Grundwissen Perl

von
Prof. Dr. Jürgen Schröter

Oldenbourg Verlag München Wien

Prof. Dr. Jürgen Schröter studierte zunächst Elektrotechnik an der Ingenieurschule in Wuppertal bevor er das Studium der Informatik an der TU Berlin aufnahm. Nach seiner Promotion an der Gesamthochschule Wuppertal erhielt er einen Lehrauftrag für Digitaltechnik an der FH Darmstadt. Seit 1985 ist er Professor an der Fachhochschule Landshut mit den Lehrgebieten Programmieren, DV-Systeme und Digitaltechnik. Er ist zudem Leiter des Rechenzentrums und des Labors Digitaltechnik.

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: Dr. Margit Roth
Herstellung: Anna Grosser
Coverentwurf: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Druckhaus „Thomas Müntzer“ GmbH, Bad Langensalza

ISBN 978-3-486-58074-7

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 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) eingesetzt, 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, sind der kostenlose Bezug und die außergewöhnliche Hilfsbereitschaft der großen Perl-Fan-Gemeinde.

Nach dem erfolgreichen Erscheinen der ersten Auflage von Perl habe ich mich für ein erweitertes Buch entschieden. In ihm werden neue Kapitel wie Objektorientierte Programmierung, CGI, Grafische Oberfächen sowie im Anhang Perl 6 behandelt.

Dank

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

Von einer Reihe aufmerksamer Leser erhielt ich Zuschriften mit nützlichen Verbesserungsvorschlägen. Ich danke all meinen Kollegen, die mich ermuntert haben, dieses neue Buch zu schreiben.

Mein aufrichtiger Dank gilt Margit Roth, die die Lektoratsleitung Buch MINT beim Oldenbourg Wissenschaftsverlag innehat. Sie hat dieses Projekt intensiv und engagiert betreut sowie stets ein offenes Ohr für Wünsche und Änderungen gehabt.

Last but not least danke ich meiner Frau Heidemarie Schröter, die mein Vorhaben immer unterstützt hat, obwohl viele private Aktivitäten - besonders Bridgeabende - 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 Warum Perl?

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

Larry Wall ist es gelungen, eine extrem ausdrucksstarke Sprache zu entwickeln. Perl ähnelt 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. Aufgrund 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.

Des Weiteren profitiert Perl von den vielfältigen Einsatzgebieten, besonders im Internet und im Web. Client-Server-Interaktionen, z. B. SQL-Anweisungen, die an einen Datenbankserver gesendet werden, und die vom Server zurücklaufenden Ergebnisse, werden in Perl realisiert. Auch Textprotokolle wie CGI (**C**ommon **G**ateway **I**nterface) können mit Perl durchgeführt werden. Perl führt 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. Die Automation der Systemadministration und die Datensicherung gehören ebenfalls 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 ist Perl eigentlich zum Beispiel 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.

Möchten Sie Perl-Programme auf einem Unix-System laufen lassen, brauchen Sie nur `#!/usr/bin/perl -w` an die erste Zeile zu 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 Interpreter `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.

2 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. Außerdem erfahren Sie alles über das Thema Kontext und den Begriff Wahrheitswert.

2.1 Variablen

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

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

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

# 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.

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 das 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.

Zustand und Gültigkeitsbereiche von Variablen

In diesem Abschnitt sollen einige Funktionen genannt werden, die Ihnen den Zustand bzw. den Gültigkeitsbereich einer Variable verdeutlichen werden.

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.

2.2 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. Dabei handelt es sich sowohl um skalare Variablen als auch um Arrays und Hash-Variablen. 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. Im Anhang werden die wichtigsten Bereiche genannt, in denen Spezial-Variablen aufgeführt sind. Umfangreiche Informationen hierzu finden Sie unter PERLVAR-Dokumentation.

