Perl - universelle Skriptsprache

1. Allgemeines

Joachim Backes

joachim.backes [at] rhrk.uni-kl.de

https://www-user.rhrk.uni-kl.de/~backes

Arbeitsschwerpunkte während der aktiven Dienstzeit:

- Systemmanagement LINUX-Dialog-Server (Fedora, z.B. lindb/lindc)
- Systemmanagement weiterer LINUX-Server
- Systemmanagement LINUX-Cluster (Redhat Enterprise LINUX / CentOS)
- AIX-Cluster
- Systemprogrammierung in C, C++, Perl, bash, ...
- Download-Services (**StarOffice**, **SOPHOS**, **MATLAB** und **NAI** Antiviren-Software, ...)
- GUI-Programmierung (X-Windows, OSF/Motif®, gtk+,...)
- HP-UX
- Apollo/Domain, SunOS, Solaris,...
- BS2000

...

Kurzreferenz zu Perl-5.004 als Booklet:

http://webdocs.cs.ualberta.ca/~lindek/perl_quickref.pdf

Dokumentation auf UNIX/Linux-Systemen:

Die Man-Pages zu Perl sind auf den **LINUX-**Dialogservern **linda und lindb** verfügbar.

Mit dem Kommando *man perl* erhält man einen **Einstieg in die Perl-Dokumentation**; anhand der dort angegebenen Verweise ist der Zugang zur Gesamt-Dokumentation (teilweise auch im Internet!) problemlos möglich.

Die Man-Page lässt sich auf UNIX®oiden System ausdrucken mit :

man -t manpage | print-kommando

Literatur:

Perl – Eine Einführung. Aus der RRZN-Schriftenreihe der UNI Hannover.

Perl. Einführung, Anwendungen, Referenz, 2. Auflage
Verfasser: Farid Hajji.
ISBN: 3-8372-1535-2
ADDISON-WESLEY, 2000

Themen

- Einführung
- Datentypen und Terme
- Pattern-Matching
- Operatoren
- Anweisungen
- Deklarationen
- Unterroutinen
- Referenzen
- Packages
- Objektorientierte Programmierung

2. Einstieg

PERL steht für:	**P**ractical **E**xtraction and **R**eport **L**anguage (Praktische Extraktions- und Berichtssprache)
Charakterisierung: von	Bietet umfangreiche Hilfsmittel zur **vereinfachten Bearbeitung Standard–Tasks,** aber auch größerer Aufgaben wie z.B. Systemmanagement. Perl-Programme sind (als Text-Skripte!) **auf allen gängigen Betriebssystemen** lauffähig. Perl ist **GNU-GPL-basiert**!
Verfügbarkeit:	alle UNIX/Linux-Betriebssysteme, alle Windows-Varianten, MS-DOS, VMS, OS/2, Macintosh.
Arbeitsweise: analysiert und	**Compilierphase**, d.h. das Programm wird eingelesen, compiliert (mit Ausgabe von Syntaxfehlern). Danach folgt die **Ausführungsphase.**
Fazit:	Eine Mischung von **Compiler** und **Interpreter** zum Schreiben **portabler Programme**!

Ausführung von PERL-Scripts (unter UNIX/LINUX)

1. **Direkter Aufruf, inline**:

   perl -e 'print "Hallo, Welt!\n";'

2. **PERL-Skript in Datei** (z.B. **hallo.pl**) mit explizitem Aufruf des Interpreters:

   print "Hallo, Welt!\n";

   Aufruf: perl hallo.pl

3. **Perl-Skript in ausführbarer Datei** (z.B. **hallo.pl**, mit folgendem Inhalt):

   #!/usr/bin/perl ≙ *Pfad des Perl-Interpreters*
   print "Hallo, Welt!\n";

   Aufruf: ./hallo.pl oder auch: perl hallo.pl

Aufruf und Optionen des PERL-Interpreters:

perl [optionen] – [scriptdatei] [argumente]

Schalter	Bedeutung
-a	Einschalten des **autosplit**-Modus (zusammen mit den Optionen -n oder -p): Splittet $_ in das Feld @F
-C	Ermögicht es PERL, "***native wide character API's***" zu verwenden.
-c	Nur Syntaxchek durchführen (BEGIN- und END-Blöcke werden aber durchlaufen!)
-d[*:debugger*]	Startet das PERL-Skript unter einem Debugger
-D[*number*|*letters*]	Setzt Debugging-Flags (Argument ist eine Bitmaske oder Flags)
-e '*perlcode*'	Inline-Skripting, d.h. das PERL-Skript steht zwischen den Hochkommas und nicht in einer Datei: Mehrere "-e"-Optionen sind erlaubt; "-e" erspart das Anlegen von Source-Dateien und ist gut für rapid Prototyping geeignet.
-F*/pattern/*	Musterangabe für das "***Autosplit***", welches mit -a aktiviert wird; "//" ist optional
-i[*extension*]	Die Dateien, die man im PERL-Programm mit <...> angegegeben hat, sollen "in-place" editiert werden (Kopie in Hilfsdateien, die man nicht selber erstellen muss). Falls eine Erweiterung unter "extension" angegeben wurde, wird diese als Erweiterung verwendet zum Herstellen von Backup-Kopien.
-l*direcory*	Angabe eines Direcrories für Include-Angaben (@INC), darf mehrfach angegeben werden.
-l[*octal*]	Schaltet die automatische Verarbeitung des Zeilenendes ein (es wird z.B. automatisch entfernt). Bei der Angabe eines Zeilentrenners (oktal!) wird dieser bei der Ausgabe entfernt.
-m *module[=<list>]* *-M module[=<list>]*	Automatisches Ausführen von 'use|no module', bevor das Skript ausgeführt wird. <list> ist eine optionale, mit Komma getrennte Liste von Items.
-n	Erweitert das Skript automatisch um 'while (<>) {<script>}'
-p	Erweitert das Skript automatisch um eine Iteration über angegebene Dateinamen, vergleichbar mit dem Programm sed.

Schalter	Bedeutung
-s	Falls hinter dem Skript-Namen, aber noch vor den Dateinamen, Schalter der Form "--..." angegeben sind, werden die Schalter nicht als Dateiname übergeben, sondern sie tauchen im Programm als Variablen auf.
-S	Der PERL-Interpreter sucht das Programm unter Verwendung der PATH-Variable!
-T	Schattet das "**Tainting**" ein (überprüft das Programm auf unsicheres Verhalten und bricht es ggfls. ab).
-t	Wie -T, gibt aber nur Warnungen aus und bricht nicht ab.
-U	Erlaubt die Verwendung unsicherer Operationen
-v	Versionsnummer, Patchnummer und weitere wichtige PERL-Informationen werden ausgegeben.
-V[:Variable]	Ausgabe des Wertes von PERL-Konfiguarations-Variablen nach STDOUT
-w/-W	Warnungen während der Compilierung des Skript beim Auftreten dubioser Konstrukte (gesteuert durch __WARN__). Bei -W werden die Warnungen bedingungslos ausgegeben. Empfohlen!
-X	Alle Warnungen unterdrücken.
-x[directory]	Jeglichen Programmtext vor einer Zeille ähnlich "#!/perl..." ignorieren. Bei der zusätzlichen Angabe eines Directories wird vor der Skript-Ausführung ein cd in dieses Diectory ausgeführt.

Format des Perl-Skriptes

- Perl ist weitgehend formatfrei
- Als **Kommentar** zählt alles zwischen einem ungeschützten Nummernzeichen (#) und dem Zeilenende, d.h. **es gibt keine mehrzeiligen Kommentare** wie z. B. in C.
- Einfache Anweisungen sind durch ein **Semikolon** (;) zu **trennen**.
- **Leerzeichen** und **Leerzeilen** außerhalb von Strings sind **irrelevant**!

Variablen

Variablen müssen in Perl nicht deklariert (anders als z.B. in C) und auch nicht initiaisiert werden (können aber!). Je nach Kontext werden sie mit einem ***Null-Wert*** angelegt: "" oder 0. Abhängig von der Verwendung werden sie als *Strings*, *Zahlen* oder *Boolsche Werte* (wahr/falsch) interpretiert. Der Variablentyp wird durch das erste (Sonder-)Zeichen des Namens festgelegt:

Typ	1. Zeichen	Beispiel	Ist ein Name für
Skalar	$	$kurs	einen individuellen Wert (Zahl oder String)
Array	@	@gar	eine Liste von Werten mit einer ganzen Zahl als Index
Hash	%	%zins	eine Gruppe von Werten mit einem String als Index
Unterrroutine	&	&up	ein aufrufbares Stück Perlcode
Typeglob	*	*SUPPE	**alles** namens SUPPE

Skalar: Die Wertzuweisung an Variable erfolgt mit dem Operator "=":

Beispiele:

```
$antwort = 42;                  # Integerwert
$pi = 3.14159265;               # Realzahl
$wiza = 6.02e23;                # Zahl in wissenschaftl.
                                # Notation
$wuestenschiff = "Kamel";       # String
```

Quoting-Mechanismen:

Zeichen	Bezeichnung	Bedeutung
"..."	Anführungszeichen	Auswertung von Variablen und Interpretation von Backslashes
'...'	Hochkommata	Keine Auswertung von Variablen und keine Interpretation von Backslashes
`...`	Backquotes	Ausführung eines Systemkommandos mit Rückgabe des Ausgabetextes als Wert. Auswertung von Variablen.

Beispiele:

```
$tier = "Ochse";
$s = "Wo ist das $tier?\n";          # Strings mit
                                     # Interpolation/Interpretation

$k = 'Es kostet $100';               # Strings ohne
$k = 'Es kostet $100\n';             # Interpolation/Interpretation

$cwd = `pwd`;                        # Textausgabe eines Kommandos
$dann = $wann;                       # Noch eine Variable
$dann = "$wann"                      # Das gleiche, falls $wann
                                     # einen Text-String enthält
```

Ein **Array** ist eine numerisch geordnete Liste, bestehend aus einer beliebigen Kombination von Skalaren, wie z.B. Zahlen und/oder Strings. Der Zugriff erfolgt mittels eines numerischem Index (**beginnt standardmäßig bei 0**).

Beispiele:

```
# Zuweisungen
@home = ("couch", "chair", "table", "stove");
@number = (1.2, 11, 4.01e11,311);
@misc = ("couch", 1, "chair", 8);
($potato, $lift, $pipe, $play) = @home;  # entspricht
$potato = "couch"; ... $play = stove;
$nhome[0] = "couch"; $nhome[3] = "stove";
# entspricht der 1. Zuweisung, das Array @nhome wird dadurch
# implizit angelegt

# Vertauschung von 2 Variablen
($alpha, $omega) = ($omega, $alpha);

$#array     # letzter Index des Arrays @array (Wert: 3)
@array      # gesamtes Array, im skalaren Kontext
# Anzahl der Elemente von
# @array, z.B. $number = @home; Hat den Wert 4
```

Arrayfunktionen: pop, push, shift, unshift

Ein **Hash** ist als Array eine **ungeordnete Liste von Skalaren; der** Zugriff erfolgt über einen Stringwert als Index (Schlüssel oder **Key** genannt - asso-ziatives Array) statt über einen numerischen Feldindex. Im Gegensatz zu einem **Array** besitzt ein **Hash defaultmäßig** keinen Anfang und kein Ende.

Beispiele:

```
# Zuweisungen: Schlüssel-/Wertpaar

%longday = (    "Mo", "Montag", "Di", "Dienstag",
                "Mi", "Mittwoch", "Do", "Donnerstag",
                "Fr", "Freitag", "Sa", "Samstag",
                "So", "Sonntag");
# Oder in anderer Schreibweise, => ersetzt Komma
%longday = (      "Mo" => "Montag", #...
                  "So" => "Sonntag");
# Zugriff auf Hash-Element
$woday = $longday{"Mi"};       # Skalar
$frau{"Adam"} = "Eva";         # Adams Frau ist Eva
```

Hashfunktionen: **keys, values**

Beispiel: Einlesen einer Studentendatei mit Namen "noten". Jede Zeile besteht aus einem Namen und, mit Zwischenräumen getrennt, die bei Prüfungen erreichte Punktzahl:

Ben	76
Clementine	49
Norma	66
Chris	66
Doug	42
Ben	12
Clementine	0
Norma	66

Die Aufgabe besteht darin, für jeden Studenten das arithmetische Mittel seiner erreichten Punkte zu berechnen und diese errechnete Liste sortiert auszugeben:

Programm:

```
01    #!/usr/local/bin/perl
02    # Studentendatei in einen Hash einlesen
03    open(NOTEN,"noten") or die "Kann 'noten' nicht öffnen, Error = $!\n";
04    while ($line = <NOTEN>)
05    {
06            chomp $line;                                  # newline entfernen
07            ($student, $note) = split(/\s+/, $line);      # \s: Whitespace(s)
08            $noten{$student} .= $note . " ";              # %noten wird implizit angelegt
09    }
10    # Mittelwerte berechnen
11    foreach $student (sort keys %noten)                   # sortiert Studentennamen (über Keys)
12    {
13            $notenzahl = 0;                               # Zählt, wieviele Noten der Student hat
14            $notensumme = 0;                              # Summe alle seiner Noten
15            @noten = split(/ /,$noten{$student});         # Achtung: @noten vs. %noten !!
16            foreach $note (@noten)
17            {
18                    $notensumme += $note;
19                    $notenzahl++;
20            }
21            $mittelwert = $notensumme / $notenzahl;
22            print "$student: $noten{$student}\tAverage: $mittelwert\n";
23    }
```

Ausgabe:

```
Ben: 76 12          Average: 44
Chris: 66           Average: 66
Clementine: 49 0    Average: 24.5
Doug: 42            Average: 42
Norma: 66 66        Average: 66
```

Filehandles

Filehandles symbolisieren im aktiven Zustand eine offene Datei, ein laufendes Gerät, eine offene Pipe oder einen offenen Socket, um Ein- und Ausgabe durchzuführen.

Vordefinierte (und beim Start des Perl-Programmes bereits geöffnete) Dateihandles sind:

STDIN	Eingabekanal
STDOUT	Ausgabekanal
STDERR	Kanal für Fehlermeldungen

Konvention: Namen für Dateihandles sollten nur aus **Großbuchstaben** bestehen!

Filehandles werden mit der open-Funktion erstellt (aktiviert, geöffnet).

Beispiele:

```
open(SESAM,"dateiname");          # Öffne Datei zum Lesen
open(SESAM,">dateiname");         # Erzeuge Datei neu zum Schreiben
open(SESAM,">>dateiname");        # Öffne und hänge an existierende Datei an

open(SESAM,"|ausgabe-pipe");      # Richte Ausgabefilter ein
open(SESAM,"|mail user";          # Beisp.: sende Mail an user

open(SESAM,"eingabe-pipe|");      # Richte Eingabefilter ein
open(SESAM,"ls -l|");             # Beisp.: Lese Dateiliste
```

Das **Schließen** eines Filehandles geschieht mit der **close**-Funktion, z.B.

```
close (SESAM)
```

Das **Öffnen eines bereits geöffneten** Dateihandles schließt die erste Datei **implizit:**

```
open(INPUT, "datei1");          # öffnet die Datei "datei1" zum Lesen
...
open(INPUT, ">datei2");         # schließt die Datei "datei1" und
                                # öffnet dann die Datei "datei2" zum
                                # Schreiben
```

Das **Lesen** mittels eines Filehandles kann mit dem Zeileneingabe-Operator <> erfolgen:

```
print STDOUT "Geben Sie eine Zahl ein: ";        # Fordere
                                                 # eine Zahl an.
$number = <STDIN>;                               # Lese die Zahl ein
print STDOUT "Die Zahl lautet $number\n";        # und gebe sie aus.
```

STDIN bzw. **STDOUT** sind die vordefinierten Filehandles für Standard-Ein– bzw. Ausgabe, man kann sie in dem entsprechenden Umfeld (wenn eindeutig) auch weglassen:

```
print "Geben Sie eine Zahl ein: ";      # Fordere eine Zahl an.
$number = <>;                           # Lese eine Zahl ein
print "Die Zahl lautet $number\n";      # und gebe sie aus.
```

Weitere nützliche Funktionen in Zusammenhang mit der Eingabe:

chop(): entfernt das letzte Zeichen in einer Variablen. Das entfernte Zeichen ist der Funktionswert.

chomp(): entfernt in einer Variablen den sog. End-of-Record-Marker, i.a. "\n"; Funktionswert ist die Anzahl der entfernten Zeichen.

