auf die CPAN Seite formatieren.

Projektumfang und Abgrenzung zu anderen Wikibooks

Themenbeschreibung

Aufbau und Bearbeitungsstatus des Buches

Für Neugierige hier zuerst ein kurzer Überblick über den Aufbau. Er ist auch für Praktiker und Fortgeschrittene nützlich, damit sie erfahren, welche Kapitel sie überspringen können.

Die ersten Kapitel behandeln Perl theoretisch. Mit wichtigen Grundinformationen beginnend wird es immer spezifischer und zum Ende hin auch ein wenig philosophisch, so dass jeder für sich entscheiden kann, wann er sich bereit für die Praxis fühlt und den Rest dieses Kapitels überspringen möchte.

Der zweite Abschnitt ist der Grundkurs, in dem es um Variablen, Funktionen, Schleifen und etwas Spielerei mit Dateien und Texten geht.

Formales

Einführung

Wikibooks.org

Liste der verwendeten Vorlagen: Perl-Programmierung: Vorlagen

Das Wikibuch: Perl Programmierung

Kurzvorstellung

Kapitel 3

Kurzvorstellung

Perl ist eine von Larry Wall entwickelte Programmiersprache, die erstmals (Version 1.0) am 18. Dezember 1987 über den damaligen Vorläufer des Internets veröffentlicht wurde. Larry Wall entwickelte Perl damals, um sich seine Arbeit als Betreuer eines USA-weit verstreuten Computernetzes zu erleichtern. Er verband dabei Ausdrucksweisen und Fähigkeiten von Programmiersprachen wie C, BASIC, Pascal und Ada und verwendete auch bekannte UNIX-Werkzeuge wie sed, awk, shell und grep als Vorlage für Perl-Befehle. Dabei orientierte er sich weder am Ideal optischer Schönheit und Eindeutigkeit, wie es die verwandte Sprache Python tut, noch an strenger Logik und Einfachheit wie beispielsweise LISP, sondern am Reichtum menschlicher Lese- und Denkgewohnheiten. Laut Larry Wall wurde Perl dafür entworfen, um möglichst frei, individuell und schnell die eigenen Ideen umsetzen zu können. Ein Perl-Programm ist auch so wie es geschrieben wurde sofort startbereit, benötigt aber einen Interpreter, um ausgeführt zu werden. Dieser Perl-Interpreter ist für jedes gängige Betriebssystem frei erhältlich und wird weiterhin vom Erfinder und vielen Freiwilligen gepflegt und weiterentwickelt.

Besonderheiten des Sprachkonzepts von Perl

Sprachen wurden von Menschen zum Nutzen von Menschen eingeführt. Da Perl von einem (sozusagen) Gelegenheitslinguisten entworfen wurde, orientiert sich die Sprache sehr nahe an der natürlichen Sprache. Selbstverständlich gibt es hierfür verschiedenste Betrachtungswinkel. Dies zu erläutern wäre zu kompliziert, deshalb sei gesagt, dass bei Perl einfache Dinge auch einfach bleiben sollen und kompliziertes immer noch möglich sein sollte. Diese Forderung, die wir an diese Sprache stellen, ist klar und eindeutig. Allerdings scheitern genau an diesen Punkten sehr viele andere Programmiersprachen, Perl nicht.

Natürliche Sprachen ändern sich stetig. Dies liegt einerseits daran, dass wir Menschen lebendig sind und nicht wollen, dass uns etwas diktiert oder absolut aufgezwungen wird, andererseits daran, dass Menschen schon immer kreativ und einfallsreich waren. Perl wurde nun extra so entwickelt, dass es wachsen kann, und somit der ||Wortschatz|| nicht immer auf dem gleichen Stand bleibt. Und ja, es ist gewachsen. Das Kamel (Logo von Perl) ist und bleibt eigenständig und absolut durchsetzungsfähig. Zudem sagt man, dass ein Kamel nicht gerade gut riecht; Perl wurde auch nie so entwickelt, dass es ||gut riecht||.

Perl ist sogar einigermaßen intelligent; es hört zu und versucht zu begreifen, was Sie machen wollen. Ein Beispiel: Wenn ich das Wort ||dog|| sage, dann hören Sie es als ein Substantiv. Ich kann das Wort aber auch auf andere Weise verwenden. Das heißt, ein Substantiv kann auch als Verb oder Adjektiv oder Adverb fungieren, wenn es der Zusammenhang verlangt. Perl beurteilt Wörter auch aus dem Kontext heraus. Wie es das macht, werden Sie später noch feststellen können. (Wenn Sie nicht gerade Unsinn reden, wird Perl auch etwas tun, und wenn das was Sie sagen auch nur irgendwie verständlich ist, dann wird es Perl auch richtig tun). Die meisten Programiersprachen treffen Unterscheidungen dieser Art meist durch Deklaration von Variablen. Perl tut sich hier einfacher; ich brauche die Variablen nicht explizit deklarieren. Perl unterscheidet hier selbstständig.

Aber dennoch ist Perl eine künstliche Sprache. Sie hat einen bestimmten Wortschatz und eine bestimmte Syntax, die sich nicht mehr ändern wird. Außer dass hin und wieder mal etwas Neues dazukommen wird, wird sich hier nichts mehr ändern. Und die Sprache ist eindeutig; jeder auf der Welt, der Perl kann, kann auch einzelne Perl-Scripte begreifen und verstehen. Denn die Sprache wird in Indien genauso geschrieben wie in Deutschland oder Amerika.

Das Richtige für mich? - Stärken, Schwächen und Alternativen zu Perl

Kapitel 4

Das Richtige für mich

Die Wahl der Sprache Damit während des ganzen Kurses Ihre Freude an Perl anhält und Sie ehrlich unsere Begeisterung für Perl teilen können, möchten wir Ihnen einige Empfehlungen nahelegen, wann Perl wirklich eine gute Wahl ist, oder ob Sie vielleicht besser eine andere Sprache lernen sollten. Orientieren Sie sich am besten an den fettgedruckten Wörtern, die Ihre Lage am besten umschreiben. Falls Sie sich sicher sind, können Sie dieses Kapitel auch gerne überspringen.

Nach Erfahrung der Benutzer

- **Anfänger** Entgegen manchen Vorurteilen ist Perl für absolute Anfänger sehr zu empfehlen. Bereits ein einzelner Befehl ist schon ein lauffähiges Programm und ein erster Erfolg. Mit einigen wenigen Grundtechniken lässt sich viel erreichen, und der Benutzer kann selbst bestimmen, wann er welche fortgeschrittene Technik erlernt.
- **Enthusiast** Wer in der Freizeit für den Eigenbedarf und den Spaß an der Sache programmiert, ist mit Perl gut beraten. Es führt schnell zu Ergebnissen. In fast allen Bereichen gibt es viele freie, fertige Module, und man kann sich in Perl nach eigenen Vorstellungen austoben wie in kaum einer anderen Sprache. Wer seine Fenster und Dialoge mit wenigen Klicks ||zusammenschieben|| möchte, kann das auch mit dem gut unterstützten TKinter und einer professionellen IDE wie Komodo[1]. Eine populäre und sehr ausgereifte Alternative von Borland, die auf der Sprache Pascal basiert, ist auf diesem Gebiet Delphi und unter Linux Kylix. Auch hier findet man einfaches Arbeiten, freie Komponenten und viele Gleichgesinnte. Der

[1] `http://aspn.activestate.com/ASPN/Downloads/Komodo/`

Kern von Object Pascal wird aber leider schon längere Zeit nicht mehr weiterentwickelt.

- **Berufsprogrammierer** Wer Geld verdienen will, sollte etwas wie C, C++, C#, Visual Basic und Java können. Auch wenn es zunehmend Perl-Jobs gibt, sind diese noch nicht so häufig und sind oft auf den Internetseiten-Bereich beschränkt. Doch es ist abzusehen, dass Perl, aber auch Python, TCL und Ruby, wegen seiner hohen Produktivität und der eingebauten Unabhängigkeit vom Betriebssystem mehr Möglichkeiten offen stehen werden.
- **Umsteiger** Wer bereits Erfahrung mit einer Sprache wie C, Java, Basic, Pascal, oder den UNIX-Werkzeugen wie sed, awk, grep usw. hat, wird sich schnell in Perl einfinden, weil er nicht nur im gleichen Stil weiterschreiben kann, sondern auch zum großen Teil sogar die Schreibweisen der Befehle übernommen wurden.
- **Studenten** Um seinen Horizont zu erweitern, neue und frische Ideen zu sammeln, ist Perl nicht die schlechteste Wahl. Es unterstützt die meisten der heute üblichen Programmierstile und besitzt viele eigenwillige Lösungen. Die neuesten Impulse kommen aber von Sprachen wie Haskell, Cecil, Dylan, Comega, Heron und Nice. Nicht jede dieser Sprachen ist rein experimentell, aber meist gibt es dort, im Gegensatz zu Perl, wenig unterstützende Infrastruktur an Programmierwerkzeugen, Bibliotheken und Internetforen.

Nach Anwendungsgebiet

- **Webdesigner** Bei der Erstellung von größeren Internetseiten ist mittlerweile das Perl recht ähnliche PHP oder Microsofts Alternative ASP gebräuchlicher. Wenige Zeilen PHP lassen sich lesbarer in den HTML-Quellcode einbinden als Perl, andererseits bietet das wesentlich ältere Perl hier immer noch mehr Möglichkeiten für anspruchsvolle Programmierer. Zudem gibt es mittlerweile für Perl diverse Templating-Systeme, die das saubere Trennen von Programmier- und HTML-Code ermöglichen. Das mod_perl Modul des häufig verwendeten Apache-Webservers und das *Active State plex modul* im Microsoft IIS Server erlauben es, mit Perl anspruchsvolle und schnelle Webportale und sogar effiziente Vielrechner-Systeme zu erstellen.
- **Hardwaretüftler** Wer Betriebssysteme, Treiber oder sonstige Software schreiben will, die direkt mit der Hardware kommuniziert oder einfach nur sehr schnell sein soll, benutzt am besten Assembler wie NASM, MASM, TASM oder hardwarenahe Hochsprachen wie C.

- **Applikationsentwickler** Perl kann man auch in eigene Programme einbauen, um es dem Benutzer zu ermöglichen, mit Erweiterungen, sogenannten Plugins oder Extensions, Funktionen hinzuzufügen, an die man selbst nicht dachte. Dafür eignen sich aber auch andere Sprachen wie Tcl, Python oder als neuere und besonders sparsame Möglichkeit: Lua.
- **Bioinformatiker**. Im Bereich der Bioinformatik ist Perl sogar so populär, dass eigens ein Buch für diesen Teilbereich geschrieben wurde. Beginning Perl for Bioinformatics: http://www.oreilly.com/catalog/begperlbio/
- **Administratoren** Auf Unix-artigen Betriebssystemen ist Perl so weit verbreitet, dass es als Standard-Tool installiert ist. Durch die starke Unterstützung von regulären Ausdrücken eignet sich Perl ideal um Log-Dateien auszuwerten, und grafisch als Statistik oder als eMail zu versenden. Die Möglichkeit Daemons zu schreiben, Prozesse zu automatisieren, Programme zusammen zu führen, die hohe Anzahl der CPAN-Modulen, die Portierung auf fast jeder Architektur und Betriebssystem, und vor allem die bereits Vorhandene Installation von Perl machen es zum Idealen Tool eines Administrators. Einmal geschriebene Skripte können flexibel ohne Anpassung auf unterschiedlicher Hardware sowie Betriebssystemen verwendet werden.

Schlussbemerkung Die Wahl der Programmiersprache hat natürlich auch immer etwas mit persönlichen Vorlieben und dem eigenen Charakter zu tun. Wer feststellt, dass Perl in der Sache gut ist, aber sich an einzelnen Regelungen stößt, könnte Ruby ausprobieren, das Perl in vielem sehr ähnlich ist, aber wesentlich strikter an der objektorientierten Programmierung ausgerichtet ist oder Python, das den Benutzer stärker zu Übersichtlichkeit und Eindeutigkeit anleitet. Wer sich bis jetzt nicht für eine Sprache entscheiden konnte, hat auch die Möglichkeit, das allgemeine Wiki-Buch über Programmieren zu lesen.

Eintauchen in die Perlenwelt

Kapitel 5

Eintauchen

Nachdem nun auf Historie und das Perl-Umfeld, die Einordnung von Perl in die Informatik und Programmierung eingegangen wurde, sollen nochmals grundlegende Eigenschaften und Fähigkeiten stichwortartig aufgezählt und kurz in jeweils ein bis zwei Sätzen erklärt werden, bevor der eigentliche Perl-Kurs anfängt.

- Alternative Ausdrucksmöglichkeiten

  Es gibt alternative Ausdrucksmöglichkeiten gleicher Algorithmendarstellungen. Dadurch kann eine für den Entwickler geeignete leicht lesbare Form fallbasiert verwendet werden.

- Objektorientierung

  Gerade die in Perl implementierte, extrem flexible Objektorientierung und die Möglichkeit Variablen und Handler zu definieren, die einfache Möglichkeit Operatoren zu überladen usw. lassen einem sehr viele Freiheiten. Perl kann als eine objektorientierte Sprache eingesetzt werden.

- funktionale Programmierung

  In Perl sind Funktionen „Bürger erster Klasse“. Unter anderem bedeutet dies, dass Funktionen zur Laufzeit erzeugt und an andere Funktionen als Argumente übergeben werden können. Dies ermöglicht funktionale Programmierung – die Erstellung von Programmen, deren Ablauf nicht durch die Manipulation und Abfrage von Variablen gesteuert wird, sondern die durch Aufruf von Funktionen ihren Zweck erfüllen. Perl kann als eine funktionale Sprache eingesetzt werden.

- Interpreter-Ähnlichkeit, Byte-Code-Interpreter und Compiler

> Perl-Quelltexte können – ähnlich einer Interpreter-Sprache – direkt gestartet werden. Obwohl Perl die meisten der Bequemlichkeiten einer interpretierten Sprache hat, interpretiert es den Quellcode nicht streng Zeile um Zeile. Das gesamte Programm wird, wenn es aufgerufen wird, zuerst in Bytecode übersetzt (ziemlich ähnlich der Programmiersprache Java), optimiert und dann ausgeführt. Es ist möglich, die Übersetzung in Bytecode schon früher durchzuführen, um beim Programmstart Zeit zu sparen; der Interpreter wird aber immer noch benötigt, um diesen Bytecode auszuführen. Auch sind bereits Möglichkeiten geschaffen, die Perl-Quelltexte zu kompilieren, so dass direkt ausführbarer Binärcode vorliegt.

Perl ist freie Software und unter der Artistic License und der GNU General Public License erhältlich. Erhältlich für die meisten Betriebssysteme, wird Perl am häufigsten auf Unix- oder Unix-ähnlichen Systemen genutzt, während die Beliebtheit auf Microsoft Windows Systemen noch steigt. Perl hat auch heute viele Anwendungsgebiete. Ein Beispiel für die Nutzung von Perl: Die Wikipedia-Software war bis zum Januar 2001 selbst ein CGI-Skript, das in Perl geschrieben wurde. Ein weiteres Beispiel ist das bekannte Informationsportal Slashdot, welches mithilfe der Perl-basierten Slashcode-Software läuft. Im Web wird Perl oft zusammen mit dem Apache-Webserver und dessen Modul mod_perl genutzt.

Geschichte einer Skriptsprache

Kapitel 6

Geschichte

Ursprünglich waren Skripte Textdateien mit Befehlen, die man auch in die Kommandozeile (System-Shell) eingeben könnte. Diese Skripte ließen sich vergleichsweise schnell schreiben und es war damit möglich, Befehle des Betriebssystems mit Programmaufrufen zu kombinieren, um so komplexere Aufgaben dem Computer zu überlassen, ohne in langer Arbeit ein neues Programm zu schreiben. Die Kommandozeileninterpreter, die diese Skripte ausführten, hatten aber wesentlich geringere Fähigkeiten, Daten zwischenzuspeichern oder den Ablauf zu regeln als eine Programmiersprache wie C. Außerdem konnte man in Skripten die Daten nur wie durch ein Nadelöhr von einem Programm hinaus und zum nächsten Programm hinein leiten.

Larry Walls Reaktion auf diese Situation war, einen erweiterten Kommandozeileninterpreter zu schreiben, der nicht nur fast den gesamten Sprachschatz von C beherrscht, sondern auch viele der kleinen nützlichen Helferlein kopiert, besonders sed, awk und grep, die die Kommandozeile des Betriebssystems UNIX so mächtig machen. Damit lassen sich viel mächtigere Skripte schreiben, die Textdateien durchsuchen und die von anderen Programmen ausgespuckten Ergebnisse sortieren und aufbereiten können. Aus diesem Grund nannte man PERL auch eine: **'P**ractical **E**xtraction and **R**eport **L**anguage' (praktische Auszugs- und Berichtsprache), was aber nur ein Backronym ist, weil die ursprüngliche Bedeutung des Namens Perl eine andere ist (siehe nächstes Kapitel). Ein anderes Backronym, das von Larry ebenso für „offiziell“ erklärt wurde, ist: **'P**athologically **E**clectic **R**ubbish **L**ister' (krankhaft ausufernder Auflister von Blödsinn), denn mit Perl kann man wirklich sehr sehr unverständliche Programme verfassen. Warum das so ist, erklär' ich auch im nächsten Kapitel.

Mit der Zeit lernte Perl auch mit binären Daten umzugehen und viele weitere nützliche Dinge, die überall abgeschaut wurden. Besonders mit seiner fünften Version veränderte sich Perl zu einer „richtigen“ Programmiersprache, als nicht nur die objektorientierte Programmierung eingeführt wurde, sondern auch die diesem Konzept zugrunde liegenden Module. Diese Module, die Perl meist um eine wertvolle Fähigkeit ergänzen, zum Beispiel graphische Ausgabe, Herunterladen von Webseiten oder Steuern von Robotern, findet man in großer Anzahl und in oft sehr guter Qualität in dem zentralen Archiv namens CPAN[1]. Kritiker und Anhänger von Perl sind sich zumindest darin einig, dass der Erfolg von Perl heute zu einem bedeutenden Teil am CPAN liegt, mit dessen Hilfe man auch anspruchsvollere Aufgaben in nur wenigen Zeilen Quellcode lösen kann, indem man mehrere fertige Module kombiniert.

Das entspricht eigentlich immer noch dem ursprünglichen Prinzip des „Skriptens“, obwohl sich dieser Begriff erweitert hat auf eine Bedeutung, die man etwa mit kürzeres, schnelleres, komprimiertes Programmieren übersetzen kann. Neben Perl enstanden nämlich noch seit den 80er Jahren eine Reihe weiterer „Skriptsprachen“ wie Ruby, Python, PHP, Tcl, Visual Basic Script, die dem Programmierer einiges an Arbeit abnehmen oder vereinfachen. Manche Autoren sehen darin eine „nächste“ oder „höhere“ Generation an Computersprachen, die sich noch näher an den Bedürfnissen der Programmierer orientieren und noch weniger an den konkreten Fähigkeiten einer Maschine.

Auch eine wichtige Gemeinsamkeit dieser Skriptsprachen ist, dass es Interpretersprachen sind. Das sind Sprachen, bei denen das Programm immer noch einen Interpreter benötigt, der den Quellcode ausführt. Ein Vorteil daran ist, dass man keinen weiteren Aufwand treibt, um ein lauffähiges Programm zu bekommen. Nur den Quellcode schreiben und das Programm ist fertig. Es läuft sogar meist unverändert auch unter anderen Betriebssystemen. Man braucht dafür nur einen neuen, unter diesem Betriebssystem lauffähigen Interpreter. Dieser Vorteil wurde aber lange Zeit wenig genutzt, weil diese Interpreter zu langsam waren, was aber mit den hohen Geschwindigkeiten neuerer Hardware ausgeglichen wird. Außerdem erkannte man mittlerweile, dass interpretierte Programme viel besser optimiert werden können als kompilierte, weil zum Zeitpunkt der Ausführung sehr viel mehr über den aktuellen Computer bekannt ist als bei der Programmierung.

Larry und die Perl-Kultur

[1] `http://www.cpan.org`

Kapitel 7

Larry und die Perl-Kultur

Was ist perlish?

Neben der ganzen Logik und der Technik übersehe bitte niemand, dass Programmieren für manche Menschen Begeisterung, Leidenschaft und Berufung ist. Nicht wenige davon halten Perl für eine Lebensphilosophie, eine Art zu denken und zu handeln, die man auf alles übertragen kann. Nach dieser Einstellung wird „perlish“ genannt, was für bequem, cool, eigenwillig, treffend oder clever gehalten wird. Larry Wall hat von Anfang an mit witzigen und spitzfindigen Bemerkungen, Vorträgen und Büchern darüber referiert, wofür Perl steht und was perlish ist. Dabei prägte er die folgenden Abkürzungen und Slogans, welche die Perl-Philosophie wiedergeben sollen.

Leitsprüche

- **TIMTOWTDI** wird auch timtoday ausgesprochen, steht aber für || **T**here **I**s **M**ore **T**han **O**ne **W**ay **T**o **D**o **I**t||, zu deutsch: „Es gibt mehrere Wege, etwas zu tun.“ Damit ist gesagt, dass es in Perl mit Absicht für jede Aufgabe viele Wege gibt, sie zu lösen, und dass die Sprache es dem Programmierer überlässt, welchen Weg er wählt. Perl wird von Larry Wall als bescheidener Diener gesehen, der möglichst vielen unterschiedlichen Menschen behilflich sein will, ohne ihre Freiheit einzuschränken.
- „Make easy jobs simple and the hard possible.“ deutsch: „Lass die einfachen Aufgaben einfach und mach die schweren Aufgaben lösbar.“ Darin stecken eigentlich 3 Aussagen.

1. Für einfache Dinge braucht man in Perl nichts anzumelden oder vorzubereiten. Nur die einfache Anweisung und Perl tut, was es soll.
2. Perl will die Möglichkeit bieten, den Quellcode großer und komplexer Programme beherrschbar zu gestalten.
3. Es wird versucht, beide Prinzipien zu harmonisieren, damit beides parallel mit einer Sprache erreicht wird.

- Eine Programmiersprache kann, wie eine natürliche Sprache auch, verschieden interpretierbar sein. Für eine Programmiersprache gilt aber, dass sie keine Mehrdeutigkeit erlaubt. Es muss also eine eindeutige Implementierung geben. Daher wurde bei der Umsetzung der Sprache dieses Prinzip verfolgt: **Perl verhält sich so, wie es der Programmierer am ehesten erwartet.**
- „Postmodernism“: Larry Wall nennt Perl auch die erste postmoderne Programmiersprache. Dahinter steht der Gedanke von der Befreiung von Dogmen, die oftmals eine Ära kennzeichnen. Techniken wie z.B. objektorientierte Programmierung sollten nämlich nicht zu Dogmen überhöht werden, die blind befolgt werden, weil es gerade schick ist. Jeder sollte das verwenden, was für ihn am nützlichsten ist oder was ihm am meisten Freude bereitet.

Es geht bei Perl also sehr um die Freiheit des Programmierers.

Der Einstieg

„Hello World!“

Kapitel 8

Hello World

Wer das Pech hat, vor einem Rechner ohne Perl zu sitzen, und niemanden gefunden hat, der ihm das abnehmen kann, sollte einen Blick in unsere Installationsanleitung werfen.

Traditionsgemäß ist das erste Programm, das man in einer neuen Programmiersprache schreibt, eines, das 'Hello World!' ausgibt.

Hello World

hello.pl: `print "Hello World!\n";`

Das typische Einstiegsbeispiel einer Programmiersprache, das Ausgeben von "Hello World!" auf dem Bildschirm, beschränkt sich bei Perl auf eine einzige Zeile. Sie enthält die Anweisung, einen Text auszugeben und den Text selbst, gefolgt von einem Semikolon, welches das Ende der Anweisung anzeigt. Der Text ist dabei in doppelte Anführungszeichen (") eingeschlossen. Dadurch werden bestimmte Zeichenfolgen durch Steuerzeichen ersetzt. In diesem Fall wird "Hello World!" als Text interpretiert, der eins zu eins auf dem Bildschirm ausgegeben wird. Das `\n` ist das Zeichen für einen Zeilenumbruch.

Auf unixartigen Systemen gibt es zwei Konzepte, das Programm ablaufen zu lassen.

- Am einfachsten ist es, den Perl-Interpreter mit einem Kommando aufzurufen, und ihm die Quellcode-Datei als Parameter zu übergeben:

```
$ perl hello.pl
```

- Am häufigsten wird meist zusätzlich zu Beginn des Quelltextes die sogenannte shebang-Zeile eingegeben. Sie gibt an, mit welchem Interpreter-Programm der nachfolgende Quellcode ausgeführt werden soll. Bei einer Standardinstallation liegt der `perl`-Interpreter bei Unix-Systemen im Verzeichnis /usr/bin, die Shebang-Zeile lautet also

```
#!/usr/bin/perl
```

Sollte sich der Perl-Interpreter woanders befinden, läßt er sich mit dem Kommando ‖which perl‖ lokalisieren.

Damit haben wir nun folgenden Code:

hello.pl: `#!/usr/bin/perl print ‖Hello World!\n‖;`

Das reicht aber noch nicht: Zusätzlich müssen wir die Quellcodedatei mit ‖chmod +x hello.pl‖ ausführbar machen. Das erlaubt uns den Programmstart mit:

$ ./hello.pl

Unter Windows hat die shebang-Zeile keine Funktion, schadet jedoch auch nicht. Unter Windows sollte unsere Komandozeile daher entsprechend anders aussehen:

C:\Perl\scripts> perl hello.pl

- Ändern Sie das Programm so ab, dass es nicht mehr die Welt, sondern Sie begrüsst.
- Ändern Sie das Programm so ab, dass es Sie nach der Begrüßung in der nächsten Zeile fragt, wie es Ihnen geht. Die Ausgabe sollte in etwa so aussehen:

Hallo Ihr_Name! Wie geht es Ihnen?

- Spielen Sie ein wenig mit dem Sonderzeichen \n herum. Was passiert, wenn Sie es weglassen oder verdoppeln?

Hinweis: Denken Sie an das Semikolon, wenn Sie eine neue print-Anweisung einfügen.

Dokumentation

Perl ist mit einer Fülle von Dokumentation ausgestattet. Diese ist über viele Handbuchseiten (engl. *manual pages*, kurz *man pages*) verstreut. Eine Übersicht über die vorhandenen Handbuchseiten kann man sich mit man perl anzeigen lassen.

man ist ein Unix-Befehl. Wenn Sie kein Unix oder ein ähnliches System wie Linux und BSD einsetzen, dann ist dieser Befehl auf Ihrem System möglicherweise nicht verfügbar.

Die Perl-Dokumentation ist auch im Web unter perldoc.perl.org erhältlich.

In diesem Wikibuch werden Sie an verschiedenen Stelle dazu aufgefordert, Informationen auf einer Handbuchseite nachzuschlagen. Damit ist dann stets gemeint, die betreffende Seite mit man oder im Web unter perldoc.perl.org zu betrachten.

Die in Perl eingebauten Funktionen sind auf der Handbuchseite perlfunc erläutert. Diese Handbuchseite ist lang und in der Regel müssen Sie etwas blättern, bis Sie die betreffende Funktion gefunden haben. Daher gibt es dafür ein Abkürzung: perldoc -f *Funktionsname* zeigt Ihnen die Dokumentation zu genau dieser einen Funktion an.

- Schlagen Sie die Handbuchseite zu man nach.
- Lesen Sie nach, wie die Funktion print funktioniert. (Sie müssen noch nicht alles verstehen.) **Hinweis:** Denken Sie daran, dass Sie Dokumentation zu einer Funktion nicht nur auf eine Art nachschlagen können. (TIMTOWTDI).
- Schlagen Sie mit perldoc im Katalog der häufig gestellten Fragen nach, welche Perl-**Version** Sie einsetzen sollten. Gibt es eine eindeutige Antwort? Welche Version setzen Sie ein?

Guter Stil

In Perl ist es sehr einfach, sehr kryptisch anmutenden Code zu schreiben, da praktisch alle Sonderzeichen mit Bedeutungen belegt sind und die Sprache Perl selbst viele Abkürzungen und implizites Verhalten bereit stellt.

Es wird im Allgemeinen als guter Stil angesehen, nicht alle Möglichkeiten in Perl auszureizen, da sich sonst leicht Fehler einschleichen und unwartbarer Code entsteht.

Einige sprachliche Möglichkeiten, die sich über die Jahre als problematisch erwiesen haben, kann man mit der Anweisung `use strict;` unterdrücken. Der Perlcompiler gibt dann bei der Compilierung eine entsprechende Warnung aus.

Eine weitere Anweisung hilft bei der Entwicklung „sauberen“ Codes. Die Perlsyntax erlaubt Ausdrücke, deren Auswertung nicht immer das bewirkt, was der Programmierer erwünscht. Perl selbst „kennt“ einige dieser Ausdrücke und kann sich selbst auffordern, bei Verwendung dieser Ausdrücke Warnungen auszugeben. Beispiel für einen solche Ausdruck ist die Addition einer Zahl und einer Zeichenkette. Obwohl dies von der Syntax her gültiges Perl ist, kann über die Sinnhaftigkeit zumindest gestritten werden. Die Anweisung `use warnings;` führt zur Ausgabe der Warnungen.