2.3 Kontext

In den natürlichen Sprachen wird der Begriff Kontext als umgebender Text bzw. als Inhalt eines Schriftstückes gedeutet.

In Perl definieren Operatoren und Funktionen einen bestimmten Kontext. Variablen sind kontext-sensitiv, da sie Werte mit verschiedenen Kontexten liefern, die auch unterschiedlich interpretiert werden. Von einer Variable 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
```

Ebenso kann die Auswertung eines Ausdrucks im Listenkontext oder in einem skalaren Kontext erfolgen. Im Listenkontext wird eine Liste und im skalaren Kontext ein einzelner Wert erwartet:

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

@array = A .. D;
print @array,"\n";                   # Listenkontext A B C D
$anzahl = @array;
print $anzahl."\n";                  # 4 skalarer Kontext
```

Im Listenkontext werden im obigen Beispiel die einzelnen Argumente der Liste vom Array `"@array"` ausgegeben. Die gleiche Ausgabe ließe sich auch in der Form `print "A", "B", "C", "D", "\n";` ausführen. Die Zuweisung eines Arrays an eine skalare Variable bewirkt einen skalaren Kontext, d. h. einen Wert. Die Variable `$anzahl` besitzt somit eine skalare Größe, nämlich den Wert `4`. Dieser Wert bedeutet: „Anzahl der Listen-Elemente", der anschließend durch die `print`-Anweisung ausgegeben wird.

Ein häufiger Fehler ensteht, wenn Sie eine Konkatenation statt mit Strings mit einem Array vornehmen:

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

@array = A .. D;
print @array,"\n";                   # Listenkontext A B C D
print @array."\n";                   # skalarer Kontext 4
```

In der ersten `print`-Anweisung werden die Argumente der Liste `"A"` bis `"D"` mit anschließendem New-Line `"\n"` im Listenkontext ausgegeben. In der zweiten `print`-Anweisung werden durch den Konkatenations-Operator `"."` nicht die einzelnen Elemente, sondern die Anzahl der Elemente, nämlich `4`, ausgegeben. Dies liegt darin begründet, dass durch die höhere Rangfolge der Konkatenation gegenüber dem Funktionsaufruf ein skalarer Kontext erzwungen wird.

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 über-

sichtlicheren 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";         # 1_158_957_250
print scalar localtime,"\n";  # Fri Sept 22 22:34:10 2006
```

In der folgenden Tabelle sind die Kontexte aufgeführt:

Tabelle 2.3: Kontexte

Kontext	Bedeutung
Skalarer Kontext	Es wird ein einziger skalarer 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

2.4 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.

3 Skalare

Perl-Variablen können in Form von Skalaren auftreten, die entweder numerische Daten oder Strings (Zeichenketten) oder auch beides enthalten. Einer skalaren Variable 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.

3.1 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 so lange 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.
- Zur übersichtlicheren Schreibweise von großen Zahlen bietet Ihnen Perl die Besonderheit an, 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.

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
```

Zur Konvertierung einer Zahl aus einem Quellsystem (mit der Basis B) in ein Zielsystem 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. Ist dies nicht der Fall, 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
```

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.

3.2 Operatoren

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

Zuweisungs-Operator

Mit dem Zuweisungs-Operator, der in Perl aus einem Gleichheitszeichen (in Anlehnung an C/C++) besteht, weisen Sie ganz simpel einer Variable 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 Variable `$a` zugewiesen. Somit erhalten `$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 stehen.

Tabelle 3.2.1: Zuweisungs-Operator

Zuweisung	Operator							
Arithmetisch	`+=`	`-=`	`*=`	`/=`	`%=`	`**=`	`<<=`	`>>=`
String	`.=`							
Logisch	`||=`	`&&=`	`^=`	`|=`	`&=`			

Inkrementieren und Dekrementieren

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

Potenzieren

Perl verwendet den `**`-Operator 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:

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

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

Arithmetischer Operator

Für die Grundrechnungsarten stehen in Perl die in der Tabelle 3.2.4: *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 3.2.4: 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.

