Die Programmiersprache Perl

Die folgenden Seiten stellen eine Einführung in die Programmierung mit Perl dar. Auf mehrfachen Wunsch hin stehen die Seiten nun nicht nur für Web-Browser im HTML-Format, sondern auch als PDF-Dokument (geeignet zum Ausdrucken) zur Verfügung.

Autor:	Eike Grote
E-Mail-Adresse:	`Eike.Grote@web.de`
WWW-Adresse:	`http://perl-seiten.homepage.t-online.de/`[1]
Version:	2.04 *(22.1.2011)*

Weitere Perl-Tutorials (vorwiegend auf Englisch) finden sich hier:

`http://perl-tutorial.org/`[2]

Download

Sowohl eine archivierte Version der HTML-Seiten als auch die PDF-Variante stehen auf den CPAN-Servern zum Herunterladen bereit:

PDF-Dokument:	`perl-tutorial-DE_2.04.pdf`
HTML (tar/gzip-Archiv):	`perl-tutorial-DE_2.04.tar.gz`
HTML (zip-Archiv):	`perl-tutorial-DE_2.04.zip`

Die oben genannten Dateien finden sich alle in meinem CPAN-Verzeichnis:

`http://www.cpan.org/authors/id/E/EI/EIKEG/doc/`

Darstellung der Beispiele

```
Der Quellcode von Beispielprogrammen wird in einem
solchen hellen Fenster präsentiert.
```

```
So werden Ausgaben dargestellt, wie sie auf einem
Terminal erscheinen würden.
```

Inhalt

1 Einführung **7**

1.1 Was ist Perl ? 7

1.2 Ein erstes Programm 7

2 Perls eigene Dokumentation **10**

2.1 Kommandozeilen-Tool `perldoc` 10

2.2 Andere Werkzeuge und Formate 11

3 Perl im Internet **12**

3.1 World Wide Web 12

3.2 Newsgroups 12

3.3 CPAN 13

4 Konstanten **14**

4.1 Zahlen 14

4.2 Zeichenketten 14

5 Variablen **17**

5.1 Skalare Variablen 17

5.2 Arrays 19

5.3 Hashes 21

6 Operatoren **23**

6.1 Zuweisung 23

6.2 Bit-Operatoren 23

6.3 Logik-Operatoren 24

6.4 Prioritäten 25

7 Mathematik **26**
7.1 Arithmetik 26
7.2 Vergleichsoperatoren 27
7.3 Funktionen 28
7.4 Zufallszahlen 28

8 Zeichenketten **30**
8.1 Operatoren 30
8.2 Vergleiche 31
8.3 Funktionen 31

9 Array-Funktionen **36**
9.1 `push` 36
9.2 `pop` 36
9.3 `unshift` 37
9.4 `shift` 37

10 Programmablauf **39**
10.1 Start 39
10.2 Einflußnahme auf Programm 40
10.3 Ende 41

11 Schleifen und Verzweigungen **43**
11.1 `while` und `until` 43
11.2 `for` und `foreach` 44
11.3 Bedingte Verzweigung mit `if` und `unless` 45
11.4 Beeinflussen des Ablaufs einer Schleife 46

12 Ein- und Ausgabe **47**
12.1 Terminal 47
12.2 Dateien 48
12.3 Pipes 51

13 Dateien und Verzeichnisse **52**
13.1 Zeitstempel einer Datei 52
13.2 Eigenschaften von Dateien 53

13.3 Globbing 54
13.4 Verzeichnisse 56
13.5 Dateifunktionen 57

14 Formate **60**
14.1 Einführung 60
14.2 Definition eines Formats 60
14.3 Musterzeile 61
14.4 Variablenzeile 63
14.5 Spezielle Variablen 64

15 Suchoperatoren **66**
15.1 Ersetzen einzelner Zeichen 66
15.2 Suchoperator 69

16 Reguläre Ausdrücke **71**
16.1 Einfache Zeichensuche 71
16.2 Zeichenklassen 74
16.3 Wiederholungen 77
16.4 Gruppierung 82
16.5 Alternativen 85
16.6 Ankerpunkte 86
16.7 Umgebung eines Musters 89
16.8 Kommentare 90

17 Mehr zur Zeichensuche **92**
17.1 Optionen beim Suchoperator 92
17.2 Optionen innerhalb eines regulären Ausdrucks 96
17.3 Spezielle Variablen 97
17.4 Suchen und Ersetzen 98

18 Unterprogramme **100**
18.1 Einführung 100
18.2 Lokale Variablen 101
18.3 Parameter 102

18.4 Rückgabewerte 103

19 Ausführen von Perl-Code mittels `eval` 105

20 Spezielle Variablen 107

20.1 Einführung 107

20.2 Die Variable `$_` 107

20.3 Allgemeine Informationen 109

20.4 PID,UID,GID 109

20.5 Fehlermeldungen 110

20.6 Weitere spezielle Variablen 110

21 Referenzen 111

21.1 Harte Referenzen 111

21.2 „Call by Reference“ 113

21.3 Referenzen auf Unterprogramme 115

22 Mehrdimensionale Arrays 118

22.1 Allgemeines 118

22.2 Wie es nicht geht 118

22.3 Verwendung von Referenzen 119

22.4 Anonyme Arrays 121

23 Binäre Bäume 124

23.1 Einführung 124

23.2 Implementierung in Perl 125

24 Schwartz’sche Transformation 130

24.1 Einführung 130

24.2 Sortieren 130

24.3 Effektive Sortierung 131

24.4 Die Funktion `map()` 134

24.5 Die Transformation 135

24.6 Geschwindigkeitsvergleich 135

25 Einbinden von Perl-Code 138

25.1 Ausführung von externem Code mit `do()` 138
25.2 Code-Einbindung mit Hilfe von `require()` 139
25.3 Verwendung von `use` 140

26 Module 143
26.1 Packages 143
26.2 Module 145
26.3 Exportieren von Namen 146
26.4 Standardmodule 147

27 Variablen und Symboltabellen 149
27.1 Symboltabellen von Packages 149
27.2 Globale Variablen und `our()` 150
27.3 Lexikalische Variablen mittels `my()` 152
27.4 Dynamische Variablen mittels `local()` 154
27.5 Unterschiede zwischen `my()` und `local()` 156

Kapitel 1

Einführung

1.1 Was ist Perl ?

Perl steht für *„practical extraction and report language"*, womit eigentlich schon alles gesagt ist. Ziel des Autors Larry Wall bei der Erstellung von Perl war es, eine Sprache zu entwickeln, die einerseits die wichtigsten Programmierbefehle wie Schleifen, Verzweigungen, etc. enthält und andererseits aber auch die Möglichkeit bietet, leicht Such- und Ersetzungsoperationen wie in einem Editor durchzuführen. Somit entstand Perl im wesentlichen als eine Synthese aus der Programmiersprache *C* und den UNIX-Funktionen `sed` und `awk`.

Die Programme, die man in Perl schreibt, werden als ASCII-Files gespeichert (wie ein Shell-Skript) und erst unmittelbar vor der Ausführung kompiliert. Dies macht Programme einerseits leicht editierbar und auch auf andere Rechnersysteme übertragbar, andererseits zeichnen sich Perl-Programme insbesondere bei Suchfunktionen durch eine hohe Geschwindigkeit aus.

Perl ist gemäß der *Artistic License* sowie (bei den neueren Versionen) unter der *GNU Public License* (*GPL*) einschließlich des Quellcodes frei verfügbar. Auf den meisten UNIX- bzw. Linux-Systemen ist Perl bereits vorinstalliert (unter dem Pfad */usr/local/bin/perl* oder */usr/bin/perl*). Außerdem existieren Portierungen für viele andere Betriebssysteme wie etwa Mac OS und Windows-Varianten, wobei bei Nicht-UNIX-Systemen bisweilen nicht die gesamte Funktionalität zur Verfügung steht. Die aktuelle (Stand Februar 2005) „stabile" Version ist Perl 5.8.6 (Anmerkung: Die Numerierung wurde beim Übergang von Version 5.005_03 zu 5.6.0 geändert). Die Nummer der auf einem Rechner installierten Perl-Version erhält man über „`perl -v`".

1.2 Ein erstes Programm

Ein Perl-Programm wird einfach mit einem Text-Editor (kein Textverarbeitungsprogramm) geschrieben und als Text-Datei (im wesentlichen ASCII-Code) abgespeichert. Unter UNIX/Linux ist noch darauf zu achten, das Execute-

Bit der Datei zu setzen, damit die Perl-Programme direkt aufgerufen werden können, also z.B. mittels

```
chmod u+x skript.pl
```

Zum Ausführen eines Perl-Programms ruft man dann auf der Kommandozeile

```
perl skript.pl
```

oder einfach

```
skript.pl
```

auf. Wird das Verzeichnis, in dem `skript.pl` liegt, nicht standardmäßig nach ausführbaren Programmen durchsucht (Umgebungsvariable `PATH`), so ist in der zweiten Variante der ganze Pfad anzugeben.

In der Datei `skript.pl` steht zum Beispiel folgendes:

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

print("Just another Perl hacker\n");
```

```
Just another Perl hacker
```

Die erste Zeile beginnt mit einem „`#`", was von Perl als Kommentar angesehen und damit nicht weiter beachtet wird. Diese Zeile dient dazu, der Shell mitzuteilen, daß es sich hierbei um ein Perl-Programm und nicht etwa ein Shell-Skript handelt. Der Pfad gibt dabei an, wo im System `perl` installiert ist (in diesem Tutorial wird immer der Pfad */usr/local/bin/perl* verwendet). Als Option für Perl empfiehlt es sich, zumindest immer „`-w`" anzugeben, da in diesem Falle umfangreichere Meldungen von Perl geliefert werden, falls beim Interpretieren des Codes Probleme auftreten.

Die letzte Zeile enthält die Perl-Funktion `print`, die als Argument eine Zeichenkette erwartet, welche dann auf dem Bildschirm ausgegeben wird. Das Symbol „`\n`" führt an dieser Stelle zu einem Zeilenvorschub. Bei der Funktion `print` können (wie bei einigen anderen Perl-Funktionen) die Klammern auch weggelassen werden:

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

print "Just another Perl hacker\n";       # (Kommentar)
```

Befehle/Funktionen werden in Perl immer mit einem Semikolon „`;`" abgeschlossen (es kann am Ende eines Blocks entfallen). Bei Schreibung der Funktionsnamen (und auch von Variablen etc.) ist stets auf Groß- oder Kleinschreibung zu

achten. An jeder Stelle eines Perl-Programms, an der ein Leerzeichen erlaubt ist, können anstelle eines einfachen Leerzeichens auch beliebig viele Tabulatoren, Zeilenvorschübe, etc. stehen. Kommentare können fast an beliebigen Stellen stehen; sie werden mit einem „`#`" eingeleitet und gelten dann bis zum Zeilenende.

Kapitel 2

Perls eigene Dokumentation

Jeder kompletten Perl-Distribution liegt eine umfangreiche Dokumentation bei, die sämtliche Funktionen und Eigenschaften der jeweiligen Version von Perl enthält. Dieses Perl-Tutorial stellt nur eine Auswahl der wichtigsten Features dieser Programmiersprache vor und läßt auch manche Feinheit weg.

Neben reinen Übersichten von Operatoren und Funktionen sind in den (englischsprachigen) Manual-Seiten auch „Frequently Asked Questions“ (*FAQ*) und Tutorials enthalten (z.B. zur objektorientierten Programmierung).

2.1 Kommandozeilen-Tool `perldoc`

Zugang zu den Manual-Seiten erhält man durch das Kommando „`perldoc`“ mit der gewünschten Seite als Argument. Als Einstieg mit einer Übersicht ist „`perl`“ gedacht.

```
perldoc perl
```

Alle Perl-Funktionen sind auf der recht umfangreichen Seite „`perlfunc`“ dokumentiert. Um eine Beschreibung dort direkt anzuspringen kann man die Option „`-f`“ verwenden.

```
perldoc -f print
```

Eine weitere sehr nützliche Option ist „`-q`“, welche eine Volltextsuche in den Fragen der FAQ-Seiten veranlaßt.

```
perldoc -q learn
```

2.2 Andere Werkzeuge und Formate

Auch für Benutzer, die keine Kommandozeile benutzen können oder wollen, gibt es Zugriffsmöglichkeiten.

Für praktisch jedes Betriebssystem geeignet sind die Manual-Seiten im Format HTML, da lediglich ein Browser benötigt wird. Der jeweilige Pfad zu den Seiten sollte in der Perl-Distribution dokumentiert sein.

Der Perl-Distribution *MacPerl* für das „klassische" Apple-Betriebssystem (bis einschließlich MacOS 9.x) liegt das Programm „Shuck" bei, mit dem die Manual-Seiten angezeigt und auch durchsucht werden können.

Sollte auf einem Rechner keine Perl-Dokumentation vorhanden sein (etwa, weil Perl dort nicht installiert ist) kann man die Seiten auch im Internet unter http://www.perl.com oder direkt unter http://www.perldoc.com finden.

Kapitel 3

Perl im Internet

Diese Seite enthält ein paar Verweise auf andere Perl-Seiten im WWW sowie Newsgroups. Außerdem gibt es einige Hinweise, wie man sich Perl-Software von den sogenannten CPAN-Servern holen kann.

3.1 World Wide Web

- www.perl.org[1]

  Eine umfangreiche Seite zu Perl mit vielen Links – hier findet man praktisch alles, was mit dieser Programmiersprache zu tun hat. *(englisch)*

- www.perl.com

  Ebenfalls ein Perl-Portal mit vielen weiterführenden Verweisen, unterstützt vom Verlag O'Reilly *(englisch)*

- FAQs, Hilfestellungen, Informationen zu Perl[2]

  Einige Dokumente rund um die deutschsprachigen Perl-Newsgroups

- Realm of CGI[3]

  Viele Leute wollen mit Perl in erster Linie CGI-Programmierung betreiben – hier eine deutschsprachige Seite dazu.

3.2 Newsgroups

Im Usenet beschäftigen sich folgende Gruppen mit Perl:

- de.comp.lang.perl.misc

  Deutschsprachige Gruppe zu Perl im allgemeinen

[1] http://www.perl.org/
[2] http://www.worldmusic.de/perl
[3] http://cgi.xwolf.de/

- de.comp.lang.perl.cgi

  Deutschsprachige Gruppe zu Perl und CGI

- comp.lang.perl.announce

  Ankündigung von Perl-Versionen, Modulen, usw. *(englisch)*

- comp.lang.perl.moderated

  Moderierte Perl-Newsgroup *(englisch)*

- comp.lang.perl.misc

  Diskussionsforum für alles, was nicht in eine der anderen Gruppen paßt *(englisch)*

- comp.lang.perl.modules

  Diskussionen über Perl-Module *(englisch)*

- comp.lang.perl.tk

  Diskussionen über Perl/Tk *(englisch)*

In alten Artikeln läßt sich beispielsweise bei Google[4] suchen und stöbern.

3.3 CPAN

Das sogenannte CPAN (*Comprehensive Perl Archive Network*) besteht aus einer Reihe von Servern auf der ganzen Welt, die jegliche Art von Perl-Software zum Download anbieten. Dort finden sich neben dem Perl-Interpreter (für eine Vielzahl von Rechnerplattformen) auch Module, Programmbeispiele und Dokumentation.

Einstiegsseite: http://www.cpan.org[5] .

[4]http://groups.google.de
[5]http://www.cpan.org

Kapitel 4

Konstanten

4.1 Zahlen

In Perl können Zahlen sowohl als Ganzzahlen (*integer*) als auch als Gleitkommazahlen (*floating-point*) dargestellt werden. Die interne Darstellung hängt von Betriebssystem und Compiler ab (z.B. *long* bzw. *double*). Um große Ganzzahlen übersichtlicher zu schreiben, können beliebige Unterstriche „`_`" eingefügt werden. Bei Gleitkommazahlen ist zu beachten, daß sie mit einem Punkt („`.`") anstelle des im deutschen Sprachraums üblichen Kommas geschrieben werden. Außerdem ist auch die wissenschaftliche Schreibweise aus Mantisse und Exponent getrennt durch den Buchstaben „`e`" oder „`E`" möglich.

Beispiele:

Ganzzahlen
`42`
`-9`
`1000000`
`1_000_000`

Gleitkommazahlen
`3.141`
`-0.005`
`1.5e-3 = 0.0015`
`2E4 = 20000`

Zahlen werden als Oktal- bzw. Hexadezimalzahlen betrachtet, wenn sie mit einer Null („`0`") bzw. „`0x`" beginnen.

dezimal	**oktal**	**hexadezimal**
`20`	`024`	`0x14`
`-63`	`-077`	`-0x3f`

4.2 Zeichenketten

Zeichenketten (*strings*) bestehen aus einer Aneinanderreihung von beliebig vielen einzelnen Zeichen (Buchstaben, Ziffern, Sonderzeichen). Bei der Definition

einer Zeichenkette in Perl ist auf die Einklammerung des Strings zu achten. Verwendet man einfache Anführungszeichen `'...'`, so wird die Kette im wesentlichen so gespeichert, wie sie geschrieben ist. Setzt man sie dagegen in doppelte Anführungszeichen `"..."`, findet u.U. eine Ersetzung bestimmter Sonderzeichen statt.

Beispiel (Hinweis: die Option „`-l`“ bewirkt hier nach jedem `print`-Befehl einen automatischen Zeilenvorschub):

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

print 'Hallo';
print 'Rock \'n\' Roll';
print 'C:\\DATA';
print 'Hello, world !\n';
print "Hello, world !\n";
print 'A\102\x43';
print "A\102\x43";
print "\"Max\"";
```

```
Hallo
Rock 'n' Roll
C:\DATA
Hello, world !\n
Hello, world !

A\102\x43
ABC
"Max"
```

Bei einfachen Anführungszeichen wird mit zwei Ausnahmen jedes Zeichen so in die Zeichenkette übernommen, wie es geschrieben wird. Will man ein Apostroph einbauen, so muß es durch einen Backslash „`\`“ markiert werden. Um schließlich einen solchen Backslash zu verwenden, wird er doppelt geschrieben.

Steht ein String in doppelten Anführungszeichen, so können Sonderzeichen, die sich nicht direkt durch ein Symbol darstellen lassen, mit Hilfe eines Backslash erzeugt werden.

Zeichen	Bedeutung
`\"`	doppeltes Anführungszeichen `"`
`\\`	Backslash `\`
`\n`	neue Zeile (*newline*)
`\r`	Wagenrücklauf (*return*)
`\f`	neue Seite (*form feed*)
`\t`	horizontaler Tabulator
`\v`	vertikaler Tabulator
`\b`	Rückschritt (*backspace*)
`\a`	akustisches Signal
`\e`	Escape
`\102`	oktaler Zeichencode (hier für 'B')
`\x43`	hexadezimaler Zeichencode (hier für 'C')
`\cC`	Control-Zeichen (hier: 'Ctrl-C' bzw. 'Strg-C')

Außerdem gibt es noch Ausdrücke, die die Groß- und Kleinschreibung beeinflussen:

Zeichen	Bedeutung
`\l`	nächster Buchstabe klein
`\u`	nächster Buchstabe groß
`\L`	Buchstaben bis `\E` klein
`\U`	Buchstaben bis `\E` groß
`\E`	(siehe `\L` und `\U`)

Beispiele:

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

print "\LHAUS";
print "\uregenwurm";
print "\UauTo\E\LBAhn\E";
```

```
haus
Regenwurm
AUTObahn
```

Kapitel 5

Variablen

5.1 Skalare Variablen

Zu einer „richtigen“ Programmiersprache gehören natürlich Variablen, die mit Werten besetzt werden können. Zunächst einmal sollen nur einfache „skalare“ Variablen betrachtet werden: Zahlen und Zeichenketten (*strings*). Im Unterschied zu vielen anderen Sprachen sehen die Variablen für Zahlen und Zeichen in Perl gleich aus: sie beginnen alle mit einem `$` gefolgt von ihrem Namen (Identifizierer). Der Name darf mit einem Buchstaben oder Unterstrich („_“) beginnen, gefolgt von im Prinzip beliebig vielen Buchstaben, Unterstrichen oder Ziffern.

Hinweis: Man sollte keine Variablen namens „`$a`“ oder „`$b`“ verwenden, da diese beim Sortieroperator `sort` eine besondere Bedeutung haben.

Um einer Variablen einen Wert zuzuweisen wird der Operator „`=`“ verwendet.

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

$ausgabe = "Just another Perl hacker\n";
$i = 123;
$x = 1e-5;

print "$ausgabe\n";
print "$i\n";
print "$x\n";
```

```
Just another Perl hacker

123
1e-05
```

Hier enthält die Variable `$ausgabe` den Wert „`Just another Perl hacker\n`“, die Variable `$i` den Wert *123* und `$x` den Wert der Gleitkommazahl *1e-05*.

Im Unterschied zu den meisten anderen Sprachen müssen Variablen in Perl nicht zu Beginn eines Programms deklariert werden. Um Fehler zu vermeiden, empfiehlt es sich aber grundsätzlich jede Variable zu deklarieren. Damit man keine Variable übersieht, setzt man „`use strict;`“ an den Anfang des Programms. In den meisten Fällen ist eine Deklaration mittels „`my`“ die richtige Wahl. Details zur Variablendeklaration finden sich im Abschnitt Variablen und Symboltabellen.

Auch für Zeichenketten muß kein Speicherplatz reserviert werden, ihre Länge ist in Perl im wesentlichen durch die Größe des Arbeitsspeichers begrenzt. Zahlen werden in Perl meistens als Gleitkommazahlen behandelt.

Ob eine Variable als Zahl oder Zeichenkette interpretiert wird, hängt von der Umgebung – dem Kontext – ab, in dem sie auftritt.

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

use strict;

my $x = 1;
my $y = 4;
my $c = $x + $y;
my $d = $x . $y;

print "$c\n";
print "$d\n";
```

```
5
14
```

Im obigen Beispiel ist „`+`“ der normale Operator zur Addition zweier Zahlen und „`.`“ der Verkettungsoperator, um zwei Strings zu einem zu verbinden.

Ein besonderer „Wert“ einer Variablen ist *undef*, der dann auftritt, wenn einer Variablen entweder überhaupt kein Wert oder explizit `undef` zugewiesen wird.

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

use strict;

my $w;
print "Initialisierung: ($w)\n";
$w = 'Hallo';
print "String: ($w)\n";
$w = '';
print "Leerstring: ($w)\n";
$w = undef;
print "undef: ($w)\n";
```

```
Use of uninitialized value in concatenation (.) or string
 at ./test.pl line 6.
Initialisierung: ()
String: (Hallo)
Leerstring: ()
Use of uninitialized value in concatenation (.) or string
 at ./test.pl line 12.
undef: ()
```

Hinweis: `undef` ist zu unterscheiden von etwa „0“ als numerischem Wert oder einem Leerstring; letztere sind im Sinne von Perl sehr wohl definiert!

Mit Hilfe der Funktion `defined()` kann man herausfinden, ob eine Variable einen definierten Wert hat oder nicht. Je nachdem ist der Rückgabewert „1“ oder ein Leerstring.

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

use strict;

my $w;
print "a) ",defined($w),"\n";
$w = 25;
print "b) ",defined($w),"\n";
```

```
a)
b) 1
```

5.2 Arrays

Eine Variable, die ein Array in Perl darstellt, ist durch das Zeichen „`@`“ gekennzeichnet. Da skalare Variablen und Arrays unabhängig voneinander verwaltet werden, darf es in einem Programm durchaus eine Zahl `$ALF` und ein Array `@ALF` geben.

Arrays sind grundsätzlich eindimensional (Vektoren) und enthalten als Elemente skalare Größen. Wie bei Zeichenketten muß auch bei Arrays keine Speicherplatzreservierung vorgenommen werden und Arrays können im Laufe eines Programms praktisch beliebige Größen annehmen. Versucht man, auf ein nicht gestetztes Element zuzugreifen, so bekommt man einen undefinierten Wert (*undef*) zurück.

Die Elemente eines Arrays können im wesentlichen auf zwei Arten mit Werten besetzt werden :

- `@vektor = (4,6,"ein String",25);`

Hier bekommen die ersten vier Elemente von `@vektor` die entsprechenden Werte.

- `$vektor[7] = 42;`

  Bei der Zuweisung über einen Index ist hier das Dollar-Zeichen `$` zu beachten, da „`vektor[7]`“ eine skalare Variable und kein Array darstellt! Die Indizierung der Arrays beginnt bei 0 (Eine Änderung dieses Verhaltens über die spezielle Variable `$[` ist nicht zu empfehlen!). In diesem Beispiel müssen übrigens die Inhalte `$vektor[0]` bis `$vektor[6]` überhaupt nicht besetzt werden.

Die Länge eines Arrays kann bestimmt werden, indem man die Array-Variable in einem skalaren Kontext verwendet; den Index des letzten Elements erhält man durch Voranstellen von „`$#`“ vor den Arraynamen.