Die beiden angesprochenen `use`-Anweisungen sollten zu Beginn eines jeden Perlskripts stehen, da sie nicht nur bei der Fehlersuche helfen, sondern den Programmierer auch dabei unterstützden, Fehler zu vermeiden.

Um diese beiden Anweisungen erweitert sieht das erste Perlskript dann wie folgt aus:

hello.pl: `#!/usr/bin/perl`

```
use strict; use warnings;

print "Hello World!\n";
```

{{Vorlage:Referenzbox IntraBuch|Guter Stil (in Stil und Struktur) }}

Variablen

Kapitel 9

Variablen

Perl-Variablen

Unter einer Variable versteht man ganz allgemein einen Bezeichner f& uuml;r Daten. Fr& uuml;her konnte man davon sprechen, dass ein Bereich des Hauptspeichers mit dem Variablennamen identifiziert wurde. Im Gegensatz dazu existiert bei Perl die M& ouml;glichkeit auch eine Datei, ein Feld in einer Exceltabelle, eine Webseite auf einem Computer am anderen Ende des Globus und viele andere Vorkommen von Daten wie eine Variable zu behandeln. Im Normalfall ist aber auch in Perl eine Variable ein Bezeichner f& uuml;r Daten, die im Hauptspeicher des Rechners liegen. Perl & uuml;bernimmt dabei die komplette Speicherverwaltung. Es verwendet ein f& uuml;r die Programmierer sehr bequemes System mit einem Zähler f& uuml;r jeden gespeicherten Wert. Mit diesem Zähler kann Perl feststellen, ob in dem Programm noch ein Verweis auf diesen Wert existiert. Variablen k& ouml;nnen in Perl beliebig gro& szlig; sein. Von undef, dem Standardwert von Perl f& uuml;r eine Variable ohne Inhalt und den vielen Gigabyte Speicher, welche das jeweilige System Perl zur Verf& uuml;gung stellen kann.

Skalare Variablen

Skalare Variablen, leicht erkennbar durch das vorangestellte Dollar-Zeichen ($), können fast jede Art von Daten speichern. Ihnen kann ein Wert mithilfe des '=' zugewiesen werden. In Skalaren werden auch Referenzen (Verweise auf andere Variablen; vergleichbar mit Zeigern in anderen Programmiersprachen) und Objekte gespeichert.

Zahlen `$var = 50; #ganze Zahl $var = 0.01; #Fließkommazahl $var = 1e2; #wissenschaftliche Notation - dasselbe wie 100 $var = 1e-2; #wie 0.01`

In Perl als Skriptsprache muss, anders als in anderen, nicht-interpretierten Sprachen wie z. B. C, nicht angegeben werden, ob eine Variable eine ganze Zahl, Fließkommazahl, reelle Zahl, oder anderes enthält. Perl übernimmt die Bestimmung des Datentyps für uns, abhängig von dem jeweiligen Umfeld, in dem die Variable verwendet wird.

Beispiel

```
#!/usr/bin/perl

use strict; use warnings;

my $a = 2; my $b = 3; print "$a + $b = ", $a + $b, "\n";
```

Die Ausgabe des Programmes:

2 + 3 = 5

Erklärung: In den ersten beiden Zeilen werden den skalaren Variablen **$a** und **$b** die Werte **2** bzw. **3** zugeordnet. Die dritte Zeile beginnt mit einer in " eingeschlossene Zeichenkette. Die Variablen werden innerhalb der Zeichenkette als String ersetzt. Hinter der Zeichenkette werden **$a** und **$b** mit dem mathematischen Operator **+** verknüpft. Sie werden also als Zahlen interpretiert und das Ergebnis wird ausgegeben. Abgeschlossen wird die Ausgabe durch einen Zeilenumbruch.

Perl stellt uns eine Vielzahl mathematischer Operatoren zur Verfügung, neben der hier verwendeten Addition ist uns u.a. auch Subtraktion, Multiplikation, Division und Potenzieren möglich (siehe Operatoren).

Zeichenketten (Strings) `$var = 'text'; $var = "text";`

Skalarvariablen können auch Zeichenketten (Strings) enthalten. Diese werden, von einfachen (') oder doppelten (") Anführungszeichen umschlossen, der Variablen zugewiesen. Der Unterschied zwischen einfachen und doppelten Anführungszeichen besteht in der sogenannten *Variableninterpolation*. Dazu ein Beispiel:

```
#!/usr/bin/perl

use strict; use warnings;
```

```
my $var = 5; my $text = "Variable enthält: $var"; my $text2
= 'Variable enthält: $var'; print $text; print "\n"; print
$text2; print "\n";
```

erzeugt folgende Ausgabe:

Variable enthält: 5 Variable enthält: $var

Was ist passiert? Bei der Zuweisung an **$text**, bei der wir doppelte Anführungszeichen benutzt haben, sucht der Interpreter in der zugewiesenen Zeichenketten nach Vorkommen von **$Variablenname**, und ersetzt diese durch den Inhalt der jeweiligen Variable – in unserem Fall also durch den Inhalt von **$var**. Dies nennt man *Interpolation*. Eine solche Interpolation wird jedoch bei einfachen Anführungszeichen nicht vorgenommen.

Nicht nur Variablennamen, sondern auch Steuercodes wie `\n` werden bei einfachen Anführungszeichen ignoriert. Beispielsweise erzeugt

```
print "\n";
```

einen Zeilenumbruch, wohingegen

```
print '\n';
```

einfach die Zeichenfolge \n ausgibt.

Beispiel Nehmen wir also nochmal das Additions-Beispiel von vorhin und verschönern es ein wenig.

```
#!/usr/bin/perl

use strict; use warnings;

my $a = 2; my $b = 3; my $c = $a + $b;

print "$a + $b = $c\n";
```

Ausgabe:

2 + 3 = 5

Arrays

Ein Array enthält im Gegensatz zu einem Skalar nicht einen, sondern mehrere Werte. Dabei können diese Werte völlig frei gemischt werden. Es ist also möglich in dem selben Array Texte, Ganzzahlen, Kommazahlen u.ä. zu speichern. Zur

Unterscheidung zwischen Arrays und Skalaren ist dem Array ein At-Zeichen (@) vorangestellt. Man kann sich ein Array als Liste von Skalaren vorstellen, dabei werden die einzelnen Werte über ganze Zahlen (Integer) eindeutig identifiziert. Nachfolgend ein kleines Beispiel:

```
#!/usr/bin/perl

use strict; use warnings;

my @halblinge = ( "Bilbo", "Frodo", "Sam" ); # einfache Liste
von Halblingen print $halblinge[0], "\n"; # gibt Bilbo aus
print $halblinge[1], "\n"; # gibt Frodo aus
```

Was geschieht hier? Nun, zuerst wird ein Array mit dem Namen **@halblinge** erzeugt. Diesem wird eine Liste zugewiesen. Listen werden in Perl in einfache Klammern eingeschlossen. Da die Werte in der Liste Strings (Text) sein sollen, müssen sie als solche kenntlich gemacht werden. Dies kann man u.a. mit doppelten Anführungszeichen tun. Als letztes müssen die Werte einer Liste dann noch mit Kommata getrennt werden. Bei Listen von Strings ist es mithilfe des Listenoperators `qw` möglich, auf Kommata und doppelte Anführungszeichen zu verzichten. Die einzelnen Strings werden dann nur durch Leerzeichen getrennt. Mit dem `qw`-Operator (**q**uoted **w**ords / *Wörter in Anführungszeichen*) sieht das obige Beispiel so aus:

```
#!/usr/bin/perl

use strict; use warnings;

my @halblinge = qw( Bilbo Frodo Sam ); # einfache Liste von
Halblingen print $halblinge[0], "\n"; # gibt Bilbo aus print
$halblinge[1], "\n"; # gibt Frodo aus
```

Man kann Werte in einer Liste auch mit => trennen, dies werden wir bei den Hashes sehen.

Aber was soll das nun mit der zweiten und dritten Zeile? Da steht ja nun wieder ein Dollar-Zeichen vor dem Variablennamen. Das ist auch richtig so, denn aus dem Array soll nur ein einzelner skalarer Wert ausgegeben werden. Den Variablennamen mit einem vorgestellten @ an print zu übergeben, würde `print` aber dazu veranlassen alle Werte des Arrays auszugeben.

Perl unterscheidet hier Arrays von Skalaren, indem es an das Ende der Variable schaut. Dort ist nämlich der Platz des Wertes, auf den zugegriffen werden soll, in eckige Klammern geschrieben. Hierbei ist zu beachten, dass Arrays immer bei Null anfangen zu zählen und nicht bei Eins, wie man evtl. annehmen könnte.

Die Anzahl der Elemente eines Arrays erhält man, wenn man das Array im skalaren Kontext interpretiert:

```
my $anzahlHalblinge=@halblinge; print $anzahlHalblinge; #
ergibt 3
```

Man könnte bei diesem Beispiel den skalaren Kontext erzwingen indem man die interne `scalar`-Funktion benutzt. Somit könnte das vorhergehende Beispiel auch so aussehen:

```
print scalar @halblinge; # ergibt 3 ( Klammern sind optional,
timtowtdi! )
```

Hashes (assoziative Arrays)

Hashes sind ganz ähnlich wie Arrays Listen von Werten. Aber im Unterschied zu Arrays sind Hashes nicht geradlinig durchnummeriert, sondern jeder Wert bekommt seinen eigenen Namen (*key*) der frei gewählt werden darf, weshalb sie auch manchmal assoziative Arrays genannt werden. Dadurch lassen sich einige Programmieraufgaben wesentlich schneller und schöner lösen, als es mit normalen Arrays möglich wäre.

Damit Hashes von Arrays und Skalaren unterschieden werden können, wird dem Hash ein Prozentzeichen (**%**) vorangestellt. Beachte, dass auch bei Hashes dieses Prozentzeichen in ein Dollarzeichen gewandelt wird, wenn auf einzelne, skalare Werte des Hashes zugegriffen wird.

Eine weitere Eigenheit von Hashes ist es, dass der Key in geschweiften Klammern, und nicht wie bei Arrays in eckigen Klammern eingeschlossen wird.

```
%telefon = ('MaxMuster' => '0815/12345', 'KarlMaier' =>
'0815/12346', 'HansMueller' => '0815/12347' ); print
$telefon{'KarlMaier'}, "\n"; # gibt 0815/12346 aus!
```

Um in diesem Beispiel auf alle Keys (in dem Fall die Namen) zugreifen zu können, kann man sich an der `keys`-Funktion von Perl bedienen. Diese Funktion liefert eine Liste der Keys aus dem Hash zurück.

```
my @list_keys = keys %telefon; for (@list_keys) { print "$_",
"\n"; }
```

oder kompakter:

```
for (keys %telefon) { # Variablen sparen print "$_", "\n"; }
```

Dieses Beispiel liefert folgenden Output:

MaxMuster KarlMaier HansMueller

Natürlich muss es hier auch eine Möglichkeit geben, die Werte der Identifier im Listenkontext auszugeben. Das Gegenstück zur `keys`-Funktion ist die `values`-Funktion. Sie funktioniert genau nach demselben Prinzip.

```
my @list_values = values %telefon; for (@list_values) { print "$_", "\n"; }
```

oder kompakter:

```
for (values %telefon) { # Variablen sparen print "$_", "\n"; }
```

liefert:

0815/12345 0815/12346 0815/12347

WARNUNG:

Die Reihenfolge in denen die Wertpaare in einem Hash angeordnet sind, ist nicht immer dieselbe. Nur weil ein Hash bei einem Durchlauf die Keys oder Values in einer bestimmten Reihenfolge ausspuckt, bedeutet das nicht zwangsläufig, dass das beim nächsten Durchlauf auch noch so sein muss. Zur Sicherheit sollte man hier von der `sort`-Funktion Gebrauch machen.

Beispiel:

```
for (sort keys %telefon) { # Sortierte Liste der Keys print "$_", "\n"; }
```
Hier ist gewährleistet, dass die Liste, sofern der Hash nicht verändert wurde, immer dieselbe bleibt.

Variablenklassifizierung und Fallbeispiele

Kapitel 10

Variablenklassifizierung und Fallbeispiele

Klassifizierung von Variablen

Eine Möglichkeit der Klassifizierung von Variablen basiert auf der Art der enthaltenen Daten. Die erste Art der Unterscheidungen ist Ein- und Mehrzahl, bzw. Skalar und Array. wobei Listen von Strings und Zahlen in der Mehrzahl vorliegen. Wir nennen einen singuläre Variable einen *Skalar* und eine plurare Variable einen *Array*. Weil ein String in einer skalaren Variable gespeichert werden kann, kann das *Hello World*-Beispiel auch so aussehen:

```
$ausg = "Guten Morgen, Welt!\n"; # Variable setzen print $ausg;
# Variablen ausgeben
```

Wenn Sie aufgepasst haben, werden Sie feststellen, dass die Variablen davor nicht deklariert werden mussten. Perl erkennt automatisch durch das $-Zeichen, dass es sich bei *ausg* um eine skalare Variable handelt, also dass diese Variable nur einen Wert enthält. Eine Array-Variable beginnt stattdessen mit einem @-Zeichen. (Wenn Sie nun Array und Skalar gar nicht auseinander halten können, versuchen Sie mal folgender Eselsbrücke: $ = $(s)kalar @ = @(a)rray Man kann sich hier sehr einfach das $ als s vorstellen und das @ als a.)

Perl besitzt aber noch mehr Variablentypen, wie z.B. *Hash*, *handle*, *typeglob*, und alle besitzen zur Unterscheidung auch so ein lustiges kleines Zeichen davor. Hier eine kleine Liste aller lustigen Zeichen, mit denen Sie sich noch herumschlagen dürfen.

Typ	Zeichen	Beispiel	Ist ein Name für:

Skalar	$	$bsp	Ein individueller Wert (Zahl oder String)
Array	@	@test	Eine Liste von Werten mit einer Ganzahl als Index
Hash	%	%zinsen	Ein Gruppe von Werten mit einem String als Index
Subroutine	&	& was	Ein aufrufbares Stück Perl-Code
Typeglob	*	*maus	Alles namens maus

Aber jetzt nicht zurückschrecken. Die Variablen können ohne weiteres und ohne besondere Syntax einfach umgewandelt werden. Perl-Scripte sind verhältnismäßig einfach zu lesen, weil Hash gegen Array etc. schnell heraussticht.

Singularitäten

So wie Sie vorher gesehen haben, können Sie Skalaren mit dem Operator einen neuen Wert zuweisen. Skalaren kann jede Form von skalaren Werten zugewiesen werden.

- Integer *(ganze Zahlen)*
- Fließkommazahlen
- Strings *(Zeichenfolgen)*
- Referenzen auf andere Variablen / Objekte
- etc.

Wie bei der Shell bei Unix, können Sie verschiedene Quoting-Mechanismen verwenden, um Werte zu erzeugen. hier einige Beispiele

```
$ausg = 56; # ein Integerwert $pi = 3.14159265; # eine
||Realzahl|| $chem = 93.5ds2; # wissenschaftliche Notation
$tier = ||hund||; # String $test = ||ich habe einen $hund||; #
String mit Interpolation $preis = 'kosten 255€'; # String ohne
```

```
Interpolation $abc = $def; # Wert einer anderen Variable $bild
= $rahmen * $wand; # ein Ausdruck $exit = system(‖vi $file‖);
# numerischer Status eines Befehls $cwd = `cwd`; # Textausgabe
eines Befehls
```

Nicht verzweifeln. Das sieht komplizierter aus, als es wirklich ist. Hiermit wollte ich nur verdeutlichen, dass ein Skalar sehr viel enthalten kann. Aber es kann noch mehr, wie z.B. Referenzen auf andere Datenstrukturen, Subroutinen und Objekte.

```
$ary = \@array; # Referenz auf ein bekanntes Array $hsh =
\%hash; # Referenz auf einen bekannten Hash $sub = \& was; #
Referenz auf eine bekannte Subroutine
```

```
$ary = [1,2,3,4,5]; # Referenz auf ein nicht benanntes Array
$hsh = {AP => 20, HP =>2}; # Referenz auf einen nicht
benannten Hash $sub = sub { print $zeit }; # Referenz auf eine
nicht benannte Subroutine
```

```
$fido = neuer Hund ‖Boss‖; # Referenz auf ein Objekt
```

... ...

Spezialvariablen

Kapitel 11

Spezialvariablen

Die Default-Variable $_/ @_

Neben den ganz normalen Variablen (wie sie auch jede andere Programmiersprache kennt) besitzt Perl eine ganz Besondere: `$_`

Diese Variable wird immer dann genutzt, wenn der/die ProgrammiererIn das Ergebnis einer Funktion/Berechnung/etc. in keiner eigenen Variablen speichern möchte oder kann, wie es bei den Postfix-Schleifen der Fall ist. Analog steht im Array-Kontext das Spezial-Array `@_` zur Verfügung.

```
sub addition { my ($a); { local $_; $a += $_for @_; } $_=$a
unless defined wantarray; return $a }

$a = addition(1,1); # $a = 2 addition(1,1); # $_= 2
```

Weiterhin benutzen einige in perl eingebaute Funktionen diese Variable, wenn sonst keine Argumente vorhanden sind. Ein gutes Beispiel dafür ist die print Funktion:

`$a = addition(1,1); print $a;` ist das gleiche wie : `addition(1,1); print;`

– DENKPAUSE –

OK. Das kleine Beispiel ist noch klein (und unnütz- man hätte auch ||print addition(1,1)|| schreiben können). Richtig Spaß macht die Variable $_aber erst, wenn man damit längere Quelltexte abkürzen kann.

z.B.: `while () { s/BöserText/GuterText/; s/Lisp/Perl/; # :-) print; }`

1. Zeilenweise einlesen von STDIN (meistens Tastatur) und speichern der gelesenen Zeile in $_
2. Ein Suchen-Ersetzten-RegEx : Ersetzt 'BöserText' durch 'GuterText' IN DER VARIABLEN $_;
3. Ein weiterer Suchen-Ersetzen-Regex. Merke : Perl nimmt $_wenn keine andere Variable gesetzt ist
4. Ausgabe von $_(welches nun nurnoch ||GutenText|| und kein ||Lisp|| mehr enthalten sollte)
5. Hole die nächste Zeile

Das gleiche Programm könnte man auch so schreiben : `while ( $line = ) { $line =~ s/BöserText/GuterText/; $line =~ s/Lisp/Perl/; # :-| print $line; }`

Über das Für und Wider dieser Spezialvariable kann man sich sicher streiten, unbestreitbar ist dessen Nutzung in diversen Beispielen im Internet. Mit etwas Erfahrung und dem Wissen um die Eigenschaften dieser Variable kann durch sie die Lesbarkeit des Quellcodes verbessert werden.

Gerade zu Begin der eigenen Perl-Karriere kann die Nutzung von $_eventuell hinderlich sein. Auch in größeren Projekten kann man sich durch übermäßige Nutzung dieses Features leider ab und an in die eigenen Füße schießen. Trotz aller dieser Widrigkeiten kann man durch $_kleine Scripte noch kleiner machen und dadurch die ||Produktionsgeschwindigkeit|| erhöhen.

Auf jeden Fall muss man beim Programmieren beachten, dass $_global ist. Wenn $_innerhalb einer Funktion geändert wird ändert sie auch den Wert außerhalb der Funktion. Um sich davor zu schützen kann man die Variable für den Bereich in dem sie geändert wird lokal machen.

Eigene Funktionen ändern $_nicht automatisch, wenn bei ihrem Aufruf kein lvalue angegeben ist, der den Wert aufnimmt. Perl kennt aber die eingebaute Funktion wantarray die beim Thema Kontext von Funktionen eine Rolle spielt. Ist ihr Rückgabewert undef dann wurde die Funktion im sogenannten void-Kontext aufgerufen und die Programmiererin könnte $_explizit setzen um das Verhalten der Funktion mehr perlish zu gestalten.

ARGV

Zur Übergabe von **Kommandozeilenparametern** dient das Spezialarray @ARGV. Es kann, was das Auslesen und (Über-)Beschreiben seiner Werte betrifft genau wie ein handelsübliches Array behandelt werden.

Einfache Ein-/Ausgabe

Kapitel 12

Ein-/Ausgabe

Ein-/Ausgabe

Nun verfeinern wir das 'Hello World!'-Programm aus den ersten Schritten etwas. Dieses Mal soll das Programm den Benutzer dynamisch ansprechen.

```
#!/usr/bin/perl use strict; use warnings;

# Zur Eingabe auffordern print ||Bitte geben Sie Ihren Namen
ein:\n||;

# Eine Zeile einlesen my $name = ;

# Zeilenumbruch entfernen chomp $name;

# Gruß ausgeben print ||Hallo, $name, einen schönen Tag
noch!\n||;
```

Die erste Anweisung fordert mit ||Bitte geben Sie Ihren Namen ein:|| den Nutzer zur Eingabe auf. Die zweite Anweisung liest eine Zeile ein und legt sie in der Variablen $name ab. Dies ist eine *skalare* Variable, die genau einen Wert aufnehmen kann, dieser eine Wert ist die gesamte eingelesene Zeile. Mit der dritten Anweisung werden abschließende Zeilenumbrüche von der Eingabe entfernt – wenn die Eingabe von der Tastatur stammt, so ist dies der Zeilenumbruch vom Drücken der Taste *Return*, mit der man die Eingabe abgeschlossen hat. Die vierte Anweisung gibt einen Gruß aus, nachdem der Wert der Variablen interpoliert wurde.

STDIN ist eine Kurzform für *standard input*, der Standard-Eingabekanal. Dieser Eingabekanal ist für ein von der Kommandozeile gestartetes Programm die Tastatur. Das Lesen von diesem Kanal ist so häufig, dass Perl dafür eine Abkürzung

vorsieht: Statt kann man <> schreiben. Aufgrund seiner Form wird dieser Operator der *Diamantoperator* genannt.

Es gibt neben STDIN noch STDOUT, den Standardausgabekanal, und STDERR, den Standardkanal für Fehlermeldungen. Ein- und Ausgaben kommen immer aus einem Kanal beziehungsweise fließen immer dahin.

Der Standard-Eingabekanal muss nicht immer auf die Tastatur zeigen, sondern kann auch umgelenkt werden.

Folgendes Kommandozeilen-Beispiel gibt den Namen Hans aus und leitet die Ausgabe so, dass sie zur Eingabe für obiges Programm, das hier eingabe.pl heißt, wird. echo Hans | ./eingabe.pl

Folgendes Kommandozeilen-Beispiel gibt den Namen Hans in eine Textdatei aus. Diese Textdatei wird dann zur Eingabe für obiges Programm, das hier eingabe.pl heißt. echo Hans > name.txt ./eingabe.pl < name.txt

Die beiden Beispiele sind nicht perlspezifisch, sondern abhängig von dem Betriebssystem, und funktionieren auf den gängigen Befehlszeilen vieler UNIX-Varianten.

Das zweite Beispiel zur Eingabeumlenkung liest die Zeile aus einer Datei. Diese Umlenkung kann man auch mit Bordmitteln von Perl erreichen. Hierzu öffnen wir die Datei name.txt selbst und lesen daraus.

```
#!/usr/bin/perl use strict; use warnings;

# Kanal schließen close (STDIN);

# Datei als Standardeingabe öffnen open (STDIN, '<name.txt') or
die 'Konnte name.txt nicht öffnen';

# Eine Zeile von STDIN einlesen my $name = <>;

# Zeilenumbruch entfernen chomp $name;

# Gruß ausgeben print "Hallo, $name, einen schönen Tag
noch!\n";
```

Die erste Anweisung schließt den (implizit geöffneten) Kanal von der Tastatur, um ihn danach so zu öffnen, dass er aus der Datei name.txt liest. Die spitze linke Klammer bedeutet dabei, dass die Datei zur Eingabe gelesen wird.

- Lesen Sie mit perldoc -f nach, wie close und open funktionieren.
- open ist eine sehr mächtige Funktion. Versuchen Sie zunächst nur, ihre Funktionsweise in Ansätzen zu verstehen.
- Schreiben Sie ein Programm, das Ihren Vor- und Nachnamen von der Tastatur einliest und danach ausgibt.
- Schreiben Sie Ihr Programm so um, dass es die Eingaben aus einer Datei liest. **Hinweis:** Der Diamantoperator liest stets eine Zeile ein.
- Was passiert bei obigem Programm, wenn Sie nichts eingeben oder wenn die Textdatei, aus der Sie einlesen, leer ist? Schlagen Sie die Fehlermeldung auf der Handbuchseite *perldiag* nach!

Dateien

Kapitel 13

Dateien

Dateitest-Operatoren

Ein Dateizugriff ist ein sehr systemnaher Vorgang. Zugriffe auf nichtexistente Dateien können zu Abstürzen unseres Programms führen. Um das zu vermeiden muß unser Perl-Programm sich häufig zur Laufzeit, vor dem eigentlichen Dateizugriff, informieren, was genau auf dem Dateisystem los ist. Dazu muß der Programmierer seinem Programm die richtigen Fragen an die Hand geben.

Perl stellt eine ganze Reihe von Operatoren zur Verfügung um Dateien zu testen. Diese sogenannten Dateitest-Operatoren (auch -X-Operatoren genannt) ermöglichen es dem Programmierer, zu überprüfen wie der Zustand einer Datei ist. Hier eine komplette Liste der vorhandenen Operatoren:

-r Datei / Verzeichnis ist für effektiven Benutzer oder Gruppe lesbar -w Datei / Verzeichnis ist für effektiven Benutzer oder Gruppe schreibbar -x Datei / Verzeichnis ist für effektiven Benutzer oder Gruppe ausführbar -o Datei / Verzeichnis gehört effektivem Benutzer -R Datei / Verzeichnis ist für tatsächlichen Benutzer oder Gruppe lesbar -W Datei / Verzeichnis ist für tatsächlichen Benutzer oder Gruppe schreibbar -X Datei / Verzeichnis ist für tatsächlichen Benutzer oder Gruppe ausführbar -O Datei / Verzeichnis gehört tatsächlichem Benutzer -e Datei / Verzeichnis existiert -z Datei existiert und hat die Größe null (für Verzeichnisse immer unwahr) -s Datei existiert und hat eine andere Größe als null (Wert ist Größe in Bytes) -f Angabe ist eine ||einfache|| Datei -d Angabe ist ein Verzeichnis -l Angabe ist ein symbolischer Link -S Angabe ist ein Socket -p Angabe ist eine benannte Pipe (fifo) -b Angabe ist eine spezielle Blockdatei -c Angabe ist eine zeichenorientierte Datei -u Datei / Verzeichnis ist setuid -g Datei / Verzeichnis ist setgid -k Datei / Verzeichnis hat das ||Sticky Bit|| gesetzt -t Dateihandle ist ein

TTY -T Datei sieht aus wie eine Text-Datei -B Datei sieht aus wie eine Binär-Datei -M ||Alter|| der Datei (in Tagen) -A Zeit des letzten (in Tagen) -C Letzte Änderung des Inode (in Tagen)

Hier ein Beispiel um die Funktion dieser Operatoren zu erläutern:

```
#!/usr/bin/perl use strict; use warnings; # überprüfen ob Datei existiert

my $datei = ||/tmp/testfile||; if ( -e $datei ) { print ||Datei existiert\n||; }
```

Dateien öffnen und schließen

Um eine Datei zu öffnen benutzt man in Perl die `open`-Funktion, zum Schließen die `close`-Funktion.

```
#!/usr/bin/perl use strict; use warnings; # Datei überprüfen und öffnen

my $datei = ||/tmp/testfile||; if ( -e $datei ) { print ||Datei $datei existiert\n||; open FILE, $datei or die ||Kann File $datei nicht oeffnen: $!||; close FILE; } else { print ||Datei $datei existiert nicht\n||; }
```

In diesem Beispiel wird zuerst überprüft ob die Datei existiert. Nachdem dieser Test erfolgreich ist, wird die entsprechende Meldung ausgegeben und direkt im Anschluss die Datei geöffnet (standardmäßig im Read-Modus). Würde das Fehlschlagen würde das Programm mit entsprechender Fehlermeldung versterben (`die`). Zuletzt wird das File wieder geschlossen.