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 3.2.5: *Vergleichs-Operatoren*).

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

Tabelle 3.2.5: 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
```

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 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 3.2.6: 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 so lange 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.

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 so um 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.

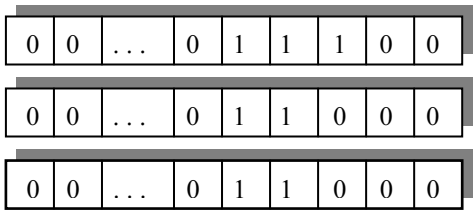
Tabelle 3.2.7: Bit-Operationen

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:



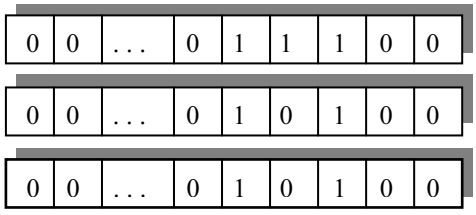
0x1C = 28 Dezimalwert

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

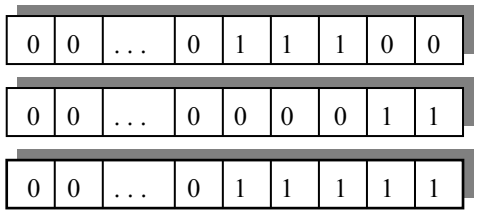
0x1C & 0x14

```
#!/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

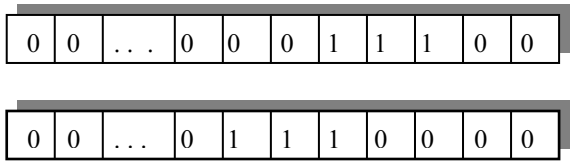
0x1C | 0x3

```
#!/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 Shift-Faktor 2**.

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



0x1C = 28 Dezimalwert

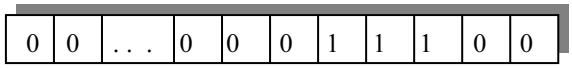
0x1C << 2

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

$bit = 0x1C;                    # 0x1C = 1 1100
$bitli = $bit  <<  2;           # 0111 0000 = 1 1100 << 2
printf "%X\n",$bitli;           # 70, Hexadezimalwert
```

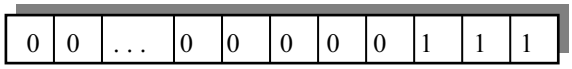
Beim Links-Shift werden entsprechende Nullen nachgezogen. Die Hexadezimalzahl `70` (Dezimalwert 112) ist das Ergebnis der Multiplikation von `1C` (Dezimalwert 28) mit 2**2.

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

Hier soll ebenfalls von der Zahl `0x1C` ausgegangen werden:



0x1C = 28 Dezimalwert

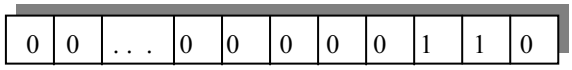


0x1C >> 2

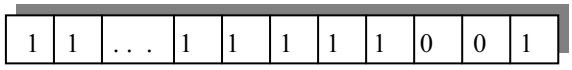
```
#!/usr/bin/perl -w

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

Auch hier ergibt sich das Ergebnis `7` (beim Rechts-Shift) durch Division von `1C` 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. Das B-Komplement erzielen Sie durch eine Addition einer 1 zum B-1-Komplement.

3.3 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.

Integrierte mathematische Standardfunktionen

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

Tabelle 3.3.1: 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*	Quadratwurzel
`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-Variable `$_` angewandt.

Es folgen nun einige Beispiele, wo mathematische Funktionen verwendet werden:

```
#!/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...
```

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";
```

3.4 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 und 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.

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.

