Einführung in Perl

Version 0.07

Email: mailto:info@fabiani.net

Homepage: http://www.fabiani.net

Bei Fragen: Email an: mailto:schulungen@fabiani.net

Vorwort

The very fact that it's possible to write messy programs in Perl is also what makes it possible to write programs that are cleaner in Perl than they could ever be in a language that attempts to enforce cleanliness.
(Larry Wall, der Vater von Perl, Linux World 1998)

Warum Perl?

1. Perl hat sehr mächtige Möglichkeiten zur Behandlung von Zeichenketten. Da sich sehr viele Problemstellungen daraufhin zurückführen lassen, sind die Anwendungsmöglichkeiten für Perl sehr groß, z.B. mächtigere Shellscripte, Logfile-Auswertungen, Webanwendungen, XML-Transformationen, Web-Spider, ...
2. Perl-Module bieten Schnittstellen zu sehr vielen verschiedenen Systemen, also kann man Perl meistens sehr gut verwenden, um diese zu verbinden oder zu automatisieren, z.B. Datenbanken, Directories, Excel ;-), Telnet, FTP, SSH, Fernsteuerung von Webanwendungen über LWP, zu anderen Programmiersprachen wie C (h2xs, Inline::C), C++, Java, ... Perl wird aus diesem Grund auch "Glue-Language" genannt, weil sie viele verschiedenen Systeme "zusammenklebt".
3. Bei sehr vielen Problemstellungen hat man eine sehr kurze Entwicklungszeit, und man kann häufig entscheiden, auf welcher Ebene man ein Problem angehen will (z.B. will ich mit einzelnen Zeichen arbeiten, oder mit Zeilen, Absätzen, kompletten Texten, ...)
4. Perl ist sehr portabel und es läuft auf sehr vielen verschiedenen Plattformen
5. Perl ist frei
6. Man findet sehr gute Hilfe im Web (z.B. http://www.perl-community.de/, http://www.perlmonks.org/)

Haupteinsatzgebiete

- Erweiterte und mächtigere Shellscripte
- Fernsteuerung und Überwachung von anderen Anwendungen; Schnittstellen zu anderen Programmen/Diensten
- Verarbeitung von Texten in irgendwelchen Formen (z.B. Textdateien, Datenbanken, Directories, Netzwerkdienste,...)
- Zugriff auf Datenbanken, Directories, Internetdienste, (ftp, http, mail, ...)
- Scannen und Parsen großer Datenmengen
- XML und dessen Kinder (smil, svg, ...)
- Anwendungen mit GUI (Tk, wxPerl, gtk, ...) oder ohne
- Web-Programmierung (Perl/CGI, mod_perl, FastCGI, Catalyst, Maypole, AJAX,...)

... Und vieles vieles mehr... in Perl ist grundsätzlich fast alles möglich, was auch in anderen Programmiersprachen möglich ist; bei manchen Problemstellungen gibt es gute Gründe, Perl zu verwenden, bei anderen hingegen, Perl nicht (oder nur teilweise) zu verwenden (z.B. Treiberprogrammierung, Betriebssystemkernel, Echtzeitanwendungen, 3D-Spiele, ...).

Was bedeutet der Name Perl?

Practical **E**xtraction and **R**eport **L**anguage

Eine weitere Deutung ist: **P**athologically **E**clectic Rubbish **L**ister

Wie alt ist Perl?

Perl 1.00 wurde 1987 freigegeben. 1994 wurde Perl5.0 released. Die aktuelle Version Perl5.8 wurde 2002 released.

Wo liegen die Wurzeln von Perl?

Perl hat Anleihen aus den verschiedensten Sprachen und Tools genommen, wie C, Shell-Programmierung, LISP, Basic, sed, awk und vielen anderen. Die Syntax ist sehr C-ähnlich, aber viel freier. Perl ist übrigens keine Sprache, die auf einer Universität entstanden ist (wie z.B. Pascal und dessen Abkömmlinge), sondern wurde von einem Sprachwissenschaftler entwickelt und versucht, sich an der gesprochenen Sprache (d.h. Englisch) zu orientieren. Manche Features von Perl haben auch in anderen Sprachen ihre Wurzeln hinterlassen, vor allem die erweiterten Regulären Ausdrücke (PHP, Python, Tcl, Ruby, Java), verschiedene Sprachelemente (z.B. Tcl), ...