Modi für Dateihandles

Im obrigen Beispiel wurde eine Datei standardmäßig lesend geöffnet. Was ist jedoch, wenn wir eine Datei schreibend öffnen wollen (zB um ein Logfile zu schreiben)? Für diese Aufgabe gibt es die sogenannten Modi mit denen eine Datei geöffnet werden kann. Hier die Übersicht:

< Pfad Zugriff nur lesend > Pfad Zugriff nur schreibend (neu anlegen / bestehende Datei überschreiben) >> Pfad Zugriff nur schreibend (neu anlegen / an bestehende Datei anhängen) +< Pfad Lese-/Schreibzugriff +> Pfad Lese-/Schreibzugriff (neu anlegen / bestehende Datei überschreiben) +>> Pfad Lese-

/Schreibzugriff (neu anlegen / an bestehende Datei anhängen) | Befehl Schreibhandle an Befehl Befehl | Lesehandle von Befehl

Somit könnte man das obige Beispiel auch wie folgt schreiben:

```
#!/usr/bin/perl use strict; use warnings; # Datei überprüfen
und öffnen

my $datei = ||/tmp/testfile||; if ( -e $datei ) { print ||Datei
$datei existiert\n||; open FILE, ||< $datei|| or die ||Kann File
nicht oeffnen: $!||; close FILE; }
```

Dateien schreiben

Jetzt, wo wir wissen, wie man eine Datei entweder lesend oder schreibend öffnet, wollen wir natürlich auch etwas mit diesen Dateien anstellen. Im nächsten Beispiel schreiben wir eine Zeile in ein File:

```
#!/usr/bin/perl use strict; use warnings; # in Datei schreiben

my $datei = ||/tmp/testfile||; if ( -e $datei ) { # Wenn Datei
existiert, tun wir garnichts print ||Datei $datei existiert\n||;
} else { # Wenn Datei nicht existiert, öffnen, bzw. anlegen.
open FILE, ||> $datei|| or die ||Kann Datei $datei nicht oeffnen:
$!\n||; print FILE ||Eine tolle Zeile voller toller Zeichen!\n||; #
hier schreiben wir rein close FILE; }
```

Wie man sieht, ist das Schreiben in eine Datei kinderleicht. Man kann der `print`-Funktion das Dateihandle übergeben und es einfach hineinschreiben lassen. **Achtung**: Kein Beistrich!

Dateien auslesen

Einfach nur eine Datei zu schreiben scheint aber auch langweilig, also wollen wir im nächsten Schritt eine Zeile aus einer Datei einlesen. `#!/usr/bin/perl use strict; use warnings; # eine Zeile auslesen`

```
my $datei = ||/tmp/testfile||; if ( -e $datei ) { print ||Datei
$datei existiert\n||;

open FILE, ||< $datei||, or die ||Fehler beim Oeffnen: $!||; my
$line = ; print $line; close FILE; }
```

Dieses Beispiel gibt die erste Zeile von */tmp/testfile* zurück. Nun ist das zwar nett, aber eventuell unzureichend für unseren Geschmack, also lesen wir im nächsten Beispiel die komplette Datei aus.

```
#!/usr/bin/perl use strict; use warnings; # Datei auslesen

my $datei = "/tmp/testfile"; if ( -e $datei ) { print "Datei
$datei existiert\n";

open FILE, "< $datei", or die "Fehler beim Oeffnen: $!"; while
() { print $_; } close FILE; }
```

Das gibt die gesamte Datei aus.

Operatoren

Kapitel 14

Operatoren

Vorbemerkung

Perl verwendet Operatoren, welche denen von C oder ähnlichen Sprachen stark gleichen. Der '='-Operator wurde ja bereits erwähnt, nun zu den Anderen.

Zuweisende und verknüpfende Operatoren

Mathematische Operatoren Zuerst die Operatoren die wahrscheinlich am meisten gebraucht werden, die mathematischen.

Grundrechenarten Am häufigsten werden wohl die meisten Programmierer die 4 Grundrechenarten gebrauchen.

```
$a = 4 + 2; # Addiere 4 und 2. $b = 4 - 9; # Subtrahiere 9
von 4. $c = 5 * 11; # Multipliziere 11 mit 5. $d = 12 / 4; #
Dividiere 12 durch 4. $d enthält nun also 3.
```

Es fehlt uns also an nichts, alle anderen mathematischen Funktionen könnten wir nun herleiten, aber Perl bietet noch mehr.

Für jede Grundrechenart gibt es nach dem Vorbild von C/C++ auch einen Zuweisungsoperator:

```
#!/usr/bin/perl

use strict; use warnings;
```

```
my $a = 1; # Deklaration und Initialisierung $a += 9; # Addiere
9 zu $a. $a -= 6; # Ziehe 6 von $a ab $a *= 3; # Multipliziere
$a mit 3 $a /= 2; # Teile $a durch 2 print "$a\n";
```

Andere Rechenarten Perl stellt auch höhere mathematische Operatoren zur Verfügung.

```
# Potenzieren $a = 2 ** 10; # 2 Hoch 10, sprich
2*2*2*2*2*2*2*2*2*2 = 1024.

# Ermittlung von Ganzzahldivisions-Resten $b = 13 % 7; # 13
Modulo 7 = 6.
```

Der Operator '**' dient zum **Potenzieren**; er sollte nicht mit dem bitweisen Operator '^' verwechselt werden, der in einigen Programmiersprachen zum Potenzieren dient.

Wurzelziehen muß im allgemeinen als Potenzieren mit dem Kehrwert des Wurzelexponenten betrachtet werden. Für die **Quadratwurzel** steht aber anstelle eines Operators die Funktion `sqrt($parameter)` zur Verfügung.

Für die **Ganzzahldivision** steht weder ein Operator, noch eine Funktion zur Verfügung, aber mit Hilfe des Modulo-Operators `%` (s.o.) läßt sich eine entsprechende eigene Funktion leicht erstellen.

```
#!/usr/bin/perl

use strict; use warnings;

print div(13, 7) ."\n";

sub div { return ($_[0] - ($_[0] % $_[1])) / $_[1] ; }
```

Hinweis: Da das Erstellen von eigenen Funktionen noch nicht erläutert wurde, hier kurz und knapp einige Hinweise:

- `sub` ist das Schlüsselwort zur Erstellung einer eigenen Funktion.
- `return` erzeugt den Ausgabewert einer Funktion.
- `$_[0]` und `$_[1]` sind die Handler für den 1. und den 2. Funktionsparameter. Perl fängt aber bei 0 an zu zählen.

Eine allgemeine Funktion für den **Logarithmus** zu einer beliebigen Basis steht nicht zur Verfügung. Aber die Funktion `log` kann den natürlichen Logarithmus berechnen. Die Mathematiker unter den Lesern wissen natürlich, daß dessen Basis die Eulersche Zahl e ist, außerdem, daß man damit auf einfache Weise eine Funktion zur Ermittlung der Logarithmen anderer Basen erstellen kann.

```
#!/usr/bin/perl

use warnings; use strict;

print logarithmus(1000, 10) . "\n";

sub logarithmus { return log($_[0])/log($_[1]); }
```

Inkrementieren und Dekrementieren Zum simplen Erhöhen oder Verringern einer Variable scheint eine normale Zuweisung oft zu lang oder (in Sonderfällen) zu unübersichtlich.

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

Zur Verkürzung gibt es, ähnlich in C, die Operatoren ++ und – um diese Statements zu verkürzen.

```
$a++; # Rückgabe von $a, dann Inkrementieren von $a. ++$a; #
Inkrementieren von $a, dann Rückgabe von $a.

$b-; # Rückgabe von $b, dann Dekrementieren von $b. -$b; #
Dekrementieren von $b, dann Rückgabe von $b.
```

Zu beachten ist hier unbedingt die Position des '++'- bzw. '–'–Operators. So hat zum Beispiel das folgendes Statement nicht die Wirkung auf $b die der Ungeübte auf den ersten Blick vermutet.

```
$a = 9; $b = $a++;
```

In diesem Beispiel besitzt die Variable $b am Ende den Wert 9, da $a zuerst zurückgegeben und dann erhöht wurde. Die gewünschte Wirkung lässt sich mit dieser Konstruktion erreichen.

```
$a = 9; $b = ++$a;
```

Nun wird $a zuerst erhöht und dann an $b zurückgegeben. Nun hat $b den Wert 10.

Stringverknüpfungs–Operatoren Des Weiteren gibt es spezielle Operatoren um Strings zusammenzufügen, welche denen der Programmiersprache PHP ähneln. Durch die Typenlosigkeit Perls ist es notwendig mitzuteilen, ob man zwei Variablen, die möglicherweise Ziffern beinhalten, wie Zahlen oder wie Zeichenketten behandeln soll.

```
$b = "Wiki"; $c = "books"; $d = 42; $a = $b . $c . $d; #
Verkette $b, $c und $d. print $a; # Gibt "Wikibooks42" aus.
```

Der Wiederholungsoperator (||x||) vervielfältigt den linken String um den angegebenen rechten Wert.

```
$b = ||o||; $b = $b x 10; # Füge $b zehnmal zusammen. print
||Wikib|| . $b . ||ks||; # Gibt ||Wikibooooooooooks|| aus.
```

Bitweise Operatoren Als nächstes die Operatoren, mit denen wir Zahlen bitweise vergleichen können.

Bitweises *AND* Der Operator ||& || für das bitweise *AND* vergleicht jeweils 2 Bits miteinander; nur wenn beide **1** sind, ist das Ergebnis auch **1**. Am besten demonstrieren wir das ganze einmal an einem Beispielprogramm:

```
#!/usr/bin/perl # bitweisesAND.pl # Demonstration bitweises
AND.

my ($foo, $bar, $baz);

$foo = 1; # In Bits: 0000 0001 $bar = 3; # In Bits: 0000 0011

$baz = $foo & $bar; # Der bitweise And-Operator. # Wir
vergleichen jedes Bit: # 0000 0001 # 0000 0011 # # 0000 0001

print '$foo: '.$foo.||\n||; print '$bar: '.$bar.||\n||; print '$baz:
'.$baz.||\n||;
```

Und die Ausgabe:

$foo: 1 $bar: 3 $baz: 1

Bitweises *OR* Der *OR*–Operator '|' dient wie auch *&* dem bitweisem Vergleich zweier Zahlen. Wie der Name schon vermuten lässt wird das Ergebnisbit nur auf *1* gesetzt wenn wenigstens eines der beiden Bits *1* ist.

Hier das Beispielprogramm:

```
#!/usr/bin/perl -w # bitweisesOR.pl # Demonstration bitweises
OR.

my ( $foo, $bar, $baz );

$foo = 5; # In Bits: 0000 0101 $bar = 15; # In Bits: 0000 1111

$baz = $foo | $bar; # Der bitweise Or-Operator. # Wir
vergleichen jedes Bit: # 0000 0101 # 0000 1111 # # 0000 1111
```

```
print '$foo: '.$foo."\n"; print '$bar: '.$bar."\n"; print '$baz:
'.$baz."\n";
```

Und hier wiedermal die Ausgabe:

$foo: 5 $bar: 15 $baz: 15

Bitweises *XOR* Der *XOR*–Operator ^ dient wie seine Geschwister dem bitweisem Vergleichen von Zahlen; er verhält sich ähnlich dem *OR*–Operator, jedoch wird beim Ergebnis ein Bit nur auf *1* gesetzt, wenn **genau** eines der beiden Bits *1* ist. Das nennt man *exklusives Oder*.

Und hier wieder mal das Beispielprogramm:

```
#!/usr/bin/perl -w # bitweisesXOR.pl # Demonstration bitweises
XOR.

my ( $foo, $bar, $baz );

$foo = 3; # In Bits: 0000 0011 $bar = 61; # In Bits: 0011 1101

$baz = $foo ^bar; # Der bitweise XOR-Operator. # Wir
vergleichen jedes Bit: # 0000 0011 # 0011 1101 # # 0011 1110

print '$foo: '.$foo."\n"; print '$bar: '.$bar."\n"; print '$baz:
'.$baz."\n";
```

Und die Ausgabe des XOR-Programmes:

$foo: 3 $bar: 61 $baz: 62

praktische Anwendung Nun wollen wir das ganze doch einmal an einem praktischen Anwendungsbeispiel demonstrieren. Wir tauschen zwei Variablen aus:

```
#!/usr/bin/perl -w # PraktischeOperatoren.pl # Praktische
Anwendung eines bitweisen XOR.

my ( $foo, $bar );

$foo = 3; # Bitweise: 0000 0011 $bar = 4; # Bitweise: 0000 0100

$foo = $foo ^bar; # $foo: 0000 0111

$bar = $foo ^bar; # $bar: 0000 0011

$foo = $foo ^bar; # $foo: 0000 0100

print '$foo: '.$foo."\n"; print '$bar: '.$bar."\n";
```

Und nun die Ausgabe:

$foo: 4 $bar: 3

Wie man sieht, haben wir die beiden Variablen ohne den Einsatz einer Hilfsvariable ausgetauscht. Diese Technik ist sehr elegant, birgt aber auch Risiken, so zum Beispiel bei negativen Zahlen, bzw Zahlen nahe am Ganzzahlvariablenmaximum.

Logische Operatoren

Logische Operatoren dienen dazu, Aussagen miteinander zu vergleichen. Soll zum Beispiel überprüft werden, ob eine Zahl $a im Bereich von 2 bis 5 liegt, so muss man zwei Vergleiche durchführen: ist $a größer oder gleich 2, und ist es kleiner oder gleich 5?

Binäre Operatoren Das oben genannte Beispiel schreibt sich in Perl so:

```
if (2 <= $a and $a <= 5) { ... }
```

der and-Operator (den man auch als '& & ' schreiben kann), liefert nur dann ein wahres Ergebnis zurück, wenn beide verknüpfte Ausdrücke wahr sind.

Analog dazu gibt es den or–Operator (der auch als '||' geschrieben werden kann), der ein wahres Ergebnis zurückgibt, wenn mindestens einer Operanden, also der Ausdrücke links und rechts des Operators, wahr ist.

Beispiel:

```
if ($a >= 2 or $a <= -2) { ... }
```

wird dann wahr, wenn $a größer gleich 2 ist oder kleiner gleich -2.

Obwohl man die Operatoren **and** und **or** in den meisten Fällen mit **& &** und || ersetzen kann, muss man beachten, dass **and** und **or** einen wesentlich niedrigeren Vorrang haben. Dazu ein Beispiel:

```
$a = 0; $b = 2; $c = 0; $abc = $a || $b || $c; # $abc bekommt den
Wert 2 zugewiesen.
```

ersetzt man in diesem Beispiel die || durch ein **or** passiert jedoch folgendes:

```
$a = 0; $b = 2; $c = 0; $abc = $a or $b or $c; # $abc wird
0 zugewiesen, weil der Zuweisungsoperator höheren Vorrang
hat. # Ist die Zuweisung erfolgreich, wird die or-Operation
durchgeführt.
```

Stringvergleichende Operatoren Aufgrund der Typenlosigkeit von Perl ist es nicht möglich bei einem Vergleich genau zu sagen was verglichen werden soll. Ein Beispiel: In **$a** enthält die Zeichenkette '1abc' und **$b** enthält den Integer **5**. Bei dem Vergleich

```
if ( $a == $b )
```

Würde $a (die Zeichenkette) zum Vergleich nach Integer *gecastet* (konvertiert) werden. Das hat keine direkten Auswirkungen auf $a, aber während des Vergleiches wird **$a** von *1abc* zu *1*. Das kann zu unerwünschten Nebenwirkungen führen. Um dieses Problem zu beheben gibt es einige Operatoren zum Vergleich von Zeichenketten. Diese sind:

- eq (*equal* / *gleich*)
- ne (*not equal* / *ungleich*)
- cmp (*compare* / *vergleiche*)
- lt (*lower than* / *kleiner als*)
- le (*lower than or equal* / *kleiner oder gleich*)
- ge (*greater than or equal* / *größer oder gleich*)
- gt (*greater than* / *größer als*)

Diese Operatoren sollen hier kurz erläutert werden.

- Der **eq**–Vergleich zweier Zeichenketten ergibt **wahr** wenn die beiden Zeichenketten **exakt gleich** sind.
- Der **ne**–Vergleich zweier Zeichenketten ergibt **wahr** wenn die beiden Zeichenketten **nicht gleich** sind.

Diese beiden Operatoren sind vergleichbar mit **==** und **!=**.

Beispiele:

'asdf' eq 'asd' # ergibt falsch 'asdf' eq 'asdf' # ergibt wahr 'asdf' lt 'asdg' # ergibt wahr 'asdf' gt 'asdg' # ergibt falsch

Spezielle Operatoren

Kombinierte Operatoren Perl erkennt die von C geläufigen Zuweisungsoperatoren und besitzt noch einige mehr. Folgende Zuweisungsoperatoren gibt es:

= *= & & = |= .= x= **= & = -= >>= %= += $< $$< $= /= ||= ≙

Beispiele:

```
$a += 2; # $a wird um 2 vergrössert. $a -= $b; # $a wird
um $b verkleinert $a .= $b; # $a und $b werden in $a
zusammengefügt(verkettet). $a & = $b; # siehe bitweise
Operatoren $a |= $b; # $a ≙ 2; #
```

Der Bereichsoperator Der Bereichsoperator erstellt eine Liste mit Werten im Bereich zwischen dem linken Operanden und dem rechten Operanden. Wobei sowohl .. als auch ... verwendet werden können (diese sind gleichwertig). Er verhält sich ziemlich intelligent und ist durchaus in der Lage Listen, welche aus Buchstaben bestehen, zu erstellen. Die Listen können zum Beispiel in Schleifen durchlaufen werden, aber auch mit simplen Befehlen.

```
#!/usr/bin/perl -w print (10 .. 21); print "\n"; #
Zeilenumbruch ausgeben.

# Das ganze geht genausogut mit Schleifen. for ( A .. F ) {
print }
```

Programmausgabe:

10111213141516171819202 1 ABCDEF

Kontrollstrukturen

Kapitel 15

Kontrollstrukturen

Fallunterscheidungen / Verzweigungen

Oft werden Programmteile erst ausgeführt, wenn eine bestimmte Bedingung eingetreten ist. Diese Bedingung muss dann zunächst geprüft werden. Mit den Mitteln, die wir bisher kennengelernt haben, kommen wir nicht zum Ziel. In Perl schaffen hier `if`-Konstrukte Abhilfe: Die einfachste Fallunterscheidung prüft, ob eine Bedingung zutrifft. Dafür wird vor einen Block die `if`-Anweisung gesetzt, der die Bedingung in Klammern folgt:

```
#!/usr/bin/perl use strict; use warnings;

# Zur Eingabe auffordern print "Bitte geben Sie ein Zahl
ein:\n";

# Zeile einlesen my $z = ;

# Zeilenumbruch entfernen chomp $z;

# Ist z 0? if ( $z == 0 ) { # ja, z ist 0 print "Die Eingabe
ist 0\n"; }
```

Obiges Programm gibt nur dann etwas aus, wenn tatsächlich 0 eingegeben wird.

> Die Bedingung, die hinter der if-Anweisung in Klammern steht, wird im boolschen Kontext ausgewertet, der eine Sonderform des skalaren Kontexts ist. Mehr dazu steht im Abschnitt über Kontexte.

Wir möchten nun ein Programm schreiben, das überprüft, ob eine ganze Zahl gerade oder ungerade ist und uns dann eine entsprechende Ausgabe liefert.

```
#!/usr/bin/perl use strict; use warnings;
```

```
# Zur Eingabe auffordern print "Bitte geben Sie ein Zahl
ein:\n";

# Zeile einlesen my $z = ;

# Zeilenumbruch entfernen chomp $z;

# Ist z gerade? if ( $z % 2 == 0 ) { printf ( "%d ist gerade\n"
, $z ); }

# Ist z ungerade? if ( $z % 2 == 1 ) { printf ( "%d ist
ungerade\n" , $z ); }
```

Die erste printf-Anweisung wird nur dann ausgeführt, wenn die Bedingung in den runden Klammern nach dem Schlüsselwort `if` erfüllt ist. Nur wenn sich **$z** ohne Rest durch 2 dividieren lässt, wird ausgegeben, dass die Zahl gerade ist. Genauso funktioniert auch die zweite `if`-Bedingung. Mithilfe von `else`, einem weiteren Schlüsselwort, könnten wir das Programm auch so realisieren:

```
#!/usr/bin/perl use strict; use warnings;

# Zur Eingabe auffordern print "Bitte geben Sie ein Zahl
ein:\n";

# Zeile einlesen my $z = ;

# Zeilenumbruch entfernen chomp $z;

if ( $z % 2 == 0 ) { printf ( "%d ist gerade\n" , $z ); } else
{ printf ( "%d ist ungerade\n" , $z ); }
```

Die `printf`-Anweisung, die in den geschweiften Klammern hinter `else` steht, wird ausgeführt, wenn die Bedingung in der vorherigen `if`-Anweisung nicht erfüllt ist. Dies entspricht auch der Bedeutung der beiden Wörter `if` und `else` in der englischen Sprache: **falls** (*engl.* if) die erste Bedingung wahr ist, wird die erste Anweisung ausgeführt, **ansonsten** (*engl.* else) wird die zweite Anweisung ausgeführt. Da eine ganze Zahl, die nicht gerade ist, zwangsweise ungerade sein muss, arbeitet das Programm nach wie vor korrekt.

- Das erste Programm prüft, ob die Eingabe 0 ist. Auf wieviele verschiedene Arte können Sie 0 eingeben?
- Geben Sie beim ersten Programm 0 but true ein. Wie erklären Sie sich die Ausgabe?
- Was passiert, wenn Sie keine Zahl eingeben? Schlagen Sie die Fehlermeldungen auf der Handbuchseite *perldiag* nach!
- Die Eingabe an ihr Programm kann interaktiv erfolgen oder von einem anderen Programm stammen. Formulieren Sie auf der Kommandozeile eine Befehlsfolge, die die Ausgabe eines Programms ihrem Programm als Eingabe zur Verfügung stellt.
- Was passiert, wenn Sie oder das andere Programm keine Daten eingeben? Schlagen Sie die Fehlermeldungen auf der Handbuchseite *perldiag* nach!
- Geben Sie beim zweiten Programm nicht ganze Zahlen ein. Wie erklären Sie sich die Ausgabe?
- Schreiben Sie ein Programm, das überprüft, ob die Eingabe positiv, negativ oder 0 ist.

In Perl können wir `if-else`-Konstrukte beliebig schachteln. Folgendes Beispiel enthält eine `if`-Abfrage, die sich in einem Anweisungsblock einer anderen `if`-Abfrage befindet:

```
#!/usr/bin/perl use strict; use warnings;

# Zur Eingabe auffordern print "Bitte geben Sie das Passwort
ein:\n";

# Passwort einlesen my $passwort = ; chomp $passwort;

# Abfragen, ob das Passwort korrekt ist if ( $passwort eq
"topsecret" ) {

# Passwort korrekt print "Wollen Sie ihren Kontostand
abfragen?\n"; my $eingabe = ; chomp $eingabe;

# Gegebenenfalls Kontostand anzeigen if ( $eingabe eq "ja" ) {
print "Kontostand: 0 Euro\n"; } } else {

# Passwort falsch print "Das Passwort war leider nicht
korrekt.\n"; } print "Auf Wiedersehen!\n";
```

Erst wenn das Passwort korrekt ist, können wir überprüfen, ob der Kontostand angezeigt werden soll. Aus diesem Grund schachteln wir die Fallunterscheidungen. Es empfiehlt sich, auf solche Schachtelungen wenn möglich zu verzichten, um den Quelltext übersichtlich zu gestalten. Hierzu folgendes Beispiel:

```
#!/usr/bin/perl use strict; use warnings;
```

```
my $eingabe = 50;

if($eingabe > 42){ if($eingabe % 2 == 0){ if($eingabe != 92){
$eingabe = 0; } } }
```

Mithilfe der logischen Operatoren, die wir schon kennengelernt haben, lässt sich das Programm übersichtlicher schreiben:

```
#!/usr/bin/perl use strict; use warnings;

my $eingabe = 50;

if($eingabe > 42 & & $eingabe % 2 == 0 & & $eingabe != 92){
$eingabe = 0; }
```

Standardfallunterscheidung Ein allgemeines Wörtchen zu **Wahrheit** und **Falschheit** in Perl: Die Ausdrücke 0, undef und alle Ausdrücke/Bedingungen, die dazu evaluiert werden, sind **falsch**. Alle anderen Ausdrücke/Bedingungen sind **wahr**.

if(BEDINGUNG1) { ANWEISUNGSBLOCK } elsif(BEDINGUNG2) { ANWEISUNGSBLOCK } elsif(BEDINGUNG3) { ANWEISUNGSBLOCK } else { ANWEISUNGSBLOCK }

Die geschweiften Klammern sind zwingend notwendig! Die einfache Fallunterscheidung ist ein Spezialfall der mehrfachen Fallunterscheidung.

negierte Fallunterscheidung unless(BEDINGUNG) { ANWEISUNGSBLOCK } else { ANWEISUNGSBLOCK }

Dies entspricht einer Verwendung von ||`if`(not BEDINGUNG)||.

abgekürzte einfache Fallunterscheidung Aus manchen Programmiersprachen ist ein abgekürztes Verfahren für einfache Fallunterscheidungen mit jeweils genau einer Anweisung je Fall bekannt. Auch Perl bietet vereinfachte Verfahren hierfür an:

Verfahren 1: BEDINGUNG & & ANWEISUNG_WAHR; BEDINGUNG || ANWEISUNG_FALSCH;

Verfahren 2: ANWEISUNG_WAHR if BEDINGUNG; ANWEISUNG_FALSCH unless BEDINGUNG;

Schleifen

Schleife mit Laufbedingung while(BEDINGUNG) { ANWEISUNGSBLOCK }

Daneben existiert folgende Form der `while`-Schleife. Sie bewirkt, dass der Anweisungsblock mindestens einmal ausgeführt wird, da die Abfrage der Bedingung erst am Ende stattfinden:

do { ANWEISUNGSBLOCK } while(BEDINGUNG);

Es gibt außerdem eine Kurzform:

ANWEISUNG while BEDINGUNG;

Schleife mit Abbruchbedingung until(BEDINGUNG) { ANWEISUNGSBLOCK }

und analog die Kurzform:

ANWEISUNG until BEDINGUNG;

Schleife mit Laufvariable for(STARTANWEISUNG; LAUFBEDINGUNG; LAUFANWEISUNG) { ANWEISUNGSBLOCK }

In der Regel wird in der Startanweisung die Laufvariable initialisiert, die Laufbedingung enthält einen Grenzwert und vergleicht diesen mit der Laufvariablen und in der Laufanweisung wird die Laufvariable inkrementiert (i.e. um 1 erhöht).

Listenabarbeitungsschleife foreach VARIABLE (LISTENVARIABLE) { ANWEISUNGSBLOCK }

Bei dieser Schleife kann auf das aktuelle Element des abgearbeiteten Arrays mit **$_** zugegriffen werden.

Beispiel: `my @liste = qw( asdf jklö 1 2 3 4.56 ); foreach ( @liste ) { print "$_" . "\n"; }` Dieses Beispiel würde folgenden Output liefern: asdf jklö 1 2 3 4.56

Natürlich kann man in gewohnter TIMTOWTDI-Manier auch hier die alternative Schreibform benutzen falls gewünscht: `#!/usr/bin/perl print for ( 1 .. 9 );`

Dieses Stück Code würde, wie erwartet, die Zahlen von 1 bis 9 ausgeben.

Sprunganweisungen

goto Von der Verwendung von `goto` wird im Allgemeinen zugunsten der Übersichtlichkeit des Quellcodes abgeraten, und das nicht nur bei Perl. Trotzdem beherrscht Perl das Konzept:

... goto SPRUNGMARKE; ... SPRUNGMARKE: ...

Als Sprungmarke lässt sich ein beliebiger Text verwenden.

redo `redo` wird benutzt um wieder zum Anfang einer Schleife zu springen, ohne dabei die Laufanweisung auszuführen. `#!/usr/bin/perl use strict; use warnings;`

```
print 'hallo'.“\n“; for (my $i = 1; $i <= 10; $i++) { print
'nun sind wir vor redo'.“\n“; if ($i == 2) { print 'wir
lassen eine Stelle aus und gehen direkt wieder zum Anfang der
Schleife'.“\n“; redo; } print 'nun sind wir nach redo'.“\n“; }
```

Das gibt folgendes aus: nun sind wir vor redo nun sind wir nach redo nun sind wir vor redo wir lassen eine Stelle aus und gehen direkt wieder zum Anfang der Schleife nun sind wir vor redo wir lassen eine Stelle aus und gehen direkt wieder zum Anfang der Schleife nun sind wir vor redo wir lassen eine Stelle aus und gehen direkt wieder zum Anfang der Schleife nun sind wir vor redo ...

continue

last Die `last`-Funktion wird benutzt um eine Schleife anzuhalten und dann im Programmcode fortzufahren. `#!/usr/bin/perl use strict; use warnings;`

```
print “hallo\n“; for (my $i = 1; $i <= 100; $i++) { print
'$i: ' . $i . “\n“; if ($i == 3) { last; } } print 'last
wurde ausgefuehrt, die Schleife wurde angehalten und es geht
weiter...'; Das wird folgendes ausgeben:
```

hallo $i: 1 $i: 2 $i: 3 last wurde ausgefuehrt, die Schleife wurde angehalten und es geht weiter...

next Dieses Kommando arbeitet ähnlich wie das bereits beschriebene `redo`. Es erfolgt ein Sprung zum Beginn der Schleife. Im Gegensatz zu `redo`, wird jedoch in

`for`, `while` und `until` Schleifen die Bedingung mit überprüft. Der Schleifenkopf wird also ausgewertet, als ob die Schleife normal wiederholt würde.

