
Oldenbourg

Perl

Grundlagen und effektive Strategien

von
Prof. Dr. Jürgen Schröter
Fachhochschule Landshut
Unter Mitwirkung von
Helmut Seidel

Oldenbourg Verlag München

Die Titelabbildung „Perl-Katze“ von Peter Kornherr gibt einen Vorgeschmack auf die zahlreichen lustigen Katzen-Beispiele im Kapitel „Reguläre Ausdrücke“.

Ebenfalls bei Oldenbourg erscheint von den Autoren Seidel und Schröter der weiterführende Band „Perl – Anwendungen und fortgeschrittene Techniken“, ISBN: 978-3-486-25902-5. Im zweiten Band werden aufbauend auf dem vorliegenden Buch Pakete, Module, Datenbanken, graphische Oberflächen mit Perl/TK und die CGI-Programmierung behandelt.

Bibliografische Information der Deutschen Nationalbibliothek

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

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Lektorat: Irmela Wedler, Christian Kornherr
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Books on Demand GmbH, Norderstedt

ISBN 978-3-486-25889-9

Inhalt

Vorwort

Als Leiter des Rechenzentrums der University of Applied Sciences Landshut halte ich Perl besonders im administrativen Bereich für eine unverzichtbare Skriptsprache. Unter Unix/Linux wird sie vor allem eingesetzt für Systempflege, Skripting auf Servern, für dynamisches Webcontent, Erstellung von CGI (**C**ommon **G**ateway **I**nterface), für Programme im Web-Umfeld, Datenbankanbindung, grafische Benutzerschnittstellen (GUI – **G**raphic **U**ser **I**nterface), um nur einige Aufgabenbereiche zu nennen. Besonders schätze ich ein weiteres Highlight von Perl, nämlich die regulären Ausdrücke, die ein flexibles Suchen nach Mustern und ein leistungsfähiges Ersetzen ermöglichen.

Ebenso haben mich die reichhaltigen Spracheigenschaften von Perl überzeugt. Zu nennen sind hier die einfache Verarbeitung von wirklich großen Datenmengen, die umfangreiche Bearbeitung von Strings, die Verwaltung und Bearbeitung von Hashes (assoziative Arrays) sowie die einfache Verwendung von Arrays. Nicht vergessen möchte ich die objektorientierte und die funktionale Programmierung. Für entsprechende Operationen brauchen Sie in Perl nur die entsprechenden Befehle einzusetzen, statt umfangreiche Funktionen neu zu erstellen.

Weitere Pluspunkte, die vor allem Studentinnen und Studenten hoch bewerten werden, ist der kostenlose Bezug von Perl und Debugger sowie die außergewöhnliche Hilfsbereitschaft der großen Perl-Fan-Gemeinde.

Ich würde mir wünschen, dass beim Lesen dieses Buches der Funke der Begeisterung für Perl bei Ihnen ebenso überspringt wie bei mir, der immer noch begeistert von Perl ist.

Dank

An dieser Stelle möchte ich allen jenen danken, die mich beim Schreiben dieses Buches unterstützt haben:

Zuallererst danke ich meinem Mitstreiter Helmut Alfons Seidel, M.A., für die engagierte Zusammenarbeit bei unseren Projekt Perl. Seine konstruktiven Anregungen und Ideen gaben wichtige Impulse für dieses Buch.

Ebenfalls danken möchte ich Prof. Dr. Udo Müller von der University of Applied Sciences Karlsruhe für die Überlassung einiger Skriptseiten. Diese und weitere Hinweise von ihm waren besonders in der Planungsphase des Buches eine große Hilfe für mich.

Ich danke den äußerst engagierten Studierenden meiner PG (Perl Group) Frau Manuela Gilch, Frau Petra Merkel, Frau Johanna Kasberger, Frau Martina Voglmeier, Herrn Friedrich Gais, Herrn Markus Mayer, Herrn Andreas Muntean und Herrn Georg Mösenlechner. Dabei möchte ich besonders Herrn Dipl.-Ing. Tim Bragulla und Herrn Dipl.-Ing. Tobias Fritz erwähnen, die sich mit faszinierender Kreativität und Lebendigkeit in Perl eingearbeitet haben.

Mein aufrichtiger Dank gilt Irmela Wedler, Lektorin Informatik/Mathematik der Oldenbourg Wissenschaftsverlag GmbH, die dieses Projekt intensiv und engagiert betreut und stets ein offenes Ohr für Wünsche und Änderungen hatte.

Last but not least danke ich meiner Frau Heidemarie Schröter die mein Vorhaben immer unterstützt hat, obwohl viele private Aktivitäten dabei zu kurz gekommen sind.

Sie hat mich vor allem dazu ermutigt komplizierte Inhalte möglichst einfach wiederzugeben und Einfaches nicht zu kompliziert auszudrücken.

Jürgen Schröter

Landshut

1 Einstieg in Perl

Falls Sie kein System besitzen, auf dem Perl installiert ist, müssen Sie eine Perl-Umgebung (Perl-Distribution) laden und installieren. Die Distributionen enthalten wichtige, detaillierte Installationsanweisungen.

1.1 Installation unter Unix/Linux-Systemen

Die Installation einer binären Version (ausführbare EXE-Datei) brauchen Sie nicht kompilieren. Sie können gleich nach dem Entpacken mit Perl beginnen.

Für die Installation des Quellcodes benötigen Sie die zusätzlichen Tools tar und gzip, um das Quell-Archiv zu entzippen, und einen C-Compiler.

Den Quellcode finden Sie unter *http://www.perl.com*, wo sich unter *http://www.perl.com/CPAN/src/stable.tar.gz* die neueste Version von Perl befindet.

1.2 Installation unter Win32-Systemen

Auch hier können Sie zwei Wege einschlagen. Zum einen können Sie den Quellcode herunterladen, den Sie anschließend kompilieren müssen, zum anderen können Sie die ausführbare Version – auch ActivePerl genannt – von ActiveState herunterladen.

Mit dem Herunterladen des Quellcodes besitzen Sie immer den neuesten Perl-Stand. Sie müssen jedoch zum Kompilieren über einen neuen C++-Compiler (Visual C++, Borland C++ usw.) verfügen. Ferner müssen noch Win32-Module (libwin32) vorhanden sein, um Zugriffe auf Windows-Optionen, wie OLE und Prozesse, zu ermöglichen.

Die neueste Version von Perl finden Sie bei ActiveState (*http://www.activestate.com*), wo Sie noch ein Perl Developer Kit, einen Perl Debugger mit grafischer Oberfläche und ein Plug-in zur Verbesserung von CGI finden.

Den Quellcode von Perl finden Sie unter *http://language.perl.com/CPAN/src/*, wo Sie unter *stable.tar.gz* wieder die neueste stabile Version finden.

Der Perl-Quellcode ist wie bereits erläutert im Unix-Format in tar-Archiven gespeichert und mit GNU Zip komprimiert. Haben Sie dieses Archiv auf Ihrer Festplatte gespeichert, können Sie die Quellcode-Dateien mit WinZip problemlos dekomprimieren und dearchivieren.

2 Warum Perl?

Perl steht für **P**ractical **E**xtraction and **R**eport **L**anguage und ist von Larry Wall entwickelt worden.

Zunächst ist es Larry Wall gelungen, eine extrem ausdrucksstarke Sprache zu entwickeln. Weiterhin ähnelt Perl mehr einer lebendig gesprochenen Sprache im Vergleich zu den Programmiersprachen, die einen genau definierten Wortschatz und eine an der Hardware ausgerichtete Grammatik besitzen. Somit ist Perl nicht so starr wie andere Sprachen. Auf Grund der Vielschichtigkeit verschiedener Sprachkonzepte und Sprachkulturen ist Perl eine Heimstätte für viele Programmierer geworden, egal von welcher Sprache sie beeinflusst worden sind. Sie können unter Perl Ihren persönlichen Programmierstil weiter fortführen.