Arithmetische Operatoren

Beispiel	Bedeutung	Ergebnis
$a + $b	Addition	Summe von $a und $b
$a - $b	Subtraktion	Differenz von $a und $b
$a * $b	Multiplikation	Produkt von $a und $b
$a / $b	Division	Quotient von $a und $b
$a % $b	Modulo	Rest von $a dividiert durch $b
$a ** $b	Potenzierung	$a hoch $b

Vorrangregelung:

1. Potenzierung
2. Multiplikation, Division, Modulbildung
3. Addition, Subtraktion

String-Operationen

1. "String-Addition" (Konkatenation):"."

```
$a = 123; $b = 456;
print $a + $b;                  # Gibt 579 aus (numerisch)
print $a . $b;                  # Gibt 123456 aus (als String)
```

2. "Multiplikation" (Repeat):"x"

```
$a = 123;
$b = 3;
print $a * $b;                  # Gibt 369 aus.
print $a x $b;                  # Gibt 123123123 (3 Mal "123" als Text) aus
```

Zuweisungsoperator "="

Berechnet den Wert des Ausdrucks auf der rechten Seite und weist ihn der linksstehenden Variablen zu: "wird gesetzt auf" (Nicht zu verwechseln mit dem numerischen Vergleichsoperator "==", der "ist gleich?" bedeutet!)

```
$a = $b;
$a = $b + 5;
$a = $a * 3;            # alternativ: $a *= 3;
```

Kurzschreibweise: op= expr # entspricht: lvalue = lvalue op expr

[lvalue: veränderbare Speicherstelle (left **value**; Variable, Arrayelement,...)]

```
$line .= "\n";          # Zeilenvorschub an $line anhängen.
$fill x= 80;            # $fill soll sich selbst 80 Mal wiederholen.
$val ||= "2";           # Setze $val auf 2, wenn $val undefiniert ist
$val = $val || "2";     # Identisch
```

Autoinkrement- und Autodekrement-Operatoren

Beispiel	Bedeutung	Ergebnis
++$a $a++	Autoinkrement,	$a **vor** der Bewertung um 1 erhöhen $a **nach** der Bewertung um 1 erhöhen
--$a $a--	Autodekrement	$a **vor** der Bewertung um 1 vermindern $a **nach** der Bewertung um 1 vermindern

In Abhängigkeit von der Stellung und dem Typ des Operators wird der Wert der Variablen geändert:

vorgestellt: erst Änderung, dann Bewertung
nachgestellt: erst Bewertung, dann Änderung

```
$a = 5;                 # $a wird der Wert 5 zugewiesen
$b = ++$a;              # $a += 1; $b = $a; $b hat Wert 6
$c = $a--;              # $c = $a; $a -= 1; $c hat Wert 6, $a 5
$d = "abc";
$d++;                   # wird zu "abd", alphanumerisches Inkrement
```

Logische Operatoren

Beispiel	Bedeutung	Ergebnis
$a && $b $a and $b	Logisches UND	TRUE, wenn $a und $b TRUE; FALSE sonst
$a || $b $a or $b	Logisches ODER	TRUE, wenn $a oder $b TRUE; FALSE sonst
!$a not $a	Logische Verneinung	FALSE, wenn $a TRUE; TRUE, wenn $a FALSE

Ein Operand wird als FALSE bewertet, wenn er (ggfls. nach einer Konversion String inZahl) numerisch den Wert 0 besitzt, undefiniert ist oder eine leeren String darstellt (beachte "**00**" hat den log. Wert TRUE, aber "0" hat FALSE als log. Wert).

Die Operatoren &&, ||, und ! binden stärker als die verbalen Äquivalente.

Zeile aus unserem Beispiel:

```
open(NOTEN, "noten") or die "Kann Notendatei nicht öffnen: $!\n";
```

Wenn die Datei "noten" geöffnet werden kann, erfolgt ein Sprung zur nächsten Anweisung des Programms, andernfalls wird eine Fehlermeldung ausgegeben und die Ausführung beendet nach dem Motto: "Öffne *noten* oder stirb!"; Die Variable **$!** enthält in diesem Fall die **vom Betriebssystem zurückgegebene Fehlermeldung in Textform**.)

Vergleichsoperatoren

werden auch relationale Operatoren genannt. Sie dienen dem Vergleich zweier skalarer Werte (zwei Arten, Zahlen oder Strings):

Ist $a das linke und $b das rechte Argument, so hat man folgende Aufstellung:

Abfrage	Als Zahlen	Als String	Rückgabewert
Gleich	$a == $b	$a eq $b	TRUE, wenn $a gleich $b ist
Ungleich	$a != $b	$a ne $b	TRUE, wenn $a ungleich $b ist
Kleiner als	$a < $b	$a lt $b	TRUE, wenn $a kleiner als $b ist
Größer als	$a > $b	$a gt $b	TRUE, wenn $a größer als $b ist
Kleiner gleich	$a <= $b	$a le $b	TRUE, wenn $a kleiner oder gleich $b ist
Größer gleich	$a >= $b	$a ge $b	TRUE, wenn $a größer oder gleich $b ist
Vergleich	$a <=> $b	$a cmp $b	0, wenn $a und $b gleich sind. 1, wenn $a größer als $b ist und -1, wenn $a kleiner $b ist.

Auswahl an Prüfoperatoren für Dateien (Überprüfung von bestimmten Dateiattributen)

Beispiel	Dateizustand	Ergebnis
-e $a	Existiert	TRUE, wenn die in $a benannte Datei existiert
-r $a	Ist lesbar	TRUE, wenn die in $a benannte Datei lesbar ist
-w $a	Ist beschreibbar	TRUE, wenn die in $a benannte Datei beschreibbar ist
-d $a	Ist ein Verzeichnis	TRUE, wenn die in $a benannte Datei ein existierendes Verzeichnis ist
-f $a	Ist lesbare Datei	TRUE, wenn die in $a benannte Datei eine existierende reguläre Datei ist
-T $a	Ist Textdatei	TRUE, wenn die in $a benannte Datei eine existierende Textdatei ist

Wahrheitswert

Der Wahrheitswert wird bei Perl immer in einem **skalaren Kontext** evaluiert.

Regeln:

1. Jeder String ist wahr, mit Ausnahme von "" und "0".
2. Jede Zahl ist wahr, mit Ausnahme von 0.
3. Jede Referenz (später!) ist wahr.
4. Jeder undefinierte Wert ist falsch.

```
0               # Falsch, nach Regel 2
1               # Wahr, da nicht 0, nach Regel 2
10-10           # 10-10 ist 0, also falsch.
"0"             # String "0", also falsch.
""              # Nullstring (leerer String), falsch nach Regel 1.
"0.00"          # String "0.00", weder leer, noch genau "0", also wahr
"0.00" + 0      # Zahl 0 (erzwungen durch +), also falsch.
\$a             # Referenz auf $a, also wahr, auch wenn $a falsch
```

Block

Unter einem **Anweisungsblock** verstehen wir eine Reihe von Anweisungen (mit ";" voneinander getrennt), die zwischen einer öffnenden und einer schließenden geschweiften Klammer stehen:

```
{
        # Block
}
```

if- und unless-Anweisung

Die **if**-Anweisung wertet einen Bedingungsausdruck aus und führt einen Block aus, wenn die Bedingung erfüllt (also wahr) ist:

```
if ( <logische oder numerische bedingung> )
{                           # then-Block / true-Block
        ...
}
else
{                           # else-Block / false-Block
        ...
}
```

Beispiel:

```
print "Wie alt sind Sie? ";
chomp($a = <STDIN>);
if ($a < 18)
{
        print "Sie sind zu jung zum Wählen!\n";          # then-Block
}
else
{
        print "Sie sind alt genug! Wähle!\n";            # else-Block
}
```

Ein **else-Block muss nicht vorhanden sein**:

```
print "Wie alt sind Sie? ";
chomp($a = <STDIN>);
if ($a < 18)
{
	print "Sie sind zu jung zum Wählen!\n";
}
```

Ist nur der else-Block relevant ("mache jenes, wenn dieses *falsch* ist"), so kann das mit der **Variante unless** oder **if !** ausgedrückt werden:

```
print "Wie alt sind Sie? ";
chomp($a = <STDIN>);
unless ($a >= 18)
{
	print "Sie sind nicht alt genug!\n";
}
```

Gibt es **mehr als zwei Auswahlmöglichkeiten**, so kann man an die **if**-Anweisung auch noch mehrere **elsif-Zweig** anhängen:

```
if ($city eq "MA")
{
        print "MA liegt nordöstlich von KL.\n";
}
elsif ($city eq "SB")
{
        print "SB liegt südwestlich von KL.\n";
}
elsif ($city eq "PS")
{
        print "PS liegt südlich von KL.\n";
}
else
{
        print "Ich weiß leider nicht, wo $city liegt.\n";
}
```

Die **if-** und **elsif**-Klauseln werden nacheinander durchlaufen, bis eine gefunden wird, die wahr ist, oder bis die **else**-Bedingung erreicht wird. Wenn eine der Bedingungen wahr ist, wird der zugehörige Block ausgeführt, und die **restlichen Verzweigungen werden nicht abgefragt**.

while- und until-Schleifen

Zuerst wird der Bedingungsteil der Anweisung geprüft. Ist die Bedingung erfüllt (d.h. wahr bei **while** und falsch bei **until**), wird der darauffolgende Anweisungsblock ausgeführt. Danach erfolgt der nächste Eintritt in die while-Abfrage und ggf. auch in den Schleifenrumpf.

```
while ($tickets_verkauft < 10000)              # Vor dem Rumpf
{
        $vorhanden = 10000 - $tickets_verkauft;
        print "$vorhanden Tickets sind noch verfügbar. ";
        print "Wie viele möchten Sie? ";
        chomp($kaufen = <STDIN>);              # Hier fehlt eine Abfrage
                                               # auf korrekte numerische Eingabe!!
        $tickets_verkauft += $kaufen;
}                                              # oder

until ($tickets_verkauft >= 10000) { ... }     # dem Rumpf nachgestellt, ist eine Alternative
```

Vgl. Zeile 4 aus einem früheren Beispiel:

```
while ($line = <NOTEN>) { ... }
```

for-Schleife

Eine **for**-Schleife läuft wie eine **while**-Schleife ab, hat aber anderes Layout (wie in den Sprachen C/C++/JAVA):

Beispiel:

```
for ($verkauft=0;                    # Anfangswert für Schleifenvariable
     $verkauft<10000;                # Prüfung der Schleifenbedingung
     $verkauft+=$bestellt;)          # Anpassung der Schleifenvariable
                                     # an den aktuellen Stand
{
     $vorhanden = 10000 - $verkauft;
     print "$vorhanden Tickets sind noch verfügbar. ";
     print "Wie viele möchten Sie? ";
     chomp($bestellt = <STDIN>);
}
```

foreach-Schleife

foreach (**Synonym für for)** wird oft genutzt, um den gleichen Code für jedes Element eines Arrays auszuführen.

Beispiel:

```
@a = (1,2,3,4,5);
foreach $b (reverse @a)
{
	print "$b\n";
}
```

Es kann ohne semantische Änderung **foreach** durch **for** ersetzt werden.

Häufig vorkommende Konstruktion: Schleife über die sortierten Schlüssel eines Hashes:

```
foreach $key (sort keys %hash)
{
	...# Mache irgendetwas mit den Keys oder dem Hash
}
```

next und **last**

Mit beiden Anweisungen kann der Ablauf einer Schleife beeinflusst werden:

next gehe zum Ende der aktuellen Iteration und starte nächsten Durchlauf

last bricht die Schleife ab.

Reguläre Ausdrücke bieten:

- die Möglichkeit, Zeichenmuster in Strings zu suchen, z.B. **/foo/**
- einen Substitutionsoperator (Stringersetzung): z.B. **s/foo/bar/**
- eine **split**-Funktion: ein regulärer Ausdruck legt das/die Trennzeichen fest:

```
($student, $grade) = split(/\s+/, $line);
```

Zeichenklassen		
Bezeichnung	**Beinhaltet**	**Kurzzeichen**
Whitespace	[\t\n\r\f]	\s
Wortzeichen	[a-zA-Z_0-9]	\w
Ziffer (Digit)	[0-9]	\d

Es werden nur ***einzelne*** Zeichen überprüft: \w prüft auf ein einzelnes Wortzeichen, nicht auf ein vollständiges Wort.

Quantifier:

Zeichen	Bedeutung
+	1 oder mehr Treffer
?	0 oder 1 Treffer
*	0 oder mehr Treffer
{N,M}	zwischen N und M Treffer
{N}	genau N Treffer
{N,}	mindestens N Treffer

"+" entspricht {1,}, "?" {0,1} und "*" {0,}. Quantifier werden nachgestellt.

Beispiele:

```
/x{5,10}/                   # zwischen 5mal und 10mal "x"
/\d{7}/                     # genau 7 Ziffern

$_ = "fred xxxxxx barney";

s/x*//;                     # keine Auswirkung,
                            # $_ behält seinen Wert, da x* das "" zu Beginn
                            # des Strings als erstes erkennt!
s/x+//;                     # alle "x" verschwinden,
                            # $_ = "fred  barney";

/ab*c/;                     # Trifft zu, wenn
                            # "a" gefolgt von keinem
                            # oder mehreren "b"s
                            # und dahinter noch ein "c" auftritt
```

3.Datentypen und Terme

Grundlegender Datentyp ist der **Skalar-Typ** (Zahl oder String). Der Typ wird bei Bedarf kontextabhängig abgeändert.

Daraus entstehenVariablen des Typs:

Vektor (**Feld**, Indizierung mit Zahlen)
Assoziativer Vektor (**Hash**, Indizierung mit Strings).

Vektoren und Hashes: Beide sind mit Listenwerten initialisierbar.

Der Inhalt ist manchmal **undefiniert** (z.B. dann, wenn einer Variablen noch nichts zugewiesen wurde). Die Funktion **defined()** macht Aussagen über eine vorhandene Definition.

Ausdruck:

Stets vom Typ Skalar oder Liste. Der Typ des Ausdrucks ergibt sich aus dem Kontext, insbesondere aus den **Operationen** verknüpfter Operanden.

Listen können auch skalar bewertet werden. Dies ist mit der **scalar( )**-Funktion erzwingbar.

Gültigkeitsbereich:

Standardmäßig: Packageglobal; es sei denn, durch **my/local** spezifiziert (insbesondere für Unterprogramme wichtig).

Bezüge auf

Skalare Variable: Beginnen mit dem **$-Zeichen**
Arrays total: Beginnen mit dem **@-Zeichen**
Hashes total: Beginnen mit dem **%-Zeichen**

Beispiele für Variable	
$days oder ${days}	Einfache skalare Variable
$days[28]	29. Element des Vektors @days
$days{"Feb"}	Wert aus dem Hash %days mit dem Index "Feb"
$#days	Letzter Index von @days
@days oder @{days}	($days[0],$days[1],...)
@days[3,4,5]	($days[3],$days[4],$days[5])
@days{'a','b'}	($days{'a'},$days{'b'})
%days = (key_1,val_1,key_2,val_2,...)	Vollständiger Hash

Jeder dieser Variablen kann ein **Wert zugewiesen** werden (L-Value). Abhängig vom Typ der **linken Seite** findet die Art der Zuweisung statt:
Beispiel:

```
$number = @days;                # Liefert Anzahl der Elemente von @days
```

Ähnlich:

```
$number = $#days;               # Letzer Feldindex von @days. Zuweisung
                                # an $#days ändert die Feldlänge! Damit
                                # lässt sich performant die Länge eines
                                # Feldes manipulieren.
```

Ein Vektor kann auch ganz geleert werden durch

```
@array = ();
```

oder

```
$#array = $[ - 1;
```

Dabei wird mit der Variablen **$[** eingestellt, welchen **Index** in einem Vektor standardmäßig das **erste Element** besitzt (**0 ist Voreinstellung**).

Die aus der Programmiersprache FORTRAN geläufigen **Matrizen** (d.h. mehrdimensionale Vektoren) werden indirekt **unterstützt** durch Felder von Referenzen auf anonyme Felder:

```
@array = ([1,2,3,4],[5,6,7,8],[9,10,11,12]);
```

Bezeichner

Verschiedene Datentypen haben ihren eigenen Namensbereich, d.h. **$x** und **@x** bezeichnen zwei unterschiedliche Dinge, auch **$x** und **$x[0]**. **Konflikte** mit reservierten Wörtern sind wegen $,% und @ **nicht möglich**.

Tipp: Labels und Filehandles immer mit Großbuchstaben anfangen lassen, dann kann kein Konflikt mit reservierten Wörtern auftreten.

Normale Namen beginnen mit einem **Buchstaben oder** "_"; ihre Länge ist auf 255 Zeichen limitiert, sie dürfen nach dem ersten Zeichen Buchstaben, Ziffern und "_" enthalten. Namen, die mit Ziffern beginnen, dürfen nur aus Ziffern bestehen.

Alle anderen **Namen** sind **auf ein Zeichen limitiert** (z.B. **$[**). Bezeichner können Variablen auch indirekt benennen. Man spricht dann von **Referenzen**.

Numerische Literale

Mit numerischen Literalen definiert man Zahlen der üblichen Formate:

```
12345
12345.67
6.2345e23
0xff
0377
4_123_568
```

(zur Untergliederung, nur bei direkter Angabe, nicht in Strings)

Gleitkommazahlen werden intern **doppelt lang** abgelegt.

Stringliterale

Stringliterale enthalten Zeichenketten, in einfache oder doppelte Hochkommas eingeschlossen; bei der letzten Schreibweise werden skalare und Listen-Variable sowie "\" interpoliert (d.h. ausgewertet, nicht aber bei kompletten Hashes).

```
$x = "123";                 # Skalare Variable
@y = ('a','*b');            # Array-Variable
print "\"${x}xx@{y}\"";     # Interpolation eines Arrays
```

gibt die Zeichenfolge: "123xxa*b" mit Hochkommata aus .