Weitere Infos rund um Perl im Web

Weitere Informationen rund um Perl findet man auf den Webseiten http://www.perl.org/ und http://www.perl.com/

Basiswissen

Das erste Programm:

Code: hello_1.pl

```
1: #! /usr/bin/perl
2:
3: print "Hello World\n";
```

In der ersten Zeile steht der "Shebang", d.h. dort wird angegeben womit dieses Script ausgeführt werden soll (analog zu Shellscripten). Nur ist es bei Perl nicht **/bin/sh** oder ähnliches, sondern **/usr/bin/perl**.
In der dritten Zeile steht ein Kommando: gebe am Bildschirm einen Text aus und danach einen Zeilenumbruch **\n**
(Die Zeilennummern gehören nicht zum Code, ich füge sie jedoch hier in diesem Vortrag immer ein, damit man die Zeilen besser benennen kann, ebenso wie ich den Code farbig mache. Normales Schwarz/Weiß genügt für Perl)

Diese Zeilen könnte man mit einem (fast) beliebigen Texteditor (siehe auch das Kapitel: **Für Perl geeignete Editoren**) in eine Datei namens *hello_1.pl* schreiben, (unter Linux/Unix der Datei dann Execute-Rechte geben (**chmod 755 hello_1.pl**)) und sie dann in der Shell ausführen: **./hello_1.pl** oder **perl hello_1.pl**. Daraufhin würde an der Shell die Ausgabe: *Hello World* erscheinen

Obwohl die Dateiendung unter Linux keine Bedeutung hat, wird für Perl-Scripte häufig die Endung **.pl** verwendet. Unter Windows hingegen werden Dateien normalerweise nach ihrer Dateiendung Programmen zugeordnet, deshalb wird fast immer die Endung **.pl** verwendet. In manchen Büchern (z.B. "Einführung in Perl" von Randal Schwartz) wird die Endung **.plx** (für Perl-Executable) empfohlen. Dies war bei älteren Versionen (Perl4) sinnvoll, um Perl-Module (=Bibliotheken) und Skripte auseinander halten zu können. Dies ist aber bei Perl5 nicht mehr nötig, weil die Module die Endung .pm bekommen haben, und somit keine Doppeldeutigkeit mehr auftritt.

Skalare Variablen:

Code: hello_2.pl

```
1: #! /usr/bin/perl
2:
3: $greetingMessage = "Hello World";
4:
5: print "$greetingMessage\n";
6: print $greetingMessage . "\n";
```

In dieser Variante wird die Zeichenkette einer Variablen namens $greetingMessage zugewiesen (Zeile 3). Variablen beginnen zunächst immer mit einem $-Zeichen und sie können Zeichen(-ketten) und Zahlen aufnehmen. Weil sie genau einen Wert aufnehmen können, bezeichnet man sie in Perl auch als **skalare Variablen**. Gültige Variablennamen beginnen mit einem $ und danach einem Buchstaben oder einem Unterstrich, gefolgt von beliebig vielen anderen Buchstaben, Zahlen und Unterstrichen: $[A-Za-z_][A-Za-z0-9_]*. Deutsche Umlaute sollten nicht verwendet werden, ebenso die Namen $a und $b (im Kapitel Sortierungen dazu später mehr).

- Eine skalare Variable beginnt immer mit dem Zeichen $
- Die Zuweisung erfolgt über das Zeichen = (Zeile 3)
- Jeder Ausdruck wird durch ein Semikolon ; abgeschlossen (ähnlich wie in C/C++/Java/Pascal)
- Die Funktion print gibt alles bis zum Funktionsende aus (Zeile5)
- Skalare Variablen können auch in Zeichenketten mit "...." vorkommen. (Zeile 5)
- Man kann mehrere Zeichenkette mit dem . zusammenhängen. Dabei sollte man darauf achten, dass vor und nach dem Operator stehts mindestens ein Leerzeichen/Tabulator/Zeilenumbruch (=Whitespace) steht. (Zeile 6)

Input von der Shell lesen und auswerten:

Code: hello_3.pl

```
1: #! /usr/bin/perl
2:
3: print "Erzaehl mir was: ";
4: $input = <STDIN>;
5: chomp( $input );
6: print "Du sagtest: $input\n";
```

- Die Zeilen 1,3 und 6 verstehen wir schon
- In Zeile 5 wird die Funktion chomp() auf eine Variable namens $input angewendet. Diese Funktion schneidet einen Zeilenumbruch am Ende einer Zeichenkette ab und berücksichtigt dabei, was das Betriebssystem unter einem Zeilenumbruch versteht.
- In Zeile 4 wird der Variablen $input <STDIN> zugewiesen. STDIN ist ein Handle, der Eingaben von der Shell liest, und zwar in unserem Beispiel eine Zeile. Da diese Eingabe mit der Enter-Taste abgeschlossen werden muß, was einem Zeilenumbruch entspricht, ist am Ende von $input ein Zeilenumbruch, der bei zukünftigen Aktionen Schwierigkeiten bereiten könnte (häufiger Fehler). Also entfernen wir diesen nach dem Einlesen, was das chomp () in Zeile 5 erledigt.
- Hilfe zu Perl-Funktionen wie **chomp()** erhält man, indem man in die Shell das folgende Kommando eingibt: `perldoc -f chomp`. Auf www.perl-community.de/ wird übrigens nach und nach die Perl-Dokumentation auf Deutsch übersetzt.