In Perl können Sie objektorientiert programmieren, da die benötigten Mechanismen vorhanden sind. Somit können Sie nicht nur große Projekte sinnvoll strukturieren. Mit der Vererbung lässt sich z. B. viel Programmcode einsparen.

Des Weiteren profitiert Perl von den vielfältigen Einsatzgebieten, besonders im Internet und Web. Client-Server Interaktionen, z. B. SQL-Anweisungen, die an einen Datenbankserver gesendet werden, und die vom Server zurücklaufenden Ergebnisse, werden in Perl realisiert. Ebenfalls können Textprotokolle wie CGI (**C**ommon **G**ateway **I**nterface) mit Perl durchgeführt werden. Ferner führt Perl Internetsites aus, die dynamisch generierte Inhalte oder Suchfunktionen zur Verfügung stellen. Natürlich lassen sich außer HTML-Codes auch C-Codes über Stubs in Perl einbetten. Somit lassen sich geschriebene C-Programme und Bibliotheken mit Perl aufrufen. Auch die Automation der Systemadministration und die Datensicherung gehören zu den Aufgaben von Perl, ebenso wie das weite Feld der Textverarbeitung mit Dateibehandlung.

Für Einsteiger ist Perl eine leicht verständliche Sprache und bietet noch weitere Besonderheiten, wie z. B. die Verwendung von regulären Ausdrücken, mit denen Sie einen Text nach Wörtern, Textpassagen oder Zeichenmustern durchsuchen können. Mit diesen regulären Ausdrücken können Sie Ihre Perl-Programme schneller, kürzer, einfacher und somit wesentlich effektiver schreiben.

Um für das entsprechende Problem das richtige Modul zur Lösung zu finden, können Sie auf das CPAN (**C**omprehensive **P**erl **A**rchive **N**etwork) zugreifen. So zum Beispiel ist Perl eigentlich nicht dazu ausgelegt, schwerpunktmäßig eine Verarbeitung und Handhabung von numerischen Daten vorzunehmen. Aber auch hier existieren Module für beliebig große Ganzzahlen und Gleitpunktzahlen sowie das PDL-Modul für die Berechnung von großen Matrizen.

Perl gilt als mächtige Sprache, da eine große Anzahl von Unix-Tools auf Perl angepasst worden sind. Sie können Anwendungen beispielsweise mit wenigen Zeilen Perl-Programmcode (Hashes, regulären Ausdrücken, Referenzen usw.) auszuführen, während Sie in anderen Sprachen dazu mehrere Seiten schwer verständlichen Programmcode benötigen. Weiterhin können Sie einen konfortablen Perl-Debugger verwenden, der Ihnen alle Möglichkeiten bietet, Fehler professionell zu beheben.

Perl ist eine frei verfügbare Sprache. Weder für den Interpreter noch für die vielen frei verfügbaren Module aus dem CPAN brauchen Sie etwas zu bezahlen. Sie finden Perl bzw. die entsprechenden Module auf den gut organisierten FTP-Servern, die im Anhang angegeben sind.

Um Ihnen weitere positive Merkmale an die Hand zu geben, die für Perl sprechen, sollen im folgenden Abschnitt neben Perl die interpretierenden Sprachen wie Java, Tcl und VisualBasic dargestellt werden.

Interpreter und Compiler

Neben den interpretierten Sprachen gibt es auch noch kompilierte Sprachen. Bei den interpretierenden Sprachen wird jede Anweisung des Programms von einem Interpreter gelesen und in den entsprechenden Maschinencode übersetzt. Fehler werden während der Programmausführung erkannt. Manche Interpreter interpretieren den Quelltext zeilenweise, somit wird Anweisung für Anweisung eingelesen, interpretiert und ausgeführt. Wie arbeitet nun der Perl-Interpreter **perl**? Wenn Sie Ihren Quellcode (Skript) ausführen wollen, wird zunächst Ihr gesamtes Programm eingelesen (eingebundene Module werden mit berücksichtigt) und optimiert in einen internen Pseudo-Maschinencode (Bytecode) übersetzt (kompiliert). Etwaige Syntax- und Semantikfehler werden sofort angezeigt. Danach erfolgt ein effizientes Interpretieren mit Ausführung. Gegenüber dem zeilenweisen Übersetzen mit Ausführung hat dies den Vorteil, dass einerseits die Qualität besser und anderseits die Ausführung schneller ist.

Bei den Sprachen, die kompiliert werden, wird das Programm von einem Compiler gelesen und in Maschinencode übersetzt, den dieser in eine ausführbare Datei abspeichert. Letztere lässt sich dann aufrufen und ausführen. Vom Compiler wird somit ein plattformspezifischer Maschinencode erzeugt. Fehler werden vor der Ausführung erkannt.

Auch bei Perl wird das ganze Programm vom Interpreter übersetzt, bevor es ausgeführt wird. Somit läuft hier die Ausführung ähnlich schnell ab wie die Ausführung eines kompilierten Programms. Ein Wermutstropfen ist aber beim Interpretieren gegenüber dem Kompilieren vorhanden: Sie müssen jedesmal den Quelltext Ihres Perl-Skriptes neu übersetzen.

Wie sieht es nun mit den oben genannten Programmiersprachen aus?

Java übersetzt seinen Quellcode per Compiler und speichert den Bytecode in eine ausführbare Datei. Dieser wird dann von einer sogenannten Virtual Machine ausgeführt. Hier ist wieder der Vorteil vorhanden, dass das Programm fehlerfrei ist, bevor es ausgeführt wird. Da der Bytecode von der Virtual Machine ausgeführt wird, lässt sich eine hohe Sicherheit erzielen.

Da aber erhebliche Anforderungen an die Hardware gestellt werden, erfolgt die Ausführung nicht allzu schnell.

Tcl interpretiert seinen Quellcode Zeile für Zeile und führt ihn auch so aus. Diese Vorgehen ist äußerst langsam. Eine deutliche Verbesserung erreichen Sie, wenn Sie übersetzte Tcl-Programme in C-Code übersetzen lassen.

VisualBasic verfolgt zwei Ansätze: Zum einen kann der Quellcode in einen P-Code übersetzt werden. Dieser Zwischencode – zwischen Quellcode und Binärcode – wird vom Interpreter ausgeführt. Zum anderen kann der Quellcode in einen Nativen Maschinencode (quasi Binärcode) übersetzt und dann ausgeführt werden.

Plattformunabhängigkeit

Perl ist auf jeder Unix-Version – vom PC bis hin zum Großrechner – ablauffähig. Auch Win32-Systeme und reine MS-DOS-PCs können Perl ausführen. Zusätzlich bietet Perl aber noch eine Reihe von Zusatzmodulen, die ganz gezielt die Stärken des jeweiligen Betriebssystems ausnutzen können. So gibt es eine Reihe von Modulen, welche die Stärken von Microsoft-Windows ausnutzen, wie ODBC, ADO, OLE, um nur einige zu nennen.

Wie sieht nun die Plattformunabhängigkeit bei den oben genannten Sprachen aus?

Java erreicht die Plattformunabhängigkeit durch die Verwendung der Virtual Machine. Durch die enorme Vielzahl an Werkzeugen sind Sie in der Lage, in sich geschlossene Systeme zu erstellen.

Tcl erreicht die Plattformunabhängigkeit dadurch, dass in vielen Bereichen der kleinste gemeinsame Nenner der verschiedenen Betriebssysteme gefunden wurde. Dies führt dazu, dass diese sehr spartanisch wirken, sofern Sie portable Programme schreiben möchten.

Gegenüberstellung von Programmen

Die Einführung einer neuen Programmiersprache wurde auf Grund langer Tradition immer mit der berühmten Ausgabe "Hello World" begonnen. In der folgenden Gegenüberstellung wird selbstverständlich auch daran festgehalten. Es wird aber zusätzlich noch Ihr Name eingegeben, der dann mit dem Gruß "Hallo" wieder ausgegeben wird.