Es gibt aber zwei Ausnahmen: Um ein Hochkomma bzw. einen Backslash im Text darzustellen, setzen Sie einfach einen Backslash davor:

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

print 'Jetzt gibt\'s einen \'Backslash\': \\';
# Jetzt gibt's einen 'Backslash': \
```

Als Alternative zu den Hochkommas können Sie auch das Präfix-Zeichen `q`, gefolgt von einem beliebigen paarigen Zeichen, verwenden. Paarige Zeichen sind passende öffnende und schließende Klammern oder sonstige Zeichen:

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

print q(Jetzt gibt's einen 'Backslash': \\ ),"\n";
# Jetzt gibt's einen 'Backslash': \
print q<Jetzt gibt's einen 'Backslash': \\ >,"\n";
# Jetzt gibt's einen 'Backslash': \
print q|Jetzt gibt's einen 'Backslash': \\ |,"\n";
# Jetzt gibt's einen 'Backslash': \
```

Anführungszeichen

Anführungszeichen unterstützen die Interpolation von Variablen und indizierten Ausdrücken. Jede skalare Variable (mit Präfix `$`) in Anführungszeichen wird also interpoliert, d. h. durch den aktuellen Wert ersetzt. Das Gleiche geschieht auch mit Array-Variablen (mit Präfix `@`). Hash-Variablen (mit Präfix `%`) können in dieser Form nicht interpoliert werden.

Bei einem String in Anführungszeichen erfahren auch spezielle Escape-Sequenzen eine besondere Bedeutung. So rufen Backslash-Interpretationen `\n` einen Zeilenvorschub, `\f` einen Seitenvorschub, `\t` einen Tabulator, `\u` einen Großbuchstaben als nächsten Buchstaben hervor, um nur einige Sequenzen zu nennen.

Es gibt auch Escape-Sequenzen mit Backslash und Zeichen, bei denen das Zeichen wörtlich interpretiert wird, das heißt es wird keine Aktion ausgeführt.

Möchten Sie obige Präfix-Zeichen `$`, `@` mit nachfolgenden Zeichen in einem String mit Anführungszeichen darstellen, brauchen Sie nur das Zeichen maskieren, d. h. einen Backslash voranstellen. Somit werden nur diese Zeichen dargestellt und es findet keine Interpolation der Variable statt:

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

$pi = 3.14;
print "\n";
print "Der Wert von \$pi ist $pi\n";
# Der Wert von $pi ist 3.14
```

Das obige Programm enthält die Escape-Sequenz `\n`. Es werden eine Backslash-Interpretation und ein Zeilenvorschub (New-Line) ausgeführt. In der nächsten Zeile wird zu-

nächst `\$pi` interpretiert, wörtlich das `$`-Zeichen dargestellt; anschließend folgt der Text `"pi"`. In der gleichen Zeile wird anschließend interpoliert und wieder New-Line ausgeführt.

Eine wörtliche Interpretation erzielen Sie auch im nächsten Beispiel. Zunächst werden die Anführungszeichen selbst durch Voranstellen des Backslashs dargestellt. Die skalare Variable `$var` wird aber interpoliert, d. h. ihr Wert `"Perl"` an dieser Stelle eingesetzt:

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

$var = 'Perl';
print "Nur \"$var\"!\n";                    # Nur "Perl"!
```

Im folgenden Beispiel ist ein Bereichs-Operator im Listenkontext gewählt worden:

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

@bereich = (1..5);
print "\@bereich von \"1 bis 5\": @bereich\n";
# @bereich von "1 bis 5": 1 2 3 4 5
print "Teilliste: @bereich[0,2]\n";
# Teilliste: 1 3
```

Auch bei diesem Beispiel erfährt die Array-Variable `@bereich` zunächst eine Interpretation des Zeichens `@`. Im Anschluss daran wird die Array-Variable interpoliert. Wie die letzte `print`-Zeile zeigt, können Sie auch bestimmte Elemente aus einem Array als Slice darstellen.

Als Alternative zu den Anführungszeichen können Sie auch das Präfix `qq` einsetzen:

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

$pi = 3.14;
print qq(Der Wert von \$pi ist $pi\n);
# Der wert von $pi ist 3.14
print qq<Der Wert von \$pi ist $pi\n>;
# Der wert von $pi ist 3.14
print qq<<Der <Wert von \$pi> ist> $pi\n>;
# <Der <Wert von $pi> ist> 3.14
```

Die letzte Zeile gibt eine Schachtelung des Strings an.