Mehr Kontrolle:

Code: hello_4.pl

```
01: #! /usr/bin/perl
02:
03: print "Sag was? ";
04: chomp( $text = <STDIN> );
05:
06: if( $text eq "Ende" or $text eq 'ende' ) {
07:     print "Ciao!\n";
08: }
09: else {
10:     print "Du sagtest: $text\n";
11: }
```

- Zeile 4: ist identisch mit den Zeilen 4 und 5 des letzten Beispiels, nur die Schreibweise ist etwas kompakter.
- Zeile 6: Zwei Zeichenketten kann man mit dem Operator **eq** vergleichen.
- Zeile 6: Mit **if** (Bedingung) { .... } kann man Anweisungen bedingt ausführen.
- Zeile 6: mit **or** oder **and** kann man mehrere Bedingungen miteinander verknüpfen.
- Zeile 9: falls die Bedingung des **if**'s nicht wahr ist, wird der Inhalt des Blocks { } nach dem if nicht ausgeführt, sondern, wenn vorhanden, der Code des **else**-Blocks
- Zeile 4: in diesem Beispiel sieht man besser, wie das chomp(...) den Vergleich in Zeile 6 vereinfacht. Würde man das chomp weglassen, müßte im Vergleich der Zeilenumbruch am Ende berücksichtig werden, und somit folgendermaßen aussehen: `if ($text eq "Ende\n" or $text eq "ende\n"){`
- Zeile 6: Eine Zeichenkette kann man nicht nur mit doppelten, sondern auch mit einfachen Anführungszeichen angeben ('ende'). Der Unterschied zwischen den beiden ist, dass bei den doppelten enthaltene Variablen und Sonderzeichen (z.B. der Zeilenumbruch \n) interpretiert (d.h. durch den Wert ersetzt) werden, während bei den einfachen Anführungszeichen alles so verwendet wird, wie es da steht; '\n' würde für die beiden Zeichen Backslash und n stehen, während "\n" für einen Zeilenumbruch steht.

Quasselei

Code: hello_5.pl

```
01: #! /usr/bin/perl
02:
03: # Konfiguration
04: $prompt = "Sag was (mit <Enter> absenden): ";
05: $answer = "Du sagtest:";
06:
07: print $prompt;
08: chomp( $input = <STDIN> );
09:
10: while( $input ne "ende" ) {
11:     print "$answer $input\n";
12:
13:     print $prompt;
14:     chomp( $input = <STDIN> );
15:
16: } # while
17:
18: print "Ciao!\n";
```

- Zeilen 3,16: Mit dem Zeichen # kann man einen Kommentar einleiten, der bis zum Zeilenende geht. # innerhalb einer Zeichenkette ist jedoch nur ein normales Zeichen, muss also nicht durch einen vorangestellten Backslash "escaped" werden.
- Zeile 10: Mit dem Operator **ne** vergleicht man zwei Zeichenketten, und es gibt wahr zurück, wenn die beiden Zeichenketten unterschiedlich sind. (**ne** steht für not equal, nicht gleich). **ne** ist also das Gegenteil von **eq**.
- Zeile 10: der Befehl **while**(Bedingung) { .... }wiederholt den Block so lange, wie die Bedingung wahr ist. In unserem Fall lautet die Bedingung: solange das $input nicht 'ende' ist, führe den Code in dem Block aus (quassel weiter, bis man ende eingibt).

Zahlen:

Code: hello_6.pl

```
01: #! /usr/bin/perl
02:
03: # Konfiguration
04: $requiredAge = 25;
05:
06: # Programmcode
07: print( "Wie alt bist du? " );
08: chomp( $age = <STDIN> );
09:
10: if ($age == $requiredAge) {
11:     print "Ah, du bist $requiredAge Jahre alt\n";
12: } # if
13:
14: elsif ($age < $requiredAge) {
15:     print( "Du bist noch nicht " . $requiredAge . "\n" );
16: } # elsif
17:
18: else {
19:     print "Ah, du hast deinen $requiredAge. Geburtstag schon hinter dir\n";
20: } # else
```

- Zeile 04 enthält nun eine Zahl, keine Zeichenkette
- Zeile 07/15: Man kann print auch als Funktion schreiben (mit Klammern), wenn man will, muß aber nicht (Zeilen 11/19)
- Zeile 14: es gibt nicht nur **if - else**, sondern man kann es auch noch erweitern: **if - elsif - else** (für C/Java-Programmierer: es heißt nicht else**if**)
- Vergleichsoperatoren sind für Zahlen anders als für Zeichenketten

Zahl	Zeichenkette	Bedeutung
==	eq	Sind die beiden gleich?
!=	ne	Sind die beiden nicht gleich?
<	lt	Ist der linke Wert kleiner?
>	gt	Ist der rechte Wert kleiner?
<=	le	Ist der linke Wert kleiner oder gleich?
>=	ge	Ist der rechte Wert kleiner?
<=>	cmp	-1 wenn links kleiner, 0 wenn gleich, +1 wenn links größer

Achtung: = ist eine Zuweisung, kein Vergleich: **$x = $y (Häufiger Fehler!)**

- Dualismus Zahl <-> Zeichenkette: was wie aufgefasst wird, ergibt sich aus dem Zusammenhang, und es wird hinter den Kulissen umgewandelt (d.h. eigentlich werden bei Bedarf beide Werte gespeichert, wenn nötig; man kann also auch Zahlen mit String-Vergleichen behandeln, oder umgekehrt, wenn man das wirklich will)
- Ist "1.0" gleich "1" ?
- Ist "abc" gleich "def" ?
- Ist "NULL" gleich "0" ?
- Ist "20x" gleich "20" ?
- Ist "4" gleich "5" ?