Java:

```
import java.io.* ;public class Hello {
      public static void main (String[] args) {
      String zeile = "";System.out.print("Geben Sie bitte
      Ihren Namen ein: ")try {
                  BufferedReader in = new BufferedReader(new
                  InputStreamReader(System.in));
                  zeile = in.readLine();in.close();
            }catch (IOException err) {
                  System.out.println("Fehler: "+
                  err.getMessage());
```

```
            }System.out.println("Hallo " + zeile + "!");
      }
}
```

Tcl:

```
puts "Geben Sie bitte Ihren Namen ein:"
gets stdin name
puts "Hallo $name!"
```

VisualBasic:

```
' Funktioniert mit VB.NETImports SystemPublic Modul modmain

Sub Main()
      dim strName as StringConsole.Write("Geben Sie bitte
      Ihren Namen ein. ")strName = Console.ReadLine()
      Console.WriteLine("Hallo " & strName & "!")
End Sub
End Module
```

Perl:

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

print "Geben Sie bitte Ihren Namen ein: ";
$name = <STDIN>;
print "Hallo $name!";
```

Wie Sie deutlich sehen können, besticht das Perl-Programm in seiner Kürze und Einfachheit. In Perl existieren viele Programme nur mit einem Einzeiler. Es wird nur eine Zeile zur Berechnung benötigt. Perl fordert kein Objektmodul. Im Gegensatz zu Java sind Sie nicht verpflichtet, Klassen zu verwenden. Sie können Ihre Skripte weiterhin OO-frei schreiben. Perl verlangt auch keine Festlegung seiner Variablen. Sie können nicht nur das obige Programm leicht lesen und verstehen. Sie werden feststellen, dass Sie Perl schnell erlernen und anwenden können. Bereits die Kenntnisse der nächsten zwei Kapitel reichen aus, um nützliche Programme zu schreiben. In den weiteren Kapiteln können Sie Ihren Wissensstand erweitern und mit den vorhandenen Features mächtige Programme schreiben.

Möchten Sie Perl-Programme auf einem Unix-System laufen lassen, brauchen Sie nur `#!/usr/bin/perl -w` an die erste Zeile setzen. Diese erste Zeile `#!/usr/bin/perl -w` wird nur für Unix-Systeme benötigt und ist die Shebang-Zeile ("sh" für Sharp, "bang" für das Ausrufezeichen). Wie bei einem Shell-Skript unter Unix legt die erste Zeile den Interpre-

ter perl fest, der auf dem verwendeten Rechner unter dem Verzeichnis `/usr/bin/` zu finden ist. Das Flag `-w` ist für die Ausgabe von Warnungen gedacht.

Wenn Sie unter Windows oder Mac arbeiten, benötigen Sie diese Zeile nicht. Sie stört aber nicht, da das #-Zeichen von Perl als ein Kommentar gewertet wird und auf anderen Plattformen als Unix ignoriert wird. Sie sollten diese Zeile also trotzdem verwenden, wenn Ihr Programm eventuell einmal unter Unix ablaufen sollte.

3 Grundlagen

In diesem Kapitel werden die wichtigsten Kategorien von Variablen dargestellt und die Gültigkeitsbereiche von benutzerdefinierten Variablen erläutert. Nicht vergessen werden die zwar unsichtbaren, aber immer präsenten Spezial-Variablen von Perl. Im Anschluss erfahren Sie alles über das Thema Kontext. Ebenso wichtig ist der Begriff Wahrheitswert.

3.1 Variablen

Um Werte in Ihrem Programm zu speichern, benötigen Sie Variablen. Perl hat drei Datentypen von Variablen:

- Skalare Werte wie Zahlen, Strings (Zeichenketten) und Referenzen lassen sich in skalaren Variablen speichern:

```
# Zahlen
$wert_1 = 4711;                    # Ganze Zahl(integer)
$wert_2 = 3.14159;                 # Gleitkommazahl

# Strings
$string_1 = "Hello World!";        # Zeichenkette
$string_2 = 'Ein String';          # Zeichenkette

# Referenzen
$ref_1 = \$wert_1;                 # Referenz von $wert_1
```

- Listen als geordnete Sammlung von skalaren Variablen lassen sich in Array-Variablen speichern:

```
# Listen
@array_1 = (123, 4711);                  # Homogene Liste
@array_2 = (1.3, \$ref, 'string');       # Heterogene Liste
```

- Schlüssel-Wert-Paare (Key-Values) werden in Hash-Variablen (assoziative Arrays) gespeichert:

```
# Hashes
%hash_1 = (ilse => 26, ida => 32);
# Homogenes Hash
%hash_2 = (skalar => $wert_1, %hash_1);
# Heterogenes Hash
```

Die Variablen, in denen diese Datentypen repräsentiert werden, unterscheiden sich durch vorgestellte Kennungen oder Präfixe (`$`, `@`, `%`).

In Perl müssen Variablen vor ihrer Verwendung im Gegensatz zu anderen Programmiersprachen nicht deklariert werden. Nicht deklarierte Variablen sind undefiniert.

Variablen können innerhalb des Programms ihren Typ wechseln (sie sind schwach typisiert). So kann eine skalare Variable zunächst eine Zahl, dann einen String oder später sogar eine Referenz enthalten.

3.1.1 Variablen-Namen

Ein gültiger Name für Variablen besteht aus Buchstaben, Zahlen und Unterstrichen. Am Anfang der Variablen muss immer ein Buchstabe oder ein Unterstrich stehen. Anschließend können beliebig viele Buchstaben, Ziffern oder Unterstriche folgen. Umlaute sowie Sonderzeichen sind im Namen verboten. Vor diesem Namen befindet sich der entsprechende Präfix des jeweiligen Typs. Große und kleine Buchstaben werden in der Namensgebung unterschieden (case-sensitive). Um Programmfehler durch ungewollte Neueinrichtung von Variablen zu vermeiden, empfiehlt es sich, das Pragma **use strict** zu verwenden.

3.1.2 Zustand und Gültigkeitsbereiche von Variablen

In diesem Abschnitt sollen einige Funktionen genannt werden, die Ihnen den Zustand bzw. den Gültigkeitsbereich einer Variablen bekannt geben.

Die Funktion `defined()`
`defined` *ausdruck*

Die Funktion `defined()` liefert den Booleschen Wert `true` zurück, wenn der Ausdruck *ausdruck* definiert ist. Wenn kein Ausdruck angegeben ist, wird die Spezial-Variable `$_` ausgewertet.

Die Funktion `undef()`
`undef` *ausdruck*

Die Funktion `undef()` setzt den Ausdruck *ausdruck* – der links vom Gleichheitszeichen stehen muss (*lvalue*) – auf einen undefinierten Wert und gibt `undef` zurück.

Die Funktion `my()`
`my` *variable*

Die Funktion `my()` deklariert eine oder mehrere private Variablen, die nur bis zum nächsten umschließenden Block, nächsten Modul oder bis zur nächsten Funktion gültig sind. Beim Aufruf einer Funktion aus einer Funktion haben Sie keinen Zugriff mehr auf die zurückliegenden Variablen. Technisch gesehen haben `my`-Variablen einen lexikalischen Gültigkeitsbereich.

Die Funktion `local()`
`local` *variable*

Die Funktion `local()` deklariert eine oder mehrere globale Variablen. Ihre Gültigkeit endet mit dem Ende des dazugehörigen Blocks, Moduls oder der zugehörigen Funktion. Mit der Funktion `local()` deklarieren Sie die lokalen Variablen, auf die Sie auch von anderen Funktionen aus zugreifen können. Technisch gesehen haben local-Variablen einen dynamischen Gültigkeitsbereich.

3.2 Spezial-Variablen

Perl kennt eine Reihe von speziellen Variablen, die nicht deklariert werden müssen, sondern bereits vorhanden sind. Sie geben nicht nur Informationen über den aktuellen Zustand des Systems bekannt, sondern können auch zur Steuerung von Abläufen dienen. Dies können sowohl skalare Variablen als auch Arrays und Hash-Variablen sein. Bei den meisten dieser Spezial-Variablen existiert eine Kurzform, bestehend aus zwei bis drei Zeichen, und eine Langform, bestehend aus aussagekräftigen Namen in Großbuchstaben. Wenn Sie die Langform verwenden möchten, müssen Sie das Pragma **use English** aktivieren. An dieser Stelle sollen nachfolgend nur die wichtigsten aufgeführt werden. Umfangreiche Informationen hierzu finden Sie unter PERLVAR-Dokumentation.

Die Spezial-Variable `$_`
Die wohl meist gebrauchte Spezial-Variable ist `$_`. Für viele Perl-Funktionen wird sie als Default-Argument eingesetzt, wenn kein Argument beim Funktionsaufruf übergeben wird. Zum anderen wird sie für Datei-Testoperationen, oder für Operationen in der Mustersuche verwendet. Vor allem findet sie ihren Einsatz in `foreach`-Schleifen, wenn keine Schleifenvariable angegeben ist.

Die Spezial-Variable `$!`
Sollte eine Perl-Durchführung nicht erfolgreich sein, enthält die Spezial-Variable `$!` die genauere Beschreibung des Fehlers. Im numerischen Kontext wird die entsprechende Fehlernummer der System-Variablen `errno` zurückgegeben. In einem Stringkontext wird die

jeweilige Zeichenkette zurückgeliefert. Ausführlichere Informationen erhalten Sie durch die System-Variable `$^E`.

Die Spezial-Array-Variable @_
Eine ebenso wichtige Rolle wie `$_` spielt die Spezial-Array-Variable `@_`. Das Array enthält alle Parameter, die beim Aufruf einer Funktion übergeben werden. Die Skalare von `@_` lassen sich wiederum über die Spezial-Variable `$_` mit entsprechenden Indizes ansprechen. So ist das erste Element durch `$_[0]`, das zweite durch `$_[1]` usw. ansprechbar.

Das Spezial-Array @ARGV
Das Spezial-Array `@ARGV` enthält die Kommandozeilen-Argumente, die beim Aufruf des Programms übergeben wurden. Die Argumente stehen dann in der entsprechenden Eingabereihenfolge im Array `@ARGV` zur Verfügung.

Das Spezial-Hash %ENV
Das Spezial-Hash `%ENV` enthält sämtliche aktuellen Umgebungsvariablen des verwendeten Systems. Dabei ist der Schlüssel der Name der Umgebungsvariablen, der Wert ihr aktueller Inhalt.

Reservierte Variablen für allgemeine Informationen
Mit weiteren reservierten Spezial-Variablen lassen sich Informationen über verschiedene Systemzustände ermitteln. So gibt die Spezial-Variable `$]` (alternativ `$PERL_VERSION`) die Perl-Versionsnummer an. Mit `$0` (alternativ `$PROGRAM_NAME`) erhalten Sie den Name der Datei, die das ausgeführte Programm enthält. Durch Verwendung der Spezial-Variable `$^O` (alternativ `$OSNAME`) erfahren Sie den Name des Betriebsystems.

Wie bereits erwähnt, sollte hier nur eine kleine beschränkte Auswahl von Spezial-Variablen dargestellt werden.

3.3 Kontext

In Perl definieren Operatoren und Funktionen einen bestimmten Kontext. Variablen sind kontext-sensitiv, da sie Werte mit verschiedene Kontexten liefern, die auch unterschiedlich interpretiert werden. Von einer Variablen in einem skalaren Kontext wird nur ein einzelner Wert erwartet. Im Listenkontext wird eine ganze Liste erwartet. Möchten Sie einen Wahrheitswert bei einer Entscheidung erfahren, entspricht dies einem Booleschen Kontext. Ebenso können Sie skalare Werte in einen numerischen Kontext (Zahl) bzw. Stringkontext (Zeichenkette) umwandeln.

Funktionen können im Skalar- und Listenkontext unterschiedliche Ergebnisse zurückliefern. Im folgenden Beispiel wird zunächst eine einzige Zeile eingelesen:

```
$zeile = <STDIN>;                    # Skalarer Kontext
```

Im nächsten Beispiel werden so lange Zeilen eingelesen, bis das Dateiende erreicht ist:

```
@liste = <STDIN>,                    # Listenkontext
```

Es existieren in Perl Funktionen, bei denen der Kontext unterschiedliche Ergebnisse zurückliefert. Die Funktion `localtime()` liefert z. B. im Listenkontext eine Liste von unformatierten Datums- und Zeitinformationen zurück, während durch den Operator `scalar` ein skalarer Kontext erzwungen wird, was eine formatierte Datums- und Zeitinformation bedeutet. Der Zeilenvorschub (New-Line `\n`) in der jeweiligen `print`-Anweisung dient der übersichtlicheren Ausgabe. Jede Ausgabezeile enthält das entsprechende Ergebnis. Zur Kontrolle können Sie die Ergebnisse vergleichen, die bei Ihnen auf dem Bildschirm und als Kommentar (#) im Programm angegeben wurden:

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

print localtime,"\n";                # 9714721013640
print scalar localtime,"\n";         # Wed Mar 7 14:07:09 2001
```

In der folgenden Tabelle sind die Kontexte aufgeführt.

Tabelle 3.3: Kontexte

Kontext	Bedeutung
Skalarer Kontext	Es wird ein einziger skalare Wert erwartet
Listenkontext	Es wird eine Liste erwartet
Boolescher Kontext	Wahrheitswert `true` oder `false`
Numerischer Kontext	Es wird eine Zahl erwartet
Stringkontext	Es wird ein String erwartet
Void Kontext	Es wird kein Wert erwartet

3.4 Wahrheitswert

Perl kennt keinen eigenen Booleschen Datentyp; somit stellen Wahrheitswerte eine besondere Form von Skalaren dar. Skalare Werte gelten als logisch wahr (`true`), wenn eine Zahl ungleich 0 ist oder ein String (Zeichenkette) kein leerer String ist. Also gelten eine Zahl 0, ein leerer String und ein undefinierter Wert als `false`. Jeder andere skalare Wert gilt als `true`. Sie können alle skalaren Daten auf wahr (`true`) oder falsch (`false`) prüfen.

4 Skalare

Perl-Variablen können in Form von Skalaren auftreten, die entweder numerische Daten oder Strings (Zeichenketten) oder auch beides enthalten können. Einer skalaren Variablen können Sie jeden Wert zuweisen, der sich in Perl darstellen lässt. Perl wandelt je nach Bedarf Werte automatisch von der numerischen Darstellung in die String-Darstellung um oder umgekehrt. Strings, die Oktal- oder Hexadezimalwerte darstellen, werden nicht automatisch umgewandelt. Hier müssen Sie die Funktionen `oct()` und `hex()` verwenden. Sie können Zahlen wie 4711 oder 3.14159e00 oder auch Zeichenketten beliebiger Länge verwenden. Die Länge der Daten ist beliebig; somit kann es sich um ein Zeichen oder auch um eine Datei von mehreren Megabytes handeln.

4.1 Zahlen

- In Perl können Sie Zahlen (Literale) sowohl als Ganzzahlen (integer) als auch als Gleitpunktzahlen (floating point) darstellen.
- In Perl existieren Zahlen intern als doppelt genaue Gleitpunktzahlen (entspricht dem double bei C/C++). Mit dem Pragma (Hilfsmodul) **use integer** erzeugen Sie Integer-Zahlen.
- Zahlen können auch als String eingegeben werden. Diese werden solange als String behandelt, bis die erste mathematische Operation ansteht. Es erfolgt dann eine Umwandlung in eine Zahl, um die mathematische Operation auszuführen.
- Mit speziellen Operatoren können Sie eine Bitmanipulation vornehmen, um einzelne Bits zu manipulieren oder entsprechende Bit-Masken zu setzen.
- Durch Einsatz von Standardmodulen sind Sie in der Lage, mit Zahlen beliebiger Größe und Genauigkeit sowie mit komplexen Zahlen zu rechnen.
- Um große Zahlen übersichtlicher zu schreiben, bietet Ihnen Perl die Besonderheit an, beliebige Unterstriche in Zahlen einzufügen.
- Zahlen werden als Oktal- oder als Hexadezimalzahlen betrachtet, wenn sie mit einer `0` bzw. `0x` beginnen. Mit den Funktionen `oct()` und `hex()` können Oktal- bzw. Hexadezimalzahlen als äquivalente Dezimalzahlen dargestellt werden.

