O'Reillys Taschenbibliothek

5. Auflage
Behandelt Perl 5.14.1

Perl

kurz & gut

O'REILLY®

Johan Vromans
Übersetzung von Peter Klicman

Perl

kurz & gut

Johan Vromans

Deutsche Übersetzung
von Peter Klicman

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag GmbH & Co. KG
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

1. Auflage 1996
2. Auflage 1999
3. Auflage 2000
4. Auflage 2003
5. Auflage 2012

Die Originalausgabe erschien 2011 unter dem Titel *Perl Pocket Reference, 5th Edition* bei O'Reilly Media, Inc.

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

Übersetzung: Peter Klicman, Köln
Lektorat: Imke Hirschmann, Köln
Korrektorat: Sibylle Feldmann, Düsseldorf
Produktion: Karin Driesen, Köln
Umschlaggestaltung: Hanna Dyer, Melanie Wang, Boston und Michael Oreal, Köln
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de
Druck: fgb freiburger graphische betriebe, Freiburg, www.fgb.de

ISBN: 978-3-86899-187-1

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Perl – kurz & gut

Perl – kurz & gut ist als kleines Nachschlagewerk für die von Larry Wall entwickelte Programmiersprache Perl gedacht. Es enthält eine Beschreibung aller Anweisungen, Funktionen und Variablen sowie viele nützliche Informationen.

Dieses Büchlein soll Perl-Benutzer dabei unterstützen, die Syntax bestimmter Funktionen und Anweisungen sowie die Bedeutung fest integrierter Variablen schnell aufzufinden. Es handelt sich jedoch nicht um ein eigenständiges Benutzerhandbuch, elementare Kenntnisse in der Sprache Perl werden vorausgesetzt. Diese Kurzreferenz ist auch nicht vollständig, einige der etwas obskureren Perl-Konstrukte wurden nicht berücksichtigt. Aber alle Funktionen und Variablen werden mit mindestens einer möglichen Anwendung genannt.

Perl 5.14.1

Perl-Releases werden über eine *Versionsnummer* identifiziert, eine Folge von mindestens zwei Zahlen, die durch Punkte voneinander getrennt sind. Dieses Büchlein beschreibt Perl 5.14.1, das am 16. Juni 2011 veröffentlicht wurde.

5.14.1	Bedeutung
5	Die Sprachrevision. Perl 5 wurde am 17. Oktober 1994 veröffentlicht.
14	Die Version dieser Revision. Eine gerade Zahl steht für eine offizielle Produktionsversion, während eine ungerade Zahl eine Entwicklungsversion anzeigt.
1	Die Unterversion. 0 (null) ist die erste Release. Höhere Zahlen stehen für Maintenance-Releases.

Konventionen

`dies`	steht für wörtlich einzugebenden Text.
`dies`	bezeichnet zu ersetzenden Text.
`dies``†`	bedeutet, dass *`dies`* standardmäßig durch $_ ersetzt wird, wenn für *`dies`* nichts eingegeben wird.
`Wort`	ist ein Schlüsselwort, d.h. ein Wort mit einer besonderen Bedeutung.
`[...]`	bezeichnet einen optionalen Teil.
	verweist auf die zugehörige Dokumentation, die mit dem `perldoc`-Befehl aufgerufen und angesehen werden kann.

Features

Seit der Version 5.10 unterstützt Perl sogenannte *Features*, um Syntax und Semantik der Sprache selektiv zu erweitern. Features können mit **`use`** `feature` aktiviert und mit **`no`** `feature` deaktiviert werden. Lesen Sie dazu den Abschnitt »Pragma-Module« auf Seite 21.

Feature	Perl	Beschreibung	Seite
say	5.10	Aktiviert das Schlüsselwort `say`	50
state	5.10	Aktiviert das Schlüsselwort `state`	64
switch	5.10	Aktiviert die Schlüsselwörter `given`, `when`, `break` und `default`	15
unicode_strings	5.12	Aktiviert die Unicode-Semantik für alle String-Operationen	
:5.10	5.10	Alle 5.10-Features	
:5.12	5.12	Alle 5.10- und 5.12-Features	
:5.14	5.14	Alle 5.10-, 5.12- und 5.14-Features	

Mit Feature-Bundles wie `:5.14` bekommen Sie schnell ein Perl mit allen Features. Allerdings ist es einfacher, versionsabhängige Features wie folgt automatisch zu aktivieren:

```
use 5.14.0;
```

Syntax

Perl ist eine Programmiersprache mit einem freien Format. Das bedeutet, dass es im Allgemeinen keine Rolle spielt, wie Ihre Zeilen aussehen und welche Einrückungen Sie für das Perl-Programm verwenden.

Die Ausnahme von dieser Regel bildet das Vorkommen eines Doppelkreuzes (#) in der Eingabe. Trifft der Perl-Compiler auf ein #, werden dieses Zeichen und der bis zum Zeilenende folgende Text entfernt. Auf diese Weise können Sie Perl-Programme mit Kommentaren versehen. Eingefleischte Programmierer tragen viele nützliche Kommentare in ihre Programme ein.

Es gibt Stellen, an denen Whitespace eine Rolle spielt: innerhalb literalen Texts, in Mustern und bei Formaten.

Entdeckt der Perl-Compiler das spezielle Token `__DATA__`, entfernt er dieses Symbol und beendet das Lesen der Eingabe. Alles, was auf dieses Token folgt, wird vom Perl-Compiler ignoriert, kann vom

Programm während der Ausführung aber über das Dateihandle DATA gelesen werden!

__END__ verhält sich in Top-Level-Skripten wie __DATA__ (jedoch nicht bei Dateien, die mit **require** oder **do** geladen wurden). Der restliche Dateiinhalt bleibt über das globale Dateihandle DATA zugänglich.

Wenn Perl eine neue Anweisung erwartet und dabei eine mit = beginnende Zeile entdeckt, überspringt es alle Eingaben bis einschließlich der Zeile, die mit =cut beginnt. Auf diese Weise kann Dokumentation eingebettet werden.

perlsyn

Eingebettete Dokumentation

Es gibt Tools, mit denen eingebettete Dokumentation extrahiert und in unterschiedliche Formate wie troff, LaTeX und HTML umgewandelt werden kann. Die folgenden Befehle können verwendet werden, um die eingebettete Dokumentation zu steuern:

=back
: Siehe =over weiter unten.

=begin *Formatierer*
: Der nachfolgende Text bis zum zugehörigen =end wird nur bei Aufbereitung für den *Formatierer* eingefügt.

=cut
: Beendet einen Dokumentationsabschnitt.

=end *Formatierer*
: Siehe =begin.

=for *Formatierer*
: Der nächste Absatz wird nur bei Aufbereitung für den *Formatierer* eingefügt.

=head *N Überschrift*
: Erzeugt eine Überschrift. *N* muss den Wert 1, 2, 3 oder 4 aufweisen.

=item *Text*
: Siehe =over.

`=over N` Beginnt eine Aufzählung mit der Einrückung *N*. Die einzelnen Elemente werden über `=item` angegeben. Die Aufzählung endet mit `=back`.

`=pod` Leitet einen Dokumentationsabschnitt ein. Jeder der =-Befehle kann verwendet werden, um einen Dokumentationsabschnitt einzuleiten.

Alle genannten Befehle werden auf den Textabsatz angewandt, der diesen Befehlen folgt. Absätze werden durch (mindestens) eine Leerzeile beendet.

Ein eingerückter Absatz wird als wortwörtlicher Text betrachtet und auch als solcher aufbereitet.

Innerhalb normaler Absätze können Markup-Sequenzen eingefügt werden:

`B<`*Text*`>` Gibt den Text fett aus (für Switches und Programme).

`C<`*Code*`>` Gibt Code in Nichtproportionalschrift (konstante Zeichenbreite) aus.

`E<`*Escape*`>`
Ein benanntes Zeichen. So stellt etwa `E<lt>` das Zeichen < dar, und `E<gt>` bedeutet >.

`F<`*Datei*`>`
Dateiname.

`I<`*Text*`>` Kursive Schrift (für Hervorhebungen und Variablen).

`L<` [*Text* |] [*Referenzziel*] [/ *Abschnitt*] `>`
Ein Querverweis. Der *Text* wird, falls vorhanden, für die Ausgabe verwendet.

`S<`*Text*`>` Den *Text* nicht bei Leerzeichen umbrechen.

`X<`*Index*`>` Indexeintrag.

`Z< >` Ein Zeichen der Breite null.

Markup-Sequenzen dürfen verschachtelt werden. Soll eine Markup-Sequenz >-Zeichen enthalten, verwenden Sie `C<< ... >>` oder `C<<<`

... >>> usw. Nach dem letzten < und vor dem ersten > muss jeweils ein Whitespace stehen.

perlpod, perlpodspec

Datentypen

Die Rolle der Sigille wird im Abschnitt »Variablen« auf Seite 11 erläutert.

Typ	Sigil	Beschreibung
Array	@	Indizierbare Liste skalarer Werte
Code	&	Perl-Code, beispielsweise eine Subroutine
Format		Ein Format zur Erzeugung von Berichten (Reports)
Glob	*	Alle Datentypen
Hash	%	Assoziatives Array skalarer Werte
IO		Dateihandle. Wird bei Eingabe- und Ausgabeoperationen verwendet
Skalar	$	Strings, Zahlen, Typeglobs und Referenzen

perldata

Quoting und Interpolation

Perl verwendet die üblichen Quoting-Zeichen zur Konstruktion von Strings, implementiert aber auch einen generischen Quoting-Mechanismus. Der String `'Hallo!'` kann beispielsweise als `q/Hallo!/`, `q;Hallo!;`, `q{Hallo!}` und so weiter geschrieben werden.

Üblich	Generisch	Bedeutung	Inter.	Seite
''	q//	Literaler String	Nein	9
""	qq//	Literaler String	Ja	9
``	qx//	Befehlsausführung	Ja	9
()	qw//	Wortliste	Nein	10
""	qr//	Regulärer Ausdruck	Ja	9

Üblich	Generisch	Bedeutung	Inter.	Seite
`//`	`m//`	Mustervergleich	Ja	51
`s///`	`s///`	Mustersetzung	Ja	52
`y///`	`tr///`	Zeichenübersetzung	Nein	53

Wenn der Quoting-Mechanismus drei Trennsymbole verlangt, können Sie auch zwei unterschiedliche Gruppierungszeichen verwenden: `m< ... >` und `s{ ... }[ ... ]`.

Die Spalte »Inter.« der obigen Tabelle gibt an, ob Escape-Sequenzen im String interpoliert werden. Wenn als Trennzeichen für die Mustererkennung oder Substitution einfache Anführungszeichen verwendet werden, erfolgt keine Interpolation.

Escape-Sequenzen für Strings:

`\a` Alarm (bell)

`\b` Backspace

`\e` Escape

`\f` Formfeed (Seitenvorschub)

`\n` Newline

`\r` Return (Zeilenvorschub)

`\t` Tabulator

In Kombination mit Präfixen lassen sich Zeichen erzeugen, zum Beispiel:

`\53` Oktal, ASCII-Zeichen +. Oktal-Escapes bestehen aus bis zu drei oktalen Ziffern, einschließlich der führenden Nullen. Der resultierende Wert darf 377 oktal nicht überschreiten.

Bei Mustern (die sich wie **qq**//-Strings verhalten), sind führende Nullen für Oktal-Escapes obligatorisch, um die Interpretation als Rückwärtsreferenz zu verhindern, falls der Wert die Anzahl der Treffer (oder 9, je nachdem, was kleiner ist) nicht übersteigt. Beachten Sie, dass der Wert

bei einer Rückwärtsreferenz nicht als Oktal-, sondern als Dezimalzahl interpretiert wird.

`\cC` Wird als Steuerzeichen interpretiert: Strg-C.

`\N{BLACK SPADE SUIT}`
Ein benanntes Zeichen: ♠. Verlangt das `charnames`-Pragma; siehe Seite 22.

`\N{U+03A3}`
Unicode-Zeichen mit Codepoint 03A3 (hex).

`\o{53}` Eine Oktalzahl (die sichere Variante).

`\xeb` Wird als Hexadezimalzeichen interpretiert: Latin-1-Zeichen ë. Hexadezimal-Escapes bestehen aus einer oder zwei hexadezimalen Ziffern.

`\x{03a3}`
Unicode-Hexadezimal, das griechische Zeichen Σ.

Diese Escape-Sequenzen verändern die Bedeutung des darauf Folgenden:

`\E` Beendet `\L`, `\Q` und `\U`

`\l` Schreibt das nachfolgende Zeichen klein

`\L` Schreibt alles bis zum nächsten `\E` klein

`\u` Schreibt das nachfolgende Zeichen groß

`\U` Schreibt alles bis zum nächsten `\E` groß

`\Q` Quoting von Nicht-Wort-Zeichen bis zum nächsten `\E`

`perlop`, `perlunicode`, `perluniintro`

Literale Werte

Skalare Werte

Array-Referenz
`[1,2,3]`

Code-Referenz

`sub { *Anweisungen* }`

Hash-Referenz

`{schlüssel1 => wert1, schlüssel2 => wert2, ... }`

Identisch mit `{schlüssel1, wert1, schlüssel2, wert2, ... }`

Numerisch

`123 1_234 123.4 5E-10 0b010101` (binär) `0xff` (hexadezimal) `0377` (oktal)

`__LINE__` (Zeilennummer im aktuellen Programm)

Reguläre Ausdrücke

`qr/*String/Modifier*`

String

`'abc'` Literaler String, keine Variableninterpolation oder Escape-Zeichen (außer \' und \\).

`"abc"` Variablen werden interpoliert, und Escape-Sequenzen werden verarbeitet.

`` `*Befehl*` ``
Evaluiert zur Ausgabe des Befehls.

`Class::` Evaluiert zur Ausgabe des Befehls.

`1.2.3 v5.6.0.1`
Ein aus den angegebenen Ordnungszahlen zusammengesetzter String (»v-String«). Die Werte können im Unicode-Bereich liegen. `v1.3` ist mit `"\x{1}\x{3}"` identisch. Geeignet zum Vergleich mit anderen v-Strings mithilfe von String-Vergleichsoperatoren.

`<< *Identifier*`
»Hier-Dokument« im Shell-Stil.

__FILE__
: Name der Programmdatei.

__PACKAGE__
: Name des aktuellen Pakets.

Listenwerte

(...) (1,2,3) ist eine Liste mit drei Elementen.

(1,2,3)[0] ist das erste Element dieser Liste.

(1,2,3) [–1] ist das letzte Element.

( ) ist eine leere Liste.

(1..4) ist das Gleiche wie (1,2,3,4); Gleiches gilt für ('a'..'z').

('a'..'z')[4,7,9] ist ein Teilstück (Slice) aus der literalen Liste.

qw qw/fo br ... / ist das gleiche wie ('fo','br', ...).

Hash-Werte

(...) (schlüssel1 => wert1, schlüssel2 => wert2, ...)

Identisch mit (schlüssel1, wert1, schlüssel2, wert2, ...)

Dateihandles

Vordefinierte Dateihandles
: STDIN, STDOUT, STDERR, ARGV, DATA.

Benutzerdefinierte Dateihandles
: Jeder Name (Identifier ohne Sigille), der weder Schlüsselwort noch Subroutinenaufruf darstellt, kann als Dateihandle verwendet werden.

Variablen

`$var` Eine einfache skalare Variable.

`$p = \$var`
: `$p` ist nun eine Referenz auf den Skalar `$var`.

`$$p` Der durch `$p` referenzierte Skalar.

`@var` Ein Array. Im skalaren Kontext die Anzahl der Elemente im Array.

`$var[6]` Das 7. Element des Arrays `@var`.

`$var[--1]`
: Das letzte Element von Array `@var`.

`$p = \@var`
: `$p` ist nun eine Referenz auf das Array `@var`.

`$$p[6]` oder `$p->[6]`
: Das oder 7. Element des durch `$p` referenzierten Arrays.

`${$p[6]}` Der über `$p[6]` referenzierte skalare Wert.

`$p = \$var[6]`
: `$p` ist nun eine Referenz auf das 7. Element von Array `@var`.

`$p=[1,3,'Affe']`
: `$p` ist nun eine Referenz auf ein anonymes Array mit drei Elementen.

`$var[$i][$j]`
: Das `$j`-te Element des `$i`-ten Elements von Array `@var`.

`$#var` Der letzte Index des Arrays `@var`.

`@var[3,4,5]`
: Ein Teilstück (Slice) des Arrays `@var`.

`%var` Ein Hash. Im skalaren Kontext wahr, wenn der Hash Elemente besitzt.

`$var{'rot'}` oder `$var{rot}`
: Ein Wert aus dem Hash `%var`. Der Hash-Schlüssel muss nicht in Quoting-Zeichen stehen, wenn es sich um einen einfachen Bezeichner handelt.

`$p = \%var`
: `$p` ist nun eine Referenz auf den Hash `%var`.

`$$p{'rot'}` oder `$p->{'rot'}`
: Ein Wert aus dem durch `$p` referenzierten Hash.

`${$p{'rot'}}`
: Der über `$p{'rot'}` referenzierte skalare Wert.

`$p = {rot => 1, blau => 2, gelb => 3}`
: `$p` ist nun eine Referenz auf einen anonymen Hash mit drei Elementen.

`@var{'a','b'}`
: Ein Teilstück von `%var`; das Gleiche wie (`$var{'a'}`, `$var{'b'}`).

`$var{'a'}[1]`
: Ein mehrdimensionaler Hash.

`$var{'a',1, ... }`
: Ein mehrdimensionaler Hash (veraltet).

`$c = \&mysub`
: `$c` ist nun eine Referenz auf die Subroutine `mysub`.

`$c = sub { ... }`
: `$c` ist nun eine Referenz auf eine anonyme Subroutine.

`&$c(`*Argumente*`)` oder `$c->(`*Argumente*`)`
: Aufruf einer Subroutine per Referenz.

`$MeinPaket::var`
: Variable `$var` im Paket `MeinPaket`.

  Gilt ebenso für `@MeinPaket::array` und so weiter.

  Variablen, die nicht Teil eines Pakets sind, gehören zum Standardpaket `main`.

`$::var` Identisch mit `$main::var`.

`%MeinPaket::`
: Die Symboltabelle des Pakets.

`*var`
: Symboltabelleneintrag (Typeglob). Verweist auf alles, was durch `var`: repräsentert wird: `$var`, `@var`, `%var` und so weiter.

`*x = \ $y`
: Macht `$x` zu einem Alias für `$y`.

`*x = *y`
: Macht alle `x` zu Aliases für `y`. Auch `*x = "y"`.

`*var{SCALAR}` *oder* `*{$::{var}}{SCALAR}`
: Identisch mit `\$var`. Ebenso ist `*var{ARRAY}` mit `\@var` identisch. Gilt auch für `HASH`, `CODE`, `FORMAT`, `GLOB` und `IO`.

Beachten Sie, dass `$var`, `@var`, `%var`, Subroutine var, Format `var` und Dateihandle `var` zwar alle den Identifier `var` verwenden, es handelt sich dabei aber um unterschiedliche Variablen.

Sie können statt eines Variablenbezeichners auch immer einen *Block* verwenden (siehe Seite 17), der den richtigen Referenztyp zurückgibt, zum Beispiel: `${ $x > 0 ? \$y[4] : \$z }`.

`perldata`, `perlref`

Kontext

Ausdrücke werden in Perl immer in einem Kontext evaluiert. Dieser Kontext bestimmt das Ergebnis der Evaluierung

Boolescher Kontext
: Eine spezielle Form des skalaren Kontexts, bei dem nur relevant ist, ob das Ergebnis wahr oder falsch ist.

  Alles, was undefiniert ist oder zu einem leeren String evaluiert, die Zahl Null sowie der String `"0"` werden als falsch betrachtet, alles andere ist wahr (auch Strings wie `"00"`).

Listenkontext
: Erwartet wird ein Listenwert. Akzeptierte Werte sind literale Listen, Arrays und Hashes. Teilstücke von Arrays,

Hashes und Listen sind ebenfalls erlaubt. Ein skalarer Wert wird als Liste mit nur einem Element interpretiert.

Skalarer Kontext
: Erwartet wird ein einzelner skalarer Wert.

Void-Kontext
: Es wird kein Wert erwartet. Wird ein Wert bereitgestellt, wird er ignoriert.

Die folgenden Funktionen sind an den Kontext gebunden:

`scalar` *Ausdruck*
: Erzwingt einen skalaren Kontext für den Ausdruck.

`wantarray`
: Gibt wahr im Listenkontext, falsch im skalaren Kontext und *undef* im Void-Kontext zurück.