Backslash-Sequenzen	
\t	Tab
\f	Formfeed
\b	Backspace
\a	Bell
\e	Escape
\0123	Oktalfolge aus bis zu 3 Oktalziffern 0...7
\0x64	Hexadezimalfolge aus Hex-Ziffern 0,...f
\cC	Controlcharacter, hier ^C
\u bzw. \l	Nächstes Zeichen groß bzw. klein
\U bzw. \L	alle nachfolgenden Zeichen groß bzw, klein
\Q	Alle nicht-Alphanum-Zeichen mit Backslash versehen
\E	beendet \U, \Q und \L

Alternative Begrenzer	
q// oder ' '	Tab
qq// oder " "	Formfeed
qx// oder ` `	Backticks
qw// oder ()	Wortliste
m// oder //	Mustervergleich
s///	Ersetzung
y/// oder tr///	Translation

Bei diesen alternativen Begrenzern kann der Slash ("/") durch jedes andere Nicht-Blank-Zeichen ersetzt werden:

```
qq(1 2 3 4)        # () ersetzen /
```

bedeutet das gleiche wie

```
qq/1 2 3 4/        # oder
"1 2 3 4"
```

oder

```
s!a!b!
```

das gleiche wie

```
s/a/b/
```

Ist bei einem quotierten String der Zusammenhang klar, können die Quotes auch ganz weggelassen werden (jedoch nicht empfehlenswert):

```
@liste = (a,b,c,d);
```

entspricht

```
@liste = ("a","b","c","d");
```

Here–Dokumente

Die String-Information folgt auf "<< ..." ab der nächsten Zeile. Sie endet, wenn die Zeichenfolge "**...**" erkannt wird. Eine Interpolation findet statt.

- ```
  $x = <<EOF;
  a
  b
  EOF
  ```

  weist der Stringvariablen $x den Wert "a\nb\n" zu;

- ```
  print <<'EOF';
  ${x}
  EOF
  ```

  druckt die Zeichenfolge '${x}' und nicht den Wert von $x;
  wirkt wie

  ```
  print '${x}' . "\n".
  ```

- $x = << `EOC`; # mit Backticks (führt Systemkommandos aus)
  echo "1 2 3"
  EOC

  weist $x den Wert "1 2 3\n" zu, wie

  $x = `echo "1 2 3"`

Weitere literale Tokens	
__LINE__	Aktuelle Zeilennummer (wird nicht interpoliert; außerhalb von Strings!).
__FILE__	Aktueller Dateiname (wird nicht interpoliert, s.o.).
__END__	Markiert für den Perl-Interpreter Ende des Quell-Programms; der Rest bis zum Dateiende kann aber über das Dateihandle **DATA** gelesen werden.
__DATA__	Wie __END__, aber nur innerhalb des Namensraumes

Kontext

Aufgerufene Operationen sind immer vom Kontext abhängig.

Beispiel: Eine Zuweisung an einen Skalar oder Vektor bewertet die rechte Seite im skalaren bzw. vektoriellen Kontext

```
@l = (2,4,6);  # Anlegen eines Arrays über Liste
$x = @l;       # Anzahl der Array-Elemente
@y = @l;       # Kopie eines Arrays
print $x," ",@l,"\n";
```

gibt aus:

```
3 246
```

Die skalare Evaluierung einer Liste läßt sich erzwingen mit der Funktion **scalar**. Im skalaren Kontext findet die Evaluierung abhängig von Datentyp ab: String, Numerisch oder "Egal".

Der **Boolsche Kontext** ist eine Sonderform des skalaren Kontextes.

Ein skalarer Wert ist "wahr", wenn er numerisch von **NULL** verschieden ist bzw. wenn keine leere Zeichenkette im String-Kontext vorliegt (weder 0 noch "" noch "0").

Der skalare **Kontext** "**void**" (d.h. leer) liegt immer dann vor, wenn der Rückgabewert ignoriert wird.

Beispiel: der Rückgabewert einer Funktion wird ignoriert, oder ein Ausdruck wird als Statement verwendet:

```
`Kamelrennen`;   # Rückgabewert uninteressant
                 # Aufruf eine Shell-Skriptes
                 # mit dem Namen Kamelrennen
```

Listenliterale

Listenliterale erhält man durch Trennen einzelner Werte mit Komma (in etwa vergleichbar mit dem Kommaausdruck in C); falls erforderlich, sind die Literale in Klammern einzuschließen.

Unterschied: der skalare Wert eines **Listenliterals** ist der Wert des letzten Elements, aber der skalare Wert einer **Liste** ist die Anzahl ihrer Elemente.

```
$x = ("11","22","33");
@l = ("11","22","33");
$y = @l;
```

Beachte: **$x** hat den Wert "33", aber $y den Wert 3.

Listen werden in Listen interpoliert, aber sie erzeugen **keine Sublisten**! Durch

```
@L = (@l,"44")
```

besitzt die Liste **@L** die Elemente "11","22","33","44". Falls anders gewünscht, sind Referenzen zu verwenden.

Ebenfalls möglich ist:

```
@l = qw(aa bb cc);                  # ist das gleiche wie
@l = ("aa", "bb", "cc");
```

Listenwerte sind indizierbar wie Vektoren:

```
(1,2,3)[0] == 1;                    # ist wahr, oder
$mtime = (stat $file)[9];
```

Paarweise Zuweisung von Listenwerten:

```
($a,$b,$c) = (3, 2, 1);
($map{red},$map{green}) = (0x0f, 0x0e);
my ($a,$b,%rest) = @arg_list;
```

Hashes

Ein **Hash** ist ein Array mit **nichtnumerischen Indizes**. Es gibt keine Hash-literale. Im **Skalarkontext** erhält man den Wert **wahr**, wenn der Hash **nichtleer** ist. Eine Zuweisung erfolgt durch eine normale Liste mit gerader Elementzahl, interpretierbar als Liste von Key-Value-Paaren:

```
%hash = ("ka","va","kb","vb");      # oder
%hash = ("ka"=>"va","kb"=>"vb");
$hash{"ka"} = "va1";
```

Der besseren Übersichtlichkeit halber kann man die Paare auch mit "=>" trennen statt mit Komma (=> ist in Perl ein Synonym für Komma). Diese Syntax ist sogar **bei Funktionsaufrufen verwendbar**. Hashes sind auch im **Listenkontext** verwendbar. Achtung: Die **Reihenfolge der Key/Value-Paare** ist aber dann i.A. unbestimmt!

```
%a = ("1"=>"a","2"=>"b");
@l = %a;
print "@l\n";     # druckt 1a2b, muss aber nicht sein
```

Typeglobs

Typeglobs werden durch einen "*" markiert. Sie repräsentieren einerseits **alle Typen** gleichzeitig (**Globaler Namensraum**, **Alias**-Namen werden in der Symboltabelle angelegt) und dienen andererseits auch dazu, **Filehandles** zu verarbeiten, für die es keinen eigenen Typ gibt mit dem Zweck, Filehandles an Funktionen übergeben zu können oder Filehandles als Rückgabewerte von Funktionen zu verwenden:

```
$fh = *STDOUT;              # richtig, Filehandle
$fhr = \*STDOUT;            # richtig, Referenz auf Filehandle
print $fhr "aaa\n";         # richtig
$fh = STDOUT;               # falsch, wird aber toleriert
$fhr = \STDOUT;             # stets falsch

sub newopen
{
        my $path = shift;
        local *FH;
        open (*FH,$path) or return undef;
        return *FH;             # Kann in main weiterverwendet werden
}
```

Allgemein:

Typeglobs erzeugen weiterhin Namens-Aliase in der Perl-Symbol-Tabelle. So bewirkt die Anweisung

```
*foo = *bar;
```

dass "foo" zu einem **Synonym** für "bar" wird, unabhängig vom Typ:

```
*foo = *bar;            # alles mit Namen bar kann auch unter
                        # foo angesprochen werden
@bar = (1,2,3);
print @foo,"\n";        # druckt 123, da mit @bar identisch
```

Einschränkend:

```
*foo = \$bar;
```

macht **$foo** zu einem Synonym nur für **$bar,** aber nicht für alle Typen!

Sinnvolles Anwendungsbeispiel: getrennte Übergabe zweier Felder an ein Unterprogramm:

Bei der Übergabe an ein Unterprogramm werden zwei hintereinander angegebene Felder zusammengefasst. Um dies zu verhindern, sollte man
(außer man übergibt Referenzen) folgende Technik anwenden:

```
sub addiere_felder
{
    local (*v1,*v2) = @_;   # Übergibt feld1 und feld getrennt
    local ($max_index) = $#v1 > $#v2 ? $#v1 : $#v2;
    local (@summenfeld);
    for (local($i) = 0; $i <= $max_index; $i++)
    {
            $summenfeld[$i] = $v1[$i] + $v2[$i];
    }
    @summenfeld;
}

@feld1 = (1,2,3);
@feld2 = (4,5,6);
@sf = addiere_felder(*feld1,*feld2);
```

Eingabeoperatoren

Eingabeoperatoren werden vom Parser als Terme interpretiert!

Der Backtick-Operator:

Zuerst werden eventuelle Variablen interpoliert, danach die Zeichenfolge an die Shell **/bin/sh** zur Ausführung übergeben (quasi als Literal).

```
$a = `finger $user`;                    # bzw.
$a = qx%finger $user%;
@a = `ls -l`;
```

Achtung: **$user** wird von Perl interpretiert! Die Variable **$?** enthält den **Return-Status** des ausgeführten Shell-Kommandos.

Der Zeileneingabe-Operator (Angle-Operator <...>)

Wirkt auf ein Dateihandle zwischen zwei spitzen Klammern: Die nächste Zeile aus der dazugehörigen Datei wird eingelesen, einschließlich Zeilenende. Der Operator liefert "wahr" zurück, solange das Datei-Ende noch nicht erreicht ist.

Wenn nicht anders angegeben, wird die **Eingabezeile** in **$_** abgelegt. Bei **Schleifen ist ein while-Konstrukt** erforderlich, oder man muss ein Array verwenden: Die Anweisungen

```
while (defined($_ = <STDIN>)) {print $_;}        # oder
while (<STDIN>) {print;}                         # oder
for (;<STDIN>;) {print;}                         # oder
print $_ while defined ($_ = <STDIN>);           # oder
print while <STDIN>;
```

bewirken stets das gleiche.

Die Filehandles **STDIN**, **STDOUT** und **STDERR** sind vordefiniert und geöffnet. Andere Handles sind mit **open()** zu erstellen.

Der Operator **<...>** wirkt kontext-abhängig:

```
$a = <HANDLE>;          # Eine Zeile einlesen
@b = <HANDLE>;          # Alle restlichen Zeilen einlesen
```

legt eine komplette **Zeile in $a** ab, die restlichen **Zeilen in dem Array @b**, wobei jedes Arrayelement von **@b** eine komplette Zeile mit Zeilenende enthält.

Das spezielle **NULL-Handle <>** bewirkt folgendes:

Die Kommandozeilenparameter des Perl-Skripaufrufes (**@ARGV**) werden als Dateinamen interpretiert ("-" zählt als STDIN) und bearbeitet. Ist **@ARGV** leer, wird "-" übergeben.

```
(<>)
{ ... }
```

bewirkt prinzipiell das gleiche wie folgender Perl–Code

```
@ARGV = ("-") unless @ARGV;
while ($ARGV = shift)
{
        open (HANDLE,$ARGV) or next;
        while (<HANDLE>) { ... }
        close HANDLE;
}
```

in den spitzen Klammer weder ein Dateihandle noch ein Skalar (als Typeglob-Variable), dann wird der Term als **Shell-Wildcard-Angabe von Files im aktuellen Directory interpretiert:**

```
while (<*.html>)
{
        chmod 0755,$_;
}
```

Noch einfacher:

```
chmod 0755,<*.html>;
```

Beachte: dieses sog. **Globbing** findet über einen **Shell-Aufruf** statt, daher ist es nicht sehr schnell! Dies ist auch problematisch, falls die Dateiliste zu lang für die Shell wird.

Im skalaren Kontext wird der **nächste Dateiname gemäß Treffer als String** geliefert. Performanter ist eine zusätzliche Variablenzuweisung an ein Array (Verwendung des Operators **glob**):

```
@files = glob "*.html";          # Hier kein Shell-Aufruf!
chmod 0755,@files;
```

4. Reguläre Ausdrücke und Pattern Matching

Die regulären Ausdrücke in Perl sind eng verwandt mit den regulären Ausdrücken, die z.B. von UNIX-Kommandos wie grep oder sed her bekannt sind.

Einsatz: Sowohl beim Suchen nach Strings als auch beim Ersetzen oder Splitten von Strings.

Wozu dient Pattern-Matching (Mustererkennung)?

Pattern-Matching erlaubt die Untersuchung, ob eine Zeichenfolge oder ein Suchmuster, d.h. ein **regulärer oder erweiterter Ausdruck**, in einem String auftreten:

```
if ($text =~ m/^xyz{1,}$/)
{
      # tue irgend etwas,
      # wenn das Muster gefunden wurde
}
```

Ferner lassen sich eine Zeichenfolge oder ein regulärer Ausdruck durch irgend einen anderen Text ersetzen:

```
$text =~ s/a{0,}b//g;      # Löscht gewisse Zeichenfolgen
                           # in einem String
$text =~ s/a*b//g;         # identisch:
                           # Löscht gewisse Zeichenfolgen
                           # in einem String
```

Aufsplitten von Strings in ein Array, wobei Zeichenfolgen oder reguläre Ausdrücke als Trennkriterium verwendet werden:

```
@arr = split /a[c-f][\t\n ]/,$expression;
```

Suchvorgänge sind steuerbar durch **Modifikatoren** hinter der Suchmusterangabe.

Modifikator	Wirkung
i	Führe das Pattern-Matching **unabhängig von Groß-/Kleinbuchstaben** durch
m	**Multiline-Mode:** Ein String darf aus mehreren logischen Zeilen bestehen: für jede einzelnen Teilzeile gibt es mehrere Anker der Form ^ und $ innerhalb des Strings, die jeweils das Suchzeichen "\n" im Stringinnern treffen.
s	Beachte den String als **eine** Zeile. Das Zeichen \n hat keine Zeilentrennende Sonderfunktion und wird wie jedes andere Zeciehn auchg durch das Metazeichen "." gefunden.
x	Das Suchmuster kann zwecks besserer Lesbarkeit **Leerzeichen und Kommentare** (#...) enthalten. Leerzeichen und Kommentarzeichen # haben dann ihre übliche PERL-Bedeutung innerhalb von Kommentaren, dürfen dann aber nicht mit \ quotiert sein.

Zeichenklassen im regulären Ausdruck:

- **Selbstrepräsentierende Zeichen ohne** Sonderbedeutung.
  Beispiel: **/Fred/** liefert "True", wenn die Zeichenfolge "Fred" in dem zu testenden String gefunden wurde.

- **Metazeichen,** oder Zeichen mit vorangestelltem Backslash:

  \<meta> | ( ) [ ] { } ^ $ * + ? .

  Beispiel : **\b** prüft auf Wortgrenze, **\t** auf Tabulatorzeichen, **^** auf Zeilenanfang und **$** auf Zeilenende.

  Oder: /Fred|Wilma|Barney|Betty/ bedeutet, daß jede dieser mit "|" getrennten **Alternativen** den Treffer auslösen kann.

  Gruppierungen mit Vorrangregeln erreicht man mit runden Klammern, beispielsweise /(Fred|Wilma|Pebbles)Flintstone/.

- Einige der Metazeichen sind Begrenzer für **Quantifier,** mit denen man angeben kann, wie oft das davorstehende Objekt im Suchstring auftauchen soll.

  Beispiele: * oder + oder ? oder *? oder {2,5}

  Quantifier werden mit {} eingeschlossen und beziehen sich immer auf das **vorausgehende Atom,** d.h.

  ```
  /muh{3}/
  ```

  meldet nur dann einen Treffer, wenn "muhmuhmuh" gefunden wird.

- **Wichtig: Reguläre Ausdrücke** im Suchmuster werden **vor** dem Suchvorgang **interpoliert**:

  ```
  $foo = "muh";
  m/$foo$/;
  ```

  ist identisch mit

  ```
  m/muh$/;
  ```

Die Suchmechanismen arbeiten mit dem Verfahren des **Backtrackings (Zurückverfolgung!)**: Ergibt sich bei der Vorwärtssuche **kein Treffer**, geht die Suchmaschine **zum letzten Startpunkt** zurück und sucht von dort aus weiter nach Übereinstimmungen.