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

use strict;

my @vektor = (4,6,"ein String",25);
my $laenge = @vektor;                 # skalarer Kontext !
print "Länge = $laenge\n";
print "Letzter Index = $#vektor\n";
print "\$vektor[1] = $vektor[1]\n";

$vektor[7] = 42;
$laenge = @vektor;
print "Länge = $laenge\n";
print "Letzter Index = $#vektor\n";
```

```
Länge = 4
Letzter Index = 3
$vektor[1] = 6
Länge = 8
Letzter Index = 7
```

Aus einem Array kann auch ein Ausschnitt (*slice*) genommen werden, indem der entsprechende Indexbereich angegeben wird.

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

use strict;

my @vektor = (2,4,6,8,10,12);
my @slice = @vektor;                # (2,4,6,8,10,12)
@slice = @vektor[2,3,4];            # (6,8,10)
@slice = @vektor[2,4];              # (6,10)
```

Ein spezielles Array ist `@ARGV`. Es enthält die Parameter/Optionen, die dem Skript beim Aufruf mit übergeben werden. Beispiel:

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

print "@ARGV\n";
```

Achtung: Im Gegensatz zur Programmiersprache C liefert `$ARGV[0]` den ersten Parameter und nicht den Namen des Programms (Dieser Filename befindet sich in Perl in der Variablen `$0`)!

5.3 Hashes

Kennzeichen eines Hashes (früher auch als „assoziatives Array“ bezeichnet) ist die Paarung von jeweils zwei Elementen in der Form „Schlüssel-Wert“. Gekennzeichnet wird ein Hash durch ein Prozentzeichen „`%`“ vor dem Variablennamen.

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

use strict;

my %alter = ("Sabine","27","Klaus","35","Ralf","22");
print "$alter{Sabine}\n";
print "$alter{Klaus}\n";
print "$alter{Ralf}\n";

my @schluessel = keys(%alter);
my @werte = values(%alter);

print "\@schluessel = @schluessel\n";
print "\@werte = @werte\n";
```

```
27
35
22
@schluessel = Ralf Sabine Klaus
@werte = 22 27 35
```

Hier wird auf die Werte der Schlüssel in `%alter` durch `$alter{...}` zugegriffen (man beachte auch hier das Dollar-Zeichen). Eine Liste aller Schlüssel in einem Hash erhält man durch die Funktion `keys()`, eine Liste aller Werte durch `values()`. Die Reihenfolge der Elemente ist dabei scheinbar willkürlich (sie hängt von der sog. *Hash-Funktion* ab, die Perl intern verwendet) und muß keineswegs mit der in der Definition übereinstimmen.

Man kann die Elemente eines Hashes mit Hilfe der Funktion `each()` auch paarweise jeweils in ein zweielementiges Array lesen. Bei jedem Aufruf wird dabei das nächste Paar zurückgegeben.

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

use strict;

my %h = ("Auto" => 1, "Haus" => 2, "Baum" => 3);

print each(%h),"\n";
print each(%h),"\n";
print each(%h),"\n";
```

```
Haus2
Baum3
Auto1
```

Auch hier ist die Reihenfolge (scheinbar) zufällig.

In diesem Beispiel wurde anstelle von Kommata ein Pfeil-Symbol („`=>`“) verwendet. Dieses Symbol ist einem Komma (fast) vollkommen gleichwertig und wird bei Hashes gerne der Übersichtlichkeit halber verwendet, um die Schlüssel-Wert-Paarungen zu verdeutlichen.

Auf Umgebungsvariablen (*environment*) kann über den speziellen Hash `%ENV` zugegriffen werden.

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

print $ENV{'LOGNAME'}."\n";
print $ENV{'DISPLAY'}."\n";
```

Kapitel 6

Operatoren

Die folgenden Abschnitte beinhalten einen Überblick über die wichtigsten Operatoren in Perl. Für eine vollständige Liste sei auf entsprechende Dokumentation hingewiesen.

6.1 Zuweisung

Die Zuweisung eines Wertes zu einer Variablen geschieht durch das „=“-Zeichen. Wie in C gibt es in Perl eine verkürzte Schreibweise für Operationen, die den Wert *einer* Variablen verändern. So kann z.B. eine Multiplikation statt

```
$i = $i * 5;
```

auch

```
$i *= 5;
```

geschrieben werden.

6.2 Bit-Operatoren

- „&“ bitweises UND
- „|“ bitweises ODER
- „^“ bitweises XOR (exklusives ODER)
- „~“ bitweises Komplement
- „<<“ bitweise Verschiebung des linken Arguments um eine (Ganz-) Zahl (rechtes Argument) nach links
- „>>“ bitweise Verschiebung des linken Arguments um eine (Ganz-) Zahl (rechtes Argument) nach rechts

6.3 Logik-Operatoren

- „`!`“ logisches NOT
- „`&&`“ logisches UND
- „`||`“ logisches ODER
- „`not`“ logisches NOT
- „`and`“ logisches UND
- „`or`“ logisches ODER
- „`xor`“ logisches XOR (exklusives ODER)

Anmerkung : die Operatoren „`not`“, „`and`“ und „`or`“ besitzen die gleiche Funktionalität wie die entprechenden symbolischen Operatoren „`!`“, „`&&`“ und „`||`“, aber eine andere Priorität.

6.4 Prioritäten

Hier nun eine Tabelle mit allen Perl-Operatoren, geordnet von der höchsten Priorität abwärts mit Angabe der jeweiligen Assoziativität:

<table>
<tr><th>Assoziativität</th><th>Operator</th></tr>
<tr><td>links</td><td>Terme (Variablen, geklammerte Ausdrücke,...)</td></tr>
<tr><td>links</td><td><code>-></code></td></tr>
<tr><td>-</td><td><code>++ --</code></td></tr>
<tr><td>rechts</td><td><code>**</code></td></tr>
<tr><td>rechts</td><td><code>! ~ \ + -</code> (unär)</td></tr>
<tr><td>links</td><td><code>=~ !~</code></td></tr>
<tr><td>links</td><td><code>* / % x</code></td></tr>
<tr><td>links</td><td><code>. + -</code> (binär)</td></tr>
<tr><td>links</td><td><code><< >></code></td></tr>
<tr><td>-</td><td>unäre Operatoren wie etwa Funktionen mit einem Argument</td></tr>
<tr><td>-</td><td><code>< > <= >= lt gt le ge</code></td></tr>
<tr><td>-</td><td><code>== != <=> eq ne cmp</code></td></tr>
<tr><td>links</td><td><code>&</code></td></tr>
<tr><td>links</td><td><code>| ^</code></td></tr>
<tr><td>links</td><td><code>&&</code></td></tr>
<tr><td>links</td><td><code>||</code></td></tr>
<tr><td>-</td><td><code>..</code></td></tr>
<tr><td>rechts</td><td><code>?:</code></td></tr>
<tr><td>rechts</td><td><code>= += -= *= usw.</code></td></tr>
<tr><td>links</td><td><code>, =></code></td></tr>
<tr><td>-</td><td>Listenoperatoren</td></tr>
<tr><td>links</td><td><code>not</code></td></tr>
<tr><td>links</td><td><code>and</code></td></tr>
<tr><td>links</td><td><code>or xor</code></td></tr>
</table>

Kapitel 7

Mathematik

Auch wenn Perl eigentlich keine Sprache für numerische Berechnungen ist, steht doch eine Reihe von mathematischen Operatoren und Funktionen zur Verfügung.

7.1 Arithmetik

- „+“ positives Vorzeichen (unär)
- „-“ negatives Vorzeichen (unär)
- „+“ Addition (binär)
- „-“ Subtraktion (binär)
- „*“ Multiplikation
- „/“ Division
- „%“ Rest einer Division (Modulo)
- „**“ Potenzbildung
- „++“ Inkrement
- „--“ Dekrement

Der In-(De-)krement-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.

Beispiel :

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

use strict;

my @array = (10,20,30);
my $i = 1;
print "$array[++$i]\n";
$i = 1;
print "$array[$i++]\n";
```

```
30
20
```

7.2 Vergleichsoperatoren

Operatoren, die ihre Operanden in irgendeiner Weise miteinander vergleichen, liefern im allgemeinen einen Wahrheitswert (oft als *boolean* bezeichnet) zurück. Ein solcher hat die beiden möglichen Werte „wahr“ (*true*) und „falsch“ (*false*). Perl kennt, im Gegensatz zu einigen anderen Programmiersprachen, hierfür keinen eigenen Datentyp, sondern setzt „`1`“ (Ziffer 1) als „wahren“ Wert und „“ (Leerstring) als „falschen“ Wert.

- „`==`“ liefert *wahr* bei Gleichheit
- „`!=`“ liefert *wahr* bei Ungleichheit
- „`>`“ liefert *wahr*, falls linkes Argument größer als rechtes Argument
- „`<`“ liefert *wahr*, falls linkes Argument kleiner als rechtes Argument
- „`>=`“ liefert *wahr*, falls linkes Argument größer oder gleich rechtem Argument
- „`<=`“ liefert *wahr*, falls linkes Argument kleiner oder gleich rechtem Argument
- „`<=>`“ liefert -1, 0 oder 1 je nachdem, ob linkes Argument kleiner, gleich oder größer als rechtem Argument

Achtung: Diese Vergleiche gelten nur im numerischen Kontext, d.h., wenn zwei Zahlen verglichen werden. Zum Vergleich von Zeichenketten sehen die Operatoren anders aus.

7.3 Funktionen

- `abs($x)`

  Absolutwert von `$x`

- `atan2($x,$y)`

  Arcustangens von `$x/$y` (zwischen $-\pi$ und $+\pi$, wobei π=*3,14159...*)

- `cos($x)`

  Cosinus von `$x` (im Bogenmaß)

- `exp($x)`

  Exponentialfunktion („*e* hoch `$x`“, wobei *e=2,71828...*)

- `log($x)`

  Natürlicher Logarithmus von `$x` (Achtung: `$x` muß positiv sein. Außerdem gilt: `$x == log(exp($x))` für alle `$x`)

- `sin($x)`

  Sinus-Funktion von `$x` (im Bogenmaß)

- `sqrt($x)`

  Quadratwurzel aus `$x` (Achtung: `$x` muß positiv sein)

Mit Ausnahme von `atan2()` können die Funktionen auch ohne explizites Argument geschrieben werden; sie werden dann auf den jeweiligen Wert von `$_` angewandt.

Anmerkung: Weitere mathematische Funktionen finden sich im sogenannten POSIX-Modul, das zur Standard-Bibliothek von Perl gehört. Beispiel:

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

use POSIX;

print log10(1000),"\n";    ## Logarithmus zur Basis 10
```

7.4 Zufallszahlen

(Pseudo-)Zufallszahlen werden in Perl mit der Funktion `rand()` erzeugt. Ohne Argument liefert sie Werte zwischen 0 und 1, ansonsten (Gleitkomma-) Zahlen zwischen 0 und dem Wert des Arguments. Um nicht jedesmal die gleiche Zahlenreihe zu erhalten, wird die Funktion `srand()` eingesetzt, die den Zufallszahlengenerator mit einer Zahl (Funktionsargument) initialisiert. Ohne Argument wird `srand(time)` ausgeführt, d.h. als Initialisierung dient die Zahl der Sekunden seit dem 1.1.1970.

Beispiel:

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

use strict;

for(my $i = 0;$i < 3;$i++) {
  print "(1)".rand()."\n"
}
print "\n";

srand(12345678);
for(my $i = 0;$i < 3;$i++) {
   print "(2)".rand(0.6)."\n"
}
print "\n";

srand();                        # srand(time);
for(my $i = 0;$i < 3;$i++) {
   print "(3)".rand(150)."\n"
}
```

```
(1)0.1558837890625
(1)0.480255126953125
(1)0.754150390625

(2)0.0451620008307167
(2)0.445994871769363
(2)0.139236410776795

(3)142.277526855469
(3)71.8826293945312
(3)148.493957519531
```

Bei mehrmaligem Abarbeiten des Skriptes ergeben sich nur bei den Zahlen der Gruppe `(2)` jeweils die gleichen Werte.

Kapitel 8

Zeichenketten

8.1 Operatoren

Zeichenketten lassen sich mit Hilfe des Operators „." verknüpfen und mit „x" vervielfachen.

Beispiel :

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

use strict;

my $x = "A";
my $y = ".";

my $c = $x.$y;
print "$c\n";

$c = $c x 2;
print "$c\n";

$c .= "*";
print "$c\n";

$c = $c x 3;
print "$c\n";
```

```
A.
A.A.
A.A.*
A.A.*A.A.*A.A.*
```

8.2 Vergleiche

Hier sind „größer" bzw. „kleiner" im Sinne einer Ordnung gemäß einer Zeichentabelle (ASCII, Unicode) zu verstehen und nicht etwa als Längenvergleich.

- „`eq`" liefert *wahr* bei Gleichheit
- „`ne`" liefert *wahr* bei Ungleichheit
- „`gt`" liefert *wahr*, falls linkes Argument größer als rechtes Argument
- „`lt`" liefert *wahr*, falls linkes Argument kleiner als rechtes Argument
- „`ge`" liefert *wahr*, falls linkes Argument größer oder gleich rechtem Argument
- „`le`" liefert *wahr*, falls linkes Argument kleiner oder gleich rechtem Argument
- „`cmp`" liefert -1, 0 oder 1 je nachdem, ob das linke Argument kleiner, gleich oder größer als das rechte Argument ist

Achtung: Diese Vergleiche gelten nur im String-Kontext, d.h., wenn zwei Zeichenketten verglichen werden. Zum Vergleich von Zahlen sehen die Operatoren anders aus.

8.3 Funktionen

- `substr()`

  Mit Hilfe von `substr()` lassen sich Teilstrings aus Zeichenketten extrahieren. Die Argumente sind der zu untersuchende String, die Startposition (beginnend bei 0) sowie die Länge des gewünschten Teilstrings. Läßt man das letzte Argument weg, so erstreckt sich der Teilstrings bis zum Ende der vorgegebenen Zeichenkette.

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

use strict;

my $t = 'Kalender';
print substr($t,1,4)."\n";
print substr($t,5)."\n";
```

```
alen
der
```

- `lc()` und `uc()`

  Diese Funktionen ersetzen *alle* Buchstaben in einem String durch die zugehörigen Klein- (`lc`, *lower case*) bzw. Großbuchstaben (`uc`, *upper case*).

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

use strict;

my $t = 'Perl';
print lc($t)."\n";
print uc($t)."\n";
```

```
perl
PERL
```

- `lcfirst()` und `ucfirst()`

  Im Gegensatz zu `lc` verwandelt `lcfirst` nur den *ersten* Buchstaben in einen Kleinbuchstaben (sofern er nicht bereits klein geschrieben ist). Analog setzt `ucfirst` nur den *ersten* Buchstaben in Großschrift.

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

use strict;

my $t = 'PERL';
print lcfirst($t)."\n";
$t = 'perl';
print ucfirst($t)."\n";
```

```
pERL
Perl
```

- `chop()`

  Hiermit wird das letzte Zeichen vom übergebenen String entfernt und als Funktionswert zurückgegeben.

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

use strict;

my $t = 'Perl';
chop($t);
print $t;
```

```
Per
```

Oft wird `chop()` dazu verwendet, den Zeilenvorschub am Ende einer Zeile abzuschneiden; hierfür eignet sich jedoch `chomp()` besser.

- `chomp()`

  Standardmäßig entfernt `chomp()` einen Zeilenvorschub („`\n`“) vom Ende eines Strings, sofern ein solcher vorhanden ist, und gibt die Anzahl der abgeschnittenen Zeichen zurück. `chomp()` sollte insbesondere dann gegenüber `chop()` bevorzugt werden, wenn nicht ganz sicher ist, ob die Zeichenkette am Ende einen Zeilenvorschub besitzt oder nicht.

  Genaugenommen entfernt `chomp()` am Ende eines Strings die Zeichenkette, die in der Variablen „`$/`“ steht (Standardwert: „`\n`“). Ist „`$/`“ leer, so werden alle Zeilenvorschübe abgeschnitten.

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

use strict;

my $string = "Hallo\n";
chomp($string);             # Abschneiden von \n

$string = "Hallo";
chomp($string);             # keine Änderung von $string

$/ = 'lo';
$string = "Hallo";
chomp($string);             # $string ist nun "Hal"

$/ = '';
$string = "Hallo\n\n\n";
chomp($string);             # Entfernen aller \n
```

- `length()`

  Hiermit wird die Länge einer Zeichenkette ermittelt (einschließlich aller Sonderzeichen).

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

use strict;

my $zeile = "Anzahl\tPreis\n";
print length($zeile)."\n";
```

```
13
```

- `join()`

  Das erste Argument ist hierbei eine Zeichenkette, die zwischen die nachfolgenden Strings beim Aneinanderhängen gesetzt wird.

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

use strict;

my $zeile = join('-','a','b','c','d');
print $zeile."\n";
```

```
a-b-c-d
```

- `split()`

  Hiermit kann eine Zeichenkette an bestimmten Stellen aufgetrennt werden. Die jeweilige Trennstelle gibt ein Suchmuster im ersten Argument an (siehe Reguläre Ausdrücke).

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

use strict;

my $zeile = "a-b-c-d";
my @a = split(/-/,$zeile);      # Anzahl der Elemente

print "@a\n";
```

```
a b c d
```

Um die Anzahl der Teilstrings zu begrenzen, kann als drittes Argument noch eine Zahl angegeben werden, die die maximale Anzahl der gebildeten Stücke darstellt.

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

use strict;

my $zeile = "a-b-c-d";
my @a = split(/-/,$zeile,2);

print "@a\n";
```

```
a b-c-d
```

In diesem Beispiel enthält `@a` am Ende nur die beiden Elemente „`a`“ und „`b-c-d`“.

- `chr()`

  Mit `chr()` kann man Zeichen über ihre Position im verwendeten Zeichensatz darstellen. Im folgenden Beispiel wird der Anfang des Alphabets im ASCII-Zeichensatz ausgegeben:

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

  print chr(65).chr(66).chr(67)."\n";
  ```

- `ord()`

  Die Umkehrung der obigen Funktion `chr()` ist `ord()`: Sie gibt die Position des ersten übergebenen Zeichens in der Zeichensatztabelle aus. Das nächste Beispiel zeigt die ASCII-Codes der ersten Buchstaben des Alphabets:

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

  print ord('A').' '.ord('B').' '.ord('C')."\n";
  ```

Kapitel 9

Array-Funktionen

9.1 push

Mit Hilfe der Funktion `push()` kann ein zusätzliches Element an ein Array angehängt werden. Die Größe des Arrays erhöht sich damit um eins. Statt eines einzelnen skalaren Elements können auch mehrere Elemente gleichzeitig hinzugefügt werden.

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

use strict;

my @s = ( 1, 2, 3 );
print "@s\n";
push(@s, 4);
print "@s\n";
push(@s, 5, 6);
print "@s\n";
```

```
1 2 3
1 2 3 4
1 2 3 4 5 6
```

9.2 pop

Das Gegenstück zu `push()` ist die Funktion `pop()`. Durch sie wird das letzte Element eines Arrays entfernt (und als Funktionswert zurückgegeben).

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

use strict;

my @s = ( 1, 2, 3 );
print "@s\n";
my $x = pop(@s);
print "@s\n";
print "\$x = $x\n";
```

```
1 2 3
1 2
$x = 3
```

9.3 unshift

Analog zu `push()` erweitert `unshift()` ein Array um ein oder mehrere Elemente. Allerdings werden die neuen Elemente am Anfang (Index 0) eingefügt, so daß alle bisherigen Elemente entsprechend „nach hinten rutschen".

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

use strict;

my @s = ( 1, 2, 3 );
print "@s\n";
unshift(@s, 0);
print "@s\n";
```

```
1 2 3
0 1 2 3
```

9.4 shift

Das Gegenteil von `unshift()` ist `shift()`. Diese Funktion entfernt das erste Element eines Arrays und läßt alle restlichen Elemente um einen Indexwert „nach vorne rücken".

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

use strict;

my @s = ( 1, 2, 3 );
print "@s\n";
my $x = shift(@s);
print "@s\n";
print "\$x = $x\n";
```

```
1 2 3
2 3
$x = 1
```

Kapitel 10

Programmablauf

10.1 Start

Der Perl-Code eines Programms ist zwar für einen (menschlichen) Programmierer mehr oder weniger gut lesbar, ein Mikroprozessor kann damit so aber nicht allzu viel anfangen. Daher muß der Code vor der Ausführung in die Sprache des Computers übersetzt bzw. kompiliert werden. Im Falle von Perl wird das komplette Programm unmittelbar vor der Ausführung in eine interne Form übersetzt. Treten dabei keine Fehler auf, arbeitet der Perl-Interpreter normalerweise die einzelnen Befehle Schritt für Schritt ab.

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

use strict;

print "1\n";
print "2\n";
print "3\n";
```

```
1
2
3
```

Mit Hilfe der speziellen Subroutine `BEGIN` kann Programmcode bereits in der Kompilierungsphase ausgeführt werden um Einfluß auf die Übersetzung des Rests des Programms zu nehmen. Mehrere `BEGIN`-Blöcke werden in der Reihenfolge des Auftretens abgearbeitet.

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

use strict;

print "1\n";
print "2\n";
BEGIN { print "BEGIN 1\n" }
print "3\n";
BEGIN { print "BEGIN 2\n" }
```

```
BEGIN 1
BEGIN 2
1
2
3
```

10.2 Einflußnahme auf Programm

Sogenannte Pragmas beeinflussen direkt die Übersetzung eines Perl-Programms. So sorgt etwa das immer empfehlenswerte `use strict` für eine strengere Kontrolle der verwendeten Variablen. Pragmas werden durch `use` aktiviert und durch `no` deaktiviert.

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

use strict;
$v = 7;

no strict;
$w = 9;
```

```
Global symbol "$v" requires explicit package name at
 ./test.pl line 4.
BEGIN not safe after errors--compilation aborted at
 ./test.pl line 6.
```

Wie man an der Fehlermeldung erkennt, wird nur die Variable `$v` beanstandet, nicht aber `$w`.

Neben Pragmas, die den Perl-Interpreter direkt beeinflussen, kann man mit Hilfe von `use` auch Module einbinden, die zusätzliche Variablen und Funktionen bereitstellen. Wie Module funktionieren und selbst entwickelt werden können ist unter „Module“ beschrieben.

Eine weitere Verwendungsmöglichkeit von `use` ist die Forderung nach einer bestimmten Mindestversion des Perl-Interpreters.

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

use strict;
use 5.8.2;

print "Nur ab Perl 5.8.2 lauffähig\n";
```

Das obige Beispielprogramm läuft nur, wenn der verwendete Perl-Interpreter die Version 5.8.2 oder höher hat; ansonsten erfolgt eine Fehlermeldung.

10.3 Ende

Nach dem Ausführen des letzten Kommandos beendet der Perl-Interpreter normalerweise automatisch die Arbeit. Analog zu `BEGIN` gibt es die spezielle Subroutine `END`, deren Inhalt nach den letzten Befehlen des Programmtextes aufgerufen wird. Existieren mehrere `END`-Blöcke, so werden sie in umgekehrter Reihenfolge abgearbeitet.

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

use strict;

print "1\n";
print "2\n";
END { print "END 1\n" }
print "3\n";
END { print "END 2\n" }
```

```
1
2
3
END 2
END 1
```

Möchte man ein Programm an einer bestimmten Stelle gezielt beenden, so kann man hierfür die Funktion `exit()` verwenden. Gibt man kein Argument an, so wird die Zahl 0 angenommen, was für einen „erfolgreichen“ Programmabbruch steht. Will man anzeigen, daß das frühe Programmende durch einen Fehler verursacht wurde, gibt man eine positive Zahl an. Die Verarbeitung dieses Wertes hängt allerdings vom Betriebssystem ab.

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

use strict;

print "1\n";
print "2\n";
exit 0;
print "3\n";
```

```
1
2
```

Soll ein Programm bei Auftreten eines Fehlers kontrolliert beendet werden, so verwendet man besser die Funktion `die()`, bei der man einen Text angeben kann, der als Fehlermeldung ausgegeben wird.

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

use strict;

die "Es ist ein Fehler aufgetreten."
```

```
Es ist ein Fehler aufgetreten. at ./test.pl line 5.
```

Schließt man den Fehlertext mit einem Zeilenvorschub („`\n`“) ab, so unterbleibt die Ausgabe von Programmname und Zeilennummer.

Kapitel 11

Schleifen und Verzweigungen

11.1 `while` und `until`

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

use strict;

my $i = 0;
while($i < 10) {
  print "$i\n";
  $i += 2;
}
until($i == 5) {
  print "$i\n";
  $i --;
}
```

Bei der `while`-Schleife wird zuerst das Argument überprüft. Falls dies *wahr* liefert, wird der Schleifenblock ausgeführt. Dies wiederholt sich so lange, bis das `while`-Argument *falsch* ist.

Die Negation der `while`-Schleife ist die Konstruktion mit Hilfe von `until`. Hier wird der Schleifenkörper so lange wiederholt bis das Argument von `until` den Wert *wahr* liefert.

Obiges Programm liefert demzufolge die Ausgabe

```
0
2
4
6
8
10
9
8
7
6
```

11.2 for und foreach

Folgendes Programm schreibt die Zahlen von 1 bis 10 auf den Bildschirm:

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

use strict;

for(my $i = 1;$i <= 10;$i++) { print "$i\n"; }
```

Zuerst wird die Anweisung im 1.Argument von `for` ausgeführt. Anschließend wird das 2.Argument überprüft. Liefert diese Abfrage eine wahre Aussage (*true*), so wird der Schleifenblock ausgeführt. Nach dem Block wird das 3.Argument von `for` ausgewertet und dann wieder das Argument Nummer 2 überprüft. Dies wiederholt sich so lange, bis die Bedingung (Argument 2) nicht mehr erfüllt ist (Wert *falsch*). Dann fährt das Programm in der Zeile nach dem Schleifenblock fort.