Operatoren und Vorrang

Perl-Operatoren besitzen die folgende Assoziativität und den folgenden Vorrang. Die Tabelle ist vom höchsten zum niedrigsten Vorrang sortiert, wobei mehrere Operatoren in einer Tabellenzelle den gleichen Vorrang aufweisen.

Assoz.	Operatoren	Beschreibung
rechts	Terme und Listenoperatoren	Siehe unten
links	`->`	Infix-Dereferenzierungsoperator
keine	`++`	Auto-Inkrement (»magisch« bei Strings)
	`--`	Auto-Dekrement
rechts	`**`	Exponentierung
rechts	`\`	Referenz auf ein Objekt (unär)
rechts	`! ~`	Unäre Negation, bitweises Komplement
rechts	`+ -`	Unäres Plus, unäres Minus
links	`=~`	Bindet einen skalaren Ausdruck an eine Mustererkennung

Assoz.	Operatoren	Beschreibung
links	! ~	Das Gleiche, negiert aber das Ergebnis
links	* / % x	Multiplikation, Division, Modulo, Wiederholung
links	+ - .	Addition, Subtraktion, Verkettung
links	>> <<	Bitweiser Rechts-Shift, bitweiser Links-Shift
rechts	benannte unäre Operatoren	Zum Beispiel sin, chdir, -f, -M
keine	< > <= >= lt gt le ge	Numerische relationale Operatoren Relationale String-Operatoren
keine	== != <=> eq ne cmp ~~	Numerisches Gleich, Ungleich und numerischer Vergleich String-orientiertes Gleich, Ungleich und string-basierter Vergleich Smartmatch
links	&	Bitweises UND
links	| ^	Bitweises ODER, Exklusiv-ODER
links	&&	Logisches UND
links	|| //	Logisches ODER, definiertes ODER ?
keine	..	Bereichsoperator
	...	Alternativer Bereichsoperator
rechts	?:	Bedingungsoperator (if ? then : else)
rechts	= += -= etc.	Zuweisungsoperatoren
links	,	Kommaoperator, auch Trennzeichen für Listenelemente
links	=>	Das gleiche, macht den linken Operanden zu einem String
rechts	Listenoperatoren (rechtsgerichtet)	Siehe unten
rechts	not	Logisches NICHT mit geringem Vorrang
links	and	Logisches UND mit geringem Vorrang
links	or	Logisches ODER mit geringem Vorrang
links	xor	Logisches Exklusiv-ODER mit geringem Vorrang

Mit Klammern kann ein Ausdruck in einen Term eingebunden werden.

Eine Liste besteht aus Ausdrücken, Variablen, Arrays, Hashes, Slices oder Listen, die voneinander durch Kommata getrennt werden. Sie wird immer als lineare Wertefolge interpretiert.

Perl-Funktionen, die als Listenoperatoren verwendet werden können, besitzen entweder einen sehr hohen oder einen sehr niedrigen Vorrang, je nachdem, ob Sie die linke oder die rechte Seite des Operators betrachten. Um Vorrangprobleme zu vermeiden, können um die Parameterlisten Klammern gesetzt werden.

Die logischen Operatoren evaluieren den rechten Operanden nicht, wenn das Ergebnis bereits nach der Evaluierung des linken Operanden bekannt ist.

Vergleichsoperatoren liefern –1 (kleiner), 0 (gleich) oder 1 (größer) zurück.

`perlop`, `perlfunc`

`perldoc -f` *`Funktion`* gibt ausführliche Informationen zur angegebenen Funktion zurück.

Anweisungen

Eine Anweisung ist ein Ausdruck (dem optional ein Modifier folgt) und wird durch ein Semikolon abgeschlossen. Anweisungen können zu einem *`Block`* kombiniert werden, indem man sie in {} einschließt. Das Semikolon kann nach der letzten Anweisung des Blocks weggelassen werden.

Die Ausführung von Ausdrücken kann von anderen Ausdrücken abhängig gemacht werden, indem einer der Modifier **`if`**, **`unless`**, **`for`**, **`foreach`**, **`when`**, **`while`** oder **`until`** verwendet wird, beispielsweise:

```
Ausdruck1 if Ausdruck2 ;
Ausdruck1 foreach Liste ;
```

Die Operatoren ||, //, && und ?: ermöglichen eine bedingte Ausführung:

Ausdruck1 || *Ausdruck2* ;
Ausdruck1 ? *Ausdruck2* : *Ausdruck3* ;

Blöcke können ebenfalls für eine bedingte Ausführung verwendet werden:

if (*Ausdruck*) *Block* [**elsif** (*Ausdruck*) *Block* ...] [**else** *Block*]
unless (*Ausdruck*) *Block* [**else** *Block*]

Schleifen

[*Label:*] **while** (*Ausdruck*) *Block* [**continue** *Block*]
[*Label:*] **until** (*Ausdruck*) *Block* [**continue** *Block*]
[*Label:*] **for** ([*Ausdruck*] ; [*Ausdruck*] ; [*Ausdruck*]) *Block*
[*Label:*] **foreach** *Variable*†(*Liste*) *Block* [**continue** *Block*]
[*Label:*] *Block* [**continue** *Block*]

Bei **foreach** ist die Iterationsvariable (standardmäßig $_) ein Alias auf das entsprechende Element der Liste. Eine Modifikation dieser Variablen verändert daher das eigentliche Listenelement.

Die Schlüsselwörter **for** und **foreach** sind austauschbar.

Bei Schleifenblöcken kann der Programmfluss wie folgt kontrolliert werden:

goto *Label*
: Sucht die Anweisung namens *Label* und setzt die Ausführung dort fort. *Label* kann ein Ausdruck sein, der zum Namen eines Labels evaluiert.

last [*Label*]
: Verlässt unmittelbar die fragliche Schleife. Überspringt den **continue**-Block.

next [*Label*]
: Führt den **continue**-Block aus und startet die nächste Iteration der Schleife.

redo [*Label*]
: Durchläuft den Schleifenblock erneut, ohne die Bedingung noch einmal zu evaluieren. Überspringt den **continue**-Block.

When-Blöcke

when-Blöcke können innerhalb eines *Topicalizer* zur Bildung einer Switch-Anweisung genutzt werden. Topicalizer sind **foreach** (oder **for**) und **given**:

```
for ( Ausdruck ) {
  [ when ( Bedingung ) Block...]
  [ default Block ]
}
given ( Ausdruck ) {...}
```

Die Prüfung der *Bedingung* erfolgt per Smartmatching (siehe Seite 42). Der Block der ersten zutreffenden Bedingung wird ausgeführt, und die Kontrolle wird an das Ende des Topicalizer-Blocks übergeben. **default** ist ein immer zutreffendes **when**.

Innerhalb eines **when**-Blocks kann mit **continue** zur nächsten Anweisung statt zum Ende des Topicalizer-Blocks gesprungen werden.

In einem **given**-Block kann der Topicalizer-Block mit **break** verlassen werden.

Beachten Sie, dass **given** ein relativ neues Feature ist und verschiedene Verhaltensaspekte in nachfolgenden Perl-Releases noch verändert werden können.

Spezielle Formen

do *Block* **while** *Ausdruck* ;
do *Block* **until** *Ausdruck* ;

Diese stellen sicher, dass *Block* einmal ausgeführt wird, bevor *Ausdruck* getestet wird.

do *Block*

macht aus dem *Block* tatsächlich einen Ausdruck.

```
...
```

Der Platzhalter `...` (Ellipse) kann als Anweisung verwendet werden, um noch nicht geschriebenen Code anzuzeigen. Der Code wird kompiliert, führt aber zu einer Ausnahme, wenn er ausgeführt wird.

`perlsyn`

Pakete und Module

import *Modul* [*Liste*]
: Importiert Subroutinen und Variablen aus *Modul* in das aktuelle Paket. **import** ist keine fest integrierte Anweisung, sondern eine gewöhnliche Klassenmethode, die von UNIVERSAL geerbt werden kann.

no *Modul* [*Liste*]
: Während der Kompilierung wird das Modul über **require** geladen, und dann wird die **unimport**-Methode für die *Liste* aufgerufen. Siehe **use** weiter unten.

package *Namensraum* [*Version*] [*Block*]
: Weist den *Block* als Paket mit einem Namensraum aus. Fehlt der *Block*, gilt das für den Rest des aktuellen Blocks oder der Datei. Falls vorhanden, wird die Paketvariable $VERSION auf *Version* gesetzt.

require *Version*
: Der Perl-Interpreter muss mindestens diese Versionsnummer haben. *Version* kann numerisch sein, etwa 5.005 oder 5.008001, oder ein v-String wie v5.8.1.

require *Ausdruck*†

Ist *Ausdruck* numerisch, entspricht das Verhalten dem von **require** *Version*. Anderenfalls muss *Ausdruck* der Name einer Datei sein, die aus der Perl-Bibliothek eingebunden wird. Bindet die Datei nur einmal ein und führt zu einem fatalen Fehler, wenn die Datei nicht zu einem wahr-Wert evaluiert. Wenn *Ausdruck* kein Suffix hat, wird für den Namen die Erweiterung `.pm` angenommen.

unimport *Modul* [*Liste*]

Hebt den Effekt eines vorangegangenen **import** oder **use** auf. Wie **import** auch, ist **unimport** nicht fest integriert, sondern eine gewöhnliche Klassenmethode.

use *Version*

use *pragma*

Siehe den nachfolgenden Abschnitt »Pragma-Module« auf Seite 21.

Per Konvention beginnen Pragma-Namen mit Kleinbuchstaben.

use *Modul* [*Version*] [*Liste*]

Während der Kompilierung wird das Modul über **require** geladen, optional wird die Version überprüft und die entsprechende **import**-Methode für die *Liste* aufgerufen.

Wird die *Liste* mit `()` angegeben, erfolgt kein Aufruf von **import**.

Wird normalerweise benutzt, um eine Liste von Variablen und Subroutinen aus dem angegebenen Modul in das aktuelle Paket zu importieren.

Modulnamen beginnen mit einem Großbuchstaben.

 `perlmod`

Pragma-Module

Pragma-Module beeinflussen die Kompilierung Ihres Programms. Pragma-Module können mit **use** aktiviert (importiert) und mit **no** deaktiviert werden. Sie sind üblicherweise blockbezogen.

version Der Perl-Interpreter muss mindestens diese Versionsnummer haben. Aktiviert das **Feature**-Bundle für diese Version. Ab Version 5.12 oder höher wird implizit das strict-Pragma aktiviert.

Version kann numerisch sein, etwa 5.005 oder 5.008001, oder ein v-String wie 5.8.1 oder V5.14. Unglücklicherweise wird 5.14 als 5.140 interpretiert.

Mit **no** muss Perl älter sein als die angegebene Version. Die Features werden nicht aktiviert.

autodie Ersetzt die eingebauten Funktionen durch solche, die bei einem Fehler automatisch per die abbrechen.

attributes
Aktiviert Attribute.

autouse *Modul* => *Funktionen*
Das Modul wird erst geladen, wenn eine der angegebenen Funktionen aufgerufen wird.

base *Klassen*
Baut während der Kompilierung eine IS-A-Beziehung mit den Basisklassen auf.

bigint [*Optionen*]
Nutzt transparent das Math::BigInt-Paket für alle Ganzzahlberechnungen.

Die *Optionen* sind accuracy, precision, trace, version und lib. Aus einem Buchstaben bestehende Abkürzungen sind erlaubt. accuracy und precision benötigen ein numerisches Argument, während lib den Namen eines Perl-Moduls verlangt, das die Berechnungen durchführt.

`bignum` [*Optionen*]
: Nutzt transparent das `Math::BigNum`-Paket für alle numerischen Berechnungen.

  Siehe `bigint` oben für die *Optionen*.

`bigrat`
: Nutzt transparent die Pakete `Math::BigNum` und `Math::BigRat` für alle numerischen Berechnungen.

  Siehe `bigint` oben für die *Optionen*.

`blib` [*Verzeichnis*]
: Verwendet die nicht mittels MakeMaker installierte Version eines Pakets. *Verzeichnis* ist mit dem aktuellen Verzeichnis voreingestellt. Wird zum Test nicht installierter Pakete verwendet.

`bytes`
: Betrachtet zeichenorientierte Daten als reine 8-Bit-Bytes (im Gegensatz zu Unicode UTF-8).

`charnames` [*Zeichensätze*]
: Erlaubt die Auflösung benannter Zeichen in Strings mittels `\N`-Escapes.

`constant` *Name* `=>` *Wert*
: Definiert *Name* als konstanten Wert.

`diagnostics` [*Ausführlichkeit*]
: Erzwingt ausführliche Warnmeldungen und unterdrückt doppelte Warnungen. `use diagnostics "-verbose"` ist noch ausführlicher.

`encoding` [*Codierung*]
`encoding` *Codierung* [`STDIN =>` *E_Codierung*] [`STDOUT =>` *A_Codierung*]
: Legt die Skriptcodierung fest und aktiviert die *Codierungs*-E/A-Schicht für die Standardein- und -ausgabe. Die zweite Form erlaubt die explizite Wahl der E/A-Schichten.

`encoding::warnings`
: Gibt eine Warnung aus, wenn ein Nicht-ASCII-Zeichen implizit in UTF-8 umgewandelt wird.

`feature` *Feature* [, *Feature*]
: Aktiviert Features. Siehe den Abschnitt »Features« auf Seite 2.

`fields` *Namen*
: Implementiert die Verifikation von Klassenfeldern während der Kompilierung.

`filetest` [*Strategie*]
: Ändert die Art und Weise, in der die Dateitestoperatoren (Seite 56) an ihre Informationen gelangen. Die Standardstrategie ist `stat`, die Alternative ist `access`.

`if` *Bedingung* , *Modul* `=>` *Argumente*
: Bindet ein Modul über `use` ein, wenn die Bedingung erfüllt wird.

`integer`
: Führt Berechnungen mit Integer-Werten statt mit Double-Werten durch.

`less` *was*
: Fordert weniger von »irgend etwas« an (nicht implementiert).

`lib` *Namen*
: Nimmt Bibliotheken während der Kompilierung in `@INC` auf oder entfernt sie.

`locale`
: Verwendet POSIX-Locales bei fest integrierten Operationen.

`mro` *Typ*
: Verwendet *Typ* bei der Auflösung der Methode. Mögliche Werte sind `dfs` (Voreinstellung) und `c3`.

`open`
: Baut Standard-E/A-Schichten für die Ein- und Ausgabe auf.

`ops` *Operationen*
: Beschränkt unsichere Operationen während der Kompilierung.

`overload` *`operator`* `=>` *`Subref`*

Pakete zur Überladung von Perl-Operatoren. *`operator`* ist der Operator (als String), *`Subref`* ist eine Referenz auf die Subroutine, die den überladenen Operator behandelt.

`overloading`

(De-)Aktiviert das lexikalische Überladen.

`parent` *`classes`*

Baut während der Kompilierung eine IS-A-Beziehung mit den genannten Klassen auf.

`re` [*`Verhalten`* | "*`/Flags`*"]

Ändert das Verhalten regulärer Ausdrücke. *`Verhalten`* ist eine beliebige Kombination aus `eval` (ermöglicht Muster mit Annahmen, die Perl-Code ausführen, selbst wenn das Muster interpolierte Variablen enthält; siehe Seite 46), `taint` (propagiert das Tainting), `debug` und `debugcolor` (erzeugen Debugging-Informationen).

Mit *`Flags`* können Sie Standard-Flags für reguläre Ausdrücke festlegen.

`sigtrap` *`Info`*

Aktiviert die einfache Signalbehandlung. *`Info`* ist eine Liste der Signale, z.B. `qw(SEGV TRAP)`.

`sort` [*`Optionen`*]

Kontrolliert das Verhalten von **`sort`**. Mögliche *`Optionen`* sind `stable`, um Stabilität zu verlangen, `_quicksort` (oder `_qsort`) zur Verwendung eines Quicksort-Algorithmus und `_mergesort` zur Verwendung eines Mergesort-Algorithmus.

`strict` [*`Konstrukte`*]

Beschränkt unsichere Konstrukte. *`Konstrukte`* ist eine beliebige Kombination aus `refs` (beschränkt die Verwendung symbolischer Referenzen), `vars` (verlangt, dass alle Variablen von Perl vordefiniert, importiert, global oder lexikalisch beschränkt oder vollständig qualifiziert sind) und `subs` (beschränkt die Verwendung von Bareword-Bezeichnern, bei denen es sich nicht um Subroutinen handelt).

`subs` *Namen*
: Vordeklaration von Subroutinennamen. Erlaubt deren Verwendung ohne Klammern, noch bevor sie deklariert werden.

`threads`
: Aktiviert die Verwendung Interpreter-basierter Threads. Beachten Sie hierzu den Abschnitt »Threads« auf Seite 92.

`threads::shared`
: Bindet die gemeinsame Nutzung von Daten zwischen Threads ein.

`utf8`
: Aktiviert bzw. deaktiviert UTF-8 (oder UTF-EBCDIC) im Quellcode.

`vars` *Namen*
: Vordeklaration von Variablennamen. Erlaubt ihre Verwendung unter dem `strict`-Pragma ohne vollständige Qualifizierung. Veraltet; verwenden Sie stattdessen **`our`** (Seite 76).

`version`
: Aktiviert die Unterstützung von Versionsobjekten.

`vmsish` [*`Features`*]
: Kontrolliert VMS-spezifische Sprachmerkmale. Gilt nur für VMS. *`Features`* ist eine beliebige Kombination aus `exit` (aktiviert VMS-artige Exit-Codes), `status` (erlaubt Systembefehlen, VMS-artige Exit-Codes an das aufrufende Programm zu übergeben) und `time` (wandelt alle Zeitangaben relativ zur lokalen Zeitzone um).

`warnings` [*`Klasse`*]
: Kontrolliert die fest eingebauten Warnungen für bestimmte Klassen von Bedingungen.

`warnings::register`
: Erzeugt eine Warnungskategorie für das aktuelle Paket.

`perlmodlib`

Subroutinen

Subroutinen müssen *deklariert* werden, d.h., man muss angeben, wie sie aufgerufen werden sollen. Darüber hinaus müssen Subroutinen auch *definiert* werden, d.h., es muss festgelegt werden, was sie bei einem Aufruf machen sollen.

sub *Name* [(*Prototyp*)] [*Attribute*] **;**
: Deklariert *Name* als Subroutine, wobei optional der Prototyp und die Attribute angegeben werden. Die Deklaration einer Subroutine ist optional, erlaubt aber ihren Aufruf wie bei den fest in Perl integrierten Operatoren.

sub [*Name*] [(*Prototyp*)] [*Attribute*] *Block*
: Definiert die Subroutine *Name* mit einem optionalen Prototyp und Attributen. Wurde die Subroutine mit einem Prototyp oder mit Attributen deklariert, muss die Definition den gleichen Prototyp und die gleichen Attribute aufweisen. Wurde *Name* weggelassen, ist die Subroutine anonym, und die Definition gibt eine Referenz auf den Code zurück.

Beim Aufruf der Subroutine werden die Anweisungen im *Block* ausgeführt. Parameter werden in einer linearen Liste von Skalaren im Array `@_` übergeben. Die Elemente von `@_` sind Aliase für die skalaren Parameter. Der Aufruf liefert den Wert des zuletzt evaluierten Ausdrucks zurück. Sie können **`wantarray`** (Seite 14) verwenden, um den Kontext zu bestimmen, in dem die Subroutine aufgerufen wurde.

Haben Subroutinen einen leeren Prototyp und geben nur einen festen Wert zurück, erfolgt ein Inlining, zum Beispiel bei `sub PI() { 3.1415 }`. Siehe hierzu auch den Abschnitt »Prototypen« auf Seite 28.

Attribute werden mit einem : (Doppelpunkt) eingeleitet. Perl unterstützt die folgenden Attribute:

`lvalue` Die Subroutine gibt eine Variable zurück, die zugewiesen werden kann.

`Methode` Die Subroutine ist eine Methode.

Eine Subroutine kann auf verschiedene Arten aufgerufen werden.

Name ([*Parameter*])
: Die übliche Form. Die Parameter werden mittels Referenz als Array `@_` übergeben.

&Name ([*Parameter*])
: Prototypspezifikationen werden, falls vorhanden, ignoriert.

&Name
: Das aktuelle `@_`-Array wird direkt an die aufzurufende Subroutine weitergegeben.

Name [*Argumente*]
: Wenn die Subroutine deklariert oder definiert wurde, kann sie wie ein fest integrierter Operator ohne Klammern aufgerufen werden.

caller [*Ausdruck*]
: Gibt eine Liste (*package*, *file*, *line*) für einen bestimmten Subroutinenaufruf zurück. `caller` gibt diese Information für die aktuelle Subroutine zurück. Ist ein *Ausdruck* vorhanden, wird eine erweiterte Liste zurückgegeben: `caller(0)` für diese Subroutine, `caller(1)` für den Aufrufer dieser Subroutine usw. Siehe die nachfolgende Tabelle.