Folgende Suchregeln gelangen in nachstehender Reihenfolge zur Anwendung:

1. Die Metazeichen und Verankerungssymbole aus den untenstehenden Tabellen werden gemäß ihrer Bedeutung ausgewertet.
2. Prüfen bei Angabe von Quantifiern **{m,n}**, *, + und **?**, welches **Greedy**-Verhalten zum Zug kommt;
3. Prüfen auf **Gruppierungen** per **()**, auf **Zeichenklassen** per **[]** und auf **Metazeichen**, die mit "\" eingeleitet werden.
4. Versuche, Treffer immer so früh wie möglich von links nach rechts zu finden;
5. Versuche bei Misserfolg, Alternativen aus dem Suchmuster heranzuziehen;

Metazeichen	Bedeutung
^	Prüft aus den Anfang eines Strings (oder einer Zeile, wenn mit /m gearbeitet wird)
$	Prüft aus das Ende eines Strings (oder einer Zeile, wenn mit /m gearbeitet wird)
\b	Prüft auf Wortgrenze (Übergang zwischen \w und \W)
\B	Prüft auf Nichtwortgrenze
\A	Prüft auf den absoluten Anfang eines Strings
\Z	Prüft auf das absolute Ende eines Strings
\G	Setze die Suche dort fort, wo ein vorausgegangenes m//g gefunden wurde.
(?=...)	Prüft, ob ... als nächstes erkannt werden würde (Lookahead)
(?!=...)	Prüft, ob ... als nächstes nicht erkannt werden würde (negatives Lookahead)

Zu beachten:

Sequentiell zu erkennende Elemente werden einfach **hintereinander geschrieben**. Es gilt die **Links-nach-rechts-Ordnung**: In einem regulären Ausdruck wie

```
/x*y*/
```

werden für jede durchgespielte Möglichkeit von x, falls erforderlich, zunächst **alle von y** durchgespielt, d.h. weiter **rechts stehende** Teile **variieren schneller**.

^ und $ und lassen Treffer nicht nur am String-Ende zu, sondern auch früher, nämlich dann, wenn die Teilzeile mit \n endet und im Multi-Line-Mode gearbeitet wird. **Geschweifte Klammern in anderem** als im Quantifier–Kontext gelten als normales Zeichen.

n und *m* dürfen nicht größer als 65536 sein.

Das **Greedy-Verhalten kann man abschalten** durch einnachfolgendes **?**. Dadurch wird nach **erstem** Treffer gesucht, und nicht nach dem **maximalen**.

Klammerausdrücke werden ausgewertet, ".", Zeichenlisten in "[...]"

Die Zeichen $ und ^ wirken normalerweise nur am Ende bzw. Anfang eines Strings. Eingebettete NewLines werden nicht erkannt! Dies kann mit dem Modifier /m geändert werden. Sie werden dann erkannt, aber der Punkt findet sie nicht!

Liegen jedoch mehrere Zeilen in einem String, findet der Punkt ein dazwischenliegendes \n nur dann, wenn mit dem Modifier /s gearbeitet wird, d.h. im Single Line Mode.

Beispiele:

```
/^([^ ]+) +([^ ]+)/$2 $1/      # Vertausche 1. und 2. Wort in String,
                               # Treffer in $1 und $2
                               # wegen Klammerung (...)
                               # Beachte: "^"  in "[^ ]" bedeutet Negation
/(\w+)\s*=\s*\1/               # Prüfe auf "irgendwas = irgendwas"
/.{80,}/                       # Prüfe Zeile auf mindestens 80 Zeichen
/^(\d+\.?\d*|\.\d+)$/          # Prüfe auf gültige Zahl im String
                               # wie z.B. 0.123 oder 123 oder 1.234

(/Time: (..):(..):(..)/)       # Ziehe Einzelfelder aus String
{
        $hours = $1;
        $minutes = $2;
        $seconds = $3
}
```

Greedy-Verhalten:

<CENTER> <H2> Heute frische Heringe </H2></CENTER>

sei der zu durchsuchende String. Der reguläre Ausdruck

/<.*>/

erzeugt dann einen Treffer bis zur **letzten schließenden spitzen Klammer** in **</CENTER>**,

/<.*?>/

dagegen nur bis zur ersten **schließenden spitzen Klammer hinter dem ersten <CENTER>** (**ungreedy** erzwungen durch das Fragezeichen "?" hinter "*").

Erweiterte regulärer Ausdrücke

Perl bietet eine konsistente Erweiterung regulärer Ausdrücke: gekennzeichnet durch ein Klammernpaar, welches eine Zeichenfolge umschließt, die mit einem "Fragezeichen" beginnt.

Metazeichen	Bedeutung
(?#text)	**Kommentar**. Text wird ignoriert. Mit dem Switch /x reicht ein #-Zeichen als Einleiter.
(?:...)	Gruppierung wie bei (...), aber ohne Back-Referenz in $1,...
(?=...)	Positive Lookahead-Behauptung ohne eigene Länge, d.h. nicht in $& (d.h. im letzten Treffer) enthalten. Z.B. erkennt /\w+(?=\t)/ ein Wortzeichen, gefolgt von TAB, aber TAB selbst ist nicht in $& enthalten (in $& steht der zuletzt gefundene Treffer)
(?!...)	Negative Lookahead-Behauptung ohne eigene Länge, d.h. nicht in $& (d.h. im letzten Treffer) enthalten. Z.B. erkennt /foo(?!bar)/ alles was mit foo anfängt, ohne dass bar folgt.
(?imsx)	**Modifikatoren**: sind vor das eigentliche Muster zu stellen. Z.B. ignoriert (?i) die Groß-Kleinschreibung im nachfolgenden Muster.

Pattern-Matching-Operatoren

$x =~ m/MUSTER/gimosx oder
/MUSTER/gimosx

Sucht String nach Muster ab und liefert entsprechend TRUE oder FALSE zurück. Wenn kein String $x via "=~" oder "!~" angegeben ist, wird $_ durchsucht.

Modifikatoren	Bedeutung
g	Suche alle Vorkommen (global)
i	Ignoriere Groß-/Kleinschreibung
m	String darf aus mehreren Zeilen bestehen, Multiline-Mode; Der Punkt (.) trifft nicht \n
o	Muster nur einmal kompilieren (Optimierung)
s	Singleline-Mode, trifft \n mit.
x	Verwende erweiterte reguläre Ausdrücke.

Variablen im Suchmuster werden stets **vor** jedem Suchvorgang interpoliert, dabei findet jedesmal eine neue Compilierung statt (langsam)!

Im Listenkontext liefert der Suchoperator **m** die Trefferliste ($1,$2, ...) zurück bei Einschließen der Suchstrings in Klammern. Rückgabe einer **leeren Liste** bei **Mismatch**.

Bei m/.../g im Skalarkontext wird bei jedem Treffer fortlaufend TRUE zurückgegeben, danach FALSE. Dies ist dann von Bedeutung, wenn man in einer Schleife nach fortlaufenden Treffern **in einem String** sucht.

$x =~ ?MUSTER?

wie m//; Suche wird aber von **reset()** zurückgesetzt (siehe m//g; reset wird innerhalb eines continue-Blocks verwendet zum Zurücksetzen von Variablen, die bei Suchvorgängen verändert werden).

$x =~ s/<MUSTER>/<ERSATZ>/egimosx

Durchsucht einen String $x nach <MUSTER> und ersetzt es durch <ERSATZ>. Der Rückgabewert ist die Anzahl der Treffer. Wenn kein String angegeben ist und mit "=~" oder "!~" gesucht wird, wird die Variable $_ durchsucht. Variablen in <MUSTER> und <ERSATZ> werden stets interpoliert. <MUSTER> kann auch ein Nullstring sein; in diesem Fall wird der zuletzt erfolgreich bearbeitete reguläre Ausdruck zum ersetzen verwendet.

Bedeutung der Modifikatoren egimosx	
e	Ersatz ist eine PERL-Codesequenz, die mittels **eval** auf dem Ersatzstring operiert
g	Global ersetzen
i	Groß-/KleinSchreibung ignorieren
m	Multiline-Mode
o	<MUSTER> nur einmal kompilieren, d.h. nachträgliche Änderungen bleiben unberücksichtigt
s	Singleline-Mode
x	Verwende erweiterte reguläre Ausdrücke.

$x =~ tr/<SUCHLISTE>/<ERSETZUNGSLISTE>/cds
$x =~ y/<SUCHLISTE>/<ERSETZUNGSLISTE>/cds

Ersetzt jedes Zeichen der <SUCHLISTE> durch das entsprechende Zeichen in <ERSETZUNGSLISTE>. Bearbeiteter String wie bei m// oder s///.

Modifikatoren:

c Komplement der <SUCHLISTE>;
d Lösche gefundene, nicht ersetzte Zeichen;
s Packe ersetzte aufeinanderfolgende gleiche Zeichen in ein einziges (squeeze [=quetschen])

5. Operatoren

Operatoren dienen der Verknüpfung von Termen.

Bindungs- und Vorrangregeln sind, wie in der nebenstehenden Tabelle beschrieben, festgelegt.

Assoziativität	Operator
Links	Terme und Listenoperatoren (nach links)
Links	->
Keine	++ --
Rechts	**
Rechts	! ~ \ sowie monadisches + und -
Links	=~ !~
Links	* / % X
Links	+ - .
Links	<< >>
Keine	Benannte monadische Operatoren
Keine	< > <= >= lt gt le ge
Keine	== != <=> eq ne cmp
Links	&
Links	| ^
Links	&&
Links	||
Keine	.. ...
Rechts	?:
Rechts	= += -= *= usw.
Links	, =>
Keine	Listenoperatoren (nach rechts)
Rechts	not
Links	and
Links	or xor

Terme

Besitzen den höchsten Rang und bestehen aus **Variablen**, **Quoting** und Quoting-ähnlichen Operatoren, **Ausdrücken in Klammern sowie Funktionen mit Argumenten** in Klammern. **Vorsicht bei Listen-Operatoren** (z.B. print), die mit Klammern verwendet werden: es gelten Vorrang-Regeln wie bei Funktionen, d.h. eine schließende Klammer beendet **vorzeitig** die Liste:

```
rand 10 * 20;          # rand (10 * 20)
rand(10) * 20;         # (rand 10) * 20
```

Aufhebung dieser Regel durch monadisches + vor "(10)":

```
rand +(10) * 20;       # rand (10 * 20); keine Listenregel
```

Achtung bei:

```
@ary=(1,3,sort 4,2,5);     # @ary = (1,3,(sort 4,2,5));
```

Listenoperatoren sind **greedy und neigen zum Verschlingen** aller nachfolgenden Argumente (ggfls. bis zur nächsten schließenden Klammer auf der entsprechenden Ebene!).

Als Terme gelten auch

- **do{}**
- **eval{}**
- der anonyme Array- und Hash-Composer **[]** bzw. **{}**
- der anonyme Unterprogramm-Composer **sub{}**.

Operator ->

Dient der Dereferenzierung bei symbolischen oder festen Referenzen auf Arrays, Hashes und Objektmethoden:

```
$array->[0];                    # Array-Referenz
$hash->{"key"};                 # Hash-Referenz
$object->method(1,2,3);         # Methodenaufruf zum Objekt
class->new();                   # Objekterstellung
```

++ und - -

Autoinkrement und -dekrement, wirken wie bei C/C++:

```
$a++;   # $a wird zuerst bewertet, dann um 1 erhöht
--$a;   # $a wird zuerst um 1 erniedrigt, dann bewertet
```

++/-- sind auch für alphanumerischen Strings der folgenden Zeichenklasse definiert:

```
/^ [a-zA-Z]*[0-9]*$/,
```

wobei dann entweder numerisch oder lexikografisch (im Sinne von Buchstaben) in-/dekrementiert wird. ++ und -- sind aber nicht auf Stringliterale anwendbar.

Operator **

Potenziert. Bindet stärker als Minuszeichen, d.h.

```
-2**4 == -(2**4);
```

Monadische Operatoren: ! ~ \ + -

!	führt eine logische Negation durch. Ist auch mit "not" möglich, dann aber niederprior in Zusammenhang mit weiteren Operanden.
~	führt eine arithmetische Negation durch, und zwar durch bitweise Umkehr, auch bei Strings!
-	Bei numerischem Argument das übliche. Bei Strings wird dieser um ein einleitendes Minuszeichen ergänzt; ist das erste Zeichen des Strings ein + oder-, wird es zu - bzw. +.
\	Erzeugt eine Referenz auf eine Variable.
+	Kein semantischer Effekt. Kann aber auch nach Funktionsnamen vor "(" stehen, um eine Fehlinterpretation der Argumente als Liste zu verhindern.

Bindungsoperatoren: =~ und !~

=~ bindet einen skalaren Ausdruck an das Pattern-Matching; Substitution oder Translation erfolgen mit (m//, s///, y//). Standardmäßig wird bei fehlendem Ausdruck $_ bearbeitet. Handelt es sich bei der rechten Seite um einen Ausdruck, wird er zur Laufzeit zuerst interpoliert (falls nicht mit /o modifiziert wurde).

!~ ist die Negation von =~, d.h.

```
$string !~ /muster/
not ($string =~ /muster/)
```

bedeuten das gleiche.

Beispiele:

```
if ($input =~ m/^1234.*$/) {$input = -$input;}
$input =~ s/AAA/BBB/g;
$input =~ tr/[a-z]/[A-Z]/;
```

Im skalaren Kontext liefert "=~ **m**//" "TRUE" oder "FALSE" zurück, im Listenkontext dagegen die Treffer-Strings als Liste, z.B.

```
if ( ($k,$v) = $string =~ /(\w+)=(\w+)/)
{
        print "Schlüssel = $k, Wert = $v\n";
}
```

Multiplikative Operatoren: * / % x

Multiplikation, Division und Divisionsrest und Vervielfachungs-Operator. Die Division erfolgt nach Umwandlung in das Gleitkommaformat (anders als in C!) Dieses Verhalten kann durch Verwendung des Integermoduls abgeschaltet werden). % wandelt Zahlen vor der Restbildung in ganze Zahlen um.

Der Operator "x" ist der Wiederholungsoperator. Er repliziert im skalaren Kontext den linken Operanden sooft, wie der rechte Operand angibt.

```
"-" x 80;                                  # String mit 80 Bindestrichen
@keys = qw(Perlen vor die Säue);
@hash{@keys} =  ("") x @keys;              # Hash–Slice
$hash{"Perlen"} = "";
... if ($hash{"vor"} eq "") {...}          # usf.
```

Additive Operatoren: + - .

+ und - sind die üblichen arithmetischen Operatoren. Sind Strings an den Operanden beteiligt, werden sie zuvor in numerische Werte gewandelt.

Der Operator "." verkettet zwei Strings miteinander (ohne Blanks). Nicht-String-Operanden werden zuvor in Strings konvertiert:

```
print +(3+4) . "=Sieben\n";        # liefert: 7=Sieben
print (3+4) . "=Sieben\n";         # liefert: 7=Sieben
(print(3+4)) . "=Sieben\n";        # liefert: 7
```

Bitshift-Operatoren: << und >>

Liefert den Wert des linken Operanden nach Links- bzw. oder Rechtsshift um soviele Bits wie der rechte Operand angibt. Ist auch für negative linke und rechte Operanden gültig! Dies bedeutet letztendlich bei << eine Multiplikation mit bzw. bei >> eine ganzzahlige Division durch $2^{\text{<rechte Seite>}}$.

Das Ergebnis ist **undefiniert für negative rechte Operanden**, denn die Ausführung erfolgt wie in C, und der C-Standard sagt: das Ergebnis ist undefiniert bei negativen rechten Operanden.

```
1 << 4;          # = 16
1 >> 4;          # = 0
13 << -4         # undefiniert
```

Benannte monadische und -Test-Operatoren

Manche "Funktionen" sind in Wirklichkeit monadische Operatoren, mit einem höheren Vorrang als einige der dyadischen (zweistelligen) Operatoren:

-<x> (Dateitests)	gethostbyname	localtime	rmdir
alarm	getnetbyname	log	scalar
caller	getpgrp	lstat	sin
chdir	getprotobyname	my	sleep
chroot	glob	oct	sqrt
cos	gmtime	ord	srand
defined	goto	quotemeta	stat
delete	hex	rand	uc
do	int	readlink	ucfirst
eval	lc	ref	umask
exists	lcfirst	require	undef
exit	length	reset	
exp	local	return	

```
sleep 3|4;    # wie (sleep 3)|4;
sleep (3|4); # wie sleep 7;
print 3|4;    # wie print (3|4); print ist Listenoperator
```

Klammern dienen im Wesentlichen der Klarheit.

Viele monadische Operatoren operieren standardmäßig mit $_ als Operand, wenn kein Argument angegeben ist.

Beachte folgende Operatoren: sie können sowohl als Operator dienen als auch einen Term einleiten:

Zeichen	Funktion des Operators	Funktion als Termeinleiter
+	Addition	Monadisches Plus
-	Subtraktion	Monadisches Minus
*	Multiplikation	*TYPEGLOB
/	Division	/muster/
<	Kleiner als, Linksshift	<HANDLE>, <END
.	Verkettung	.333 (Zahl)
?	?:	?MUSTER?
%	Modulo	%hash
&	&, &&	&unterprogramm

Beispiel:

```
if length < 80 { next };
next if length() < 80;                    # besser
```

Dateitest-Operatoren

Dateitest-Operatoren sind **monadische Operatoren** und verlangen als Argument entweder einen Dateinamen (bzw. $_) oder ein Dateihandle. Sie liefern **1** für "**TRUE**" und "" für "**FALSE**" zurück bzw. Zahlen bei Zugriffs-Datum-Abfragen. Existiert bei letzteren die Datei nicht, ist der Rückgabewert undefiniert.

Beispiele:

```
if ! -e ;                       # Datei in $_ existiert nicht
unlink $file if -w $file;       # Beschreibbare Datei löschen
```

Mit "-T" und "-B" kann man abfragen, ob eine Datei eine Text- oder Binär-Datei ist.