Die `for`-Schleife ist eigentlich nur eine vereinfachte Schreibweise; obiges Beispielprogramm läßt sich auch wie folgt schreiben:

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

use strict;

my $i = 1;
while($i <= 10) {
  print "$i\n";
  $i++;
}
```

Anstatt der drei durch Semikolons getrennten Argumente, kann `for` auch eine Liste übergeben werden, die dann sukzessive abgearbeitet wird. Aus Gründen der Übersichtlichkeit sollte in so einem Falle aber an Stelle von `for` das äquivalente `foreach` stehen. Die einzelnen Zeilen des folgenden Skripts leisten alle das gleiche: Sie geben die Zahlen von 1 bis 10 aus.

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

use strict;

for((1,2,3,4,5,6,7,8,9,10)) { print $_."\n" }
foreach((1,2,3,4,5,6,7,8,9,10)) { print $_."\n" }
foreach(1..10) { print $_."\n" }
foreach my $nr (1..10) { print $nr."\n" }
```

Die dritte und vierte Zeile verwenden den Bereichsoperator (*range operator*) aus zwei Punkten: „`..`“. Wird er, wie in diesem Falle, im Listenkontext benutzt, so liefert er eine Liste, die mit der ersten Zahl beginnt und mit der zweiten endet (ist der erste Operand größer als der zweite, erhält man eine leere Liste). Dies funktioniert auch mit Zeichenketten, wobei die Liste dann aus Strings besteht, die sich im alphabetischen Sinne vom ersten zum zweiten Operanden erstreckt.

In der letzten Zeile sieht man, wie das gerade von `foreach` bearbeitete Listenelement einer Variablen zugewiesen werden kann. Wird keine solche Variable angegeben, so steht der aktuelle Listenwert in der speziellen Variablen `$_`.

11.3 Bedingte Verzweigung mit `if` und `unless`

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

use strict;

for(my $i = 1;$i <= 5;$i ++) {
  if    ($i < 3)  { print "($i) kleiner als 3\n" }
  elsif ($i == 4) { print "($i) gleich 4\n" }
  elsif ($i > 4)  { print "($i) größer als 4\n" }
  else      { print "($i) keine Bedingung erfüllt\n" }

  unless($i == 2) { print "[$i] ungleich 2\n" }
}
```

```
(1) kleiner als 3
[1] ungleich 2
(2) kleiner als 3
(3) keine Bedingung erfüllt
[3] ungleich 2
(4) gleich 4
[4] ungleich 2
(5) größer als 4
[5] ungleich 2
```

Das Argument von `if` ist eine Bedingung, die entweder wahr (*true*) oder falsch (*false*) ist. Im Falle von „wahr“ wird der Block nach `if` ausgeführt. Ist die `if`-Bedingung falsch, so wird (falls vorhanden) das Argument des ersten `elsif` ausgewertet. Liefert auch dieses nicht den Wert *wahr*, kommt das nächste `elsif` an die Reihe; so lange bis entweder eine Bedingung wahr ist, keine Bedingungen mehr vorhanden sind oder ein abschließendes `else` erreicht wird, dessen Block ausgeführt wird, falls kein anderer Block in Frage kommt.

Im Gegensatz zu anderen Programmiersprachen wie z.B. C muß der Block nach einem `if`, `elsif` oder `else` immer in geschweiften Klammern stehen; dadurch werden eventuelle Mehrdeutigkeiten vermieden.

Das Gegenstück von `if` ist `unless`, dessen Block nur ausgeführt wird, wenn die Bedingung im Argument den Wert *falsch* liefert.

Eine Alternative zu einer `if-else`-Konstruktion ist die Verwendung des Operators „`?:`“:

```
$v == 1 ? print "v gleich 1" : print "v ungleich 1";
```

11.4 Beeinflussen des Ablaufs einer Schleife

Mit Hilfe der Funktionen `next` und `last` läßt sich der Ablauf einer Schleife gezielt steuern. So kann man einzelne Durchläufe überspringen oder eine Schleife vorzeitig beenden.

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

use strict;

for(my $i = 0; $i < 9; $i++) { print "$i " }
print "\n";

for(my $i = 0; $i < 9; $i++) {
    if($i == 4) { next }     # '4' überspringen
    print "$i ";
}
print "\n";

for(my $i = 0; $i < 9; $i++) {
    if($i == 4) { last }     # bei '4' abbrechen
    print "$i ";
}
print "\n";
```

```
0 1 2 3 4 5 6 7 8
0 1 2 3 5 6 7 8
0 1 2 3
```

Kapitel 12

Ein- und Ausgabe

12.1 Terminal

Um Ausgaben auf die Benutzerschnittstelle zu schreiben, wird am einfachsten die Funktion `print` verwendet. Es gibt außerdem noch die Funktion `printf`, die als Argument zunächst eine Zeichenkette erwartet, die das gewünschte Ausgabeformat beschreibt, und anschließend die entsprechenden Variablen (im wesentlichen identisch zur Funktion `printf()` in C). Eine interaktive Eingabe kann über `<STDIN>` eingelesen werden.

Beispiel:

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

use strict;

print "Name : ";
my $name = <STDIN>;
chomp($name);
printf("Hallo, %s !\n",$name);
```

```
Name : Eike
Hallo, Eike !
```

Hier wurde noch die Funktion `chomp()` benutzt; sie entfernt von der Zeichenkette im Argument den Zeilenvorschub „`\n`", mit dem die Eingabe von der Tastatur abgeschlossen wird.

Bei der Zuweisung von `<STDIN>` wird (wie oben im skalaren Kontext) jeweils eine Zeile, d.h., bis zum nächsten Vorschub, eingelesen.

Im Formatierungsstring von `printf` hat das Prozentzeichen „`%`" eine besondere Bedeutung, indem es ein Ausgabeformat einleitet. Die wichtigsten Formate zeigt folgende Tabelle:

Format	Beschreibung
%c	einzelnes Zeichen
%s	Zeichenkette
%d, %i	Ganzzahl
%o	Oktalzahl
%x	Hexadezimalzahl
%f	Gleitkommazahl
%e	wissenschaftliche Schreibweise
%%	explizites Prozentzeichen

Zwischen „%“ und dem Formatzeichen kann die gewünschte Ausgabebreite (nur vergrößern möglich, also z.B. Auffüllen mit Leerzeichen) bzw. Genauigkeit angegeben werden:

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

use strict;

printf "%d\n",12345;
printf "%7d\n",12345;

printf "%4.0f\n",123.45;
printf "%4.1f\n",123.45;
printf "%4.3f\n",123.45;

printf "%%%8s%%\n","Hallo";
```

```
12345
  12345
 123
123.5
123.450
%   Hallo%
```

12.2 Dateien

Will man Daten von einem File lesen, so muß diese Datei erst geöffnet und ihr ein „Filehandle“ zugewiesen werden. Anschließend kann über dieses Filehandle auf die Daten zugegriffen werden. Insbesondere beim Umgang mit Dateien (wie auch bei anderen Systemfunktionen) sollte immer anhand des Rückgabewertes kontrolliert werden, ob der Aufruf erfolgreich war.

Beispiel:

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

use strict;

my $rw = open(FILE,"demo.dat");
if(not defined($rw)) {
    die "Fehler beim Öffnen der Datei: $!\n";
}
while(defined(my $i = <FILE>)) { print $i }
close(FILE);
```

Die Funktion `open()` erwartet als erstes Argument das sogenannte Filehandle, das per Konvention aus Großbuchstaben besteht. Anschließend wird die zu öffnende Datei angegeben.

Obiges Programm ordnet über `<FILE>` der Variablen `$i` nacheinander jeweils eine Zeile der geöffneten Datei zu. Ist das Dateiende erreicht, kann keine Zuordnung mehr stattfinden, daher bleibt dann der Wert von `$i` undefiniert; dies wird durch die Funktion `defined()`überprüft („Lese solange Zeilen ein wie `$i` einen definierten Wert besitzt"). Nach Beendigung des Lesevorgangs wird die Datei mit `close()` wieder geschlossen; fehlt diese Funktion, so werden bei Programmende automatisch alle noch offenen Files geschlossen.

Im Falle eines Fehlers beim Dateizugriff ist der Rückgabewert von `open()` nicht definiert, und die spezielle Variable `$!` enthält die Fehlermeldung des Betriebssystems.

Kürzer und einprägsamer läßt sich die Überprüfung des Rückgabewertes wie folgt schreiben:

```
open(FILE,"demo.dat")
  or die "Fehler beim Öffnen der Datei: $!\n";
```

Um eine Datei zum Schreiben zu öffnen, muß dies durch ein „`>`" vor dem Dateinamen angegeben werden.

Beispiel:

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

use strict;

open(LESEN,"alt.dat")
  or die "Fehler beim Öffnen von 'alt.dat': $!\n";
open(SCHREIBEN,"> neu.dat")
  or die "Fehler beim Öffnen von 'neu.dat': $!\n";

while(defined(my $i = <LESEN>)) { print SCHREIBEN $i }

close(LESEN);
close(SCHREIBEN)
  or die "Fehler beim Schließen von 'neu.dat': $! \n"
```

Hier wird also der gesamte Inhalt von `alt.dat` gelesen und in die Datei `neu.dat` geschrieben. Zur Ausgabe auf ein File wird der Funktion `print` als erstes das Filehandle übergeben.

Beim Schreiben in eine Datei empfiehlt sich auch eine Überprüfung des Rückgabewertes von `close()`, um sicherzugehen, daß der Schreibvorgang erfolgreich abgeschlossen wurde.

Außer den vom Programmierer definierten Filehandles gibt es drei Standard-Handles:

- *STDIN*: Eingabe von Tastatur
- *STDOUT*: Ausgabe auf Terminal
- *STDERR*: Ausgabe von Fehlermeldungen (i.a. auch Terminal)

Das meist ohne Filehandle verwendete „`print`" ist also eigentlich nur eine Abkürzung für „`print STDOUT`".

Ebenso ist die Richtung des Datentransfers (Lesen oder Schreiben) an die entsprechenden Shell-Konventionen angelehnt. Zum Lesen einer Datei kann auch

```
open(LESEN,"< alt.dat");
```

geschrieben werden.

Will man Daten nur an die Daten eines bestehenden Files anhängen, so schreibt man „`>>`" vor den Dateinamen:

```
open(SCHREIBEN,">> neu.dat");
```

12.3 Pipes

Außer mit Dateien kann ein Perl-Programm auch Daten direkt mit Programmen austauschen (sofern dies vom Betriebssystem – wie z.B. UNIX – unterstützt wird). Dies funktioniert mit Hilfe von sogenannten Pipes. Zur Markierung wird dabei das Pipe-Symbol (ein senkrechter Strich „`|`“) verwendet.

Beispiel:

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

use strict;

open(DATUM,"date |");            # date : Datumsfunktion
print <DATUM>;
close(DATUM)
  or die "Fehler bei Ausführung von 'date'\n";

open(WORT,"| wc > wort.dat");        # wc : Zählprogramm
print WORT "eins zwei drei vier\n";
close(WORT)
  or die "Fehler bei Ausführung von 'wc'\n";
```

Im ersten Teil des Programms wird unter UNIX der Befehl `date` ausgeführt und das Ergebnis an das Perl-Skript geliefert. `print` gibt somit Datum und Uhrzeit des Systems aus.

Im zweiten Teil des Programms wird die Zeichenkette „`eins zwei drei vier`“ an das Programm `wc` übergeben, das dann die Zeichen, Wörter und Zeilen der Eingabe liest und das Ergebnis (1 Zeile, 4 Wörter, 20 Zeichen) in die Datei `wort.dat` schreibt (das Öffnen von Ein- und Ausgabepipes in einem Filehandle gleichzeitig ist nicht möglich).

Bei Pipes ist es wichtig, den Rückgabewert von `close()` (und nicht den von `open()`) zu überprüfen, um festzustellen, ob das Kommando ausgeführt werden konnte.

Kapitel 13

Dateien und Verzeichnisse

13.1 Zeitstempel einer Datei

Mit einer Datei sind i.a. Zeit- bzw. Datumsangaben verknüpft, mit deren Hilfe man beispielsweise herausfinden kann, wann der Inhalt einer Datei zum letzten Mal verändert wurde. Allerdings hängt es stark vom Filesystem (und damit vom Betriebssystem) ab, welche Zeitangaben mit einer Datei gespeichert werden und welche nicht.

Das ursprünglich unter UNIX entwickelte Perl ermöglicht den Zugriff auf die dort bekannten drei Zeitstempel:

- **Zugriffszeit** (*atime*): Hier wird festgehalten, wann zum letzten Male auf den Inhalt einer Datei zugegriffen wurde.
- **Modifikationsszeit** (*mtime*): Dieser Zeitstempel wird jedesmal aktualisiert, wenn der Inhalt einer Datei verändert wird.
- **Änderungszeit** (*ctime*): Hiermit werden Veränderungen von Dateieigenschaften (Größe, Zugriffsrechte,...) markiert.

Zeitstempel	**Dateiattribut**	**Funktion**	**veränderbar durch `utime()`?**
atime	`-A`	`stat[8]`	ja
mtime	`-M`	`stat[9]`	ja
ctime	`-C`	`stat[10]`	nein

Sowohl die Dateiattributoperatoren „`-A`“, „`-M`“ und „`-C`“ als auch die Funktion `stat()` erwarten entweder ein Filehandle oder einen Dateinamen als Argument. Allerdings unterscheiden sich die Rückgabewerte insofern, als daß erstere die Zeiten in Einheiten von Tagen in Bezug auf den Start des aufrufenden Programms (genauer: des Inhalts der speziellen Variable `$^T` bzw. `$BASETIME`) ausgeben, während `stat()` im Listenkontext in den Elementen 8 bis 10 jeweils den entsprechenden Zeitpunkt in Sekunden seit der „Epoche“ (1.1.1970, 00:00 GMT) enthält.

Anwendungsbeipiel:

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

use strict;

my $alter = (-M "test.pl");            # Gleitkommazahl!
print "Letzte Änderung vor $alter Tagen\n";
```

Es sei noch einmal darauf hingewiesen, daß obiges für UNIX-artige Systeme gilt – bei anderen Betriebssystemen sollte man in der jeweiligen Dokumentation nachsehen, welche Bedeutung die Rückgabewerte der genannten Operatoren bzw. `stat()` dort haben.

13.2 Eigenschaften von Dateien

Neben den bereits erläuterten Zeitangaben gibt es noch eine Reihe weiterer Dateieigenschaften, die mit Hilfe von Dateiattributoperatoren abgefragt werden können. Die folgende Liste enthält nur die wichtigsten davon:

- `-r` File kann gelesen werden
- `-w` File kann beschrieben/verändert werden
- `-x` Ausführbares Programm
- `-e` Datei existiert
- `-z` Datei hat die Länge Null
- `-s` Rückgabe der Dateilänge
- `-f` File ist einfache Datei
- `-d` File ist ein Verzeichnis
- `-l` File ist ein symbolischer Link

Anwendungsbeipiel:

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

use strict;

if(-r "test.pl") { print "lesbar\n" }
else { print "nicht lesbar\n" }

print "Länge ist : ".(-s "test.pl")."\n";
```

Die ebenfalls schon erwähnte Funktion `stat()` liefert im Listenkontext eine Reihe von Dateieigenschaften auf einmal zurück:

Index	**Eigenschaft**
0	Gerätenummer des Filesystems
1	Inode-Nummer
2	Zugriffsrechte
3	Anzahl der Hardlinks
4	Benutzer-ID des Dateieigentümers
5	Gruppen-ID der Datei
6	Geräteidentifikation
7	Dateigröße (in Bytes)
8	*atime*
9	*mtime*
10	*ctime*
11	Bevorzugte Blockgröße für Datentransfer
12	Anzahl der allozierten Blöcke

Da die Funktion `stat()` direkt die UNIX-Systemfunktion `stat()` abbildet, gilt obige Tabelle unter anderen Betriebssystemen u.U. nur mit Einschränkungen.

Beispiel:

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

use strict;

my $datei = "test.pl";
my @a = stat($datei);

print "Die Datei '$datei' ist $a[7] Bytes groß.\n";
```

13.3 Globbing

In einer UNIX-Shell gibt es die Möglichkeit, eine Liste von Files zu erhalten, indem man ein Muster vorgibt, das Platzhalter wie `*` oder `?` enthält. So liefert

```
ls -1 *.html
```

eine einspaltige (die Option hinter „`ls`“ ist die Ziffer „1“) Liste aller Files, die auf `.html` enden.

Einen analogen Mechanismus gibt es in Perl durch die Funktion `glob()` sowie den sog. Rhombus-Operator „`<>`“. Obiges Shell-Beispiel läßt sich dann wie folgt implementieren:

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

use strict;

foreach my $filename (glob("*.html")) {
    print $filename."\n"
}
```

Oder auch:

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

use strict;

foreach my $filename (<*.html>) {
    print $filename."\n"
}
```

Man kann auch die File-Liste direkt an ein Array übergeben:

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

use strict;

my @fileliste = glob("*.html");
foreach my $filename (@fileliste) {
    print $filename."\n"
}
```

Selbst Variablen können im Dateimuster verwendet werden:

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

use strict;

my $pfad = 'perl';
my @fileliste = <$pfad/*.html>;
foreach my $filename (@fileliste) { print $filename."\n" }

my $muster = '*.html';
@fileliste = <${muster}>;                    # Spezialfall
foreach my $filename (@fileliste) { print $filename."\n" }
```

Im obigen Beispiel muß im zweiten Teil `<${muster}>` anstelle von `<$muster>` stehen, da hier eine Variable alleine im Rhombus-Operator steht.

Wer es etwas genauer wissen will: in einem Ausdruck wie `<$muster>` wird `$muster` als „indirektes Filehandle“ betrachtet. Damit kann man beispielsweise Filehandles als Parameter an Unterprogramme übergeben.

Beispiel:

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

use strict;

open(FILE,'test.dat')
  or die "Kann Datei nicht lesen: $!\n";
print_file(\*FILE);
close(FILE);

sub print_file {
    my $handle = $_[0];
    while(<$handle>) { print }
}
```

13.4 Verzeichnisse

Ähnlich wie man Dateien zum Lesen öffnen kann, lassen sich auch Verzeichnisse behandeln, um auf die Dateinamen des Ordnerinhalts zugreifen zu können.

Beispiel:

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

use strict;

opendir(DIR,"perl");
while(my $datei = readdir(DIR)) { print $datei."\n" }
closedir(DIR);
```

Mit `opendir()` wird dabei das Verzeichnis geöffnet. Das erste Argument ist hierbei das sog. „Directoryhandle“, dessen Name analog zu den Filehandles aus Großbuchstaben bestehen sollte. Die Funktion `readdir()` liefert dann die einzelnen Einträge des Ordners. `closedir()` schließlich beendet den Zugriff auf das Verzeichnis.

Das aktuelle Verzeichnis kann plattformunabhängig mit Hilfe der Funktion `cwd()` des Standard-Moduls `Cwd` bestimmt werden.

Beispiel:

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

use strict;
use Cwd;

my $akt_verz = cwd();
print "aktuelles Verzeichnis : $akt_verz\n";
```

Ein Verzeichniswechsel ist durch die Funktion `chdir()` möglich. Dies ist allerdings abhängig vom Betriebssystem, unter dem Perl gerade läuft, wobei insbesondere auf das Zeichen zu achten ist, das Datei-/Verzeichnisnamen im Pfad voneinander trennt.

Als Beispiel seien hier UNIX und das „klassische" MacOS gegenübergestellt:

	UNIX	**MacOS Classic**
Pfad-Trennzeichen	`/`	`:`
Wechseln in höheres Verzeichnis	`chdir('..');`	`chdir('::');`
Öffnen des aktuellen Verzeichnisses	`opendir(ABC,'.');`	`opendir(ABC,':');`

13.5 Dateifunktionen

Zum Arbeiten mit Dateien gibt es folgende Funktionen; abhängig vom Betriebssystem stehen nicht immer alle zur Verfügung (z.B. gibt es keine Hard Links auf der klassischen Macintosh-Plattform):

- Änderung des Namens: `rename()`

```
#!/usr/local/bin/perl -w
use strict;

rename("alter_name","neuer_name");
```

- Löschen einer Datei: `unlink()`

```
#!/usr/local/bin/perl -w
use strict;

unlink("dateiname");
```

- Erzeugen eines Verzeichnisses: `mkdir()`

```
#!/usr/local/bin/perl -w
use strict;

my $permissions = 0777;
mkdir("dirname",$permissions);
```

- Löschen eines (leeren) Verzeichnisses: `rmdir()`

```
#!/usr/local/bin/perl -w
use strict;

rmdir("dirname");
```

- Hard Link erzeugen: `link()`

```
#!/usr/local/bin/perl -w
use strict;

link("filename","linkname");
```

- Soft Link erzeugen: `symlink()`

```
#!/usr/local/bin/perl -w
use strict;

symlink("filename","linkname");
```

- Ändern der Zugriffsrechte: `chmod()`

```
#!/usr/local/bin/perl -w
use strict;

my $permissions = 0755;
chmod($permissions,"prog_1","prog_2","prog_3");
```

- Ändern des Besitzers: `chown()`

```
#!/usr/local/bin/perl -w
use strict;

my $uid = 4987;                       # Benutzer-ID
my $gid = 33;                         # Gruppen-ID
chown($uid,$gid,"datei");
```

- Ändern der Zeitdaten: `utime()`

```
#!/usr/local/bin/perl -w
use strict;

my $atime = 812_000_000;          # letzter Zugriff
my $mtime = 822_000_000;          # letzte Inhaltsänderung
utime($atime,$mtime,"datei");
```

Die Zeiten sind hierbei in Sekunden seit dem 1.1.1970 anzugeben. Die aktuelle Systemzeit kann über den Operator `time` abgefragt werden:

```
#!/usr/local/bin/perl -w
use strict;

print "Seit dem 1.1.1970 sind ";
print time;
print " Sekunden vergangen.\n";
```

Kapitel 14

Formate

14.1 Einführung

Eine einfache Möglichkeit, Protokolle oder Tabellen übersichtlich auszugeben, bieten sogenannte Formate. Dort können beispielsweise Spalten einer Tabelle definiert werden; jeweils mit Breite und Positionierung des Eintrags (rechts-, linksbündig oder zentriert). Außerdem kann ein Seitenkopf definiert werden, der bei Ausgaben, die sich über mehrere Seiten erstrecken, auf jeder einzelnen Seite vor den eigentlichen Daten ausgedruckt wird.

Um Daten formatiert auszugeben, muß der Befehl `write` (optional mit einem Filehandle) benutzt werden. Es können für jedes Filehandle unabhängig voneinander Formate definiert werden.

14.2 Definition eines Formats

Das Schema einer Formatdefinition sieht wie folgt aus:

> `format` *Name* `=`
> *Musterzeile*
> *Variablenzeile*
> .

Im einfachsten Falle ist der Name eines Formats gleich dem Namen des Filehandles, für das das Format verwendet werden soll (Standardwert: `STDOUT`). Will man einer Formatdefiniton einen anderen Namen geben, so kann die entsprechende Zuordnung von Formatname und aktuellem Filehandle durch Setzen der Variablen `$~` geschehen.

Um bei mehrseitigen Dokumenten jeweils automatisch einen Seitenkopf ausgeben zu lassen, kann ein spezielles Format hierfür definiert werden. Der Name wird gebildet durch das Anhängen von „`_TOP`“ an das Filehandle (Standardwert: `STDOUT_TOP`). Alternativ dazu kann eine beliebiger Name durch Setzen von `$^`

verwendet werden. Ansonsten erfolgt die Definition vollkommen analog zu der eines normalen Formats.

Die Musterzeile enthält die Definitionen der einzelnen Felder, in die dann später die Werte der Variablen der darauffolgenden Zeile eingetragen werden. Es dürfen mehrere Muster- und Variablenzeilen angegeben werden; allerdings ist darauf zu achten, daß sie immer paarweise auftreten (jede Variablenliste „füllt“ die darüberstehende Musterzeile).

Außerdem können noch überall Kommentarzeilen eingefügt werden, die mit einem „#“ beginnen.

Die Definitionen von Formaten dürfen an beliebiger Stelle im Programmcode stehen (wie Unterprogramme).

14.3 Musterzeile

Jede dieser Zeilen bestimmt Felder in der Ausgabe, in die dann Variablenwerte (festgelegt in der jeweils darauffolgenden Zeile) eingesetzt werden sollen.

Ein normales Feld besteht aus einem „@“ gefolgt von null oder mehr Positionierungszeichen eines Typs:

„<“ (linksbündig)

„|“ (zentriert)

„>“ (rechtsbündig)

Beispiel:

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

use strict;

my $v = 12;

write;

format STDOUT =
#    eine Spalte, vierstellig, zentriert
@|||
$v
.
```

```
 12
```

Eine mehrzeilige Ausgabe von Zeichenketten wird durch Felder bewerkstelligt, die mit einem „^“ beginnen. Dabei wird dann dort, wo das Feld in der Formatdefinition zum ersten Mal erscheint, ein möglichst großer Teil des Ausgabestrings

dargestellt. Beim zweiten Auftreten wird ggf. dann ein Teil des Restes dargestellt, usw. (Achtung: in der Zeichenkette wird dabei sukzessive der jeweils dargestellte Teil entfernt, d.h., der Wert der Stringvariablen ändert sich u.U. bei jedem `write`).