4.1.1 Ganze Zahlen

Zahlen werden als Folge von Ziffern mit eventuellen Vorzeichen dargestellt:

```
123, -34
```

Zur übersichtlichen Darstellung lassen sich Zahlen auch wie folgt darstellen:

```
56_789
```

Ebenso können Sie Zahlen aus einem anderen Zahlensystem darstellen:

```
024                                    # Oktalzahl
0x14                                   # Hexadezimalzahl
```

Um die Konvertierung einer Zahl aus einem Quellsystem (mit der Basis B) in ein Zielsystem zu realisieren, bietet sich für die Berechnung das *Hornerschema* an, denn in dieser Darstellung treten keine Potenzen auf.

Darstellung ganzer Zahlen nach dem Hornerschema
Das Hornerschema für ganze Zahlen Z_B lautet:

$$Z_B = \pm((Z_{n-1} \times B + Z_{n-2})\ B + \ldots + Z_1)\ B + Z_0$$

Konvertierung einer Oktalzahl in eine Dezimalzahl

$$234_8 = (2 \times 8 + 3)8 + 4 = 2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0$$
$$= 156_{10}$$

Konvertierung einer Hexadezimalzahl in eine Dezimalzahl

$$234_{16} = (2 \times 16 + 3)16 + 4 = 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0$$
$$= 564_{10}$$

Zahlensysteme und deren Konvertierung
Perl besitzt zwei Funktionen, mit denen Sie Zahlen aus dem Oktal- bzw. Hexadezimal-Zahlensystem in das Dezimal-Zahlensystem umwandeln können. Die Funktionen `oct()` und `hex()` übernehmen Strings und liefern Dezimalzahlen zurück. Hierbei sollten Sie beachten, dass die Funktion `oct()` nicht überprüft, ob der String wirklich in Oktal-Darstellung vorliegt. Sie müssen überprüfen, ob der String tatsächlich mit einer `0` beginnt.

Oktaldarstellung
Im Oktalsystem wird 8 als Basis verwendet. Mit acht verschiedenen Ziffern (0, 1, 2, 3, 4, 5, 6 und 7) werden die Zahlen dargestellt. Wie in C/C++ werden diese mit einer führenden Null gekennzeichnet. Hierbei sollten Sie berücksichtigen, dass eine führende `0` unterschiedliche Bedeutung hat. Im Programm selbst wird die Zahl als oktale Konvertierung aufgefasst. Als

Eingabewert erfolgt keine oktale Konvertierung, so dass der Eingabewert als Dezimalzahl gewertet wird:

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

print 024;                          # 20  Dezimalzahl
print -077;                         # -63 Dezimalzahl
```

Im folgenden Beispiel erfolgt die Eingabe 024, also Dezimalzahl:

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

$ein = <STDIN>;                     # Eingabe z. B. 024
print $ein;                         # 24 Dezimalzahl
```

Die Funktion `oct()`
`oct` *oktalzahl*

Die Funktion `oct()` gibt den oktalen Zahlenwert als Dezimalwert zurück. Beginnt das Argument *oktalzahl* mit `0x`, erfolgt eine Konvertierung der hexadezimalen Zahlendarstellung in eine dezimale Zahlendarstellung. Beginnt das Argument *oktalzahl* mit `0b`, wird die duale Zahlendarstellung ebenfalls in eine äquivalente dezimale Zahl konvertiert. Sie können nur ganze Zahlen größer oder gleich 0 konvertieren. Wenn Sie die Funktion `oct()` zur Konvertierung einsetzen, müssen Sie *oktalzahl* als String angeben. Die Funktion `oct()` kann aus dem Kontext schließen, ob es sich im String um eine Oktal-, Hexadezimal- oder Binärzahl handelt:

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

print oct "024";                    # Oktalzahl
# 20 Dezimalzahl
print oct "0x14";                   # Hexadezimalzahl
# 20 Dezimalzahl
print oct "0b10100";                # Binärzahl
# 20 Dezimalzahl
```

Im obigen Beispiel konvertiert jeweils der Oktal-, Hexadezimal- und der Binärwert in einen Dezimalwert.

Hexadezimaldarstellung
Im Hexadezimalsystem wird zur Zahlendarstellung die Basis 16 gewählt, so dass 16 verschiedene Ziffern zum Aufbau von Hexadezimalzahlen benötigt werden; neben den Ziffern 0 bis 9 sind dies üblicherweise die ersten sechs Buchstaben des Alphabeths a .. f oder A .. F. Hierbei entsprechen a .. f oder A .. F den Ziffern 10 .. 15. Hexadezimalzahlen sind in Perl durch vorangestelltes `0x` gekennzeichnet:

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

print 0x14;                          # 20  Dezimalzahl
print -0x3f;                         # -63 Dezimalzahl
print 0xab;                          # 171 Dezimalzahl
```

Die Funktion `hex()`
`hex` *hexzahl*

Die Funktion `hex()` gibt den Dezimalwert einer Hexadezimalzahl an. Die Hexadezimalzahl muss dabei als String angegeben werden. Tun Sie dies nicht, wird zunächst intern die Funktion `oct()` ausgeführt. Dieser konvertierte Zahlenwert wird anschließend intern mit der Funktion `hex()` vom Hexadezimal- ins Dezimal-System überführt. Sie können auch hier nur Zahlen größer oder gleich 0 konvertieren:

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

print hex "0x14";                    # 20 Dezimalwert
print hex 0x14;                      # 32 Dezimalwert
```

Es folgt das Beispiel als Ablauf ohne String-Angabe:

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

print hex 0x14;                      # 32 Dezimalzahl

# erste Konvertierung:
print oct "0x14";                    # 20 Dezimalzahl

# zweite Konvertierung:
print hex "20";                      # 32 Dezimalzahl
```

4.1.2 Gleitpunktzahlen

Wie in anderen Programmiersprachen müssen Sie einen Dezimalpunkt setzen, wenn Sie Stellen hinter dem Komma verwenden wollen. Ein Komma in der Zahlendarstellung ist nicht erlaubt. Perl bricht dort die Zahlendarstellung ab:

```
1.23, -3.4, 5.678_9, 0.321, .543, 7., 7.0
```

Sie können auch die wissenschaftliche Notation von Zahlen ausführen. Der dabei verwendete Exponent `E` kann sowohl klein als auch groß vor der restlichen Zahl geschrieben werden.

E bzw. e steht für Exponent (10 hoch ...). Die obligatorische Null vor dem Dezimalpunkt kann auch hier entfallen, der Dezimalpunkt natürlich nicht:

```
.4395E2, -12.345e2, 5.2e-2
```

Selbstverständlich können Sie – wie im obigen Beispiel dargestellt – auch negative Exponenten verwenden.

4.2 Operatoren

In Perl existieren eine Reihe von Operatoren, zum Beispiel Operatoren für die Grundrechnungsarten, für Tests und Vergleiche, für logische Verknüpfungen oder der häufigste genutzte Operator für Zuweisungen, um nur einige zu nennen.

4.2.1 Zuweisungs-Operator

Mit dem Zuweisungs-Operator, der in Perl aus einem Gleichheitszeichen (in Anlehnung an C/C++) besteht, weisen Sie ganz simpel einer Variablen auf der linken Seite einen Wert der rechten Seite zu. Dabei können auf der rechten Seite verschiedene Arten von Zuweisungen stehen, wie z. B. Aneinanderreihung von Werten, Werte anderer Variablen, Werte von Ausdrücken, Kontextbestimmung, Ergebnisse von Funktionsaufrufen usw.:

```
$a = $b = 4711;
$neu = $a;
@liste = 1..5;
$anzahl = @liste;
$wert = $liste[1];
$last = pop (@liste);
```

Der Ausdruck, der auf der linken Seite des Zuweisungs-Operators steht, wird lvalue (l für left) genannt und gibt den Speicherplatz an. Alles, was auf der rechten Seite des Gleichheitszeichens steht, wird rvalue (r für right) genannt und ausgewertet; der Wert wird dann dem Speicherplatz zugewiesen.