Quote Words Syntax

Neben den oben genannten quoting-Operatoren gibt es auch das `qw`-Konstrukt, mit dessen Hilfe Listen von Wörtern quotiert werden können, die durch Leerzeichen (Whitespace) getrennt sind. Die jeweiligen Listenelemente, die mit Anführungszeichen und durch Kommas

getrennt dargestellt werden, können Sie mit der qw-Syntax wesentlich lesbarer darstellen. Berücksichtigen Sie beim Einsatz dieser Syntax, dass Sie keine Kommas als Leerzeichen und auch keine Kommentare einfügen und dass nicht interpoliert wird.

Eine homogene Liste ohne qw-Syntax:

```
("ida", "ilse", "udo", "uwe");
```

Die gleiche Liste als qw-Konstrukt:

```
qw(ida ilse udo uwe);
```

Weitere Quote-Zeichen sind sogenannte Backticks.

Backticks
Die in Backticks ` ` eingeschlossenen Strings haben eine ganz spezielle Bedeutung. Zunächst werden sie wie doppelt quotierte Strings interpoliert. Anschließend werden sie von der Kommando-Shell als Befehl in Ihrem System ausgeführt:

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

print `echo \"Perl ist spitze\"`;
# "Perl ist spitze";
```

Berücksichtigen Sie, dass Sonderzeichen schon vor der Übergabe an die Shell ausgewertet werden. Sie müssen diese mit einem Backslash schützen:

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

print `echo \*`;
# *;
```

Im folgenden Beispiel werden alle Dateien - mit dem Platzhalter (Wildcart) „*“ und der Dateinamenserweiterung "pl" - aus dem aktuellen Verzeichnis sortiert ausgegeben:

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

$var = `dir *.pl`;      # Unix ls
print sort $var;
```

Werden im skalaren Kontext mehrere Zeilen ausgegeben, fassen die Backticks diese mit entsprechenden New-Lines zu einem großen String zusammen. Im Listenkontext wird die Ausgabe als Liste zurückgegeben.

Im nächsten Beispiel können Sie das Datum bzw. die Uhrzeit bestimmen:

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

print `date`;
print `time`;
```

Groß- und Kleinschreibung

Perl bietet zwei Varianten zur Konvertierung in Groß- bzw. Kleinschreibung. Operatoren zur Umwandlung der Groß- und Kleinschreibung verwenden die Unicode-Übersetzungstabellen. Mit den Funktionen `uc()` und `lc()` werden alle Strings in Groß- bzw. Kleinbuchstaben konvertiert. Mittels der Funktionen `ucfirst()` und `lcfirst()` wird jeweils nur das erste Zeichen des Strings in einen Groß- bzw. Kleinbuchstaben umgewandelt. Beachten Sie, dass `uc()` in Großbuchstaben umwandelt, während `ucfirst()` in Titelschreibweise (title case) umwandelt.

Beispiele mit Funktionen
Nachstehend einige Beispiele mit den oben genannten Funktionen:

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

$text = "strInG";
print uc($text),"\n";              # STRING
print lc($text),"\n";              # string
print ucfirst(lc($text)),"\n";     # String
```

Beispiele mit Backslash-Interpretation
Wie in der folgenden Tabelle dargestellt, können Sie mit der Backslash-Interpretation die Umschaltzeichen innerhalb von Stringliteralen nach folgendem Muster verändern:

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

$text = "strInG";
print "
\u$text                      # StrInG wie ucfirst()
\l$text                      # strInG wie lcfirst()
\U$text                      # STRING wie uc()
\L$text                      # string wie lc()
\Ustr\E\LInG\E V\LARIA\EBLE  # STRing VariaBLE
\n";
```

Für Umlaute und weitere nicht ASCII-Zeichen wird das entsprechende Pragma eingesetzt.