Beispiel:

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

use strict;

my $text = "Dies ist ein Beispiel dafür, wie in einem ";
$text .= "Format mehrzeilige Textbeiträge dargestellt ";
$text .= "werden können.";

write;

format STDOUT =
#    bis zu drei Zeilen Text
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$text
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$text
^<<<<<<<<<<<<<<<<<<<<<<<< ...
$text
.
```

```
Dies ist ein Beispiel dafür,
wie in einem Format
mehrzeilige Textbeiträge ...
```

Oft weiß man vorher nicht, wie lang der Text wird. Um unnötige Leerzeilen zu vermeiden, setzt man einfach in die entsprechende Musterzeile eine Tilde („`~`“). Will man einen längeren Text auf jeden Fall komplett ausgeben, so kann man dies durch zwei aufeinanderfolgende Tilden („`~~`“) in einer Musterzeile erreichen. In diesem Falle wird die Ausgabe dieser Zeile so oft wiederholt bis die Zeichenkette in der dazugehörenden Stringvariable vollständig dargestellt ist. Schließlich kann ein Text in seiner natürlichen Zeilenaufteilung in ganzer Länge ausgegeben werden, indem der Stringvariablen in der Musterzeile das Feld „`@*`“ zugeordnet wird.

Eine besondere Art der Positionierung bietet das „`#`“. Nach einem „`@`“ bewirkt es eine rechtsbündige Darstellung, wobei bei der Ausgabe einer Zahl ein optionaler Dezimalpunkt berücksichtigt wird. Nach einem „`^`“ wird ein so markiertes Feld nur dann dargestellt, wenn die dazugehörende Variable definiert ist.

Beispiel:

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

use strict;

my $v = 117.127;
my $w = undef;
write;

format STDOUT =
Betrag: @###.## EUR (^###)
$v, $w
.
```

```
Betrag:  117.13 EUR (    )
```

14.4 Variablenzeile

Eine Zeile dieser Art legt fest, welche Variablenwerte in die jeweils vorangegangene Musterzeile eingetragen werden sollen.

Dies kann einfach durch eine durch Kommata getrennte Liste sein; aber es ist auch möglich, Arrays (die mehrere Felder der Musterzeile abdecken) oder gar ganze Ausdrücke anzugeben, die dann beim Aufruf des Formats (via `write`) abgearbeitet werden. Die Variablenliste kann über mehrere Zeilen ausgedehnt werden, indem geschweifte Klammern („`{...}`") verwendet werden.

Beispiel:

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

use strict;

my @datum = (6,1,1997);

my $x = 12;

write;

format STDOUT =
#    Datumsdarstellung
Datum: @>.@>.@>>>
@datum
#   Potenzen von $a
@>>>> @>>>> @>>>>
{
   $x,
   $x*$x,
   $x*$x*$x
}
.
```

```
Datum:  6. 1.1997
   12   144  1728
```

14.5 Spezielle Variablen

Wenn man statt der Kurzbezeichnungen sprechende Namen benutzen möchte, so kann kann man dies mit Hilfe des Standard-Moduls `English` erreichen, indem man am Programmanfang „`use English;`“ schreibt. Dann können auch die in Klammern stehenden Bezeichnungen verwendet werden.

- `$~` (`$FORMAT_NAME`)
  Name des aktuellen Formats
- `$^` (`$FORMAT_TOP_NAME`)
  Name des aktuellen Seitenkopfes
- `$%` (`$FORMAT_PAGE_NUMBER`)
  Aktuelle Seitenzahl
- `$=` (`$FORMAT_LINES_PER_PAGE`)
  Anzahl der Zeilen pro Seite

- `$-` (`$FORMAT_LINES_LEFT`)

  Anzahl der restlichen Zeilen auf der aktuellen Seite

- `$^L` (`$FORMAT_FORMFEED`)

  Zeichenkette, die vor jedem Seitenkopf ausgegeben wird (z.B. Seitenvorschub auf einem Drucker).

- `$:` (`$FORMAT_LINE_BREAK_CHARACTERS`)

  Liste von Zeichen, an deren Stelle in einem String dieser getrennt werden darf (zur Aufsplittung bei Ausgabe von längeren Texten über mehrere Zeilen)

Kapitel 15

Suchoperatoren

15.1 Ersetzen einzelner Zeichen

Will man bestimmte Symbole durch andere ersetzen (i.a. tritt anstelle eines Zeichens genau ein anderes), so verwendet man den Operator

`$string =~ tr/`*Suchliste*`/`*Ersetzungsliste*`/`*Optionen*`;`

Dabei sind Such- und Ersetzungsliste jeweils Listen aus Symbolen. Beim Ersetzen wird wie folgt vorgegangen: tritt in `$string` das erste Zeichen der Suchliste auf, so wird dieses Zeichen entfernt und an die Stelle wird das erste Zeichen der Ersetzungsliste gesetzt. Analog wird jedes Auftreten des zweiten Zeichens der Suchliste durch das zweite Zeichen der Ersetzungliste ersetzt usw.

Beispiel:

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

use strict;

my $s = "--abcd--cde--";
print "$s\n";

$s =~ tr/abc/ABC/;
print "$s\n";
```

```
--abcd--cde--
--ABCd--Cde--
```

Zur Vereinfachung können mittels eines Minuszeichens „`-`“ auch Zeichenbereiche angegeben werden (z.B. „alle Kleinbuchstaben“ durch „`a-z`“). Die Reihenfolge

der Zeichen ist dabei durch den Zeichensatz bestimmt, man sollte sich hier auf Buchstaben und Ziffern beschränken. Will man ein Minus ersetzen, so muß es wegen dieser Sonderbedeutung an Anfang oder Ende von Such- bzw. Ersetzungliste stehen.

Tritt ein Zeichen mehrmals in der Suchliste auf, so wird es durch dasjenige Symbol der Ersetzungliste ersetzt, das als erstes als Partner auftritt. Ist die Suchliste länger als die Ersetzungsliste, so wird das letzte Zeichen der Ersetzungsliste so oft wiederholt, bis beide Listen gleich lang sind (die überzähligen Symbole der Suchliste werden also alle dem letzten Zeichen der Ersetzungsliste zugeordnet). Ist die Ersetzungsliste leer, so wird der Inhalt der Suchliste dort eingetragen (dadurch finden zwar keine ersichtlichen Ersetzungen statt, aber der Operator kann dann zum Zählen von Zeichen verwendet werden, da der Rückgabewert die Anzahl der ersetzten Zeichen ist).

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

use strict;

my $s = "--abcd--cde--";
print "$s\n\n";

(my $t = $s) =~ tr/a-c/A-C/;   # entspricht 'tr/abc/ABC/'
print "$t\n";
($t = $s) =~ tr/ccc/123/;      # entspricht 'tr/c/1/'
print "$t\n";
($t = $s) =~ tr/-b/+-/;
print "$t\n";
($t = $s) =~ tr/abcde/xy/;  # entspricht 'tr/abcde/xyyyy/'
print "$t\n";
my $z = ($t = $s) =~ tr/-//;   # entspricht 'tr/-/-/'
print "$t ($z Minuszeichen in $s)\n";
```

```
--abcd--cde--

--ABCd--Cde--
--ab1d--1de--
++a-cd++cde++
--xyyy--yyy--
--abcd--cde-- (6 Minuszeichen in --abcd--cde--)
```

Anmerkung zur Zuweisung: Im obigen Beispiel wird jeweils der Variablen `$t` zunächst der Inhalt von `$s` zugewiesen, anschließend erfolgt dann die Ersetzung (nur in `$t`!). Die vorletzte Zeile zeigt noch, wie man den Rückgabewert der Ersetzungsoperation erhält.

Optionen:

- `c` („*complement*“)

Diese Option bedeutet für die Suchliste soviel wie „alle Zeichen außer...“. Beispiel:

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

use strict;

my $s = "abcd?%(1)---23ABCDEF4 xxx5\n";
print "$s\n";

$s =~ tr/0-9/./c;
print "$s\n";
```

```
abcd?%(1)---23ABCDEF4 xxx5

.......1....23......4....5.
```

Hier werden alle Zeichen, die keine Ziffern sind, durch Punkte ersetzt.

- `s` („*squash down*“)

  Werden aufeinander folgende Zeichen durch das gleiche Symbol ersetzt, so wird bei Verwendung dieser Option dieses Symbol nur einmal gesetzt.

  Erweiterung des obigen Beispiels:

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

use strict;

my $s = "abcd?%(1)---23ABCDEF4 xxx5\n";
print "$s\n";

$s =~ tr/0-9/./cs;
print "$s\n";
```

```
abcd?%(1)---23ABCDEF4 xxx5

.1.23.4.5.
```

- `d` („*delete*")

  Diese Option verhindert die Verlängerung der Ersetzungsliste, falls die Suchliste kürzer ist. Dies bewirkt, daß die überzähligen Zeichen der Suchliste gelöscht werden.

  Beispiel:

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

use strict;

my $s = "--abcd--cde--";
print "$s\n\n";

my $t;
($t = $s) =~ tr/abcde/xy/;  # entspricht 'tr/abcde/xyyyy/'
print "$t\n";
($t = $s) =~ tr/abcde/xy/d;
print "$t\n";
```

```
--abcd--cde--

--xyyy--yyy--
--xy----
```

Anstelle der Schrägstriche können im Ersetzungsoperator auch andere Zeichen gesetzt werden, wobei Klammern gesondert behandelt werden. Außerdem kann (in Anlehnung an den Ersetzungsoperator in *sed*) anstelle von „`tr`" auch „`y`" stehen.

15.2 Suchoperator

Eine der besonderen Stärken von Perl liegt in der Verwendung sogenannter „regulärer Ausdrücke" (*regular expressions*). Mit deren Hilfe lassen sich beispielsweise Zeichenketten nach bestimmten Teilstrings durchsuchen, wobei der reguläre Ausdruck wie eine Art Schablone wirkt und die Stelle gesucht wird, wo dieses Muster zum ersten Male paßt.

Der einfache Suchoperator, der reguläre Ausdrücke verwendet, sieht allgemein so aus:

`$string =~ m/`*Regexp*`/`*Optionen*`;`

Hier wird die Zeichenkette in `$string` danach durchsucht, ob sich der reguläre Ausdruck „*Regexp*" an irgendeiner Stelle darin anwenden läßt. Der Rückgabewert ist „wahr" (*true*), falls dies zutrifft, sonst „falsch" (*false*). Verwendet man

anstelle von „`=~`“ den Operator „`!~`“, so ist der Rückgabewert genau entgegengesetzt: *wahr* bei erfolgloser Suche.

Diejenige Zeichenkette, die auf das Suchmuster paßt, wird – sofern die Suche erfolgreich verläuft – in der speziellen Variablen „`$&`“ gespeichert.

Statt der Schrägstriche („`/.../`“) können auch einige andere Zeichen benutzt werden, sofern es sich dabei nicht um alphanumerische Symbole (Buchstaben, Ziffern, Unterstrich) oder Leerzeichen (*space*, *newline*, *tab*,...) handelt; verwendet man Schrägstriche, kann das „`m`“ am Anfang weggelassen werden.

Erläuterungen, was reguläre Ausdrücke sind und wie sie verwendet werden, finden sich in den nachfolgenden Abschnitten.

Kapitel 16

Reguläre Ausdrücke

16.1 Einfache Zeichensuche

Wir fangen mit dem einfachsten Beispiel an: Die Suche nach einem bestimmten Zeichen:

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

use strict;

my $t = "Demo-Zeichenkette";

if($t =~ /e/) { print "wahr\n" } else { print "falsch\n" }
```

Hier ist der reguläre Ausdruck nur der Buchstabe „`e`“ und der Operator „`=~`“ veranlaßt eine Suche nach diesem Zeichen in dem String `$t`. Diese Suche beginnt immer am Anfang einer Zeichenkette, so daß sie schon beim zweiten Zeichen erfolgreich ist (`Demo-Zeichenkette`); die Ausgabe des Programms ist also „`wahr`“.

Anstelle eines Zeichens können auch beliebig viele gesetzt werden – gesucht wird dann nach dieser Folge von Zeichen:

```
#!/usr/local/bin/perl -w
use strict;

if("Leder"  =~ /ede/) {                 # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if("Eder"   =~ /ede/) {                 # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
if("Gerade" =~ /ede/) {                 # Test 3
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Dieses Skript liefert nur in dem ersten Test „`wahr`“, ansonsten immer „`falsch`“, da der reguläre Ausdruck „`ede`“ genau passen muß. D.h., es wird hier auf Groß- und Kleinschreibung geachtet (Test 2) und der Ausdruck muß so wie er definiert ist in der Zeichenkette auftreten – es genügt nicht, daß die Buchstaben „`e`“, „`d`“ und „`e`“ irgendwo im durchsuchten String stehen (Test 3).

Zu beachten ist, daß nicht nach jedem Zeichen einfach gesucht werden kann wie im obigen Beispiel, da einige in regulären Ausdrücken besondere Bedeutung haben. Es handelt sich dabei um

```
. ? * + ^ $ | \ ( ) [ {
```

sowie um das Zeichen, das als „Klammer“ um den regulären Ausdruck verwendet wird (meistens der Schrägstrich „`/`“). Will man nach diesen Symbolen suchen, so muß man einen Backslash („`\`“) voranstellen:

```
#!/usr/local/bin/perl -w
use strict;

if("1...3"    =~ /\.\./)    {             # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if("/usr/bin" =~ /\/usr\//) {             # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
if("/usr/bin" =~ m#/usr/#)  {             # Test 3
    print "wahr\n";
} else {
    print "falsch\n";
}
if('??\$'     =~ /\\\$/)    {             # Test 4
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Diese Tests liefern alle „`wahr`“. Wie man beim Vergleich von Test 2 und Test 3 sieht, ist es beim Testen von UNIX-Pfadnamen oft sinnvoll, die Schrägstrich-Klammerung zu ersetzen, um allzuviele Backslashes zu vermeiden. Man beachte im Test 4 noch die einfachen Anführungsstriche, die verhindern, daß im String aus `\$` ein einfaches `$` wird.

Für einige häufig gebrauchte Zeichen, die sich schlecht direkt darstellen lassen, gibt es besondere Symbole (siehe auch unter Stringkonstanten):

Zeichen	Bedeutung
`\n`	neue Zeile
`\r`	Wagenrücklauf
`\f`	Seitenvorschub
`\t`	Tabulator
`\a`	Signalton
`\e`	Escape

Außerdem lassen sich auch alle Zeichen durch ihren ASCII-Code (in oktaler Darstellung) schreiben, indem man dem 3-stelligen Code einen Backslash voranstellt (so ist beispielsweise „`\101`“ äquivalent zum Buchstaben „`A`“). Anmerkung: es müssen immer drei Ziffern sein – also notfalls vorne mit einer oder zwei Nullen auffüllen.

Eine ganz besondere Bedeutung hat in regulären Ausdrücken der Punkt „`.`“: er steht normalerweise für jedes beliebige Zeichen mit Ausnahme des Zeilenvorschubs „`\n`“.

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

use strict;

my $t = "1.Spalte\t2.Spalte";    # mit Tabulatorzeichen \t

if($t =~ /e\t2/)  {                     # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /\t\t/)  {                     # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /p...e/) {                     # Test 3
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Test 1 ergibt „`wahr`“, da die Suche in der Mitte von `$t` erfolgreich ist. Test 2 dagegen liefert „`falsch`“ (keine zwei aufeinanderfolgenen Tabulatoren in `$t`). Test 3 wiederum verläuft erfolgreich – er findet einen zum regulären Ausdruck passenden Substring (`1.Spalte\t2.Spalte`).

16.2 Zeichenklassen

Bisweilen möchte man nicht nur nach einem bestimmten Zeichen suchen (beispielsweise dem Buchstaben „a“), sondern nach einem Zeichen, das einer Gruppe angehört (z.B. nach einem „Vokal“, d.h., nach einem Buchstaben aus der Menge {`a`,`e`,`i`,`o`,`u`}).

Eine solche Klasse kann mit Hilfe der eckigen Klammern („`[...]`“) definiert werden. Damit wird dann im Vorgabestring nach einem Symbol aus dieser Klasse gesucht.

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

use strict;

my $t = "Computer";

if($t =~ /[aeiou]/) {                    # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /[xyz]/)   {                    # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Hier liefert nur Test 1 „`wahr`“ („`o`“ aus der Klasse „`[aeiou]`“ in „`Computer`“ gefunden); die zweite Suche aber schlägt fehl (weder „`x`“ noch „`y`“ noch „`z`“ kommen in „`Computer`“ vor).

Auch in Zeichenklassen muß (wie bei einfachen Symbolen in regulären Ausdrücken) darauf geachtet werden, daß einige Zeichen eine besondere Bedeutung haben:

```
^ - ] \ $
```

Benötigt man eines dieser Zeichen explizit innerhalb einer Zeichenklasse, so muß man einen Backslash voranstellen.

Hier sei speziell die Wirkung von Zirkumflex („`^`“) und Minuszeichen („`-`“) für Zeichenklassen beschrieben. Ersterer bedeutet (wenn er unmittelbar nach der öffnenden Klammer („`[`“) steht) soviel wie „alle Zeichen außer ...“. Das Minus wird verwendet, um eine Reihe aufeinanderfolgender (im Sinne des verwendeten Zeichensatzes) Symbole abkürzend darzustellen (z.B. alle Kleinbuchstaben durch „`[a-z]`“).

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

use strict;

my $t = 'Ist "1" prim ?';

if($t =~ /[0-9]/)  {                    # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /["']/)   {                    # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /[^A-Z]/) {                    # Test 3
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Die Ausgabe ist jeweils „`wahr`“. Im ersten Test wird die Zahl `1` als Mitglied der Klasse der Ziffern `0-9` erkannt. Test 2 ist erfolgreich beim ersten Auftreten von doppelten Anführungszeichen vor der `1` (die einfachen Anführungszeichen in der Defintion `$t = ...` gehören ja nicht zu `$t`). Im dritten Test schließlich wird das „`s`“ in „`Ist`“ gefunden, da es das erste Zeichen ist, das nicht zur Klasse der Großbuchstaben gehört.

Für einige oft verwendete Zeichenklassen gibt es abkürzende Schreibweisen:

Zeichen	**Entsprechung**	**Bedeutung**
`\d`	`[0-9]`	Ziffer
`\D`	`[^0-9]`	Gegenstück zu `\d`
`\w`	`[a-zA-Z_0-9]`	Buchstabe, Unterstrich oder Ziffer
`\W`	`[^a-zA-Z_0-9]`	Gegenstück zu `\w`
`\s`	`[ \t\n\f\r]`	Leerzeichen
`\S`	`[^ \t\n\f\r]`	Gegenstück zu `\s`

(zur Erinnerung: `\t`, `\n`, `\f` und `\r` stehen für *Tabulator*, *neue Zeile*, *Seitenvorschub* und *Wagenrücklauf*.)

Ein Sonderfall ist das Zeichen „`\b`“, das innerhalb einer Klasse für einen Rückschritt (*backspace*) steht. Außerhalb einer Zeichenklasse besitzt „`\b`“ aber eine völlig andere Bedeutung, wie wir später sehen werden.

Diese Abkürzungen können wiederum in Klassen verwendet werden.

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

use strict;

my $t = 'Eine Zahl: -3.6209';

if($t =~ /[-\d.]/) {
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Die Klasse „`[-\d.]`“ veranlaßt die Suche nach dem ersten Auftreten eines der Symbole aus der Menge {`-`,`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`,`.`} (hier wird das Minuszeichen zuerst gefunden).

Anmerkungen: Das Minuszeichen besitzt hier keine Sonderbedeutung (wie in „`[a-z]`“), da es am Anfang der Klasse steht. Beim Punkt kann in Zeichenklassen auf das Voranstellen eines Backslashes verzichtet werden.

Selbstverständlich können einfache Zeichensuche und Zeichenklassen in regulären Ausdrücken miteinander kombiniert werden. Beispiel: Suche nach einem Teilstring, der eine Versionsnummer beschreibt.

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

use strict;

my $t = 'Hier ist Perl 5.002 installiert.';

if($t =~ /Perl [1-5]\.\d\d\d/) {
    print "wahr\n";
} else {
    print "falsch\n";
}
```

16.3 Wiederholungen

Besonders flexibel werden reguläre Ausdrücke durch die Möglichkeit nach dem mehrfachen Auftreten von Zeichen zu suchen (wobei „mehrfach“ auch „keinmal“ einschließt).

Dies erfolgt im allgemeinsten Falle durch die Angabe eines Zahlenbereichs in geschweiften Klammern. So bedeutet beispielsweise „`{2,5}`“, daß das vorangehende Symbol 2, 3, 4 oder 5-mal auftreten darf.

Ein Beispiel:

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

use strict;

my $t = "A----Z";

if($t =~ /-{1,3}/)  {                   # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /-{2,6}/)  {                   # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /-{5,10}/) {                   # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Test 1 gibt den Wert „`wahr`“ aus. Hier ist das gefundene Muster (in der Variablen `$&`) von besonderem Interesse: es werden drei Minuszeichen ausgegeben. Darin erkennt man eine wichtige Eigenschaft der Suche nach mehrfachen Zeichen: Perl versucht, möglichst viele davon zu finden. Dies sehen wir auch im zweiten Test: hier wird die Maximalzahl von vier Minuszeichen als gefundener Substring ausgegeben. Dagegen liefert der Test 3 den Wert „`falsch`“, da die Bedingung nicht einmal durch die kleinste Zahl (5) erfüllt werden kann.

Da solche Mehrfachsuchen sehr häufig in Perl benutzt werden, gibt es einige Abkürzungen, die die Schreibarbeit vereinfachen (und auch zur Übersichtlichkeit beitragen):

Abkürzung	**Entsprechung**	**Bedeutung**
{*n*}	{*n*,*n*}	genau *n*-mal
{*n*,}		mindestens *n*-mal
?	{*0*,*1*}	höchstens einmal
+	{*1*,}	mindestens einmal
*	{*0*,}	beliebig oft

Beispiele:

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

use strict;

my $t = "xxxAzz";

if($t =~ /x{2}/)  {                        # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /x{2,}/) {                        # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /z+/)    {                        # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /B?/)    {                        # Test 4
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /xAy*/)  {                        # Test 5
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /x*z+/)  {                        # Test 6
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Die Variable `$t` besteht alle Tests erfolgreich; die folgende Liste zeigt, an welchen Stellen jeweils der reguläre Ausdruck paßt (Perl sucht hier immer die größtmögliche Lösung):

- Test 1: `xxxAzz` (genau zwei aufeinanderfolgende „`x`")
- Test 2: `xxxAzz` (mindestens zwei aufeinanderfolgende „`x`")
- Test 3: `xxxAzz` (mindestens ein „`z`")
- Test 4: `xxxAzz` („`B`" wird 0-mal „gefunden" – daher auch hier „`wahr`")

- Test 5: `xxxAzz` („x“ gefolgt von „A“ evt. gefolgt von „y“)
- Test 6: `xxxAzz` (beliebig viele „x“ gefolgt von mindestens einem „z“)

Wie man in Test 5 sieht, bezieht sich ein Sonderzeichen zur Mehrfachsuche nur auf das direkt davor stehende Zeichen (im Beispiel also nur auf das „y“), nicht aber auf andere Symbole weiter vorne.

Wenn gesagt wird, daß Perl die größtmögliche Lösung sucht, so ist dies eigentlich nicht ganz richtig; man sollte genauer sagen: die längste Lösung, die sich bei der Suche ab dem Startpunkt des vorgegebenen Strings ergibt. Eine Suche beginnt immer vor dem ersten Zeichen der zu durchsuchenden Zeichenkette. Wird, von dieser Startposition ausgehend, eine Lösung gefunden, so wird die Suche (erfolgreich) abgebrochen, auch wenn es vielleicht weiter hinten noch eine längere Lösung gäbe. Bei einem Fehlschlag beginnt eine erneute Suche ein Zeichen weiter hinten; dieses Verfahren wiederholt sich so lange, bis eine Lösung gefunden oder das Ende des Strings erreicht wird.

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

use strict;

my $t = "aa==aaaa";

if($t =~ /a*/) {                          # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /=*/) {                          # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /=+/) {                          # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

In Test 1 bietet Perl als gefundenen Substring „aa“ („aa==aaaa“) an, obwohl die Lösung „aaaa“ („aa==aaaa“) länger wäre. Ein scheinbar paradoxes Verhalten zeigt Test 2: hier ist die Lösung leer! Die Erklärung dafür ist, daß von der Startposition ausgehend eine Lösung gefunden wurde (0-mal „=“) und dann die Suche abgebrochen wird. Test 3 schließlich gibt „==“ aus, da ja hier mindestens ein „=“ gefunden werden muß.

Die Problematik, daß Perl immer versucht, eine möglichst große Lösung zu finden, wurde schon angesprochen. Dies ist bisweilen unerwünscht; daher gibt es seit Perl 5 auch eine Variante, um eine möglichst *kurze* Lösung zu suchen. Dies

geschieht, indem einfach ein Fragezeichen angehängt wird. Die entsprechenden Kombinationen sehen dann so aus:

Quantifizierer	Bedeutung
`{`n`,`m`}?`	mindestens n-mal, höchstens m-mal
`{`n`}?`	genau n-mal (äquivalent zu `{`n`}`)
`{`n`,}?`	mindestens n-mal
`??`	höchstens einmal
`+?`	mindestens einmal
`*?`	beliebig oft

Ein Beispiel, das den Unterschied zwischen minimaler und maximaler Suche verdeutlicht:

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

use strict;

my $t = "mmmmm";

if($t =~ /m+/)  {                              # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /m+?/) {                              # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /m*?/) {                              # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Der erste Test gibt „`mmmmm`“ als (maximale) Lösung aus, während Test 2 nur „`m`“ (die minimale Lösung) liefert. Der dritte Test schließlich bietet als Lösung gar einen leeren String an (0-mal „`m`“).