  Gibt falsch zurück, wenn es keinen Aufrufer gibt.

  Elemente der erweiterten Ergebnisliste:

Index	Name	Beschreibung
0	package	Das aufrufende Paket
1	filename	Der Dateiname
2	line	Die Zeilennummer
3	subroutine	Der Name der Subroutine
4	hasargs	Wahr, wenn Argumente übergeben wurden
5	wantarray	Kontext des Aufrufs
6	evaltext	Text, falls der Aufruf durch **eval** erfolgt
7	is_require	Aufruf über **use** oder **require**

Index	Name	Beschreibung
8	hints	Pragma-Hints
9	bitmask	Pragma-Hints
10	hinthash	Pragma-Hints

defined *&Name*
: Prüft, ob die Subroutine definiert wurde (einen Body besitzt).

do *Name Liste*
: Veraltete Form von *&Name*`.`

exists *&Name*
: Wahr, wenn die benannte Subroutine durch einen Body oder eine Vorwärtsdeklaration deklariert wurde.

goto *&Name*
: Ersetzt die aktuelle Subroutine durch einen Aufruf von *Name*.

return [*Ausdruck*]
: Kehrt mit dem angegebenen Wert aus einer Subroutine, einem **eval** oder **do** *Datei* zurück. Ohne *Ausdruck* wird im skalaren Kontext *undef* und im Listenkontext eine leere Liste zurückgegeben.

perlsub, perlmod

Prototypen

Prototyp	Bedeutung
`$`	Erzwingt skalaren Kontext
`@ %`	Erzwingt Listenkontext; alle verbliebenen Argumente werden verbraucht
`*`	Akzeptiert einen reinen Namen, eine Konstante, einen skalaren Ausdruck, ein Typeglob oder eine Referenz auf ein Typeglob
`&`	Verlangt eine Referenz auf Programmcode (Subroutine)
`\$`	Das Argument muss mit einem `$` beginnen

Prototyp	Bedeutung
\@ \%	Das Argument muss mit einem @ bzw. % beginnen
* \&	Das Argument muss mit einem * bzw. & beginnen
\[...]	Das Argument muss mit einer der angegebenen Sigillen beginnen
	(Leere Klammern) Gibt die Subroutine einen festen Wert zurück, erfolgt ein Inlining des Codes
_	Verlangt einen skalaren Ausdruck. Voreingestellt mit $_
+	Verlangt ein Array, ein Hash oder eine Referenz
;	Zeigt an, dass die nachfolgenden Argumente optional sind

prototype *Name*

Gibt den Prototyp der Funktion als String zurück oder *undef*, wenn die Funktion keinen verwendbaren Prototyp besitzt.

Nutzen Sie CORE::*Name* für fest eingebaute Funktionen.

perlsub, perlmod

Spezielle Subroutinen

Spezielle Subroutinen werden vom Benutzer definiert und von Perl während der Verarbeitung des Programms aufgerufen. Sie können verwendet werden, um die Reihenfolge zu verändern, in der einzelne Teile eines Programms ausgeführt werden.

[**sub**] **AUTOLOAD** *Block*

Der Code im *Block* wird ausgeführt, wenn das Programm eine undefinierte Subroutine aufruft. $AUTOLOAD enthält den Namen der aufgerufenen Subroutine, und @_ enthält die Parameter.

[**sub**] **BEGIN** *Block*

Der Code im *Block* wird direkt ausgeführt, sobald die Kompilierung des Blocks abgeschlossen ist.

[**sub**] **CHECK** *Block*

CHECK-Blöcke werden in der umgekehrten Reihenfolge ihres Auftretens ausgeführt, sobald die Kompilierung des Programms abgeschlossen ist.

[**sub**] **END** *Block*

END-Blöcke werden in der umgekehrten Reihenfolge ihres Auftretens ausgeführt, wenn der Perl-Interpreter beendet wird. Innerhalb der **END**-Blöcke enthält `$?` den **exit**-Status, mit dem das Programm beendet wird.

[**sub**] **INIT** *Block*

INIT-Blöcke werden vom Perl-Interpreter unmittelbar vor der Ausführung des Programms verarbeitet.

[**sub**] **UNITCHECK** *Block*

UNITCHECK-Blöcke werden in der umgekehrten Reihenfolge ihres Auftretens ausgeführt, sobald die Kompilierung der Programmeinheit abgeschlossen ist.

`perlsub`, `perlmod`

Objektorientierte Programmierung

Ein *Objekt* ist eine Referenz (genauer gesagt, das Referenzziel), die weiß, welcher Klasse sie angehört.

Eine *Klasse* ist einfach ein Paket, das Methoden zur Verfügung stellt. Kann ein Paket eine Methode nicht zur Verfügung stellen, werden die in `@ISA` aufgeführten Basisklassen durchsucht (Tiefensuche).

Eine *Methode* ist einfach eine Subroutine, die einen Aufrufer (eine Objektreferenz oder – bei statischen Methoden – einen Paketnamen) als erstes Argument erwartet.

bless *Referenzziel* [, *Klassenname*]

Wandelt das *Referenzziel* in ein Objekt in *Klassenname* um (standardmäßig das aktuelle Paket). Gibt die Referenz zurück.

Aufrufer -> *Methode* [(*Parameter*)]
: Ruft die genannte Methode auf.

Methode Aufrufer [*Parameter*]
: Stellt eine alternative Möglichkeit zum Aufruf einer Methode über die manchmal nicht ganz eindeutige *indirekte Objektsyntax* zur Verfügung.

Siehe auch **ref** (Seite 76) und den nächsten Abschnitt.

perlobj, perlboot, perltoot, perltooc

Spezielle Klassen

Die spezielle Klasse UNIVERSAL enthält Methoden, die automatisch von allen anderen Klassen geerbt werden:

can *Methode*
: Gibt eine Referenz auf die Methode zurück, wenn deren Aufrufer sie besitzt, andernfalls *undef*.

DOES *Rolle*
: Prüft, ob das Objekt/die Klasse die angegebene Rolle übernimmt.

isa *Klasse*
: Gibt wahr zurück, wenn der Aufrufer diese *Klasse* ist oder irgendeiner von *Klasse* erbenden Klasse angehört.

VERSION [*benötigt*]
: Gibt die Version des Aufrufers zurück. Prüft die Version, wenn *benötigt* angegeben wurde.

Diese Methoden können auch wie normale Funktionen verwendet werden, z.B. `UNIVERSAL::isa($c,Math::Complex::)`.

Das Pseudopaket CORE ermöglicht den Zugriff auf alle in Perl fest integrierten Funktionen, auch wenn diese überschrieben wurden.

Das Pseudopaket SUPER ermöglicht den Zugriff auf eine Methode der Basisklasse, ohne angeben zu müssen, welche Klasse diese Methode definiert hat. Das ist nur innerhalb einer Methode sinnvoll.

Arithmetische Funktionen

abs *Ausdruck*†
: Gibt den Absolutwert des Operanden zurück.

atan2 *y* **,** *x*
: Gibt den Arkus-Tangens von *y/x* im Bereich von $-\frac{\pi}{2}$ bis $+\frac{\pi}{2}$ zurück.

cos *Ausdruck*†
: Gibt den Kosinus von *Ausdruck* (in Bogenmaß) zurück.

exp *Ausdruck*†
: Gibt den Wert von *e* hoch *Ausdruck* zurück.

int *Ausdruck*†
: Gibt den ganzzahligen Teil von *Ausdruck* zurück.

log *Ausdruck*†
: Gibt den natürlichen Logarithmus (Basis *e*) von *Ausdruck* zurück.

rand [*Ausdruck*]
: Gibt eine Zufallszahl (als Dezimalbruch) zwischen 0 (inklusive) und dem Wert von *Ausdruck* (exklusive) zurück. Fehlt *Ausdruck*, wird standardmäßig der Wert 1 verwendet.

sin *Ausdruck*†
: Gibt den Sinus von *Ausdruck* (in Bogenmaß) zurück.

sqrt *Ausdruck*†
: Gibt die Quadratwurzel von *Ausdruck* zurück.

srand [*Ausdruck*]
: Initialisiert den vom **rand**-Operator verwendeten Zufallszahlengenerator.

time
: Gibt die Anzahl von Sekunden zurück, die seit dem Beginn der Epoche vergangen sind. (Die Definition der Epoche hängt vom jeweiligen System ab.) Geeignet für die Übergabe an **gmtime** und **localtime**.

Konvertierungsfunktionen

chr *Ausdruck*†

Gibt das Zeichen zurück, das durch den Dezimalwert *Ausdruck* repräsentiert wird.

gmtime [*Ausdruck*]

Im Listenkontext wandelt diese Funktion einen Zeitwert, der von der **time**-Funktion geliefert wird, in eine aus neun Elementen bestehende Liste um. Die Zeit wird dabei für die Zeitzone UTC (GMT) aufbereitet.

Im skalaren Kontext wird der Zeitwert in ein Ausgabeformat umgewandelt.

Fehlt der *Ausdruck*, wird die aktuelle Zeit verwendet.

Nutzen Sie das Standardmodul `Time::gmtime`, um auf die Elemente der Liste über deren Namen zugreifen zu können; siehe **localtime** weiter unten.

hex *Ausdruck*†

Gibt den Dezimalwert von *Ausdruck*, interpretiert als Hexadezimalstring, zurück. Der String kann, muss aber nicht mit `0x` beginnen. Weitere Konvertierungen finden Sie unter **oct** weiter unten.

localtime [*Ausdruck*]

Wie **gmtime**, verwendet jedoch die lokale Zeitzone.

Nutzen Sie das Standardmodul `Time::localtime`, um auf die Elemente der Liste über deren Namen zugreifen zu können:

Index	Name	Beschreibung
0	sec	Sekunden
1	min	Minuten
2	hour	Stunden
3	mday	Tag des Monats
4	mon	Monat, 0 = Januar

Index	Name	Beschreibung
5	year	Jahr seit 1900
6	wday	Wochentag, 0 = Sonntag
7	yday	Tag im Jahr, 0 = 1. Januar
8	isdst	Wahr während der Sommerzeit

oct *Ausdruck*†
: Gibt den Dezimalwert von *Ausdruck*, interpretiert als Oktalstring, zurück. Beginnt der String mit `0x`, wird er als Hexadezimalstring interpretiert. Beginnt er mit `0b`, wird er als Binärstring interpretiert.

ord *Ausdruck*†
: Gibt den numerischen Wert des ersten Zeichens von *Ausdruck* zurück.

vec *Ausdruck*, *Offset*, *Bits*
: Betrachtet den String *Ausdruck* als Vektor von vorzeichenlosen Integer-Werten mit jeweils *Bits* Bits und liefert den Dezimalwert des an *Offset* liegenden Elements zurück. *Bits* muss eine Zweierpotenz größer 0 sein. Der ganze Ausdruck kann auch einen Wert zugewiesen bekommen.

Strukturkonvertierung

pack *Template*, *Liste*
: Packt die Werte in *Liste* mithilfe von *Template* in eine Folge von Bytes. Gibt diese Folge in Form eines Strings zurück.

unpack *Template*, *Ausdruck*†
: Entpackt die Byte-Folge in *Ausdruck* mithilfe von *Template* in eine Liste.

Template ist eine Sequenz der folgenden Zeichen:

`a` / `A` Byte-String, mit Null-/Leerzeichen aufgefüllt

`b` / `B` Bit-String in aufsteigender/absteigender Reihenfolge

`c` / `C`	Byte-Wert mit/ohne Vorzeichen
`d` / `D`	double/long double im Maschinenformat (native)
`f` / `F`	float/Perl-internes float im Maschinenformat (native)
`h` / `H`	Hex-String, Low-Nibble/High-Nibble zuerst
`i` / `I`	Integer-Wert mit/ohne Vorzeichen
`j` / `J`	Perl-interner Integer-Wert mit/ohne Vorzeichen
`l` / `L`	Long-Wert mit/ohne Vorzeichen
`n` / `N`	Short/Long in Netzwerk-Byte-Ordnung (Big Endian)
`p` / `P`	Zeiger auf einen null-terminierten String/String fester Länge
`q` / `Q`	Quad-Wert mit/ohne Vorzeichen
`s` / `S`	Short-Wert mit/ohne Vorzeichen
`u` / `U`	Mit UUencode codierter String/Unicode UTF-8-Zeichencodierung
`v` / `V`	Short/Long in VAX-Byte-Ordnung (Little Endian)
`w`	BER-codierter Integer-Wert
`W`	Zeichenwert ohne Vorzeichen
`x` / `X`	Null-Byte (ein Byte vor)/ein Byte zurück
`Z`	Null-terminierter String
`.` / `@`	Auffüllen mit Null-Bytes bis zur absoluten/relativen Position

Die Größe der von `i` und `I` verwendeten Integer-Werte hängt von der Systemarchitektur ab. Nibbles, Bytes, Short-, Long- und Quad-Werte sind immer genau 4, 8, 16, 32 und 64 Bit groß. Den Zeichen `s`, `S`, `l` und `L` kann unmittelbar ein `!` folgen, um Short- und Long-Werte im Maschinenformat (native) anzuzeigen. Auf `x` und `X` kann unmittelbar ein `!` folgen, um die Ausrichtung festzulegen.

Jedem Zeichen (bzw. jeder in runden Klammern stehenden Gruppe von Zeichen) kann eine (optional zwischen [und] stehende)

Dezimalzahl folgen, die als Wiederholungszähler verwendet wird. Ein Sternchen (*) kennzeichnet alle verbliebenen Argumente. Ein zwischen [und] stehendes Template dient als Wiederholungszähler mit der gleichen Länge wie das gepackte Template. Mit < (Little-Endian) und > (Big-Endian) können Sie eine bestimmte Byte-Ordnung für ein Zeichen oder eine Gruppe erzwingen.

Beginnt das Template mit `U0`, erzwingen Sie ein Ergebnis in UTF-8, `C0` liefert Bytes. Voreingestellt ist UTF-8.

Steht vor dem Format ein *%n*, gibt **unpack** eine Prüfsumme mit *n* Bit zurück. *n* ist mit 16 voreingestellt.

Der Lesbarkeit halber dürfen Whitespaces im Template enthalten sein, und für Kommentare kann des #-Zeichen verwendet werden.

Ein besonderer Fall ist ein numerischer Zeichencode, gefolgt von einem Schrägstrich und einem String, z.B. `C/a`. In diesem Fall bestimmt der numerische Wert die Länge des String-Elements.

`q` und `Q` sind nur verfügbar, wenn Perl mit aktivierter 64-Bit-Unterstützung kompiliert wurde. `D` ist nur verfügbar, wenn Perl mit aktivierter long double-Unterstützung kompiliert wurde.

`perlpacktut`

String-Funktionen

chomp *Liste*†
: Entfernt `$/` (Seite 84) aus allen Elementen der Liste. Gibt die Anzahl (aller) entfernten Zeichen zurück.

chop *Liste*†
: Entfernt das letzte Zeichen aller Elemente der Liste. Gibt das zuletzt entfernte Zeichen zurück.

crypt *Klartext, Faktor*
: Verschlüsselt einen String (nicht umkehrbar).

eval *Ausdruck*†
: *Ausdruck* wird als Perl-Programm interpretiert und ausgeführt. Der Rückgabewert ist der Wert des zuletzt aus-

gewerteten Ausdrucks. Ist ein Syntax- oder Laufzeitfehler aufgetreten, gibt `eval` *undef* zurück, und `$@` enthält die Fehlermeldung. Beachten Sie auch **eval** auf Seite 75.

index *String*, *Substring* [, *Offset*]
: Gibt die Position von *Substring* in *String* an oder nach *Offset* zurück. Gibt die Position von –1 zurück, wenn der Substring nicht gefunden wurde.

lc *Ausdruck*†
: Wandelt *Ausdruck* in Kleinbuchstaben um. Siehe auch `\L` auf Seite 8.

lcfirst *Ausdruck*†
: Wandelt den ersten Buchstaben von *Ausdruck* in einen Kleinbuchstaben um. Siehe auch `\l` auf Seite 8.

length *Ausdruck*†
: Gibt die Länge von *Ausdruck* in Zeichen zurück.

quotemeta *Ausdruck*†
: Gibt *Ausdruck* als String zurück, bei dem alle Metazeichen für reguläre Ausdrücke in Quoting-Zeichen stehen. Siehe auch `\Q` auf Seite 8.

rindex *String*, *Substring* [, *Offset*]
: Gibt die Position des letzten Vorkommens von *Substring* in *String* an oder vor *Offset* zurück. Wurde der Substring nicht gefunden, wird –1 zurückgegeben.

substr *Ausdruck*, *Offset* [, *Länge* [, *Neuertext*]]
: Extrahiert aus *Ausdruck* an der Position *Offset* einen Substring mit der angegebenen *Länge* und gibt ihn zurück. Bei einem negativen *Offset* wird vom Ende des Strings gezählt. Bei einer negativen *Länge* werden entsprechend viele Zeichen vom Ende des Strings weggelassen. Ersetzt den Substring durch *Neuertext* (falls vorhanden). Der ganze Ausdruck kann andererseits auch einen Wert zugewiesen bekommen.

uc *Ausdruck*†
: Wandelt *Ausdruck* in Großbuchstaben um. Siehe auch `\U` auf Seite 8.

ucfirst *Ausdruck*†

Wandelt den ersten Buchstaben von *Ausdruck* in einen Großbuchstaben um. Siehe auch \u auf Seite 8.

Array- und Listenfunktionen

defined *Ausdruck*†

Keine spezielle Array-Funktion, bietet aber eine bequeme Möglichkeit, um zu prüfen, ob ein Array-Element einen definierten Wert besitzt.

delete [**local**] *elt*

delete [**local**] *@array*[*index1*, ...]

elt muss ein Array-Element wie *$array*[*index*] oder *Ausdruck*->[*index*] sein. Entfernt die angegebenen Elemente aus dem Array. Mit **local** erfolgt das Entfernen lokal zum umschließenden Block. Gibt Aliase auf den/die entfernten Wert(e) zurück. Das Entfernen der/des letzten Elemente(s) kürzt das Array entsprechend.

each *@ array*

Gibt im Listenkontext eine aus zwei Elementen bestehende Liste zurück, die den Index und einen Alias auf das nächste Element im Array enthält. Im skalaren Kontext wird nur der Index zurückgegeben. Nachdem alle Werte zurückgegeben wurden, wird eine leere Liste zurückgegeben. Der daraufhin folgende Aufruf von **each** startet die Iteration dann erneut. Ein Aufruf von **keys** oder **values** setzt die Iteration zurück.

exists *elt*

elt muss ein Array-Element sein (siehe **delete** weiter oben). Prüft, ob das angegebene Array-Element existiert.

grep *Ausdruck, Liste*

grep *block Liste*

Evaluiert *Ausdruck* oder *Block* für jedes Element der Liste. Dabei wird $_ lokal so gesetzt, dass es auf dieses Element verweist. Im Listenkontext wird eine *Liste* der Elemente zurückgegeben, für die *Ausdruck* oder *Block* wahr ist. Im skalaren Kontext wird die Anzahl dieser Elemente zurückgegeben.

join *Ausdruck, Liste*

Verbindet die einzelnen Elemente der *Liste* zu einem einzigen String, dessen Felder durch den Wert von *Ausdruck* getrennt sind. Zurückgegeben wird der String.

keys @*Array*

Im Listenkontext wird eine Liste aller Schlüssel im benannten Array zurückgegeben. Im skalaren Kontext wird die Anzahl der Elemente im Array zurückgeliefert.

map *Ausdruck, Liste*

map *Block Liste*

Evaluiert *Ausdruck* oder *Block* für jedes Element der Liste. Dabei wird $_ lokal so gesetzt, dass es auf dieses Element verweist. Gibt die Liste der Ergebnisse zurück.

pop [@*Array*]

Entfernt den letzten Wert aus dem Array und gibt ihn zurück. Wird @*Array* nicht angegeben, wird innerhalb einer Subroutine aus @_, andernfalls aus @ARGV entfernt.

push @*Array, Liste*

Hängt die Werte der Liste an das Ende des Arrays an. Gibt die neue Länge des Arrays zurück.

reverse *Liste*

Im Listenkontext wird die Liste in umgekehrter Reihenfolge zurückgegeben. Im skalaren Kontext werden die Listenelemente zu einem String verkettet, und der umgekehrte String wird zurückgegeben.

scalar @*Array*

Gibt die Anzahl der Elemente im Array zurück.

shift [*@Array*]

Verschiebt den ersten Wert aus dem Array und gibt ihn zurück. Dabei wird das Array um ein Element gekürzt, und alles wird nach unten verschoben. Wird *@Array* nicht angegeben, wird innerhalb einer Subroutine aus `@_` verschoben, andernfalls aus `@ARGV`.

sort [*Subroutine*] *Liste*

Sortiert die *Liste* und gibt den sortierten Listenwert zurück. Die *Subroutine* muss, wenn sie angegeben wird, kleiner null, null oder größer null zurückgeben, je nachdem, wie die Elemente der Liste geordnet werden sollen.