Die Aneinanderreihung von Werten können Sie als einen Ausdruck mit einem Wert interpretieren: Beim Ausdruck `$b = 4711` (siehe erste Programmzeile) erhält `$b` den Wert 4711, dieser Ausdruck, nämlich Wert 4711, wird dann der Variablen `$a` zugewiesen. Somit erhalten `$a`, `$b` allesamt den Wert 4711. Mit Ausnahme von JavaScript beherrscht fast jede Sprache diese Technik.

Außerdem bestimmt die linke Seite den Kontext für die rechte Seite. Im obigen Beispiel zwingen Sie durch eine skalare Variable `$anzahl` die Liste dazu, dass sie im skalaren Kontext ausgewertet wird und einen skalaren Wert zurückliefert.

Ferner steht für die meisten Operatoren eine abkürzende Schreibweise zur Verfügung:

```
$var = $var + 4711;
```

können Sie kürzer wie folgt schreiben:

```
$var += 4711;
```

Es lassen sich folgende abkürzende Operatoren verwenden. Zwischen Operator und Gleichheitszeichen darf kein Leerzeichen sein.

Tabelle 4.2.1: Zuweisungs-Operator

Arithmetisch	+=	-=	*=	/=	%=	**=	<<=	>>=
Strings	.=							
Logisch	||=	&&=	^=	|=	&=			

4.2.2 Inkrementieren und Dekrementieren

Der In-/Dekrement-Operator erhöht/verringert den Wert des Operanden um 1. Steht der Operator vor der Variablen, so wird zuerst die Inkrementierung/Dekrementierung und anschließend die Auswertung der Variablen durchgeführt; umgekehrt ist es, wenn der Operator hinter der Variablen steht:

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

$var = 3.14;
$praefix = ++$var;
print "\$praefix = $praefix\n";          # $praefix = 4.14
$postfix = $var++;
print "\$postfix = $postfix\n";          # $postfix = 4.14
print "\$var = $var\n";                  # $var = 5.14
```

Wie Sie am obigen Programm sehen können, lassen sich die Operatoren ++ und -- sogar auf Gleitpunktzahlen anwenden.

4.2.3 Potenzieren

Perl verwendet den **-Operator (die Funktion ist mit der C/C++-Funktion pow() realisiert), zum Potenzieren. Hierbei wird die linke Zahl zur Potenz der rechten Zahl erhoben. Wie das Beispiel unten zeigt, ist der **-Operator rechts-assoziativ. So ist -5**2 gleich -(5**2) und nicht gleich (-5)**2. Wie die meisten gängigen Programmiersprachen erlaubt auch Perl das Rechnen mit gebrochenen Exponenten:

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

$var = -5**2; # -(5**2)        print "$var\n";   # -25
$var = (-5)**2;                print "$var\n";   # 25
```

4.2.4 Arithmetischer Operator

Für die Grundrechnungsarten stehen in Perl die in der Tabelle 4.2.4: *Arithmetische Operationen* aufgeführten Operatoren zur Verfügung.

Da Perl – wie bereits erwähnt – intern ganze Zahlen und Gleitpunktzahlen nicht unterscheidet, müssen Sie, wenn Sie eine Integerdivision durchführen möchten, die Funktion `int()` verwenden.

Tabelle 4.2.4: Arithmetische Operationen

Operatoren	Bedeutung
`++, --`	Inkrement, Dekrement
`**`	Potenz
`!, ~, \, +, -`	Unäre Operatoren
`*, /, %, x`	Multiplikation, Division, Restbildung (Modulo), Vervielfachung
`+, -, .`	Addition, Subtraktion, Konkatenation

Die Operatoren werden nach Assoziativität und Priorität abgearbeitet. Die Assoziativität gibt an, in welcher Richtung (z. B. von links nach rechts oder umgekehrt) Operatoren und Operanden zusammengefasst werden. Die Priorität gibt die Rangfolge der Ausführung an. So haben Multiplikation, Division und Restbildung Vorrang vor Addition, Subtraktion und Konkatenation. Operatoren, die auf gleicher Ebene stehen, werden nach den Regeln der Assoziativität behandelt.

4.2.5 Vergleichs-Operatoren

Dieser Abschnitt widmet sich den Wahrheitswerten von Vergleichen. Es gibt Vergleiche zwischen numerischen Werten und Vergleiche zwischen Strings. Das Ergebnis dieser Vergleiche durch einen entsprechenden Operator ergibt den Wahrheitswert: entweder `true` oder `false` (siehe Tabelle 4.2.5: *Vergleichs-Operatoren*).

Bei numerischen Größen entscheidet beim Vergleich die Reihenfolge. Bei Vergleichen von Strings ist die Reihenfolge im ASCII-Code maßgebend.

Tabelle 4.2.5: Vergleichs-Operatoren

Numerisch	String	Ergebnis
`$x == $y`	`$x eq $y`	True, wenn $x gleich $y
`$x != $y`	`$x ne $y`	True, wenn $x ungleich $y
`$x > $y`	`$x gt $y`	True, wenn $x größer $y
`$x < $y`	`$x lt $y`	True, wenn $x kleiner $y
`$x >= $y`	`$x ge $y`	True, wenn $x größer oder gleich $y
`$x <= $y`	`$x le $y`	True, wenn $x kleiner oder gleich $y
`$x <=> $y`	`$x cmp $y`	-1, 0, 1, wenn $x kleiner, gleich, größer $y

Wie in der obigen Tabelle angezeigt, liefert der Operator `<=>` beim Vergleich den Wahrheitswert `false`, wenn beide Argumente gleich sind:

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

print 4711 <=> 4711;          # 0
print 4711 <=> 4712;          # -1
print 4711 <=> 4710;          # 1
```

4.2.6 Logische Operatoren

Logische Operatoren dienen dem Verknüpfen Boolescher Ausdrücke. Sie werten ihre Argumente im Booleschen Kontext aus und liefern als Ergebnis den Wahrheitswert `true` oder `false` zurück. Sie können zwei Formen logischer Operatoren nutzen, die C-Operatoren (`!`, `^`, `&&`, `||`) und die Perl-Operatoren (`not`, `xor`, `and`, `or`). Wie in der unten aufgeführten Tabelle ersichtlich, besitzen die C-Operatoren eine höhere Rangfolge (Präzedenz) und werden in Ausdrücken vorrangiger behandelt als die Perl-Operatoren. Dies sollten Sie berücksichtigen, wenn Sie die logische **UND**-Verknüpfung `and` oder `&&` bzw. die logische **ODER**-Verknüpfung `or` oder `||` verwenden wollen.

Diesem Umstand der Rangfolge (wie das folgende Beispiel zeigt) müssen Sie, sollten Sie keine Klammern verwenden, Rechnung tragen, wenn Sie eine Zuweisung einer logischen Verknüpfung vornehmen möchten:

```
$erg = $u || $v;          # $erg = ($u || $v);
$erg = $u or $v;          # Fehler: ($erg = $u) or $v;
```

Im obigen Beispiel bindet im ersten Fall der ||-Operator stärker als der Zuweisungs-Operator `=`. Im zweiten Fall bindet der Zuweisungs-Operator `=` stärker als der `or`-Operator. Diesen Tatbestand sollten Sie berücksichtigen und somit `and`, `or` für Bedingungen und `&&`, `||` für Berechnungen verwenden.

Tabelle 4.2.6: Logische Verknüpfungsoperatoren

Hohe Rangfolge	Niedrige Rangfolge	Bedeutung
!	not	Logische Negation
^	xor	Exklusives ODER
&&	and	Logisches UND
||	or	Logisches ODER

Short-cut-evaluation
Logische Ausdrücke werden abkürzend ausgewertet (Short-cut-evaluation), d. h. die Vergleichs-Operatoren &&, | |, and und or werden nur solange ausgewertet, bis der Wahrheitswert des gesamten Ausdrucks feststeht. Ein logischer **UND**-Ausdruck braucht nicht weiter ausgewertet werden, wenn der erste Teilausdruck `false` ergibt. Mit anderen Worten: Ein gesamter logischer **UND**-Ausdruck wird genau dann `false`, wenn bei einer Auswertung von links nach rechts ein Teilausdruck `false` ergibt. Ein logischer **ODER**-Audruck braucht nicht weiter ausgewertet werden, wenn der erste Teilausdruck `true` ergibt. Anders ausgedrückt: Ein gesamter logischer **ODER**-Ausdruck wird genau dann `true`, wenn bei einer Auswertung von links nach rechts ein Teilausdruck `true` ergibt.

4.2.7 Bit-Operatoren

Wie in C/C++ existieren auch in Perl bitlogische Operatoren. Mit **UND**, **ODER**, exklusivem **ODER** und **NICHT** (Komplement) können Sie mit jedem Bit logische Operationen des ganzzahligen Operanden ausführen. Sie können damit gezielt einzelne Bits verändern bzw. Bit-Masken erstellen, um Bit-Werte zu manipulieren bzw. diese zu testen.

Mit den Schiebeoperatoren << und >> verschieben Sie alle Bits des linken Operanden um so viele Stellen nach links oder nach rechts, wie es der rechte Operand angibt. Bevor einige Beispiele hierzu dargestellt werden, sollen zunächst in der folgenden Tabelle die Bit-Operatoren und deren Bedeutung aufgelistet werden.

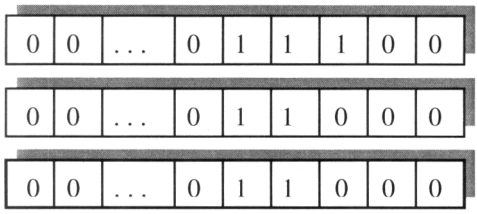
Tabelle 4.2.7: Bitoperationen

Operator	Bedeutung
&	Bitweises UND
|	Bitweises ODER
^	Bitweises XOR
~	Bitweises Komplement
<<	Bitweise Links-Verschiebung
>>	Bitweise Rechts-Verschiebung

Durch obige Operatoren haben Sie die Möglichkeit, Bit-Masken zu setzen:

Bits gezielt auf 0 setzen
Mit dem &-Operator lassen sich Bits gezielt auf 0 setzen. In den folgenden Beispielen soll eine willkürliche Hexadezimalzahl `0x1C` wie folgt maskiert werden:

Im ersten Beispiel werden die letzten 3 Bits der Hexadezimalzahl auf 0 gesetzt:



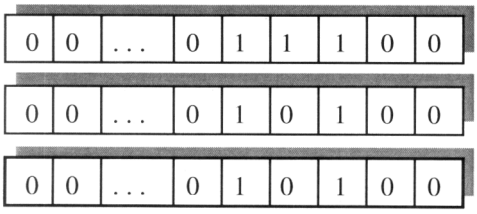
0x1C = 28 Dezimalwert

0x18 = 24 Dezimalwert

0x1C & 0x18

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

$bit = 0x1C;                    # 0x1C = 1 1100
$bit = $bit & 0x18;             # 1 1000 = 1 1100 & 1 1000
printf "%X\n",$bit;             # 18, Hexadezimalwert
```

Im zweiten Beispiel wird bei der Hexadezimalzahl `0x1C` das vierte Bit auf 0 gesetzt und anschließend wird überprüft, ob das dritte Bit mit einer 1 besetzt ist:



0x1C = 28 Dezimalwert

0x14 = 20 Dezimalwert

0x1C & 0x14

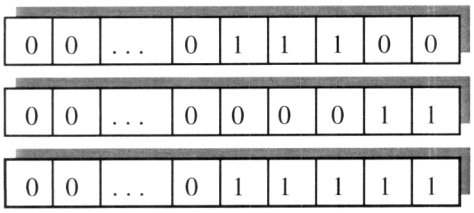
```
#!/usr/bin/perl -w

$bit = 0x1C;                    # 0x1C = 1 1100
$bit = $bit & 0x14;             # 1 0100 = 1 1100 & 1 0100
printf "%X\n",$bit;             # 14, Hexadezimalwert
print "3. Bit 1 gesetzt " if $bit & 0x4; # 3. Bit 1 gesetzt
```

Mit Hilfe der Funktion `printf()` geben Sie mit einem Formatstring an, wie die Ausgabe zu formatieren ist. Der Formatstring kann aus den folgenden drei Komponenten bestehen:

- **Literaler Text** erscheint genau so in der Ausgabe, wie Sie ihn im Formatstring festlegen.
- **Escape-Sequenzen** bieten besondere Möglichkeiten (z. B. New-Line `\n`) zur Formatierung.

- **Konvertierungsspezifizierer** bestehen aus einem Prozentzeichen `%`, gefolgt von einem weiteren Zeichen. Im obigen Beispiel lautet der Konvertierungsspezifizierer `%X` für Hexadezimalzahlen.

Bits gezielt auf 1 setzen
Mit dem |-Operator lassen sich Bits gezielt auf 1 setzen. Im nächsten Beispiel sollen bei der willkürlich gewählten Hexadezimalzahl `0x1C` die letzten zwei Bits auf 1 gesetzt werden:



0x1C = 28 Dezimalwert

0x3 = 3 Dezimalwert

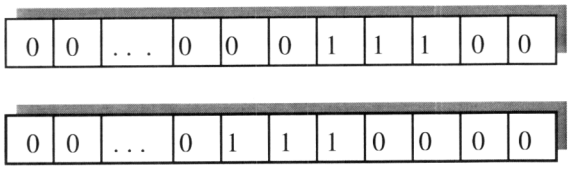
0x1C | 0x3

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

$bit = 0x1C;                    # 0x1C = 1 1100
$bit = $bit | 0x3;              # 1 1111 = 1 1100 | 0 0011
printf "%X\n",$bit;             # 1F, Hexadezimalwert
```

Perl bietet die Möglichkeit, Zahlen binär sowohl nach links << als auch nach rechts >> zu verschieben. Dies wirkt angesichts der Tatsache, dass Zahlen als Gleitpunktzahlen gespeichert und als solche behandelt werden, etwas ungewöhnlich. Perl behandelt aber Zahlen wie Integer, wenn eine der obigen Bit-Operatoren angewendet wird. Mit dem Pragma **use integer** erfolgt die Darstellung als signed integer; mit **no integer** erfolgt eine unsigned integer Darstellung.

Links-Shift
Das Shiften nach links entspricht der Multiplikation mit dem Faktor 2**Shift.

Auch in den folgenden Beispielen soll wieder von der willkürlichen Hexadezimahl `0x1C` ausgegangen werden:



0x1C = 28 Dezimalwert

0x1C << 2

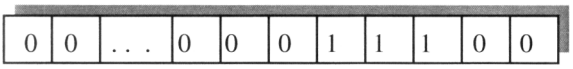
```
#!/usr/bin/perl -w

$bit = 0x1C;                    # 0x1C = 1 1100
$bitli = $bit  <<  2;           # 0111 0000 = 1 1100 << 2
```

```
printf "%X\n",$bitli;          # 70, Hexadezimalwert
```

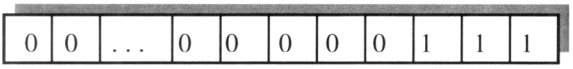
Beim Links-Shift werden entsprechende Nullen nachgezogen. Die Hexadezimalzahl 70 (Dezimalwert 112) ist das Ergebnis der Multiplikation von `1C` (Dezimalwert 28) mit 2**2.

Rechts-Shift
Das Shiften nach rechts entspricht einer Division durch 2**Shift.

Hier soll ebenfalls von der Zahl 0x1C ausgegangen werden:



0x1C = 28 Dezimalwert

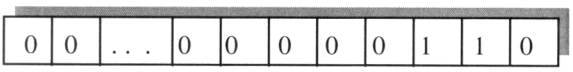


0x1C >> 2

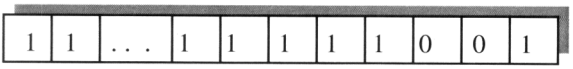
```
#!/usr/bin/perl -w