Es sei hier noch erwähnt, daß beispielweise „`m*?`“ keinesfalls immer automatisch null Zeichen bedeuten muß wie man nach dem obigen Beispiel vielleicht zunächst glauben mag.

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

use strict;

my $t = "AmmmAmmmA";

if($t =~ /Am*?A/) {                          # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /A.+A/)  {                          # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /A.+?A/) {                          # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Hier enthält der gefundene Substring im Test 1 drei „`m`“ („`AmmmAmmmA`“). Lehrreich ist auch der Vergleich der beiden anderen Tests: maximale Lösung in Test 2 („`AmmmAmmmA`“) und minimale Lösung in Test 3 („`AmmmAmmmA`“).

16.4 Gruppierung

Im Abschnitt zur Mehrfachsuche wirkte eine entsprechende Anweisung immer nur auf das eine unmittelbar davor stehende Zeichen. Mit Hilfe von Klammern („`(...)`“) können auch längere Ausdrücke mehrfach gesucht werden.

Dazu ein Beispiel:

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

use strict;

my $t = "1.030303";

if($t =~ /03{2,3}/)   {                       # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /(03){2,3}/) {                       # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Test 1 liefert hier „`falsch`“ (es wird nach einer „`0`“ gefolgt von 2 oder 3 „`3`“en gesucht). Erfolgreich dagegen verläuft Test 2: er findet „`030303`“ (längste Lösung).

Klammern bewirken auch noch einen Nebeneffekt: Derjenige Substring, der auf das Muster der ersten Klammer paßt, wird in die Variable `$1` geschrieben, der zweite Substring in `$2`, usw. Zur Festlegung der Reihenfolge der Klammern zählt die jeweils öffnende Klammer (dies ist wichtig bei ineinander verschachtelten Klammerpaaren).

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

use strict;

my $t = "Breite: 7.5 m";

if($t =~ /(\w+): ([\d.]+ m)/) {
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}

print "\$1: $1\n";
print "\$2: $2\n";
```

Auch dieser recht kompliziert aussehende reguläre Ausdruck läßt sich einfach auflösen:

- „`(\w+)`“ sucht nach mindestens einem alphanumerischen Zeichen und speichert ggf. den gefundenen Substring in der Variablen `$1`.

- anschließend wird nach Doppelpunkt und Leerzeichen gesucht
- „`([\d.]+ m)`“ sucht nach mindestens einem Zeichen aus der Gruppe „Ziffern und Punkt“ gefolgt von einem Leerzeichen und dem Buchstaben „`m`“; das Ergebnis landet ggf. in `$2`.

Somit enthält schließlich `$1` den Wert „`Breite`“ und `$2` den Wert „`7.5 m`“. Da alle Bedingungen der Suche erfüllt sind, ergibt sich natürlich der Gesamtwert „`wahr`“.

Möchte man den Nebeneffekt der Zuweisung von `$1`, `$2`, `$3`,... verhindern, fügt man nach der öffnenden Klammer „`?:`“ ein.

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

use strict;

my $t = "a b";

$t =~ /(.)\s(.)/;
print "\$1: $1\n";      # gibt 'a' aus

$t =~ /(?:.)\s(.)/;
print "\$1: $1\n";      # gibt 'b' aus
```

Im ersten Test werden sowohl `$1` (mit „`a`“) als auch `$2` (mit „`b`“) belegt. Im zweiten Test dagegen findet bei der ersten Klammer keine Zuweisung statt und der Inhalt der zweiten Klammer („`b`“) wird in `$1` geschrieben.

Es ist sogar möglich, schon innerhalb des regulären Ausdrucks auf vorher im String gefundene Teilstrings zuzugreifen. So findet sich der Wert der ersten Klammer in `\1`, der der zweiten Klammer in `\2`, usw. Die Zuordnung ist identisch zu der von `$1`, `$2`, usw., allerdings dürfen die „Backslash-Variablen“ nur innerhalb des Suchmusters und die „Dollar-Variablen“ nur außerhalb verwendet werden.

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

use strict;

my $t = "abc--defg-h----ijk";

if($t =~ /(.{2}).+?\1/) {
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
print "\$1: $1\n";
```

Der reguläre Ausdruck sucht nach genau zwei gleichen Zeichen, die aber ansonsten beliebig gewählt sein können, gefolgt von einer möglichst kurzen Zeichenkette aus mindestens einem Symbol gefolgt von dem Zeichenpaar, das in der ersten Klammer gefunden wurde. Der gefundene Substring lautet dann „`--defg-h--`" und `$1` enthält wie erwartet „`--`".

Die Verwendung von `\1`, `\2`, usw. kann zu Problemen führen, wenn sich daran eine Suche nach Ziffern anschließt; eine mögliche Lösung ist die Benutzung von Klammern des Typs „`(?:...)`":

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

use strict;

my $t = "12--456--12";

if($t =~ /(--)\d+?\112/)     {                  # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /(--)\d+?(?:\1)12/) {                  # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Hier schlägt Test 1 fehl, da „`\112`" als Oktal-Code interpretiert wird (Zeichen „`J`"). Die zusätzliche Klammer in Test 2 sorgt für die korrekte Lösung („`--456--12`").

16.5 Alternativen

Wie man bei der Suche nach einzelnen Zeichen eine Auswahl vorgeben kann, haben wir im Abschnitt zu den Zeichenklassen gesehen. Natürlich möchte man diese Möglichkeit auch bei längeren Ausdrücken zur Verfügung haben. Dazu wird der senkrechte Strich („`|`") verwendet, der die einzelnen Alternativen voneinander trennt.

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

use strict;

my $t = "Perl";

if($t =~ /FORTRAN|C|Pascal|Perl/) {
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Bei der Suche werden die Alternativen von links beginnend durchprobiert bis ein passendes Muster gefunden wurde (hier: „`Perl`“).

Bei komplexeren Ausdrücken müssen meist Klammern gesetzt werden, damit klar ist, welche Teile zu den Alternativen gehören und welche nicht.

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

use strict;

my $t = "1997";

if($t =~ /1996|97|98/)   {
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /19(96|97|98)/) {
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Beide Tests sind zwar erfolgreich, liefern aber unterschiedliche Ergebnisse in `$&` („`97`“ bzw. „`1997`“)

16.6 Ankerpunkte

Ankerpunkte bieten die Möglichkeit festzulegen, daß der gesuchte Substring an einer bestimmten Stelle in der vorgegebenen Zeichenkette auftreten muß.

Soll nach einem Muster nur am Beginn eines Strings gesucht werden, so setze man einen Zirkumflex („`^`“) oder „`\A`“ an den Beginn des regulären Ausdrucks. Solange die untersuchte Zeichenkette keinen Zeilenvorschub enthält verhalten sich die beiden Ankerpunkte identisch (mehr hierzu im Abschnitt zu den Optionen von regulären Ausdrücken).

Analog erfolgt die Suche am Ende eines Strings: hier sind die Symbole, die am Ende des regulären Ausdrucks stehen, das Dollar-Zeichen („$“) bzw. „\Z“. Auch diese Ankerpunkte unterscheiden sich nicht in ihrer Wirkung solange keine Zeilenvorschübe vorhanden sind.

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

use strict;

my $t = "abrakadabra";

if($t =~ /^abra/)  {                      # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /abra$/)  {                      # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /^kada/)  {                      # Test 3
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /^abra$/) {                      # Test 4
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Test 1 findet den passenden Substring am Anfang von `$t` (`abrakadabra`), während Test 2 am Ende fündig wird (`abrakadabra`). Die Suche in Test 3 dagegen bleibt erfolglos; es ist zwar ein Substring „`kada`“ vorhanden, der steht aber nicht wie gefordert am Anfang von `$t`. Auch Test 4 liefert „`falsch`“; auch wenn „`abra`“ sowohl am Anfang als auch am Ende von `$t` steht, so sind dies doch zwei verschiedene Substrings.

Man beachte, daß diese Ankerpunkte gewissermaßen eine höhere Priorität besitzen als beispielsweise das Alternativ-Symbol („`|`“):

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

use strict;

my $t = "Mein Computer";

if($t =~ /^Rechner|Computer$/)   {          # Test 1
    print "wahr\n";
} else {
    print "falsch\n";
}
if($t =~ /^(Rechner|Computer)$/) {          # Test 2
    print "wahr\n";
} else {
    print "falsch\n";
}
```

Test 1 liefert den Wert „`wahr`“, da er „`Computer`“ am Ende von `$t` findet („Suche nach ‚`Rechner`‘am Anfang oder ‚`Computer`‘am Ende“). Test 2 dagegen gibt die Antwort „`falsch`“, da hier der Suchauftrag lautet: „Suche nach ‚`Rechner`‘oder ‚`Computer`‘, die sich von Anfang bis Ende des Strings erstrecken“.

Weitere Ankerpunkte können Wortgrenzen sein. Dabei ist eine solche Grenze definiert als der Punkt zwischen einem Zeichen aus der Klasse `\w` und einem aus `\W`. Eine solche Wortgrenze wird durch „`\b`“ dargestellt (dabei handelt es sich nicht um ein Zeichen an sich, sondern den Raum zwischen zwei Symbolen!). Das Gegenstück zu „`\b`“ ist „`\B`“ (Ankerpunkt inner- oder außerhalb eines Wortes).

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

use strict;

my $t = "Der ASCII-Code";

if($t =~ /\b[A-Z]/) {                 # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /\B[A-Z]/) {                 # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

Im Test 1 wird nach dem ersten Großbuchstaben an einer Wortgrenze gesucht; hier wird das „D“ („Der ASCII-Code“) gefunden, auch wenn es hier nicht (am Anfang des Strings) nach einem Zeichen aus `\W` steht, da Anfang und Ende

eine entsprechende Sonderbehandlung erfahren. Test 2 dagegen sucht nach einem Großbuchstaben innerhalb eines Wortes; die Lösung lautet hier „S“ („Der ASCII-Code“).

16.7 Umgebung eines Musters

Insbesondere bei der vorausschauenden und zurückblickenden Suche gab es im Laufe der einzelnen Perl-5-Versionen einige Erweiterungen.

Man kann damit den Erfolg einer Mustersuche davon abhängig machen, ob nicht nur das gesuchte Muster selbst paßt, sondern auch die Umgebung, d.h. die Zeichen davor und dahinter, mit berücksichtigen. So kann man etwa sagen: „Suche den Teilstring ‚Regen‘, aber nur, wenn er nicht von ‚wurm‘gefolgt wird.“

Folgende Klammerausdrücke stehen zur Verfügung:

Ausdruck	**Bedeutung**	**Erläuterung**
`(?=...)`	positive Vorausschau	nur dann erfolgreich, wenn ... folgt
`(?!...)`	negative Vorausschau	nur dann erfolgreich, wenn ... nicht folgt
`(?<=...)`	positive Rückschau	nur dann erfolgreich, wenn ... vorangeht
`(?<!...)`	negative Rückschau	nur dann erfolgreich, wenn ... nicht vorangeht

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

use strict;

my $t = "Bandwurm Regenwurm Regenschirm Regenschauer";

if($t =~ /Regen(?=scha)\w+/)  {                 # Test 1
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /Regen(?!wurm)\w+/)  {                 # Test 2
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /\w+(?<=Regen)wurm/) {                 # Test 3
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
if($t =~ /\w+(?<!Regen)wurm/) {                 # Test 4
    print "wahr ($&)\n";
} else {
    print "falsch\n";
}
```

```
wahr (Regenschauer)
wahr (Regenschirm)
wahr (Regenwurm)
wahr (Bandwurm)
```

Hierbei ist unbedingt zu beachten, daß der Teilstring, der zu den Ausdrücken in einer der Klammern gehört, nicht Bestandteil der Lösungsvariablen `$&` ist (er gehört nicht zum gefundenen Muster, sondern stellt nur eine Bedingung an die Umgebung dar). Im obigen Beispiel werden die vollständigen Wörter nur deswegen ausgegeben, weil die jeweiligen Worthälften auf das zusätzliche Muster „`\w+`“ passen.

16.8 Kommentare

Um bei komplexen regulären Ausdrücken die Übersicht zu behalten, können Kommentare in Klammern wie diesen „`(?#...)`“ eingefügt werden.

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

use strict;

my $t = "Wasserfall";

if($t =~ /((.)\2)(?# suche zwei gleiche Zeichen)/) {
    print "\$1: $1\n";              # gibt 'ss' aus
}
```

Man beachte, daß hier „\2“ verwendet werden muß, um das Zeichen, das auf „.“ paßt, zu verwenden, da der Punkt nach der *zweiten öffnenden* Klammer steht.

Kapitel 17

Mehr zur Zeichensuche

17.1 Optionen beim Suchoperator

Das Verhalten bei der Suche kann durch nachgestellte Optionen beeinflußt werden:

- g („*global*“)

  Während standardmäßig die Suche beim ersten Treffer abgebrochen wird, veranlaßt diese Option die Ausgabe einer Liste von Lösungen, so als würde man die Stelle nach einem gefundenen Substring als Startpunkt für eine erneute Suche verwenden.

  Beispiel:

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

use strict;

my $t = "123-456-789";

my @a = ($t =~ /[1-9]{3}/g);    # Suche nach Zifferntripel
foreach my $x (@a) { print "$x " }
print "\n";

foreach my $x ($t =~ /[1-9]{3}/g) { print "$x " }
print "\n";

while($t =~ /[1-9]{3}/g) { print "$& " }
print "\n";
```

```
123 456 789
123 456 789
123 456 789
```

Der erste Teil zeigt, wie die Suchoperation eine Liste aller gefundenen Substrings zurückgibt. Anstelle der Zuweisung zu einem Array kann auch `foreach` direkt die Lösungsliste durchlaufen. Sehr häufig verwendet man auch die dritte dargestellte Methode: schrittweises Durchsuchen der Zeichenkette mittels einer `while`-Schleife.

Es ist zu beachten, daß die Suche nach einer weiteren Lösung jeweils hinter dem zuvor gefundenen Muster beginnt. Daher gibt folgendes Skript nur die eine Lösung „`-ab-`" aus (und nicht noch „`-cd-`" zusätzlich).

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

use strict;

my $t = "-ab-cd-";

foreach($t =~ /-[a-z][a-z]-/g) { print "$_ " }
```

- `i` („*case-insensitive*")

  Verwendet man diese Option, wird bei der Suche nach Buchstaben nicht zwischen Groß- und Kleinschreibung unterschieden (dies funktioniert auch bei nationalen Sonderzeichen wie den deutschen Umlauten sofern das Betriebssystem die entsprechende Sprachumgebung unterstützt).

  Beispiel:

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

use strict;

my $t = "Ähnlichkeit";

if($t =~ /ähn/)  { print "wahr\n" }
    else { print "falsch\n" }
if($t =~ /ähn/i) { print "wahr\n" }
    else { print "falsch\n" }
```

```
falsch
wahr
```

- `s` („*single line*")

  Hiermit wird eine Zeichenkette als einzelne Zeile betrachtet, auch wenn sie Zeilenvorschübe („`\n`") enthält. Dadurch entfällt dann in einem Suchmuster die Sonderrolle von „`\n`" bezüglich des Punktes („`.`"), der sonst nur auf alle anderen Zeichen, nicht aber den Zeilenvorschub, paßt.

Beispiel:

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

use strict;

my $t = "AAA\nBBB\n";

if($t =~ /A.B/)  { print "wahr\n" }
    else { print "falsch\n" }
if($t =~ /A.B/s) { print "wahr\n" }
    else { print "falsch\n" }
```

```
falsch
wahr
```

- `m` („*multiple lines*“)

  Verwendet man diese Option, so passen die Ankerpunkte „`^`“ und „`$`“ nicht nur am Anfang bzw. Ende der vorgegebenen Zeichenkette, sondern auch an jedem Zeilenanfang und -ende. Um einen Ankerpunkt an den Anfang oder das Ende des untersuchten Strings zu setzen, muß in diesem Falle „`\A`“ bzw. „`\Z`“ verwendet werden.

  Beispiel:

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

use strict;

my $t = "Kugel 1\nPyramide 2\nQuader 3\n";

my @a = ($t =~ /^\w+/g);
foreach (@a) { print "1) $_\n" }
@a = ($t =~ /^\w+/gm);
foreach (@a) { print "2) $_\n" }
```

```
1) Kugel
2) Kugel
2) Pyramide
2) Quader
```

- `o` („*compile once*“)

  Diese Option dient der Optimierung der Mustersuche (hinsichtlich der Geschwindigkeit) und sollte nur verwendet werden, wenn dies unbedingt nötig ist. Ihre Wirkung besteht darin, daß eine Variablenersetzung im Suchmuster nur einmal zu Beginn stattfindet. Daher muß sichergestellt sein, daß sich die entsprechenden Variablen während der Suche nicht ändern.

Beispiel:

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

use strict;

my $t = "123..abc...456";

my $muster = '0-9';
while($t =~ /[$muster]+/g)
{
    $muster = 'a-z';          # Zeichenklasse ändern
    print "1) $&\n";
}

$muster = '0-9';
while($t =~ /[$muster]+/go)
{
    $muster = 'a-z';          # hier wirkungslos
    print "2) $&\n";
}
```

```
1) 123
1) abc
2) 123
2) 456
```

Hier sieht man nebenbei, daß Suchmuster durchaus auch in Variablen gespeichert werden können.

- x („*extended*“)

  Um komplizierte reguläre Ausdrücke übersichtlicher darstellen zu können, gibt es die Option „x“, die es ermöglicht, Ausdrücke auf mehrere Zeilen zu verteilen und (normale) Perl-Kommentare einzufügen.

Beispiel:

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

use strict;

my $t = '...<IMG SRC="pfad/bild.gif" WIDTH=110>...';

$t =~ /<IMG         # HTML-Tag für Bild (Beginn)
           \s+        # Leerzeichen
           SRC="      # Hier kommt der URL
             (.+?)    # Diesen Pfad suchen wir
           "          # Ende des URL
           .*?        # vielleicht noch Optionen
         >          # Ende des HTML-Tags
      /ix;        # case-insensitive, extended

print "$1\n";
```

```
pfad/bild.gif
```

17.2 Optionen innerhalb eines regulären Ausdrucks

Die im vorherigen Abschnitt beschriebenen global wirkenden Optionen können auch innerhalb eines regulären Ausdrucks gesetzt werden, um beispielsweise nur Teile davon zu beeinflussen. Dies wird durch „`(?`*Option(en)*`)`“ bewerkstelligt. Hierbei können die Optionen `i`, `m`, `s` und `x` verwendet werden. Die Wirkung erstreckt sich bis zum Ende des regulären Ausdrucks, innerhalb einer Gruppierung oder bis zu einer Deaktivierung mittels „`(?-`*Option(en)*`)`“.

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

use strict;

my $t = "ABcd";

$t =~ /[a-z][a-z]/i;         print "$&\n"; # 1
$t =~ /(?i)[a-z][a-z]/;      print "$&\n"; # 2
$t =~ /((?i)[a-z])[a-z]/;    print "$&\n"; # 3
$t =~ /[a-z](?i)[a-z]/;      print "$&\n"; # 4
$t =~ /(?i)[a-z](?-i)[a-z]/; print "$&\n"; # 5
```

```
AB
AB
Bc
cd
Bc
```

Test 2 ist äquivalent zu Test 1. In Test 3 ist wegen der Klammerung nur für den ersten Buchstaben Großschrift erlaubt. Da in Test 4 der erste Buchstabe klein geschrieben sein muß, kommt nur „`cd`“ als Lösung infrage. Die letzte Zeile zeigt wie man eine Option wieder abschalten kann (entspricht Test 3).

17.3 Spezielle Variablen

In den vorherigen Abschnitten wurden schon einige spezielle Perl-Variablen im Zusammenhang mit der Mustersuche erwähnt; hier eine Übersicht:

- `$&` gibt das gefundene Muster zurück
- `$1,$2,$3,...` enthält das Muster der 1.,2.,3.,... runden Klammer
- `$+` enthält das Muster der letzten runden Klammer
- `` $` `` enthält die Zeichenkette, die vor dem gefundenen Muster steht
- `$'` enthält die Zeichenkette, die hinter dem gefundenen Muster steht

Beispiel:

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

use strict;

my $s = "Dies ist ein Test ...";

if($s =~ /is([a-z]) (.*) Test/) {
  print $&."\n";
  print $1."\n";
  print $2."\n";
  print $+."\n";
  print $`."\n";
  print $'."\n";
}
```

```
ist ein Test
t
ein
ein
Dies
 ...
```

17.4 Suchen und Ersetzen

Anstelle einer einfachen Suche kann man in Perl auch den gefundenen Ausdruck direkt durch einen neuen ersetzen. Allgemein:

`$string =~ s/`*Regexp*`/`*Ersatz*`/`*Optionen*`;`

Dabei wird wie beim Suchoperator („`m/.../`“) `$string` nach einer zu *Regexp* passenden Zeichenkette durchsucht. Wird ein solcher Teilstring gefunden, so wird an dessen Stelle der Text *Ersatz* gesetzt.

Beispiel:

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

use strict;

my $t = "Beispiel";
$t =~ s/e/-e-/;
print "$t\n";
```

```
B-e-ispiel
```

Im obigen Beispiel wird der String `$t` nach dem Buchstaben „`e`“ abgesucht und (beim ersten Auffinden) durch die Zeichenkette „`-e-`“ ersetzt.

Auch beim Suchen und Ersetzen können die Schrägstriche des Operators durch andere Zeichen ersetzt werden (keine alphanumerischen Symbole oder Leerzeichen).

Es können die gleichen Optionen wie beim Suchoperator verwendet werden, ihre Wirkung bezieht sich dabei auf den regulären Ausdruck. Eine zusätzliche Option ermöglicht die Auswertung des *Ersatz*-Teils.

Optionen:

- `g` („*global*“)
  (siehe Optionen beim Suchoperator)
- `i` („*case-insensitive*“)
  (siehe Optionen beim Suchoperator)
- `s` („*single line*“)
  (siehe Optionen beim Suchoperator)
- `m` („*multiple lines*“)
  (siehe Optionen beim Suchoperator)

- `o` („*compile once*“)

  (siehe Optionen beim Suchoperator)

- `x` („*extended*“)

  (siehe Optionen beim Suchoperator)

- `e` („*evaluate*“)

  Diese Option bewirkt, daß beim Ersetzen der einzusetzende Ausdruck wie ein Befehl in Perl behandelt und ausgewertet wird. Dies kann sogar mehrfach geschehen.

  Beispiel:

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

use strict;

my $t = "5+7";

$t =~ s|^(\d+)\+(\d+)$|$1+$2|e;     # hier '|' statt '/'
print "$t\n\n";

$t = '_x_';

(my $s = $t) =~ s/x+/'"m" x 3' x 1/;
print "$s\n";

($s = $t) =~ s/x+/'"m" x 3' x 1/e;
print "$s\n";

($s = $t) =~ s/x+/'"m" x 3' x 1/ee;
print "$s\n";
```

```
12

_'"m" x 3' x 1_
_"m" x 3_
_mmm_
```

Kapitel 18

Unterprogramme

18.1 Einführung

Unterprogramme (auch als „Funktionen“ oder „Subroutinen“ bezeichnet) dienen dazu, Programmteile, die an mehreren Stellen (nahezu) identisch verwendet werden, durch einen einzelnen Aufruf zu ersetzen, was oft die Übersichtlichkeit erheblich verbessert. In Perl werden sie durch das Schlüsselwort `sub` eingeleitet. Der Aufruf eines Unterprogramms erfolgt durch das Voranstellen eines „`&`“ vor den Namen. Werden Klammern für Parameter (eventuell leer) verwendet, kann auf das „`&`“ verzichtet werden.

Beispiel:

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

use strict;

print "Hier ist das Hauptprogramm.\n";
&UNTER;
UNTER();

sub UNTER {
  print "Hier ist das Unterprogramm.\n";
}
```

```
Hier ist das Hauptprogramm.
Hier ist das Unterprogramm.
Hier ist das Unterprogramm.
```

Steht der Aufruf des Unterprogramms weiter hinten im Skript als die Definition oder ein eventuell vorhandener Prototyp, kann auch der Name alleine benutzt werden. Ein Prototyp besteht lediglich aus dem Schlüsselwort `sub` gefolgt vom Namen der Subroutine; die Definition wird sozusagen später nachgeliefert.