Programmabbrüche

exit Die `exit`-Funktion beendet sofort die Programmausführung mit einem angegebenen Exit-Status. Diese Funktion wird verwendet, um das Programm bei Fehlern zu beenden und um mithilfe des zurückgegeben Wertes verschiedene Arten von Fehlern zu unterscheiden.

Beispiel: `if(!-e "nix.txt") { # alternativ: unless ( -e "nix.txt" ) print "Datei nicht gefunden!"; exit(1); }` Hier wird überprüft, ob eine Datei namens *nix.txt* existiert. Wenn nicht, wird eine Fehlermeldung ausgegeben und und das Programm mit dem Wert 1 beendet.

warn Die `warn`-Funktion gibt einen Text sowie die Zeilennummer der betreffenden Zeile im Quelltext auf die Standardfehlerausgabe (*STDERR*) aus. Diese Funktion wird oft in Kombination mit der `exit`-Funktion bei der Fehlersuche benützt, um Fehlermeldungen in eine log-Datei zu schreiben und das Programm zu beenden.

Beispiel: `if(!-e "nix.txt"){ warn "Datei nicht gefunden!"; exit(1); }` Hier wird zuerst überprüft, ob eine Datei "nix.txt" existiert. Wenn nicht, wird eine Fehlermeldung ausgegeben und das Programm beendet. **Die Ausgabe würde so aussehen:** Datei nicht gefunden! at beispiel.p line 2.

Die unten beschriebene Funktion `die` ist eine vereinfachte Kombination der Befehle `warn` und `exit`.

die Die `die`-Funktion gibt die Fehlermeldung, die ihr übergeben wird auf der Standardfehlerausgabe (*STDERR*) aus und beendet das Programm mit einem Exit-Status ungleich 0. Dies kann praktisch sein, um schwere Fehler im Vorhinein abzufangen (zB wenn eine notwendige Datei nicht geöffnet werden kann).

Beispiel: `open LOGDATEI, "> /tmp/log_mein_prog" or die "Kann Log-Datei nicht oeffnen: $!\n";` In diesem Beispiel wird versucht, die Logdatei "*/tmp/log_mein_prog*" zum Schreiben (>) zu oeffnen. Falls dies fehlschlägt, wird das Programm mit die und der mitgegebenen Meldung beendet. Die Spezialvariable *$!* enthält die System-Fehlermeldung in lesbarem Format (zB. *permission denied*).

Subroutinen

Kapitel 16

Subroutinen

Vorbemerkung

Es wird bei Subroutinen in der allgemeinen Programmierung unterschieden zwischen Prozeduren und Funktionen. Funktionen erzeugen Rückgabewerte, Prozeduren nicht. Diese Unterscheidung ist bei Perl unerheblich, insofern wird an dieser Stelle in der Folge von Funktionen gesprochen.

Funktionen werden immer dann verwendet, wenn sich ein Vorgang mehrfach verwenden lässt. Die Routinenlogik wird dann aus dem Hauptprogramm ausgelagert und mit einem Funktionsaufruf eingebunden. Der Vorteil, dass der Vorgang dann nur einmal programmiert werden muss, liegt auf der Hand. Ein weiterer Vorteil ist, dass auch die Wartung und Modifikation des Programms erleichtert wird, weil Änderungen ebenfalls nur an einer Stelle vorgenommen werden müssen. Ein ausgiebiger Gebrauch von Funktionen ist also empfehlenswert.

Allgemeine Deklaration

Aufbau sub FUNKTIONSNAME{ DEKLARATIONSBLOCK ANWEISUNGSBLOCK return RÜCKGABEVARIABLE; }

Die Deklaration von Funktionsinternen Variablen erfolgt entweder mit 'local' oder mit 'my', z.B.: `my $param1 = 0; local $param2 = 0;`

Bei der Verwendung von 'local' steht die Variable auch untergeordneten Funktionen zur Verfügung.

Die Parameterübergabe erfolgt in Perl nicht wie üblich im Funktionskopf, stattdessen steht das vordefinierte Default-Array '@_' zur Verfügung, zum Beispiel: `my $param1 = $_[0]; local $param2 = $_[1]; ...`

Prototyp Ein Prototyp hat die Aufgabe, dem Hauptprogramm den Namen und die Parametertypen mitzuteilen. Er schreibt sich wie ein reduzierter Funktionskopf:

Beispiel 1, Funktion mit zwei skalaren Übergabeparametern und einem optionalen skalaren Übergabeparameter: sub FUNKTIONSNAME($$;$)

Beispiel 2, Funktion mit einer Übergabeliste: sub FUNKTIONSNAME(@)

Aufruf Funktionen werden mit Symbol & und FUNKTIONSNAME aufgerufen.

Möglichkeit 1: & FUNKTIONSNAME(PARAMETERLISTE);

Möglichkeit 2: FUNKTIONSNAME(PARAMETERLISTE);

Möglichkeit 3: & FUNKTIONSNAME PARAMETERLISTE;

Bei Verwendung eines Prototyps oder einer Paketfunktion gibt es auch noch die

Möglichkeit 4: FUNKTIONSNAME PARAMETERLISTE;

Anmerkung:

Subroutinen lassen sich ohne und mit Parameterlisten aufrufen. Die Parameter finden sich innerhalb der Funktion in der Struktur @_wieder und können mit $_[] ($_[0], $_[1] ... usw.) separat angesprochen werden.

Beispiel:

```
sub groesser{ if ($_[0] > $_[1]){ $_[0]; #ergebnis }else{ $_-
[1]; #ergebnis = rueckgabewert } }
```

Aufruf mit & groesser(24,15) liefert als Ergebnis 24. ($_[0]=24 der Rückgabewert).

mit lokalen Variablen: `sub groesser{ local ($a,$b) = ($_[0],$_[1]); if ($a > $b){ $a; }else{ $b; } }` liefert das gleiche Ergebnis wie oben.

Das in C, C++ und anderen Programmiersprachen verwendete return zur R& uuml;ckgabe des Funktionswertes ist in Perl am Ende einer Funktion optional. Von der Funktion wird immer der R& uuml;ckgabewert des letzten Befehls zur&

uuml;ck gegeben. Die folgende Funktion test liefert zum Beispiel den Wert 1, da dieser bei erfolgreicher Ausf& uuml;hrung der festgelegte R& uuml;ckgabewert von print ist. `sub test{ print ||Nichts zu tun.\n\n||; }`

Kontext Eine Besonderheit in Perl ist, dass der Rückgabewert einer Funktion nicht nur von den Parametern abhängt, sondern auch von dem Kontext, in dem der Aufruf erfolgt. Deshalb sind solche Unterprogramme keine Funktionen im strengen Sinne. Das Schlüsselwort 'sub' in Perl ist von ||subroutine|| = ||Unterprogramm|| abgeleitet.

Den Kontext muss man auch bei der Verwendung einiger eingebauter Funktionen beachten, zum Beispiel bei der Verwendung der Funktion 'each'. `my %hash = (Kuh => 'Milch', Kamel => 'Wasser', Gnu => 'Freie Software');`

`while( my $tier,$topic = each(%hash) ){ print $tier,|| => ||,$topic,||\n|| }`

`while( my ($tier,$topic) = each(%hash) ){ print $tier,|| => ||,$topic,||\n|| }` Liefert die Ausgabe:

=> Gnu => Kamel => Kuh Gnu => Freie Software Kamel => Wasser Kuh => Milch

Bei der zweiten Schleife wird mit den Klammern ein Listenkontext hergestellt. In diesem liefert each ein Schlüssel-Wert Paar. Im skalaren Kontext oben, wird der Variable $topic bei jedem Schleifendurchlauf ein Schlüsselwert zugewiesen.

Um in den eigenen Unterprogrammen, diese Funktionalität nutzen zu können, liefert die Funktion wantarray einen wahren Wert, wenn ein Aufruf in einem Listenkontext erfolgt.

Externe Subroutinen

Viele Funktionen sind bereits in den vielen hundert Modulen im CPAN und anderswo als freie Software verfügbar. Um dabei Namenskonflikte zu vermeiden werden zusammengehörige Funktionen in einem Namensraum zusammengefasst.

Als Beispiel möchte ich das Modul File::Temp nennen. Es verwendet den Namensraum File::Temp und enthält zwei Funktionen, eine zur Erstellung von temporären Dateien (tempfile) und eine für Verzeichnisse (tempdir).

Namensräume sind in Perl hierarchisch aufgebaut und sollten vom Programmierer sinnvoll gewählt werden. Außerdem ist, wie man sieht, der Namensraum meist

identisch mit dem Modul. Es kann aber sein, dass umfangreiche Module viele Namensräume nutzen.

Es sollte so sein, dass der gewählte Name in einem Zusammenhang mit der Funktion des Moduls steht. So stehen alle Module im Namensraum File, in einem Zusammenhang mit Dateien.

Laden eines Moduls mit use use File::Temp;

In dieser Form wird das Modul während der Übersetzungszeit geladen. Außerdem wird eine spezielle Funktion in dem zu ladenden Modul aufgerufen. Ihr Name ist import.

use File::Temp ();

Schreibt man hinter das Modul eine leere Liste, so wird der Aufruf von import übergangen.

use File::Temp qw/ tempfile tempdir /;

In dieser Form zeigt sich woher die spezielle Funktion ihren Namen hat. Es ist häufig so, dass Funktionen in aktuellen Namensraum geladen werden können, sie werden quasi importiert. Genau das soll mit dieser Form von use ausgedrückt werden.

Erstellung

Bibliothek Eine Bibliothek ist eine Datei, die Funktionsdefinitionen enthält, deren Dateiname auf *.pl endet, und deren letzte Zeile aus dem Code 1; besteht.

Modul

Einbindung und Aufruf Eine Bibliothek wird mit 'require' eingebunden: `require(‖DATEINAME.pl‖);`

Bibliotheksfunktionen werden dann wie interne Funktion aufgerufen.

Ein Modul wird mit 'use' eingebunden `use MODULNAME;`

und mit `MODULNAME::FUNKTIONSNAME();` aufgerufen.

Einfache Beispiele für den Einstieg

Kapitel 17

Einfache Beispiele

Fallunterscheidung

Ein Beispiel für eine if/elseif/else-Struktur

```
#!/usr/bin/perl -w # if/elsif/else # 20041110

if (1233 > 2333) { print "Ergebnis 1\n"; } elsif (3333 > 4444)
{ print "Ergebnis 2\n"; } else { print "Ergebnis 3\n"; }

# ergibt: Ergebnis 3 # das selbe ohne if/elsif/else
(TIMTOWTDI-Prinzip)

print "Ergebnis ", 1233 > 2333 ? 1 : 3333 > 4444 ? 2 : 3,
"\n";
```

Stringvergleiche

Ein Beispiel für den Vergleich von Strings

```
#!/usr/bin/perl -w # Stringvergleich eq, ne (equal, not equal)

if ("huhu" eq "huhu") { print "beide Zeichenketten gleich"; }

if ("wiki" ne "books") { print "strings sind unterschiedlich";
}
```

Ein weiteres Beispiel

```
#!/usr/bin/perl -w # 20041110
```

```
print "1+1 = 4\n" if 1+1 == 2;
```

something else

```
#!/usr/bin/perl -w print "\aAlarm\n"; # \a gibt einen Piepton
aus, es folgt die Zeichenkette 'Alarm' # und \n gibt eine neue
Zeile aus; print `hostname`; # fuehrt das Kommando "hostname"
in einer Shell aus und liest # die Standardausgabe der Shell
und zeigt diese als Resultat # an. "hostname" ist also kein
Perl-Befehl!;
```

```
$var0 = "0123456789"; print substr($var0, 3, 5); # gibt '34567'
aus; substr holt aus $var0 von Zeichen nr.3 die # darauf
folgenden 5 Zeichen und gibt sie aus ...;
```

Dateihandling

Ausgabe des Inhaltes von $text in die Datei test.txt

```
#!/usr/bin/perl print "Wie heißt Du? "; $text = ; chop($text);
open(file, ">test.txt") or die "Fehler beim Öffnen der Datei:
$!\n"; print file $text; close (file) or die "Fehler beim
Schließen von 'test.txt': $! \n";
```

Umwandlung in HTML

Funktion zur Umwandlung von Umlauten in deren HTML-Äquivalente. Alternativ können diese mit dem Zusatzparameter '2' auch in der Doppellautschreibweise dargestellt werden. Die ganze Funktion kann in eine Bibliotheks-Datei im cgi-bin-Verzeichnis ausgelagert werden, um dann bei Bedarf von allen CGI-Perl-Dateien mit "require" eingebunden zu werden.

```
sub umlautwechseln { my $return = $_[0]; # erster
Parameter my $matchcode = $_[1]; # zweiter Parameter
if(!defined($matchcode)) { $matchcode = 1; }
```

```
my @vorlage = ( ['ä', 'ö', 'ü', 'Ä', 'Ö', 'Ü', 'ß'], ['&
auml;', '& ouml;', '& uuml;', '& Auml;', '& Ouml;', '& Uuml;',
'& szlig;'], ['ae', 'oe', 'ue', 'Ae', 'Oe', 'Ue', 'ss'] );
```

```
my $vorlage = 0; for (my $i=0; $i<=6; $i++) { # alternativ:
for ( 0 .. 6 ) while ( index($return, $vorlage[0][$i], 0) > -1
```

```
) { substr ($return, index($return, $vorlage[0][$i], 0), 1) =
$vorlage[$matchcode][$i]; } }

return $return; }

1;
```

Das selbe Beispiel könnte ebenfalls mit Regular-Expressions gelöst werden, wie man im folgenden Beispiel sehen kann:

```
#!/usr/bin/perl use strict; use warnings;

sub umlautwechseln { my $messystring = shift; my $conversion =
shift || "1";

my @vorlage = ( [ 'ä', 'ö', 'ü', 'Ä', 'Ö', 'Ü', 'ß'], ['&
auml;', '& ouml;', '& uuml;', '& Auml;', '& Ouml;','& Uuml;',
'& szlig;'], [ 'ae', 'oe', 'ue', 'Ae', 'Oe', 'Ue', 'ss'] );

for (0 .. 6) { $messystring =~ s/$vorlage[0][$_-
]/$vorlage[$conversion][$_]/g; }

return $messystring; }
```

Erläuterung der Funktion:

Diese Funktion erwartet 2 Parameter, wobei der zweite Parameter optional ist (**$conversion**). Falls der zweite Parameter nicht angegeben wird, wird der Wert "1" angenommen. Der erste Parameter ist der Skalar mit den Sonderzeichen.

In Zeile 16 erfolgt die eigentliche Arbeit. **$messystring** wird mittels RegEx untersucht und entsprechende Treffer werden ersetzt.

Erzeugung von SQL-Code

Dieses Script wird benötigt für die Beispieldatenbank der Einführung in SQL. Es erzeugt etwas mehr als 100.000 SQL-Anweisungen für MySQL, die mit einer Kanalumleitung in die Beispieldatenbank eingespielt werden können, und dort jeweils einen Datensatz erzeugen.

Für Perl-Neulinge von Interesse sind vermutlich eher

- die Verwendung der ForEach-Schleife,
- die alternative Textausgabe, die besonders für längere Texte geeignet ist,

- der alternative Zugriff auf die Datenbank, der ohne das DBI-Modul auskommt.

```
#!/usr/bin/perl

use strict;

my @namen=(‖Meyer‖, ‖Müller‖, ‖Schulze‖, ‖Schneider‖,
‖Schubert‖, ‖Lehmann‖, ‖Bischof‖, ‖Kretschmer‖, ‖Kirchhoff‖,
‖Schmitz‖, ‖Arndt‖); my @vornamen=(‖Anton‖, ‖Berta‖,
‖Christoph‖, ‖Dieter‖, ‖Emil‖, ‖Fritz‖, ‖Gustav‖, ‖Harald‖,
‖Ida‖, ‖Joachim‖, ‖Kunibert‖, ‖Leopold‖, ‖Martin‖, ‖Norbert‖,
‖Otto‖, ‖Peter‖, ‖Quentin‖, ‖Richard‖, ‖Siegfried‖,
‖Theodor‖, ‖Ulf‖, ‖Volker‖, ‖Walter‖, ‖Xaver‖, ‖Yvonne‖,
‖Zacharias‖); my @orte=(‖Essen‖, ‖Dortmund‖, ‖Bochum‖,
‖Mülheim‖, ‖Duisburg‖, ‖Bottrop‖, ‖Oberhausen‖, ‖Herne‖,
‖Witten‖, ‖Recklinghausen‖, ‖Gelsenkirchen‖, ‖Castrop-Rauxel‖,
‖Hamm‖, ‖Unna‖, ‖Herten‖, ‖Gladbeck‖); my $orte=‖‖; my
@strassen=(‖Goethestr.‖, ‖Schillerstr.‖, ‖Lessingstr.‖,
‖Badstr.‖, ‖Turmstr.‖, ‖Chausseestr.‖, ‖Elisenstr.‖, ‖Poststr.‖,
‖Hafenstr.‖, ‖Seestr.‖, ‖Neue Str.‖, ‖Münchener Str.‖,
‖Wiener Str.‖, ‖Berliner Str.‖, ‖Museumsstr.‖, ‖Theaterstr.‖,
‖Opernplatz‖, ‖Rathausplatz‖, ‖Bahnhofstr.‖, ‖Hauptstr.‖,
‖Parkstr.‖, ‖Schlossallee‖); my @gesellschaften=(‖Zweite
allgemeine Verabsicherung‖, ‖Sofortix Unfallversicherung‖,
‖Buvaria Autofutsch‖, ‖Provinziell‖, ‖Vesta Blanca‖); my
@beschreibungen=(‖Standardbeschreibung Nr 502‖, ‖08/15‖,
‖Blablabla‖, ‖Der andere war schuld!‖, ‖Die Ampel war schuld!‖,
‖Die Sonne war schuld!‖, ‖Die Welt ist schlecht!!‖); my
$beschreibungen=‖‖; my $gesellschaften=0; my $gebdat=‖‖; my
$fdat=‖‖; my $hnr=0; my $eigen=‖‖;

foreach my $ort (@orte) { my $gplz=int(rand(90000))+10000;
foreach my $strasse (@strassen) { my $plz=$gplz+int(rand(20));
foreach my $name (@namen) { foreach my $vorname(@vornamen)
{ $gebdat=dating(80, 1907); $fdat=dating(80, 1927);
$hnr=int(rand(100))+1; if(rand(2)>1) {$eigen=‖TRUE‖;} else
{$eigen=‖FALSE‖;} my $vers=int(rand(5));

print <<OUT1 insert into VERSICHERUNGSNEHMER(VNE_NAME, VNE_-
VORNAME, VNE_GEBURTSDATUM, VNE_DATUM_FUEHRERSCHEIN, VNE_ORT,
VNE_PLZ, VNE_STRASSE, VNE_HAUSNUMMER, VNE_EIGENER_KUNDE_J_N,
VNE_VERSGESELLSCHAFT_ID) values (‖$name‖, ‖$vorname‖, ‖$gebdat‖,
```

```
||$fdat||, ||$ort||, ||$plz||, ||$strasse||, ||$hnr||, ||$eigen||, ||$vers||);
OUT1 }}}}
```

```
for(my $a=0; $a<=500; $a++) { my $udat=dating(3,
2004); my $ort=$orte[int(rand(16))]; my
$beschreibung=$beschreibungen[int(rand(7))]; my
$shoehe=int(rand(20000000))/100; my $verletzte; if(rand(2)>1)
{$verletzte=||TRUE||;} else {$verletzte=||FALSE||;} my
$mitarbeiter=int(rand(10))+1; print <<OUT2 insert into
SCHADENSFAELLE(SCF_DATUM, SCF_ORT, SCF_BESCHREIBUNG, SCF_-
SCHADENSHOEHE, SCF_VERLETZTE_J_N, SCF_MITARBEITER_ID) values
(||$udat||, ||$ort||, ||$beschreibung||, $shoehe, ||$verletzte||,
$mitarbeiter); OUT2 }
```

```
for(my $a=1; $a<=500; $a++) { my $vne=int(rand(100000))+1;
print <<OUT3 insert into ZUORD_VNE_SCF(ZVS_SCHADENSFALL_ID,
ZVS_VERSICHERUNGSNEHMER_ID) values ($a, $vne); OUT3 }
```

```
sub dating { my $range=$_[0]; my $radix=$_[1]; my
$y=int(rand($range))+$radix; my $m=int(rand(12))+1; my
$d=int(rand(28))+1; my $return=$y . ||-|| . $m . ||-|| . $d; return
$return; }
```

Fortgeschrittene Themen

Programmierstil und -struktur

Kapitel 18

Stil und Struktur

Kommentare

Kommentare sollten gut und reichlich verwendet werden. Es ist kein Fehler, wenn das Verhältnis zwischen Kommentaren und effektivem Code bei 1:1 liegt. Da man in Perl sehr kompakten Code schreiben kann, ist bei veröffentlichtem Code vom CPAN ein Verhältnis 2:1 und darüber völlig normal.

Zu unterscheiden sind dabei die zwei verschiedenen Formen von Kommentar, die Perl erlaubt.

Die einfachste Form ist die Kommentarzeile, die alles hinter dem #-Zeichen bis zum Zeilenende beinhaltet. Daneben unterstützt Perl von Haus aus ein Werkzeug für eingebettete Dokumentation. Dieses trägt den Titel Plain Old Documentation oder kurz POD. Es erlaubt ähnlich WikiSyntax einfache Formatierungen und Verknüpfungen.

Direkt nach der Shebang-Zeile sollte der Programmname aufgeführt werden. Das ist hilfreich, weil man häufig mit less oder more sich mehrere Quelltexte in Folge zu Gemüte führt, und man sich so orientieren kann, in welchem Skript man sich gerade befindet. Das ist auch praktisch, wenn man die Möglichkeit hat, Programmlistings auf Endlospapier zu drucken.

Anschließend sollte eine ausführliche Programmbeschreibung inline in den Quelltext eingearbeitet werden. Bei Teamprojekten ist es außerdem sinnvoll, den Namen des Autors zu hinterlegen, sowie Modifikationen festzuhalten mit Beschreibung der Modifikation, Datum und Name des modifizierenden Autors.

Häufig ist es außerdem sinnvoll, Unterfunktionen, Schleifen und Verzweigungen mit beschreibenden Kommentaren zu bedenken. Meilensteine/Wendepunkte

in der Programmlogik sollten mit einem markierenden/benennenden Kommentar versehen werden. Auch wichtige Variablen sollten mit einem Kommentar beschrieben werden.

Der Deklarationsteil

Die Variablendeklaration sollte der Übersichtlichkeit halber ausschließlich zu Beginn der Funktion (bei der Hauptfunktion zu Beginn des Programms) erfolgen. Die Programmlogik sollte sich klar vom Deklarationsteil abheben, und diesem nachfolgen.

Variablen sollten in erster Linie lokal innerhalb der Funktionen verwendet werden. Globale Variablen sollten auf das absolut nötige Mindestmaß beschränkt bleiben. Wenn ein Fehler auf eine Variable zurückgeführt werden kann, dann schränkt sich der zu durchsuchende Quelltextbereich auf diese Weise extrem ein.

Die Variablenbezeichnung sollte, trotz des Mehraufwandes beim Tippen, nicht kryptisch, sondern verständlich sein. $countkunden, $countmitarbeiter und $countlieferanten sagt mehr als $ck, $cm und $cl, vor allem, wenn der Chef sagt, daß man das Programm mal für vier Wochen links liegen lassen soll, weil eine wichtige Auftragsarbeit anliegt.

use strict; Wie schon beim "Hello World!"-Programm angedeutet, sollte der Perl-Programmierer grundsätzlich die Anweisung

```
use strict;
```

verwenden. Dadurch zwingt er sich selber, nur Variablen zu verwenden, die er vorher deklariert hat. Der augenfälligste Vorteil dabei ist, daß der Programmierer eine Fehlermeldung erhält, wenn ihm bei der Verwendung einer Variablen ein Tippfehler unterläuft. Unterläßt er dies, kann sich das in sehr merkwürdigen und unvorhergesehenen Programmabläufen und stunden- bis tagelangen Fehlersuchen äußern.

use warnings; Mit dem Pragma „warnings“ werden in dem aktuellen Block Warnungen an- und abgeschaltet. Ohne dieses Pragma führt Perl stillschweigend viele Operationen durch, die der Programmierer so nicht gemeint, aber leider eingegeben hat. In folgendem Beispiel hat sich ein Tippfehler eingeschlichen:

```
my $summe = "1:" + 2;
```

Ohne „use warnings“ wird die Operation stillschweigend durchgeführt. Mit diesem Pragma erfährt der Programmierer, dass etwas nicht stimmt:

```
use warnings; my $summe = "1:" + 2;
```

ergibt die Warnung

Argument "1:" isn't numeric in addition (+) at -e line 2.

Dieses Pragma ist in den manpages warnings und perllexwarn erläutert. In perldiag sind alle Warnungen aufgeführt, die Perl ausgibt.

selbstdefinierte Funktionen

Sobald sich Programmschritte wiederholen, sollten diese in eine Funktion ausgelagert werden. Damit wird der betreffende Code an einer Stelle zusammengefasst und nicht mehrfach über ein Programm verteilt. Die Fehlersuche und -behebung erleichtert dies ungemein, da dann nur in dieser einen Funktion gesucht werden muss und nicht an den vielen Stellen, die durch ein Kopieren eines Programmabschnittes entstanden sind.

Eine Funktion sollte wie ein gutes Werkzeug nur einen Zweck erfüllen, diesen aber dafür sehr gut. Dieser Zweck sollte sich in einem treffenden Namen niederschlagen.

Eine Funktion sollte wenn möglich auf keine nicht lokalen Variablen zugreifen und allein über ihre Argumente gesteuert werden.

Eine Funktion sollte mit POD dokumentiert werden. Dies ist unabhängig davon, ob sie exportiert wird oder nur zur internen Verwendung gedacht ist. Vermutlich wird der Autor der Funktion nach sechs Monaten nicht mehr wissen, warum er oder sie genau diesen Code für den Zweck gewählt hat, daher sollte dies kommentiert werden.

{{Vorlage:Referenzbox IntraBuch|Perl-Programmierung: Subroutinen, Kommentar von Funktionen (in Perl-Programmierung: POD), nicht lokale Variablen (in Perl-Programmierung: Variablen) }}

Logging

Bei wirklich umfangreichen Projekten ist ein Einsatz der Bibliotheken Log::Dispatch[1] und Log::Log4Perl[2] empfehlenswert, sie stellen ein ausgereiftes Logging Framework zur Verfügung. Eine kurze Einführung vom Author des letztgenannten Moduls ist hier zu finden: http://perlmeister.com/snapshots/200301/index.html .

Vorerst aber kann man es umgehen, sich dort einzuarbeiten. Völlig ausreichend bei mittelgroßen Projekten ist es, unterschiedlichste Laufzeitergebnisse in eine Log-Datei umzuleiten. Bei mir hat sich folgender, vielleicht noch ausbaufähiger Usus entwickelt:

Schritt 1:

```
#Deklarationsteil my $loglevel=5;
```

Schritt 2:

```
#Anfang der Programmlogik open (LOG, ">laufzeitausgaben.log") ||
die "Blabla\n";

. . .

#Ende der Programmlogik close(LOG);
```

Schritt 3: `#Mitten in der Programmlogik if($loglevel>4) {print LOG "VARIABLE 1: ". $variable1 . "\n";}`

```
#oder if($loglevel>6) {print LOG "VARIABLE 2: ". $variable2 .
"\n";}
```

In diesem Beispiel würde die erste Logausgabe zugelassen, die zweite unterdrückt. Diese Möglichkeit ist erwünscht.

Mit dem numerischen Wert wird jeder Logausgabe eine Priorität zugewiesen, je wichtiger, desto kleiner. Mit einer kleinen Änderung im Deklarationsteil kann ab jetzt der Loglevel bestimmt werden, und damit der Detail-Grad der Logausgaben. Ohne groß zu stören, können die Ausgabezeilen jetzt im Quelltext verbleiben, bis sie gebraucht werden.

Gültigkeitsbereich von Variablen

[1] http://search.cpan.org/~drolsky/Log-Dispatch-2.20/lib/Log/Dispatch.pm
[2] http://search.cpan.org/~mschilli/Log-Log4perl-1.14/lib/Log/Log4perl.pm

Kapitel 19

Gültigkeitsbereich von Variablen

Allgemeines

In einigen Punkten ist Perl einfacher gestrickt als andere Programmiersprachen, was den Gültigkeitsbereich von Variablen betrifft ist dies jedoch nicht zutreffend. Dabei muss unterschieden werden, zwischen dem Bereich in dem der Variablenname gültig ist und dem Bereich in dem der Inhalt verfügbar ist. Diese beide Bereiche sind nur für lexikalische Variablen unter bestimmten Bedingungen identisch, nämlich nur dann, wenn keine weitere Referenz auf den Wert der Variable angelegt wurde.