Relationale Operatoren

Vergleichsoperatoren existieren sowohl **numerisch** als auch für **Strings**:

Numerisch	String
>	gt
>=	ge
<	lt
<=	le

Ergebnis: **1** für "TRUE" und "" für "FALSE". Bei Strings werden die ASCII-Werte miteinander verglichen (vom Zeichencode abhängig), auch die Leerzeichen am Ende. Die Operatoren sind **nicht assoziativ**:

```
if ($a <  $b < $c) { ... };       # ist ein Syntaxfehler
if (($a <  $b) < $c) { ... };     # ist möglich (Sematik?)
```

Gleichheitsoperatoren

Numerisch	String	Bedeutung
==	eq	Gleich
!=	ne	Ungleich
<=>	cmp	Vergleich; Liefert -1, 0 oder 1 bei "<", gleich oder ">"

Bitoperatoren & (und), | (oder), ^ (exklusives oder)

Das Verhalten ist bei **Strings anders** als bei **numerischen** Werten, da Strings beliebig lang sein können, numerische Werte aber nicht wegen der Begrenzung durch die Architektur. Unabhängig davon wird die **Operation immer bitweise** durchgeführt, so dass

```
xf0 & 0x0f
```

den Wert 0 hat (Unterschied zwischen "&&" und "&"), aber

```
"123.45" & "234.56"
```

den String "020.44" ergibt.

Sind an der Operation ein String und ein numerischer Wert beteiligt, wird der String **zuerst in eine Zahl verwandelt**, und das Ergebnis ist numerisch. Strings außer "" und "0" sind immer wahr!

Logische Operatoren

Die logischen Operatoren **&&** und **||** stellen logisches "**und**" bzw. "**oder**" dar. Dabei bindet **&& stärker**. Die Abfrage des logischen Ausdrucks bricht dann ab, wenn das Ergebnis wahr ist. Die Anzahl der Abfragen ist minimal.

```
open(FILE,"/a/b") || die "Fehler bei Eröffnen von Datei '/a/b'\n";
```

Statt jener beiden Operatoren können auch "**and**" und "**or**" verwendet werden; sie binden aber nicht so stark.

Einen **xor**-Operator für "exklusives oder" gibt es nur in **Textform**, nicht aber in **Zeichenausführung**.

Bereichsoperator ..

Unterschiedliches Verhalten im Skalar- und Listenkontext. Im **Listenkontext** wird eine Liste zurückgegeben:

```
@l = (1..10);              # @l hat den Wert (1,2,3,4,5,6,7,8,9,10)
@l1 = @l[2..4];            # @l1 hat den Wert (3,4,5)
@l = @l[-5 .. -1];         # Teilfeld: die letzten 5
                           # Elemente von @l
@l2 = ('a'..'z');          # Wie ('a','b',...,'z');

print ('0'..'9','a'..'f')[$digit & 0x0f];
                           # Druckt den Hexadezimalwert der Ziffer $digit
foreach $i (1..10)         # Wie: for ($i=1; $i <= 10; $i++)
```

Der skalare Operator .. dient als **Zeilenbereichsoperator**. Er liefert "FALSE" zurück, wenn die aktuelle Zeilennummer kleiner als der linke Operand ist. Ist sie dagegen nicht kleiner, dann wird "TRUE" zurückgeliefert, solange für die aktuelle Zeile der rechte Operand falsch ist, sonst "FALSE":

```
while (<HANDLE>)
{ print if (10 .. and / ^ $/); }      # Druckt ab Zeile 10 alle Zeilen,die
                                      # keine Leerzeile sind
```

Konditional-Operator:

```
$a ? $b : $c;
```

Der Wert des Ausdrucks ist $b, wenn $a "TRUE" ist, ansonsten $c.

```
print ($x == 0 ? 1 : 2),"\n";
```

druckt 1, wenn $x den Wert 0 hat, ansonsten 2.

Operator-Ketten sind ebenfalls möglich:

```
($x == 0 ? 1 : $x == 1 ? 2 : 3),"\n";
```

Zuweisungsoperatoren

```
=   **=   +=        *=       &=       <<=          &&=
-=    /=  |=        >>=      |     |=
.=    %=  ^=
x=
```

Auf der linken Seite muss ein L-Value stehen, auf der rechten irgendein Ausdruck.

"$lvar <op>= $exp" bedeutet das gleiche wie "$lvar = $lvar <op> exp".

```
($a = $b) += $c;                    # Wie $a = $b;$a = $a + $c;
($neu = $alt) =~ s/foo/bar/g;       # weist zuerst $alt $neu zu
$a = $b = $c = 0;                   # Wie: $a = ($b = ($c = 0));
```

Bei Listen ist nur die **einfache Zuweisung** erlaubt. Im skalaren Kontext wird die Anzahl der Listenwerte zurückgeliefert, im Listenkontext werden die Listen einander zugewiesen:

```
while (($key,$value) = each %hash_table) { ... }
```

Komma-Operatoren

Im Klammern-Kontext wird das linke Argument evaluiert, dann verworfen, dann das rechte Argument evaluiert und als Ergebnis zur Verfügung gestellt.

```
$a = ($b=1,$b-3);       # $a hat dann den Wert -2, aber:
$a = 1,3;               # entspricht: ($a = 1),3; (3 wird weggeworfen)
```

nicht verwechseln mit

```
$a = @vektor;
```

oder mit Funktionsaufrufen.

Statt des Kommas (..,..) kann auch (..=>..) verwendet werden, um einen Zusammenhang zwischen linker und rechter Seite aufzuzeigen.

Was gibt es bei C/C++, aber nicht in Perl?

- Den Adress-Operator & (siehe Referenzen!)
- Monadisches * zur Dereferenzierung
- Casting mit (..)

6. Anweisungen

Perl-Programme bestehen aus Deklarationen und Anweisungen. Deklarationen können dort stehen, wo Anweisungen stehen dürfen, aber:

Deklarationen sind nur zum Compilierzeitpunkt relevant!

Einfache Anweisungen

Einfache Anweisungen sind Ausdrücke, die wegen ihrer Seiteneffekte bewertet werden. Sie müssen **mit ";" abgeschlossen** werden, außer am Blockende (nicht zu empfehlen).

Vor dem ";" einer einfachen Anweisung kann ein **Modifier** stehen, der das Ausführen der Anweisung von Bedingungen abhängig macht:

```
<anweisung> if <ausdruck>;
<anweisung> unless <ausdruck>;
<anweisung> while <ausdruck>
<anweisung> until <ausdruck>;
```

Beispiel:

1. `() unless /You want to speak to me/;`

2. ```
   {
           $line = <STDIN>;
                   ...
   }
   until $line =~ /^EOF/;
   ```
   ```

Zusammengesetzte Anweisungen

Zusammengesetzte Anweisungen bestehen aus Ausdrücken und Blöcken. Dabei kann als Begrenzer für einen Block eine Datei dienen (z.B. beim **require** Statement), oder die Länge eines Strings (z.B. bei **eval**). Im Allgemeinen aber dienen die Block–Klammern-
paare {} als Block-Begrenzer:

```
if (<expr>) <block>
if (<expr>) <block> else <block>
if (<expr>) <block> elsif (<expr>) <block> ...
if (<expr>) <block> elsif (<expr>) <block> ...
else <block>

[<label>]: while (<expr>) <block>
[<label>]: while (<expr>) <block> continue <block>

[<label>]: for (<expr>;<expr>;<expr>) <block>
[<label>]: foreach <var> (<list>) <block>
[<label>]: foreach <var> (<list>) <block>
continue <block>

[<label>]: <block>
[<label>: <block> continue <block>
```

Achtung: anders als in C/C++ sind die **geschweiften Klammern** als Blockbegrenzer **obligat**. Dies macht manches einfacher!

if-Anweisungen

Wegen obligater Klammern ist stets geregelt, zu welchem **if** ein **else** oder **elsif** gehört. Der **else**-Block wird immer dann ausgeführt, wenn der bei **if** bewertete Ausdruck falsch ist.

if (not <expr>) ist identisch mit **unless (<expr>)**

while-Anweisungen

Der Block wird solange ausgeführt, wie <expr> den Wert TRUE hat. Dagegen wird in einer **until**-Anweisung der Block solange ausgeführt, wie <expr> den Wert FALSE hat.

Am Ende der **while**-Anweisung kann ein optionaler **continue**-Block stehen, der prinzipiell nach jedem Schleifendurchlauf durchlaufen wird, auch wenn mit **next** die Schleifensteuerung angestoßen wird.

for-Schleifen

for-Schleifen sind eine spezielle Form von while-Schleifen, z.B.:

```
for ($i = 0; $i < 10; $i++)
{
 ...
}
```

ist gleichbedeutend mit (aber nicht so kompakt):

```
$i = 0;
while ($i < 10)
{
        ...;
}
continue
{
        $i++;
}
```

Allgemein hat eine for-Schleife folgendes Layout:

for (<expr1>;<expr2>;<expr3>) { ... }

Vor Beginn der Schleife wird, falls vorhanden, zunächst <expr1> bewertet.
Die eigentliche Schleife sieht dann so aus:

- Bewerte <expr2>.
- Wenn FALSE, dann Abbruch der Schleife (auch ohne durchlaufen zu werden). Ein leerer <expr2> zählt immer als TRUE.
- Führe Block aus.
- Bewerte <expr3>, falls vorhanden.

Beispiel:

```
for ($i=0, $bit = 1; $mask & $bit; ++$i, $bit <<= 1)
{
        print "Bit $i ist gesetzt\n";
}
```

Eine andere äquivalente Form der **for**-Schleife ist die **foreach-Schleife**:

foreach <var> (<list>) <block>

Die Variable <var> durchläuft dabei alle Elemente der Liste <list> und ist im Block über <var> erreichbar, und dort ein **Synonym** für das jeweilige Listenelement (Zuweisungen an <var> ändern das entspr. Listenelement!);

```
foreach $key (sort (keys %hash))
{       $hash{$key} =~ s/otto/hahn/g;
        $key .= "key";
}
```

oder

```
for $elem (1..9,"\nBumm!")
{
        print STDERR $elem;
        sleep 1 if $elem ne "\nBumm!";
}
```

Merke: **for** und **foreach** bedeuten das selbe.

Schleifen-Steuerung

Schleifen sind über die Anweisungen **last, next** und **redo** steuerbar.

Mit **last** ohne Angabe eines Labels wird die aktuelle Schleife abgebrochen; bei vorhandenem Label-Argument die Schleife, die zum Label gehört.

next ruft die Schleifensteuerung der aktuellen Schleife auf. Ist zusätzlich ein Label angegeben, wird die Schleifensteuerung der entspr. Schleife aufgerufen:

```
LINE:       while (<STDIN>)
            {
                    next LINE if /^#/;
                    next LINE if /^$/;
            }
            continue
            {
                    ++$count;
            }
```

redo ruft ohne Schleifensteuerung erneut den Schleifenrumpf auf:

```
while (<STDIN>)
{
        chomp;
        if (s/\\$//)
        {
                $_ .= <STDIN>;
                redo;
        }
        # Hier Weiterverarbeitung
}
```

Wirkung:

Packt diejenigen Zeilen einer Datei, die mit dem Zeichen "\" enden, zu einer Zeile zusammen, ähnlich einem Makroprozessor.

Ungebundene Blöcke (d.h. solche, die nicht an eine Schleife gebunden sind,) können auch die **last**-Anweisung verwenden. Beispiel: das C/C++-Statement **switch** kann damit emuliert werden.

```
SWITCH:
{
        if (/abc/)      { $abc = 1; last SWITCH; }
        if (/bcd/)      { $bcd = 1; last SWITCH; }
        if (/cde/)      { $cde = 1; last SWITCH; }
        $nothing = 1;
}
```

goto-Anweisungen

goto-Anweisungen besitzen drei Ausprägungen:

- **goto <label>;** — Nicht verwendbar zum Springen in Konstrukte, die eine Initialisierung benötigen wie z.B. Unterprogramme oder for[each]-Schleifen.
- **goto <expr>;** — <expr> muss dabei den Namen eines Labels als Wert besitzen:

  goto ("L1","L2","L3")[$i];

  Entspricht z.B. dem **Computed Goto** von Fortran.
- **goto &<name>;** — Springt in die Unterroutine <name> von der aktuellen Unterroutine her, d.h. die aktuelle Unterroutine wird ersetzt, und dem Aufrufer wird der Aufruf von <name> **vorgetäuscht**. Wird von **AUTOLOAD**-Routinen verwendet.

7. Deklarationen

Deklarationen für Unterprogramme

Deklarationen von **Unterprogrammen** oder **Formaten** sind globale Deklarationen, d.h. unabhängig vom Ort im Package, in dem sie stehen, und sie sind **global sichtbar**. Beim Ablauf des Programmes sind sie ohne Bedeutung, wohl aber bei der **Compilierung**.

Deklarationen von Unterprogrammen sind nicht unbedingt erforderlich: eine **Definition** bewirkt zugleich eine **Deklaration**. Formal gesehen lässt sich ein Unterprogramm-Name benutzen, sobald die Sourcedatei compiliert ist, und zwar wie ein Listenoperator.

```
sub meinname;              # Deklaration, auch Definition irgendwo erforderlich
$ich = meinname(50) or die "Routine meinname nicht gefunden!\n";
```

Im Zweifelsfalle müssen beim Aufruf der Subroutine Klammern um die Argumente gesetzt werden. Mittels der Anweisung **require** lassen sich Unterprogramm-Definitionen oder -Deklarationen aus anderen Dateien laden. **require** wirkt dynamisch und wird erst zur Laufzeit wirksam. Die Verwendung von **use** ist besser geeignet, da diese Anweisung die Routine in den eigenen Namensraum importiert.

Deklarationen mit eingeschränktem Geltungsbereich

Der Wirkungsbereich nicht globaler Deklarationen erstreckt sich von der Stelle der Deklaration bis zum Ende des umgebenden Blocks.

Häufigste Form: Deklaration von **my-Variablen** und **local-Variablen**. Dadurch werden **lokale**(my) bzw. **semilokale**(local) Werte definiert, und zwar innerhalb

- des umschließenden Blocks
- der umschließenden Unterroutine
- der umschließenden Datei
- des umschließenden **eval**'s

```
my ($a, $b, $c) = @_;
my @l = (1,2,3);
```

my-Variable sind streng lokal begrenzt, d.h. bei rekursiven Unterprogramm-Aufrufen wird jedesmal ein neuer Satz von my-Variablen eingerichtet.

local-Variable werden dann gebraucht, wenn man globale Variable in der **momentanen dynamischen Aufrufhierarchie** temporär ändern will (z.B. bei irgendwelchen Spezialvariablen).

Syntaktisch gesehen sind **die Funktionen (!!) my** und **local einfache Modifikatoren** für lvalue-Ausdrücke:

```
my $foo = <STDIN>;          # skalarer Kontext
my ($foo) = <STDIN>;        # Listenkontext
my @foo = <STDIN>;          # Ebenso
```

Beachte:

```
my $x = $x;
```

deklariert die lokale Variable $x und initialisiert sie mit dem Wert der äußeren Variablen $x.

Im lokalen Kontext läßt sich eine globale Variable referieren durch zusätzliche Angabe ihres Package-Namens:

```
$package_name::varname;
```

Pragmas

Pragmas sind spezielle Perl-Moduln; sie dienen dazu, dem Perl-Interpreter zur Compilezeit spezielle Hinweise zu geben. Sie werden mittels der **use-Anweisung** übermittelt:

```
use integer;        # Integer-Arithmetik statt Double
use strict;         # Verlangt sauberes Programmieren
use lib;            # Erweitert Suchpfad @INC zur Compilezeit
use sigtrap;        # Installiert Signalhandler zum Abfangen
                    # von Signalen
use subs;           # Vorwärtsdeklaration von Subroutinen
use vars;           # Prädeklaration globaler Variablen
```

- use integer
  teilt Perl mit, dass bis zum Blockende mit ganzen Zahlen gearbeitet werden soll; lokale Ausnahmen (bei inneren Blöcken) sind dann mit no integer möglich.
- use strict 'vars'
  heißt, dass alle Variablenbezüge bis Blockende entweder auf lokale Variable verweisen, oder Zugriffe müssen vollqualifiziert über den Paketnamen erfolgen.
- use lib '<lib1>'
  erlaubt Perl, zur Compilezeit nach Moduln auch in der Modulbibliothek lib1 zu suchen.

8. Unterroutinen

Perl erlaubt die Definition benutzereigener Unterroutinen, auch Funktionen genannt. Sie können irgendwo im Programm definiert sein, mit **do**, **require** oder **use** aus anderen Dateien geladen oder mit **eval** dynamisch erzeugt werden:

```
sub <name>;                          # Deklaration ohne Prototypen
sub <name>(<protos>);                # Deklaration mit Prototypen
$var = sub { ... };                  # Referenz auf anonyme Funktion
use <package> qw(<name_1> ... <name_n>);
                                     # Import
sub <name> { ... }                   # Definition
<name> (<liste>);                    # Aufruf
&<name> (<liste>);                   # Identischer Aufruf
&$<var> (<liste>);                   # Aufruf über Referenz
&$<var>;                             # Aufruf über Referenz mit @_ als
                                     # Parameterliste
```

Die Parameterübergabe ist einfach:

Alle angegeben Parameter sind hintereinander im Unterprogramm in dem Array @_ sichtbar, zu finden in $_[0], $_[1],....