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

use strict;

sub UNTER;                        # Prototyp

print "Hauptprogramm\n";
UNTER;                            # Name allein genügt

sub UNTER {                       # Definition
  print "Unterprogramm\n";
}
```

Die Definition des Unterprogramms kann (im Gegensatz zu manch anderen Programmiersprachen) an einer beliebigen Stelle im Quelltext stehen.

18.2 Lokale Variablen

In einem Unterprogramm kann beliebig auf die (globalen) Variablen des Hauptprogramms zugegriffen werden. Um Variablen lokal zu definieren gibt es die Operatoren `local` und `my`. Sie unterscheiden sich in bezug auf Unterprogramme darin, daß Variablen, die in `my` deklariert werden, nur dort im Unterprogramm definiert sind (genauer: innnerhalb des Blockes, in dem sie deklariert werden). Auf „`local`“-Variablen dagegen kann auch in weiteren Unterprogrammen zugegriffen werden, deren Aufruf innerhalb des Unterprogramms erfolgt, in dem die Variablen deklariert werden.

Beispiel (der Übersichtlichkeit halber wird hier auf `use strict` verzichtet):

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

$gl = 1;
$loc = 2;
$my = 3;

print "main: \$gl = $gl, \$loc = $loc, \$my = $my\n";
sub1();
print "main: \$gl = $gl, \$loc = $loc, \$my = $my\n";

sub sub1 {
    local $loc = 7;
    my $my = 8;

    print "sub1: \$gl = $gl, \$loc = $loc, \$my = $my\n";
    sub2();
}

sub sub2 {
    print "sub2: \$gl = $gl, \$loc = $loc, \$my = $my\n";
}
```

```
main: $gl = 1, $loc = 2, $my = 3
sub1: $gl = 1, $loc = 7, $my = 8
sub2: $gl = 1, $loc = 7, $my = 3
main: $gl = 1, $loc = 2, $my = 3
```

Wie man sieht, ist `$gl` überall unverändert verfügbar, während `$loc` in `sub1` durch eine neue Variable gleichen Namens ersetzt wird. Deren Wirkung erstreckt sich aber bis in das zweite Unterprogramm. Dagegen ist die Zuweisung `$my = 8` nur in `sub1` von Bedeutung. An der letzten Ausgabezeile erkennt man, daß die Werte der globalen Variablen von den lokalen Variablen gleichen Namens nicht beeinflußt werden.

Eine genauere Beschreibung der Wirkungsweise der Operatoren `my()` und `local()` findet sich unter Variablen und Symboltabellen.

18.3 Parameter

Die Übergabe von Parametern erfolgt durch das spezielle Array „`@_`“. Somit kann innerhalb des Unterprogramms auf die (im Prinzip beliebig vielen) Parameter über `$_[0]`, `$_[1]`, `$_[2]`,... zugegriffen werden.

Beispiel:

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

use strict;

&S(1,2);
&S("aaa","bbb","ccc","ddd");

sub S {
  my $i;

  for($i = 0;$i < @_;$i++) { print "$_[$i]\n" }
}
```

```
1
2
aaa
bbb
ccc
ddd
```

Zur Erinnerung: `@_` steht im obigen Beispiel im skalaren Kontext und gibt daher die Zahl der Elemente in dem Parameter-Array an.

18.4 Rückgabewerte

Um einen bestimmten Wert an das aufrufende Hauptprogramm zu liefern, kann die Funktion `return()` (Klammern optional) verwendet werden. Fehlt eine solche Angabe, so ist der Rückgabewert automatisch das Ergebnis der zuletzt ausgeführten Operation im Unterprogramm.

Beispiel:

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

use strict;

my $s = &S(7,19);
print $s."\n";
$s = &T;
print $s."\n";

sub S { $_[0] + $_[1] }
sub T { return 100; }
```

```
26
100
```

Kapitel 19

Ausführen von Perl-Code mittels eval

Normalerweise wird der auszuführende Code in eine Datei geschrieben, deren Inhalt dann durch Perl abgearbeitet wird. Will man aber einem Benutzer beispielsweise die Möglichkeit geben eigene Perl-Befehle einzugeben und ausführen zu lassen, so ergibt sich das Problem, daß diese Kommandos zum Zeitpunkt des Programmstarts noch nicht bekannt sind.

Die Funktion `eval()` erwartet als Argument Perl-Code, der entweder in Form einer Zeichenkette oder als Code-Block übergeben wird. Dieser Code wird dann zur Programmlaufzeit übersetzt und ausgeführt.

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

use strict;

my $v = 5;
print "\$v = $v\n";

eval '$v++';
print "\$v = $v\n";

eval { $v++ };
print "\$v = $v\n";
```

```
$v = 5
$v = 6
$v = 7
```

Die erste Variante, bei der ein String übergeben wird, erlaubt eine Fehlerbehandlung zur Laufzeit, ohne daß notwendigerweise das Programm beendet wird. Im

Fehlerfalle wird dazu die spezielle Variable `$@` gesetzt.

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

use strict;

my $v = 5;
print "\$v = $v\n";

eval '$v+';
if($@) { print "Fehler: $@" }
print "\$v = $v\n";
```

```
$v = 5
Fehler: syntax error at (eval 1) line 2, at EOF
$v = 5
```

Kapitel 20

Spezielle Variablen

20.1 Einführung

Es gibt in Perl einige reservierte Variablen, die Informationen über aktuelle Einstellungen geben und die auch dazu verwendet werden können, einige dieser Einstellungen zu verändern, indem die Variablen entsprechend gesetzt werden.

Verwendet man das Modul `English` der Standard-Bibliothek („`use English;`“), so stehen auch begriffliche Namen dieser Variablen zur Verfügung (sie sind in der folgenden Liste in Klammern angegeben).

Anmerkung: Einige der Variablen liefern auf Nicht-UNIX-Betriebssystemen nicht unbedingt sinnvolle Werte (wenn überhaupt), falls die jeweils zugrundeliegende Funktion (z.B. Verwendung von Benutzer-IDs) nicht zur Verfügung steht.

20.2 Die Variable $_

Wenn beispielsweise bei einem Funktionsaufruf der Rückgabewert nicht explizit einer Variablen zugewiesen wird, so wird er oft doch gespeichert, und zwar in der Variablen „`$_`“. Auch können eine Reihe von Funktionen (die eigentlich ein Argument erwarten) ohne Argument aufgerufen werden – es wird dann stillschweigend der Inhalt von `$_` übergeben. `$_` ist also so etwas wie eine Variable, die temporär Daten aufnimmt, und es daher erlaubt, Funktionsaufrufe miteinander zu verketten, ohne jedesmal explizit Zuweisungen an Variablen vornehmen zu müssen, die sonst nirgendwo im Programm gebraucht werden.

Beispiel:

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

use strict;

my $string = 'Hallo';
chop($string);
print $string;
print "\n";

$_ = 'Hallo';
chop($_);
print $_;
print "\n";

$_ = 'Hallo';
chop;
print;
print "\n";
```

Während im ersten Teil eine Variable namens `$string` verwendet wird, um den Text zu speichern und die anschließenden Funktionsaufrufe durchzuführen, zeigt das zweite Beispiel, daß man als Variable genausogut `$_` verwenden kann. Der dritte Teil demonstriert, wie bei Aufruf der Funktion `chop()` ohne Argument standardmäßig der Inhalt von `$_` übergeben wird und das Ergebnis wiederum in `$_` landet. Schließlich wird der Inhalt von `$_` mit der Funktion `print()` (ebenso ohne Argument) ausgegeben.

Auch wenn im obigen Beispiel `$_` wie eine normale Variable verwendet wird, gibt es ein doch ein paar Unterschiede; so ist `$_` eine „echte“ globale Variable, und sie läßt sich auch nicht mit `my()` in einer Subroutine lokalisieren.

Ein weiteres Beispiel:

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

use strict;

my @array = ('a', 'b', 'c');

foreach (@array) {
    tr/a-z/A-Z/;
    print;
}
```

```
ABC
```

Da hier bei `foreach` keine Laufvariable angegeben ist, wird `$_` angenommen. `$_` erhält also nacheinander die Werte „a“, „b“ und „c“. Im Inneren der Schleife wird die Funktion `tr` auf `$_` angewandt (ausführlich: `$_ =~ tr/a-z/A-Z/;`), die hier die Kleinbuchstaben von a bis z durch die entsprechenden Großbuchstaben ersetzt.

20.3 Allgemeine Informationen

- `$0` (`$PROGRAM_NAME`)

  Name des gerade laufenden Programms

- `$]` *(veraltet, siehe $^V ab Version 5.6!)*

  Version des Perl-Interpreters (z.B. `5.008002` in Perl 5.8.2)

- `$^V` (`$PERL_VERSION`)

  Version des Perl-Interpreters (Achtung: `$^V` setzt sich aus den Symbolen mit den ASCII-Code-Werten der Versionsnummer zusammen, z.B. *chr(5)chr(8)chr(2)* in Perl 5.8.2)

- `$^O` (`$OSNAME`)

  Name des Betriebssystems

- `$^X` (`$EXECUTABLE_NAME`)

  Name des Perl-Interpreters

- `$^W` (`$WARNING`)

  Zeigt an, ob Warnungen ausgegeben werden sollen (z.B. durch die Option „`-w`“).

- `$^T` (`$BASETIME`)

  Zeitpunkt, an dem das aktuelle Skript gestartet wurde (unter UNIX: Anzahl der Sekunden seit dem 1.1.1970)

20.4 PID,UID,GID

- `$$` (`$PROCESS_ID`)

  Prozeß-Nummer von Perl

- `$<` (`$REAL_USER_ID`,`$UID`)

  Benutzer-ID, unter der der Prozeß gestartet wurde

- `$>` (`$EFFECTIVE_USER_ID`,`$EUID`)

  Benutzer-ID, unter der der Prozeß aktuell läuft (verschieden von `$<` im Falle von „Setuid“-Programmen)

- `$(` (`$REAL_GROUP_ID`,`$GID`)

  Liste der Gruppen des Benutzers, unter dem der Prozeß gestartet wurde

- `$)` (`$EFFECTIVE_GROUP_ID`,`$EGID`)

  aktuelle Liste der Gruppen des Benutzers (verschieden von `$(` im Falle von „Setgid"-Programmen)

20.5 Fehlermeldungen

- `$?` (`$CHILD_ERROR`)

  Status-Nummer des letzten Systemaufrufs

- `$!` (`$OS_ERROR`,`$ERRNO`)

  Fehlerummer oder -meldung (je nach Kontext) des letzten Systemaufrufs

- `$@` (`$EVAL_ERROR`)

  Fehlermeldung des letzten `eval()`-Aufrufs

20.6 Weitere spezielle Variablen

- `@_`

  Dieses Array enthält die an ein Unterprogramm übergebenen Argumente.

- `$[`

  Index des ersten Elements eines Arrays (Standardwert: 0)

  Diese Variable sollte (insbesondere ab Perl Version 5) nicht mehr verwendet werden, da dies bei Programmen, die aus mehreren Dateien (Modulen) bestehen, zu Problemen führen kann.

Kapitel 21

Referenzen

21.1 Harte Referenzen

Sogenannte „harte“ Referenzen sind skalare Größen, die auf eine beliebige Variable „zeigen“. Das Ziel kann dabei eine einfache skalare Variable sein, aber auch ein Array oder Hash oder auch ein Unterprogramm. Auch kann man eine Referenz auf eine Referenz erstellen.

Beispiele:

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

use strict;

my $nr = 42;
my $ref_nr = \$nr;

my @planeten = ( 'Erde', 'Mars', 'Jupiter' );
my $ref_planeten = \@planeten;

my %tiere = ( 'Tiger' => 'Indien', 'Löwe' => 'Afrika' );
my $ref_hash = \%tiere;
```

Wie man sieht, wird eine Referenz einfach dadurch erzeugt, daß vor die zu referenzierende Variable ein Backslash („\“) gesetzt wird. Unabhängig davon, von welchem Typ das referenzierte Objekt ist, ist die Referenz immer ein Skalar – sie besitzt also die Form $*name*.

Um von einer Referenz wieder zurück zur referenzierten Variablen zu kommen („Dereferenzierung“), stellt man der Referenz das Symbol voran, das dem Typ des referenzierten Objekts entspricht.

Beispiele:

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

use strict;

my $nr = 42;
my $ref_nr = \$nr;
my $deref_nr = $$ref_nr;

my @planeten = ( 'Erde', 'Mars', 'Jupiter' );
my $ref_planeten = \@planeten;
my @deref_planeten = @$ref_planeten;

my %tiere = ( 'Tiger' => 'Indien', 'Löwe' => 'Afrika' );
my $ref_tiere = \%tiere;
my %deref_tiere = %$ref_tiere;
```

Hier muß man also den jeweiligen Datentyp beachten!

(De-)Referenzierung kann auch beliebig mehrstufig angewandt werden:

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

use strict;

my $stern = 'Sirius';

my $ref_ref_ref_stern = \\\$stern;

$stern = $$$$ref_ref_ref_stern;
```

Auch Elemente eines Arrays lassen sich (de-)referenzieren:

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

use strict;

my @zahlen = ( 'null', 'eins', 'zwei', 'drei' );

my $ref_zahlen = \@zahlen;
print $$ref_zahlen[2]."\n";

my $ref_zwei = \$zahlen[2];
print $$ref_zwei."\n";
```

Im ersten Teil wird hier eine Referenz auf das Array als Ganzes erstellt. Sie wird dann dereferenziert (`$$ref_zahlen`) und von dem Ergebnis (dem Array)

das Element mit dem Index 2 ausgewählt. Im zweiten Teil wird dagegen eine Referenz nur eines einzelnen Elements des Arrays erzeugt und wieder dereferenziert.

21.2 „Call by Reference“

Bei dem Aufruf von Unterprogrammen unterscheidet man im wesentlichen zwei Typen, was die Behandlung der Argumente betrifft: „call by value“ und „call by reference“.

Ersteres beschreibt den Mechanismus, den Perl für gewöhnliche Variablen benutzt, bei dem die Werte (*values*) der Argumente an das Unterprogramm übergeben werden. Dies bedeutet, daß die lokalen Variablen, denen die Argumente zugewiesen werden, völlig unabhängig von den Variablen existieren, die die Argumente beim Aufruf bestimmen.

Manchmal möchte man aber eine Variable in einem Unterprogramm manipulieren, d.h., etwaige Operationen, die im Rumpf der Subroutine durchgeführt werden, sollen sich auch auf die Variablen im aufrufenden Programmteil auswirken. Eine Möglichkeit besteht darin, die manipulierten Variablen als Rückgabewerte an den aufrufenden Programmteil zu übergeben.

Eleganter und effektiver geht es aber mit Referenzen („*call by reference*“). Übergibt man eine Referenz als Argument, so zeigt die (lokale) Referenz auf die Variable im Argument und kann sie direkt bearbeiten.

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

use strict;

sub gross_1 {        ### einfacher "call by value"
    my $s = $_[0];
    $s = ucfirst($s);
}

sub gross_2 {        ### "call by value" mit Rückgabewert
    my $s = $_[0];
    $s = ucfirst($s);
    return($s);
}

sub gross_3 {        ### Übergabe einer Referenz
    my $ref_s = $_[0];
    $$ref_s = ucfirst($$ref_s);
}

sub gross_4 {        ### Sonderfall @_
    $_[0] = ucfirst($_[0]);
}

print "1) ".(my $wort = 'kamel')." -> ";
gross_1($wort);
print "$wort\n";

print "2) ".($wort = 'kamel')." -> ";
$wort = gross_2($wort);
print "$wort\n";

print "3) ".($wort = 'kamel')." -> ";
gross_3(\$wort);
print "$wort\n";

print "4) ".($wort = 'kamel')." -> ";
gross_4($wort);
print "$wort\n";
```

```
1) kamel -> kamel
2) kamel -> Kamel
3) kamel -> Kamel
4) kamel -> Kamel
```

Wie man sieht, wirkt sich im ersten Aufruf die Manipulation in „`gross_1`" nicht auf die globale Variable „`$wort`" aus. „`gross_2`" zeigt, wie sich das Problem über den Rückgabewert lösen läßt. In „`gross_3`" wird eine Referenz übergeben,

mit Hilfe derer die globale Variable verändert werden kann.

Das vierte Beispiel zeigt, daß das spezielle Array `@_`, das die Unterprogrammparameter enthält, nicht wirklich aus Kopien der übergebenen Variablen besteht. Verändert man die Elemente von `@_`, so wirkt sich dies auch auf die Variablen der Parameter im aufrufenden Programmteil aus. Dieser Trick sollte aber aus Gründen der Übersichtlichkeit nicht unbedingt genutzt werden – er stammt aus der Zeit, als es in Perl noch keine Referenzen gab.

21.3 Referenzen auf Unterprogramme

Auch Referenzen auf Subroutinen sind möglich:

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

use strict;

sub hallo { return "Hallo, $_[0] !" };

my $ref_sub = \&hallo;

print &$ref_sub('Welt')."\n";
```

```
Hallo, Welt !
```

Bei der Bildung der Referenz ist darauf zu achten, daß sie durch Voranstellen eines Backslashes vor den Namen des Unterprogramms mit dem „`&`" erfolgt. Es dürfen dabei keine Klammern oder gar Argumente angehängt werden (in diesem Falle würde eine Referenz auf den Rückgabewert der Routine gebildet werden). Auch darf das „`&`" nicht weggelassen werden. Analog zur Dereferenzierung bei Variablen muß der Typ des Objektes durch Angabe des entsprechenden Symbols (hier: „`&`") vermerkt werden.

Will man ein Unterprogramm nur über eine Referenz aufrufen, bietet sich eine sogenannte anonyme Subroutine an. Sie besitzt keinen eigenen Namen und liefert bei der Definition eine Referenz auf den Programmcode zurück.

Beispiel:

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

use strict;

my $ref_sub = sub { return "Hallo, $_[0] !" };

print &$ref_sub('Welt')."\n";
```

Man beachte hierbei zwei Dinge: zum einen hat das Unterprogramm, wie schon erwähnt, keinen Namen. Daher folgt unmittelbar auf das Schlüsselwort „`sub`“ die Definition in geschweiften Klammern. Außerdem darf nicht übersehen werden, daß es sich hierbei um eine Zuweisung handelt, die durch ein Semikolon abgeschlossen werden muß (es sei denn, diese Zeile steht beispielsweise am Ende eines Blocks).

Referenzen können genauso wie andere skalare Größen als Rückgabewerte von Subroutinen auftreten; d.h., ein Unterprogramm kann eine Referenz auf ein anderes Unterprogramm liefern.

Beispiel:

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

use strict;

sub master {
    my $ref_sub;
    $ref_sub = sub { print "Hallo !\n" };
    return($ref_sub);
}

my $ref_hallo = master();

&$ref_hallo;
```

```
Hallo !
```

Ein besonderer Fall in diesem Zusammenhang sind sogenannte „Closures“. Dabei betrachtet man Unterprogramme mit lokalen Variablen (deklariert durch `my`). Die genannten lokalen Variablen werden bei der Definition und Zuweisung zu einer Referenz sozusagen mit eingeschlossen.

Beispiel:

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

use strict;

sub hallo {
    my $planet = $_[0];
    my $ref = sub { print "Hallo, $planet !\n" };
    return($ref);
}

my $ref_erde = hallo('Erde');
my $ref_mars = hallo('Mars');

&$ref_erde;
&$ref_mars;
```

```
Hallo, Erde !
Hallo, Mars !
```

Das Bemerkenswerte daran ist, daß zu dem Zeitpunkt, an dem die anonyme Subroutine ausgeführt wird (hier: am Ende des Skripts), eigentlich der Gültigkeitsbereich der lokalen Variablen `$planet` schon längst verlassen worden ist. Dennoch „erinnern“ sich noch beide Referenzen an den lokalen Wert, der zum Zeitpunkt ihrer Zuweisung gültig war.

Kapitel 22

Mehrdimensionale Arrays

22.1 Allgemeines

Ein Beispiel für ein zweidimensionales Array ist eine sogenannte Matrix. Dabei handelt es sich zunächst einfach um eine Anordnung von Zahlen (oder Variablen oder Termen oder...) in einem Feld, wobei jeder Eintrag durch zwei Kennziffern (*Indizes*) bestimmt ist.

Beispiel einer *2x3*-Matrix (2 Zeilen, 3 Spalten):

$$A = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

Um eine einzelne Variable zu beschreiben, setzt man Zeile und Spalte als Index an den Namen der Matrix, z.B.:

$$A_{1,2} = b$$

Will man nun ein Perl-Programm schreiben, das mit Matrizen arbeiten soll, muß man sich überlegen, wie man sie geeignet speichert. Da Arrays ein geordnetes **ein**dimensionales (**ein** Index) Schema darstellen, ließe sich eine Matrix wohl aus der Kombination von Arrays repräsentieren.

22.2 Wie es nicht geht

Die einfachste Idee scheint zu sein, die Zeilen jeweils in einzelnen Arrays zu speichern und dann diese Zeilen-Arrays in ein Matrix-Array einzusetzen:

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

use strict;

my @zeile_1 = ( 'a', 'b', 'c' );
my @zeile_2 = ( 'd', 'e', 'f' );

my @matrix = ( @zeile_1, @zeile_2 );  # 6-elementiges Array
```

Obiger Code erzeugt keineswegs eine zweidimensionale Matrix in `@matrix`. In der letzten Zeile werden nämlich zuerst die beiden Zeilen-Arrays `@zeile_1` und `@zeile_2` als Listen dargestellt, die dann vor der Zuweisung zu einer 6-elementigen Liste vereinigt werden. Die letzte Zeile des obigen Programms ist also äquivalent zu:

```
my @matrix = ( 'a', 'b', 'c', 'd', 'e', 'f' );
```

Die Ursache für dieses Verhalten liegt darin begründet, daß Arrays in Perl grundsätzlich nur skalare Größen enthalten, aber keine Arrays oder Hashes (sie werden, wie oben beschrieben, vorher umgewandelt).

22.3 Verwendung von Referenzen

Einen Ausweg aus dem Dilemma bieten Referenzen, da sie skalare Variablen sind, aber auf beliebige Datentypen (so auch Arrays) „zeigen“ können.

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

use strict;

my @zeile_1 = ( 'a', 'b', 'c' );
my @zeile_2 = ( 'd', 'e', 'f' );

my $ref_zeile_1 = \@zeile_1;
my $ref_zeile_2 = \@zeile_2;

my @matrix = ( $ref_zeile_1, $ref_zeile_2 );
```

Nun enthält das Array `@matrix` zwei (skalare) Elemente, die ihrerseits jeweils eine Referenz auf ein Zeilen-Array sind.

Wie kann man nun auf die einzelnen Matrixelemente zugreifen? Die Elemente `$matrix[0]` und `$matrix[1]` enthalten jeweils eine Referenz auf ein Array, so daß nach der Dereferenzierung die Zeilen-Arrays zur Verfügung stehen.

An dieser Stelle sei noch einmal darauf hingewiesen, daß in Perl Array-Indizes üblicherweise bei 0 anfangen. Man sollte nicht die Variable `$[` auf 1 setzen, um

bei 1 mit der Zählung zu beginnen, sondern besser die Elemente oder Zeilen-Arrays gezielt an die Positionen 1,2,... der jeweiligen Arrays schreiben. Der Einfachheit halber wird in den Beispielen hier darauf verzichtet, so daß zu beachten ist, daß von den Matrix-Indizes immer jeweils 1 zu subtrahieren ist, um die Array-Indizes zu erhalten.

Ein ausführliches Beispiel sieht dann so aus:

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

use strict;

my @zeile_1 = ( 'a', 'b', 'c' );
my @zeile_2 = ( 'd', 'e', 'f' );

my $ref_zeile_1 = \@zeile_1;
my $ref_zeile_2 = \@zeile_2;

my @matrix = ( $ref_zeile_1, $ref_zeile_2 );

my $ref_1 = $matrix[0];
@zeile_1 = @$ref_1;

my $ref_2 = $matrix[1];
@zeile_2 = @$ref_2;

print "1) @zeile_1\n";
print "2) @zeile_2\n";
```

```
1) a b c
2) d e f
```

Der Zugriff läßt sich natürlich auch kompakter programmieren:

```
my @matrix = ( $ref_zeile_1, $ref_zeile_2 );

print "1) @{$matrix[0]}\n";
print "2) @{$matrix[1]}\n";
```

Wegen der Präzedenzregeln müssen bei den Dereferenzierungen hier geschweifte Klammern gesetzt werden (ansonsten würde Perl zuerst versuchen, `$matrix` zu dereferenzieren und erst dann dort das Element mit dem entsprechenden Index suchen).