In Perl muss man sich, wie in der Einleitung zu den Variablen erklärt, weniger Gedanken darum machen, ob der Wert auf den ein Variablenname verweist noch gültig ist, da die Speicherverwaltung fast vollkommen selbständig von perl übernommen wird. Nur bei rekursiven Datenstrukturen muss man dafür sorgen, dass der Speicher am Ende der Lebensdauer wieder freigegeben wird, da der Referenzzähler nicht von selbst wieder auf 0 gesetzt wird.

Für Variablen findet man in imperativen Programmiersprachen die Unterscheidung in globale, das heißt im ganzen Programm gültige Variablen und lokale, nur in einem begrenzten Block zum Beispiel einer einzelnen Funktion gültige Variablen. Um Perl zu verstehen muss man wissen dass Perl für die Verwaltung seiner Variablen zwei verschiedene Bereiche kennt.

Es gibt die mit dem Schlüsselwort `my` deklarierten lexikalischen Variablen. Ein andere Beschreibung für sie, ist statisch gebundene Variablen, da der Gültigkeitsbereich nach der Deklaration nicht mehr geändert werden kann. Ein solcher Bereich kommt in Perl in verschiedenen Formen vor. Es kann ein Codeblock, eine

Funktion, ein eval Aufruf oder einfach eine Datei sein. Ein direkter Zugriff von ausserhalb des Bereichs auf diese Variablen ist ebenfalls nicht möglich.

Das Gegenstück dazu sind die dynamisch gebunden Variablen. Diese werden innerhalb der Namensräume verwaltet. Die Lebensdauer wird von Perl nicht beschränkt so dass man hier auch von globalen Variablen spricht.

Aus diesem Grund und weil bei lexikalischen Variablen der Zugriff ein wenig schneller erfolgt, sollte diese Form den Vorzug erhalten.

Für die globalen Variablen bietet Perl mit dem Schlüsselwort `local` die Möglichkeit den Wert der Variable auf einen besimmten Bereich des Programms begrenzt zu verändern. Die Änderung geht beim Verlassen des Bereichs verloren. Das ist der gewünschte Effekt, denn so sind globale Variablen sinnvoller einsetzbar, da sichergestellt ist, dass der ursprüngliche Wert für alle anderen Teile des Programms, wieder hergestellt wird.

Auch wenn in den folgenden Beispielen meist Skalare verwendet werden, gilt das in diesem Kapitel gesagte ebenso für Arrays und Hashes.

Globale Variablen

Für das Anlegen globaler Variablen gibt es wie üblich in Perl gleich mehrere Möglichkeiten. Die erste und einfachste Variante, die jedoch in der Praxis nicht zu empfehlen ist, besteht darin, das Pragma `strict` nicht zu verwenden.

```
package Irgendwie::Irgendwo;

$hier="Große Stadt";
```

Benutzt man dann eine Variable, so hat man die Variable mit dem Gültigkeitsbereich des aktuellen *package* definiert.

```
use strict "vars";
```

Jetzt funktioniert das nicht mehr sondern der Variablenname muss vor der Verwendung bekannt gemacht werden. Dafür gibt es auch wieder zwei Wege. Der ältere, den man nutzen sollte, falls es darauf ankommt, dass die Programme auch mit Versionen vor 5.6 laufen, sieht so aus:

```
use vars qw/$hier $da/;
```

Mit perl 5.6 wurde das Schlüsselwort our eingeführt, das fast genau den selben Effekt hat.

```
package Irgendwann; use strict;
```

```
our $zeit;

sub verlauf { our $zeit; ... }
```

Normalerweise wird einer Variable bei ihrer Deklaration ein Wert zugewiesen. Entweder explizit im Programm mittels einer Wertzuweisung

```
our $gestern="heute";
```

oder implizit durch Perl, nämlich undef für Skalare und die leere Liste bzw. Hash für Listen und Hashes.

```
our @tage # entspricht () our %sonnentage # ebenso
```

Existiert eine dynamisch gebundene Variable bereits, wird eine weitere Deklaration mit `our`, keine implizite Wertzuweisung vornehmen. Das ist sicher auch das Verhalten, welches von einer globalen Variable erwartet werden kann. Man kann so gefahrlos innerhalb einer Funktion (wie in dem Beispiel "verlauf") die Deklaration wiederholen. Sie ist dann nur für alle, die den Programmtext studieren müssen, ein Hinweis darauf, woher diese Variable kommt und dass es sich um eine Paketvariable handelt.

Perlkenner werden es vermuten beziehungsweise wissen, dass es noch andere Möglichkeiten gibt, globale Variablen zu erstellen. Eine ist der ersten vorgestellten Möglichkeit sehr ähnlich und besteht darin, den vollständigen Variablennamen anzugeben. Dieser besteht aus dem Paketnamen und dem eigentlichen Variablennamen.

```
$Irgendwie::Irgendwo::hier = "ein kleines Dorf";
```

Diese Variante sollte aber nur dann eingesetzt werden, wenn es keine andere Möglichkeit gibt, da in diesem Fall die Schutzmechanismen, die `use strict` vor Schreibfehlern bietet, nicht greifen. Nötig ist diese Art, auf eine Variable zuzugreifen, immer dann, wenn von einem anderen Namensraum aus auf eine Variable zugegriffen wird. Außerdem ist bei dieser Form zu beachten, dass zwar die Variable erstellt, aber der Name nicht bekannt gemacht wird. Nur **$hier** ohne Paketname ist für den Übersetzungsvorgang eine noch nicht deklarierte Variable und würde bei aktiviertem `use strict` einen Fehler bedeuten.

Das Perl-Standardmodul Exporter[1] bietet eine Möglichkeit, Variablennamen zu exportieren, also den Variablennamen für einen anderen Namensraum nutzbar zu machen. Damit kann eine Variable von mehreren Paketen gemeinsam genutzt werden, ohne immer den vollständigen Paketnamen angeben zu müssen. Es wird dafür im aktuellen Paket ein Alias erzeugt, um auf die Variable zuzugreifen.

[1] `http://search.cpan.org/~ferreira/Exporter-5.62c/lib/Exporter.pm`

Lokale Variablen

Eine solche Variable hat nur in dem Bereich Gültigkeit in dem sie deklariert wurde. Zu diesem Zweck werden überwiegend lexikalisch gebundene Variablen eingesetzt. Theoretisch könnte man zwar auch mit our eine lokale Variable erstellen. Der Übersetzer beschränkt, ebenso wie bei my die Gültigkeit des Variablennamens auf den aktuellen Block. Am Ende des Blocks müsste die Variable vom Programm aber explizit wieder aus dem Namensraum gelöscht werden. Dieser Aufwand wird nur selten notwendig sein.

Lexikalische Variablen in einem inneren Block überdecken gleichnamige Variablen eines umschließenden Blocks.

Lokale Wertänderung einer globalen Variable

Zu diesem Zweck wird das Schlüsselwort `local` verwendet. Wichtig ist, dass mit einer Wertzuweisung ein neuer Wert bekannt gemacht werden muss, solange der vorgegebene Wert, meist oder immer `undef` & (), nicht schon das Gewünschte ist.

Reguläre Ausdrücke

Kapitel 20

Reguläre Ausdrücke

Einleitung

Als erstes stellt sich die Frage, was Reguläre Ausdrücke (kurz: regex) sind und was man mit ihnen machen kann. Zum anderen warum sie gerade in einem Perl-Buch erklärt werden.

Reguläre Ausdrücke sind eine Art Mini-Programme. Sie wurden entwickelt, um das Arbeiten mit Texten komfortabler und leichter zu gestalten. Reguläre Ausdrücke sind vergleichbar mit ||Mustern|| oder ||Schablonen||, die auf einen oder mehrere unterschiedliche Strings passen können. Diese Muster können entweder auf einen String passen oder nicht passen.

Hierbei existieren zwei Kategorien von Regulären Ausdrücken. Einmal eine normale Mustersuche, zum anderen eine ||Suche und Ersetze||-Funktion. Die normale Mustersuche wird zum einen darin verwendet, um ganze Eingaben oder Strings zu überprüfen oder einzelne Informationen aus einen String auszulesen. Die Suchen-und-Ersetzen-Funktion hat dabei eine ähnliche Funktion, wie Sie es von grafischen Texteditoren gewohnt sind, nur sind diese in Perl deutlich mächtiger.

Die englischen Begriffe für die Muster- und ||Suche & Ersetze||-Funktion sind dabei: ||Pattern matching|| und ||Substitution||. Diese Begriffe sollte man kennen, da sie oft benutzt werden. Diese werden hier ebenfalls im weiteren Verlauf Verwendung finden.

Zum anderen stellt sich immer noch die Frage, was das Ganze mit Perl zu tun hat. Reguläre Ausdrücke werden deshalb behandelt, da sie ein zentraler Bestandteil von Perl sind. Sie werden in Ihrer Perl-Karriere wohl kein Perl-Programm finden, in dem nicht mindestens ein Regulärer Ausdruck vorkommt. Reguläre Ausdrücke

sind dabei so fest in Perl verankert und wurden im Laufe der Zeit so stark ausgebaut, dass dieses viele andere Programmiersprachen kopierten. Sie werden Reguläre Ausdrücke in Sprachen wie: PHP, Java, C#, Python, JavaScript, Tcl, Ruby und noch vielen weiteren Programmiersprachen finden. Dabei wird diese Erweiterung meist als "Perl-kompatibel" beschrieben. Jedoch sind Reguläre Ausdrücke in keiner dieser Sprachen so fest implementiert und so leicht anzuwenden wie in Perl.

Man kann schon fast sagen, dass Reguläre Ausdrücke erst durch Perl ihre heutige Funktionalität bekommen haben.

Dabei reißt der Artikel die Möglichkeiten von diesen Ausdrücken nur an.

Der Aufbau der beiden genannten Typen sieht dabei folgendermaßen aus:

Pattern Matching:

m/Regex/Optionen

Substitution:

s/Regex/String/Optionen

Begrenzer Wie man am Pattern Matching sowie der Substitution erkennen kann, trennen die Slashes "/" die einzelnen Teile eines Regulären Ausdruckes. Allerdings haben Sie bei Perl die Wahl, jedes beliebige Sonderzeichen als Begrenzer zu wählen. Die Vorteile davon werden Sie zu schätzen wissen, wenn Sie praktische Erfahrung mit Regulären Ausdrücken sammeln. Um es kurz vorweg zu sagen: Sie haben damit die Wahl ein Zeichen zu wählen das nicht in Ihrer Regex vorkommt, und Sie können sich somit das Escapen der Zeichen sparen.

Perl weiß anhand des ersten Zeichen, das nach dem "m" respektive "s" folgt, welcher Begrenzer gewählt wurde. Es sind also auch folgende Schreibweisen erlaubt:

m!regex! s#Regex#String#Optionen s$Regex$String$Optionen s"Regex"String"Optionen ...

Eine spezielle Regel gilt für Zeichen, die ein öffnendes sowie schließendes Zeichen besitzen. Denn dort müssen die einzelnen Teile eingeklammert werden. Dies schaut folgendermaßen aus:

m(Regex) s(Regex)(String)Optionen s{Regex}{String}Optionen sOptionen s[Regex][String]Optionen

Wie man sieht werden hier unterschiedliche Anfangs- sowie Endzeichen verwendet. Eine weitere Eigenschaft ist, dass nur das wirkliche Endzeichen die Regex re-

spektive den String einklammert. Im folgenden Praktischen Beispiel ist also auch folgendes möglich:

s((H)allo)(Welt)i

Wenn Sie etwas Erfahrung mit Regulären Ausdrücken haben, werden Sie sehen, dass diese Regex keinen Sinn ergibt. Allerdings dient das lediglich zur Veranschaulichung, dass hier wirklich ||(H)allo|| als Regex erkannt wird, und nicht ||(H|| wie man annehmen könnte.

Wenn Sie noch nicht verstehen, was hier passiert, dann ist dies nicht weiter schlimm, wir werden später darauf zurück kommen.

Bindungsoperator Wie die Form eines Regulären Ausdrucks ausschaut, wissen Sie. Jedoch wissen Sie noch nicht, wie man diesen auf einen String anwendet. Hierfür existiert der sogenannte Bindungsoperator. Um einen String mit einer Regex zu verbinden, egal ob nun ||Pattern Matching|| oder ||Substitution||, schreiben Sie einfach folgendes:

```
$text =~ m/Regex/; $text =~ s/Regex/String/;
```

Dieser Bindungsoperator prüft im ersten Fall, ob in $text das angegebene Muster auf der rechten Seite vorkommt. Zur Substitution kommen wir noch später; allerdings sei hierzu gesagt, dass jedes Vorkommen der angegebenen Regex in $text durch String ersetzt wird.

Weiterhin besitzt dieser Ausdruck einen Rückgabewert. Wenn das Muster, das in Regex angegeben wird, wirklich in $text passt, dann wird ||true|| zurück gegeben. Das gleiche gilt für die Substitution. Wenn die Regex auf $text gepasst hat, wurde eine Ersetzung durchgeführt und es wird der Wert ||true|| zurück gegeben.

Sollte die Regex in beiden Fällen nicht auf den String in $text gepasst haben, dann wird ||false|| als Wert zurück geliefert. Bei der Substitution hat dies noch die Auswirkung, dass eine Ersetzung nicht stattgefunden hat.

Grundlegendes zu Regulären Ausdrücken

In diesem Kapitel, erfahren Sie Grundlegende Einzelheiten wie Reguläre Ausdrücke Funktionieren. Sie werden einzelne Stolpersteine kennen lernen, und lernen sie zu vermeiden. Nach diesem Kapitel sollten Sie in der Lage sein Reguläre Ausdrücke auf beliebige Strings anzuwenden, und nur das zu finden, zu ersetzen oder die Informationen zu extrahieren die Sie wollen.

Pattern Matching Mit dem jetzigen Wissen wollen wir ein Pattern Matching ausführen. Wir wollen nach einem bestimmten Wort in einem String suchen, und sofern gefunden, wollen wir dies dem Benutzer mitteilen.

Ein konkretes Beispiel:

```
$string = ||Wir mögen Kamele, auch wenn sie übel riechen||; if
($string =~ m/Kamel/) { print ||Kamel gefunden!||; } else { print
||Schade, kein Kamel weit und breit.||; }
```

Die Ausgabe des Perl-Scripts wäre ||Kamel gefunden||. Gehen wir die einzelnen Zeilen zusammen durch. In Zeile 1 wird ein Skalar mit dem Text ||Wir mögen Kamele, auch wenn sie übel riechen|| definiert. Zeile 2 ist nun neu für uns. Hier wird der Rückgabewert von unserem Ausdruck ||$string =~ m/Kamel/|| in einer bedingten Anweisung (if) ausgewertet. Wie wir wissen, wird ||true|| zurück gegeben, wenn unser Suchmuster in $string vorkommt. Wenn unser Suchmuster nicht vorkommt, wird ||false|| als Wert zurück gegeben. Dies hat direkte Auswirkung auf unsere if-Kontrollstruktur. Wenn unser Muster gefunden wird, dann wird ||true|| zurück gegeben und dadurch Zeile 3 ausgeführt. Wenn das Muster nicht in $string vorkommt, wird ||false|| zurück gegeben und Zeile 5 ausgeführt.

Aber warum finden wir den überhaupt unser ||Kamel||? Im String steht doch ||Kamele||. Dies ist zwar richtig, aber Perl interessiert sich nicht für Wörter. Perl liest unser Muster folgendermaßen: Ein ||K||, gefolgt von ||a||, gefolgt von ||m||, gefolgt von ||e||, gefolgt von ||l||. Unser komplettes Muster wird also an folgender unterstrichener Stelle gefunden.

Wir mögen Kamele, auch wenn sie übel riechen

An diesem Punkt wird unsere Suche auch sofort abgebrochen. Was danach kommt ist unwichtig. Unsere Regex hat auf das Muster gepasst und es wird als Rückgabewert ||true|| zurück gegeben.

Hier haben wir 2 Sachen gelernt:

- Die Mustersuche erkennt lediglich ein Zeichen nach dem anderen, ohne einen höheren Zusammenhang, also z.B. ein Wort daraus Bilden zu können.
- Die Mustersuche wird sofort beendet sobald unsere Regex vollständig passt.

Zwei wichtige Punkte, auf die Sie später immer wieder stoßen werden. Wissen Sie, was ausgegeben werden würde, wenn Sie den Ausdruck in der oberen if-Kontrollstruktur folgendermaßen anpassen würden?

```
$string =~ m/kamel/
```

Richtig! Es würde ‖Schade, kein Kamel weit und breit.‖ ausgegeben werden! Warum? Perl interpretiert die Mustersuche folgendermaßen: Ein kleines ‖k‖, gefolgt von einem kleinen ‖a‖, gefolgt von einem kleinen ‖m‖ ...

Also nochmals zur Verdeutlichung. Es werden wirklich nur Folgen von Zeichen erkannt, ohne diese zu Interpretieren. Ein großes ‖K‖ ist ein anderes Zeichen als ein kleines ‖k‖ und wird folglich nicht gefunden. Da im ganzen $string kein ‖k‖ gefolgt von ‖a‖ ... vorkommt wird letztendlich ‖false‖ als Wert zurück gegeben, und unser ‖else‖-Block wird ausgeführt.

Allerdings brauchen Sie keine Angst haben, dass Sie jetzt jede Schreibweise von ‖Kamel‖ (‖kAmEl‖, ‖KAMel‖, ...) überprüfen müssen. Es gibt bestimmte Optionen die Sie aktivieren können, wodurch eine Groß- und Kleinschreibung nicht unterschieden wird.

Substitution Bevor wir uns nun mit den Optionen befassen, wollen wir eine Substitution durchführen. Hierfür passen wir das vorherige Beispiel etwas an:

```
$string = ‖Wir mögen Kamele, auch wenn Kamele übel riechen‖;
$string =~ s/Kamel/Lama/; print $string;
```

In Zeile 1 wird ein String initialisiert, danach führen wir eine Substitution auf $string aus. Dort wollen wir ‖Kamel‖ durch ‖Lama‖ ersetzen. Unser Ergebnis wird danach zurück gegeben.

Wissen Sie bereits was zurück gegeben wird? Denken Sie erst darüber nach, bevor Sie sich die Lösung anschauen:

Wir mögen Lamae, auch wenn Kamele übel riechen

Überrascht? Gehen wir die Substitution einzeln durch. Unsere Substitution sagt das wir das vorkommen von einem großen ‖K‖ gefolgt von einem kleinen ‖a‖ ... durch ‖Lama‖ ersetzen wollen. Wir müssen also jedes vorkommen von ‖Kamel‖ erst einmal finden, und es danach durch ‖Lama‖ ersetzen. Damit wir unser ‖Kamel‖ überhaupt finden, beginnt unsere Substitution an der ersten Stelle unseres Strings. Hier wird überprüft ob das ‖K‖ auf das ‖W‖ von ‖Wir‖ passt. Das passt natürlich nicht, also springen wir ein Zeichen weiter. Es wird nun überprüft ob das ‖i‖ auf das ‖K‖ passt. Dies passt natürlich nicht, und unsere Substitution geht Solange den String weiter bis es auf das erste ‖K‖ von ‖Kamel‖ passt. Jetzt ist gefordert, dass ein ‖a‖ folgt. Dies kommt wirklich vor, und die Substitution springt ein Zeichen weiter, solange bis unser ganzes ‖Kamel‖ erkannt wurde. Nun ist unsere Regex erfolgreich erkannt worden, und unser ‖Kamel‖ wird durch ‖Lama‖ ersetzt. Wichtig ist, dass unser folgendes ‖e‖ von ‖Kamele‖ nicht erkannt wur-

de, da wir nicht wissen wie wir Wörter erkennen können (jedenfalls noch nicht). Die Ersetzung wird also durchgeführt, und unsere Substitution war erfolgreich. Es wird ||true|| als Rückgabewert zurück geliefert, und die Substitution beendet. Allerdings wird der Rückgabewert ||true|| verworfen. Unsere Substitution dringt erst gar nicht bis zum zweiten ||Kamele|| vor, und ersetzt dieses auch nicht, da schon lange vorher unsere Substitution erfolgreich war.

Hier lernen wir also wieder einige Neuigkeiten:

- Ein String wird von Links nach rechts durchgearbeitet.
- Es wird immer der frühestmöglichste Treffer links gefunden.

Um jetzt jedes Vorkommen vom ||Kamel|| durch ||Lama|| zu ersetzen, könnten wir folgendes schreiben:

```
$string = ||Wir mögen Kamele, auch wenn Kamele übel riechen||;
while ($string =~ s/Kamel/Lama/) {} print $string;
```

Wenn also unser ||Kamel|| gefunden und erfolgreich ersetzt wurde, wird als Rückgabewert ||true|| geliefert. Dadurch wird unser Schleifeninhalt durchgeführt, der allerdings Leer ist. Danach wird erneuert unsere Bedingung durchgeführt. Zu beachten ist, dass unsere Substitution wieder ganz von vorne beginnt, also an der ersten Stelle im String anfängt, und nicht an der Stelle, an der wir zuletzt etwas verändert haben. Manchmal ist dieses Verhalten gewünscht, aber in den seltesten Fällen ist dies der Fall, und es kann hierbei zu Problemen kommen. Stellen Sie sich folgendes while Konstrukt vor:

```
while ( $string =~ s/e/ee/ ) {}
```

Hiermit hätten wir eine Endlossschleife gebaut. Das erste vorkommen eines ||e|| wird durch ein ||ee|| ersetzt. Der Rückgabewert ist true und unsere Substitution beginnt wieder von vorne. Jetzt sind zwei ||e|| an der Stelle vorhanden, und dort ersetzen wir wieder das erste ||e|| durch ||ee||. Es existieren also schon 3 ||e|| an dieser Stelle. Dieses wird jetzt unendlich oft durchgeführt. Jedenfalls solange bis Perl abstürzt, weil Sie nicht genügend RAM haben um so einen großen String zu speichern.

Sie sollten also aufpassen bei solch einer Verwendung. Es gibt zwar eine Option, die unserem gewünschten Verhalten entspricht und bei der letzten Substitution weiter macht. Aber manchmal benötigen wir das obrige Verhalten.

Optionen Optionen dienen dazu das Verhalten unserer Regex zu verändern. Mit ihnen sind Sachen möglich, die so ohne weiteres nicht möglich wären. Jede Opti-

on kann dabei unsere Regex grundlegend verändern. Die Optionen werden hierbei hinter den Regulären Ausdruck angehängt. Es ist auch möglich mehrere Optionen zu benutzen, die Reihenfolge der Optionen spielt hierbei keine Rolle. Den grundlegenden Aufbau sehen Sie in der Einleitung, hier aber nochmals ein paar praktische Beispiele:

```
m/kamel/i m/k a m e l/xi s/kamel/lama/ig s/k a m e l/igx
```

Einige wichtige Optionen sollen hierbei bereits erläutert werden, andere wenn sie wichtig werden:

Option: /i Mit der Option ‖i‖ *ignore-case* können wir Perl dazu veranlassen, dass zwischen Groß- und Kleinschreibung nicht mehr unterschieden wird. Folgendes Muster:

```
m/kamel/i
```

Würde also auch auf ‖KAMEL‖, ‖KAmel‖, ‖KaMeL‖, ... passen.

Option: /g Im Kapitel ‖Substitution‖ wollten wir, dass jedes Vorkommen von ‖Kamel‖ durch ‖Lama‖ ersetzt wird, bei der folgenden Lösung mit der while-Schleife funktionierte das zwar, jedoch könnte dieses unter Umständen zu neuen Problemen führen. Das ‖g‖ steht hier für *global* und es möchte damit ausdrücken, dass wir den ganzen String durcharbeiten. Das Verhalten von ‖/g‖ ist folgendermaßen zu erklären, dass die Substitution nach einer Ersetzung nicht beendet wird, sondern an der letzten Position weiter macht, und nach weiteren Treffern sucht. Der Rückgabewert ist hierbei die Anzahl von Substitutionen, die innerhalb des Strings durchgeführt wurden. Wenn also mindestens eine Substitution durchgeführt wurde, ist der Rückgabewert automatisch ‖> 0‖ und somit ein ‖wahrer‖ Wert.

Hierbei ist zu beachten, dass die Option nur Auswirkung auf eine Substitution hat. Wir können z.B. nicht mit:

```
$string =~ m/e/g;
```

Die Anzahl der ‖e‖ Buchstaben innerhalb eines Strings zählen.

Dadurch, dass unsere Substitution jedoch nicht immer wieder von vorne anfängt, können wir das letzte Problem im Kapitel ‖Substitution‖ lösen.

```
$string = ‖Wir mögen Kamele, auch wenn Kamele übel riechen‖;
$string =~ s/e/ee/g; print $string;
```

Dies würde nun nicht mehr in eine Endlosschleife enden, sondern folgenden String zurück geben:

Wir mögeen Kameelee, auch weenn Kameelee übeel rieecheen

Option: /x Normalerweise werden innerhalb einer Regex Whitespace-Zeichen als zu dem Muster gehörend angesehen.

```
m/a b/
```

Diese Regex würde auf ein ‖a‖ gefolgt von einem Leerzeichen gefolgt von einem ‖b‖ passen. Mit dieser Option verliert das Leerzeichen seine Bedeutung, und wir würden ein ‖a‖ gefolgt von einem ‖b‖ suchen.

Unter Whitespace-Zeichen versteht man alle Zeichen, die nicht direkt etwas auf dem Bildschirm Zeichnen. Dies sind zum Beispiel Leerzeichen, Tabulatoren, Newlines und Formfeeds.

Wahrscheinlich werden Sie den Sinn dahinter noch nicht nachvollziehen können. Für die sehr kleinen Regexe, die wir bisher geschrieben haben, ist dieses auch unbrauchbar. Allerdings werden wir später auf sehr viel komplexere Regexe stoßen. Damit diese besser lesbar sind, wurde diese Option implementiert. Damit kann man eine Regex über mehrere Zeilen verteilen und ihnen sogar Kommentare geben.

Zusammenfassung

- ‖i‖, *case-insensitive*: Groß- und Kleinschreibung wird ignoriert
- ‖g‖, *global*: Es werden alle Stellen gesucht, die passen. Die Funktion ist eher für das Ersetzen mit regulären Ausdrücken interessant.
- ‖m‖, *multi-line*: Verändert die Bedeutung des $-Zeichen (siehe Sonderzeichen), damit es an der Position vor einem ‖\n‖ passt.
- ‖s‖, *single-line*: Verändert die Bedeutung des .-Zeichen (siehe Sonderzeichen), damit es auch auf einem ‖\n‖ passt.
- ‖x‖, *extended*: Alle Whitespace-Zeichen innerhalb des Regulären Ausdrucks verlieren ihre Bedeutung. Dadurch kann man Reguläre Ausdrücke optisch besser aufbereiten und über mehrere Zeilen schreiben.
- ‖o‖, *compile once*: Ausdruck wird nur einmal kompiliert und kann nicht mehr verändert werden.

- ||e||, *evaluate*: Das Ersetzmuster wird als Perl-Code interpretiert und ausgeführt.

Quantoren Quantoren sind eine Möglichkeit auszudrücken, dass etwas bestimmt oft nacheinander folgt. Hierbei kann man die Anzahl selber bestimmen, oder man benutzt bereits vordefinierte Quantoren. Diese vordefinierten Quantoren wurden eingebaut, weil diese besonders oft vorgekommen sind. Sie werden durch Sonderzeichen definiert. Quantoren beziehen sich immer auf die Einheit die vor ihnen steht. Dies kann dabei ein einfacher Buchstabe, ein Zahl oder aber auch eine Gruppierung von Zeichen sein.

Stellen Sie sich vor, Sie möchten eine Folge von 15 ||a|| hintereinander erkennen. Sie könnten in Ihrer Regex das a nun 15mal schreiben, allerdings können Sie sich dabei leicht verzählen. Genauso wird es schwieriger Ihren Code zu Lesen. Daher wäre die Benutzung eines Quantors ideal.

```
m/a{15}b/i
```

Wie man erkennen kann, wird ein Quantor durch geschweifte Klammern definiert. Da sich der Quantor auf die vorherige Einheit bezieht, wird hier gesagt, dass das ||a|| 15 mal vorkommen muss. Der Quantor bezieht sich hier nicht auf das ||b||, wie man annehmen könnte. Was wäre aber, wenn wir nun eine Mindestgrenze und eine Maximalgrenze angeben wollen? Dies wäre auch kein Problem.

```
m/a{10,15}b/i
```

Hiermit sagen wir, dass das ||a|| mindestens 10 mal vorkommen muss und maximal 15 mal vorkommen darf. Danach muss ein ||b|| folgen. Nun wollen wir aber das ein ||a|| nur eine mindestgrenze erfüllen muss, es danach aber unendlich oft folgen darf. Um dies auszudrücken, lassen wir einfach die Maximalgrenze weg.

```
m/a{10,}b/i
```

Wichtig hierbei ist, dass das Komma stehen bleiben muss. Würden wir das Komma weg lassen, hätten wir eine genaue Angabe wie oft das ||a|| vorkommen muss. In diesem Beispiel muss das ||a|| nun mindestens 10 mal hintereinander vorkommen, darf aber auch unendlich oft vorkommen. Das Gleiche können wir auch für die Maximalangabe machen.

```
m/a{,10}b/i
```

Hiermit würde wir ein ||a|| maximal 10 mal finden, allerdings darf es auch kein einziges mal vorkommen.