Zahlenformate

Zahlen können auf viele verschiedene Arten angegeben werden, z.B.

Zahl	Beschreibung
123	"normale" Dezimalzahl
123_456_789	bessere Lesbarkeit für lange Zahlen (optional mit _)
0123	Oktalzahl (führende Ziffer 0)
0xFF	Hexadezimalzahl (führende Ziffer 0 und danach ein x)
0b1101_1100	Binär (führende Ziffer 0 und danach ein b) (_ für bessere Lesbarkeit)
6.02e23	Wissenschaftliche Notation
1.14159	Fließkommazahl (Punkt ist Trennzeichen)

Rechenoperationen

Operation	Erklärung
$z = $x + $y	Addition
$z = $x - $y	Subtraktion
$z = $x * $y	Multiplikation
$z = $x / $y	Division (Ergebnis: wenn möglich, eine Integerzahl, sonst eine Fließkommazahl).
$z = $x % $y	Modulo (Rest der Integerdivision)
$z = $x ** $y	Potenzierung

```
print 4 + 2 * 3; # 10, nicht 18: Punkt- vor Strichrechnung
```

Wenn der Vorrang nicht auf den erste Blick eindeutig ist, besser ein
paar Klammern setzen.

```
print ((4+2) * 3); # 18
```

Kurzschreibweisen

Kurzschreibweise	Langschreibweise	Erklärung
**$i**++ oder ++$i	$i = $i + 1	$i wird um eins erhöht
**$i**-- oder --$i	$i = $i -1	$i wird um eins vermindert
$i += 2	$i = $i + 2	$i wird um zwei erhöht
$i -= 5	$i = $i - 5	$i wird um 5 vermindert
$i *= 2	$i = $i * 2	$i wird verdoppelt
$i .= 0	$i = $i . "0"	An $i wird eine 0 angehängt (Behandlung als Zeichenkette durch den Punkt!)

Wenn man ++$i oder $i++ als eigenständigen Ausdruck verwendet, sind die beiden vom Verhalten her identisch (ebenso --$i und $i--). Wenn es sich jedoch um eine Zuweisung oder ähnliches handelt, verhalten sich die beiden unterschiedlich:

$i++ gib zuerst den alten Wert von $i zurück und erhöht dann $i um eins.
++$i erhöht zuerst den Wert von $i und gibt dann den erhöhten Wert zurück.
Analog verhalten sich $i-- und --$i

```
$i = 5;
$j = $i++; # $j=5, $i=5+1=6
$k = ++$i; # $k = $i+1=7, $i=6+1=7
```

Wahrheit oder Lüge?

In Perl gibt es weder TRUE noch FALSE, sondern diese Werte werden durch Zahlen oder Zeichenketten dargestellt (fast genauso wie in C):

Wert	Bedeutung
0 oder "0", Leere Zeichenkette ""	Falsch
Wert nicht initialisiert: undef(ined)	Falsch
Alles andere ist:	Wahr

Verschiedene Zählerschleifen

While-Schleife: kompliziert

Code: loop_1.pl

```
01: #! /usr/bin/perl
02:
03: $max = 5;
04:
05: $counter = 0;
06: while( $counter <= $max ) {
07:     print "$counter\t";
08:
09:     $counter++;
10: } # while
11:
12: print "\n";
```

Output:

```
bash-2.05b$ perl 1_loop.pl

0       1       2       3       4       5
```

- Zeile 3: die Variable, die die obere Grenze enthält, wird initialisiert
- Zeile 5: die Zählvariable wird initialisiert
- Zeile 6: Die Schleifenbedingung: solange $counter kleiner oder gleich **$max** ist, führe den Block aus
- Zeile 7: Gebe den Wert aus, und danach einen Tabulator **\t**
- Zeile 9: Erhöhe den **$counter** um eins

for-Schleife im C-Stil: noch komplizierter

Code: loop_2.pl

```
1: #! /usr/bin/perl
2:
3: $max = 5;
4:
5: for( $counter=0; $counter <= $max; $counter++ ) {
6:     print "$counter\t";
7: } # for
8:
9: print "\n";
```

Der Output ist identisch

foreach-Schleife im Perl-Stil:

Code: loop_3.pl

```
1: #! /usr/bin/perl
2:
3: $max = 5;
4:
5: foreach $counter (0..$max) {
6:     print "$counter\t";
7: } # foreach
8:
9: print "\n";
```

- Zeile 5: foreach $laufvariable ($start..$end) { ... }. Anstelle von foreach kann man auch for schreiben, oder umgekehrt
- **$start..$end** erzeugt eine Liste von Werten. Nicht nur Zahlen können verwendet werden, sondern auch Buchstaben:
- Bei jedem Durchlauf dieser Schleife verweist $laufvariable auf den Wert, über den man gerade iteriert. (nur bei for(each))
- $laufvariable behält seinen Wert nur innerhalb des for(each)-Blockes; der vorherige Wert (hier undef) wird nach der Schleife wiederhergestellt (nur bei for(each))
- Seit perl5.6 kann man diese Variante auch ohne Gefahr bei Schleifen verwenden, die über viele Elemente iterieren.