Perl unterstützt damit in natürlicher Weise eine variable Parameteranzahl.

Änderungen an diesen Listenelementen ändern die Aufrufparameter direkt (call by reference). **Arrays** oder **Hashes in der Parameterliste** sind in der Unterroutine **nicht mehr** als solche **erkennbar**!

Rückgabewert einer Routine ist der **letzte bewertete Ausdruck** in Form einer Liste. Explizite Rücksprünge sind mit **return <Ausdruck>** möglich.

Beispiele

1. Maximum eines alphanumerischen Arrays

```
sub max
{
  my $max = shift;
  foreach $foo (@_)
  {
    $max = $max > $foo ? $max : $foo;
  }
}

$max = max(values %hash, values %hash_1, @list); # Aufruf
```

2. Rückgabewert, abhängig von skalarem oder Listenkontext

```
sub upcase
{
  my @params = @_;
  for (@params)
  {
    tr/a-z/A-Z/;
  }
  return wantarray ? @params : $params[0];
}
($v3, $v4) = upcase($v1, $v2);
```

Die Funktion **wantarray** prüft, ob **upcase** im Listenkontext aufgerufen wurde.

Beachte:

```
(@a,@b) = upcase (@list1, @list2);
```

hinterlässt in **@b** ein **leeres Array**! Alles wird in **@a** abgespeichert.

Wenn die Unterroutine über &<name> ohne Parameterliste aufgerufen wird, wird implizit das aktuelle @_–Array übergeben. Dies ist z.B. bei rekursiven Aufrufen einer Unterroutine angewendbar.

Aber: in diesem Fall entfällt die ***Prototypen-Überprüfung***.

Übergabe von Typeglobs

Es ist möglich, statt eines Arrays nur seinen **Namen** zu **übergeben** (Emulation von Referenzen). Dadurch arbeitet das Unterprogramm nicht auf einer lokalen Kopie, sondern auf der globalen Version:

```
sub double_ary
{
  local (*ARY) = @_;
  foreach $elem (@ARY)        # arbeitet letztendlich
  {                           # auf dem übergebenen
    $elem *= 2;               # Array! Verdoppelt Feldelemente
  }

double_ary(*array_1);         # nicht double_ary(@array_1)!
double_ary(*array_2);
```

Dieser Typeglob-Mechanismus **muss** immer dann verwendet werden, wenn man **in einer Unterroutine** die **Array-Größe verändern** will, z.B. die Operatoren **push** oder **pop** auf ein übergebenes Array anwenden möchte.

Übergabe von Referenzen

Will man mehr als ein Array übergeben oder einen Hash, und zwar mit Erhaltung der Daten-Struktur, so ist man gezwungen, Referenzen als Parameter zu übergeben.

Beispiel: Übergabe von Arrays an Unterroutinen, Arrays poppen, und Rückgabe einer Liste mit allen letzten Elementen:

```
sub popmany                          # Parameterliste besteht aus Referenzen auf Listen
{
    my @ret_list = ();               # leere Liste anlegen
    foreach my $aref (@_)            # Operiere auf Übergabeparametern
    {
      push @ret_list, pop @$aref;    # letztes Element der
                                     # aktuellen Liste
    }
    return @ret_list;                # neue Liste zurückgeben
}
@lasts = popmany(\@a,\@b,\@c,\@d);
```

Prototypen

Unterroutinen sind so deklarierbar, dass die Parameterübergabe kontrolliert werden kann, z.B. hinsichtlich Parameteranzahl oder -Typen. Dies wirkt aber nur auf **Funktionsaufrufe ohne das &-Zeichen**. Auf **Klassenmethoden** ist dies **nicht anwendbar**.

Im Prinzip verhält sich dann eine solche Unterroutine wie eine in Perl einge-baute Funktion, d.h. solchermaßen deklarierte Funktionen verhalten sich in etwa wie die entsprechenden Bibliotheks-Routinen von Perl.

UP deklariert als	Beispielhafter Aufruf
sub mylink($$)	mylink $old,$new
sub myvec($$$)	myvec $var,$offset,1
sub myindex($$;$)	myindex &getstring,"substr"
sub mysyswrite($$$;$)	mysyswrite $buf,0,length($buf)-$off,$off
sub myreverse(@)	myreverse $a,$b,$c
sub myjoin($@)	myjoin ":" $a,$b,$c
sub mypop(\@)	mypop @array
sub mysplice(\@$$@)	mysplice @array,@array,0,@pushme
sub mykeys(\%)	mykeys %{hashref}
sub mypopen(*;$)	mypopen HANDLE,$name
sub mypipe(**)	mypipe READHANDLE,WRITEHANDLE
sub mygrep(&@)	mygrep {/foo/} $a,$b,$c

Prototypzeichen	Wirkung
\	Parameter muss als Referenz mit dem nachfolgenden Zeichen beginnen (mykeys z.B. beginnt mit %)
@	Erzwingt Listenkontext
%	Erzwingt Listenkontext
$	Erzwingt Skalarkontext
&	Verlangt anonyme Unterroutine als Parameter; das Komma danach kann entfallen, wenn erster Parameter
*	Verlangt Typeglob
;	Trennt optionale von obligaten Argumenten

9. Referenzen

Im allgemeinen ist **Perl** auf einfache, **lineare Datenstrukturen** ausgerichtet. Manchmal sind aber auch komplexere, z.B. hierarchische Datenstrukturen erforderlich. Wie würde man dabei ohne Referenzen vorgehen?

Beispiel: Es wird eine Tabelle benötigt, die Attribute von Personen enthält (Alter, Augenfarbe, Gewicht). Dazu würde man beispielsweise ohne Refe-renzen für jede Person ein Array anlegen:

```
@john = (47,"Braun",93);
@mary = (23,"Himmelblau",64);
@billy = (35,"Grün",78);
```

und ein Array, welches die Namen der obigen Arrays verwaltet:

```
@people = ("john","mary","billy");
```

Um z.B. die Augenfarbe von john auf "Blau" zu setzen, ist folgender PERL-Code erforderlich:

```
$name = $people[0];
eval "\$$name[1] = 'Blau'";       # erzeugt PERL-Code
${$name}[1] = 'Blau';             # Alternative; bedeutet:
$john[1] = 'Blau';
```

Man könnte auch mittels eines Hashs Zeiger emulieren und dann in den Arrays den Namen suchen lassen.

Beides sind umständliche Methoden, um die Augenfarbe neu zu setzen.

Referenzen lassen eine viel einfachere Lösung des Problems zu. Im obigen Beispiel hat **$people[0]** den Wert 'john'. Dies lässt sich aus so ausdrücken: Die **Variable $people[0]** enthält einen String, dessen Wert der **Name einer anderen Variablen** ist. Man sagt auch: Es liegt eine **symbolische Referenz** vor.

Ein **weiterer Typ** von Referenzen sind **feste Referenzen**. Eine solche ver-weist nicht auf den Namen, sondern direkt auf die Variable. Der Verweis geschieht über Symbol-Tabellen-Einträge.

Kurz gesagt:

Bei symbolischen Referenzen geht es um einen Ort, der einen bestimmten Namen hat, bei festen Referenzen dagegen direkt um das referenzierte "Ding".

Eine feste Referenz wird durch eine skalare Variable dargestellt, und sie kann auf **beliebige Datenstruktur zeigen.** Auf diese Art lassen sich einfach mehrdimensionale Strukturen erzeugen: Arrays von Arrays, von Hashes usf.

Feste Referenzen erzeugen

Der **monadische Backslash-Operator** "\" erzeugt eine Referenz auf eine benannte Variable oder Unterroutine:

```
$skalarref = \$foo;
$konstref = \12345.77;
$arraydref = \@ARGV;
$hashref = \%ENV;
$coderef = \&handler;
$globref = \*STDOUT;
```

Auf **anonyme Arrays** lassen sich mit Hilfe von (mehrfachen) "[]"-Paaren ebenfalls feste Referenzen erzeugen:

```
$anon_l_ref = [1,2,["3","4","5"],6];   # anonym
@reflist = (\$a,\$b,\$c);              # oder die Kurzform
@reflist = \($a,$b,$c);                # Keine Referenz auf Liste
```

Auf ähnliche Weise lassen sich **Referenzen auf anonyme Hashes** mit Hilfe von "{}"-Paaren erzeugen:

```
$anonhref = {"a"=>"AAAAAA","b"=>"BBBBBB"};
```

Achtung, wenn eine Funktion einen anonymen Hash anlegt und die Referenz darauf zurückgeben soll:

```
sub hashmem                     # Falsch
{
        { @_  };                # Hier wird nur ein Block geöffnet,
}                               # aber keine Referenz erzeugt (keine
                                # Dynamik)

sub hashmem                     # Richtig
{
         return { @_  };        # Referenz erzeugt und
                                # zurückgegeben
                                # Der Unterschied ist, dass {...} in
                                # einem
                                # return-statement als Expression
                                # auftritt
}
```

Referenzen auf anonyme Unterroutinen erzeugt man durch Verwendung von **sub** ohne Routinen-Namen:

```
$subref = sub { print "porky!\n"; };
```

Beachte, dass der Code nicht ausgeführt wird; es wird lediglich eine **neue Referenz** darauf generiert, sogar bei mehreren Referenzen.

Unterroutinen können feste **Referenzen zurückgeben**.

Beispiele sind die Konstruktoren von Klassen, die Referenzen auf Objekte zurückgeben. Dies geschieht dadurch, dass sie mittels der Funktion **bless** einen anonymen Hash in ein Objekt umwandeln und eine Referenz auf diesen Hash (auf diese Objekt!) zurückgeben:

```
$objref = Huendchen->new(Schwanz=>"kurz",Ohren=>"lang");
```

Dateihandles können an Unterprogramme **nur über Referenzen auf das Handle** übergeben werden, bzw. auf den entsprechenden Typeglob:

```
sub Hundefutter
{
        my $fh = $_[0];
        print $fh "mmmmmmmhhhhhh\n";
}

Hundefutter(\*STDOUT);          # Referenz auf STDOUT als 1. Parameter
```

Verwendung von festen Referenzen

```
$foo = "Zwei Hoecker";
$fooref = \$foo;
$Kamel_modell = $$fooref;       # Kamel_modell ist nun "Zwei Hoecker"
```

Unterschiedliche Arten der Dereferenzierung:

```
$bar = $$scalref;
push (@$arrayref,$filename);
$$arrayref[0] = "Januar";
%$hashref = %hash_0;
$$hashref{"Schluessel"} = "Value";
&$suberef(1,2,3);
print $globref "Hallo\n";

$refrefref = \\\"Hallo";
print $$$$refref,"\n";
```

Die Dereferenzierung erfolgt höherprior als Array- oder Hash-Zugriffe! Das **erste Zeichen** steuert die Art der Dereferenzierung.

Bei der indirekten Dereferenzierung wird **eine Referenz als Wert eines Blocks** statt einer Variablen zurückgegeben: an jeder Stelle, wo ein alphanumerischer Identifier erlaubt ist, kann der Identifier durch einen Block ersetzt werden, der eine Referenz auf den korrekten Typ zurückgibt.

```
$bar = ${my $dummy = 10;\$dummy};      # Weist $dummy $bar zu
push @{$arrayref},$filename;           # $filename im Array
                                       # @{$arrayref} ablegen

${$arrayref}[0] = "Januar";
${$hashref}{"Schluessel"} = "Value";
&{$suberef}(1,2,3);                    # Aufruf einer Subroutine
```

Beispiel: das Hash-Element **$dispatch{$index}** enthalte eine Referenz auf eine Unterroutine. Dann ruft

```
&{$dispatch{$index}} (1,2,3);
```

diese Unterroutine mit der Parameterliste **(1,2,3)** auf.

Der Arrow-Operator

Der Arrow-Operator -> bietet eine einfache Dereferenzierungs-Schreibweise bei Referenzen auf Arrays oder Hashs (siehe auch C/C++). Neben

```
${$arrayref}[0] = "Januar";
$$hashref{"key"} = "value";
$${$array[3]}{"English"}[0] = "january";
```

ist auch

```
$arrayref->[0] = "Januar";
$hashref->{"key"} = "value";          # oder
$array[3]->{"English"}->[0] = "january";
```

korrekt. Dabei sei @array ein Array von Hash-Referenzen, bei denen der zu einem Key gehörige Value eine Referenz auf ein Array darstellt. Die Zuweisung erzeugt sogar Array– und Hash–Referenzen automatisch, d.h. z.B. das Array @array und Hash werden im Extremfall sogar neu angelegt oder erweitert.

Ref-Operatoren

Mittels des **ref-Operators** ist es möglich, sich über den Typ des Datums zu informieren, auf das die Referenz zeigt. Der **ref**-Operator gibt den Typ zurück; integriert sind

- **REF**
- **SCALAR**
- **ARRAY**
- **HASH**
- **CODE**
- **GLOB**

Verwendet man eine feste Referenz in einem String-Kontext, wird sie in einen String umgewandelt, und zwar in der Form

```
"SCALAR(0x1fc00e)"
```

Bei einer Objekt-Referenz (mit **bless** auf einen anonymen Hash erzeugt) erhält man z.B.:

```
"MyType=HASH(0x20d10)"
```

Unterroutinen in Strings interpolieren

```
sub mysub { return (0,1,2,3); }# Liefere eine Liste zurück
print "Unterroutine mysub liefert das gesamte Array (@{[mysub( )]}) zurueck.\n";
```

Was passiert?

{ ... } im zweiten Statement stellt sich beim Compilieren als Block heraus, dessen Wert eine Referenz auf ein anonymes Feld ist, im dem eine Kopie des Arrays abgelegt ist, das der Funktionsaufruf **mysub()** zurückliefert. Durch "**@{...}**" wird im Array-Kontext eine **Interpolation** dieses anonymen Arrays innerhalb des print-String **erzwungen** zwischen den runden Klammern und damit folgendes ausgedruckt:

```
Unterroutine mysub liefert das gesamte Array (0 1 2 3) zurück
```

Die Blanks rühren von der Interpolation des Feldes innerhalb des Textstringes her (Feldelemente werden bei der Expansion eines Feldes innerhalb Strings mit Blanks expandiert!).

10. Packages

Ziel:

Einmal geschriebene PERL-Routinen, die auch anderswo einsetzbar sind, der Allgemeinheit zur Verfügung stellen, ähnlich wie in dies in C beispielsweise mit Include-Files bzw. binären Libraries gehandhabt wird.

Vermeide Kopieren der Sourcen, oder Copy/Paste! Denn dann treten beinahe zwangsläufig Namenskonflikte auf. Eine Trennung von Namensräumen erfolgt mithilfe von Packages. Jedes Package verwaltet einen eigenen Namensraum und ist nicht an Dateien gebunden. Ein Spezialfall von Packages sind die PERL-Moduln: In diesem Fall erstreckt sich ein Package genau über eine Datei, deren Name mit dem Suffix .pm endet.

Moduln werden mit der Anweisung **use** dem Perl-Programm bekannt gegeben. Mittels Moduln lassen sich auch **Objektklassen** implementieren, die ebenfalls eine Wiederverwendbarkeit ermöglichen.

Zwischen verschiedenen Packages wechselt man mit der **package**–Dekla-ration hin und her. Zu Beginn ist das **main-Package** das aktuelle Package.

Die package-Anweisung legt fest, welche **Symboltabelle** verwendet wird (Lookup zur Compile-Zeit, z.B. bei **eval**-Anweisungen).

Geltungsbereich der package-Deklaration: von ihrem Auftreten bis zum innersten umschließenden Block oder bis zur nächsten package-Deklaration auf gleicher Ebene.

Wirkung:

alle **Identifier** werden in die dazugehörige **Symboltabelle eingetragen**.

```
file.pm:  XXXXX;
          ...
PERL-Script:    require "file";          # oder
                use "file" ();
```

Die Verwendung von Variablen aus verschiedenen Packages ist möglich über **<packagename>::<variable>**. Voreinstellung für <packagename> ist main, d.h. in normalen PERL-Scripts sind die folgenden Angaben identisch:

```
<variable>              # Verkürzt
$::<variable>           # Verkürzt
$main::<variable>
```

Sogar **Schachtelungen** sind möglich:

```
$AUSSEN::INNEN::var
```

In der "Symboltabelle" des aktuellen Paketes werden nur Identifier abgelegt, die mit Buchstaben oder Underline beginnen. Alle anderen (z.B. solche, die mit $ beginnen) werden im main-Package abgelegt.

Vorsicht bei Packages mit den Namen **m**, **s**, **y** oder **tr** (Patter-Match-Operatoren): Verwendung in qualifizierter Form kann zu Fehlinterpretationen führen (wegen Pattern-Match, z.B. **use s::p1::p2::p3;** Fehlinterpreation als Musteränderung mit ":" als Musterbegrenzer).