Quantor: ? Dieser Quantor ist identisch mit {0,1}. Er drückt aus, dass das vorherige optional ist, d.h. 1-mal oder keinmal vorkommen darf.

```
m/\.html?/i
```

Dies würde z.B. auf die Dateiendung ||.htm|| sowie auch ||.html|| passen. Der Punkt muss hier durch ein ||Backslash|| escaped werden, da der Punkt eine besondere Bedeutung hat. Genauso wie das Fragezeichen. Wollen Sie nicht die besondere Bedeutung des Fragezeichen haben, weil Sie nach einem Fragezeichen suchen wollen, müssen Sie dieses auch escapen. Durch ||\?|| verliert das Fragezeichen seine Bedeutung als Quantor. Dies gilt ebenfalls für alle anderen Sonderzeichen.

Quantor: * Dieser Quantor ist identisch mit {0,}. Er drückt aus, dass das vorherige beliebig häufig vorkommen darf, also auch keinmal. Dieser Quantor wird erst interessant mit sogenannten Zeichenklassen.

```
m/\d*/i
```

Hier greife ich etwas vorweg. Das ||\d|| steht dabei für eine beliebe Ziffer von 0-9. Wenn wir den Stern-Quantor auf das ||\d|| anwenden, dann finden wir eine Folge von Ziffern. Allerdings muss keine Ziffer an dieser Stelle stehen, denn keinmal ist ja ebenfalls erlaubt. Bei der Verwendung des Sterns müssen Sie aufpassen. Nicht immer ist es das, was Sie wollen. Er würde auch auf ||zweiundvierzig|| passen, da es nunmal auch erlaubt ist, wenn keine Ziffer vorkommt.

Quantor: + Der Quantor ist identisch mit {1,}. Er drückt aus, dass das vorherige mindestens einmal vorkommen muss, aber unendlich oft folgen darf. Wie auch der Stern-Quantor ist dieser erst mit einer Zeichenklassen interessant.

```
m/\d+/i
```

Meistens ist es das, was Sie wollen. Es erkennt eine beliebige Anzahl von Ziffern. Dabei muss jedoch mindestens eine Ziffer vorkommen.

Gierigkeit Vielleicht haben Sie sich bei dem ||?||-Quantor gefragt, was nun als erstes erkannt wird. Das ||htm|| oder das ||html||. Alle Quantoren die hier vorgestellt wurden sind gierig. Das bedeutet, dass, wenn sie die Wahl haben ein Zeichen zu erkennen oder nicht zu erkennen, sie es immer zuerst versuchen werden, es zu erkennen.

Bei dem Beispiel mit dem ||?||-Quantor würde zuerst das ||.htm|| erkannt werden. Das ||l|| ist Optional, da es aber gierig ist, schaut es nach, ob das ||l|| wirklich

danach kommt, und wird es ebenfalls aufnehmen wenn es folgt. Wenn es nicht folgen sollte, ist unsere Regex ebenfalls erfüllt, da es auch erlaubt war, wenn das ||1|| nicht auf ||.htm|| folgt.

Zeichenklasse Manchmal möchten Sie, dass an einer bestimmten Stelle eine Auswahl unterschiedlicher Zeichen stehen kann. Für dieses Problem gibt es die sogenannten Zeichenklassen. Eine Zeichenklasse wird mit eckigen Klammern umgeben. Hierbei ist zu beachten, dass die komplette Zeichenklasse für ein einziges Zeichen steht. Innerhalb der Zeichenklasse wird definiert, auf welches Zeichen die Zeichenklasse passen darf.

```
m/[234][12345]/
```

Dieses Matching würde auf eine zweistellige Zahl passen. Dabei muss die erste Zahl 2, 3 oder 4 sein, und die zweite Zahl 1, 2, 3, 4 oder 5. 50 oder 20 würden z.B. nicht gefunden werden.

Zeichenklassen-Metazeichen Innerhalb von Zeichenklassen ist der Bindestrich als sogenanntes Zeichenklassen-Metazeichen erlaubt. Hiermit können wir ganze Bereiche von Zeichen angeben. Wollen wir z.B. alle Zahlen erkennen, können wir das auf zwei Arten tun.

[0123456789] [0-9]

Diese beiden Zeichenklassen erkennen genau das gleiche. Bei Buchstaben wird der Vorteil offensichtlicher:

[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
[a-zA-Z]

Möchten wir ein Bindestrich innerhalb einer Zeichenklasse erkennen, dann müssen wir diesen als erstes Zeichen schreiben.

[-a-zA-Z]

Dies würde auf alle Zeichen inklusive des Bindestrichen passen.

Negative Zeichenklasse Weiterhin gibt es eine negative Zeichenklasse. Dort können wir definieren, auf welche Zeichen unsere Zeichenklasse nicht passen darf. Möchten wir eine negative Zeichenklasse verwenden, dann muss das erste Zeichen innerhalb der Zeichenklasse ein Zirkumflex ($\hat{}$) sein. Wenn das Zirkumflex nicht an erster Stelle steht, wird es als normales Zeichen erkannt.

Wenn also jedes beliebige Zeichen außer einer Zahl erkannt werden soll, können wir folgendes schreiben:

[ô-9]

Zeichenklassen Dieser Abschnitt ist sehr verwandt mit der Zeichenklasse. Während wir bei der Zeichenklasse unsere Zeichen wählen konnten, können wir dies bei den Zeichenklassen nicht mehr. Diese Zeichenklassen dienen lediglich als Abkürzung für eine bestimmte Zeichenklasse:

- \d = *digit* Erkennt Ziffern. Ist das gleiche wie **[0123456789]**
- \D = Erkennt alles andere außer Ziffern. Ist das gleiche wie **[ô0123456789]**
- \s = Erkennt jegliche Art von Whitespace-Zeichen: Leerzeichen, Tabulatoren, Newlines, Formfeed ...
- \S = Erkennt alles außer Whitespace-Zeichen.
- \w = *word* Erkennt Ziffern, Buchstaben und Unterstrich. Ist das gleiche wie **[a-zA-Z0-9_]**
- \W = Erkennt alles außer einem Wortzeichen.
- . = *Punkt* Steht für jedes Zeichen außer Newline. Ist das gleiche wie **[ê\n]**

Alternation Unter Alternation verstehen wir dass entweder das eine, oder das andere passen kann. Es ist also eine ||Oder||-Verknüpfung. Eine Alternation schreiben wir als ||\||| (Pipe) Zeichen. Hierbei wird dann entweder das genommen was links von diesem Zeichen steht, oder was rechts von diesem Zeichen steht.

Pattern Matching Bei der Alternation innerhalb des Pattern Matching gibt es wenig zu beachten. Stellen Sie sich vor, Sie schreiben ein Programm und möchten, dass sich das Programm nach diversen Eingaben des Benutzers beendet. Sie könnten folgendes Schreiben

```
$input = ; if ( $input =~ m/q|quit|exit/i ) { exit; } print
||Hallo, Welt!\n||;
```

Dieses Programm wartet auf eine Eingabe des Benutzers. Würden wir ||q||, ||quit|| oder ||exit|| eingeben, dann würde sich unser Programm beenden. Bei allem anderen würden wir die Meldung ||Hallo, Welt!|| auf dem Bildschirm sehen.

Substitution Innerhalb einer Substitution gibt es einige Punkte die wir beachten müssen. Stellen Sie sich vor, Sie möchten jedes vorkommen von ||q|| oder ||quit|| durch ||exit|| ersetzen. Vielleicht würden Sie so etwas schreiben:

```
s/q|quit/exit/ig;
```

Dies funktioniert aber nicht richtig. Sollte wirklich ||quit|| im String vorkommen auf dem Sie diese Regex anwenden, dann würde folgendes dabei raus kommen:

exituit

Um die genaue Vorgehensweise zu verstehen, müssen Sie wissen wie Perl eine Alternation behandelt. Es wird wieder Zeichen für Zeichen überprüft. Dabei wird aber auch die Reihenfolge der Alternation beachtet. Es wird zuerst der Ausdruck ganz links überprüft und erst wenn dieser nicht passt, wird der nächste Ausdruck überprüft. Sollte ein Ausdruck passen, wird sofort die Substitution ausgeführt. Bei ||quit|| passt das ||q|| sofort und es würde hier eine Substitution mit ||exit|| statt finden. Dadurch würde ein ||quit|| nie im Text ersetzt werden, da wir schon vorher das ||q|| durch ||exit|| ersetzt haben. Wir müssen also die Reihenfolge vertauschen:

```
s/quit|q/exit/ig;
```

Merke:

- Bei der Alternation ist die Reihenfolge wichtig

Dieser Punkt gilt auch für das Pattern Matching, allerdings hängt es von der Verwendung ab, ob wir die Reihenfolge anpassen müssen. Im obrigen Beispiel ist es egal wodurch unser Programm beendet wird. Würden wir aber Informationen auslesen, kann es sehr wohl von Bedeutung werden, welche Reihenfolge wir in der Alternation gewählt haben.

Sonderzeichen Für die Suchmuster stehen diverse Sonderzeichen zur Verfügung. Diese können dann beispielsweise für beliebige Zeichen oder für das Zeilenende stehen.

- . steht für *ein* beliebiges Zeichen außer dem ||\n||.
- ? steht für das ein- oder nullmalige Vorkommen des vorangegangenen Zeichens oder Gruppierung. Folgt das Fragezeichen auf einen Quantor dann wird dieser zu einem nicht gierigen Quantor.
- + steht für das ein- oder mehrmalige Vorkommen des vorangegangenen Zeichen oder Gruppierung.

- * steht für das null- oder mehrmalige Vorkommen des vorangegangenen Zeichen oder Gruppierung.
- ŝteht für den Beginn einer Zeile.
- $ steht für das Ende eines Strings. Wenn die /m Option benutzt wird, wird die Bedeutung so verändert das es für ein ||\n|| Zeichen steht.
- \A steht für den Beginn eines Strings.
- \Z steht immer für das Ende eines Strings, unabhängig ob die Option /m verwendet wurde.
- \- das nächste Zeichen wird ohne Funktion interpretiert. Um einen Punkt zu suchen muss ||\.|| benutzt werden, sonst wird ein beliebiges Zeichen gefunden.
- [] wird Zeichenklasse genannt und steht für ein angegebenes Zeichen. [dhb]ot würde ||dot||, ||hot|| oder ||bot|| finden.
- () 1. Funktion: Gruppiert Ausdrücke. 2.Funktion: Speichert den tatsächlichen Wert, der an dieser Stelle gefunden wurde, in einer Variable.
- | steht für das logische ODER. ||ist|war|| gibt sowohl true, wenn ||ist|| im Ausdruck vorkommt, als auch wenn ||war|| im Ausdruck enthalten ist.

Zeichen ersetzen

(todo)

Zeichen ersetzen *ohne* Reguläre Ausdrücke

Geht es nur darum, Zeichen durch andere zu ersetzen, sind Reguläre Ausdrücke häufig 'überqualifiziert', da es mit `tr//` eine deutlich einfachere und performantere Alternative gibt.

```
$string =~ tr/SUCHEN/ERSETZEN/Optionen
```

Einige nützliche Beispiele:

```
$string =~ tr/A-Z/a-z/; # ersetzt alle Großbuchstaben durch
Kleinbuchstaben $string =~ tr/+/ /; # ersetzt das + durch ein
Leerzeichen
```

Mit `tr//` gibt es auch die Möglichkeit Zeichen in einem String zu zählen. Dazu wird die Liste der ersetzenden Zeichen einfach leergelassen. Normalerweise geht das nicht weil `tr//` für jedes zu suchende Zeichen passenden Ersatz braucht. So aber ist es ideal die zu suchenden Zeichen einfach nur zu zählen:

```
# Zählt die Vokale im String $anzahlVokale = ($string =~
tr/AEIOUaeiou//);
```

Objektorientiert Programmieren in Perl

Kapitel 21

Objektorientiert Programmieren in Perl

Deklaration einer Klasse in Perl

Klassen werden in Perl nicht deklariert, sondern es werden nur Methoden einer Klasse definiert. Wenn eine Methode definiert wird, ordnet der Compiler die Methode dem aktuellen Paket zu. Ein Paket wird durch die Funktion `package` eingeleitet.

```
package MeineKlasse;
```

Kapselung

Eigenschaft

Eigenschaften gehören zu einer Instanz, und nicht zu einer Klasse.

Erstellung und Aufruf einer Methode in Perl

```
package MeineKlasse;

# Methode definieren sub meineklassenmethode { print "Die
Ausführung meiner Klassenmethode ist gelungen!!"; }

# Methode aufrufen MeineKlasse::meineklassenmethode();
```

Konstruktor-Methoden in Perl Ein Konstruktor ist eine Methode, die dabei mithilft, eine Instanz einer Klasse zu erzeugen. In Perl gibt es keine zwingenden Vorschriften, wie diese benannt wird. Gewöhnlich wird diese ||new|| genannt, und wir empfehlen, diesen Gebrauch beizubehalten.

```
package MeineKlasse;

# Konstruktor definieren sub new { my $invocant=shift;

# wurde Konstruktor von Instanz- oder # von der Klasse
aufgerufen? my $class=ref($invocant) || $invocant;

# Instanz ist eine Hashreferenz my $self={};

# Hashreferenz zur Instanz machen bless $self, $class; print
||Die Instanz wurde erzeugt!!||;

# Instanz zurückgeben return $self }
```

Die Destruktor-Methode in Perl Für den Destruktor ist der Methodenname `DESTROY` zwingend vorgegeben. Der Destruktor ist aber nur notwendig, wenn bei der Auflösung einer Instanz noch besondere Anweisungen ausgeführt werden sollen.

Der Destruktor wird aufgerufen, sobald der *garbage collector* eine nicht mehr referenzierte Instanz gefunden hat und ihren Speicher freigeben möchte. Möchte man vor der Freigabe noch Handlungen durchführen, wie zum Beispiel das Schließen von Dateien oder das Kappen von Datenbankverbindungen, so muss man dies im Destruktor tun.

Die Reihenfolge, in der Destruktoren aufgerufen werden, hat nichts mit der Reihenfolge zu tun, in der die Instanzen nicht mehr refenziert werden. Auf die Reihenfolge des Destruktoraufrufs kann und soll sich der Programmierer nicht verlassen.

Erzeugung einer Instanz

Mit Hilfe des Konstruktors ist es möglich, eine Instanz einer Klasse zu erzeugen:

Wertebelegung von Instanzeigenschaften

privat und öffentlich

Vererbung in Perl

```
{ package meineunterklasse; @ISA=(||meineklasse||); }
```

`@ISA` ist kein Skalar, sondern ein Array. Damit ist sichergestellt, dass Kindklassen in Perl mehrere Elternklassen haben können.

Polymorphie

Überlagerung einer Klassenmethode in Perl

```
{ package meineunterklasse; @ISA=(||meineklasse||); sub
meineklassenmethode { print ||Die Ausführung meiner überlagerten
Klassenmethode ist gelungen!!||; } }
```

Vordefinierte Variablen

Kapitel 22

Vordefinierte Variablen

Vordefinierte Variablen

Perl bietet eine Reihe von vordefinierten Variablen, die vom Perl-Interpreter automatisch deklariert werden. In den Variablen werden Informationen über die Laufzeitumgebung und das Perl-Skript gespeichert.

Verschiedene Skalarvariablen Um die alternativen Variablennamen nutzen zu können, muss das Modul English importiert werden: `use English;`

$Ô	Hier wird das Betriebssystem angegeben (alternativ: $OSNAME)
$T̂	Gibt - in Unix-Zeitrechnung - den Zeitpunkt, zu dem das Programm gestartet wurde, an (alternativ: $BASETIME)
$Ŵ	Ob der Perl-Interpreter mit der Option -w gestartet wurde (alternativ: $WARNING)
$0	Name des Skripts (alternativ: $PROGRAM_NAME)
$<	Gibt an, unter welcher BenutzerInnen-Kennung (User-ID) das Skript gestartet wurde (alternativ: $UID)

$(	Die Gruppenkennung (Group-ID), unter das Skript ausgeführt wird (alternativ: $GID)
$$	Die Prozess-ID des Programms (alternativ: $PID)

$UID und $GID funktionieren nicht unter Windows!

Umgebungs-Variablen Über den Hash %ENV kann auf die Umgebungsvariablen (*environment*) eines Betriebssystems zugegriffen werden **Aufruf:** `foreach (keys(%ENV)){ print $_. ": " . $ENV{$_} . "\n"; }` **Mögliche Ausgabe unter Windows:**
USERPROFILE: C:\Dokumente und Einstellungen\Test
HOMEDRIVE: C:
TEMP: C:\DOKUME~1\TEST~1\LOKALE~1\Temp
SYSTEMDRIVE: C:
PROCESSOR_REVISION: 2302
SYSTEMROOT: C:\WINDOWS
COMMONPROGRAMFILES: C:\Programme\Gemeinsame Dateien
COMSPEC: C:\WINDOWS\system32\cmd.exe
SESSIONNAME: Console
LOGONSERVER: \\NEO
OSTYPE: cygwin32
APPDATA: C:\Dokumente und Einstellungen\Test\Anwendungsdaten
WINDIR: C:\WINDOWS
PROGRAMFILES: C:\Programme
OS: Windows_NT
CYGNUS_HOME: C:\Programme\cygwin
PROCESSOR_LEVEL: 15
PATHEXT: .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
USERNAME: Test
PROMPT: PG
NUMBER_OF_PROCESSORS: 2
FP_NO_HOST_CHECK: NO
HOMEPATH: \Dokumente und Einstellungen\Test
PROCESSOR_IDENTIFIER: x86 Family 15 Model 35 Stepping 2, AuthenticAMD
PATH: C:\Programme\cygwin\bin;C:\Perl\site\bin;C:\Perl\bin;C:\WINDOWS\system32;
C:\WINDOWS;C:\WINDOWS\System32\Wbem;;C:\PROGRA~1\GEMEIN~1\MUVEET~1\0306
USERDOMAIN: NEO

GPC_EXEC_PREFIX: C:\Programme\cygwin/lib/gcc-lib/
COMPUTERNAME: NEO
ALLUSERSPROFILE: C:\Dokumente und Einstellungen\All Users
PROCESSOR_ARCHITECTURE: x86
TMP: C:\DOKUME~1\TEST~1\LOKALE~1\Temp

Kommandozeilenparameter In der Listenvariablen @ARGV werden die Kommandozeilenparameter, die beim Aufruf angegeben werden, gespeichert.
Aufruf: perl beispiel.p -h

Anwendung:
```
my $arg = shift(@ARGV); if($arg eq "-h"){
#Hilfe-Funktion... }
```

Funktionsvariablen Die beim Aufruf einer Funktion mitgegebenen Variablen werden in der Liste @_gespeichert. Diese ähnelt in der Verwendung der oben beschriebenen @ARGV, welche an das Hauptprogramm mitgegebene Werte enthält

Beispiel
```
sub Addition{ my $zahl1 = shift(@_); my $zahl2 =
shift(@_); print "Die Summe ist" . ($zahl1 + $zahl2); }
```

Fehlermeldungen Die Variable $! wird benutzt, wenn eine Systemfunktion einen Fehler verursacht hat. $! enthält in diesem Fall die Fehlermeldung, die dem Perl-Programm zurückgegeben wurde

```
open(IN,"<nix.txt") or print $!;
```

$_ Die Variable $_wurde bereits im Kapitel Besonderheiten von Perl ausführlich besprochen. Jedesmal, wenn bei Funktionen oder Schleifen keine Variablen angegegeben werden, speichert der Interpreter die jeweiligen Werte in $_-. Mithilfe von $_lässt sich sehr kurzer, allerdings auch sehr unleserlicher Programmcode produzieren! `open $IN, "< text.txt" or die "Fehler: $!"; while(){ print; }`

Dieses Programm liest eine Datei ein und gibt ihren Inhalt zeilenweise aus.
Eine leserlichere Version desselben Programms wäre: `open $IN, "< text.txt" or die "Error: $!"; while(my $input = ){ print $input; }`

Perl-Schnittstellen

Perl/TK

Kapitel 23

TK

Perl/TK

Erstellen einer GUI Zur Nutzung von Perl/Tk wird das Modul Tk benötigt.

```
use Tk;
```

Eine neue Instanz wird mit der Funktion Mainwindow aufgerufen.

```
my $Hauptfenster = new MainWindow;
```

Zur Anzeige und Verwendung wird die Funktion MainLoop verwendet, deren Aufruf einfacher nicht sein kann.

```
MainLoop;
```

Beispiel:

```
#!/usr/bin/perl use Tk; use strict; my $mw = new MainWindow;
$mw->title (||Perl/Tk - Erste Schritte||); $mw->configure
(-width=>||640||,-height=>||480||); MainLoop; #das wars schon
```

Widgets Als Gestaltungselemente grafischer Benutzeroberflächen stellt Tk sogenannte Widgets zur Verfügung. Sie sind der Kern der GUI-Programmierung mit Tk. Im Wesentlichen sind dies:

Das Frame-Widget zur Erzeugung von Containerwidgets. Diese sind unsichtbar, erlauben aber die Gruppierung diverser sichtbarer Widgets.

Das Menu-Widget und das Menubutton-Widget zur Erzeugung von Kopfzeilenmenüs.

Das Label-Widget zur Anzeige von Strings.

Das Entry-Widget zur Eingabe von Werten per Tastatur.

Das Text-Widget zur Anzeige und Bearbeitung von Texten.

Das Button-Widget als Befehlsschaltfläche.

Das Scrolled-Widget zur Erzeugung von Rollbalken.

Das Listbox-Widget zur Darstellung von mehreren Daten in einem Widget(in einer Box).

...

Geometriemanager Um die Positionierung der Widgets zu erleichtern, stellt Tk Geometriemanager zur Verfügung.

pack verwendet die nächstmögliche Position innerhalb der Richtungen top, right, bottom, left.

grid positioniert Widgets innerhalb einer Tabelle anhand der übergebenen Tabellenposition

place gestattet eine Pixelgenaue Zuordnung.

Zu beachten ist, daß die direkt dem Hauptfenster oder einem Frame zugeordneten Widgets nicht mit vermischten Geometriemanagern angeordnet werden.

pack Der Geometriemanager pack erlaubt die Anordnung der Widgets nach Richtungen, top,left und bottom,right.

Ausserdem werden sogenannte Anker gesetzt, die bei einer Grössenänderung des Hauptfenster die Widgets wie gewünscht in Position halten:

pack Optionen:

-side {'left','right','top','bottom'} , wo -anchor {'n','ne','nw','w','sw','s','se','e'} , der Anker

Beispiel:

```
#!/usr/bin/perl use Tk; my $mw = new MainWindow; #configure
mit Fenstergroesse macht bei pack keinen Sinn $mw->title
(||Perl/Tk - Erste Schritte||); my $label1 = $mw->Label
(-text=>||Mein Name||); my $entry1 = $mw->Entry (-width=>60); my
$button1= $mw->Button(-text=>||Exit||,-command=>\& cmd_button);
```

```
$label1->pack (-side=>"left"); $entry1->pack (-side=>"left");
$button1->pack(-side=>"left");

sub cmd_button { $mw->messageBox (-message=>"Progamm wird
beendet!"); exit; } MainLoop;
```

Wenn man nun die Grösse des Fensters verändert, wandern die Widgets in einer Reihe links in der Mitte mit, was nicht immer erwünscht ist, darum werden zur Positionierung meist eingesetzt.

Beispiel mit Ankern:

```
#!/usr/bin/perl use Tk; my $mw = new MainWindow; #configure
mit Fenstergroesse macht bei pack keinen Sinn $mw->title
("Perl/Tk - Erste Schritte"); my $label1 = $mw->Label
(-text=>"Mein Name"); my $entry1 = $mw->Entry (-width=>60);
my $button1= $mw->Button(-text=>"Exit",-command=>\& cmd_-
button); $label1->pack (-side=>"left",-anchor=>'nw');
#Nordwest Ecke $entry1->pack (-side=>"left",-anchor=>'nw');
$button1->pack(-side=>"left",-anchor=>'nw');

sub cmd_button { $mw->messageBox (-message=>"Progamm wird
beendet!"); exit; } MainLoop;
```

Interessant ist nun die Frage wie bekommt man den Exit Button unter das Label und das Eingabefeld (Entry). Dazu stellt das MainWindow Frames zur Verfügung, die eine weitere Untergliederung innerhalb des Fensters erlauben. Es wird nun ein Frame erzeugt und das Label und Entry mittels der Option -in von pack innerhalb des Frames positioniert, der wiederum innerhalb des Hauptfensters angeordnet ist.

=>"Frames können mit den Optionen -width,-height und -bg wie jedes Fenster in Ihrer Groesse und Hintergrundfarbe geändert werde"

pack option:

-in {Frame} Widgets in ein untergeordnetes Fenster einsortien -expand {0,1} 1= Widget nimmt restliche Breite des Fensters ein -fill {'x','y','none','both'} expansion in 'x','y','keine' oder 'beide' Richtung(en)

```
#!/usr/bin/perl use Tk; my $mw = new MainWindow;

my $frame = $mw->Frame();

#configure mit Fenstergroesse macht bei pack keinen
Sinn $mw->title ("Perl/Tk - Erste Schritte"); my
$label1 = $mw->Label (-text=>"Mein Name"); my
```

```
$entry1 = $mw->Entry (-width=>60); my $button1=
$mw->Button(-text=>"Exit",-command=>\& cmd_button);

$frame->pack (-side=>"top");

#using -in $label1->pack (-in=>$frame,-side=>"left",-anchor=>'nw');
#Nordwest Ecke $entry1->pack (-in=>$frame,-side=>"left",-anchor=>'nw');

$button1->pack(-side=>"left",-fill=>"x",-expand=>1);

sub cmd_button { $mw->messageBox (-message=>"Progamm wird
beendet!"); exit; } MainLoop;
```

Jetzt befindet sich der Button unterhalb des Frame mit dem Eingabefeld. Die Kombination der Optionen **-fill** und **-expand** führt dazu, das der Button die komplette Fensterbreite einnimmt.

grid Der Geometriemanager **grid** erlaubt die Anordnung der Widgets tabellarisch in Zellen, beginnend in der linken oberen Ecke mit -row=0 -column=0. Außerdem ist das verbinden von Zellen möglich mit -rowspan und -columnspan. Des Weiteren können Widgets widerum in einer Zelle an Richtungen gebunden werden (-sticky). Die Untergliederung des in Layouts in Frames ist genauso wie in **pack** möglich.

grid optionen:

-row {0..x} Nummer der Zeile -column {0..x} Nummer der Spalte -rowspan {2..x} Anzahl der zu zusammenzufassenden Zeilen -columnspan{2..x} Anzahl der zu zusammenzufassenden Spalten -sticky {'n','s','e','w'} Kombinationen erlaubt (auch durch Komma getrennt} 'ew' bzw. 'ns' zieht Widget auf komplette Zellenbreite (auch bei columnspan) bzw. Zellenhöhe (auch bei rowspan)

-in weitere Untergliederung mit Hilfe von Frames

Beispiel mit grid

```
#!/usr/bin/perl use Tk; my $mw = new MainWindow; #configure mit
Fenstergroesse macht bei gridkeinen Sinn $mw->title ("Perl/Tk
- Erste Schritte"); my $label1 = $mw->Label (-text=>"Mein
Name"); my $entry1 = $mw->Entry (-width=>60); my $button1=
$mw->Button(-text=>"Exit",-command=>\& cmd_button);

$label1->grid (-row=>0, -column=>0); $entry1->grid(-row=>0,
-column=>1); # eine Spalte weiter rechts
$button1->grid(-row=>0, -column=>2);
```

```
sub cmd_button { $mw->messageBox (-message=>"Progamm wird
beendet!"); exit; } MainLoop;
```

Das ergibt das gleiche Bild wie mit **pack**. Wenn jetzt der Button in die nächste Zeile soll, dann wird für $button einfach -row geändert:

```
$button->grid (-row=>1,-column=>0);
```

soll der Button in der Mitte stehen und vielleicht noch die gesamte Fensterbreite in Anspruch nehmen:

```
$button->grid (-row=>1,-column=>0,-columnspan=>2,-sticky=>'ew');
#zwei Spalten überbrücken, Ost bis West
```

Mit Frames ändert sich eigentlich nicht viel, $label und $entry werden im Frame mit **grid** plaziert, $button braucht keine columnspan mehr, da nur eine Spalte vom Frame beansprucht wird:

```
#!/usr/bin/perl use Tk; my $mw = new MainWindow; my
$frame = $mw->Frame(); #configure mit Fenstergroesse
macht bei grid keinen Sinn $mw->title ("Perl/Tk - Erste
Schritte"); my $label1 = $mw->Label (-text=>"Mein Name");
my $entry1 = $mw->Entry (-width=>60); my $button1=
$mw->Button(-text=>"Exit",-command=>\& cmd_button);

$frame->grid (-row=>0,-column=>0);

$label1->grid (-in=>$frame,-row=>0, -column=>0);
$entry1->grid (-in=>$frame, -row=>0, -column=>1);
# einer Spalte weiter rechts $button1->grid(-row=>1,
-column=>0,-sticky=>'ew');

sub cmd_button { $mw->messageBox (-message=>"Progamm wird
beendet!"); exit; } MainLoop;
```

DBI: Datenbankzugriffe in Perl

Kapitel 24

DBI

Einleitung

Perl besitzt dank des DBI-Moduls[1] und diversen Datenbanktreibern (engl. *database drivers*, DBD) eine Schnittstelle, um mit verschiedenen Datenbanksystemen arbeiten zu können. Darunter fallen populäre Open-Source-Produkte wie MySQL[3] und PostgreSQL[4] sowie kommerzielle Riesen wie Oracle. Derzeit umfasst dieser Abschnitt die Möglichkeiten der Anbindung an MySQL, PostgreSQL und CSV-Dateien.