$bit = 0x1C;                   # 0x1C = 1 1100
$bitre = $bit  >>  2;          # 0 0111 = 1 1100 >> 2
printf "%X\n",$bitre;          # 7, Hexadezimalwert
```

Auch hier entspricht das Ergebnis 7 (beim Rechts-Shift) der Division von `1C` dividiert durch 2**2.

Bitweises Negieren (Einerkomplement)
In Perl bewirkt das monadische Tildezeichen ~ eine Bitweise Negation (B-1-Komplement):



0x6 = 6 Dezimalwert



~ 6

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

printf "%X   %d", ~0x6, ~6;          # FFFFFFF9   -7 Dezimal
```

Im obigen Beispiel ergibt die bitweise Negation von `0x6` als Ergebnis `FFFFFFF9`. Dieses entspricht als Dezimalzahl im B-Komplement dem Wert –7. Bekanntlich wird beim B-1-Komplement eine 0 zu 1 und umgekehrt gebildet. Das B-Komplement erzielen Sie durch eine Addition einer 1 zum (B-1)-Komplement.

4.3 Mathematische Funktionen

Perl bietet in seinem Funktionsumfang auch eine Reihe von wichtigen mathematischen Funktionen. Ferner stehen Ihnen im POSIX-Modul weitere mathematische Funktionen zur Verfügung.

4.3.1 Integrierte mathematische Standardfunktionen

In der nachfolgenden Tabelle erhalten Sie eine Übersicht über die integrierten mathematischen Standardfunktionen von Perl.

Tabelle 4.3.1.: Mathematische Standardfunktionen

Name	Bedeutung
`abs` *wert*	Absolutbetrag
`atan2` *y, x*	Arcustangens y/x (-pi bis +pi)
`cos` *zahl*	Cosinus (Bogenmaß)
`exp` *zahl*	Exponentialfunktion
`hex` *hexzahl*	Umwandlung Hexa- in Dezimalzahl
`int` *zahl*	Ganzzahl
`log` *zahl*	Natürlicher Logarithmus (Basis e)
`oct` *octzahl*	Umwandlung Octal- in Dezimalzahl
`rand` *zahl*	Zufallswert
`sin` *zahl*	Sinus (Bogenmaß)
`sqrt` *zahl*	Quadratzahl
`srand` *ausdruck*	Zufallsgenerator von rand

Mit Ausnahme der Funktion `atan2()` können die übrigen Funktionen auch ohne explizites Argument geschrieben werden; sie werden dann auf den jeweiligen Wert der Spezial-Variablen `$_` angewandt.

Es sollen nun einige Beispiele folgen, wo mathematische Funktionen verwendet worden sind:

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

$_ = -4711;
print abs,"\n";                          # 4711
print abs 4711,"\n";                     # 4711
print atan2(0.5, 0.7),"\n";              # 0.62024...
print exp 1,"\n";                        # 2.71828...
print int 1.9999,"\n";                   # 1
print log 2.71828,"\n";                  # 0.99999...
```

```
print rand 4,"\n";                    # 1.54895...
print sqrt 4711,"\n";                 # 68.63672...
```

4.3.2 Zufallszahlen

Die Perl-Funktion `rand()` erzeugt Pseudozufallszahlen nach einem einfachen Algorithmus. Ohne Argument liefert sie die Werte zwischen 0 und 1, ansonsten (Gleitpunkt-)Zahlen zwischen 0 und dem Argument. Die Funktion `srand()` initialisiert automatisch den Zufallsgenerator, falls er nicht vorher schon definiert wurde. Möchten Sie eine ganze Zahl erhalten, brauchen Sie nur eine Umwandlung mit der Funktion `int()` vornehmen.

Im folgenden Beispiel sollen sechs Integerzahlen ausgegeben werden. Doppelte Integerzahlen werden nicht berücksichtigt:

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

for ($i = 0; $i < 6; $i++) {
      $wert[$i] = int (rand 49) + 1;
}
print "Sechs Integer-Zahlen: @wert\n";
```

4.4 Strings

Strings zählen in Perl zu den Grundtypen und bestehen aus Sequenzen aufeinanderfolgender Zeichen. Im Gegensatz zu C/C++ oder anderen Sprachen sind diese bei Perl nicht als Array von Zeichen konzipiert. Diese Eigenschaft vereinfacht den Umgang und die Manipulation von String-Text erheblich. Den Zugriff auf einzelne Zeichen oder Bereiche eines Strings realisieren Sie mit einer Vielzahl von Funktionen.

Im Normalfall können die Zeichen eines Strings alle Werte eines 8-Bit-Zeichens (Byte 0-255) und somit jede Art von ASCII-Daten annehmen. Der Unicode (16-Bit-Zeichen) wird ab Version 5.006 unterstützt.

Typische Strings sind druckbare Sequenzen von Buchstaben, Ziffern usw. Ferner können Strings Escape-Zeichen (Fluchtsymbole) enthalten. Sie können Operatoren für Tests und Vergleiche von Strings sowie für Variableninterpolation, Groß- und Kleinschreibung verwenden.

Die Länge eines Strings ist beliebig und nur durch den zur Verfügung stehenden Speicher begrenzt. Perl kennt zwei grundlegende Arten von Strings. Sie können String-Literale in einfachen Hochkommas (Apostroph, single-quotes) oder in Anführungszeichen (doppeltes Hochkomma, double-quotes) darstellen. Weiterhin existieren noch Strings, die in Gravus oder Backticks eingeschlossen sind. Das entsprechende Präfix `qx` dient zur Ausführung von Betriebssystemkommandos. Ferner existieren noch Strings, die als HERE-Dokument (aus

der Shell-Programmierung bekannt) geschrieben und von zwei identischen, frei wählbaren Tokens eingeschlossen werden.

Zusammenfassend lässt sich Folgendes sagen:

- Strings sind keine Zeichenarrays,
- Strings haben eine unbeschränkte Länge,
- Strings wachsen und verkleinern sich je nach Bedarf automatisch,
- Strings können beliebige Binärdaten enthalten,
- Strings können durch Funktionen einfach manipuliert werden,
- Strings können durch Operatoren miteinander verglichen werden.

4.4.1 Quoting-Regeln

Eine Perl-Variable erhält ihren Typ durch die Zuweisung eines Wertes, dessen Typ sie annimmt.

Bei der folgenden Zuweisung eines Integerwertes wird dieser zum Typ Integer:

```
$var = 4711;
```

Die gleiche Variable können Sie bereits in der nächsten Zeile Ihres Programms zum Typ String machen:

```
$var = "Perl ist spitze";
```

Diese Eigenschaft wird als schwache Typisierung bezeichnet.

Um Strings von anderen Werten zu unterscheiden, werden Strings in Anführungszeichen (quotes) eingeschlossen.

Da Sie in Perl beide Arten, nämlich sowohl Hochkommas (single-quotes) als auch Anführungszeichen (double-quotes), verwenden können, führt dieses zu folgenden Konsequenzen:

Hochkomma
Beim Hochkomma handelt es sich um die "wörtliche" Form. Ein String in dieser Form wird genauso interpretiert, wie Sie ihn angegeben haben. Enthaltene Dollarzeichen mit anschließendem Namen (skalare Variable) werden nicht interpoliert, sondern genauso an Ort und Stelle ausgegeben, wie sie angegeben wurden:

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

$pi = 3.14;
print 'Der Wert von $pi ist: ',$pi,'\n';
# Der Wert von $pi ist: 3.14\n
```

Der Text zwischen den Hochkommas wird wörtlich genommen. Escape-Sequenzen *mit* einfachen Hochkommas bleiben ohne Wirkung.