Die Symboltabellen sind auch vom PERL-Script aus zugreifbar: sie sind in einem Hash mit dem Namen des Packages abgelegt:

%<packagename>::

Die Keys sind die Identifier, und die Werte die Typeglob-Werte. Das heißt: verwendet man die Typeglob-Notation ***name**, dann greift man letztendlich auf das entsprechende Hash-Feld zu:

```
local *XXX = *STDOUT;
print $main::{"XXX"};              # Druckt: *main::STDOUT
```

Möglich ist auch:

```
$key (keys %main::)
{
        local *sym = $main::{$key};
        print "\$$key ist definiert\n" if defined $sym;
        print "\@$key ist definiert\n" if defined @sym;
        print "\%$key ist definiert\n" if defined %sym;
}
```

BEGIN- und **END-**Blöcke

Diese speziellen Unterroutinen für Packages haben ähnliche Funktionen wie Konstruktoren bzw. Destruktoren in C++. In vergleichbarer Funktion sind sie auch von dem Tool **awk** her bekannt.

Der **BEGIN-Block** wird so früh wie möglich ausgeführt, sobald er komplett definiert ist, d.h. bereits bei der Kompilation. Mehrere **BEGIN**-Blöcke werden zusammengefasst und dann bearbeitet.

Im **BEGIN**-Block können Definitionen und Unterroutinen aus anderen Dateien eingebunden werden.

Der **END-Block** wird ausgeführt, wenn der Interpreter beendet wird, d.h. noch später als die **die-Funktion**. Bei mehreren **END**-Blöcken werden alle ausge-führt, aber in umgekehrter Reihenfolge.

```
die "Grün\n";
END { print "Blau\n"; }
BEGIN { print "Rot\n"; }
```

gibt die Texte "Rot", "Grün" und "Blau" aus (in dieser Reihenfolge).

Achtung: **BEGIN- und END-Blöcke werden immer** ausgeführt, auch bei Verwendung des Schalters **-c** (nur Syntaxprüfungen)!

Autoloading

Autoloading erlaubt das automatische Nachladen von Routinen, die nicht definiert sind: Enthält ein Package eine Unterroutine namens **AUTOLOAD**, und wird in dem Package eine nicht definierte Routine aufgerufen, dann wird stattdessen die Package-Routine **AUTOLOAD** aufgerufen, und zwar mit der gleichen Parameterliste wie die ursprüngliche aufgerufene Routine; der **Name der Unterroutine wird als Hash-String** in der Variablen **$AUTOLOAD** abgelegt. Die Funktionalität von **AUTOLOAD** ist völlig beliebig: es kann die ursprüngliche Routine definiert und aufgerufen werden oder nur eine Emulation der Funktionalität oder irgend etwas anderes.

```
sub AUTOLOAD
{
    my $program = $AUTOLOAD;    # z.B. "main::<routine>"
    $program =~ s/.*::/;        # Extrahiere Routinen-Namen
    system($program,@_);
}

date();                         # Die Shell-Kommandos date,
who("am","i");                  # who am i und
ls "-l";                        # ls -l werden aufgerufen
```

Module

Module sind wiederverwendbare Packages, die in **einer** Bibliotheksdatei abgelegt sind. Der Package-Name ist der Name der Datei, deren Namens-Endung ".pm" ist. Symbole können exportiert werden (nicht so gut, Kollisionsgefahr!), oder ein Package kann wie eine Klasse (in C++) gehandhabt werden, indem durch Methodenaufrufe Operationen und Objekte zugänglich gemacht werden ohne Symbol-Export.

Module werden eingebunden per

```
use <module>;                          # oder
use <module> (<symbol> ...);
```

Dadurch wird der Modul <module> beim Compilieren eingebunden, und die in der Liste <symbol>... angegebenen Symbole werden, sofern angegeben, importiert (≙ zur Verfügung gestellt). Die beiden obigen Anweisungen sind identisch mit:

```
BEGIN
{
        require "module.pm";
        module->import();              # Version 1
        module::import();              # oder Version 2
        module->import(@LISTE);        # Import mit Symbolliste
}
```

Ohne Angabe einer Import-Liste werden alle vom Modul <module> exportierten Symbole importiert, außer bei

```
use module();
```

Der Unterschied zwischen **require** und **use** liegt darin, dass bei use automatisch die **exportierten Symbole des Moduls importiert** werden, bei require aber nicht:

```
require "Cwd";                  # Cwd:: durch Qualifikation zugänglich
$here = Cwd::getcwd();          # hier muss man für getcwd
                                # qualifizieren

use CWD;                        # Namen aus Cwd:: sind importiert.
$here = getcwd();               # Keine Qualifizierung für getcwd
                                # erforderlich
```

Geschachtelte Moduln wie **Module1::Module2** werden in der Bibliotheksdatei "***<modulpath>*/Module1/Module2.pm**" gesucht.

Achtung:

Moduln sollten nicht ohne Bedacht Namen wie z.B. **open** oder **chdir** standardmäßig exportieren, weil damit die Semantik des Programmes möglicherweise geändert wird durch **Überschreiben eingebauter Funktionen**.

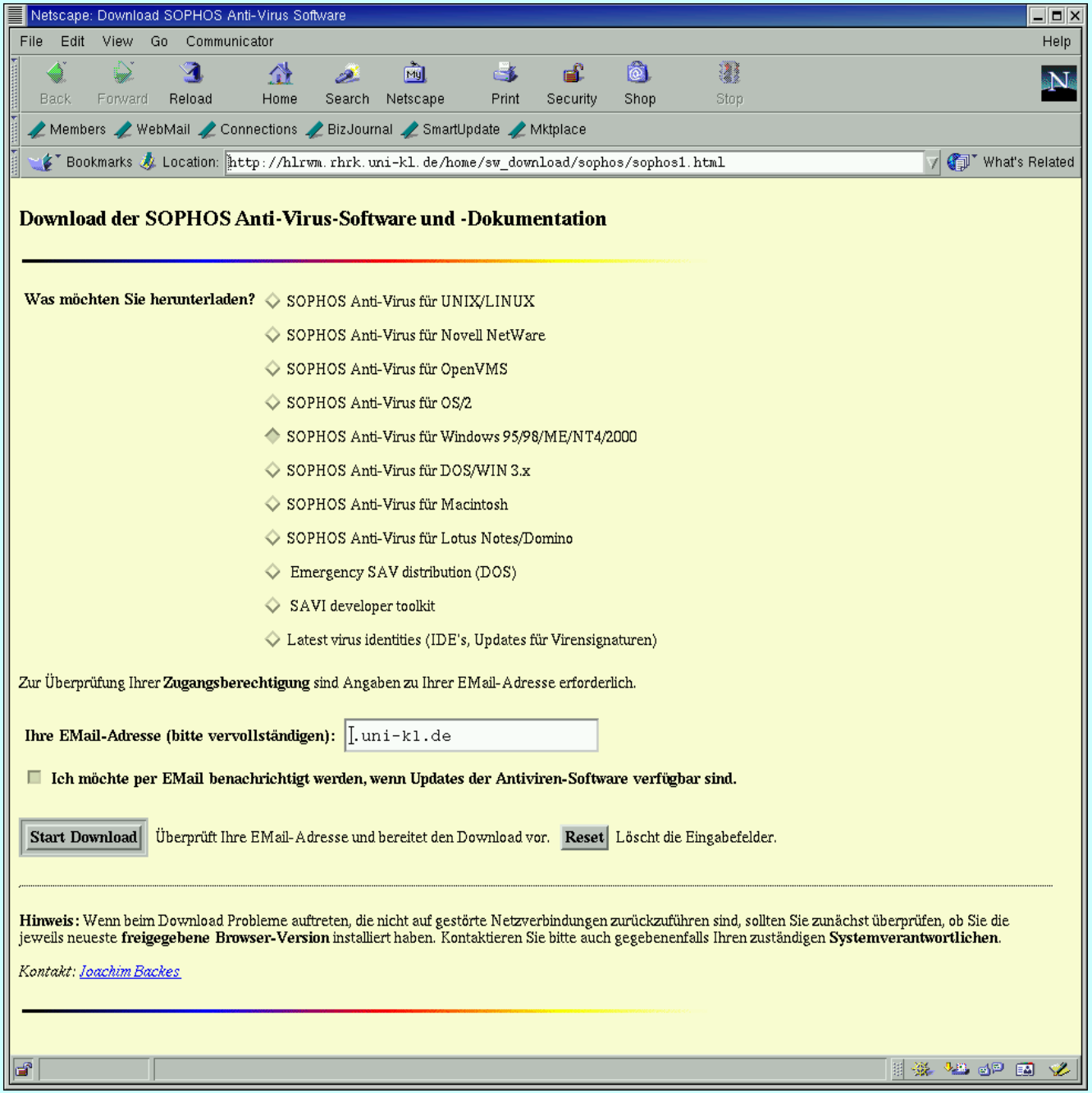
Solche Dinge sollte man nur dann durchführen, wenn man **nicht vorhandene Systemleistungen emulieren** möchte. Die Originalversionen sind stets über **CORE::*routine*()** erreichbar.

11. CGI-Skripte

Beispiel für eine Webseite: Antiviren-Software downloaden.

Parameter:

- Typ des Betriebssystems, für das die Software heruntergeladen werden soll
- E-mail-Adresse als "Berechtigungsausweis"
- Schalter, ob über Updates informiert werden soll
- Durch Dücken des "**Start Download**"-**Buttons** wird ein **Auftrag des Browsers** an den **WEB-Server** geschickt, der wiederum einen Prozess startet zur Ausführung des dazugehörigen sogenannten **CGI-Skriptes (CGI:** *C*ommon *Ga*teway *I*nterface**).**


Netscape: Download SOPHOS Anti-Virus Software
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Members WebMail Connections BizJournal SmartUpdate Mktplace
Bookmarks Location: http://hlrwm.rhrk.uni-kl.de/home/sw_download/sophos/sophos1.html What's Related
Download der SOPHOS Anti-Virus-Software und -Dokumentation
Was möchten Sie herunterladen?
SOPHOS Anti-Virus für UNIX/LINUX
SOPHOS Anti-Virus für Novell NetWare
SOPHOS Anti-Virus für OpenVMS
SOPHOS Anti-Virus für OS/2
SOPHOS Anti-Virus für Windows 95/98/ME/NT4/2000
SOPHOS Anti-Virus für DOS/WIN 3.x
SOPHOS Anti-Virus für Macintosh
SOPHOS Anti-Virus für Lotus Notes/Domino
Emergency SAV distribution (DOS)
SAVI developer toolkit
Latest virus identities (IDE's, Updates für Virensignaturen)
Zur Überprüfung Ihrer Zugangsberechtigung sind Angaben zu Ihrer EMail-Adresse erforderlich.
Ihre EMail-Adresse (bitte vervollständigen): .uni-kl.de
Ich möchte per EMail benachrichtigt werden, wenn Updates der Antiviren-Software verfügbar sind.
Start Download Überprüft Ihre EMail-Adresse und bereitet den Download vor. Reset Löscht die Eingabefelder.
Hinweis: Wenn beim Download Probleme auftreten, die nicht auf gestörte Netzverbindungen zurückzuführen sind, sollten Sie zunächst überprüfen, ob Sie die jeweils neueste freigegebene Browser-Version installiert haben. Kontaktieren Sie bitte auch gegebenenfalls Ihren zuständigen Systemverantwortlichen.
Kontakt: Joachim Backes

Beispielhafter HTML-Code (ohne Anspruch auf Vollständigkeit!)):

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD><TITLE> Download SOPHOS Anti-Virus Software </TITLE></HEAD>
...
<form method=post action="/cgi-bin/sophos.pl">
<table>
<tr><td><strong>Was m&ouml;chten Sie herunterladen?</strong></td>
<td><input type=radio name="sys" value="unix">  SOPHOS Anti-Virus f&uuml;r UNIX/LINUX</td> </tr>
<tr height=20><td></td>
<td><INPUT TYPE=radio name="sys"  value="netware">  SOPHOS Anti-Virus f&uuml;r Novell NetWare</td> </tr>
<tr>
<td></td><td>
<input type=radio name="sys" option value="openvms">  SOPHOS Anti-Virus f&uuml;r  OpenVMS
</td></tr>
...
<tr><td></td>
<td><input type=radio name="sys" option value="IDE">  Latest virus identities (IDE's, Updates f&uuml;r Virensignaturen)</td> </tr>
</table>
<br>
Zur &Uuml;berpr&uuml;fung Ihrer <strong>Zugangsberechtigung</strong> sind Angaben zu Ihrer EMail-Adresse erforderlich.
<br><br>
<table>
<tr><td align=left><strong>Ihre EMail-Adresse (bitte vervollst&auml;ndigen):  </strong><INPUT type=text name=mail size=24 value=".uni-
kl.de"></td></tr>
<tr><td align=left><INPUT type=checkbox name=inf value="+" checked>
<strong>  Ich m&ouml;chte per EMail benachrichtigt werden, wenn Updates der Antiviren-Software verf&uuml;gbar sind.</strong></td></tr>
</table><br>
<strong><INPUT TYPE=submit VALUE="Start Download"></strong>  &Uuml;berpr&uuml;ft Ihre EMail-Adresse und bereitet den Download
vor.  
<strong><INPUT TYPE=reset VALUE="Reset"></strong>  L&ouml;scht die Eingabefelder.
</FORM>
...
</BODY>
</HTML>
```

Werde nun z.B. in das E-Mail-Feld **mayr@eit.uni-kl.de** eingegeben und als System win32 ausgewählt:

Aufgabe des CGI-Scripts ist es nun, aus diesem Eingabestring die einzelnen Parameter und deren Werte zu extrahieren und ein entsprechendes Fenster zu öffnen, in dem der Download der Sophos-Software für win32-Betriebssysteme angeboten wird (z.B. über ftp oder http). Wird der Button über die Benachrichtigung aktiviert, sendet der Browser aufgrund der hier angewendeten POST-Methode entsprechende Anweisungen an den WEB-Server, und dieser wiederum startet den CGI–Prozess und übergibt ihm über **STDIN** folgenden Text:

sys=win32&mail=mayr%40eit.uni-kl.de&inf=%2B

(Das Zeichen "&" dient als Trennzeichen der Aufrufparameter).

Aufgabe des CGI-Scripts ist es nun, aus diesem Eingabestring die einzelnen Parameter und deren Werte zu extrahieren, und ein entsprechendes Fenster zu öffnen, in dem der Download der Sophos-Software für win32-Betriebssysteme angeboten wird (über ftp oder http).

```
#!/usr/bin/perl
#-----------------------------------------------------------------
#
# Example CGI script for download of Sophos Antivirus Software
#
#-----------------------------------------------------------------
sub err
{
        $err_text = shift;
        $title = "Fehler beim Download der SOPHOS-Antivirus-Software.";
        print <<EOF;
        Content-type: text/html  <HTML>
        <HEAD><TITLE>$title</TITLE></HEAD>
        <BODY bgcolor="#FFFFDA">
        <H1>$title</H1>
        <tt>$err_text</tt>
        <BR></BODY></HTML>
EOF
}
require "url_encoding.pm";
use Fcntl qw (:DEFAULT :flock);
use Mail::CheckUser qw(check_email);
$admin = "\nBitte wenden Sie sich an: Joachim Backes <joachim.backes\@rhrk.uni-kl.de>";
read(STDIN,$buf,$ENV{"CONTENT_LENGTH"}); # read URL encoded string into buf
%FIELDS = URLEncoding::parse_cgi($buf);
$sys = $FIELDS{"sys"};
$inf = "-";
$inf = $FIELDS{"inf"} if exists $FIELDS{"inf"};
$mail = $FIELDS{"mail"};
$mail = "??????" if $mail eq "";
$mail =~ tr/A-Z/a-z/;if ($mail =~ /^[a-z]{1,1}[_a-z0-9\.\-]*\@[a-z]{1,1}[a-z0-9]*\.uni\-kl\.de$/)
{
 if (!Mail::CheckUser::check_email($mail))
  {
   err "Unerlaubter Zugriff!";
   exit;
  }

  $db = "/usr/local/download_databases/download_db_sophos/database";
  if (!open(FH, ">>$db"))
```