MySQL-Zugriff mit DBI

Mit Perl und dem DBI-Modul ist es verhältnismäßig leicht, sich mit einer MySQL-Datenbank zu verbinden und Abfragen abzusetzen. So fern das geeignete Perl-Modul für die Verbindung zu MySQL installiert ist (DBD::mysql[5]), kann es auch schon losgehen. Beim Verbindungsaufbau kann man noch allerhand Fein-Tuning betreiben, wie zB mit dem RaiseError- oder dem AutoCommit-Switch. Für weiterführende Erklärungen sollte man die Dokumentation des DBI-Moduls konsultieren. Des Weiteren benötigt man ausreichende SQL-Kenntnisse.

Im Sinne der Sicherheit, Portabilität und Faulheit ist anzumerken, dass es nur von Vorteil ist, die Zugangsdaten in einer externen Datei (eventuell in Form ei-

[1] CPAN: DBI[2]
[3] `http://www.mysql.com`
[4] `http://www.postgresql.org`
[5] CPAN: DBD::mysql[6]

nes Moduls) einfach in jedes Script einzubinden, damit, falls sich die Zugangsdaten ändern sollten, der Verwaltungsaufwand gering gehalten wird. Ansonsten müssten die Daten in jedem Script händisch angepasst werden. Außerdem ist in diesem Fall auch die Weitergabe des Quelltextes eher unbedenklich, da die brisante Information in einer separaten Datei steckt. Dieses externe File sollte dann verständlicherweise nicht für jedermann lesbar sein. In folgendem Code-Beispiel wurde jedoch aus Gründen der Einfachheit darauf verzichtet.

Einfaches Beispiel

```
#!/usr/bin/perl use strict; use warnings; use DBI;

# Deklaration der noetigen Variablen fuer die Verbindung #
Falls die Datenbank nicht lokal liegt, muss man zusaetzlich #
die Variablen $dbhost und/oder $dbport angeben # my $db_host =
"127.0.0.1"; # my $db_port = "3306";

my ($db_user, $db_name, $db_pass) = ("deinuser", "deineDB",
"deinpass");

# Verbindung zur DB herstellen # alternativ ( wenn DB nicht
lokal ): # my $dbh = DBI->connect("DBI:mysql:database=$db_-
name;host=$db_host;port=$db_port", # "$db_user", "$db_pass");

my $dbh = DBI->connect("DBI:mysql:database=$db_name", "$db_-
user", "$db_pass");

# Vorbereiten des SQL-Statements

my $query_test = $dbh->prepare("SELECT * FROM deinetabelle");

# Ausfuehren des Statements

$query_test->execute() or die $query_test->err_str;

# Hier koennte weitergehende Verarbeitung erfolgen, eine
Auflistung aller Eintraege zB:

while (my ($col_1, $col_2, $col_3) = $query_test->fetchrow_-
array) { print "Spalte 1: " . $col_1 . "\n"; print "Spalte 2: "
. $col_2 . "\n"; print "Spalte 3: " . $col_3 . "\n"; }

# Nun erstellen wir doch noch gleich eine neue Tabelle #
Statt der Möglichkeit mittels vorherigem prepare und execute,
benutzen # wir hier die einfache Methode do
```

```
my $create_query = ||CREATE table testtable||;

$dbh->do($create_query);

# Nachdem alles erledigt ist, schließen wir die
Datenbankverbindung

$dbh->disconnect;
```

Die eigentliche Arbeit in diesem Script steckt in Zeile 31. Hier wird mittels der Funktion `fetchrow_array` jeweils eine Zeile in eine Liste/Array eingelesen und zurückgeliefert. Die ersten 3 Werte dieser Liste werden danach den Werten **$col_1** bis **$col_3** zugewiesen. Sobald `fetchrow_array` keine Zeile mehr zurückliefert, beendet sich die `while`-Schleife.

Erweitertes Beispiel mit Modulanbindung

Hier eine von vielen Möglichkeiten die Verbindungsdaten in einem externen File unterzubringen. Der folgende Ansatz bindet die Daten mittels eines Moduls ein.

Das externe File mit den Daten (**ExterneDaten.pm**): `#!/usr/bin/perl use strict; use warnings; our @EXPORT_OK = qw($DB_USER $DB_PASSWD $DATABASE); use Exporter; our @ISA = qw(Exporter);`

`package ExterneDaten; our $DB_USER = ||youruser||; our $DB_PASSWD = ||yourpasswd||; our $DATABASE = ||yourDB||;`

`1;`

Dieses File sollte nun mit entsprehend sicheren Zugriffsrechten versehen werden und den Namen **ExterneDaten.pm** erhalten. Nachdem das erledigt ist, kann man das erste Beispiel auch wie folgt formulieren:

```
#!/usr/bin/perl use strict; use warnings; use DBI; use lib ||.||;
use ExterneDaten;

# Verbindung zur DB herstellen

my $dbh = DBI->connect(||DBI:mysql:database=$ExterneDaten::DATABASE||,
||$ExterneDaten::DB_USER||, ||$ExterneDaten::DB_PASSWD||);

# Vorbereiten des SQL-Statements

my $query_test = $dbh->prepare(||SELECT * FROM deinetabelle||);

# ... # ... alles andere bleibt gleich # ...
```

Erklärung

In Zeile 5 wird Perl angewiesen das aktuelle Verzeichnis in das **@INC-Array** aufzunehmen, dies ist notwendig, damit **ExterneDaten.pm** danach auch, wie in Zeile 6 gefordert, gefunden wird. Durch die package-Anweisung im Modul befinden sich die Variablen im **ExterneDaten**-Namensraum, weshalb den eigentlichen Variablennamen **ExterneDaten::** vorangestellt wird.

DBI-Zugriff auf andere Datenbanksysteme

Im wesentlichen können hier die Erkenntnisse des letzten Kapitels übernommen werden. Den größten Unterschied stellt lediglich die `connect`-Anweisung dar, die für jedes DBMS etwas anders ausschaut. Des weiteren sollten, um die folgenden Beispiele ausführen zu können, die Module DBD::Pg[7] für PostgreSQL sowie DBD::CSV[9] für CSV installiert sein. Beide Module sind im CPAN[11] zu finden.

PostgreSQL

```
my $dbh = DBI->connect("DBI:Pg:dbname=$db_name", "$db_user",
"$db_pass");
```

Ein CSV-Verzeichnis

CSV steht für "Comma-separated Values", und bezeichnet keine Datenbank im eigentlichen Sinne, sondern einen Dateityp, eigentlich reine Textdateien im ASCII-Format, die von jedem Texteditor gelesen und bearbeitet werden können. Damit hat der Programmierer die Möglichkeit, SQL zu nutzen, ohne ein DBMS installieren zu müssen. Vorteilhaft, wenn man sparsam mit den Ressourcen umgehen muß.

Als Datenbank verwendet man ein Verzeichnis, und jede CSV-Datei steht für eine Tabelle. Benutzername und Passwort entfallen.

```
my $dbh = DBI->connect("DBI:CSV:f_dir=/pfad/zu/deinem/csvverzeichnis");

#Oder bei Windows my $dbh = DBI->connect("DBI:CSV:f_-
dir=c:\\pfad\\zu\\deinem\\csvverzeichnis");
```

[7] CPAN: DBD::Pg[8]
[9] CPAN: DBD::CSV[10]
[11] `http://search.cpan.org`

SDBM

SDBM (Standard-Perl-Quelldistribution) ist eine sehr einfache Datenbank, die häufig bei einer gewöhnlichen Linux-Distribution standardmäßig installiert wird. Es werden zwei Dateien erzeugt. Eine mit der Endung .dir und eine .pag, wobei letztere die Daten beinhaltet.

Hier ein erklärendes Beispiel eingebettet in eine Klasse:

```
package cl_db;

use Fcntl; # O_RDWR, O_CREAT, etc. use SDBM_File;

sub new # Konstruktor { my $this = shift; my $class =
ref($this) || $this;

my $self = { temperatur => {}, # Temperaturwerte };

bless $self, $class; # Objekt erzeugen

tie(%{$self->{temperatur}}, 'SDBM_File',"temperatur.db", O_-
RDWR|O_CREAT, 0666);

return $self; }
```

Der Konstruktor verbindet das Hash %self{temperatur} mit der Datenbank (temperatur.db). Es werden im aktuellen Verzeichnis die beiden Dateien temperatur.db.dir und temperatur.dp.pag angelegt bzw. benutzt.

```
sub writeDB { my $self = shift; my $key = shift; #
Datenschluessel my $data = shift; # Datenwert

$self->{temperatur}->{$key} = $data;

return 1; }
```

Die Funktion writeDB schreibt Daten in die Datenbank.

```
sub readDB { my $self = shift; my $key = shift; #
Datenschluessel

return $self->{temperatur}->{$key}; }
```

Die Funktion readDB liest bestimmte Daten der Datenbank aus.

```
sub DESTROY { my $self = shift;

untie %{$self->{temperatur}}; } 1; # Returnwert für Package
```

Der Destruktor löst die Verbindung wieder auf und schließt somit die Datenbank wieder.

Das folgende Beispiel zeigt, wie diese Klasse benutzt werden kann:

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

use cl_db;

my $db = cl_db->new(); # Konstruktor wird aufgerufen
$db->writeDB("InnenTemperatur", 21.0); # Daten schreiben print
"Innentemperatur: ", $db->readDB("InnenTemperatur"), "\n"; #
Daten lesen
```

Module

Anhänge

Installation

Kapitel 25

Installation

Perl ist *freie Software*. Sie können den Quelltext von Perl von der Seite www.cpan.org[1] herunterladen. Der Quelltext von Perl ist in der Programmiersprache C geschrieben. Wenn ein C-Compiler und genügend Ressourcen vorhanden sind, können Sie den Quelltext selbst in einen lauffähigen Perl-Interpreter übersetzen. Alternativ können Sie vorkompilierte Pakete herunterladen und auf ihrem Rechner installieren.

Auf vielen Systemen ist Perl bereits vorinstalliert

Auf vielen Betriebssystemen wird Perl vom Hersteller oder Distributor mitgeliefert und ist bereits installiert. Dies ist z.B. bei den meisten Linux-Systemen, bei vielen Unix-Systemen und bei Apple's Mac OS X der Fall. Sie brauchen dann nur ein Terminal zu öffnen und

perl -v

einzugeben. Antwortet der Rechner mit

This is perl, version 5.005_03 built for sun4-solaris

Copyright 1987-1999, Larry Wall

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

[1] `http://www.cpan.org/src/README.html`

Complete documentation for Perl, including FAQ lists, should be found on this system using ‘man perl’ or ‘perldoc perl’. If you have access to the Internet, point your browser at http://www.perl.com/, the Perl Home Page.

oder etwas Ähnlichem, so können Sie Perl sofort verwenden.

Vorübersetztes Perl

Erhalten Sie dagegen

perl: command not found

so ist Perl möglicherweise noch nicht installiert.

Allerdings müssen Sie dann immer noch nicht unbedingt selbst Perl übersetzen. Auch hier hat in den meisten Fällen bereits vorher jemand für Sie die Arbeit getan und eine spezielle Binärdistribution erzeugt (ein übersetztes Programm nennt man *Binär*code, daher der Name). Dann können Sie eine solche Binärdistribution verwenden. Unter www.cpan.org/ports[2] befinden sich Verweise auf Binärdistributionen für eine Vielzahl von Betriebssystemen. Diese enthalten auch eine Installationsanleitung.

Die Installation einer Binärdistribution ähnelt meist der Installation von anderen Programmen auf einem System und ist daher leicht durchzuführen.

Wenn Sie Microsoft Windows verwenden, kommt neben speziellen Distributionen (siehe Absatz ‖Perl für Windows‖) auch die freie Cygwin-Umgebung in Frage, die viele Linux/Unix-Programme enthält, unter anderem auch Perl (www.cygwin.com[3]).

Perl selbst übersetzen

Falls Perl nicht vorhanden sein sollte und eine Binärdistribution nicht in Frage kommt, kann man Perl selbst übersetzen. Dazu muss ein Linux- bzw. Unix-ähnliches Umfeld vorhanden sein, insbesondere ein C-Compiler und Unix-Werkzeuge wie die Shell (*sh*) oder der Programm-Manager *make*. MS Windows-Benutzer können hierzu *cygwin* installieren.

[2] `http://www.cpan.org/ports`
[3] `http://www.cygwin.com`

Wenn das installierte Perl älter ist oder bestimmte Funktionen vermisst werden, ist eine eigene Übersetzung auch hilfreich. Neue Funktionen wie *Threads* oder mehrere Interpreter sind oft in der Standardinstallation nicht implementiert.

Vorhandene Perl-Installation Ein vorhandenes Perl befindet sich meist in */usr/bin/perl* oder */usr/local/bin/perl*. Diese Dateien sollten zunächst nicht durch eine eigene Version überschrieben werden, da andere Programme im System gefährdet werden können, insbesondere wenn die neue Version nicht richtig funktioniert oder Funktionen wie *Threads* fehlen.

Oft darf man als normaler Benutzer die Systeminstallation auch nicht überschreiben. Daher empfiehlt es sich, eine neue Installation im Benutzerbereich vorzunehmen und zu testen. Als Vorschlag wird im Folgenden die Installation in *~/bin* und die Perl-Umgebung wie Debugger und Module in *~/perl* empfohlen. Die eigentliche Übersetzung wird in *~/src* durchgeführt. Hierzu sind keine Systemrechte nötig.

Vorbereitung Zunächst erzeugen wir die Verzeichnisse *~/bin* und *~/perl* und sorgen dafür, dass Programme in *~/bin* gefunden werden:

cd ~ mkdir -p bin src perl PATH=~/bin:$PATH export PATH

oder für C-Shell Benutzer:

cd ~ mkdir -p bin src perl set path=(~/bin $path)

Entpacken der Quelldateien Die Datei www.cpan.org/src/stable.tar.gz[4] wird jetzt lokal unter *~/src* abgelegt und mit dem *tar*-Kommando ausgepackt:

cd ~/src gunzip stable.tar.gz tar xf stable.tar

Jetzt ist ein Unterverzeichnis, etwa *perl-5.8.5*, mit den Quelldateien entstanden, in das wir wechseln und die Dateien README und INSTALL betrachten:

ls -F perl-5.8.5/ cd perl-5.8.5 less README INSTALL

Möglicherweise existiert noch eine systemabhängige Datei wie README.cygwin oder README.macosx.

[4] `http://www.cpan.org/src/stable.tar.gz`

Konfiguration Das Skript *Configure* kann jetzt ausgeführt werden. Es stellt verschiedene Fragen, wobei die Vorschläge meist übernommen werden können. Auf die Frage nach dem Installationsort antworten wir jedoch statt */usr/local* mit *~/perl* und die Binärdateien installieren wir *~/bin*:

sh Configure.sh 2>& 1 | tee log.Configure

Beginning of configuration questions for perl5.

Checking echo to see how to suppress newlines... ... Installation prefix to use? [/usr/local] **~/perl** ... Pathname where the public executables will reside? [~/perl/bin] **~/bin** ...

Bei allen anderen Fragen kann man zunächst einfach Return drücken.

Es gibt auch die Möglichkeit Configure mit Parametern zu starten.

./Configure -d -s -Dprefix=~/perl

Übersetzung, Test und Installation Folgendermassen kann man Perl für das eigene System übersetzen, testen und installieren. Dies kann durchaus einige Zeit in Anspruch nehmen (systemabhängig). Der Autor empfiehlt Kaffee zu kochen.

make ... make test ... make install ...

Nun sollte Perl installiert sein. Durch folgende Eingabe kann danach der Erfolg ermittelt werden.

~/bin/perl -e 'print "OKAY.\n"' OKAY.

Wird Okay ausgegeben, wurde Perl erfolgreich installiert und ist lauffähig. Herzlichen Glückwunsch.

Perl für Windows

Falls Sie Perl auf einem Windows-System benutzen wollen, gibt es von ActiveState ein kostenloses vorkompilertes Perl mit Installer, genannt ActivePerl. Von ActiveState kann man auch für Win32 vorkompilierte Module beziehen (über einen in ActivePerl integrierten Paketmanager oder als .zip Archiv).

Für CGI- bzw. mod_perl-Programmierer eignet sich auch gut das XAMPP-Komplettpaket von den Apache Friends. Hier ist bereits ein Apache Webserver, Perl und die Datenbanken MySQL und SQLite (mein Geheimtipp) enthalten. Für Perl-Programmierer gibt es auch ein Add-on mit einer kompletten Perl-Distribution (alle Core Module) und mod_perl.

Links

- Perl Sourcecode (CPAN): http://www.cpan.org/src/README.html
- Suche nach Perlmodulen (CPAN): http://search.cpan.org/
- ActivePerl: http://www.activestate.com/Products/ActivePerl/
- Suche nach Perl Modulen (ActiveState): http://aspn.activestate.com/ASPN/Perl/Modules/
- XAMPP für Windows http://www.apachefriends.org/de/xampp-windows.html

Nützliche Module

Kapitel 26

Nützliche Module

Vorbemerkung In diesem Abschnitt werden nüzliche Module vorgestellt, die Programmieraufgaben erleichtern können.

Term::ANSIColor Dieses Modul ermöglicht die Ausgabe auf der Kommandozeile einzufärben, um so die Übersichtlichkeit der Ausgabe zu steigern. Ein Programmbeispiel:

#!/usr/bin/perl -w # Programm zur Demonstration der eingefärbten Kommandozeilenausgabe.

use Term::ANSIColor;

print colored(" Hallo Welt \n",'Bold Blue');

Das Programm gibt *Hallo Welt* in einem satten Blau aus. Die Farbgebung ist sehr assoziativ, so würde zum Beispiel 'Yellow on_magenta' ebenfalls als Farbgebung funktionieren. Weitere Informationen stehen in der Anleitung.

perldoc Term::ANSIColor

Das Modul sollte nur verwendet werden, wenn eine unterschiedliche Farbgebung sinnvoll ist. Auf diese Weise werden Netzhautschäden beim Benutzer des Programms vermieden, was diesen sicherlich freut.

Data::Dumper Wer kennt das nicht, man hat komplizierte Datenstrukturen und irgend etwas stimmt damit nicht. Wenn man jetzt nur die Möglichkeit hätte, diese Struktur auszugeben. Ein 'print $hash' liefert leider nur so etwas unverständliches wie 'HASH(0x814cc20)'.

Hier kann das Modul *Data::Dumper* helfen. Es importiert automatisch eine Funktion *Dumper*, die beliebige skalare Werte ausgeben kann (auch mehrere davon) und sogar mit Zyklen in der Datenstruktur zurechtkommt. Dazu ist die Ausgabe sehr gut lesbar. Hier ein Beispiel mit einer einfacheren und einer komplizierten Struktur:

use Data::Dumper;

my $simple = { a => 1 };

my $complicated = [sub { ucfirst(shift) }, $simple];

push @$complicated, $complicated;

print Dumper($simple, $complicated);

Die erste Variable ($simple) ist einfach nur eine Hashreferenz, die zweite Variable ($complicated) arbeitet mit Codereferenzen und beinhaltet einen Zyklus. Bei der Ausgabe benennt *Dumper* die Variablen mit einem automatisch generierten Wert und benutzt diesen, um Zyklen und Verweise auf bereits dargestellte Inhalte zu verdeutlichen. Hier die Ausgabe des obigen Programms:

$VAR1 = { 'a' => 1 }; $VAR2 = [sub { ||DUMMY|| }, $VAR1, $VAR2];

Natürlich kann Dumper den Variablen noch vom Nutzer vorgegebene Namen geben oder die Rekursivtiefe der Darstellung begrenzen, diese und noch viele anderen Informationen sind in der Manualpage zu finden, einfach:

perldoc Data::Dumper

aufrufen.

Für mich ist *Data::Dumper* ein unentbehrliches Werkzeug und ein 'die Dumper($var)' ist ein schnelles Mittel um Probleme mit den Datenstrukturen zu finden.

File::Slurp Dieses Modul beinhaltet Funktionen um eine Datei in einem Rutsch einzulesen oder zu schreiben. Es ist entstanden, weil diese Funktionen an unzähligen Stellen in Perlprogrammen und -modulen vorkommen. Außerdem ist die `slurp`-Funktion Bestandteil des kommenden Perl6, so daß man seinen Code schon für diese kommende Version vorbereiten kann.

Ein kleines Beispielprogramm das einen Textdatei neu formatiert. Warnung: nicht zur Verwendung mit wichtigen Textdateien vorgesehen!

```
; use strict; use warnings; use utf8
```

```
; use Text::Wrap qw(wrap) ; use File::Slurp qw(slurp write_-
file) ; use IO::File

; my $x=@ARGV ; my $columns=72;

; if($x==2) { $columns = shift @ARGV } elsif($x==0) { print
STDERR "$0 [columns] file\n" ; exit 1 }

; my $filename=shift @ARGV ; my @file

; { # Read in ; my $in=new IO::File "<$filename" or die
"$filename is unreadable\n" ; @file=slurp($in) }