Subroutine kann der Name einer benutzerdefinierten Routine (bzw. eine diesen Namen enthaltende Variable) oder ein *Block* sein. Wurde die Subroutine mit dem Prototyp (`$$`) deklariert, werden die zu vergleichenden Werte als normale Parameter übergeben. Andernfalls stehen sie der Routine als globale Paketvariablen `$a` und `$b` zur Verfügung.

splice *@Array*, *Offset* [, *Länge* [, *Liste*]]

Entfernt die durch *Offset* und *Länge* gekennzeichneten Elemente aus *@Array* und ersetzt sie durch die *Liste* (falls angegeben). Gibt die entfernten Elemente zurück. Bei einem negativen *Offset* wird vom Ende des Arrays gezählt. Ist die *Länge* negativ, werden Elemente am Ende des Arrays ausgelassen.

split [*Muster* [, *Ausdruck*† [, *Limit*]]]

Verwendet das *Muster*, um *Ausdruck* (einen String) in eine Liste von Strings zu zerlegen, und gibt diese zurück. Bei einem positiven *Limit* wird der String maximal in diese Anzahl von Feldern zerlegt. Ein negativer Wert gibt alle Felder an. Fehlt das *Limit* oder hat es den Wert 0, werden führende leere Felder nicht zurückgegeben. Wird das *Muster* weggelassen, wird an Whitespace zerlegt (nachdem die führenden Whitespaces entfernt wurden). Außerhalb des Listenkontexts wird die Anzahl der Felder zurückgegeben und nach `@_` zerlegt.

unshift @*Array*, *Liste*
: Stellt die *Liste* an den Anfang des Arrays und gibt die neue Länge des Arrays zurück.

values @*Array*
: Gibt eine Liste zurück, die Aliase auf alle Werte des angegebenen Arrays enthält.

each, **keys**, **pop**, **push**, **shift**, **splice**, **unshift** und **values** können als erstes Argument eine Referenz auf ein Array übergeben – experimentell.

Hash-Funktionen

defined *Ausdruck*†
: Keine spezielle Hash-Funktion, bietet aber eine bequeme Möglichkeit, um zu prüfen, ob ein Hash-Element einen definierten Wert besitzt.

delete [**local**] *elt*

delete [**local**] @*hash*{*key1*, *key2*, ... }
: *elt* muss ein Hash-Element wie $*hash*{*key*} oder *Ausdruck*->{*key*} sein. Entfernt die angegebenen Elemente aus dem Hash. Mit **local** erfolgt das Entfernen lokal zum umschließenden Block. Gibt Aliase auf den/die entfernten Wert(e) zurück.

each *%hash*
: Gibt im Listenkontext eine aus zwei Elementen bestehende Liste zurück, die den Schlüssel und einen Alias auf das nächste Element im Hash enthält. Im skalaren Kontext wird nur der Schlüssel zurückgegeben. Nachdem alle Werte zurückgegeben wurden, wird eine leere Liste zurückgegeben. Der daraufhin folgende Aufruf von **each** startet die Iteration dann erneut. Ein Aufruf von **keys** oder **values** setzt die Iteration zurück.

exists *elt*

elt muss ein Array-Element sein (siehe **delete** weiter oben). Prüft, ob der angegebene Hash-Schlüssel existiert.

keys *%Hash*

Im Listenkontext wird eine Liste aller Schlüssel im benannten Hash zurückgegeben. Im skalaren Kontext wird die Anzahl der Elemente im Hash zurückgeliefert. Auch eine Zuweisung ist möglich, um den Hash im Vorfeld zu vergrößern.

scalar *%Hash*

Gibt wahr zurück, wenn Schlüssel im Hash definiert sind.

values *%Hash*

Gibt eine Liste zurück, die Aliase auf alle Werte des angegebenen Hashs enthält.

each, **keys** und **values** geben die Elemente in einer scheinbar zufälligen Reihenfolge zurück, doch diese ist für alle gleich.

each, **keys** und **values** können als erstes Argument eine Referenz auf ein Array übergeben – experimentell.

Smartmatching

Bitte beachten Sie: Smartmatching ist ein relativ neues Feature. Verschiedene Verhaltensaspekte können sich in nachfolgenden Perl-Releases noch ändern.

Ausdruck1 ~~ *Ausdruck2*

Smartmatching von *Ausdruck2* gegen *Ausdruck1*.

when (*Ausdruck*)

In den meisten Fällen erfolgt das Smartmatching von $_ gegen *Ausdruck*.

In anderen Fällen wird *Ausdruck* als boolescher Ausdruck betrachtet.

Das Smartmatch-Verhalten hängt davon ab, welcher Art die Argumente sind. Das Verhalten wird durch die folgende Tabelle bestimmt: Die erste passende Zeile bestimmt das Matching-Verhalten (d.h., es wird meist durch den Typ des rechten Operanden bestimmt). Beachten Sie, dass das Smartmatching implizit alle nicht über `bless` gebundenen Hash- oder Array-Referenzen dereferenziert, d.h., es gelten in diesen Fällen die *`Hash`*- und *`Array`*-Einträge. Bei über `bless` gebundenen Referenzen gelten die *`Objekt`*-Einträge.

Links	Rechts	Implizierter Matching-Typ
beliebig	undef	»Undefiniertheit«
beliebig	Objekt	Versucht ein ~~-Überladen des Objekts oder bricht ab
Hash	CodeRef	wahr, wenn Subroutine für jeden Schlüssel wahr zurückliefert; wahr, wenn Hash leer ist
Array	CodeRef	wahr, wenn Subroutine für jedes Element wahr zurückliefert; wahr, wenn Array leer ist
Any	CodeRef	wahr, wenn Subroutine für <Any> wahr zurückliefert
Hash	Hash	Wahr, wenn jeder Schlüssel in beiden Hashes gefunden wird
Array	Hash	Schnittmenge der Hash-Schlüssel
Regex	Hash	grep der Hash-Schlüssel
undef	Hash	Immer falsch
beliebig	Hash	Existenz eines Hash-Eintrags
Hash	Array	Schnittmenge der Hash-Schlüssel
Array	Array	Smartmatching der entsprechenden Elemente
Regex	Array	grep auf Array
undef	Array	Array enthält undef
beliebig	Array	Matching eines Array-Elements
Hash	Regex	grep auf Hash-Schlüssel
Array	Regex	grep auf Array
beliebig	Regex	Mustererkennung.
Objekt	beliebig	Versucht ein ~~-Überladen des Objekts oder Fallback
beliebig	Zahl	Numerische Gleichheit

Links	Rechts	Implizierter Matching-Typ
Num	zahlenartig	Numerische Gleichheit, wenn es wie eine Zahl aussieht
undef	beliebig	»Undefiniertheit«
Any	beliebig	String-Gleichheit

perlop (siehe »Smartmatch Operator«)

Regex-Muster

Mithilfe des Operators **qr** können Sie Muster aufbauen.

qr */pattern/* [*Modifier*]
: Erzeugt ein Muster.

Muster können beim Einsatz von Such- und Ersetzungsfunktionen auch implizit erzeugt werden.

Modifier können wie angegeben, aber auch innerhalb des *Musters* mittels (?*Modifier*) genutzt werden. In diesem Fall können die Modifier imsx ein- und ausgeschaltet werden.

`i`
: Ignoriert die Groß-/Kleinschreibung bei der Suche

`m`
: Mehrzeilenmodus (multiline): `^` und `$` erkennen eingebettete Newline-Zeichen

`o`
: Interpoliert Variablen nur einmal

`p`
: Bewahrt `${^PREMATCH}`,`${^MATCH}` und `${^POSTMATCH}`

`s`
: Einzeilenmodus: `.` erkennt eingebettete Newline-Zeichen

`x`
: Erlaubt Whitespace und Kommentare

`^`
: Bei `(?^ ... )` wird die Systemvoreinstellung `d-imsx` verwendet

Diese Modifier schließen sich gegenseitig aus:

`a`
: Erzwingt striktes ASCII-Matching noch strikter, wenn Sie es wiederholen

`d`
: Semantik hängt von anderen Einstellungen ab

`l` Behandelt das Muster mit locale-Eigenschaften

`u` Behandelt das Muster mit vollständiger Unicode-Semantik

? `perlre`, `perlretut`, `perlrequick`, `perlunicode`

Die Elemente, aus denen das Muster besteht, basieren auf *regulären Ausdrücken*, enthalten heutzutage aber viele Erweiterungen.

Jedes Zeichen steht für sich selbst, solange es sich nicht um eines der Spezialzeichen `+?.*^$()[{|\.` handelt. Die spezielle Bedeutung dieser Zeichen kann mit einem vorangestellten `\` aufgehoben werden.

`.` Steht für ein beliebiges Zeichen außer Newline. Im einzeiligen Modus werden auch Newline-Zeichen erkannt.

`( ... )` Gruppiert eine Reihe von Musterelementen zu einem einzigen Element. Der dieser Gruppe entsprechende Text wird zur späteren Verwendung festgehalten. Darüber hinaus wird er direkt `$^N` zugewiesen, um während des Matchings verwendet werden zu können, etwa in einem `(?{ ... })`.

`^` Steht für den Anfang des Ziels. Im mehrzeiligen Modus wird auch jedes Newline-Zeichen erkannt.

`$` Erkennt das Zeilenende bzw. das Zeichen unmittelbar vor dem letzten Newline. Im mehrzeiligen Modus wird auch das Zeichen unmittelbar vor *jedem* Newline erkannt.

`[...]` Bezeichnet eine Klasse von zu erkennenden Zeichen. `[^...]` negiert diese Klasse.

`...|...|...`
Erkennt (von links nach rechts) die erste der angegebenen Alternativen.

Erweiterte Muster:

`(?#` *`Text`* `)`
Kommentar.

`(?` [*`Modifier`*] `:` *`Muster`* `)`
Verhält sich wie (*`Muster`*), nur dass der erkannte Text nicht festgehalten wird. Die Modifier `i`, `m`, `s` und `x` können

deaktiviert werden, indem man den/dem Buchstaben ein Minuszeichen voranstellt, etwa `si-xm`.

(?= *Muster*)
: Positive Lookahead-Bedingung (null Zeichen).

(?! *Muster*)
: Negative Lookahead-Bedingung (null Zeichen).

(?<= *Muster*)
: Positive Lookbehind-Bedingung (null Zeichen). Der Einsatz von `\K` ist oft besser und einfacher.

(?<! *Muster*)
: Negative Lookbehind-Bedingung (null Zeichen).

(?<*Name*> ...) or (?'*Name*' ...)
: Gruppiert eine Folge von Musterelementen. Der durch die Gruppe erkannte Text wird zur späteren Verwendung festgehalten.

(?| *Muster*)
: Capture-Gruppen werden in jeder Verzweigung des Musters vom gleichen Anfangspunkt aus nummeriert.

(?{ *code* })
: Führt Perl-Code während des Matchings aus. Wird immer bei null Zeichen ausgeführt. Kann als Bedingung bei einer Musterwahl verwendet werden. Wenn nicht, steht das Ergebnis des ausgeführten *code* in `$^R`.

(??{ *code* })
: Führt Perl-Code während des Matchings aus. Interpretiert das Ergebnis als Muster.

(?> *Muster*)
: Wie (?: *Muster*), verhindert aber ein Backtracking.

(?(*Bedingung*) *Musterwahr* [|*Musterfalsch*])
: Wählt ein Muster in Abhängigkeit von der Bedingung. *Bedingung* kann die Nummer eines geklammerten Submusters, der <*Name*> eines Submusters oder eine der oben genannten Lookahead-, Lookbehind-oder Evaluierungsannahmen sein.

Mit R können Sie die Rekursion kontrollieren: (R), (R*Nummer*), (R&*Name*) und so weiter.

(?(DEFINE) *Muster*)
: Das Muster wird evaluiert, aber nicht ausgeführt. Damit lassen sich benannte Submuster definieren, die sich von den (?&*Name*)-Konstrukten verwenden lassen.

(? *Nummer*)
: Rekursion in das Muster der Capture-Gruppe *Nummer*. 0 oder R steht für das gesamte Muster, +1 für die nächste Capture-Gruppe, -1 für die vorherige und so weiter.

(?& *Name*) oder (?P> *Name*)
: Rekursion in das durch *Name* bezeichnete Muster.

(? *Modifier*)
: Eingebetteter Mustererkennungsmodifier. *Modifier* kann eines oder mehrere der Zeichen i, m, s oder x sein. Modifier können deaktiviert werden, indem man den/dem Buchstaben ein Minuszeichen voranstellt, z.B. (?si-xm). Ein führendes ^ kehrt zu den Voreinstellungen zurück.

Mit einer speziellen Gruppe von Mustern können Sie das Backtracking kontrollieren.

(*ACCEPT)
: Experimentell. Beendet das Matching und signalisiert Erfolg.

(*COMMIT)
: Matching-Fehler beim Backtracking in dieses Muster.

(*FAIL) (*F)
: Beendet das Matching und signalisiert Misserfolg.

(* [MARK] : [*Name*])
: Markiert eine Position, die später von SKIP verwendet wird. Siehe auch die Spezialvariablen $REGMARK und $REGERROR auf Seite 73.

(*PRUNE [:*Name*])
: Beendet das Matching beim Backtracking in dieses Muster.

(*SKIP [:*Name*])
: Ähnlich wie PRUNE.

(*THEN [:*Name*])
: Ähnlich wie PRUNE. Innerhalb einer Alternierung wird beim Backtracking zur nächsten Alternative gewechselt.

Quantifizierte Submuster werden so oft wie möglich erkannt. Folgt den Mustern ein ?, wird die minimale Anzahl erkannt. Folgt ihnen ein +, werden sie so oft wie möglich erkannt, und es erfolgt kein Backtracking. Die folgenden Mengenmodifier (»Quantifier«) stehen zur Verfügung:

\+
: Erkennt das vorstehende Musterelement einmal oder mehrmals.

?
: Erkennt ein Element nicht oder einmal.

*
: Erkennt ein Element nicht oder einmal.

{*n*,*m*}
: Gibt die minimale *n* und maximale *m* Anzahl von Treffern an. {*n*} bedeutet genau *n*-mal; {*n*,} bedeutet mindestens *n*-mal.

Muster werden verarbeitet, als wären sie in Anführungszeichen stehende Strings. Die normalen String-Escapes haben also ihre übliche Bedeutung (siehe Seite 7). Eine Ausnahme bildet \b, das für Wortgrenzen (word boundaries) steht, in Zeichenklassen aber wieder das Backspace-Zeichen repräsentiert.

Ein \ hebt jedwede spezielle Bedeutung eines nicht alphanumerischen Zeichens auf. Gleichzeitig macht es aus den meisten alphanumerischen Zeichen etwas Besonderes:

\1, \2, \3, ...
: Verweist innerhalb einer Erkennung auf mit () gruppierte Subausdrücke. \10 bedeutet \1 gefolgt von 0, es sei denn, das Muster hat mindestens 10 Subausdrücke. Siehe auch \g.

`\A` Erkennt den Anfang des Strings.

`\b` Erkennt Wortgrenzen; `\B` erkennt Nicht-Wortgrenzen.

`\C` Erkennt ein einzelnes 8-Bit-Byte.

`\d` Erkennt Ziffern; `\D` erkennt Nicht-Ziffern.

`\g` *n* `\g{` *n* `}`
Wie `\`*n*. `\g{-1}` steht für die vorige Gruppe usw.

`\g{` *Name* `}`
Der erkannte Subausdruck (`?<`*Name*`>` ...).

`\G` Erkennt die Stelle, an der die vorherige Suche mit einem g aufgehört hat.

`\h` Passt auf ein horizontales Leerzeichen. Komplement ist `\H`.

`\k<` *Name* `>` `\k'` *Name* `'` `\k{` *Name* `}`
Varianten für `\g{`*Name*`}`.

`\K` Verwirft bei Substitutionen alles bisher Erkannte.

`\N` Erkennt alles außer Newline.

`\p` *Eigenschaft*
Erkennt eine benannte Eigenschaft. `\P`*Eigenschaft* erkennt Nicht-*p*. Verwenden Sie `\p{`*Eigenschaft*`}` für Namen mit mehr als einem Buchstaben.

`\R` Erkennt generischen Zeilenumbruch (linebreak).

`\s` Erkennt Whitespace. `\S` erkennt Nicht-Whitespace.

`\X` Erkennt eine erweiterte »kombinierte Zeichensequenz« in Unicode.

`\v` Passt auf ein vertikales Leerzeichen. Komplement ist `\V`.

`\w` Erkennt alphanumerische Zeichen und den Unterstrich (_).`\W` erkennt Nicht-`\w`.

`\Z` Erkennt das Ende des Strings oder das Zeichen unmittelbar vor dem Newline-Zeichen am Ende des Strings.

`\z` Erkennt das physikalische Ende des Strings.

`\1` und höher, `\d`, `\D`, `\p`, `\P`, `\s`, `\S`, `\w` und `\W` können innerhalb und außerhalb von Zeichenklassen verwendet werden.

POSIX-artige Klassen wie `[[:word:]]` werden innerhalb von Zeichenklassen verwendet. Nachfolgend sehen Sie die POSIX-Klassen und ihre Unicode-Eigenschaftsnamen. Die zweite Eigenschaft gilt, wenn der Modifier `a` aktiv ist.

`[:alpha:]` `\p{XPosixAlpha}` `\p{PosixAlpha}`
: Erkennt ein Zeichen des Alphabets.

`[:alnum:]` `\p{XPosixAlnum}` `\p{PosixAlnum}`
: Erkennt ein alphanumerisches Zeichen.

`[:ascii:]` `\p{ASCII}` `\p{ASCII}`
: Erkennt ein ASCII-Zeichen.

`[:blank:]` `\p{XPosixSpace}` `\p{PosixSpace}`
: Erkennt ein Whitespace-Zeichen. Nahezu wie `\s`.

`[:cntrl:]` `\p{XPosixCntrl}` `\p{PosixCntrl}`
: Erkennt ein Control-Zeichen.

`[:digit:]` `\p{XPosixDigit}` `\p{PosixDigit}`
: Erkennt eine Ziffer (wie `\d`).

`[:graph:]` `\p{XPosixGraph}` `\p{PosixGraph}`
: Erkennt ein alphanumerisches Zeichen oder ein Interpunktionszeichen.

`[:lower:]` `\p{XPosixLower}` `\p{PosixLower}`
: Erkennt einen Kleinbuchstaben.

`[:print:]` `\p{XPosixPrint}` `\p{PosixPrint}`
: Erkennt ein alphanumerisches Zeichen oder ein Interpunktionszeichen oder ein Leerzeichen.

`[:punct:]` `\p{XPosixPunct}` `\p{PosixPunct}`
: Erkennt ein Interpunktionszeichen.

`[:space:]` `\p{XPosixSpace}` `\p{PosixSpace}`
: Erkennt ein Whitespace-Zeichen. Nahezu wie `\s`.

`[:upper:]` `\p{XPosixUpper}` `\p{PosixUpper}`
: Erkennt einen Großbuchstaben.

`[:word:]` `\p{XPosixWord}` `\p{PosixWord}`
: Erkennt ein Wortzeichen (wie `\w`).

`[:xdigit:]` `\p{XPosixXDigit}` `\p{PosixXDigit}`
: Erkennt eine hexadezimale Ziffer.

Die Äquivalente zu `\s` sind `\p{XPerlSpace}` und `\p{PerlSpace}`.

Die POSIX-Klassen können mit einem `^` negiert werden, etwa `[:^print:]`, die benannten Eigenschaften mit `\P`, z.B. `\P{PosixPrint}`.

Siehe auch `$1` ... `$9`, `$+`, `$``, `$&`, `$'`, `$^R` und `$^N` auf Seite 88, `@-`, `@+` auf Seite 89 sowie `%+` und `%-` auf Seite 91.

7 `perlre`, `perlretut`, `perlrequick`, `perlunicode`

Such- und Ersetzungsfunktionen

[*Ausdruck*`=~`] [**m**] */Muster/* [*modifiers*]
: Sucht in *Ausdruck* (Default `$_`) nach einem Muster.

  Für `=~` kann auch die Negationsform `!~` verwendet werden, die wahr zurückgibt, wenn `=~` falsch zurückgeben würde. Das gilt auch für den umgekehrten Fall.