Mit Hilfe dieses Mechanismus lassen sich auch gezielt einzelne Matrixelemente auslesen oder mit Werten besetzen:

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

use strict;

my @zeile_1 = ( 'a', 'b', 'c' );
my @zeile_2 = ( 'd', 'e', 'f' );
my $ref_zeile_1 = \@zeile_1;
my $ref_zeile_2 = \@zeile_2;
my @matrix = ( $ref_zeile_1, $ref_zeile_2 );

print "Matrix(1,2) = ${$matrix[0]}[1]\n";

${$matrix[1]}[2] = 'x';
print "1) @{$matrix[0]}\n";
print "2) @{$matrix[1]}\n";
```

```
Matrix(1,2) = b
1) a b c
2) d e x
```

Eine alternative Schreibweise für den Zugriff auf ein einzelnes Element bietet der Pfeil-Operator „->" (nicht zu verwechseln mit „=>" als Kommaersatz):

```
print "Matrix(1,2) = $matrix[0]->[1]\n";

$matrix[1]->[2] = 'x';
```

Und auch dies läßt sich noch verkürzen, da Perl zwischen zwei aufeinander folgenden Klammern (eckig oder geschweift) automatisch einen Pfeil-Operator setzt. Dadurch läßt sich eine sehr intuitive und übersichtliche Schreibweise erreichen:

```
print "Matrix(1,2) = $matrix[0][1]\n";

$matrix[1][2] = 'x';
```

22.4 Anonyme Arrays

Wie der Name schon vermuten läßt, handelt es sich dabei um Arrays, die keinen eigenen Variablennamen besitzen. Die einzige Möglichkeit, auf den Inhalt zuzugreifen, besteht in einer Referenz auf dieses Array. Erzeugen kann man ein solches Array, indem man bei einer Liste eckige anstelle von runden Klammern verwendet. Der Rückgabewert ist dann eine Referenz auf diese Liste.

Beispiel:

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

use strict;

my @array     = ( 10, 20, 30, 40 );     # normales Array
my $ref_array = [ 10, 20, 30, 40 ];     # anonymes Array
```

Da ein solches anonymes Array eine Referenz liefert, kann man daraus direkt mehrdimensionale Felder erstellen. Die weiter oben als Beispiel verwendete Matrix ließe sich dann auch so erzeugen:

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

use strict;

my @matrix = ( [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ] );
```

Noch einmal zur Erinnerung: würde man hier runde statt eckige Klammern verwenden, erhielte man ein einfaches 6-elementiges Array in `@matrix`.

Auf die Matrixeinträge kann hier genauso zugegriffen werden wie weiter oben bei den benannten Arrays beschrieben.

Ein Beispiel, wie man die gesamte Matrix ausgeben kann:

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

use strict;

my @matrix = ( [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ] );

foreach my $ref_zeile (@matrix) {
    foreach my $spalte (@$ref_zeile) { print "$spalte " }
    print "\n";
}
```

```
a b c
d e f
```

Da eine solche zweidimensionale Datenstruktur letztlich auf (eindimensionalen) Arrays beruht, kann man sie mit Hilfe bekannter Funktionen wie `push()` oder `pop()` dynamisch verändern. So läßt sich die oben definierte Matrix beliebig bearbeiten:

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

use strict;

my @matrix = ( [ 'a', 'b', 'c' ],
               [ 'd', 'e', 'f' ] );

pop( @{$matrix[1]} );                # 'f' entfernen
unshift( @{$matrix[0]}, 'M' );       # 'M' einfügen

push(@matrix, [ 'g', 'h', 'i' ] );   # neue 3.Zeile

foreach my $ref_zeile (@matrix) {
    foreach my $spalte (@$ref_zeile) { print "$spalte " }
    print "\n";
}
```

```
M a b c
d e
g h i
```

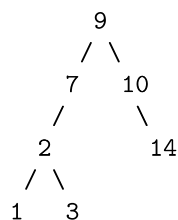
Kapitel 23

Binäre Bäume

23.1 Einführung

Als Beispiel einer Anwendung komplexer Datenstrukturen soll hier ein sogenannter binärer Suchbaum dienen, der zum Sortieren von Daten benutzt wird. Dies kann dann sinnvoll sein, wenn so große Datensätze verwaltet und sortiert werden müssen, daß sie nicht mehr im Hauptspeicher des Rechners bearbeitet werden können.

Ein sortierter binärer Baum ist durch miteinander verknüpfte Datensätze (oft „Knoten“ genannt) gekennzeichnet. Jeder dieser Knoten enthält im folgenden Beispiel eine Zahl sowie Referenzen auf bis zu zwei „Nachfolger“. Hier soll der sogenannte „linke“ eine Referenz auf einen Knoten sein, der eine kleinere Zahl beinhaltet, und der „rechte“ eine größere Zahl (wir gehen hier der Einfachheit halber davon aus, daß keine Zahl mehrmals auftritt).



```
        9
       / \
      7   10
     /      \
    2        14
   / \
  1   3
```

Ein solcher Baum läßt sich schrittweise von der „Wurzel“ aus (hier: Nummer **9**) aufbauen. Wie man sieht, befinden sich in den Teilbäumen, die an den Knoten jeweils links hängen, nur kleinere Zahlen als im betreffenden Knoten und im jeweils rechten Teilbaum nur größere. Diejenigen Knoten, die keine Nachfolger mehr besitzen (hier: **1**, **3** und **14**) werden „Blätter“ genannt.

23.2 Implementierung in Perl

Als erstes definieren wir eine geeignete Datenstrukur für die einzelnen Knoten des Baumes. Hier bietet sich die Verwendung eines Hash an. Da der Zugriff schließlich über Referenzen erfolgt, benutzt man am besten gleich einen anonymen Hash (man beachte die geschweiften Klammern anstelle von runden):

```
$ref_hash = { 'zahl'   => $zahl,
              'links'  => $ref_links,
              'rechts' => $ref_rechts  };
```

Im fertigen Baum wird dann jeder einzelne Knoten durch eine Referenz auf einen solchen Hash dargestellt, wobei `$zahl` jeweils die (zu sortierende) Zahl enthält und `$ref_links` sowie `$ref_rechts` auf die beiden Nachfolger zeigen. Ist nur ein oder gar kein Nachfolger (wie in einem Blatt) vorhanden, so seien die entsprechenden Referenzen gleich „`undef`“.

Um einen solchen Knoten mit einer entsprechenden Zahl zum Leben zu erwecken, definieren wir folgendes Unterprogramm:

```
sub knoten {
    return ( { 'zahl'   => $_[0],
               'links'  => undef,
               'rechts' => undef  } );
}
```

D.h., der Aufruf `knoten(9)` liefert eine Referenz auf einen Hash, dessen Schlüssel `zahl` den Wert 9 hat und dessen Werte von `links` und `rechts` nicht definiert sind.

Damit kann man schon die Wurzel des Baumes erzeugen:

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

use strict;

sub knoten {
    return ( { 'zahl'   => $_[0],
               'links'  => undef,
               'rechts' => undef  } );
}

my $ref_wurzel = knoten(9);
```

Um aus den restlichen Elementen einer zu sortierenden Liste den Suchbaum aufzubauen, beginnt man jeweils bei der Wurzel und vergleicht die neue Zahl mit der Zahl in der Wurzel. Ist das neue Element kleiner, so geht man zum linken Nachfolger und wiederholt den Vergleich, ansonsten geht man zum rechten

Nachfolger. Dieses Verfahren wird so lange wiederholt, bis man auf eine undefinierte Referenz stößt. Dort wird dann das neue Element als Blatt an den Baum gehängt, indem in dem letzten Knoten die entsprechende Referenz `links` oder `rechts` mit Hilfe des `knoten()`-Unterprogramms definiert wird.

In Perl ließe sich dieser Algorithmus zum Beispiel wie folgt programmieren:

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

use strict;

my @liste = ( 9, 10, 7, 14, 2, 3, 1 );

sub knoten {
    return ( { 'zahl'   => $_[0],
               'links'  => undef,
               'rechts' => undef  } );
}

sub erstelle_baum {
    my $ref_liste = shift;

    my $zahl = shift(@$ref_liste);    ### Wurzel erstellen
    my $ref_wurzel = knoten($zahl);

    foreach $zahl (@$ref_liste) {
        my $ref = $ref_wurzel;      ### beginne bei Wurzel

        while(1) {                 ### (Endlosschleife)
            ### Vergleich der Zahlen
            if($zahl < $$ref{'zahl'}) {
                if(defined($$ref{'links'})) {
                    ### gehe nach links
                    $ref = $$ref{'links'};
                }
                else {    ### neues Blatt
                    $$ref{'links'} = knoten($zahl);
                    last;   ### verlasse while(1) { }
                }
            }
            else {
                if(defined($$ref{'rechts'})) {
                    ### gehe nach rechts
                    $ref = $$ref{'rechts'};
                }
                else {    ### neues Blatt
                    $$ref{'rechts'} = knoten($zahl);
                    last;   ### verlasse while(1) { }
                }
            }
        }
    }
    return($ref_wurzel);
}

my $ref_wurzel = erstelle_baum(\@liste);  ### Hauptprogramm
```

Durch den Rückgabewert der Subroutine (eine Referenz auf die Wurzel) kann später auf den Suchbaum zugegriffen werden.

Nun benötigen wir natürlich noch Programmcode, mit dessen Hilfe die im Suchbaum sortierten Daten ausgegeben werden können. Dabei geht man am besten nach folgender Methode vor: Beginnend bei der Wurzel betrachte man einen (aktuellen) Knoten. Besitzt dieser Knoten einen linken Teilbaum, mache den linken Nachfolger zum aktuellen Knoten und wiederhole die Abfrage. Falls kein linker Nachfolger vorhanden ist (`links` ist `undef`), gebe die Zahl des aktuellen Knotens aus. Anschließend teste, ob ein rechter Teilbaum vorhanden ist. Falls ja, wird der rechte Nachfolger zum aktuellen Knoten, sonst geht man zum letzten (darüberliegenden) Knoten zurück. Wie man zeigen kann, wird mit Hilfe dieses Verfahrens der gesamte Suchbaum durchlaufen und alle Elemente werden dabei (sortiert) ausgegeben.

Implementierung in Perl:

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

use strict;

my @liste = ( 9, 10, 7, 14, 2, 3, 1 );

sub knoten {
    return ( { 'zahl'   => $_[0],
               'links'  => undef,
               'rechts' => undef  } );
}

## 'sub erstelle_baum { ... }' wie im letzten Beispiel

my $ref_wurzel = erstelle_baum(\@liste);

sub ausgabe {
    ### 'my' hier wichtig wegen Rekursion !
    my $ref = shift;

    ### suche links (rekursiv)
    if(defined($$ref{'links'})) {
        ausgabe($$ref{'links'})
     }
    ### Ausgabe der Zahl
    print "$$ref{'zahl'}\n";
    ### suche rechts (rekursiv)
    if(defined($$ref{'rechts'})) {
        ausgabe($$ref{'rechts'})
    }
}

ausgabe($ref_wurzel);
```

```
1
2
3
7
9
10
14
```

Kapitel 24

Schwartz'sche Transformation

24.1 Einführung

Diese Seite behandelt ein äußerst elegantes Sortierverfahren: die „Schwartz'sche Transformation“ (nach Randal L. Schwartz[1]). Obwohl zunächst recht aufwendig aussehend, offenbahrt dieses Konstrukt ungeahnte Möglichkeiten, hat man es erst einmal verstanden.

Im Grunde genommen handelt es sich nur um einen von vielen Wegen Daten zu sortieren. Allerdings ist es was Flexibilität und Effizienz angeht, wohl allen anderen Lösungen überlegen. Ein ähnliches Verfahren, das auf demselben Prinzip basiert, ist unter der Bezeichnung "Decorate-Sort-Undecorate" (DSU) bekannt.

24.2 Sortieren

Zunächst ein paar Worte dazu, wie in Perl Daten sortiert werden. Um den internen Algorithmus braucht man sich normalerweise keine Gedanken machen. Für den Benutzer bzw. Programmierer reduziert sich die Sortierung auf den Vergleich zweier Datenelemente – ist nichts anderes angegeben, d.h., ruft man einfach „`sort @liste`“ auf, so erfolgt die Sortierung gemäß ASCII-Code in aufsteigender Reihenfolge.

Durch Angabe einer eigenen Subroutine läßt sich die Sortierung gezielt steuern. Die dort definierten Kommandos werden während des Sortierens mehrfach aufgerufen, wobei jedesmal zwei Elemente (dargestellt durch `$a` und `$b`) miteinander verglichen werden. Ist der Rückgabewert gleich 1, so gilt das erste Element (`$a`) als „größer“ im Sinne der gewünschten Sortierung. Bei -1 geht Perl davon aus, daß `$b` „größer“ ist, bei 0 wird „Gleichheit“ angenommen.

[1] http://www.stonehenge.com/merlyn/

Beispiele:

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

use strict;

my @liste = ( 41, 37, 10, 30, 127, 512, 111 );

print "@liste\n\n";

my @sort_1 = sort @liste;
print "1) @sort_1  (Standard)\n";

my @sort_2 = sort { $a cmp $b } @liste;
print "2) @sort_2  (ASCII - aufsteigend)\n";

my @sort_3 = sort { $a <=> $b } @liste;
print "3) @sort_3  (numerisch - aufsteigend)\n";

my @sort_4 = sort { $b <=> $a } @liste;
print "4) @sort_4  (numerisch - absteigend)\n";

my @sort_5 =
  sort { substr($a,1,1) <=> substr($b,1,1) } @liste;
print "5) @sort_5  (2.Ziffer - numer. - aufst.)\n";
```

```
41 37 10 30 127 512 111

1) 10 111 127 30 37 41 512  (Standard)
2) 10 111 127 30 37 41 512  (ASCII - aufsteigend)
3) 10 30 37 41 111 127 512  (numerisch - aufsteigend)
4) 512 127 111 41 37 30 10  (numerisch - absteigend)
5) 10 30 111 512 41 127 37  (2.Ziffer - numer. - aufst.)
```

Am letzten dieser Beispiele kann man schon ein Problem erkennen: jedesmal, wenn zwei Daten miteinander verglichen werden, muß die Funktion `substr()` ausgeführt werden. Dies kann bei großen Datensätzen dazu führen, daß die meiste Rechenzeit in Operationen auf einzelne Elemente verbraucht wird, denn jedes einzelne Datum wird i.a. während der Sortierung mehr als einmal zu einem Vergleich herangezogen (wer es genauer wissen will: bei n Daten im Mittel $n \log n$-mal).

24.3 Effektive Sortierung

Eine Lösung des oben beschriebenen Problems besteht darin, in einem ersten Schritt zunächst für jedes Element des Datensatzes die entsprechende Operation

(hier: `substr()`) durchzuführen, anschließend eine Sortierung dieser temporären Daten vorzunehmen und schließlich von diesen wieder zu den ursprünglichen Daten zurückzukehren.

Als Beispiel soll nun eine Datei dienen, aus deren Zeilen jeweils eine Zahl extrahiert werden muß, die dann als Suchkriterium dient. Der Inhalt einer solchen Datei sähe beispielhaft etwa so aus:

```
oexkwch<37>jy
yunq<100>zmwi
ikbkwe<545>bcljvbry
ojudnle<818>tgum
gpmlxp<972>lud
```

Erzeugen kann man sich derartige Daten mit diesem Programm:

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

use strict;

srand;

sub zufall {
    my $zeilen = shift;
    my $s;
    my @liste = ();

    for( my $i=0; $i<$zeilen; $i++ ) {
        $s = '';
        for( my $j=0; $j<rand(15); $j++ ) {
            $s .= chr(rand(26)+97);
        }
        $s .= '<'.int(rand(1000)).'>';
        for( my $j=0; $j<rand(15); $j++ ) {
            $s .= chr(rand(26)+97);
        }
        push(@liste,$s);
    }
    return(@liste);
}

my @liste = zufall(5);
```

Ein Ansatz zum Sortieren dieser Daten könnte so aussehen:

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

use strict;

srand;
my @liste = zufall(5);   # 'sub zufall{...}': siehe oben

sub by_number {
    my($x,$y);

    ($x) = ( $a =~ /<(\d+)>/ );
    ($y) = ( $b =~ /<(\d+)>/ );

    $x <=> $y;
}

my @ergebnis = sort by_number @liste;
```

Dabei wird allerdings viel Rechenzeit durch die vielfache Auswertung der regulären Ausdrücke in `by_number` verbraucht. Schneller geht es, wenn man aus jedem Datum ein zweielementiges (anonymes) Array konstruiert, dessen eines Element das Datum selbst und das andere das Ergebnis des regulären Ausdruckes (hier: die Zahl) ist.

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

use strict;

srand;
my @liste = zufall(5);   # 'sub zufall{...}': siehe oben

my @temp_1 = ();

foreach my $elem (@liste) {
    push(@temp_1,[ $elem, ( $elem =~ /<(\d+)>/ )[0] ]);
}

my @temp_2 = sort { $a->[1] <=> $b->[1] } @temp_1;

my @ergebnis = ();

foreach my $t (@temp_2) {
    push(@ergebnis, $t->[0]);
}
```

Im obigen Skript wird in der ersten `foreach`-Schleife ein Array namens `@temp_1`

aufgebaut, dessen Elemente jeweils Referenzen auf zweielementige anonyme Arrays sind. Diese zweielementigen Arrays enthalten unter dem Index 0 die ursprüngliche Zeile und unter dem Index 1 die extrahierte Zahl.

Beim `sort` sind nun die beiden zu vergleichenden Elemente in den Variablen `$a` und `$b` die Referenzen aus dem Array `@temp_1`. Auf die für die Sortierung benutzte Zahl (unter dem Index 1) wird dann durch `$a->[1]` bzw. `$b->[1]` zugegriffen. `<=>` sorgt dann wie gewohnt für die (numerische) Sortierung der beiden Zahlen.

Danach befindet sich in `@temp_2` wiederum eine Liste aus Referenzen auf zweielementige anonyme Arrays, allerdings nun nach den jeweiligen Zahlen sortiert.

In der abschließenden `foreach`-Schleife wird nun aus `@temp_2` jeweils das erste Element dereferenziert und in das Array `@ergebnis` gepackt. Dieses Array enthält dann die ursprünglichen Datenzeilen, nun aber gemäß der enthaltenen Zahlen sortiert.

24.4 Die Funktion `map()`

Mit Hilfe von `map()` kann man auf einfache Art und Weise eine Operation auf alle Elemente einer Liste anwenden. `map()` erwartet entweder einen Ausdruck oder einen Programmblock, der dann nacheinander für jedes Arrayelement, auf das über `$_` zugegriffen wird, aufgerufen wird.

Beispiel:

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

use strict;

my @liste = ( 1, 2, 3 );

@liste = map { $_ * $_ } @liste;
print join(",",@liste),"\n";

@liste = map sqrt($_), @liste;
print join(",",@liste),"\n";
```

```
1,4,9
1,2,3
```

Zunächst wird jedes Element mit sich selbst multipliziert und die Ergebnisliste wieder in `@liste` gespeichert. Anschließend wird mit Hilfe der Funktion `sqrt()` jeweils die Quadratwurzel gezogen, wodurch sich wieder die ursprünglichen Werte ergeben.

24.5 Die Transformation

Die Schwartz'sche Transformation nutzt die Möglichkeiten der `map`-Funktion aus, um die ganze Sortierung in einer Befehlszeile unterzubringen und ohne explizit temporäre Arrays zu verwenden.

Damit reduziert sich das letzte Sortierbeispiel auf diesen Code:

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

use strict;

srand;
my @liste = zufall(5);   # 'sub zufall{...}': siehe oben

my @ergebnis = map { $_->[0] }
               sort { $a->[1] <=> $b->[1] }
               map { [ $_, ( /<(\d+)>/ )[0] ] } @liste;
```

Man beachte dabei, daß die letzten drei Zeilen des Skriptes nur ein Perl-Kommando darstellen, das sozusagen von hinten gelesen werden muß: zuerst wird aus `@liste` ein Array aus Referenzen auf zweielementige Arrays erstellt. Dieses Array ist dann das Argument von `sort` und aus dessen Rückgabewert wiederum wird das jeweils erste Element (mit dem Index 0) extrahiert. Das dabei entstehende Array wird schließlich `@ergebnis` zugewiesen.

24.6 Geschwindigkeitsvergleich

Wie schon weiter oben erwähnt war das Hauptziel der Schwartz'schen Transformation eine Steigerung der Effizienz. Dies läßt sich mit Hilfe des Benchmark-Moduls, das der Perl-Distribution beiliegt, recht einfach überprüfen.

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

use strict;
use Benchmark;

my $z = 1000;         # Länge der zu sortierenden Liste
my $c = 50;           # Anzahl der Durchläufe

srand;
my @liste = zufall($z);   # 'sub zufall{...}': siehe oben

### Einfache Sortierung

sub by_number {
    my($x,$y);

    ($x) = ( $a =~ /<(\d+)>/ );
    ($y) = ( $b =~ /<(\d+)>/ );

    $x <=> $y;
}

timethese($c, {
  "Einfache Sortierung" => sub {
        my @sorted = sort by_number @liste
    }
});

### Schwartz'sche Transformation

timethese($c, {
  "Schwartz'sche Trafo" => sub {
        my @sorted = map { $_->[0] }
        sort { $a->[1] <=> $b->[1] }
        map { [$_, ( /<(\d+)>/ )[0] ]} @liste
    }
});

print "($c Sortierungen von $z-elementigen Listen)\n";
```

```
Benchmark: timing 50 iterations of Einfache Sortierung...
Einfache Sortierung: 15 wallclock secs (14.07 usr +  0.04
 sys = 14.11 CPU) @  3.54/s (n=50)
Benchmark: timing 50 iterations of Schwartz'sche Trafo...
Schwartz'sche Trafo:  3 wallclock secs ( 3.07 usr +  0.04
 sys =  3.11 CPU) @ 16.08/s (n=50)
(50 Sortierungen von 1000-elementigen Listen)
```

Wie man sieht, benötigt die Schwartz'sche Transformation in diesem Falle nur etwa ein Fünftel der Rechenzeit im Vergleich zum einfachen „`sort by_number`".

Kapitel 25

Einbinden von Perl-Code

25.1 Ausführung von externem Code mit do()

Wird die Zahl der Subroutinen in einem Programm immer größer oder verwendet man dieselben (oder ähnliche) Unterprogramme immer wieder in verschiedenen Programmen, so sollte man den Programmcode auf mehrere Dateien verteilen. Es empfiehlt sich, das Hauptprogramm zusammen mit ein paar wichtigen Subroutinen in einer Datei zu speichern (die dann als Perl-Skript ausgeführt wird), und alle anderen Subroutinen (bzw. Klassen bei objekt-orientierter Programmierung) – in logischen Gruppen zusammengefaßt – in eigenen Dateien abzulegen.

Um nun Programmcode aus einer zusätzlichen Datei in ein Hauptprogramm einzubinden, gibt es in Perl verschiedene Möglichkeiten.

Die einfachste ist die Verwendung der Funktion „`do()`“. Hiermit wird Programmcode aus einer externen Datei so wie er dort abgelegt ist in das aktuelle Programm (also an die Stelle, wo `do` steht) eingebaut und ausgeführt (so als würde man `eval()` auf den Dateiinhalt anwenden).

Datei `prog.pl`:

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

use strict;

print "Erste Zeile des Hauptprogramms\n";
do "funkt.pl";
print "Ende des Hauptprogramms\n";
```

Datei `funkt.pl`:

```
print "Ausgabe aus 'funkt.pl'\n";
```

```
Erste Zeile des Hauptprogramms
Ausgabe aus 'funkt.pl'
Ende des Hauptprogramms
```

Wie man sieht, erwartet `do()` als Argument einen Dateinamen. Diese Datei wird dann in allen Verzeichnissen gesucht, die im Array `@INC` aufgelistet sind (siehe auch den Abschnitt Standardmodule). Befindet sich die gesuchte Datei in einem Verzeichnis, das nicht standardmäßig zu `@INC` gehört, so kann der Pfad (vor dem Aufruf von `do()`) beispielsweise durch

```
push(@INC,$verzeichnispfad);
```

hinzugefügt werden.

Ein Nachteil von `do()` ist die Tatsache, das jedesmal, wenn diese Funktion aufgerufen wird, die entsprechende Datei geöffnet, gelesen und geparst wird, weswegen sich der Einsatz z.B. innerhalb einer mehrfach durchlaufenen Schleife nicht gerade empfiehlt.

25.2 Code-Einbindung mit Hilfe von `require()`

Die Funktion `require()` bindet im Prinzip genauso wie `do()` Programmcode aus einer externen Datei ein, allerdings mit dem Unterschied, daß das Einlesen nur einmal geschieht und Perl beim nächsten Aufruf schon „weiß“, daß sich der entsprechende Code bereits im Speicher befindet und nicht wieder neu geladen werden muß.

Datei `prog.pl`:

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

use strict;

print "Erste Zeile des Hauptprogramms\n";
require "funkt.pl";
print "Ende des Hauptprogramms\n";
```

Datei `funkt.pl`:

```
print "Ausgabe aus 'funkt.pl'\n";
```

```
Erste Zeile des Hauptprogramms
Ausgabe aus 'funkt.pl'
Ende des Hauptprogramms
```

Ebenso wie `do()` durchsucht auch `require()` alle Verzeichnispfade in `@INC`.

Ein wesentlicher Unterschied zu `do()` besteht aber darin, daß der Code in der eingebundenen Datei bei der Ausführung einen „wahren“ Wert (*true*) zurückliefern muß. Im obigen Beispiel ist dies relativ offensichtlich, da der Aufruf „`print ...`“ einen wahren Wert („1“) zurückgibt (auch wenn er meist nie verwendet wird). Um sicherzugehen, daß externer Code wirklich zum Schluß *wahr* liefert, hat es sich eingebürgert, ans Ende der Datei eine kurze Zeile mit dem Inhalt „`1;`“ anzuhängen – sie bewirkt die Auswertung von „1“, was *wahr* entspricht und ist der letzte Aufruf der Datei und somit der Rückgabewert an `require()`.

Die Datei `funkt.pl` aus obigem Beispiel sollte also besser so aussehen:

```
print "Ausgabe aus 'funkt.pl'\n";

1;
```

Eine zweite Besonderheit von `require()`, die bei der Verwendung von sogenannten Modulen ausgenutzt wird, ist die Tatsache, daß falls als Argument ein Name angegeben wird, der nicht einen in Anführungsstrichen stehenden String darstellt, an diesen Namen automatisch die Endung *.pm* hinzugefügt wird. Somit sind die Aufrufe im folgenden Beispiel völlig äquivalent:

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

use strict;

my $dateiname = "extern.pm";
require $dateiname;
require 'extern.pm';
require extern;        # Automatische Ergänzung von .pm
```

Während die Dateiendung *.pl* üblicherweise für Skripten/Programme genutzt wird, verwendet man *.pm* für Dateien, die **P**erl-**M**odule enthalten.

25.3 Verwendung von `use`

Noch leistungsfähiger als `require()` ist `use`. Von der Funktion her entspricht ein `use`-Aufruf dem einen von `require()` gefolgt von `import()`. Letztere ist keine von Perl vordefinierte Funktion sondern eine Funktion, die in einem einzubindenden Modul definiert wird und üblicherweise dazu verwendet wird, um Funktionsnamen zu *import*ieren, damit sie dann genauso wie andere Funktionen im Hauptprogramm aufgerufen werden können.