; { # write out ; my $out=new IO::File ">$filename" or die
"$filename is unwritable\n" ; $Text::Wrap::columns=$columns ;
write_file($out, wrap(,,@file)) }
```

Webseiten und mehr

Kapitel 27

Webseiten und mehr

Webseiten und mehr

im web unter:

- www.perl.org[1] die zentrale Netzseite zu Perl (enthält schon alle Links die man braucht);
- www.cpan.org[2] zentrales Download-Register für Perl und Perl-Module, inklusive praktischer Suchfunktion;
- ActivePerl[3] ist die unter Windows am weitesten verbreitete Perl-Distribution;
- http://www.perldoc.com/ offizielle Perl-Dokumentation und http://learn.perl.org/library/ Perl-Bücher online;
- http://use.perl.org/ Perl-Nachrichten und http://www.perl.com/ aktuelle Berichte;
- Perl Monks[4] großes Portal für Perl-Benutzer: Hilfe, Diskussionen, Anleitungen (auf englisch);
- Perl Mongers[5] weltweites Netzwerk lokaler Benutzergruppen, http://perlmongers.de/ auch in Deutschland;

[1] `http://www.perl.org`
[2] `http://www.cpan.org`
[3] `http://www.activestate.com/Products/ActivePerl/`
[4] `http://www.perlmonks.org`
[5] `http://www.pm.org`

- http://www.perl-community.de Portal deutscher Perl-Gemeindschaft mit Forum, Wiki, Links, etc.;
- http://perl.plover.com/ witzige und intelligente Schriften, nicht nur über Perl;
- http://www.perlmeister.com/index.html Kolumnen und Programmierbeispiele vom Perlmeister;
- http://www.stonehenge.com/merlyn/columns.html Kolumnen von Merlin, Altmeister der Perl-Magie;
- http://perl-seiten.homepage.t-online.de/html/perl_inhalt.html Perl-Seiten - eine Einführung zu Perl
- ...

Hilfreiche Befehle

- man perl
- perldoc xy
- perldoc -f *(z.B. **perldoc -f print**)* gibt uns eine praktische Anleitung zu der Perlfunktion
- man perlfunc *(Übersicht aller wichtigen Perlfunktionen)*
- ...

Buchtipps

Kapitel 28

Buchtipps

zum Einstieg:

- Randal L. Schwartz, Tom Phoenix & Brian D. Foy: **Learning Perl, 4th Edition**, ISBN 0596101058 (in deutsch: ISBN 3897214342)
- Simon Cozens: **Beginning Perl**, ISBN 1861003145 (nicht in deutsch erhältlich) (auch online: http://www.perl.org/books/beginning-perl/)
- Larry Wall, Tom Christiansen, Jon Orwant: **Programming Perl** (3rd Edition) ISBN 0596000278 (in deutsch: ISBN 3897211440)

automatisiertes Testen:

- Ian Langworth, chromatic: **Perl Testing - A Developer's Notebook**, O'Reilly Media, Inc., 2005, ISBN 0-596-10092-2

mit Praxisbeispielen:

- Michael Schilli: **GoTo Perl 5**, Addison-Wesley, 1998, ISBN 3827313783 (ist deutsch)
- Tom Christiansen, Nathan Torkington: **Perl Cookbook** ISBN 0596003137 (in deutsch: ISBN 3897213664)

zur Weiterbildung:

- Randal L. Schwartz, Tom Phoenix: **Learning Perl Objects, References & Modules**, ISBN 0596004788 (in deutsch: ISBN 3897211491)
- Damian Conway: **Object Oriented Perl** ISBN 1884777791 (in deutsch: ISBN 3-8273-1812-2)

- N. Hall, Randal L. Schwartz: **Effective Perl Programming** ISBN 0201419750 (in deutsch: ISBN 3827314062)
- Sriram Srinivasan: **Advanced Perl Programming**, ISBN 1-56592-220-4 (in deutsch: ISBN 3897211076)
- Damian Conway: **Perl Best Practices**, ISBN 0596001738 (in deutsch: ISBN 3897214547)

Glossar

Kapitel 29

Glossar

Vorbemerkung

Das hier soll als kleines Nachschlagewerk dienen.

- abgeleitete Klasse

  Eine Klasse, die ihre Methoden über eine allgemeinere Klasse, eine so genannte Basisklasse, beschreibt. Beachten Sie, dass Klassen nicht ausschließlich in Basisklassen oder abgeleitete Klassen unterteilt werden; eine Klasse kann gleichzeitig sowohl eine abgeleitete Klasse als auch eine Basisklasse darstellen.

- abschneiden

  (engl. »truncate«) Das Entfernen eines vorhandenen Inhalts aus einer Datei. Das kann automatisch beim Öffnen einer Datei mit Schreibrechten oder explizit über die truncate-Funktion erfolgen.

- Accessor-Methode

  Eine Methode, die zur indirekten Inspektion bzw. Aktualisierung des Zustandes eines Objekts (seiner Instanzvariablen) verwendet wird.

- Adressoperator

  Einige Sprachen arbeiten direkt mit den Speicheradressen von Werten, was aber leicht zu einem Spiel mit dem Feuer werden kann. Perl stellt Ihnen einige Funktionen, die Ihnen das Speichermanagement abnehmen. Bei Perl kommt der Backslash-Operator einem Adress-

operator am nächsten. Er versorgt sie aber mit einer harten Referenz , was wesentlich sicherer ist, als eine Speicheradresse.

- aktuelles Paket

  Das Paket, in dem die aktuelle Anweisung kompiliert wird. Gehen Sie in Ihrem Programm zurück, bis Sie eine Paketdeklaration im gleichen lexikalischen Geltungsbereich oder in allen umschließenden lexikalischen Geltungsbereichen finden. Das ist der Name Ihres aktuellen Paketes.

- Algorithmus

  Eine genau definierte Folge von Schritten, die so umfassend erläutert sind, dass sie ein Computer ausführen kann.

- Alias

  Ein Spitzname für irgendetwas, der sich in allen Fällen so verhält, als würden Sie den ursprünglichen Namen verwenden und nicht dessen Spitznamen. Temporäre Aliase werden implizit in der Schleifenvariable für „foreach-Schleifen“, bei der „$Variable“ für die „map-“ oder „grep-Operatoren“, für „$a“ und „$b“ während der sort-Vergleichsfunktionen und bei jedem Element von „@_“ für die tatsächlichen Argumente eines Subroutinen-Aufrufs erzeugt. Permanente Aliase werden in Paketen explizit erzeugt, indem man Symbole importiert oder an Typeglobs zuweist. Lexikalisch beschränkte Aliase für Paketvariablen werden explizit durch die our-Deklaration erzeugt.

- Alternativen

  Eine Liste mit mehreren Wahlmöglichkeiten, unter denen Sie sich eine aussuchen können. Alternativen werden in regulären Ausdrücken durch den vertikalen Strich (|) voneinander getrennt. Alternativen in normalen Perlausdrücken werden von zwei vertikalen Strichen (||) voneinander getrennt. Logische Alternativen in Booleschen werden durch (||) oder durch „or“ getrennt.

- anonym

  Wird verwendet um einen Referenten zu beschreiben, der nicht direkt durch eine benannte Variable zu erreichen ist. Ein solcher Referent muss indirekt über mindestens eine harte Referenz zugänglich sein. Wenn die letzte harte Referenz verschwindet, wird der anonyme Referent gnadenlos entfernt.

- Anweisung

  Ein Befehl an einem Computer, der angibt, was als nächstes zu geschehen hat.Ist nicht mit der Deklaration zu verwechseln, die ihren Computer nicht anweist, etwas zu tun, sondern nur, etwas zu lernen.

- Anweisungsmodifizierer

  Eine Bedingungsanweisung die hinter einer Schleife steht, und nicht davor.

- Arbeitsverzeichnis

  Ihr aktuelles Verzeichnis, von dem ausgehend relative Pfadnamen vom Betriebssystem interpretiert werden. Das Betriebssystem kennt das aktuelle Verzeichnis, weil sie mit „chdir“ dorthin verzweigt sind oder weil sie am gleichen Ort angefangen haben wie der Parent-Prozess, von dem Sie abstammen.

- Architektur

  Die Art von Computer, mit dem sie arbeiten, wobei eine einzelne »Art« alle diejenigen Computer umfasst, deren Maschinensprache kompatibel sind. Weil Perl Programme einfache Textdateien sind, und keine Ausführbaren Binärprogramme, ist ein Perl Programm wesentlich weniger von der verwendeten Architektur abhängig, als Programme in anderen Sprachen (z.B. C) die direkt in Maschinencode übersetzt werden. Siehe auch Plattform und Betriebssystem.

- Argument-Vektor

- Argument

  Ein Datenelement, das als Eingabe an ein Programm, eine Subroutine eine Funktion oder eine Methode übergeben wird. Das Argument gibt an, was zu tun ist. Es wird auch als »Parameter« bezeichnet.

- ARGV

  Der Name des Arrays, dass den Argument-Vektor, der Befehlszeile enthält. Wenn sie den leeren <>-Operator verwenden, ist „ARGV“ sowohl der Name des Dateihandles, mit dem die Argumente durchgegangen werden, als auch derjenige des Skalars, der den Namen der aktuellen Eingabe Datei enthält.

- arithmetischer Operator

Ein Symbol wie + oder /, mit dem sie Perl anweisen, Arithmetik zu betreiben.

- Array

  Eine geordnete Sequenz von Werten, die so abgelegt sind, das jeder Wert auf einfache Weise über einen ganzzahligen Index zugänglich ist, der den Offset des Wertes innerhalb dieser Sequenz bestimmt.

- Arraykontext

  Ein archaischer Ausdruck für etwas, was man genauer als Listenkontext bezeichnen sollte.

- ASCII

  Wird ganz allgemein für den »American Standard Code for Information Interchange« (einen 7-Bit-Zeichensatz, der nur einfachen englischen Text darstellen kann) verwendet. Wird häufig auch für die unteren 128 Werte der verschiedenen ISO-8859-X-Zeichensätze verwendet (einer reihe untereinander inkompatibler internationaler 8-Bit-Codes). Siehe auch Unicode

- Assertion

  Siehe Zusicherung

- assoziatives Array

  Siehe Hash

- Assoziativität

  Bestimmt, ob zuerst der linke Operator oder der rechte Operator ausgeführt wird, wenn ein Konstrukt wie »A Operator B Operator C« vorliegt und zwei Operatoren den gleichen Vorrang besitze. Operatoren wie + sind links assoziativ, während Operatoren wie ** rechts assoziativ sind.

- asynchron

  Asynchron sind Ergebnisse oder Aktivitäten, deren relative zeitliche Anordnung nicht vorherzusagen ist, weil zu viele Dinge auf einmal geschehen. Ein asynchrones Ereignis ist also eines, von dem Sie nicht wissen, wann es eintritt

- Atom

Eine Komponente eines regulären Ausdrucks die potentiell einen Teilstring erkennt und aus einem oder mehreren Zeichen besteht, die von jedem nachfolgenden Quantifizierer als unsichtbare syntaktische Einheit betrachtet werden. (Das Gegenteil dazu bildet die Behauptung, die etwas mit Nulllänge erkennt und nicht quantifiziert werden kann.)

- atomische Operation
- Attribut
- Aufruf (»Innovation«)
- Ausdruck
- ausführbare Datei
- Ausführen
- Ausführungsphase
- Ausnahme
- Ausnahmebehandlung
- Autogenerierung
- Autoinkrement

  Automatisches Erhöhen einer Zählvariablen (zum Beispiel $i++)
- Autoload

- Autosplit
- Autovivication
- AV
- awk

  Der Name einer alten Textverarbeitungssprache, von der Perl einige Ideen übernommen hat

B

- Backtracking

  Die praktizierte Variante des Ausspruchs: Aus mathematischer Sicht handelt es sich um die Rückkehr einer erfolglosen Rekursion in einem verzweigten Baum von Möglichkeiten. Backtracking kommt bei Perl vor, wenn ein Muster mit einem regulären Ausdruck erkannt werden soll, und eine frühere Vermutung nicht zutrifft.

- Bareword

  Ein ausreichendes mehrdeutiges Wort, um es unter „use strict 'subs'“ als ungültig zu betrachten. Fehlt diese Beschränkung, wird ein Bareword so behandelt, als wäre es von Quotingzeichen umschlossen.

- Basisklasse

  Ein generischer Objekttyp, also eine Klasse, von der sich andere, spezifischere Klassen mit Hilfe der Vererbung ableiten. Wird auch als »Superklasse« bezeichnet.

- Bedingungsanweisung

  Etwas mit Wenn, aber ohne Aber. Siehe boolescher Kontext.

- Befehl

Bei der Shell-Programmierung, die syntaktische Kombination eines Programmnamens mit seinen Argumenten.

- Befehlsname

  Der Name des gerade laufenden Programms, wie er in der Befehlszeile eingegeben wurde. Bei C wird der Befehlsname dem Programm als erstes Argument übergeben. Bei Perl steht er in der eigens dafür vorgesehenen Variable $0.

- Befehlspufferung

  Ein Perl-Mechanismus, der nach jedem Perl-Befehl, der eine Ausgabe bewirkt, diese sofort an das Betriebssystem weitergibt. Wird durch das Setzen von $|($AUTOFLUSH) auf einen Wahr-Wert aktiviert. Das ist manchmal notwendig, wenn Daten durch Pufferung ihr Ziel nicht sofort erreichen. Standardmäßig wird bei Dateien oder Pipes mit Blockpufferung gearbeitet.

- Behauptung

- benannte Pipe

  Eine Pipe mit einem im Dateisystem eingebetteten Namen, auf die dann von zwei unabhängigen Prozessen aus zugegriffen werden kann.

- Besitzer

  Der Benutzer, der (neben dem Superuser) die volle Kontrolle über die Datei hat. eine Datei kann auch einer Gruppe von Benutzern zugewiesen sein, die gemeinsam auf sie zugreifen, wenn der eigentliche Besitzer das erlaubt. Siehe Zugriffsflags.

- Betriebssystem

  Ein spezielles Programm, das direkt auf der Maschine läuft und so unangenehme Details wie die Verwaltung von Prozessen und Geräten vor Ihnen versteckt. Wird gelegentlich in nicht ganz so strengen Sinne verwendet, um eine bestimmte Programmierkultur zu bezeichnen.

- Bibliothek

  Eine Sammlung von Prozeduren. In früheren Tagen eine Sammlung von Subroutinen in eine .pl-Datei. Heutzutage wird damit häufig die gesamte Sammlung von Perl-Modulen Ihres Systems bezeichnet.

- Big-Endian

  Wird auch für Computer verwendet, die das höherwertige Byte eines Wortes an einer niedrigeren Byteadresse ablegen als das niederwerige Byte des Wortes. Siehe auch Little-Endian

- binär

  Hat was mit Zahlen zu tun, die auf Grundlage der Basis 2 repräsentiert werden, d.h. es gibt grundsätzlich nur zwei Ziffern, 0 und 1.

- binärer (dyadischer) Operator

  Ein Operator, der mit zwei Operanden arbeitet.

- Bindung

  Die Zuweisung einer bestimmten Netzwerkadresse an einen Socket

- Bit

  Ein Integerwert im Bereich von 0 bis 1 einschließlich. Die kleinste zu speichernde Informationseinheit. Ein Achtel Byte

- Bitshift

  Die Bewegung von Bits in einem (Computer-)Wort nach links oder rechts. Bewirkt eine Multiplikation oder Division mit einer Zweierpotenz.

- Bitstring

  Eine Folge von Bits, die tatsächlich als Folge von Bits betrachtet wird.

- Block

  Ein Datensegment von einer Größe, mit der das Betriebssystem gern arbeitet (normalerweise eine Zweierpotenz wie 64 oder 1024). Verweist typischerweise auf ein Datensegment, das von einer Festplatte gelesen, oder geschrieben werden soll.

- BLOCK

  Ein syntaktisches Konstrukt, das aus einer Sequenz von Perl-Anweisungen besteht, die von geschweiften Klammern umschlossen sind. Die „if“- und „while“-Anweisung sind als „Block“-Konstrukte definiert.

- Blockpufferung

Eine Methode, mit der Ein- und Ausgaben effizienter durchgeführt werden, weil jeweils ein ganzer Block verarbeitet wird. Per Voreinstellung führt Perl die Blockpufferung bei Dateien durch, Die auf der Festplatte abgelegt werden. Siehe Puffer und Befehlspufferung.

- Boolescher Kontext

  Eine spezielle Form des skalaren Kontexts, bei der das Programm nur entscheidet, ob der durch einen Ausdruck zurückgelieferte skalare Wert wahr oder falsch ist. Evaluiert weder als String noch als Zahl. Siehe Kontext

- Boolescher Wert

  Ein Wert, der entweder wahr oder falsch ist.

- Breakpunkt

  Ein Punkt in Ihrem Programm, an dem der Debugger die Ausführung anhalten soll.

- Broadcast

  Das gleichzeitige Senden eines Datagramms an mehrere Ziele.

- BSD

  Berkeley **S**tandard **D**istribution – eine Psychoaktive Substanz, populär in den achtziger Jahren, die im Dunstkreis der U.C. Berkeley entwickelt wurde. In vielen Fällen dem verschreibungspflichtigem Medikament namens »System V« ähnlich, aber unendlich viel nützlicher.

- Bucket

  Ein Punkt in einer Hashtabelle der (eventuell) mehrere Einträge enthält, deren Schlüssel, entsprechend der Hashfunktion auf den gleichen Hashwert abgebildet werden.

- Bundle

  Eine Gruppe verwandter Module im CPAN.

- Byte

  Ein in den meisten Fällen aus acht Bits bestehenden Datei.

- Bytecode

  Eine Mischsprache, die zwischen Androiden gesprochen wird, wenn sie ihre genaue Ausrichtung nicht preisgeben wollen. (Siehe Endian

Diese Sprache zeichnet sich dadurch aus, dass sie alles in einer architekturunabhängigen Folge von Bytes darstellen.

C

- C
- C-Präprozessor
- Call by Reference
- Call by Value
- Callback
- Capturing
- Client
- Cloister
- Closure
- Cluster
- CODE
- Codegenerator

- code subpattern
- Compiler
- Composer
- Coredump
- CPAN
- Cracker
- CV

D

- Datagramm
- Datei
- Dateideskriptor
- Dateiglob
- Dateihandle

- Dateiname
- Dateisystem
- Datei-Testoperator
- Datensatz
- Datentyp
- DBM
- definiert
- Deklaration
- Dekrement
- Dereferenzierung
- Deskriptor
- Destruktor
- Direktive
- Dispatch

- Distribution
- Dweomer
- Dwimmer
- dynamsiches Scoping

E

- Eigenschaft
- einbetten
- einfache Vererbung
- Einzeiler
- eklektisch
- Element
- en passant
- Endian

F

- Feld
- fest eingebaut (built-in)
- FIFO
- Filter
- Flag
- Fließkomma
- Flush
- Fork
- formale Argumente
- Format
- Fortsetzungszeilen
- Freeware
- Freigabe

- frei verfügbar
- frei verteilbar
- Funktion
- Funny Character

G

- Garbage Collection
- Geltungsvereich (Scoping)
- Gerät
- GID
- gierig

  Beschreibt das Verhalten eines Suchmusters in einem regulären Ausdruck. Ein Muster gefolgt von einem Quantifier, welches auf eine Zeichenfolge passt, wird die Länge der passenden Zeichenfolge maximieren. Dies ist das normale Verhalten. Mit einem ? hinter dem Quantifier wird angezeigt, dass der Ausdruck „nicht gierig“ sein soll, also die kürzeste mögliche Zeichenfolge erfassen soll.
- Glob
- Global

- globale Destruktion

- Glue Language

  „Leimsprache“ – verdeutlicht, dass mit Perl Verbindungen zwischen Anwendungen und Bibliotheken die nicht in Perl geschrieben sind und deren Datenformaten hergestellt werden.

- Golf

  Programmierwettbewerb, bei dem es darauf ankommt, ein festgelegtes Problem mit möglichst wenig Zeichen Programmcode zu lösen. Für Perl-Programmierer ist dies eine sehr beliebte Form des Wettbewerbs, da die Sprache der Kreativität breiten Raum bietet.

- Granularität

- grep

- Gruppe

- GV

H

- Hacker

- Handler

- hängende Anweisung

- harte Referenz

I

- indirekt
- indirekter Objekt-Slot
- indirektes Dateihandel
- indirektes Objekt
- Indizierung
- Infix
- Inkrement
- Instanz
- Instanzvariable
- Integer
- Interpolation
- Interpreter
- I/O

- IO
- IP
- IPC
- Iteration
- Iterator
- IV

J

- JAPH

  »Just Another Perl Hacker«, ein cleveres, aber etwas kryptisches Stück Perl-Code, das bei der Ausführung zu diesem String evaluiert. Wird häuffig zur Illustration eines bestimmten Perl-Features verwendet. Außerdem so etwas wie ein fortwährender »Obfuscated Perl Contest« in Newsgruppen-Signaturen.

K

- kanonisch
- Kapselung
- Klasse

- Klassenmethode
- Kollationssequenz
- Kommandozeilenargumente
- Kommentar
- Kompilerungseinheit
- Kompilerungsphase
- Kompilerungszeit
- Konstrukt
- Konstruktor
- Kontext

L

- Label
- Laufzeit
- Laufzeitmuster

- Leere-Subklasse-Test
- Leftmost longest
- lesbar
- Lexem
- Lexer
- lexikalische Analyse
- lexikalsiches Scoping
- lexikalsiche Variable
- LIFO
- Link
- Linkshift
- LISTE
- Liste

- Listenkontext
- Listenoperator
- Listenwert
- Literal
- Little-Endian
- logischer Operator
- lokal
- Lookhead
- Lookbehind
- Lvalue
- Lvalue-fähig
- Lvalue-Modifier

M

- magisch

- magisches Inkrement
- magische Variable
- Makefile
- man

  Programm zur Anzeige von Manual-Seiten.
- Manpage
- Matching
- mehrdimensionales Array

  In Perl bekommt man mehrdimensionale Arrays nur über Umwege hin, da Perl die Arrays einfach hintereinander hängt und einen einzigen Array bildet. Warum? In Perl ist ein Array nur für skalare Daten vorgesehen!
  Lösung: Man baut einen Array von Referenzen auf Arrays. Dies funktioniert wiederum, da Referenzen skalare sind.
- Mehrfachvererbung
- Member-Daten
- Metasymbol
- Metazeichen
- Methode

- Minimalismus
- Modifier
- Modul
- Modulo
- Modus
- momentan gewählter Ausgabekanal
- Monger
- mortal
- Muster

N

- Namensraum
- Nebeneffekte
- Netzwerkadresse

- NFS

- Nibble

  Heute nur noch selten verwendeter Bergriff für 4 zusammenhängende Bits. Ein Byte (8 Bit) hat ein oberes und unteres Nibble.

- Nulllänge

- Null-Liste

- Nullstring

- Nullzeichen

- numerischer Kontext

- NV

O

- Objekt

- Offset

- oktal

- Open-Source-Software

- Operand
- Operator
- Optionen

P

- Pad
- Paket
- Parameter
- Parent-Klasse
- Parsing
- Parsingbaum
- Patch
- PATH

  Der Name bezeicht unter Unix und Windows Betriebssytemen eine Umgebungsvariable mit Pfadangaben zu Verzeichnissen, die ausführbare Dateien enthalten. Trennzeichen unter Unix ist der Doppelpunkt, unter Windows ist es das Semikolon.
- Pattern-Matching

- perl

  Ist die Bezeichnung für das ausführbare Programm, mit und in dem in Perl geschriebene Programme verarbeitet werden. perl für Version 5 der Sprache ist in wesentlichen Teilen in C programmiert. Für die Version 6 gibt es eine Arbeitsversion namens pugs, die in haskell programmiert ist.

- Perl

  Name der Programmiersprache.

- PERL

  Von Anfängern und Menschen die Perl nicht mögen, verwendete Schreibweise für Perl oder perl.

- Pern

- Pfadname

- Pipe

- Pipeline

- Plattform

- POD

- Polymorphismus

- Port

- portabel

- Porter
- POSIX
- Postfix
- pp
- Präfix
- Pragma
- Pragma Module
- Preprocessing
- Programm
- Programmgenerator
- progressives Matching
- Protokoll
- Prototyp

- Prozedur
- Prozess
- Pseudofunktion
- Pseudohash
- Pseudoliteral
- Public Domain

  Zur allgemeinen Verwendung freigegeben. Der Urheber verzichtet auf jegliche Form des Copyrights.
- Puffer
- Pumpkin
- Pumpking
- PV

Q

- qualifiziert

  Die explizite Verwendung eines vollständigen Namens. Das Symbol $Ent::moot ist qualifiziert, $moot ist unqualifiziert. Ein vollständig qualifizierter Dateiname wird vom obersten Verzeichnis aus spezifiziert.

- Quantifier

  Eine Komponente eines regulären Ausdrucks, die festlegt, wie oft das vorstehende Atom vorkommen darf.

- Quellfilter (Source Filter)

  Eine Spezielle Art von Modul, das ein spezielles Preprocessing Ihres Skripts vornimmt, bevor es überhaupt zum Tokenizer gelangt.

R

- Reaping
- Rechtsshift
- Referent
- Referenz
- Regex
- Regex-Modifier
- reguläre Datei
- regulärer Ausdruck
- Rekursion
- relationaler Operator

- reservierte Wörter
- RFC
- Root
- RTFM
- Rückgabewert
- Rückwärtsreferenz
- RV
- Rvalue

S

- Schleife
- Schleifenkontrollanweisung
- Schleifenlabel
- Schlüssel
- Schlüsselwort

- Schnittstelle
- Scratchpad
- Script Kiddie
- sed
- Semaphor
- Sepaerator
- Serialisierung
- Server
- Service
- setgid
- setuid
- Shared Memory
- Shebang

- Shell
- Signal
- Signalhandler
- Skalar
- skalares Literal
- skalarer Kontext
- skalare Variable
- skalarer Wert
- Skript
- Slice
- slurp
- Socket
- Speicher
- Stack

- Standard
- Standard-I/O
- Standardausgabe
- Standardeingabe
- Standardfehler
- Standardwert
- start-Strukor
- statisch
- statische Methode
- statisches Scoping
- statische Variable
- Status
- STDERR

- STDIN
- STDIO
- STDOUT
- Stream
- Subroutine
- Symbol

T

- Tainted
- tatsächliche Argumente
- TCP
- Teilmuster
- Teilstring
- Term
- Terminator

- ternärer (triadischer) Operator
- Text
- Thread
- tie
- Token
- Tokener
- Tokenizing
- Toolboxansatz
- Transliteration
- Trennzeichen
- Trigger
- troff
- Typ

- Type-Casting
- Typedef
- Typeglob
- Typemap
- typisierte lexikalische Variable

U

- Überladung
- Überladung von Operatoren
- Überschreiben
- UDP
- UID
- Umask
- Umgebung
- Umgebungsvariable

V

- Verzeichnishandel
- virtuell
- void-Kontext
- Vorrang
- v-String

W

- wahr

  Jeder skalare Wert, der nicht zu 0 oder „ “ evaluiert.

- Warnung

  Eine Nachricht, die an den Stream „STDERR“ übergeben wird, wenn etwas möglicherweise fehlerhaft, gleichzeitig aber nicht so schlimm ist, dass sofort abgebrochen werden müsste. Beachten sie „warn“ in und das Pragma „use warnings“. Siehe auch Pragma Module.

- Watch

  Watch-Ausdruck; ein Ausdruck, der, wenn sich sein Wert ändert, zu einem Breakpunkt im Perl-Debugger führt.

- weiche Referenz

  Siehe symbolische Referenz. Gegenteil einer harten Referenz.

- Wert

  Ein reales Stück Daten im Gegensatz zu all den Variablen, Referenzen, Schlüsseln, Indizies, Operatoren und was man sonst noch so alles braucht, um auf den Wert zuzugreifen.

- Whitespace

Ein Zeichen, das dem Cursor bewegt, aber nichts auf ihrem Bildschirm hinterlässt. Typischerweise ein Sammelbegriff für die folgenden Zeichen: Leerzeichen, Tabulator, Zeilenvorschub, Wagenrücklauf und Seitenvorschub.

- Wort

  Ein Datenstück von der Größe, mit der Ihr Computer am effizientesten umgehen kann, üblicherweise 32 Bit oder ein, zwei Zweierpotenzen mehr oder weniger. In der Perl-Kultur bezeichnet es häufiger einen alphanumerischen Identifier (inklusive Unterstriche) oder einen String, der selbst keine Whitespaces enthält, aber durch Whitespace oder Stringgrenzen von anderen abgegrenzt wird.

- Wrapper

  Ein Programm oder eine Subroutine, das bzw. die ein anderes Programm oder eine andere Subroutine für sie ausführt und dabei einige seiner bzw. ihrer Ein- und Ausgaben modifiziert, um Ihren Absichten besser dienen zu können.

X

- XS

  Eine außergewöhnlich exportierte, extrem schnelle, ... Subroutine, die in C oder C++ oder in einer neuen Erweiterungssprache namens XS ausgeführt wird.

- XSUB

  Eine in XS definierte Subroutine

Y

- yacc

  »Yet Another Compiler Compiler« Ein Parser-Generator, ohne den es Perl wahrscheinlich nie gegeben hätte. Sehen sie sich die Datei perly.y in der Perl-Quelldistribution an.

Z

- Zeichen

  Ein kleiner Integerwert, der eine orthografische Einheit repräsentiert.

- Zeicheneigenschaft

  Eine Vordefinierte Zeichenklasse, die über das Metasymbol erkannt werden kann. Viele Standardeigenschaften sind für Unicode definiert.

- Zeichenklasse

  Eine in eckigen Klammern stehende Liste von Zeichen. Wird bei regulären Ausdrücken verwendet, um anzuzeigen, dass irgendeines dieser Zeichen an dieser Stelle vorkommen darf. Allgemeiner: jeder vordefinierte Satz von Zeichen, der auf diese Weise verwendet wird.

- Zeiger

  Bei Sprachen wie C eine Variable, die die genaue Speicherposition eines anderen Objekts enthält. Perl behandelt Zeiger intern, d.h. Sie müssen sich weiter keine Gedanken darum machen. Statt dessen arbeiten Sie nur mit symbolischen Zeigern in Form von Schlüsseln und Variablennamen oder mit harten Referenzen, die keine Zeiger sind (sich aber wie Zeiger verhalten und tatsächlich auch Zeiger enthalten).

- Zeile

  Bei Unix eine Reihe von null oder mehr Zeichen ohne Zeilenvorschubzeichen, die durch ein Zeilenvorschubzeichen beendet werden. Bei Maschinen, die nicht mit Unix arbeiten, wird dies emuliert, selbst wenn die C-Bibliothek des verwendeten Betriebssystems hiervon andere Vorstellungen hat.

- Zeilennummer

  Die Anzahl der Zeilen, die vor einer gegebenen Zeile gelesen wurden, plus 1. Perl verwaltet für jedes Script und jede Eingabedatei eine seperate Zeilennummer. Die Zeilennummer des aktuellen Scripts wird durch repräsentiert. Die aktuelle Zeilennummer der Eingabe (für die Datei aus der zuletzt etwas über gelesen wurde) wird durch $.($INPUT_LINE_NUMBER) repräsentiert. Viele Fehlermeldungen geben, wenn vorhanden, beide Werte aus.

- Zeilenpufferung

  Wird von einem Standard-I/O-Ausgabestream verwendet, der seinen Puffer nach dedem Zeilenvorschub leer. Viele Standard-I/O-Bibliotheken richten dies automatisch bei Ausgabe ein, die an ein Terminal gehen.

- Zeilenvorschub

  Ein einzelnes Zeichen, das das Ende einer Zeile repräsentiert. Bei Unix hat es des ASCII Wert 012 oktal (aber den Wert 015 bei Macs) und wird in Perl-Strings durch \n repräsentiert. Bei Windows-Maschinen, die Textdateien schreiben, und bestimmten physischen Geräten wie Terminals wird dieses Zeichen von Ihren C-Bibliothek in einen Zeilenvorschub und einen Wagenrücklauf übersetzt. Normalerweise erfolgt aber keine Übersetzung.

- Zirkunfix-Operator

  Ein Operator, der seinen Operanden umschließt wie etwa der Zeileneingabeoperator, runde Klammern oder Koalas. (»Zirkumfix« bedeutet als linguistischer Begriff einen Kombination aus Präfix und Suffix. Im Deutschen treten Zirkumfixe z.B. oft in Wörtern wie gesündigt auf.)

- Zombie

  Ein beendeter Prozess, dessen Parents von „wait“ oder „waitpid“ noch nicht über dessen Ableben informiert wurden. Wenn Sie mit „fork“ arbeiten, müssen Sie beim Beenden Ihrer Child-Prozesse hinter diesen aufräumen, weil sich andernfalls die Prozesstabelle füllt und ihr Systemadministrator mit Ihnen nicht sehr zufrieden sein wird.

- Zugriffsflags

  Bits, die vom Besitzer einer Datei gesetzt oder gelöscht werden, um anderen Leuten den Zugriff zu erlauben oder zu verweigern. Diese Flags sind Teil des Modus-Wortes, der vom „stat“-Operator zurückgegeben wird, wenn Informationen über eine Datei angefordert werden. Bei Unix-Systemen können Sie sich die Manpage zu ls lesen, wenn sie weitere Informationen über Zugriffsflags wünschen.

- Zusicherung

- Zuweisung

Eine Anweisung,die den Wert einer Variablen ändert. Syntaktisch wird eine Zuweisung mit einem Zuweisungsoperator notiert.

- Zuweisungsoperator

  Operator, mit dem eine Zuweisung notiert wird.
  Der einfache Zuweisungsoperator ist das Gleichheitszeichen. Wie in anderen Sprachen gibt es auch in Perl zusammengesetzte Operatoren, die zunächst mit dem Wert der Variablen eine Operation durchführen und das Ergebnis dann in der Variablen ablegen.
  Beispiele:

Variable Wert zuweisen $n = 42

Variablenwert um drei erhöhen $n += 3

Kapitel 30

Autoren

Edits	User
34	MGla
4	Caveman
3	Japh
30	Glauschwuffel
2	Jan
3	Keyanoo
2	André Bonhôte
2	Hernani
2	Daniel B
1	Gobold
1	FeG
8	Count Adder
1	Klartext
29	Lichtkind
1	Braegel
6	Merkel
3	Klaus Eifert
1	Gronau
1	Betterworld
3	WeißNix
9	Heuler06
1	Sentropie
2	Transporter
1	Penma
10	Hubi
3	Wutzofant

72	Giftnuss
21	Nowotoj
93	Turelion
7	Daniel Mex
2	Myrkr
92	Ap0calypse
8	MichaelFrey
5	Sanduar
11	Mark e
3	ThePacker
11	Sid Burn
1	Ramiro
7	Taulmarill
13	Manuels
1	Jogi
1	Bastie
1	E(*nix*)
19	Plaicy

Kapitel 31

Bildnachweis

In der nachfolgenden Tabelle sind alle Bilder mit ihren Autoren und Lizenen aufgelistet.

Für die Namen der Lizenzen wurden folgende Abkürzungen verwendet:

- GFDL: Gnu Free Documentation License. Der Text dieser Lizenz ist in einem Kapitel diese Buches vollständig angegeben.
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/3.0/ nachgelesen werden.
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.5/ nachgelesen werden.
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.0/ nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.0/ nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- PD: This image is in the public domain. Dieses Bild ist gemeinfrei.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

Bild	Autor	Lizenz

Kapitel 32

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section “Entitled XYZ“ means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements“, “Dedications“, “Endorsements“, or “History“.) To “Preserve the Title“ of such a section when you modify the Document means that it remains a section “Entitled XYZ“ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.

- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- H. Include an unaltered copy of this License.

- Preserve the section Entitled “History“, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of

the Modified Version as given on the Title Page. If there is no section Entitled “History“ in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History“ section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. For any section Entitled “Acknowledgements“ or “Dedications“, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- M. Delete any section Entitled “Endorsements“. Such a section may not be included in the Modified Version.

- N. Do not retitle any existing section to be Entitled “Endorsements“ or to conflict in title with any Invariant Section.

- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements“, provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History“ in the various original documents, forming one section Entitled “History“; likewise combine any sections Entitled “Acknowledgements“, and any sections Entitled “Dedications“. You must delete all sections Entitled “Endorsements.“

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate“ if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggre-

gate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.