```
  {
   err "Interner Fehler: die SOPHOS-Datenbank-Datei kann nicht geöffnet werden.{$admin}.";
   exit;
  }
  if (!flock(FH,LOCK_EX))
  {
    err "Interner Fehler: die SOPHOS-Datenbank-Datei kann nicht gesperrt werden.${admin}.";
    exit;
  }
  printf FH "%s %s SOPHOS $sys\n",scalar(localtime), $inf . $mail,$vers;
  flock(FH,LOCK_UN);
  close(FH);
  print "Location: ftp://dl-rechner.rhrk.uni-kl.de/pub/SOPHOS-Antivirus/$sys/Actual\n\n";
  exit;
}
err " Unerlaubter Zugriff!";
```

12. Objektorientierte Programmierung

- Ein Problem wird nach einem Objektmodell modelliert mit ständiger Verfeinerung (objektorientierte Analyse), d.h.: Teile der realen Welt werden auf ein Modell abgebildet, in dem Objekte untereinander agieren und kommunizieren.
- Die Kommunikation der Objekte erfolgt über Nachrichten.
- Objekte haben Gemeinsamkeiten – sie werden aus Schablonen (Templates) erzeugt.
- Jedes Objekt hat eine eigene Identität.

Beispiel:

Eine **Fensterhierarchie** aus grafischen oder pseudografischen Bild-schirmen: Fenster haben **grundlegende gemeinsame Eigenschaften (z.B. rechteckig)**, aber auch **unterscheidende Merkmale** (Farben, Buttons, Scrollbars, Menüs usf.).

Nachrichten sind z.B. Aufforderungen des Windowmanagers (WM) an ein Fenster, seine **Größe zu ändern** sowie Rückmeldungen des Fensters an den WM. Das Fenster verwaltet selbst seine Position und Geometrie-Daten, diese sind also ebenfalls **privater Natur**.

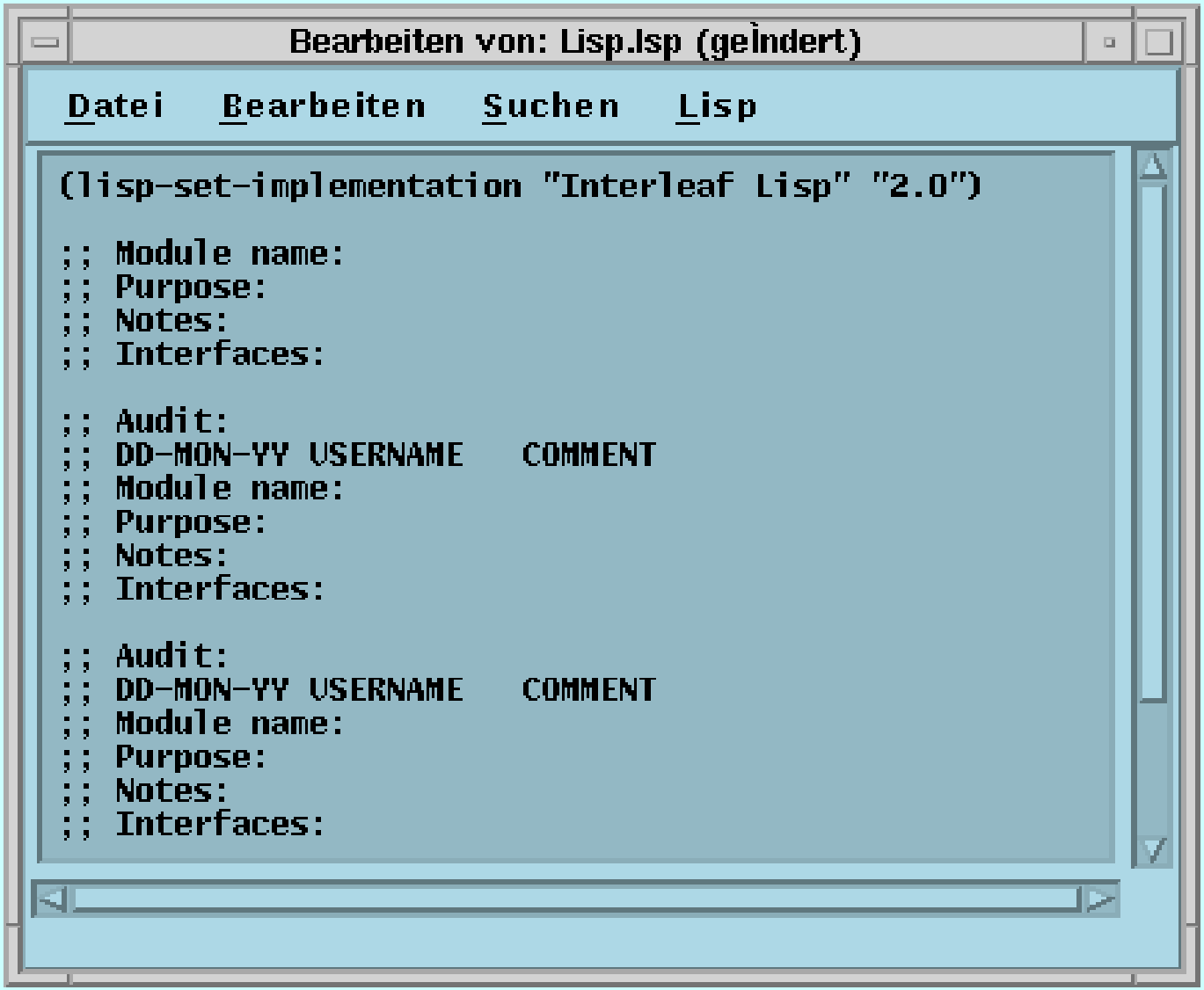
Klassen und Objekte

- Objekte haben öffentliche Interface – **Methoden**
- Objekte haben einen Satz privater Attribute, sind also individuell.
- Private Daten kann **nur das Objekt selbst ändern**, verursacht z.B. durch ein anderes Objekt, welches Methoden des ersteren aufruft.
- Gleichartige Objekte werden aus einer **Schablone** erzeugt, der **Klasse**.
- Erzeugungsvorgang: **Instantiierung**.
- Erzeuger: **Konstruktor-Funktion** (z.B. Speicherplatzreservierung, Dateninitialisierung).
- Verwendung beendet: Löschen des Objektes durch **Destruktor** oder einfach Freigabe des Speicherplatzes.
- **Klassenfunktionen**: objektunabhängig.

Abgeleitete Klassen

- **Vererbung**: wichtigste Klassenbeziehung
- Codeeinsparung bei Gemeinsamkeiten zwischen Klassen, **Vererbung von Klasseneigenschaften** an **abgeleitete Klassen** durch **Spezialisierung**
- In abgeleiteten Klassen werden nur neue Erscheinungsformen ausformuliert und programmiert.
- Beziehung: **Basisklasse ⇔ Abgeleitete Klasse**
- Beispiel: Fensterklasse ⇔ Buttonklasse ⇔ PushButtonklasse
- Objektbeziehung: Objekte können zur Laufzeit andere Objekte enthalten

Beispiel: Reales Fenster enthält Textfenster und Scrollbars (für Fensterhierarchie)



Objektorientiertes Programmieren in PERL

Das objektorientierte Programmieren wurde in PERL Version 5 eingeführt. In PERL werden **Klassen über Packages in Moduln** definiert. Sie sind z.B. folgendermaßen zu laden:

```
use ST::SomeClass;
```

Objekte werden instantiiert durch Aufruf des Konstruktors (der aber anders als in C++ einen beliebigen Namen haben kann) – nicht unbedingt **new**).

Als Beispiel das TK-Toolkit:

```
$button1 = Tk::Button(-test => 'Click me', -action =>\&callme);
$button2 = Tk::Button("-test",'Click me',"-action",\&callme);
```

Der Konstruktor (hier **Tk::Button**) liefert eine **Referenz auf ein neues Objekt** zurück (hier in **$button1/2** abgespeichert).

Weiter verbreitet ist auch folgende Schreibweise:

```
$obj = ST::SomeClass->new(); # Pfeil hat nichts mit Referenz zu tun
```

oder

```
$obj = ST::SomeClass->new("key","22222"); #Oder, äquivalent
$obj = ST::SomeClass->new("key" => "22222");
```

oder

```
$obj = new ST::SomeClass;  # Hier ist der Konstruktor anders
                           # implementiert
$obj = new(ST::SomeClass);       # Geht auch
```

Von **einer Klasse** sind **mehrere Objekte instantiierbar**.

Beachte: Bei der ersten Schreibweise **ST::SomeClass->new()** handelt es sich um eine **Sonderregelung (ein sog. goody)**, da trotz Pfeilnotation **ST::SomeClass keine Referenz** darstellt.

Aufruf von Methoden:

```
$obj->set_white();
$old_stipple = $obj->get_stipple();
$obj->set_stipple("tile");
```

Objekte sind als Referenzen sind **in Arrays abspeicherbar**.

Destruktoren:

Werden aufgerufen, wenn sie vorhanden sind und das Objekt obsolet wird.

Beispiel für eine Package mit Klassendefinition:

```
package MyClass;

sub m1 { ... }                  # Methode 1
sub m2 { ... }                  # Methode 2
sub new { ... }                 # Konstruktor, kann anders lauten
sub cf { ... }                  # Klassenfunktion
$cv = "aaaaa bbbbb";            # öffentliche Klassenvariable
my $counter = 0;                # Private Klassenvariable
sub DESTROY { ... }             # Destruktor
1;                              # für use
```

Die Klassendefinition wird im Anwendercode verfügbar durch

```
use MyClass;                    # Muss im @INC-Pfad stehen
```

Ein Zugriff auf die **Klassenvariable** $counter ist **nicht möglich, da lokal!**

```
$obj1 = MyClass->new(10);
$obj2 = MyClass->new($obj1);          # Fast ein Copy-Konstruktor

$obj1->m2(1,'aaa');                   # Methodenaufruf von Methode m2
MyClass::cf();                        # Aufruf einer Klassenfunktion

@l = split /\s+/,$MyClass::cv;        # Zugriff auf Klassenvariable
++$MyClass::counter;                  # geht nicht, da private
                                      # Klassenvariable
```

Wie werden Objekte innerhalb des Packages erzeugt?

Einfachster Konstruktor:

```
sub new ()
{
   $classname = shift();       # String, Name der Klasse, 1. impliziter Parameter
   bless({},$classname);       # 1. Parameter von bless: Zeiger auf
                               # anonymen Hash erstellen
}
```

Aufruf:

```
$obj = ClassNam->new();
```

Der Aufruf des Konstruktors in der Form **ClassNam->new()** oder auch **new ClassNam** übergibt den Klassen(Package-)namen an die Subroutine new() (und **an alle Subroutinen** der Klasse) als verborgenen 1. Aufrufparameter (dies liegt an der Ausprägung der Klasse als Package). Der Ausdruck "**{}**" hat als Wert einen **Zeiger auf einen anonymen Hash**.

Die Subroutine **bless()** erwartet als Parameter genau eine solche Referenz auf einen Hash und einen Klassennamen und erweitert die Funktionalität dieses Hashs zu einem (Objekt) Hash, welcher zusätzlich ein Objekt beschreibt und dessen Daten hält.

Wird ein anderer Konstruktor **new1** mit einem bereits existierende Objekt als Parameter aufgerufen (Copykonstruktor!), **reicht shift()** für die Bestimmung des Klassennamens **nicht aus**.

Beispiel:

```
sub new1 ()
{
        $object = shift;                # Bereits existierendes Objekt, 1.
                                        # Parameter
        my @otherpar = @_;              # restliche Parameter für den
                                        # Konstruktor
        $classnam = ref($object);       # String, enthält Klassenname des Objektes
        ...                             # Irgendeinen Hash erstellen und
        $h = {};                        # ggfls. aus @otherpar initialisieren
        ...
        return bless($h,$classnam);
}
```

Aufruf:

```
$obj0 = Classnam->new();
$obj0 = new Classnam;
$obj1 = $obj0->new(...);
```

Merke: Die Funktion **ref()** liefert bei Objekten einer Klasse den **Klassennamen** zurück, und nicht den Variablentyp. In **new()** muss man also auf irgendeine Art und Weise unterscheiden können, welche Aufrufparameter welchen Typs angegeben wurden.

Methodendefiniton

```
sub meth()
{
  my $this = shift();          # Zeiger auf Objekt, das Methode aufruft

                               # 1. Parameter ist immer ein impizit
                               # übergebener
                               # Parameter
  my @pars = @_;               # Restliche "normale" Parameter
  ...                          # Mache etwas in der Methode
}
```

Verwendung objektlokaler Datenelemente

Objektlokale (objektspezifische) **Datenelemente** dürfen **nicht im Package** abgelegt werden (da sonst Änderungen sich auf alle Objekte auswirken).

Lösung: der Konstruktor speichert die als Aufrufparameter erhaltenen objektspezifischen Daten in genau **dem (anonymen) Hash** ab, der ***geblessed*** wurde:

```
sub new()
{
  $classnam = shift;
  my $ref = {};                          # Referenz auf anonymen Hash
  my @otherpar = @_;
  bless($ref,$classnam);
  $ref->_init(@otherpar);                # Lege uebrige Parameter in %$ref ab
  return $ref;                           # Referenz auf Objekt
}
sub _init()                              # Anonymen Hash mit Userdaten initialisieren
{
  my $obj = shift;                       # Objekt-Referenz, impliziter 1. Parameter,
                                         # da geblessed
  my %param = @_;                        # Datenpaare in einem Hash ablegen
  foreach $k(keys %param)
  { $obj->{$k} = $param{$k} }            # und in den anonymen Hash kopieren
}
sub get_data()                           # Objektdaten auslesen
{
  my $obj = shift;                       # Objektreferenz, impliziter 1, Parameter
  my $att_name = shift;                  # Attributsbezeichnung
  return $obj->{$att_name};              # Attributsinhalt
}
```

```
sub set_data()                          # Objektdaten ändern
{
  my $obj = shift;                      # Objektreferenz
  my $att_name = shift;                 # Attributsbezeichnung
  my $value = shift;                    # Neuer Wert
  $obj->{$att_name} = $value;           # Attributsinhalt
}
```

Methoden-Vererbung unter Verwendung des Arrays @ISA

Eine Vererbung ist problemlos möglich unter Verwendung des **Arrays @ISA**, in das man innerhalb der abgeleiteten Klasse die Namen der Basisklasse(n) einträgt.

```
package Abgeleitet;          # Klasse Abgeleitet
use vars qw(@ISA);           # Deklariere globales Array @ISA
@ISA = qw(Basis);            # Abgeleitet von Basis oder
@ISA = qw(Base1 Base2);      # Abgeleitet von Basis1 und Basis2
```

Bemerkung: Das Statement

```
use vars qw(<variablenliste>)
```

führt eine Prä-Deklaration der in der Liste aufgeführten Variablen als globale Variablen durch.

Beispiel

```
package Base;                              # Klasse Base
sub meth1 { ... }
sub meth2 { ... }
package Abgeleitet_1;                      # Klasse Abgeleitet_1
use vars qw(@ISA);
@ISA = qw(Base);                           # Abgeleitet_1 von Base
                                           # abgeleitet
sub meth1_1 { ... }
sub meth1_2 { ... }
package Abgeleitet_2;                      # Klasse Abgeleitet_2
use vars qw(@ISA);
@ISA = qw(Base);                           # Abgeleitet_2 von Base
                                           # abgeleitet
sub meth2_1 { ... }
sub meth2_2 { ... }
package Derived;                           # Klasse Derived
use vars qw(@ISA);
@ISA = qw(Abgeleitet_1 Abgeleitet_2);      # abgeleitet von Abgeleitet_1
                                           # und Abgeleitet_2 abgeleitet
sub meth3_1 { ... }
sub meth3_2 { ... }
package main;
use Derived;
($derived=Derived->new())->meth2_1();
```

Vererbung von Datenelementen

Die Vererbung von Datenelementen erfolgt durch den Aufruf des Konstruktors der Basisklasse im Konstruktor der abgeleiteten Klasse.

Die Technik ergibt sich aus dem folgenden Beispiel:

```
# Vererbung von Datenelementen (Aus dem Perlbuch von Farid Hajji)
#=========================================================
package Base;                                   # Basisklasse
sub new {
  my ($cn, $self) = (shift,{});
  $self->{"name"} = shift;                      # Member "name" im Objekt anlegen
  return bless($self,$cn);
}
sub stringify {                                 # Zeigt ein Objekt in Stringform an
  my $self = shift;
  return join(" ", ref($self),map { "$_/$self->{$_}" } sort keys %{$self});
} #=========================================================
package Derived;                                # Abgeleitete Klasse Derived
use vars qw(@ISA);
@ISA = qw(Base);                                # Abgeleitet von Base
sub new {                                       # Zusätzlich: Element "phone"
  my $cn = shift;                               # classname
  my $name = shift;                             # Name eines Members als Aufrufparameter
  my $self = $cn->SUPER::new($name);            # Aufruf des Konstruktors der Basisklasse
  $self->{"phone"} = shift;                     # Erweiterung in Derived um das Member 'phone'
  return $self;                                 # bless() unnötig, wurde schon aufgerufen in
                                                # Basiskl.–Konstr.
}
package main;
my $obj = Derived->new("mary","555-1234");
print $obj->{"name"}, " has phone ", $obj->{"phone"}, "\n";
                                                # mary has phone 555-1234
print $obj->stringify(), "\n";              # Derived name/mary phone/555-1234
```

Überladen von Operatoren

Das **Pragma overload** kann für Klassen die Bedeutung von Operatoren ändern (**man overload**).

Beispiel: Änderung des + oder cmp-Operators.

Verwendung:

```
use overload
            '+' => sub { ... },        # Routine für "addition"
            '-' => sub { ... ],        # Routine für "subtraktion"
            '*' => \&multiply,         # Routine für "multiplikation"
            '""' => 'printIt',         # Stringifizierung
            'cmp' => \&comp;           # Routine für Vergleich
```

Beispiel:

```
package SomeThing;

use overload '+',\&myadd;

package main;
$a = SomeThing->new(57);
$b=5+$a;                # Hier wird für die eigens definierte Addition
                        # myadd aus dem Package aufgerufen
```

13. Was gibt es sonst noch?

Wichtige Themen, die nicht behandelt wurden:

- Umgebungsvariable
- Signal-Behandlung
- System-Calls
- Perl-Funktionen
- CPAN: http://www.cpan.org/CPAN.html
- ...