Tabelle 3.4.2: Escape-Sequenzen (Backslash-Interpretation)

Sequenz	Bedeutung
`\a`	Alarm (Bell)
`\b`	Rückschritt (Backspace)
`\c`	Escape (ESC)
`\f`	Seitenvorschub (Formfeed)
`\n`	Zeilenvorschub (New-Line)
`\r`	Wagenrücklauf (Carriage Return)
`\t`	Horizontaler Tabulator
`\v`	Vertikaler Tabulator
`\"`	Anführungszeichen
`\\`	Backslash
`\102`	Oktaler Zeichencode (hier für ‚B')
`\x43`	Hexadezimaler Zeichencode (hier für ‚C')
`\cC`	Control-Darstellung
`\l`	Folgender Buchstabe klein
`\u`	Folgender Buchstabe groß
`\L`	Kleinbuchstabe bis zum `\E`
`\U`	Großbuchstabe bis zum `\E`
`\Q`	Backslash nicht alphanumerisch
`\E`	Ende `\L`, `\U` oder `\Q`

Konkatenation und Vervielfachen von Strings

Speziell für die Verknüpfung von Strings besitzt Perl einen Operator, den Konkatenations-Operator. Mit dem `x`-Operator können Sie einen String beliebig oft wiederholen.

Konkatenation
Mit Hilfe des Konkatenations-Operators – ein Punkt „`.`" – lassen sich Strings zu einem neuen String verketten. Der Verkettungs-Operator wird auch als DOT-Operator bezeichnet. Es werden keine Leerzeichen zwischen den verketteten Strings erzeugt. Häufig erfolgt die Verkettung aber auch innerhalb der in Anführungszeichen stehenden Strings:

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

$vorname = "ida";
$nachname = "puschkin";
$name = $vorname  .  $nachname;
print "$name\n";              # idapuschkin
```

Mittels Konkatenations-Operators lassen sich die Variablen `$ida`, `$lisa` und `$ilse` wie folgt verketten:

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

$ida   = "Maschen";
$lisa  = "draht";
$ilse  = "zaun";

print "1. Ist der ",$ida.$lisa.$ilse," neu?\n";
# 1. Ist der Maschendrahtzaun neu?
```

Im nächsten Beispiel werden verschiedene Kombinationen der Verkettung angegeben:

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

$ida   = "Maschen";
$lisa  = "draht";
$ilse  = "zaun";

$uwe = "Maschen"  .  "draht"  .  "zaun";
# oder:
$udo = "Ist der " . $ida  .  $lisa  .  $ilse  .  " neu?";
# oder:
$tom = "Ist der $ida"  .  "$lisa"  .  "$ilse neu?";
# oder:
$kai = "Ist der ${ida}${lisa}${ilse} neu?";
print "
2. Ist der $uwe neu?
3. Ist der ",$uwe," neu?
4. Ist der ".$uwe." neu?
5. $udo
6. $tom
7. $kai
\n";

# Die Ausgabe der Kombinationen sieht wie folgt aus:
# 2. Ist der Maschendrahtzaun neu?
# 3. Ist der Maschendrahtzaun neu?
# 4. Ist der Maschendrahtzaun neu?
# 5. Ist der Maschendrahtzaun neu?
# 6. Ist der Maschendrahtzaun neu?
# 7. Ist der Maschendrahtzaun neu?
```

Vervielfachung
Beim Einsatz des dyadischen Operators `x` in Verbindung mit einer Zahl erfolgt eine Vervielfachung des jeweiligen Strings. Im skalaren Kontext wird dabei der linke Operand so oft wiederholt, wie es die Zahl des rechten Operanden spezifiziert:

```
print "**Alarm**" x 3,"\n";
# **Alarm****Alarm****Alarm**
```

Vervielfachung durch das HERE-Dokument:

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

$doku = <<BILD
*************
*     Nur   *
*     noch  *
*     Perl! *
*************
BILD
x 2;        print "HERE-Dokument:\n",$doku,"\n";
```

Die Ausgabe ist:

```
HERE-Dokument:
*************
*     Nur   *
*     noch  *
*     Perl! *
*************
*************
*     Nur   *
*     noch  *
*     Perl! *
*************
```

Operatoren für Vergleiche

Perl besitzt zwei unterschiedliche Arten von Vergleichs-Operatoren, eine zum Vergleich von Zahlen und eine zum Vergleich von Strings. Strings werden gemäß ASCII-Ordnung verglichen. Mit Gleichheits-Operatoren prüfen Sie, ob zwei Skalare gleich sind. Die relationalen Operatoren geben Auskunft darüber, ob ein Skalar größer ist als der andere. Logische Operatoren verwenden Sie für Bedingungs- und Schleifenoperationen, bei denen als Wahrheitswert ein Boolescher Wert `true` oder `false` vorhanden ist.

Tabelle 3.4.4: Operatoren für Vergleiche

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`

Da Perl bei Bedarf Strings in Zahlen und umgekehrt konvertiert, ist besonders darauf zu achten, ob es sich um Strings oder um Zahlen handelt. Das Vergleichen von Zahlen ist einfach, da nur in numerischer Reihenfolge verglichen wird. Beim Vergleich von Strings ist die ASCII-Reihenfolge maßgebend, d. h. Ziffern rangieren vor Großbuchstaben, Großbuchstaben vor Kleinbuchstaben. Für eine korrekte Einordnung der Umlaute verwenden Sie das Pragma **use locale.** Die nächsten Zeilen zeigen einige Beispiele:

```
#!/usr/bin/perl
                                        # Ohne Warmeldung
use locale;                             # Pragma POSIX-Locales
print "ü" gt "u", "\n";                 # 1, True
print "a" lt "b", "\n";                 # 1, True
print "a" eq "A", "\n";                 # Leerzeichen, False
print "hallo" eq "hallo  ","\n";        # Leerzeichen, False
print "2" gt "10", "\n";                # 1, True ,"1" ist vor "2"
print "abc" == "xyz","\n";              # 1, True ,beide 0
print 4 > "hallo","\n";                 # 1, True ,hallo = 0
print 4 lt "hallo","\n";                # 1, True 4 vor h
print "4.0" != "4","\n";                # Leerzeichen, False
print "4.0" ne "4","\n";                # 1, True 4.0 > 4
```

Bei diesen Vergleichen ist zu berücksichtigen, dass bei den Beispielen `"abc" == "xyz"` und `4 > "hallo"` die Strings numerisch verglichen werden und somit die vorhandenen Strings zum Wert `0` konvertieren.

Bei Vergleichen von einfachen Zeichenketten sind die obigen Vergleichs-Operatoren schneller als reguläre Ausdrücke und deshalb vorzuziehen. Bei einfachen Zeichenkettenoperationen, wie Extraktion von Zeichenketten und Umwandlung von Zeichen, empfehlen sich aufgrund ihrer Effizienz die später beschriebenen Funktionen `index()`, `rindex()` und `substr()`. Reguläre Ausdrücke sind sparsam einzusetzen.

Stringinkrement

Sowohl beim Präinkrement-Operator als auch beim Postinkrement-Operator wird inkrementiert. Der Unterschied liegt jedoch darin, ob der Operator vor einem Operanden (Präfix-Schreibweise) oder dahinter (Postfix-Schreibweise) steht.