  Nach einer erfolgreichen Erkennung werden die folgenden speziellen Variablen gesetzt:

`$&`	Der erkannte String.
`$``	Der String vor dem erkannten String.
`$'`	Der String nach dem erkannten String.
`$1`	Der erste in Klammern stehende Subausdruck, der erkannt wird. `$2` ist der zweite usw.
`$+`	Der letzte erkannte Subausdruck.
`@-`	Die Start-Offsets der Treffer und Subtreffer.
`@+`	Die dazugehörigen End-Offsets.
`%+`	Erkannte benannte Submuster.
`%-`	Alle benannten Submuster.

Im Listenkontext wird eine Liste zurückgegeben, die aus den durch die Klammern erkannten Subausdrücken besteht, d.h. (`$1,$2,$3, ...`).

`c` Ermöglicht (zusammen mit `g`) eine Fortsetzung der Suche.

`g` Ermöglicht eine Fortsetzung der Suche.

Ist `Muster` leer, wird das letzte Muster einer vorangegangenen erfolgreichen **`m//`**- oder **`s///`**-Operation verwendet.

Mit `g` kann das gefundene Muster im skalaren Kontext als Iterator verwendet werden. Der Iterator wird bei einem Fehler zurückgesetzt, wenn nicht auch `c` angegeben wird.

[*Ausdruck*`=~`] **`m?`***Muster*`?` [*modifiers*]
Genau wie bei der Suche mit */Muster/*, allerdings wird das Muster hier zwischen den Aufrufen des **`reset`**-Operators nur einmal erkannt.

[`$`*var*`=~`] **`s/`***Muster*`/`*Neuertext*`/` [*Modifier*]
Durchsucht den String *var* (standardmäßig `$_`) nach einem Muster und ersetzt diesen Teil (wenn er gefunden wird) durch den Ersetzungstext.

Bei Erfolg werden die bei **`m//`** beschriebenen speziellen Variablen entsprechend gesetzt, und die Anzahl der Ersetzungen wird zurückgegeben bzw. das modifizierte Ergebnis, wenn der Modifier `r` angegeben wurde. Anderenfalls wird falsch zurückgegeben.

Optionale Modifier sind (wie im vorigen Abschnitt beschrieben) `adilmopsux`. Zusätzliche Modifier sind:

`g` Ersetzt alle Vorkommen des Musters.

`e` Evaluiert `Neuertext` als Perl-Ausdruck.

`ee` Evaluiert zweimal usw.

`r` Ändert `var` nicht, sondern gibt das Ergebnis der Ersetzung zurück.

Ist *Muster* leer, wird das letzte Muster einer vorangegangenen erfolgreichen **m**//- oder **s**///-Operation verwendet.

[$*var*=~] **tr**/*search*/*replacement*/ [*modifiers*]
Übersetzt alle Vorkommen von in der Suchliste stehenden Zeichen in die entsprechenden Zeichen der Ersetzungsliste. Zurückgegeben wird, mit Ausnahme des Modifiers r (s.u.), die Anzahl der ersetzten Zeichen.

Optionale Modifier sind:

c	Ergänzt die Suchliste.
d	Entfernt alle in der Suchliste vorkommenden Zeichen, die kein entsprechendes Zeichen in der Ersetzungsliste besitzen.
r	Verändert var nicht, sondern gibt das Ergebnis der Ersetzung zurück.
s	Ersetzt alle Sequenzen von Zeichen, die in das gleiche Zielzeichen umgewandelt werden, durch ein einzelnes Vorkommen dieses Zeichens.

[$*var*=~] **y**/*search*/*Ersetzungsliste*/ [*Modifier*]
Mit **tr** identisch.

Ist die rechte Seite von =~ oder !~ ein Ausdruck statt ein Suchmuster, eine Substitution oder eine Übersetzung und ist dessen Wert nicht das Ergebnis eines **qr**-Operators, wird der Ausdruck als String interpretiert und während der Laufzeit in ein Suchmuster kompiliert.

Quoting-Regeln und String-Interpolation finden Sie auf Seite 6.

pos *Skalar*†
Gibt die Position zurück, an der die letzte /g-Suche in *Skalar* unterbrochen wurde. Verändert den Wert von \G, wenn eine Zuweisung erfolgt.

study *Skalar*†
Dazu gedacht, eine Folge von Mustererkennungen in einer Variablen zu optimieren. Macht faktisch nichts.

Dateioperationen

Funktionen, die mit Listen von Dateien arbeiten, geben die Anzahl der erfolgreich bearbeiteten Dateien zurück.

chmod *Liste*
: Ändert die Zugriffsrechte für eine Liste von Dateien. Das erste Element der Liste muss den numerischen Modus als Oktalzahl angeben (z.B. `0644`). Die Dateien können über den Namen oder über ein Handle angegeben werden.

chown *Liste*
: Ändert den Besitzer und die Gruppe einer Liste von Dateien. Die beiden ersten Elemente der Liste müssen die numerische UID und GID enthalten. Hat eines dieser Elemente den Wert –1, wird der entsprechende Wert nicht verändert. Die Dateien können über den Namen oder über ein Handle angegeben werden.

link *Altedatei* **,** *Neuedatei*
: Erzeugt einen neuen Dateinamen, der auf die alte Datei verweist.

lstat *Datei*†
: Wie **stat**. Ist die letzte Komponente des Dateinamens ein symbolischer Link, wird **stat** auf den Link und nicht auf die Datei ausgeführt. *Datei* kann ein Ausdruck sein, der zu einem Dateinamen evaluiert, oder _(um die letzte -l-Dateitestoperation bzw. den letzten **lstat**-Aufruf zu verwenden).

mkdir *Ausdruck*† [**,** *Rechte*]
: Erzeugt ein Verzeichnis mit den angegebenen Zugriffsrechten *Rechte* unter Berücksichtigung der aktuellen Umask. Ist *Rechte* eine Zahl, muss es sich um eine Oktalzahl handeln. Der Standardwert für *Rechte* lautet `0777`. Siehe auch **umask** auf Seite 72.

readlink *Ausdruck*†
: Gibt den Namen der Datei zurück, auf die durch den symbolischen Link in *Ausdruck* verwiesen wird.

rename *Alternname , Neuername*
: Ändert den Namen einer Datei.

rmdir *Ausdruck*†
: Entfernt das Verzeichnis, wenn es leer ist.

stat *Datei*†
: Gibt eine aus 13 Elementen bestehende Liste mit Dateiinformationen zurück. *Datei* kann ein Dateihandle sein, ein zu einem Dateinamen evaluierender Ausdruck oder _ (um die letzte Dateitestoperation bzw. den letzten **stat**-Aufruf zu verwenden). Gibt eine leere Liste zurück, wenn **stat** fehlschlägt.

  Verwenden Sie das Standardmodul `File::stat`, um auf die Elemente der Liste über deren Namen zugreifen zu können:

Index	Name	Beschreibung
0	dev	Gerätecode.
1	ino	Inode-Nummer.
2	mode	Typ- und Zugriffs-Flags.
3	nlink	Anzahl harter Links.
4	uid	Benutzer-ID des Besitzers.
5	gid	Gruppen-ID des Besitzers.
6	rdev	Gerätetyp.
7	size	Größe in Bytes.
8	atime	Timestamp des letzten Zugriffs.
9	mtime	Timestamp der letzten Modifikation.
10	ctime	Timestamp der letzten Statusänderung.
11	blksize	Blockgröße des Dateisystems.
12	blocks	Größe in Blöcken.

symlink *Altedatei , Neuedatei*
: Erzeugt einen neuen Dateinamen, der einen symbolischen Link auf den alten Dateinamen bildet.

truncate *Datei* **,** *Größe*
: Kürzt die Länge der *Datei* auf die angegebene *Größe*. *Datei* kann ein Dateiname oder ein Dateihandle sein.

unlink *Liste*†
: Löscht eine Liste von Dateien.

utime *Liste*
: Ändert die Zugriffs- und Modifikationszeiten. Die ersten beiden Elemente der Liste mussen die numerischen Zugriffs- und Modifikationszeiten enthalten. Sind beide Elemente undefiniert, wird die aktuelle Zeit verwendet.

  Die Inode-Änderungszeit wird auf die aktuelle Zeit gesetzt.

Dateitestoperatoren

Diese unären (monadischen) Operatoren verlangen ein Argument (einen Dateinamen oder ein Dateihandle) und prüfen, ob die entsprechende Datei eine bestimmte Bedingung erfüllt. Wird das Argument weggelassen, erfolgt der Test über `$_` (mit Ausnahme von `-t`, wo `STDIN` geprüft wird). Wird das spezielle Argument `_` (underscore, Unterstrich) übergeben, werden die Informationen des vorangegangenen Tests oder **stat**-Aufrufs verwendet. Dateitestoperatoren können gestapelt werden, z.B. `-r -w -x` *Datei*.

Beachten Sie auch das Pragma `filetest` auf Seite 23.

`-r -w -x` Datei kann vom effektiven UID/GID gelesen/geschrieben/ausgeführt werden.

`-R -W -X` Datei kann vom realen UID/GID gelesen/geschrieben/ausgeführt werden.

`-o -O` Datei gehört effektivem/realem UID.

`-e -z` Datei existiert/hat die Größe Null.

`-s` Datei existiert und hat eine Größe ungleich null. Zurückgegeben wird die Größe.

`-f -d` Datei ist eine reine Textdatei/ein Verzeichnis.

`-l -S -p` Datei ist ein symbolischer Link/ein Socket/eine benannte Pipe (FIFO).

`-b -c` Datei ist eine block-/zeichenorientierte Spezialdatei.

`-u -g -k` Datei hat das setuid-/setgid-/sticky-Bit gesetzt.

`-t` Das Dateihandle (standardmäßig `STDIN`) ist mit einem TTY verknüpft.

`-T -B` Datei ist eine Text-/Binärdatei. Diese Tests geben wahr bei einer leeren Datei zurück oder wenn sich die Datei während des Tests am Dateiende (EOF) befindet.

`-M -A -C` Modifikations-/Zugriffs-/Inode-Änderungszeit der Datei, gemessen in Tagen. Der zurückgegebene Wert spiegelt das Alter der Datei seit dem Start des Skripts wider. Siehe auch `$^T` auf Seite 87.

Eingabe und Ausgabe

Bei Eingabe-/Ausgabeoperationen kann *`Dateihandle`* ein über den **`open`**-Operator geöffnetes Dateihandle, ein vordefiniertes Dateihandle (z.B. `STDOUT`) oder eine skalare Variable sein, die zu einer Referenz auf einen Dateinamen oder ein Dateihandle evaluiert.

`<` *`Dateihandle`* `>`

Im skalaren Kontext wird ein einzelner Datensatz, üblicherweise eine Zeile, aus der über *`Dateihandle`* geöffneten Datei gelesen. Im Listenkontext wird der Rest der Datei eingelesen.

`< >`

`<ARGV>` Liest aus dem Eingabe-Stream, der durch die in `@ARGV` angegebenen Dateien gebildet wird. Es wird über die Standardeingabe gelesen, wenn keine Argumente übergeben werden.

`binmode` *`Dateihandle`* [**`,`** *`Schichten`*]

Sorgt dafür, dass die über *`Dateihandle`* geöffnete Datei in der angegebenen E/A-Schicht gelesen oder geschrieben

wird (Voreinstellung: :raw). Eine Liste der Standard-E/A-Schichten finden Sie auf Seite 64.

close [*Dateihandle*]
: Schließt das Dateihandle. Setzt $ zurück, wenn es eine Eingabedatei war. Fehlt *Dateihandle*, wird das aktuell gewählte Dateihandle geschlossen.

dbmclose *%Hash*
: Schließt die mit dem Hash verknüpfte Datei. Durch **untie** abgelöst, siehe Seite 64.

dbmopen *%Hash, Dbmname, Modus*
: Öffnet eine *dbm*-Datei und verknüpft sie mit dem Hash. Durch **tie** abgelöst, siehe Seite 76.

eof *Dateihandle*
: Gibt wahr zurück, wenn die nächste Leseoperation das Dateiende (EOF) anzeigt oder wenn die Datei nicht geöffnet ist.

eof
: Gibt den EOF-Status der zuletzt gelesenen Datei zurück.

eof ()
: Zeigt EOF für die Pseudodatei an, die aus den in der Kommandozeile übergebenen Dateien besteht.

fcntl *Dateihandle, Funktion,* $*Variable*
: Ruft systemabhängige Dateisteuerungsfunktionen auf.

fileno *Dateihandle*
: Liefert den Dateideskriptor für die gegebene (offene) Datei zurück.

flock *Dateihandle, Operation*
: Ruft eine systemabhängige Locking-Routine für die Datei auf. Die *Operation* ergibt sich aus der Addition eines oder mehrerer Werte oder LOCK_-Konstanten aus der Tabelle auf Seite 64.

getc [*Dateihandle*]
: Gibt das nächste Zeichen aus der Datei zurück bzw. einen Leerstring am Dateiende. Fehlt *Dateihandle*, wird aus STDIN gelesen.

ioctl *Dateihandle*, *Funktion*, $*Variable*

Ruft systemabhängige E/A-Steuerungsfunktionen auf.

open *Dateihandle* [, *ModusundName*]

open *Dateihandle* , *Modus* , *Name* [, ...]

Öffnet eine Datei und assoziiert sie mit *Dateihandle*. Ist *Dateihandle* eine nicht initialisierte skalare Variable, wird automatisch ein neues, eindeutiges Dateihandle angelegt.

ModusundName muss den Dateinamen enthalten, dem der Modus voranstehen muss, mit dem die Datei geöffnet werden soll. Fehlt *ModusundName*, muss eine globale (Paket-)Variable mit dem gleichen Namen wie *Dateihandle* den Modus und den Namen enthalten.

Die allgemeinen open-Modi finden Sie auf Seite 61.

In *ModusundName* kann - stellvertretend für die Standardeingabe bzw. -ausgabe verwendet werden. Whitespace ist erlaubt, was bedeutet, dass diese Form von **open** nicht einfach zum Öffnen von Dateien verwendet werden kann, deren Namen mit Whitespace beginnen oder enden.

Die Form mit drei oder mehr Argumenten erlaubt eine bessere Kontrolle über den Modus und den Dateinamen.

Wird der *Name* mit **undef** angegeben, wird eine anonyme temporäre Datei geöffnet. Ist der *Name* eine Referenz auf einen Skalar, wird der Inhalt des Skalars gelesen bzw. in den Skalar geschrieben.

Dem *Modus* kann eine Liste von E/A-Schichten angehängt werden, die auf das Handle angewandt werden. Eine Liste der Standard-E/A-Schichten finden Sie auf Seite 64.

pipe *Lesehandle*, *Schreibhandle*

Liefert ein Paar verbundener Pipes zurück. Ist eines der Handles eine nicht-initialisierte skalare Variable, wird automatisch ein neues, eindeutiges Dateihandle erzeugt.

print [*Dateihandle*] *Liste*†

Gibt die Elemente der *Liste* aus und wandelt sie bei Bedarf in Strings um. Fehlt *Dateihandle*, wird über das aktuell gewählte Ausgabehandle ausgegeben.

printf [*Dateihandle*] *Liste*†

Entspricht **print** *Dateihandle* **sprintf** *Liste*.

read *Dateihandle*, *$Variable*, *Länge* [, *Offset*]

Liest *Länge* Zeichen aus der Datei an *Offset* in die Variable ein. Gibt die Anzahl der tatsächlich gelesenen Zeichen zurück bzw. 0 bei EOF und *undef* bei einem Fehler.

readline *Ausdruck*†

Interne Funktion, die den < >-Operator implementiert.

readpipe *Ausdruck*†

Interne Funktion, die den **qx**-Operator implementiert. *Ausdruck* wird als Systembefehl ausgeführt.

say [*Dateihandle*] *Liste*†

Entspricht **print**, hängt aber implizit ein Newline an.

seek *Dateihandle*, *position*, *Bezugspunkt*

Positionierung der Datei an einer beliebigen Byte-Position. *Bezugspunkt* kann einer der Werte sein oder eine SEEK_-Konstante aus der Tabelle auf Seite 62.

select [*Dateihandle*]

Setzt ein neues Standard-Dateihandle für Ausgabeoperationen, wenn *Dateihandle* angegeben wird. Gibt das momentan gewählte Dateihandle zurück.

select *Rbits*, *Wbits*, *Nbits*, *Timeout*

Führt einen *select*-Systemaufruf mit den entsprechenden Parametern aus.

sprintf *Format*, *Liste*

Liefert einen String zurück, der aus der Formatierung einer (möglicherweise leeren) Liste von Werten entsteht. Eine vollständige Liste der Formatumwandlungen finden Sie im Abschnitt »Formatierte Ausgabe« auf Seite 64. Eine

Alternative zur formatierten Ausgabe wird im Abschnitt »Formate« auf Seite 67 beschrieben.

sysopen *Dateihandle*, *Pfad*, *Modus* [, *Rechte*]
: Führt einen *open*-Systemaufruf aus. Die möglichen Werte und Flag-Bits für *Modus* und die *Rechte* sind systemabhängig und stehen über das Standardmodul `Fcntl` zur Verfügung. Ist *Dateihandle* eine nicht-initialisierte skalare Variable, wird automatisch ein neues, eindeutiges Dateihandle erzeugt.

  Modus wird durch die Addition eines oder mehrerer Werte oder `O_`-Konstanten aus der Tablle auf Seite 63 gebildet.

sysread *Dateihandle*, $*Variable*, *Länge* [, *Offset*]
: Liest *Länge* Zeichen an $*Offsez* in *Variable* ein. Gibt die Anzahl der tatsächlich gelesenen Zeichen zurück bzw. 0 bei EOF und *undef* bei einem Fehler.

sysseek *Dateihandle*, *Position*, *Bezugspunkt*
: Positionierung der Datei an einer beliebigen Byte-Position. Zum Einsatz mit **sysread** und **syswrite** gedacht. *Bezugspunkt* kann ein Wert oder eine `SEEK_`-Konstante aus der Tabelle auf Seite 63 sein.

syswrite *Dateihandle*, *Skalar* [, *Länge* [, *Offset*]]
: Schreibt *Länge* Zeichen aus *Skalar* ab *Offset*. Gibt die Anzahl der tatsächlich geschriebenen Zeichen zurück oder *undef* bei einem Fehler.

tell [*Dateihandle*]
: Gibt die aktuelle Byte-Position innerhalb der Datei zurück. Fehlt *Dateihandle*, wird von der zuletzt gelesenen Datei ausgegangen.

open-Modi

Die folgenden Modi sind für alle Formen von **open** möglich:

`<`
: Reine Eingabedatei. Voreinstellung bei leerem Modus.

`>`
: Reine Ausgabedatei. Die Datei wird nach Bedarf angelegt oder gekürzt.

>> Reine Ausgabedatei. Die Datei wird nach Bedarf angelegt oder gekürzt.

+< Lese-/Schreibzugriff.

+> Schreib-/Lesezugriff.

+>> Lese-/Append-Zugriff.

Auf diese Modi kann ein `&` folgen, um ein bereits geöffnetes Dateihandle oder – bei einem numerischen Wert – einen Dateideskriptor zu duplizieren. Verwenden Sie `&=` mit einem numerischen Argument, um einen Alias für einen bereits geöffneten Dateideskriptor zu erzeugen.

Die Modi für die mit zwei Argumenten arbeitende Form von **open** umfassen:

| Öffnet eine Pipe, um von einem Befehl zu lesen oder zu einem Befehl zu schreiben.

|- Fork, bei dem die Datei mit der Standardeingabe des Child-Prozesses verbunden wird.

-| Fork, bei dem die Datei mit der Standardausgabe des Child-Prozesses verbunden wird.

Die Modi für die mit drei Argumenten arbeitende Form von **open** umfassen:

|- Öffnet eine Pipe zu einem Befehl.

-| Öffnet eine Pipe von einem Befehl.

`perlopentut`

Gängige Konstanten

Verschiedene eingabe-/ausgabebezogene Konstanten können aus dem Standardmodul `Fcntl` importiert werden.

Mit **open** und **sysopen** verknüpfte Konstanten werden per Voreinstellung importiert. Für einige Konstanten werden die weithin akzeptierten Werte in Oktalschreibweise angegeben.