Code: loop_4.pl

```
1: #! /usr/bin/perl
2:
3: foreach $char ('a'..'z') {
4:     print "$char ";
5: } # foreach
6:
7: print "\n";
```

Output:

```
bash-2.05b$ perl 4_loop.pl

a b c d e f g h i j k l m n o p q r s t u v w x y z

bash-2.05b$
```

Aus der letzten foreach-Schleife kann man auch die Liste herausziehen:

Code: loop_5.pl

```
1: #! /usr/bin/perl
2:
3: @liste = ('a'..'z');
4:
5: foreach $char ( @liste ) {
6:     print "$char ";
7: } # foreach
8:
9: print "\n";
```

- Zeile 3: Die Liste @liste wird initialisiert
- Zeile 5: Über die Liste @liste wird iteriert

Einfache Suchfunktionen:

- **$pos = index($text, $suchString);** sucht nach dem Vorkommen von **$suchString** in **$text** und gibt die Position (als Zahl) aus, an der **$suchString** gefunden wurde. Die Zählweise beginnt dabei (wie bei Listen) mit 0, also das erste Zeichen ist 0, das zweite 1, usw. Wird **$suchString** nicht in **$text** gefunden, wird die Zahl **-1** zurückgegeben, nicht **0 (Achtung: häufiger Fehler)**.
- **$pos = index($text, $suchString, $startPosition)**: zusätzlich kann eine **$startPosition** angegeben werden, d.h. ab welchem Zeichen gesucht werden soll. Das ist oft nützlich, wenn man mehrere Vorkommen von **$suchString** finden will.
- **$pos = rindex($text, $suchString);** wie **index(...)**, nur beginne die Suche von hinten
- Mächtigere Wege finden wir im Bereich der regulären Ausdrücke

Nützliche Stringfunktionen:

- **$str = substr($text, $pos)**: gibt den Inhalt von **$text** von der Position **$pos** bis zum Ende von **$text** zurück
- **$str = substr($text, $pos, $laenge)**: gibt **$laenge** Zeichen von **$text** ab der Position **$pos** zurück.
- Mächtigere Wege finden wir im Bereich der regulären Ausdrücke

```
$string = "Perl ist toll";

print substr( $string, -4   ); # toll
print substr( $string, 9    ); # toll
print substr( $string, 0, 1 ); # P
print substr( $string, 5, 3 ); # ist
```

Variablentyp eindimensionale Liste/Array:

- Eine Variable, die eine Liste aufnehmen will, beginnt stets mit dem Zeichen @ (z.B. **@liste**)
- Auf einen Wert der Liste kann man über $ vor und ein Index in [] nach dem Listennamen zugreifen, z.B. **$liste[3]**. @liste[3] verhält sich zwar auf den ersten Blick gleich, bietet jedoch einige Gefahren (siehe die weiteren Kapitel). (In Perl6 bleibt dann auch da das @-Zeichen: @liste[3])
- Den Index des letzten Elementes bekommt man durch den Wert von **$#liste** heraus
- Die Anzahl der Elemente bekommt man durch **scalar(@liste);**

Code: loop_6.pl

```
01: #! /usr/bin/perl
02:
03: @liste = ('a'..'z');
04:
05: foreach $index (0..$#liste) {
06:     print "$index => $liste[$index]\t";
07: } # foreach
08:
09: print "\n\n";
10: print "Index des letzten Elementes: $#liste\n";
11: print "Anzahl Elemente: ", scalar(@liste), "\n";
12: print "Anzahl Elemente: " . @liste . "\n";
```

- Zeile 5: Iteriere über eine Liste mit den Zahlen 0 bis zum letzten Index der Liste **@liste**
- Zeile 6: Gib den aktuellen Wert von $index aus, danach => und danach den Wert des entsprechenden Elementes von **@liste**, und danach einen Tabulator **\t**
- Zeile 9: zwei Zeilenumbrüche
- Zeile 11: Da **scalar(...)** eine Funktion ist, kann sie nicht innerhalb einer Zeichenkette ausgeführt werden. Deshalb wurde die Zeichenkette unterbrochen.
- Zeile 12: Der . Operator erzwingt einen skalaren Kontext. Wenn man ein Array im skalaren Kontext ausführt, gibt es die Anzahl seiner Elemente zurück, wirkt also genauso wie scalar(...). Ich empfehle jedoch, besser das explizite scalar(...) zu verwenden.
- Was ergibt: `print "Elemente: @liste\n";`
- Was ergibt: `print "Elemente: ", @liste, "\n";`
- Was ergibt: `print "Elemente: ", @liste . "\n";`
- Was ergibt: `print "Elemente: . @liste . \n";`