Bei der Präfix-Schreibweise wird die Variable inkrementiert, bevor der Wert zurückgegeben wird. Bei der Postfix-Schreibweise wird erst der Wert zurückgegeben und dann inkrementiert. Es lassen sich auch Gleitpunktzahlen inkrementieren. Der Operator „`--`“ (Dekrement) lässt sich nur bei numerischen Argumenten einsetzen, da der String als Wert `0` interpretiert wird und beim Dekrement das Ergebnis um `1` vermindert ist:

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

$var = "nur";
$praefix = ++$var;
print "\$praefix = $praefix\n";          # $praefix = nus
print "\$var = $var\n";                  # $var = nus

$postfix = $var++;                       # $postfix = nus
print "\$postfix = $postfix\n";          # $postfix = nus
print "\$var = $var\n";                  # $var = nut
```

Wird die Variable nur in einem Stringkonzept verwendet und entspricht der String dem Muster von `/^[a-zA-Z]*[0-9]*$/`, dann realisiert das Inkrement wieder einen String, wobei jedes Zeichen innerhalb seiner Grenzen bleibt oder eventuell einen Übertrag erzeugt. Hierbei sind folgende Gesetzmäßigkeiten gegeben:

- Zahlen gelten von 0 bis 9,
- Kleine Buchstaben gelten von a bis z,
- Große Buchstaben gelten von A bis Z.

Die einzelnen Bereiche werden von Anfang bis Ende durchlaufen und der Durchlauf beginnt von Neuem. Erfolgt ein Überlauf bei Buchstaben, entscheidet der erste Buchstabe, ob kleine oder große Buchstaben verwendet werden:

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

$var = "Izz";
$praefix = ++$var;
print "\$praefix = $praefix\n";     # $praefix = Jaa

$var = "Zzz";
$praefix = ++$var;                  # Ueberlauf
print "\$praefix = $praefix\n";     # $praefix = AAaa
```

```
$var = "z9";
$praefix = ++$var;                    # Ueberlauf
print "\$praefix = $praefix\n";       # $praefix = aa0
```

In den obigen Beispielen wurde das letzte Zeichen des Strings jeweils alphabetisch um ein Zeichen erhöht. Nach Überschreiten der Grenzen des Bereiches wird das davor liegende Zeichen um ein Zeichen erhöht. Bei einem Überlauf hängt es vom vordersten Buchstaben ab, ob der folgende Buchstabe groß oder klein geschrieben wird.

Variablen mit Stringinkrement führen nicht zu den gleichen Ergebnissen wie Variablen im numerischen Kontext:

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

$var = "Izz";
$var += 1;                            # "Izz" numerisch 0
print "\$var = $var\n";               # $var = 1
```

Auch beim Dekrementieren wird der numerische Kontext verwendet:

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

$var = "Izz";
$neuvar = --$var;                     # "Izz" numerisch 0
print "\$neuvar = $neuvar\n";         # $neuvar = -1
```

Bestimmung der Stringlänge

Wenn Sie die Länge eines Strings bestimmen möchten, verwenden Sie die Funktion `length()`.

Die Funktion `length()`
`length` [*string*]

Die Funktion `length()` berechnet die Länge der Zeichenkette im *string*. Fehlt das Argument, wird der Inhalt der Spezial-Variable `$_` verwendet. Bei Perl brauchen Sie im Gegensatz zu C/C++ das abschließende `'\0'`-Zeichen eines Strings nicht zu berücksichtigen. Zur Bestimmung der Größe eines Arrays oder eines Hashes ist diese Funktion nicht geeignet. Hierfür bieten sich die Aufrufe `scalar` *@array* bzw. `scalar keys` *%hash* an:

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

print length "String-Laenge\n";       # 14
print length "Perl" x 2,   "\n";      # 8
```

Das `\n`-Zeichen am Zeilenende hat die Länge 1.