Wert	Name	Beschreibung
00000	O_RDONLY	Reiner Lesezugriff.
00001	O_WRONLY	Reiner Schreibzugriff.
00002	O_RDWR	Schreib-/Lesezugriff.
00100	O_CREAT	Datei anlegen, falls sie nicht existiert.
00200	O_EXCL	Fehler, wenn Datei bereits existiert.
02000	O_APPEND	Daten an Dateiende anhängen.
01000	O_TRUNC	Datei kürzen.
	O_NONBLOCK	Nicht sperrende Ein-/Ausgabe.
	O_NDELAY	Wie O_NONBLOCK.
	O_SYNC	Synchrone Ein-/Ausgabe.
	O_EXLOCK	Exklusives Locking.
	O_SHLOCK	Shared Locking.
	O_DIRECTORY	Datei muss ein Verzeichnis sein.
	O_NOFOLLOW	Symbolische Links werden nicht verfolgt.
	O_BINARY	Binärmodus für Ein-/Ausgabe.
	O_LARGEFILE	Erlaubt Dateigrößen über 4 GByte.
	O_NOCTTY	Terminal wird nicht zum kontrollierenden TTY.

Mit **seek** und **sysseek** verknüpfte Konstanten müssen explizit importiert werden, indem man :seek in der Importliste von Fcntl angibt

Wert	Name	Beschreibung
00	SEEK_SET	Positionierung zur angegebenen Position.
01	SEEK_CUR	Positionierung als Offset von aktueller Position.
02	SEEK_END	Positionierung als Offset vom Dateiende

Mit **flock** verknüpfte Konstanten müssen explizit importiert werden, indem man :flock in der Importliste von Fcntl angibt.

Wert	Name	Beschreibung
001	LOCK_SH	Shared Locking.
002	LOCK_EX	Exklusives Locking.
004	LOCK_NB	Nicht sperrendes Locking.
010	LOCK_UN	Locking aufheben.

Standard-E/A-Schichten

Layer	Beschreibung
:bytes	Arbeitet mit 8-Bit-Bytes und nicht mit :utf8.
:crlf	Wandelt CR/LF in Newline um und umgekehrt.
:encoding(enc)	Wählt eine bestimmte Codierung.
:perlio	Verwendet die PerlIO-Implementierung von Perl.
:raw	Verwendet das Low-Level-E/A-System.
:stdio	Verwendet die Standard-E/A-Implementierung des Systems.
:unix	Verwendet ein Unix-kompatibles Low-Level-E/A-System.
:utf8	Verwendet die Perl-interne Unicode-Codierung.
:Via(module)	Verwendet das angegebene Modul zur Verarbeitung der Ein-/Ausgabe.
:win32	Verwendet die Windows-eigene (native) Ein-/Ausgabe (nur für Microsoft Windows-Plattformen).

PerlIO, perlrun (unter ENVIRONMENT/PERLIO).

Formatierte Ausgabe

printf und **sprintf** formatieren eine Liste von Werten entsprechend einem Formatstring, der die folgenden Umwandlungen erlaubt:

%% Ein Prozentzeichen.

%b Eine Zahl (binär) ohne Vorzeichen.

`%c`	Das dem Wert entsprechende Zeichen.
`%d`	Ein Integer-Wert mit Vorzeichen.
`%e`	Eine Fließkommazahl (wissenschaftliche Notation).
`%f`	Eine Fließkommazahl (in fester Dezimalnotation).
`%g`	Eine Fließkommazahl (in `%e`- oder `%f`-Notation).
`%i`	Ein Synonym für `%d`.
`%n`	Die Anzahl der bislang formatierten Zeichen wird in der entsprechenden Variablen der Parameterliste gespeichert.
`%o`	Ein oktaler Integer-Wert ohne Vorzeichen.
`%p`	Ein Zeiger (Adresse hexadezimal).
`%s`	Ein String.
`%u`	Ein dezimaler Integer-Wert ohne Vorzeichen.
`%x`	Ein hexadezimaler Integer-Wert ohne Vorzeichen.
`%B`	Wie `%b`, nur mit einem großen `B`.
`%D`	Ein veraltetes Synonym für `%ld`.
`%E`	Wie `%e`, aber mit einem großen `E`.
`%F`	Ein veraltetes Synonym für `%f`.
`%G`	Wie `%g`, aber mit einem großen `G` (wenn zutreffend).
`%O`	Ein veraltetes Synonym für `%lo`.
`%U`	Ein veraltetes Synonym für `%lu`.
`%X`	Wie `%x`, aber mit Großbuchstaben.

Die folgenden Flags können zwischen dem `%` und dem Umwandlungsbuchstaben stehen:

`Leerzeichen`	Einer positiven Zahl wird ein Leerzeichen vorangestellt.
`+`	Einer positiven Zahl wird ein Pluszeichen vorangestellt.
`-`	Linksbündige Ausrichtung innerhalb des Felds.

`0`	Bei rechtsbündiger Ausrichtung werden Nullen anstelle von Leerzeichen verwendet.
`#`	Mit `o`, `b`, `x` und 45 `X:`. Einer Zahl ungleich null wird `0`, `0b`, `0x` oder `0X` vorangestellt.
`Zahl`	Minimale Feldweite
`.`*`Zahl`*	Bei einer Fließkommazahl: die Anzahl der Ziffern hinter dem Dezimalpunkt. Bei einem String: die maximale Länge. Bei einer Integer-Zahl: die minimale Breite. Die Integer-Zahl wird rechtsbündig ausgerichtet und mit Nullen aufgefüllt.
`h`	Interpretiert den Integer-Wert entsprechend dem C-Typ als »short« oder »unsigned short«.
`hh`	Interpretiert den Integer-Wert entsprechend dem C-Typ als char.
`j`	Interpretiert den Integer-Wert entsprechend dem C99-Typ als intmax_t.
`l`	Interpretiert den Integer-Wert entsprechend dem C-Typ als »long« oder »unsigned long«.
`ll`, `L`, `q`	Interpretiert den Integer-Wert entsprechend dem C-Typ als »quad« (64 Bit).
`t`	Interpretiert den Integer-Wert entsprechend dem C-Typ als ptrdiff_t.
`v`	Mit `d`, `o`, `b`, `x` oder `X`: Gibt den String als Folge seiner Werte aus.
`V`	Interpretiert den Integer-Wert entsprechend dem Perl-Typ.
`z`	Interpretiert den Integer-Wert entsprechend dem C-Typ als size_t.

Ein Sternchen (*) kann anstelle einer Zahl verwendet werden. Dann wird der Wert des nächsten Elements der Liste verwendet. Bei `%*v` wird das nächste Element der Liste zur Trennung der Werte genutzt.

Die Anordnung von Parametern kann bestimmt werden, indem man *n*$ direkt hinter einem % oder * einfügt. Diese Konvertierung verwendet dann das *n*-te Argument der Liste.

Im folgenden Abschnitt finden Sie eine Alternative zur formatierten Ausgabe.

Formate

formline *Picture* **,** *Liste*
: Formatiert *Liste* entsprechend dem *picture* und akkumuliert das Ergebnis in $^A.

write [*Dateihandle*]
: Schreibt ein formatiertes Record in die angegebene Datei und verwendet dabei das mit dieser Datei verknüpfte Format. Fehlt *Dateihandle*, wird das aktuell aktive Dateihandle verwendet.

Formate sind wie folgt definiert:

format [*Name*] = *Formatliste*.

Formatliste ist eine Reihe von Zeilen, von denen jede entweder eine Kommentarzeile (# in der ersten Spalte), eine Picture-Zeile oder eine Argumentzeile ist. Eine Picture-Zeile enthält Feldbeschreibungen. Sie kann auch anderen Text enthalten, der unverändert ausgegeben wird. Argumentzeilen enthalten Listen mit Werten, die im Format und in der Reihenfolge der vorstehenden Picture-Zeile ausgegeben werden.

Name ist per Voreinstellung STDOUT, wenn kein Wert angegeben wird.

Um ein Format mit dem aktuellen Ausgabe-Stream zu verknüpfen, weisen Sie dessen Namen der Spezialvariablen $~ zu. Ein Format, das Seitenumbrüche behandelt, kann $^ zugewiesen werden. Um einen Seitenumbruch bei der nächsten **write**-Operation zu erzwingen, setzen Sie $- auf null.

Picture-Felder sind:

`@<<<` Linksbündiges Feld. Die wiederholte Angabe von < legt die gewünschte Breite fest.

`@>>>` Rechtsbündiges Feld.

`@|||` Zentriertes Feld.

`@##.##` Numerisches Format mit vorgegebenem Dezimalpunkt.

`@0#.##` Wie oben, nur dass bei Bedarf ein Padding mit führenden Nullen erfolgt.

`@*` Mehrzeiliges (Multiline-)Feld.

Verwenden Sie ^ anstelle von @ zum Füllen mehrzeiliger Blöcke.

Verwenden Sie ~ in der Picture-Zeile, um unerwünschte Leerzeilen zu unterdrücken.

Verwenden Sie ~~ in der Picture-Zeile, um diese Formatzeile wiederholt anzuwenden, bis eine vollständig leere Zeile ausgegeben würde. Wird für ^-Felder verwendet, um deren vollständige Ausgabe zu erreichen.

Siehe auch `$^`, `$~`, `$^A`, `$%`, `$:`, `$^L`, `$-` und `$=` im Abschnitt »Spezialvariablen« auf Seite 83.

`perlform`

Routinen zum Lesen von Verzeichnissen

closedir *Verzeichnishandle*
: Schließt ein mit **opendir** geöffnetes Verzeichnis.

opendir *Verzeichnishandle, Verzeichnisname*
: Öffnet ein Verzeichnis im angegebenen Handle. Ist *Verzeichnishandle* eine nicht-initialisierte skalare Variable, wird automatisch ein neues, eindeutiges Handle erzeugt.

readdir *Verzeichnishandle*
: Im skalaren Kontext wird der nächste Eintrag aus dem Verzeichnis zurückgegeben. Ist kein weiterer Eintrag vor-

handen, wird *undef* zurückgegeben. Der Eintrag ist die Namenskomponente innerhalb des Verzeichnisses, nicht der vollständige Name.

Im Listenkontext wird eine Liste aller noch im Verzeichnis vorhandenen Einträge zurückgegeben.

rewinddir *Verzeichnishandle*
: Setzt die aktuelle Position auf den Anfang des Verzeichnisses zurück.

seekdir *Verzeichnishandle, Position*
: Legt die Position für **readdir** im Verzeichnis fest. *Position* muss ein Datei-Offset sein, wie er von **telldir** zurückgegeben wird.

telldir *Verzeichnishandle*
: Gibt die Position im Verzeichnis zurück.

Interaktion mit dem System

alarm *Ausdruck*†
: Löst ein SIGALRM-Signal nach *Ausdruck* Sekunden aus. Hat *Ausdruck* den Wert 0, wird ein noch offenes Timer-Ereignis aufgehoben.

chdir [*Ausdruck*]
: Wechselt das Arbeitsverzeichnis. *Ausdruck* kann ein Dateiname oder ein Handle sein. Verwendet `$ENV{HOME}` oder `$ENV{LOGNAME}`, wenn *Ausdruck* weggelassen wird.

chroot *Dateiname*†
: Ändert das Wurzel-(Root-)Verzeichnis für den Prozess und alle zukünftigen Child-Prozesse.

die [*Liste*]
: Gibt den Wert der *Liste* über STDERR aus und bricht mit dem Wert `$!`(wenn ungleich null) oder (`$?>> 8`) (wenn ungleich null) oder 255 ab. *Liste* ist mit `"Died"` voreingestellt.

Die Meldung enthält den Namen und die Zeilennummer des Programms sowie die gerade gelesene Zeile. Diese Information wird unterdrückt, wenn das letzte Zeichen des letzten Elements der *Liste* ein Newline ist.

Innerhalb eines **eval** wird die Fehlermeldung in $@ abgelegt, **eval** wird abgebrochen und gibt *undef* zurück. Auf diese Weise können **die** und **eval** Ausnahmen auslösen und abfangen.

exec [*Programm*] *Liste*
: Führt den Systembefehl in *Liste* aus und kehrt nicht zurück. *Programm* kann verwendet werden, um ein Programm zur Ausführung des Befehls zu bestimmen.

exit [*Ausdruck*]
: Bricht unmittelbar mit dem Wert von *Ausdruck* (standardmäßig 0) ab. Ruft vor dem Abbruch **END**-Routinen und Objektdestruktoren auf.

fork
: Führt den Systemaufruf *fork* durch. Gibt die Prozess-ID des Child-Prozesses an den Parent-Prozess (bzw. *undef* bei einem Fehler) und den Wert 0 an den Child-Prozess zurück.

getlogin
: Gibt den aktuellen Login-Namen zurück. Wird falsch zurückgegeben, müssen Sie auf **getpwuid** zurückgreifen.

getpgrp [*PID*]
: Gibt die Prozessgruppe für den Prozess *PID* zurück (der Wert 0 bzw. eine fehlende Angabe steht für den aktuellen Prozess).

getppid
: Gibt die Prozess-ID des Parent-Prozesses zurück.

getpriority *was*, *wer*
: Gibt die aktuelle Prioritat eines Prozesses, einer Prozessgruppe oder eines Benutzers zurück. Verwenden Sie `getpriority 0,0` für den aktuellen Prozess.

glob *Ausdruck*†

Gibt eine Liste aller Dateinamen zurück, die dem C-Shell-Muster in *Ausdruck* entsprechen. <*Muster*> ist ein veraltetes Kürzel für `glob("Muster")`.

Eine bessere Kontrolle über das Globbing bietet Ihnen `File::Glob`.

kill *Liste*

Sendet ein Signal an eine Liste von Prozessen. Das erste Element der Liste muss das zu sendende Signal – numerisch (z.B. 1) oder seinen Namen als String (z.B. `HUP`) – enthalten. Negative Signale beenden Prozessgruppen anstelle von Prozessen.

setpgrp *PID, Prozeßgruppe*

Legt die Prozessgruppe für *PID* fest. Der *PID*-Wert 0 steht für den aktuellen Prozess.

setpriority *was, wer, Priorität*

Legt die Priorität für einen Prozess, eine Prozessgruppe oder einen Benutzer fest.

sleep [*Ausdruck*]

Lässt das Programm für *Ausdruck* Sekunden pausieren (»schlafen«). Ohne *Ausdruck* pausiert es für immer. Gibt die Anzahl von Sekunden zurück, die das Programm tatsächlich pausiert hat.

syscall *Liste*

Führt den Systemaufruf aus, der als erstes Element der Liste angegeben wurde. Die restlichen Elemente der Liste werden als Argumente an den Aufruf übergeben. Gibt bei einem Fehler –1 zurück (und setzt `$!`).

system [*Programm*] *Liste*

Wie **exec**, nur dass zuerst ein Child-Prozess erzeugt (`fork` ausgeführt) wird und der Parent-Prozess darauf wartet, dass der Child-Prozess zurückkehrt. Während des Wartens werden die Signale `SIGINT` und `SIGQUIT` an den Child-Prozess weitergegeben.

Gibt den Exit-Status des Child-Prozesses zurück. 0 steht für Erfolg, nicht für einen Fehler.

Programm kann verwendet werden, um explizit ein Programm zur Ausführung des Befehls anzugeben.

times Gibt eine Liste mit vier Elementen zurück (`user`, `system`, `cuser`, `csystem`), die die Benutzer- und Systemzeiten für diesen Prozess sowie für die Child-Prozesse dieses Prozesses angeben (in Sekunden).

umask [*Ausdruck*]
Legt die `umask` für diesen Prozess fest und gibt die alte zurück. Ist *Ausdruck* eine Zahl, muss es sich um eine Oktalzahl handeln. Lassen Sie *Ausdruck* weg, bleibt der aktuelle **umask**-Wert unverändert.

wait Wartet auf die Beendigung eines Child-Prozesses und gibt die Prozess-ID des beendeten Prozesses zurück (–1, falls nicht vorhanden). Der Status wird in `$?` zurückgegeben.

waitpid *PID, Flags*
Führt die gleiche Funktion aus wie der entsprechende Systemaufruf. Gibt 1 zurück, wenn der Prozess *PID* nicht mehr läuft, und –1, wenn er nicht existiert.

warn [*Liste*]
Gibt wie *die* die `Liste` über **STDERR** aus, bricht aber nicht ab. *Liste* ist mit `"Warning: something's wrong"` voreingestellt. Gibt die gleichen Programminformationen aus wie **die**.

Netzwerkprogrammierung

accept *Neuersocket, Listeningsocket*
Akzeptiert einen neuen Socket. Ist *Neuersocket* eine nichtinitialisierte skalare Variable, wird automatisch ein neues, eindeutiges Handle erzeugt.

bind *Socket, Name*
Bindet den Namen an den Socket.

connect *socket, name*
: Verbindet den Namen mit dem Socket.

getpeername *Socket*
: Gibt die Socket-Adresse des anderen Endes des Sockets zurück.

getsockname *Socket*
: Gibt den Namen des Sockets zurück.

getsockopt *Socket, Level, Optionsname*
: Gibt die Socket-Optionen zurück.

listen *Socket, queuesize*
: Beginnt die Überwachung (»Listening«) auf dem angegebenen Socket und erlaubt *queuesize* Verbindungen.

recv *Socket*, $ *Variable*, *Länge*, *Flags*
: Empfängt eine Nachricht mit *Länge* Zeichen auf dem Socket und legt diese in der skalaren $*Variable* ab.

send *Socket, Nachricht, Flags* [, *an*]
: Sendet eine Nachricht an den Socket.

setsockopt *Socket, Level, Optionsname, Optionswert*
: Setzt die gewünschte Socket-Option.

shutdown *Socket, wie*
: Fährt den Socket herunter.

socket *Socket, Domain, Typ, Protokoll*
: Erzeugt einen Socket in der Domain mit dem angegebenen Typ und Protokoll. Ist *Socket* eine nicht-initialisierte skalare Variable, wird ein neues, eindeutiges Handle erzeugt.

socketpair *Socket1, Socket2, Domain, Typ, Protokoll*
: Arbeitet wie **socket**, erzeugt aber ein Paar bidirektionaler Sockets.

System V IPC

Verwenden Sie (mittels **use**) das Standardmodul IPC::SysV, um auf die Nachrichten-und Semaphore-spezifischen Operationsnamen zugreifen zu können.

msgctl *ID, Befehl, Argumente*
: Ruft *msgctl* auf. Hat *Befehl* den Wert IPC_STAT, muss *Argumente* eine skalare Variable sein.

msgget *Schlüssel, Flags*
: Erzeugt eine Nachrichten-Queue für *Schlüssel*. Gibt die ID der Nachrichten-Queue zurück.

msgrcv *ID*, $*Variable, Größe, Typ, Flags*
: Empfängt eine Nachricht aus der Queue-*ID* und legt sie in $*Variable* ab.

msgsnd *ID, Nachricht, Flags*
: Sendet eine *Nachricht* an die Queue-*ID*.

semctl *ID, Semaphore, Befehl, Argument*
: Ruft *semctl* auf. Hat *Befehl* den Wert IPC_STAT oder GETALL, muss *Argument* eine skalare Variable sein.

semget *Schlüssel, Nachrichtensemaphore, Größe, Flags*
: Erzeugt einen Satz Semaphoren für *Schlüssel*. Gibt die ID der Nachrichten-Semaphore zurück.

semop *Schlüssel*, ...
: Führt Semaphore-Operationen aus.

shmctl *ID, Befehl, Argument*
: Ruft *shmctl* auf. Hat *Befehl* den Wert IPC_STAT, muss *Argument* eine skalare Variable sein.

shmget *Schlüssel, Größe, Flags*
: Erzeugt einen Shared-Memory-Bereich (Bereich gemeinsam genutzten Speichers). Gibt die ID des Shared-Memory-Segments zurück.

shmread *sID*, $*Variable*, *Position*, *Größe*
: Liest bis zu *Größe* Bytes aus dem Shared-Memory-Segment *ID*, beginnend an Offset *Position*, in die $*Variable* ein.

shmwrite *ID*, *String*, *Position*, *Größe*
: Schreibt bis zu *Größe* Bytes aus *String* in das Shared-Memory-Segment *ID* an Offset *Position*.

perlipc

Vermischtes

defined *Ausdruck*†
: Prüft, ob der skalare Ausdruck einen Wert besitzt.

do { *Ausdruck* ; ... }
: Führt den Block aus und gibt den Wert des letzten Ausdrucks zurück. Beachten Sie auch den Abschnitt »Anweisungen« auf Seite 16.

do *Dateiname*
: Führt *Dateiname* als Perl-Skript aus. Siehe auch **require** auf Seite 20.

eval { *Ausdruck* ; ... }
: Führt den Code zwischen { und } aus. Fängt Laufzeitfehler ab und gibt sie, wie bei **eval**(*Ausdruck*) auf Seite 20 beschrieben, zurück.

local [**our**] *Variable*
: Weist der Paketvariablen einen temporären Wert zu. Dieser ist gültig, bis der umschließende Block, die Datei oder **eval** endet. *Variable* kann ein Skalar, Array, Hash oder ein Element (oder Slice) eines Arrays oder Hashs sein.

my *Variablenliste*
: *Variablenliste* ist eine Variable oder eine in Klammern stehende Liste mit Variablen. Erzeugt einen Geltungsbereich für eine Variable, die lexikalisch gesehen lokal im umschließenden Block, in der Datei oder in **eval** liegt.

my [*Klasse*] *Variablenliste* [*Attribute*]
: Experimentell. Fest eingebautes Attribut ist `:shared`. Moduleigene `Attribute::Handlers` können zur Definition zusätzlicher Attribute verwendet werden.

our *Variablenliste*
: Deklariert die Variable als global gültig innerhalb des umschließenden Blocks, der Datei oder **eval**.

our [*Klasse*] *Variablenliste* [*Attribute*]
: Experimentell. Fest eingebaute Attribute sind `:shared` und `:unique`. Moduleigene `Attribute::Handler` können zur Definition zusätzlicher Attribute verwendet werden.

ref *Ausdruck*†
: Gibt den Referenztyp zurück, wenn *Ausdruck* eine Referenz ist. Gibt den Paketnamen zurück, wenn *Ausdruck* über `bless` in einem Paket bekannt gemacht wurde.

reset [*Ausdruck*]
: *Ausdruck* ist ein String aus einzelnen Buchstaben. Alle Variablen des aktuellen Pakets, die mit einem dieser Buchstaben beginnen, werden in ihren ursprünglichen Zustand zurückgesetzt. Fehlt *Ausdruck*, werden `??`-Suchoperationen so zurückgesetzt, dass sie wieder funktionieren.

state *Variablenliste*
: Wie **my**, nur dass die Variablen beim Wiedereintreten in den umschließenden Block nicht reinitialisiert werden.

state [*Klasse*] *Variablenliste* [*Attribute*]
: Experimentelle Erweiterung von **state** *Variablenliste*.

undef [*Lvalue*]
: Entfernt *Lvalue*. Gibt immer *undef* zurück.