Output:

```
F:\apacheweb\test_8085\html\codes>6_loop.pl

0 => a  1 => b  2 => c  3 => d  4 => e  5 => f  6 => g  7 => h  8 => i  9 => j
10 => k 11 => l 12 => m 13 => n 14 => o 15 => p 16 => q 17 => r 18 => s 19 => t
20 => u 21 => v 22 => w 23 => x 24 => y 25 => z

Index des letzten Elementes: 25
Anzahl Elemente: 26
F:\apacheweb\test_8085\html\codes>
```

Initialisierung einer Liste:

1. Jedes Element einzeln zuweisen:

```
$liste[0] = 'A';
$liste[1] = 'B';
$liste[2] = 'C';
$liste[3] = 'D';
```

2. Mehrere Elemente auf einmal zuweisen:

```
@liste = ('A', 'B', 'C', 'D');
```

3.Einen Bereich zuweisen

```
@liste2 = ('A'..'D');
```

4. Komfortablere Schreibweise mit **Whitespace** als Trennzeichen

```
@liste3 = qw(A B C D);
```

Whitespace ist eine Zeichenklasse: Leerzeichen, Tabulator, Zeilenumbruch. Auch folgendes ist erlaubt:

```
@liste4 = qw(A B
             C D);
```

Einfache Listenfunktionen:

Funktion:	Beschreibung:	Beispiel:
`$str = join ("sep", @list);`	Vereinigt die Elemente einer Liste zu einer Zeichenkette, indem sie mit dem Trennzeichen **sep** zusammengehängt werden.	`$string = join("\n", @liste4);`
`@list = split (/sep/, $str, $anzahl);`	Teilt eine Zeichenkette in eine Liste (mit - wenn vorhanden - $anzahl Elementen) auf, wobei immer beim Trennzeichen **sep** aufgeteilt wird	`@liste = split(/\n/, $string);`
`push(@list, $e1, $e2);`	Fügt ein neues Element (oder eine Liste) ans Ende von @list	`push(@list, $#list+1, $#list+2);`
`$lastE = pop(@list);`	Entfernt das letzte Element von @list und gibt es zurück	`$lastE = pop(@list);`
`$firstE = shift(@list);`	Entfernt das erste Element von @list und gibt es zurück	`$firstE = shift(@list);`
`unshift(@list, $e1, $e2);`	Fügt ein neues Element (oder eine Liste) an den Beginn von @list	`unshift (@list, -2, -1);`

Code: array_1.pl

```
01: #! /usr/bin/perl
02:
03: @liste = ('A'..'J');
04:
05: $string = join(" x ", @liste);
06: print "JOIN : $string\n";
07:
08: @liste2 = split(/ x /, $string);
09:
10: print "SPLIT: ";
11: foreach $buchstabe (@liste2) {
12:     print "$buchstabe   ";
13: } # foreach
14:
15: print "\n";
```

Output:

```
F:\apacheweb\test_8085\html\codes>perl 1_array.pl
JOIN : A x B x C x D x E x F x G x H x I x J
SPLIT: A   B   C   D   E   F   G   H   I   J

F:\apacheweb\test_8085\html\codes>
```

Lesen von Textdateien:

Vorgehensweise:

1. Öffne eine Datei
2. eine oder mehrere Zeilen lesen
3. Datei schließen

In Perl könnte das folgendermaßen aussehen:

```
1. open( FILEHANDLE, "datei.txt" );
2. foreach $line ( <FILEHANDLE> ) { print $line; }
3. close (FILEHANDLE);
```

Durch das **open** wird die sich im Ausführungsverzeichnis befindliche Datei "datei.txt" dem Programm unter FILEHANDLE bekannt gemacht, mit dem dann das Programm was anfangen kann (<....>, close...)

Was passiert, wenn es die Datei "datei.txt" nicht gibt oder sie nicht gelesen werden kann? Perl selbst versucht da einfach, stillschweigend weiterzumachen, als wäre alles ok. Da muß man sich dann als Programmierer selbst drum kümmern. Perl hilft einem dabei, indem **open** einen wahren Wert (meistens 1) zurückgibt, wenn es ok ist, und einen falschen Wert (**undef**) bei einem Fehler. Eine Fehlerbeschreibung steht in der Perl-Variable **$!**

Code: file_1.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "gibtEsNicht.txt";
04:
05: $ergebnis = open(FH, $datei);
06:
07: if ($ergebnis) { # alles ok gegangen
08:
09:     print "ok\n";
10:     close (FH); # wieder schliessen
11: } # if
12:
13: else { # Fehler
14:     die "Error: couldn't open file '$datei': $!";
15: } # else
```

Output:

```
F:\apacheweb\test_8085\html\codes>perl 1_file.pl
Error: couldn't open file 'gibtEsNicht.txt': No such file or directory at 1_file
.pl line 14.
```

Ein Zeilenumbruch nach der Ausgabe des **die** unterdrückt das **at 1_file.pl line 14.**, z.B.
`die "Error: couldn't open file '$datei': $!\n";`
Es hilft oft, wenn man den Dateinamen in einer Variable abspeichert, und diese Variable dann sowohl beim open als auch bei der Fehlermeldung ausgibt. Wenn man den Namen nämlich in der Fehlermeldung hardcoded, dann passt oft die Fehlermeldung nicht zur Ursache, und man sucht den Fehler oft an der falschen Stelle.

Es gibt jedoch noch schönere Möglichkeiten zur Fehlerabfrage:

Code: file_2.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "gibtEsNicht.txt";
04:
05: if( open(FH, $datei) ) {
06:     print "ok\n";
07:     close (FH); # wieder schliessen
08: } # if
09:
10: else { # Fehler
11:     die "Error: couldn't open file '$datei': $!\n";
12: } # else
```

Code: file_3.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "gibtEsNicht.txt";
04:
05: unless( open(FH, $datei) ) {
06:     die "Error: couldn't open file '$datei': $!\n";
07: } # if
08:
09: else { # ok
10:     print "ok\n";
11:     close (FH); # wieder schliessen
12: } # else
```

- **unless (...)** bedeutet **if (not ...)** oder **if (! ...)**
- **unless** sollte man nicht verwenden, wenn dadurch die Bedingung zur doppelten Verneinung wird , weil das Programm sonst schwieriger verständlich wird.
- Da durch das **die** das Programm beendet wird, könnte man auf das **else** auch verzichten

Code: file_4.pl

```
1: #! /usr/bin/perl
2:
3: $datei = "gibtEsNicht.txt";
4:
5: open(FH, $datei) or die "Error: couldn't open file '$datei': $!\n";
6:
7: print "ok\n";
8: close (FH); # wieder schliessen
```

- In Perl gibt es (wie auch in C/C++, Java, ...) eine Kurzschlußauswertung bei boolschen Ausdrücken, d.h. wenn ein Ausdruck nicht mehr wahr werden kann, wird er auch nicht mehr ausgewertet. Das **or** ist ein logisches ODER, d.h. der Gesammtausdruck ist wahr, wenn eine der beiden Seiten wahr ist (oder auch beide) und falsch, wenn beide Seiten falsch sind. Dieses Verhalten wird in Perl häufig dazu mißbraucht, ein paar Buchstaben Code zu sparen. Der Ausdruck **Aktion or die ...** wird sehr häufig verwendet, um eine Aktion auf Fehler zu überprüfen.
  Anmerkung: Falls der erste Ausdruck falsch ist, wird der zweite ausgewertet und zurückgegeben.
  `print 0 or 4; => 4`
  `print 2 or 4; => 2`
- Das logische UND heißt **and** und funktioniert analog, nur daß da beide Seiten wahr sein müssen, um als Ergebnis wahr zurückzuliefern.
  `print 0 and 4; => 0`
  `print 2 and 4; => 2`
- **and** und **or** werden häufig auch als Bedingungen verwendet, z.B.
  `if ($x > 10 and $x < 20) { Anweisung; }`
  `if ($x < 10 or $x > 20)  { Anweisung; }`
- Für C-Programmierer gibt es auch noch || und **&&**, allerdings mit höherer Bindung als **or** und **and**
  `open (FH, $filename) || die;`
  `($value > 2) && print "Der Wert $value ist groesser als 2\n";`
- Ich empfehle, im Zweifel **and** und **or** zu verwenden (eventuell mit Klammern), weil die einfach besser lesbar sind.

Anmerkung: Viele Leute "vergessen" diese Fehlerüberprüfung und wundern sich dann, wenn dann ein Fehler oder eine Warnung (dazu später mehr) an einer Stelle auftritt, die mit der Fehlerursache nicht mehr mittelbar was zu tun hat. Wenn sie da ein paar Buchstaben mehr getippt hätten, hätten sie sich oder anderen vermutlich eine stundenlange Sucherei ersparen können... Ich schätze mal, daß etwa 80% der Fehler in Perl auf solchen vergessenen Fehlerabfragemöglichkeiten basieren. Und selbst wenn man sich ziemlich sicher ist, daß da eigentlich nichts schief gehen kann: es kann immer mal. passieren, daß eine Datei nicht vorhanden ist oder aus Mangel an Rechten nicht gelesen werden kann. Und wenn man bei der Frage um Hilfe eine Fehlermeldung vorweisen kann, wird oft viel schneller geholfen.

Datei einlesen und am Bildschrim ausgeben:

Code: file_5.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "file_5.pl";
04:
05: unless( open(FH, $datei) ) {
06:     die "Error: couldn't open file '$datei': $!\n";
07: } # if
08:
09: foreach $line (<FH>) {
10:     print $line;
11: } # foreach
12:
13: close (FH);
```