Ein zweiter Unterschied besteht darin, daß `use` nicht (wie `require()`) an der entsprechenden Stelle zur Laufzeit des Hauptprogramms abgearbeitet wird, sondern schon bei der Kompilierung des Programms.

Datei `prog.pl`:

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

use strict;

print "Erste Zeile des Hauptprogramms\n";
use modul;     # eigentlich "modul.pm"
print "Ende des Hauptprogramms\n";
```

Datei `modul.pm`:

```
print "Ausgabe aus 'modul.pm'\n";

1;
```

```
Ausgabe aus 'modul.pm'
Erste Zeile des Hauptprogramms
Ende des Hauptprogramms
```

An der Reihenfolge der Ausgabe erkennt man schon, daß der Code der Datei *modul.pm* schon abgearbeitet wird, bevor das eigentliche Hauptprogramm beginnt.

Eine wichtige Konsequenz davon ist, daß man nun nicht mehr durch eine Zeile wie

```
push(@INC,$verzeichnispfad);
```

vor dem `use`-Aufruf im Hautprogramm einen zusätzlichen Suchpfad angeben kann, da `use` schon abgearbeitet wird, bevor überhaupt eine Zeile des Hauptprogramms zur Ausführung kommt. Zur Lösung dieses Problems kann man den entsprechenden `push`-Befehl in einen `BEGIN { ... }`-Block einbetten, dessen Inhalt bereits zur Kompilierungszeit aufgerufen wird.

Datei `prog.pl`:

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

use strict;

BEGIN {
  my $pfad = 'subdir';
  push(@INC,$pfad);
}
print "Erste Zeile des Hauptprogramms\n";
use modul;
print "Ende des Hauptprogramms\n";
```

Besser ist aber die Verwendung von `use lib`, um einen oder mehrere Pfade zu `@INC` hinzuzufügen:

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

use strict;

use lib ('subdir');

print "Erste Zeile des Hauptprogramms\n";
use modul;
print "Ende des Hauptprogramms\n";
```

Datei `modul.pm` im Unterverzeichnis `subdir` :

```
print "Ausgabe aus 'modul.pm'\n";

1;
```

Kapitel 26

Module

26.1 Packages

In einem Unterprogramm können mit Hilfe von `local()` oder `my()` lokale Variablen definiert werden, die dann nur einen entsprechend eingeschränkten Gültigkeitsbereich besitzen. Etwas ähnliches gilt auch für Variablen, die außerhalb von Subroutinen (im „Hauptprogramm“) verwendet werden: Sie sind eigentlich nicht wirklich globale Variablen, sondern gelten nur innerhalb eines sogenannten Packages. Wird kein solches Package deklariert, nimmt Perl das Standardpackage „main“ an.

Tritt bei der Programmausführung eine `package`-Anweisung auf, so gilt ab dort der Namensraum des entsprechenden Packages mit eigenen Variablen. Das heißt, jede Variable ist an ein bestimmtes Package gebunden. Wichtige Ausnahmen hiervon sind Variablen, die mit `my` deklariert werden, sowie spezielle Variablen wie `$_` oder `@ARGV`.

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

$v = 123;
print $v."\n";                   # (1)

### Ab hier gilt ein neuer Namensraum.
package p;
print $v."\n";                   # (2)
print $main::v."\n";             # (3)
print $::v."\n";                 # (4)
$w = 456;
print $w."\n";                   # (5)
print $p::w."\n";                # (6)

### Hier kehren wir wieder zu "main" zurück.
package main;
print $v."\n";                   # (7)
print $w."\n";                   # (8)
print $p::w."\n";                # (9)
```

Anmerkungen hierzu:

- **(1)** Gibt wie gewohnt den Inhalt von `$v` aus („123“).
- **(2)** Dies führt zu einer Fehlermeldung, da `$v` in „main“ angelegt wurde, und daher im Package „p“ nicht bekannt ist.
- **(3)** So kann über den Package-Namen auch in „p“ auf `$v` zugegriffen werden.
- **(4)** Für main-Variablen kann der explizite Package-Name auch weggelassen werden.
- **(5)** Gibt den Inhalt der Variablen `$w` aus (zum Package „p“ gehörend!).
- **(6)** Wie (5), nur mit Package-Namen
- **(7)** Da hier wieder der Namensraum von „main“ gilt, gibt es diesmal keine Fehlermeldung (vgl. (2)).
- **(8)** Da `$w` in „p“ angelegt wurde, darf sie nicht ohne weiteres in „main“ verwendet werden, d.h., hier erfolgt eine Fehlermeldung.
- **(9)** So kann in „main“ auf Variablen eines anderen Packages zugegriffen werden.

Die Zeilen (3), (4), (6) und (9) zeigen, wie man eine (skalare) Variable vollständig beschreibt:

`$`*Package-Name*`::`*Variablenname*

Man beachte, daß das Dollarzeichen dabei ganz vorne steht (und nicht etwa unmittelbar vor dem Variablennamen).

Wie schon weiter oben erwähnt, sind `my`-Variablen nicht an ein Package gebunden, daher wird bei folgendem Beispiel keine Fehlermeldung ausgegeben:

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

my $v = 123;
print $v."\n";

package p;
print $v."\n";
```

26.2 Module

In Perl wird ein Package, das in einer eigenen Datei abgelegt wird, und üblicherweise von anderen Programmen mittels `use` eingebunden wird, als Modul bezeichnet. Dabei ist der Dateiname gleich dem Package-Namen, ergänzt um die Endung „`.pm`“. Wie schon bereits beim Einbinden von Perl-Code beschrieben, muß der Rückgabewert beim Importieren von Code aus einem Modul *wahr* sein, daher die Zeile „`1;`“ am Ende.

Ein Modul in der Datei `hallo.pm` könnte beispielweise so aussehen:

```
package hallo;

sub sag_hallo {
    print "Hallo, Welt!\n";
}

1;
```

Es kann dann in einem Perl-Programm so verwendet werden:

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

use strict;
use hallo;

&hallo::sag_hallo();
```

```
Hallo, Welt!
```

Da die Subroutine `sag_hallo` in einem eigenen Package definiert wird (nämlich „hallo“), muß beim Aufruf des Unterprogramms auch der Package-Name mit angegeben werden.

26.3 Exportieren von Namen

Damit beim Einbinden eines Moduls nicht bei jedem Aufruf einer Funktion der Package-Name mit angegeben werden muß, kann man Namen (skalare Variablen, Arrays, Subroutinen,...) aus einem Modul heraus exportieren. Jene Variablen werden dann beim Aufruf von `use` ins Hauptprogramm importiert, d.h., sie können dann dort so verwendet werden, als wären sie im Hauptprogramm deklariert worden.

Dazu gibt es ein Modul namens „Exporter“, das diese Aufgabe übernimmt. Es definiert zum einen einige Arrays, die dem Exportieren von Namen dienen. Die beiden wichtigsten sind `@EXPORT` und `@EXPORT_OK`. Außerdem wird die bei der Beschreibung von `use` erwähnte Routine `import()` definiert. Da sich die Routine `import()` nicht im Modul selbst befindet (sondern in „Exporter“), muß man den Perl-Interpreter durch das `@ISA`-Array darauf hinweisen:

```
@ISA = ( 'Exporter' );
```

Genauer gesagt wird damit das aktuelle Modul wie eine von `Exporter` abgeleitete Klasse im Sinne objektorientierter Programmierung betrachtet, worauf hier aber nicht weiter eingegangen werden soll.

Alle Namen in `@EXPORT` werden automatisch in das aufrufende Programm exportiert, während diejenigen Namen, die in `@EXPORT_OK` stehen, nur auf Verlangen exportiert werden. Namen, die sich nicht in einem der beiden genannten Arrays befinden, können nur zusammen mit dem Package-Namen verwendet werden.

Beispiel eines Moduls in der Datei `modul.pm`:

```
package modul;

use Exporter;
@ISA = ('Exporter');

@EXPORT = ( 'routine_1' );
@EXPORT_OK = ( 'routine_2' );

sub routine_1 { print "Routine Nr.1\n" }
sub routine_2 { print "Routine Nr.2\n" }
sub routine_3 { print "Routine Nr.3\n" }

1;
```

So werden nur die Namen aus `@EXPORT` importiert, alle anderen benötigen den Package-Namen:

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

use strict;
use modul;

routine_1();
&modul::routine_2();
&modul::routine_3();
```

Hier werden die Namen aus `@EXPORT_OK` explizit importiert:

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

use strict;
use modul 'routine_2';

&modul::routine_1();
routine_2();
&modul::routine_3();
```

So importiert man beide exportierten Funktionen, `routine_3` kann allerdings nicht importiert werden:

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

use strict;
use modul 'routine_1','routine_2';

routine_1();
routine_2();
&modul::routine_3();
```

Weitere Informationen enthält die Manual Page von „Exporter“, die beispielsweise durch „`perldoc Exporter`“ ausgegeben wird.

26.4 Standardmodule

Bei der Perl-Distribution wird schon eine Reihe von Modulen mitgeliefert, wie beispielsweise das oben erwähnte Modul „Exporter“. Um herauszufinden, welche Module wo installiert sind, betrachte man das Array `@INC`, das eine Liste von Pfaden enthält, in denen Perl bei Bedarf nach Modulen sucht.

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

use strict;

foreach my $pfad ( @INC ) {
    print "$pfad\n";
}
```

Die Dokumentation eines Moduls erhält man (zumindest bei einem gut geschriebenen Modul) durch den Aufruf von „`perldoc`“ mit dem Namen der Moduldatei als Argument oder (unter MacOS) durch Öffnen der Moduldatei mit Hilfe des Programms „`Shuck`“.

Es lohnt sich auf jeden Fall, einen Blick in die Bibliothek der Standardmodule zu werfen, da sie viele Funktionen bereitstellen, die oft benötigt werden, wie das Lesen von Kommandozeilenoptionen, das Arbeiten mit Dateibäumen, das Kopieren von Dateien und vieles mehr.

Eine große Sammlung weiterer Module für die verschiedensten Aufgaben findet sich im *Comprehensive Perl Archive Network* (CPAN).

Kapitel 27

Variablen und Symboltabellen

27.1 Symboltabellen von Packages

Wie im Abschnitt Packages beschrieben, gibt es in Perl Namensräume (*Packages*), die jeweils ihre eigenen Variablen besitzen. Diese Variablen sind in sogenannten Symboltabellen gespeichert. Die Tabelle eines jeden Package ist in Form eines Hash abgelegt; um darauf zuzugreifen stellt man dem Package-Namen ein „`%`“ voran und hängt zwei Doppelpunkte „`::`“ an.

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

### Package "main"

$main_var1;
@main_var2 = (2,3);

package hallo;     ### Package "hallo"

$hallo_var;

package main;      ### wieder zurück in "main"

foreach $v (keys %hallo::) { print "hallo> $v\n" }
foreach $v (keys %main::) { print "main> $v\n" }
```

Die erzeugte Ausgabe ist recht umfangreich, da auch alle vordefinierten Variablen in der Symboltabelle von `main` enthalten sind. Man beachte, daß in `%main::` (man kann hier stattdessen auch die Kurzform `%::` verwenden) nur der jeweilige Name einer Variablen erscheint; ein Präfix (wie etwa „`$`“) wird weggelassen.

Hier ein Ausschnitt der Programmausgabe:

```
hallo> hallo_var
main> main_var1
main> main_var2
main> v
main> hallo::
main> 0
```

Die Symboltabelle des Packages `hallo` enthält nur die Variable `$hallo_var`, während sich in `main` neben den explizit verwendeten Variablen `$main_var1`, `@main_var2` und `$v` auch der Name der neuen Symboltabelle `%hallo::` findet. Außerdem stehen dort vordefinierte Variablen wie `$0`, welche den Namen des gerade ausgeführten Skripts beinhaltet.

27.2 Globale Variablen und `our()`

Wie im obigen Abschnitt schon gezeigt wurde, kann auf Variablen, die „einfach so“ deklariert oder verwendet werden, innerhalb des jeweiligen Packages direkt zugegriffen werden. Solche Variablen werden *global* genannt, da sie an jeder Stelle des Programms zur Verfügung stehen; selbst wenn sie in einem Unterprogramm deklariert werden.

Oft sind globale Variablen zwar praktisch, sie können ein Programm aber auch schnell unübersichtlich machen. Deswegen empfiehlt es sich, globale Variablen nur dort zu verwenden, wo sie notwendig sind. Damit man nicht versehentlich globale Variablen einführt, sollte man immer das Pragma `use strict` verwenden.

Benutzt man `use strict`, so muß man bei jeder Variable angeben, welchen Gültigkeitsbereich sie haben soll. Globale Variablen markiert man ab Perl 5.6.x mit der Deklaration `our()`.

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

use strict;

use 5.6.0;              ### "our" funktioniert
                        ### erst ab Perl 5.6.0

our $var = 123;

sub routine {
    our $vr = "Hallo";
    print "$vr\n";
}

routine();

print "$var\n";
print "$main::vr\n";
```

```
Hallo
123
Hallo
```

Auf eine mit `our` deklarierte Variable kann man innerhalb des umschließenden Blocks, der Datei oder dem Argument der Funktion `eval()` zugreifen, ohne einen Packagenamen angeben zu müssen. Im obigen Beispiel ist der Gültigkeitsbereich von `$var` die gesamte Datei, während der „einfache“ Zugriff auf `$vr` nur innerhalb des Unterprogramms `routine` (in diesem Falle der umschließende Block) möglich ist. Allerdings kann man mit Hilfe des Packagenamens auch außerhalb der Subroutine auf `$vr` zugreifen.

Eine `our`-Variable ist u.U. sogar über Package-Grenzen hinweg sichtbar, es kann jedoch jedes Package jeweils eigene Variablen mittels `our` deklarieren.

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

use strict;

use 5.6.0;              ### "our" funktioniert
                        ### erst ab Perl 5.6.0

our($v1,$v2);

$v1 = 1;
$v2 = 2;

package name;

print "$v1 $v2\n";

our $v2 = 9;

print "$v1 $v2 $main::v2\n";
```

```
1 2
1 9 2
```

Die Variable `$v1` ist in der ganzen Datei einfach über ihren Namen verfügbar, während es von `$v2` zwei Exemplare gibt. Nach der Deklaration im Package `name` kann auf die in `main` deklarierte Variable über den Package-Namen zugegriffen werden. Ohne Angabe des Packages wird die zuletzt deklarierte `our`-Variable verwendet (unabhängig vom aktuell gültigen Package).

Für ältere Perl-Versionen (< 5.6.0) deklariert man globale Variablen außerhalb von Blöcken mit Hilfe von `my()` (siehe folgenden Abschnitt).

27.3 Lexikalische Variablen mittels `my()`

Jede Variable, die mit `our` deklariert oder auch „einfach so“ ohne eine Deklaration verwendet wird, wird in die Symboltabelle des jeweils aktuellen Packages aufgenommen.

Deklariert man dagegen eine Variable mit dem Operator `my`, so wird die entsprechende Variable in einer anderen Tabelle abgelegt, auf die kein expliziter Zugriff möglich ist.

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

use strict;

use 5.6.0;             ### "our" funktioniert
                       ### erst ab Perl 5.6.0

our $var_1;
my $var_2;

$var_1 = 42;
$var_2 = "Perl";

foreach $v (keys %::) {      ### Symboltabelle von "main"
    if($v =~ /^var/) { print "$v\n" }
}
```

Hier erscheint in der Programmausgabe nur „`$var_1`“, nicht aber „`$var_2`“.

Neben der Tatsache, daß `my`-Variablen in einer eigenen Tabelle verwaltet werden, ist von besonderer Bedeutung, daß sie nur einen recht beschränkten Gültigkeitsbereich besitzen. Eine durch `my` erzeugte Variable steht nur in dem aktuellen Block (definiert durch geschweifte Klammern „`{...}`“), der aktuellen Datei oder innerhalb eines Arguments von `eval()` zur Verfügung. Außerhalb davon existieren diese Variablen nicht, es ist also (im Gegensatz zu `our`-Variablen) auch mit Hilfe des Package-Namens dort kein Zugriff möglich

Es kann in einem Package durchaus zwei Variablen gleichen Namens geben: eine in der Symboltabelle und eine, die durch einen Aufruf von `my` entstanden ist. In einem solchen Falle wird bei einfacher Verwendung des Bezeichners auf die `my`-Variable zugegriffen.

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

use strict;

use 5.6.0;             ### "our" funktioniert
                       ### erst ab Perl 5.6.0

our $ab = "main";
{
    my $ab;
    $ab = "Block";

    print "$ab\n";             ### "Block"
    print "$main::ab\n";       ### "main"
}
print "$ab\n";                 ### "main"
```

Im Block in der Mitte des Programms existiert neben „`$ab`“ der Symboltabelle

von `main` (Zugriff über voll qualifizierten Namen „`$main::ab`“ möglich) auch eine `my`-Variable gleichen Namens. Letztere verschwindet aber wieder, sobald der umschließende Block verlassen wird.

Es ist übrigens durchaus möglich, `my`-Deklarationen zu „schachteln“:

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

use strict;

my $ab = "main";
{
    my $ab;
    $ab = "Block";
    print "$ab\n";   ### "Block"
}
print "$ab\n";       ### "main"
```

Man bezeichnet solche Variablen als *lexikalisch*, weil deren Gültigkeitsbereich schon alleine durch Untersuchung des Programmcodes feststellbar ist.

Benötigt man in einem Perl-Programm eine lokale Variable, so sollte man im allgemeinen `my` verwenden. Lediglich in speziellen Situationen ist es angebracht, stattdessen `local` (siehe nächsten Abschnitt) zu benutzen.

27.4 Dynamische Variablen mittels `local()`

Auch der Operator `local` schränkt den Gültigkeitsbereich einer Variablen ein. Allerdings unterscheidet sich der Mechanismus grundlegend von dem des Operators `my`.

Wird eine Variable durch `local` deklariert, so wird der aktuelle Wert dieser Variablen (sofern vorhanden) gesichert. Anschließend können dieser Variablen neue Werte zugewisen werden. Wird der Block, in dem die `local`-Deklaration erfolgte, verlassen, so wird der ursprüngliche (gesicherte) Wert wiederhergestellt.

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

$ab = 123;
print "Main:  $ab\n";

{
   local $ab;
   $ab = 456;
   print "Block: $ab\n";
}

print "Main:  $ab\n";
```

```
Main:  123
Block: 456
Main:  123
```

Bei der Deklaration mit `local` wird *keine* neue Variable auf irgendeinem Stack erzeugt (wie bei `my`), sondern es wird nur der Inhalt neu zur Verfügung gestellt. Die entsprechende Variable ist also nach wie vor global zugänglich.

Daß die Variable nach wie vor eine globale Variable ist, erkennt man auch im folgenden Beispiel:

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

$x = 'x (main)';
print "Main: \$x = $x\n";

{
    local $x = 'x (local)';
    print "Block: \$x = $x\n";
    print "Symboltabelle: \$::x = $::x\n";
    unterpr();
}

print "Main: \$x = $x\n";

sub unterpr {
    print "unterpr(): \$x = $x\n";
}
```

```
Main: $x = x (main)
Block: $x = x (local)
Symboltabelle: $::x = x (local)
unterpr(): $x = x (local)
Main: $x = x (main)
```

Sowohl in der (globalen) Symboltabelle als auch in der Subroutine `unterpr()`, die sich, betrachtet man den Quell-Code des Programms, außerhalb des Blockes mit der `local`-Deklaration befindet, wird für `$x` der Wert „`x (local)`“ ausgegeben. Der ursprüngliche Wert wird erst wiederhergestellt, wenn in der Folge des Programmablaufs der Block verlassen wird, in dem die `local`-Deklaration stattfand.

Da die Gültigkeit von `local`-Variablen somit durch den Programmablauf bestimmt wird, spricht man auch von *Laufzeit-* oder *dynamischen* Variablen.

27.5 Unterschiede zwischen `my()` und `local()`

Die folgende Tabelle zeigt noch einmal schematisch den Unterschied zwischen `my` und `local`. **Markiert** sind diejenigen Werte, die ein „`print $v;`“ an der entsprechenden Stelle im Code ausgeben würde.

Zeile	Code	Symbol-tabelle von `main`	`local`-Stack	Tabelle für `my`-Variablen
1	`$v = 42;`	v = **42**		
2	`{ local $v = 100;`	v = **100**	v = 42	
3	`{ my $v = 7;`	v = 100	v = 42	v = **7**
4	`}`	v = **100**	v = 42	
5	`}`	v = **42**		

In Zeile 1 wird eine globale Variable „`$v`“ angelegt und mit dem Wert 42 belegt. Wird eine `local`-Variable erzeugt (Zeile 2), so überdeckt der neue Wert den alten; letzterer wird auf einem internen „Stapel“ (*stack*) gesichert. Unabhängig davon können mit Hilfe von „`my`“ lexikalische Variablen erzeugt werden (Zeile 3), die in einer eigenen internen Tabelle verwaltet werden. Die Gültigkeit der `my`-Variable endet mit der schließenden Klammer in Zeile 4. Mit dem Blockende in Zeile 5 wird der Gültigkeitsbereich der `local`-Variable verlassen und der ursprüngliche Wert wird vom Stack wiederhergestellt.

Auf die Variable in der Symboltabelle kann in diesem Beispiel immer mittels „`$main::v`“ zugegriffen werden. Existiert zusätzlich eine `my`-Variable, so bezieht sich „`$v`“ (ohne „`main::`“) immer auf diese (nicht auf das globale „`$v`“). Ein Zugriff auf den `local`-Stack ist hingegen nicht möglich.

Eine Situation, in der lokale Variablen *nicht* mittels `my` erzeugt werden können, sind die speziellen in Perl vordefinierten Variablen, wie etwa `$"` (Zeichen, das die Elemente eines Arrays voneinander trennt, wenn es innerhalb von doppelten Anführungszeichen interpoliert wird). Man kann wie folgt in einem Programmblock eine Neudefinition einer globalen Variable vornehmen, ohne den ursprünglichen Wert explizit sichern und am Ende wieder zurücksetzen zu müssen:

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

$" = ',';
@a = (10,20,30);
print "@a\n";

{
  local $" = ' - ';
  print "@a\n";
}

print "@a\n";
```

```
10,20,30
10 - 20 - 30
10,20,30
```

Index

@ARGV, 21
@EXPORT, 146
@EXPORT_OK, 146
@INC, 147
@ISA, 146
@_, 110
$!, 110
$<, 109
$>, 109
$(, 109
$), 110
$0, 109
$?, 110
$@, 106
$[, 110
$$, 109
$^, 109
$^O, 109
$^T, 109
$^V, 109
$^X, 109
$_, 107
$", 156
$], 109
%ENV, 22

abs, 28
Addition, 26
anonyme Subroutine, 115
anonymer Hash, 125
anonymes Array, 121
Array, 19
atan2, 28
atime, 52

BEGIN, 39
Benchmark, 135
Benutzer-ID, 109
Bereichsoperator, 45

call by reference, 113
call by value, 113
chdir, 57
chmod, 58
chomp, 33
chop, 32
chown, 58
chr, 35
close, 49
closedir, 56
Closure, 116
cos, 28
CPAN, 12
ctime, 52
cwd, 56

defined, 19
Dekrement, 26
die, 42
Division, 26
do, 138

each, 22
else, 46
elsif, 46
END, 41
English, 64
eval, 105
exit, 41
exp, 28
Exporter, 146

Fehlermeldung, 110
Filehandle, 48
for, 44
foreach, 44
format, 60
Funktion, 100

Ganzzahl, 14
Gleitkommazahl (Konstante), 14
Gleitkommazahl (Variable), 18
glob, 54

Globbing, 54
Gruppen-ID, 109

Hash, 21
Hexadezimalzahl, 14

if, 45
import, 146
Inkrement, 26

join, 34

keys, 21
Kommentar, 8

last, 46
lc, 32
lcfirst, 32
length, 33
link, 58
local, 101
log, 28

m/../, 69
map, 134
mkdir, 58
Modulo, 26
mtime, 52
Multiplikation, 26
my, 101

next, 46
NOT (logisch), 24

ODER (bitweise), 23
ODER (logisch), 24
Oktalzahl, 14
open, 49
opendir, 56
ord, 35
our(), 150

package, 143
perldoc, 10
Pipe, 51
pop, 36
POSIX, 28
Potenzbildung, 26
Pragma, 40
print, 8
Priorität, 25
Prozeß-ID, 109
push, 36

rand, 28
readdir, 56
Referenz, 111
regulärer Ausdruck, 69
regular expression, 69
rename, 57
require, 139
return, 103
Rhombus-Operator, 54
rmdir, 58

s/../../, 98
shift, 37
sin, 28
sort, 130
split, 34
sqrt, 28
srand, 28
stat, 54
STDERR, 50
STDIN, 50
STDOUT, 50
String, 14
sub, 100
Subroutine, 100
substr, 31
Subtraktion, 26
Suchoperatoren, 66
Symboltabelle, 149
symlink, 58

time, 59
tr/../../, 66

uc, 32
ucfirst, 32
UND (logisch), 24
undef, 18
unless, 45
unlink, 57
unshift, 37
Unterprogramm, 100
until, 43
use, 40
use strict, 18
use lib, 142
utime, 59

values, 21
Vergleiche (numerisch), 27
Vergleiche (Zeichenketten), 31
Verkettung, 18
Vorzeichen, 26

while, 43

XOR (bitweise), 23
XOR (logisch), 24

y/../../, 69

Zeichenkette, 14
Zufallszahl, 28
Zuweisung, 23