Bindung von Variablen (tie)

tie *Variable, Klassenname,* [*Liste*]
: Bindet eine Variable an eine Klasse, die sie behandelt. *Liste* wird an den Klassenkonstruktor übergeben.

tied *Variable*
: Gibt eine Referenz auf das *Variable* zugrunde liegende Objekt zurück bzw. *undef*, wenn *Variable* nicht an eine Klasse gebunden ist.

untie *Variable*
: Hebt die Bindung zwischen Variable und Klasse auf. Ruft (falls vorhanden) eine UNTIE-Methode auf.

Eine Klasse, die einen gebundenen Skalar implementiert, sollte die Methoden TIESCALAR, DESTROY, FETCH und STORE definieren.

Eine Klasse, die ein gebundenes normales Array implementiert, sollte die Methoden TIEARRAY, CLEAR, DESTROY, EXTEND, FETCHSIZE, FETCH, POP, PUSH, SHIFT, SPLICE, STORESIZE, STORE und UNSHIFT definieren.

Eine Klasse, die einen gebundenen Hash implementiert, sollte die Methoden TIEHASH, CLEAR, DELETE, DESTROY, EXISTS, FETCH, FIRSTKEY, NEXTKEY, SCALAR und STORE implementieren.

Eine Klasse, die ein gebundenes Dateihandle implementiert, sollte die Methoden TIEHANDLE, CLOSE, DESTROY, GETC, PRINTF, PRINT, READLINE, READ und WRITE implementieren.

Verschiedene Basisklassen zur Implementierung gebundener Variablen sind in der Standardbibliothek verfügbar: Tie::Array, Tie::Handle, Tie::Hash, Tie::RefHash und Tie::Scalar.

perltie

Informationen aus Systemdatenbanken

Informationen über Benutzer

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul User::pwent, um auf diese Elemente über deren Namen zugreifen zu können:

Index	Name	Beschreibung
0	Name	Benutzername
1	passwd	Passwortinformationen
2	uid	ID dieses Benutzers
3	gid	Gruppen-ID dieses Benutzers
4	quota	Quota-Informationen
5	comment	Kommentare
6	gecos	Vollständiger Name
7	dir	Home-Verzeichnis
8	shell	Login-Shell
9	expire	Informationen über Passwortgültigkeit

`endpwent`
: Beendet die Lookup-Verarbeitung.

`getpwent`
: Liefert die nächsten Benutzerinformationen.

  Im skalaren Kontext wird der Benutzername zurückgegeben.

`getpwnam` *Name*
: Gibt Informationen basierend auf dem Namen zurück.

  Im skalaren Kontext wird die Benutzer-ID zurückgegeben.

`getpwuid` *UID*
: Gibt Informationen basierend auf der Benutzer-ID zurück.

  Im skalaren Kontext wird der Benutzername zurückgegeben.

`setpwent`
: Setzt die Lookup-Verarbeitung zurück.

Informationen über Gruppen

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul `User::grent`, um auf die Elemente der Liste über ihre Namen zugreifen zu können:

Index	Name	Beschreibung
0	Name	Gruppenname
1	passwd	Passwortinformationen
2	gid	ID dieser Gruppe
3	members	Leerzeichenseparierte Liste der Login-Namen der Gruppenmitglieder

endgrent
: Beendet die Lookup-Verarbeitung.

getgrent
: Liefert die nächsten Gruppeninformationen.

  Im skalaren Kontext wird der Gruppenname zurückgegeben.

getgrgid *gid*
: Gibt Informationen basierend auf der Gruppen-ID zurück.

  Im skalaren Kontext wird der Gruppenname zurückgegeben.

getgrnam *Name*
: Gibt Informationen basierend auf dem Namen zurück.

  Im skalaren Kontext wird die Gruppen-ID zurückgegeben.

setgrent
: Setzt die Lookup-Verarbeitung zurück.

Informationen über Netzwerke

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul `Net::netent`, um über die Namen darauf zugreifen zu können:

Index	Name	Beschreibung
0	Name	Name des Netzwerks
1	aliases	Aliasnamen
2	addrtype	Adresstyp
3	net	Netzwerkadresse

`endnetent`
: Beendet die Lookup-Verarbeitung.

`getnetbyaddr` *`Adresse, Typ`*
: Gibt Informationen basierend auf Adresse und Typ zurück.

  Im skalaren Kontext wird der Netzwerkname zurückgegeben.

`getnetbyname` *`Name`*
: Gibt Informationen basierend auf dem Netzwerknamen zurück.

  Im skalaren Kontext wird die Netzwerknummer zurückgegeben.

`getnetent`
: Liefert die nächsten Netzwerkinformationen.

  Im skalaren Kontext wird der Netzwerkname zurückgegeben.

`setnetent` *`stayopen`*
: Setzt die Lookup-Verarbeitung zurück.

Informationen über Netzwerkhosts

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul `Net::hostent`, um auf die einzelnen Elemente über ihren Namen zugreifen zu können:

Index	Name	Beschreibung
0	name	Hostname
1	aliases	Aliasnamen
2	addrtype	Adresstyp
3	length	Länge der Adresse
4	addr	Adresse oder Adressen

endhostent
: Beendet die Lookup-Verarbeitung.

gethostbyaddr *Adresse* **,** *Adresstyp*
: Gibt Informationen basierend auf der IP-Adresse zurück.

  Im skalaren Kontext wird der Hostname zurückgegeben.

gethostbyname *Name*
: Gibt Informationen basierend auf dem Hostnamen zurück.

  Im skalaren Kontext wird die Hostadresse zurückgegeben.

gethostent
: Liefert die nächsten Hostinformationen.

  Im skalaren Kontext wird der Hostname zurückgegeben.

sethostent *bleiboffen*
: Setzt die Lookup-Verarbeitung zurück.

Informationen über Netzwerkdienste

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul `Net::servent`, um auf die einzelnen Elemente über ihre Namen zugreifen zu können:

Index	Name	Beschreibung
0	Name	Dienstname
1	aliases	Aliasnamen
2	port	Portnummer
3	proto	Protokollnummer

endservent
: Beendet die Lookup-Verarbeitung.

getservbyname *Name*, *protocol*
: Gibt Informationen für das angegebene Protokoll basierend auf dem Dienstnamen zurück.

  Im skalaren Kontext wird die Portnummer zurückgegeben.

getservbyport *port*, *protocol*
: Gibt Informationen für das angegebene Protokoll basierend auf dem Port zurück.

  Im skalaren Kontext wird der Dienstname zurückgegeben.

getservent
: Liefert die nächsten Dienste-Informationen.

  Im skalaren Kontext wird der Dienstname zurückgegeben.

setservent *bleiboffen*
: Setzt die Lookup-Verarbeitung zurück.

Informationen über Netzwerkprotokolle

Im Listenkontext gibt jede dieser Routinen eine Liste mit Werten zurück. Verwenden Sie das Standardmodul `Net::protoent`, um auf die Elemente dieser Liste über ihre Namen zugreifen zu können:

Index	Name	Beschreibung
0	Name	Protokollname
1	aliases	Aliasnamen
2	proto	Protokollnummer

endprotoent
: Beendet die Lookup-Verarbeitung.

getprotobyname *Name*
: Gibt Informationen basierend auf dem Protokollnamen zurück.

  Im skalaren Kontext wird die Protokollnummer zurückgegeben.

getprotobynumber *number*
: Gibt Informationen basierend auf der Protokollnummer zurück.

  Im skalaren Kontext wird der Name des Protokolls zurückgegeben.

getprotoent
: Liefert die nächsten Protokollinformationen.

  Im skalaren Kontext wird der Name des Protokolls zurückgegeben.

setprotoent *bleiboffen*
: Setzt die Lookup-Verarbeitung zurück.

Spezialvariablen

Die alternativen Namen der Spezialvariablen werden vom Standardmodul `English` zur Verfügung gestellt.

Die folgenden Variablen sind global und sollten in Subroutinen lokalisiert werden:

`$_` Alternativ: `$ARG`.

Das Standardargument vieler Funktionen und Operationen.

`$.` Alternativ: `$INPUT_LINE_NUMBER`, `$NR`.

Die Nummer der aktuellen Eingabezeile aus dem zuletzt gelesenen Dateihandle. Wird nur zurückgesetzt, wenn das Dateihandle explizit geschlossen wird.

`$/` Alternativ: `$INPUT_RECORD_SEPARATOR`, `$RS`.

Das Trennzeichen für Eingabe-Records. Voreingestellt ist das Newline-Zeichen.

`$,` Alternativ: `$OUTPUT_FIELD_SEPARATOR`, `$OFS`.

Das Trennzeichen für Ausgabefelder für die print-Funktion. Voreingestellt ist der Leerstring.

`$"` Alternativ: `$LIST_SEPARATOR`.

Das Trennzeichen, das für die Einbindung von in Strings interpolierten Array-Elementen verwendet wird. Voreingestellt ist ein einzelnes Leerzeichen.

`$\` Alternativ: `$OUTPUT_RECORD_SEPARATOR`, `$ORS`.

Das Trennzeichen für Ausgabe-Records für die print-Funktionen. Voreingestellt ist der Leerstring.

`$?` Alternativ: `$CHILD_ERROR`.

Status, der beim zuletzt ausgeführten `` `...` ``-Befehl oder Pipe-**close** oder der letzten **wait**-, **waitpid**- bzw. **system**-Funktion zurückgegeben wurde.

`$]` Die Perl-Versionsnummer, z.B. `5.006`. Siehe auch `$^V` auf Seite 87.

`$[` Der Index des ersten Elements in einem Array oder einer Liste und des ersten Zeichens in einem Substring. Voreingestellt ist 0. Veraltet, nicht mehr verwenden.

`$;` Alternativ: `$SUBSCRIPT_SEPARATOR`, `$SUBSEP`.

Das Indextrennsymbol für die Emulation mehrdimensionaler Hashes. Voreingestellt ist `"\034"`.

`$!` Alternativ: `$OS_ERROR`, `$ERRNO`.

In einem numerischen Kontext wird der aktuelle Wert von `errno` zurückgegeben. In einem String-Kontext wird der entsprechende Fehlerstring zurückgeliefert.

`$@` Alternativ: `$EVAL_ERROR`.

Die Perl-Fehlermeldung des letzten **`eval`**- oder **`do`** *Ausdruck*-Befehls.

`$:` Alternativ: `$FORMAT_LINE_BREAK_CHARACTERS`.

Die Gruppe von Zeichen, an denen ein String umbrochen werden kann, um in einem Format Fortsetzungszeilen (beginnend mit ^) aufzufüllen.

`$0` Alternativ: `$PROGRAM_NAME`.

Der Name der Datei, die das ausgeführte Perl-Skript enthält. Eine Zuweisung ist möglich.

`$$` Alternativ: `$PROCESS_ID`, `$PID`.

Die Prozess-ID des Perl-Interpreters, der das Skript ausführt. Wird (im Child-Prozess) durch **`fork`** geändert.

`$<` Alternativ: `$REAL_USER_ID`, `$UID`.

Die reale UID dieses Prozesses.

`$>` Alternativ: `$EFFECTIVE_USER_ID`, `$EUID`.

Die effektive UID dieses Prozesses.

`$(` Alternativ: `$REAL_GROUP_ID`, `$GID`.

Die reale GID dieses Prozesses.

`$)` Alternativ: `$EFFECTIVE_GROUP_ID`, `$EGID`.

Die effektive GID oder eine Liste von durch Leerzeichen getrennten GIDs dieses Prozesses.

`$^A` Alternativ: `$ACCUMULATOR`.

Der Akkumulator für **`formline`**- und **`write`**-Operationen.

`$^{CHILD_ERROR_NATIVE}`
Wie `$?`, gibt aber den nativen Status zurück.

`$^C` Alternativ: `$COMPILING`.

Wahr, wenn Perl über die Kommandozeilenoption `-c` nur im Compiler-Modus läuft.

`$^D` Alternativ: `$DEBUGGING`.

Die Debug-Flags, die mit `-D` an Perl übergeben wurden.

`$^E` Alternativ: `$EXTENDED_OS_ERROR`.

Betriebssystemabhängige Fehlerinformationen.

`$^F` Alternativ: `$SYSTEM_FD_MAX`.

Der höchste Systemdateideskriptor, normalerweise 2.

`$^H` Der aktuelle Zustand von Syntaxprüfungen.

`$^I` Alternativ: `$INPLACE_EDIT`.

Inplace-Edit-Erweiterung, wie über die Kommandozeilenoption `-i` angegeben.

`$^L` Alternativ: `$FORMAT_FORMFEED`.

In Formaten verwendetes Seitenvorschubzeichen.

`$^M` Notfallspeicherbereich.

`$^O` Alternativ: `$OSNAME`.

Betriebssystemname.

`$^P` Alternativ: `$PERLDB`.

Internes Debugging-Flag.

`${^RE_DEBUG_FLAGS}`
Flags für das Debugging.

`${^RE_TRIE_MAXBUF}`
: Legt die Größe der temporären Caches für die Trie-Optimierung fest. Mit negativem Wert wird die Optimierung ausgeschaltet.

`$^S` Alternativ: `$EXCEPTIONS_BEING_CAUGHT`.

: Aktueller Zustand des Perl-Interpreters.

`$^T` Alternativ: `$BASETIME`.

: Der Zeitpunkt, zu dem das Programm gestartet wurde (wie er von **`time`** geliefert wird). Dieser Wert wird von den Dateitestoperatoren `-M`, `-A` und `-C` verwendet.

`${^TAINT}`
: Der aktuelle Zustand des Taint-Modus.

`$^V` Alternativ: `$PERL_VERSION`.

: Die Perl-Version als version-Objekt. Verwenden Sie zur Ausgabe das Format `%vd`.

`$^W` Alternativ: `$WARNING`.

: Der Wert der an Perl übergebenen `-w`-Option.

`${^WIN32_SLOPPY_STAT}`
: Aktiviert die nachlässigen stat ()-Aufrufe unter Windows. Dateien werden dann bei stat ()-Aufrufen nicht mehr geöffnet.

`$^X` Alternativ: `$EXECUTABLE_NAME`.

: Der Name, unter dem Perl aufgerufen wurde.

`$AUTOLOAD`
: Der Name der aufgerufenen undefinierten Subroutine.

`$REGERROR`
: Bei einem Matching-Fehler der Name der fehlgeschlagenen Backtrack-Kontrolle oder des letzten (`*MARK:`*Name*)-Musters.

`$REGMARK`

Bei einem erfolgreichen Matching der *Name* des letzten (`*MARK:`*Name*)-Musters.

Die folgenden Variablen sind kontextabhängig und müssen nicht lokalisiert werden:

`$%` Alternativ: `$FORMAT_PAGE_NUMBER`.

Die aktuelle Seitennummer des aktuell gewählten Ausgabehandles.

`$=` Alternativ: `$FORMAT_LINES_PER_PAGE`.

Die Seitenlänge des aktuellen Ausgabehandles. Voreingestellt sind 60 Zeilen.

`$-` Alternativ: `$FORMAT_LINES_LEFT`.

Die Anzahl der auf der Seite noch verbliebenen Zeilen.

`$~` Alternativ: `$FORMAT_NAME`.

Der Name des aktuellen Reportformats.

`$^` Alternativ: `$FORMAT_TOP_NAME`.

Der Name des aktuellen Formats der Kopfzeile(n) (top-of-page).

`$|` Alternativ: `$OUTPUT_AUTOFLUSH`.

Bei einem Wert ungleich null wird nach jedem write oder print auf das aktuelle Ausgabehandle die sofortige Leerung des Ausgabepuffers erzwungen (Flush). Voreingestellt ist 0.

`$ARGV` Der Name der aktuellen Datei, wenn über < > gelesen wird.

Die folgenden Variablen sind immer lokal zum aktuellen Block:

`$&` Alternativ: `$MATCH`.

Der von der letzten Mustererkennung erfolgreich erkannte String.

`$`` Alternativ: `$PREMATCH`.

Der String, der bei der letzten erfolgreichen Mustererkennung vor dem Treffer steht.

`$'` Alternativ: `$POSTMATCH`.

Der String, der bei der letzten erfolgreichen Mustererkennung hinter dem Treffer steht.

`$+` Alternativ: `$LAST_PAREN_MATCH`.

Die letzte Klammer, die vom letzten Suchmuster erkannt wird.

`$1...$9...`

Enthalten die Submuster der entsprechenden Klammernpaare des zuletzt erkannten Musters. `$10` und höher sind nur verfügbar, wenn der Treffer entsprechend viele Submuster enthält.

`$^N` Alternativ: `$LAST_SUBMATCH_RESULT`.

Der von der letzten geschlossenen Gruppe erkannte Text.

`$^R` Alternativ: `$LAST_REGEXP_CODE_RESULT`.

Das Ergebnis der letzten (`?{` *Code* `}`)-Operation.

`perlvar`

Spezial-Arrays

Die alternativen Namen werden vom Standardmodul `English` zur Verfügung gestellt.

`@_` Alternativ: `@ARG`.

Parameter-Array für Subroutinen. Wird auch von **`split`** verwendet, wenn kein Listenkontext vorliegt.

`@-` Alternativ: `@LAST_MATCH_START`.

Nach einer erfolgreichen Mustererkennung sind hier die Offsets der Anfänge der erfolgreichen Subtreffer zu finden. `$-[0]` ist der Offset des gesamten Treffers.

`@+` Alternativ: `@LAST_MATCH_END`.

Wie `@-`, aber die Offsets verweisen auf das Ende der Subtreffer. `$+[0]` ist der Offset für das Ende des gesamten Treffers.

`@ARGV` Enthält die Kommandozeilenargumente für das Skript (aber ohne den Befehlsnamen, der in `$0` steht).

`@EXPORT` Benennt die Methoden und andere Symbole, die ein Paket standardmäßig exportiert. Wird vom `Exporter`-Modul verwendet.

`@EXPORT_OK`
Benennt die Methoden und andere Symbole, die ein Paket auf explizite Anforderung exportieren kann. Wird vom `Exporter`-Modul verwendet.

`@F` Wird die Kommandozeilenoption `-a` verwendet, sind hier die zerlegten Eingabezeilen zu finden.

`@INC` Enthält eine Liste der Orte, an denen nach Perl-Skripten gesucht wird, die von den Befehlen **`do`** *`Dateiname`*, **`use`** und **`require`** ausgeführt werden sollen.

Modifizieren Sie `@INC` nicht direkt, sondern verwenden Sie stattdessen das Pragma `lib` oder die Kommandozeilenoption `-I`.

`@ISA` Liste der Basisklassen eines Pakets.

`perlvar`

Spezial-Hashes

`%!` Jedes Element von `%!` hat nur dann einen Wert ungleich null, wenn `$!` auf diesen Wert gesetzt wurde.

`%+` Enthält die benannten Submuster mit definierten Werten der letzten erfolgreichen Mustererkennung.

`%-` Enthält alle benannten Submuster der letzten erfolgreichen Mustererkennung.

`%ENV` Enthält die aktuelle Umgebung. Der Schlüssel ist der Name einer Umgebungsvariablen, der Wert ihr aktueller Inhalt.

`%EXPORT_TAGS`
Definiert Namen für Gruppen von Symbolen. Wird vom `Exporter`-Modul verwendet.

`%INC` Enthält die Liste der Dateien, die mit **`use`**, **`require`** oder **`do`** eingebunden wurden. Der Schlüssel ist der Dateiname, wie er im Befehl übergeben wurde, der Wert ist der Pfad zur Datei.

`%SIG` Wird zur Einrichtung von Signalhandlern für verschiedene Signale verwendet. Der Schlüssel ist der Name des Signals (ohne das Präfix `SIG`); der Wert ist eine Subroutine, die ausgeführt wird, wenn das Signal auftritt.

`__WARN__` und `__DIE__` sind Pseudosignale, mit denen Handler für Perl-Warnungen und -Ausnahmen festgelegt werden können.

`perlvar`

Umgebungsvariablen

Perl nutzt verschiedene Umgebungsvariablen. Mit einigen wenigen Ausnahmen beginnen alle mit `PERL`. Bibliothekspakete und plattformabhängige Features verwenden eigene Umgebungsvariablen.

Die gängigsten Umgebungsvariablen sind:

`HOME` Wird verwendet, wenn **`chdir`** kein Argument besitzt.

LC_ALL, LC_CTYPE, LC_COLLATE, LC_NUMERIC, PERL_BADLANG, LANGUAGE, LANG
: Kontrollieren, wie Perl für bestimmte natürliche Sprachen spezifische Daten behandelt.

LOGDIR
: Wird verwendet, wenn **chdir** kein Argument besitzt und HOME nicht gesetzt ist.

PATH
: Wird zur Ausführung von Unterprozessen und (wenn -S benutzt wird) zum Auffinden von Perl-Skripten verwendet.

PERL5LIB
: Eine durch Doppelpunkte getrennte Liste von Verzeichnissen. In diesen Verzeichnissen wird nach Perl-Bibliotheksdateien gesucht, bevor in der Standardbibliothek und im aktuellen Verzeichnis nachgesehen wird.

PERLLIB
: Wird anstelle von PERL5LIB verwendet, wenn PERL5LIB nicht definiert ist.

PERL5OPT
: Anfangs-(Kommandozeilen-)Optionen für Perl.

Threads

Die Unterstützung von Threads muss in das Perl-Executable einkompiliert sein.

Das threads-Pragma implementiert die Thread-Objekte und die für Threads notwendigen Operationen. Einige der wichtigsten Operationen sind:

async *Block*
: Startet einen Thread, um den Block auszuführen. Gibt ein Thread-Objekt zurück.

threads->create(*Subroutine* [, *Argumente*])
: Erzeugt einen neuen Thread und beginnt mit der Ausführung der referenzierten Subroutine. Die Argumente werden an diese Subroutine übergeben. Zurückgegeben wird ein Thread-Objekt.

`threads->list`
: Gibt eine Liste nutzbarer Threads zurück.

`threads->self`
: Gibt ein Objekt zurück, das den aktuellen Thread repräsentiert.

`threads->tid`
: Gibt die Thread-ID des aktuellen Threads zurück.

`threads->yield`
: Der aktuelle Thread gibt die CPU zugunsten eines anderen Threads auf.

`thread`-Objekte unterstützen (unter anderem) die folgenden Methoden:

`detach` Trennt einen Thread ab, sodass er unabhängig läuft.

`equal` (*`Thread`*)
: Gibt wahr zurück, wenn der Thread und *`Thread`* übereinstimmen. Sie können Thread-Objekte mithilfe des Operators `==` auch direkt vergleichen.

`join` Wartet auf die Beendigung des Threads. Der zurückgegebene Wert ist der von der Subroutine des Threads zurückgegebene Wert.

`tid` Gibt die Thread-ID eines Threads zurück.

Das Pragma `threads::shared` implementiert Operationen, die eine gemeinsame Nutzung von Variablen zwischen Threads ermöglichen:

`cond_broadcast` *`Variable`*
: Entsperrt alle Threads, die auf diese Variable warten. Die *`Variable`* muss gesperrt sein.

`cond_signal` *`Variable`*
: Entsperrt einen Thread, der auf diese Variable wartet. Die *`Variable`* muss gesperrt sein.

cond_timed_wait [*condvar* **,**] *Variable*, *Zeit*
: Wie **cond_wait**, doch mit einem Timeout nach der angegebenen Zeit.

cond_wait [*condvar* **,**] *Variable*
: Wartet darauf, dass ein anderer Thread ein **cond_signal** oder **cond_broadcast** für die Variable ausgibt. Die *Variable* muss gesperrt sein und wird während des Wartens temporär entsperrt. Bei *condvar* wird die *Variable* entsperrt, während auf ein *condvar*-Signal gewartet wird.

is_shared *Variable*
: Prüft, ob die angegebene Variable gemeinsam genutzt wird.

lock *Variable*
: Sperrt eine gemeinsam genutzte Variable vor dem Zugriff durch andere Threads. Die Sperrung wird automatisch aufgehoben, wenn der Geltungsbereich verlassen wird.

share *Variable*
: Markiert die Variable als gemeinsam genutzt.

shared_clone *Referenz*
: Macht aus der Referenz eine gemeinsam genutzte Version des Arguments.

`perlthrtut`, `threads`, `threads::shared`

ANHANG A
Kommandozeilenoptionen

`--` Beendet die Verarbeitung von Optionen.

`-0` [*`Oktalzahl`*]
(Zahl 0) Legt einen (oktalen) Anfangswert für den Record-Separator `$/` fest. Siehe auch `-l` weiter unten.

`-a` Aktiviert den Autosplit-Modus, wenn sie zusammen mit `-n` oder `-p` verwendet wird. Das Ergebnis steht in `@F`.

`-c` Prüft die Syntax, führt aber das Programm nicht aus. **`BEGIN`**-, **`CHECK`**- und **`UNITCHECK`**-Blöcke werden ausgeführt.

`-C` [*`Nummer / Liste`*]
Kontrolliert einen Teil der Unicode-Features von Perl.

`-d`[`t`] [`:`*`Modul`*] [`=`*`Argument`* [`,`*`Argument...`*]]
Führt das Skript unter dem angegebenen Modul aus. Mit `-dt` werden Threads aktiviert. Das Standardmodul ist der Perl-Debugger. Verwenden Sie `-de 0`, um den Debugger ohne ein Skript zu starten.

`-D` *`Flags`*
Setzt Debugging-Flags.

`-e` *`Kommandozeile`*
Kann verwendet werden, um eine einzelne Skriptzeile zu übergeben. Mehrere `-e`-Befehle können angegeben werden, um ein mehrzeiliges Skript aufzubauen.

`-E` Wie `-e`, aktiviert aber implizit alle optionalen Features. Siehe Abschnitt »Pragma-Module« auf Seite 21.

`-F` *Muster*
: Legt ein Muster fest, auf dem das Splitting basiert, wenn `-a` aktiv ist.

`-h`
: Gibt eine Zusammenfassung der Aufrufoptionen aus. Eine Ausführung erfolgt nicht.

`-i` [*Erweiterung*]
: Über < > verarbeitete Dateien werden direkt (»in-place«, an Ort und Stelle) editiert.

`-I` *Verzeichnis*
: Das Verzeichnis wird dem Suchpfad für Perl-Module, `@INC`, vorangestellt.

`-l` [*Oktalzahl*]
: (Buchstabe l) Aktiviert die automatische Zeilenendeverarbeitung, z.B. `-l013`.

`-m` [`-`] *Modul* [`=`*Argument* [`,`*Argument*`...`]]

`-M` [`-`] *Modul* [`=`*Argument* [`,`*Argument*`...`]]
: Führt ein **`use`** *Modul* vor der Ausführung des Skripts aus.

  Mit - wird hingegen ein **`no`** *Modul* ausgeführt.

  Ohne Argumente importiert `-M` den Standardsatz, während `-m` nichts importiert.

  Anderenfalls werden die Argumente an die **`import`**-Methode des Moduls übergeben.

`-n`
: Legt eine Eingabeschleife um Ihr Skript. Zeilen werden nicht ausgegeben.

`-p`
: Legt eine Eingabeschleife um Ihr Skript. Zeilen werden ausgegeben.

`-s`
: Interpretiert *-xxx* in der Kommandozeile als einen Switch und setzt die entsprechende Variable $*xxx* im Skript auf 1. Weist der Switch die Form *-xxx=yyy* auf, wird die Variable $*xxx* auf den Wert *yyy* gesetzt.

`-S` Verwendet die Umgebungsvariable `PATH`, um nach dem Skript zu suchen.

`-t` Aktiviert das *Taint*-Checking. Gibt bei Taint-Vergehen über **`warn`** eine Warnung aus.

`-T` Aktiviert das *Taint*-Checking. Bei Taint-Vergehen wird das Skript über **`die`** abgebrochen.

`-u` Erzeugt nach der Kompilierung des Skripts einen Coredump, der dann mit dem Programm *undump* weiterverarbeitet werden kann. Veraltet.

`-U` Erlaubt Perl die Durchführung bestimmter unsicherer Operationen.

`-v` Gibt die Version und den Patchlevel Ihres Perl-Interpreters aus. Es erfolgt keinerlei Ausführung.

`-V [ :`*Variable*` ]`
Gibt die Perl-Konfigurationsinformationen aus, wie z.B. `-V:man.dir`. Es erfolgt keinerlei Ausführung.

`-w` Gibt Warnungen zu möglichen Schreibfehlern und anderen fehleranfälligen Konstrukten im Skript aus. Kann vom Programm aus aktiviert und deaktiviert werden.

`-W` Warnungen werden fest aktiviert.

`-x [ `*Verzeichnis*` ]`
Extrahiert das Skript aus dem Eingabe-Stream.

Ist *Verzeichnis* angegeben, wechselt Perl zuerst in dieses Verzeichnis.

`-X` Warnungen werden fest deaktiviert.

`perlrun`.

PERL5OPT

In der Umgebungsvariablen `PERL5OPT` können Sie die folgenden Kommandozeilenoptionen festlegen: `-D`, `-I`, `-M`, `-T`, `-U`, `-W`, `-d`, `-m`, `-t` und `-w`.

#!

Alle Optionen außer `-M` und `-m` können auch in der `#!`-Zeile des Perl-Skripts angegeben werden.

Die Optionen `-C` und `-T` können in der `#!`-Zeile verwendet werden, solange sie auch in der Kommandozeile angegeben werden.

ANHANG B
Der Perl-Debugger

Der symbolische Perl-Debugger wird mit `perl -d` aufgerufen. Der Befehl `perl -de 0` bietet eine gute Möglichkeit, mit Perl und dem Debugger zu spielen.

Beim Start versucht der Debugger, Einstellungen und Befehle aus der Datei *.perldb* (*perldb.ini* unter Windows) im aktuellen Verzeichnis oder, falls nicht vorhanden, im Homeverzeichnis einzulesen.

Jede Eingabe, die keinem der nachfolgend aufgeführten Befehle entspricht, wird als Perl-Ausdruck evaluiert.

`a` [*Zeile*] *Befehl*
: Legt eine Aktion für *Zeile* fest.

`A` [*Zeile*]
: Löscht die Aktion für die angegebene Zeile. Per Voreinstellung wird die aktuelle Zeile verwendet. Wird die *Zeile* mit * angegeben, werden die Aktionen für alle Zeilen gelöscht.

`b` [*Zeile* [*Bedingung*]]
: Setzt einen Breakpunkt an *Zeile*; standardmäßig ist das die aktuelle Zeile.

`b` *Subname* [*Bedingung*]
: Setzt einen Breakpunkt an der benannten Subroutine.

`b compile` *Subname*
: Hält an, nachdem die Subroutine kompiliert wurde.

`b load` *`Datei`*
: Setzt einen Breakpunkt beim Anfordern der Datei über **`require`**.

`b postpone` *`Subname`* [*`Bedingung`*]
: Setzt einen Breakpunkt an der ersten Zeile der Subroutine nach der Kompilierung.

`B` [*`Zeile`*]
: Löscht den Breakpunkt an der angegebenen Zeile. Voreingestellt ist die aktuelle Zeile. Wird die *`Zeile`* mit * angegeben, werden alle Breakpunkte gelöscht.

`c` [*`Zeile`*]
: Fortsetzung (bis *`Zeile`* oder bis zu einem anderen Breakpunkt oder bis zum Ende).

`f` *`Datei`*
: Wechselt zu *`Datei`* und beginnt, sie auszugeben.

`h`
: Gibt eine lange Hilfsmeldung aus.

`h` *`Befehl`*
: Gibt eine Hilfe zum Debugger-Befehl *`Befehl`* aus.

`h h`
: Gibt eine knappe Hilfsmeldung aus.

`H` [*`-Anzahl`*]
: Gibt die letzten *`-Anzahl`* Befehle aus.

`l` [*`Bereich`*]
: Gibt einen Bereich von Zeilen aus. *`Bereich`* kann eine Zahl, *`Anfang - Ende`*, *`Start + Anzahl`* oder ein Subroutinenname sein. Fehlt *`Bereich`*, wird das nächste Fenster ausgegeben.

`l` *`Subname`*
: Gibt die Subroutine aus.

`L` [`a`|`b`|`w`]
: Gibt die Zeilen an, bei denen Aktionen, Breakpunkte oder Watches definiert sind.

`m` *`Klasse`*
: Gibt die Methoden aus, die über die angegebene Klasse aufrufbar sind.

`m` *Ausdruck*
: Evaluiert den Ausdruck im Listenkontext und gibt die Methoden aus, die über das erste Element des Ergebnisses aufgerufen werden können.

`man` [*topic*]
: Gibt die Systemdokumentation aus.

`M`
: Gibt die Versionen der geladenen Module aus.

`n` [*Ausdruck*]
: Einzelschritte beim Subroutinenaufruf.

`o` [*Option* [`=`*Wert*]]
: Setzt Werte von Debugger-Optionen. Der Standardwert ist wahr.

`o` *opt* `?`
: Fragt Werte der Debugger-Optionen ab.

`p` *Ausdruck*†
: Evaluiert *Ausdruck* im Listenkontext und gibt das Ergebnis aus.

  Siehe auch `x` weiter unten.

`q`
: Beendet den Debugger. Auch ein EOF der Debugger-Eingabe beendet den Debugger.

`r`
: Rückkehr aus der aktuellen Subroutine.

`R`
: Neustart des Debuggers.

`s` [*Ausdruck*]
: Einzelschrittsteuerung.

`source` *Datei*
: Führt die Debugger-Befehle in der benannten Datei aus.

`S` [`!`] *Muster*
: Führt die Namen aller Subroutinen auf, die dem Muster [nicht] entsprechen.

`t`
: Wechsel des Trace-Modus.

`t` *Ausdruck*
: Trace durch die Ausführung von *Ausdruck*.

`T` Ausgabe eines Stack-Trace.

`v [ Zeile ]`
Gibt eine Bildschirmseite von Zeilen um die angegebene Zeile aus.

`V [ Paket [ Muster ] ]`
Gibt die Variablen aus, die dem *Muster* in einem *Paket* entsprechen. Voreingestellt ist das Paket `main`.

`w Ausdruck`
Fügt einen globalen Watch-Ausdruck hinzu.

`W [ Ausdruck ]`
Löscht den globalen Watch-Ausdruck. Wird der *Ausdruck* mit * angegeben, werden alle Watch-Ausdrücke gelöscht.

`x [ Tiefe ] Ausdruck`
Evaluiert *Ausdruck* im Listenkontext und »dumpt« das Ergebnis. Mit *Tiefe* wird der Dump auf *Tiefe* Level beschränkt.

`X [ Muster ]`
Wie `V`, geht aber vom aktuellen Paket aus.

`y [ n [ Muster ] ]`
Wie `V`, führt aber die lexikalischen Variablen in höheren Geltungsbereichen (*n*) auf. Benötigt das optionale Modul `PadWalker`.

`.` Kehrt zur ausgeführten Zeile zurück.

`-` Gibt das vorige Fenster aus.

`= [ Alias [ Wert ] ]`
Setzt einen Alias, fragt ihn ab oder gibt alle aktuellen Aliase aus.

`/ Muster [ / ]`
Sucht vorwärts nach *Muster*.

`? Muster [ ? ]`
Sucht rückwärts nach *Muster*.

`< Befehl`

Legt eine Aktion fest, die jedes Mal ausgeführt werden soll, bevor der Debugger-Prompt erscheint. Lautet der *Befehl* ?, werden die aktuellen Aktionen ausgegeben. Mit * werden alle Aktionen gelöscht.

`<< Befehl`

Fügt einer Liste von Aktionen eine weitere hinzu. Diese Aktionen werden jedes Mal ausgeführt, bevor der Debugger-Prompt erscheint.

`> Befehl`

Legt eine Aktion fest, die nach jedem Debugger-Prompt ausgeführt wird. Lautet der *Befehl* ?, werden die aktuellen Aktionen ausgegeben. Mit * werden alle Aktionen gelöscht.

`>> Befehl`

Fügt einer Liste von Aktionen eine weitere hinzu. Diese werden nach jedem Debugger-Prompt ausgeführt.

`{ Befehl`

Definiert einen Debugger-Befehl, der vor jedem Prompt ausgeführt wird. Lautet der *Befehl* ?, werden die aktuellen Befehle ausgegeben. Lautet er *, werden alle Aktionen gelöscht.

`{{ Befehl`

Fügt einer Liste von Debugger-Befehlen einen weiteren hinzu. Diese werden vor jedem Prompt ausgeführt.

`! [ [ - ] Nummer ]`

Führt einen Befehl erneut aus. Voreingestellt ist der vorangegangene Befehl.

`! [ Muster ]`

Führt den letzten Befehl erneut aus, der mit *Muster* beginnt.

`!! [ Befehl ]`

Führt den externen *Befehl* in einem Unterprozess aus.

| *Befehl*
: Führt den Debugger-Befehl *Befehl* über den aktuellen Pager aus.

|| *Befehl*
: Wie | *Befehl*, aber es wird mit **select** auch DB::OUT ausgewählt.

Das Drücken der Eingabe- bzw. Return-Taste am Debugger-Prompt wiederholt den letzten s- oder n-Befehl.

Der Debugger verwendet die Umgebungsvariablen DISPLAY, EMACS, LESS, MANPATH, PERL5DB, PAGER, OS2_SHELL, SHELL, TERM und WINDOWID sowie verschiedene andere Variablen, die alle mit PERLDB_ beginnen.

perldebug, perldebtut

ANHANG C
Perl-Links

Organisatorisches

http://www.perl.org/
: Die Heimat von Perl. Hier finden Sie Neuigkeiten und Informationen, Downloads, Dokumentation, Events und mehr.

http://perlfoundation.org/
: Die Perl Foundation. Widmet sich der Weiterentwicklung der Programmiersprache Perl mit offener Diskussion, Zusammenarbeit, Design und Code.

http://www.perl.com/
: Perl News-Site.

http://www.pm.org/
: Heimat der Perl Mongers, der De-facto-Perl-Benutzergruppe.

http://www.perlmonks.org
: Onlinecommunity von Perl-Benutzern.

http://www.yapc.org
: Basisdemokratisches Symposium zur Programmiersprache Perl.

Quellen, Dokumentation, Support

http://www.cpan.org/
: Comprehensive Perl Archive Network (CPAN).

http://search.cpan.org/
: CPAN-Suchmaschine.

http://lists.perl.org/
: Umfangreiche Sammlung Perl-bezogener Mailinglisten.

http://bugs.perl.org/
: Die Bug-Datenbank für Perl.

http://history.perl.org/
: Homepage von CPAST und der Perl-Zeitleiste.

News, Blogs, Publikationen

http://johan.vromans.org/perlref.html
: Homepage der *Perl Pocket Reference*, dem Original dieser Übersetzung.

http://johan.vromans.org/
: Homepage des Autors.

http://www.theperlreview.com/
: Die Perl Review.

http://perlnews.org
: Perl-Newsportal.

http://p3rl.org/
: Kurz-URL-Dienst für die Perl-Dokumentation.

http://perlsphere.net/
: Ein weiter Perl Blog-Aggregator.

Plattformen, Distributionen

http://www.enlightenedperl.org/
: Widmet sich der Suche und Verbesserung der besten Module.

http://www.citrusperl.org
: Eine anwendungsorientierte Perl-Distribution.

http://win32.perl.org
: Homepage der Win32-Perl-Community.

Index