- Zeile 9: **foreach $line (<FH>) {** liest die Datei auf einmal ein:

Code: file_6.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "file_6.pl";
04:
05: unless( open(FH, $datei) ) {
06:     die "Error: couldn't open file '$datei': $!\n";
07: } # if
08:
09: print foreach <FH>;
10:
11: close (FH);
```

- Zeile 9: dasselbe wie oben, nur eine Kurzschreibweise. Da **print** eine Liste erwartet, könnte man das foreach auch weglassen: **print <FH>;**

Code: file_7.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "file_7.pl";
04:
05: unless( open(FH, $datei) ) {
06:     die "Error: couldn't open file '$datei': $!\n";
07: } # if
08:
09: while ($line = <FH>) {
10:     print $line;
11: } # foreach
12:
13: close (FH);
```

- Zeile 9: dasselbe mit einer **while**-Schleife: Da **while** jedoch nicht wie **for(each)** eine Liste aufnimmt, sondern einen skalaren Ausdruck, wird die Datei Zeile für Zeile eingelesen und beim nächsten Zeilenbeginn durch die nächste Zeile ersetzt. Dies ist vorteilhaft, wenn man große Dateien hat, weil da immer nur eine Zeile im Speicher landet, während bei **for(each)** die komplette Datei in den Speicher geladen wird.
- **<FH>** gibt also, je nach Kontext, das Ergebnis in unterschiedlicher Weise zurück (siehe die beiden folgenden Beispiele):
    - Skalarer Kontext: gibt eine Zeile der Datei zurück

    - Listenkontext: gibt die komplette Datei als Liste zurück
- Da **while** die Schleife solange ausführt, solange die Schleifenbedingung wahr ist, wird die komplette Datei eingelesen.
- Was passiert, wenn in der letzten Zeile der Datei eine 0 ohne einen folgenden Zeilenumbruch steht? Bei neueren Perl-Versionen wird da 0 ordentlich verarbeitet. Manche ältere Perl-Versionen werten da die 0 als Falsch aus, und brechen die Schleife ab, ohne daß die 0 noch behandelt wird. Deshalb ist es sicherer, da folgende Schreibweise zu verwenden:

Code: file_8.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "file_8.pl";
04:
05: unless( open(FH, $datei) ) {
06:     die "Error: couldn't open file '$datei': $!\n";
07: } # if
08:
09: while (defined($line =<FH>)) {
10:     print $line;
11: } # foreach
12:
13: close (FH);
```

- Zeile 9: Die Funktion **defined($variable)** ist ein wenig strenger und überprüft, ob die Variable **$variable** einen Wert enthält oder nicht. Wenn sie **undef** ist, gibt sie falsch zurück, sonst wahr.

Wert	Wahrheitswert	defined(Wert)
"abcde", 3	Wahr	Wahr
0, ""	Falsch	Wahr
undef	Falsch	Falsch

- **undef** bedeutet, die Variable hat keinen Wert (in anderen Sprachen: NUL, NULL, NILL, ...)
- Man kann auch einer skalaren Variablen den Wert **undef** zuweisen: **$line = undef;**
- **undef** kann man auch als Prozedur verwenden: **undef(@liste);** Das leert den Speicher von **@liste** und wirft sie aus dem Programm raus.
- Wenn man nur den Inhalt einer Liste leeren will, kann man schreiben: **@liste = ();** Das weist der **@liste** eine leere Liste zu.
- Eine Liste kann man auch leeren, indem man ihren letzten Index auf -1 setzt: **$#liste = -1;**

In den folgenden beiden Beispielen sieht man das unterschiedliche Verhalten von **while** und **foreach**:

Code: file_9.pl

```
01: #! /usr/bin/perl
02:
03: print "Eingabe: ";
04: while (defined($line = <STDIN>)) {
05:     chomp($line);
06:     print "Input: $line\n";
07:     if ($line eq 'END') {
08:         last;
09:     } # if
10:     print "Eingabe: ";
11: } # while
```

- Dieses Beispiel wirkt sehr interaktiv, weil man eine Zeile eingibt und sofort eine Antwort herausbekommt
- Man kann in der Schleife überprüfen, ob ein Ende der Eingaben gewünscht ist (hier durch die Eingabe der Zeichenkette END).

Code: file_10.pl

```
1: #! /usr/bin/perl
2:
3: foreach $line (<STDIN>) {
4:     chomp($line);
5:     print "Input: $line\n";
6: } # while
```

- In diesem Beispiel werden im Schleifenkopf alle Eingaben eingelesen, und erst dann beginnt die Schleife zu laufen und gibt alle Zeilen aus. Da der Einlesevorgang nicht von der Schleife, sondern von <STDIN> gesteuert wird, muß man den Einlesevorgang abbrechen, indem man mit der Tastenkombination <Strg><d> (oder unter Windows auch: <Strg><z>) ein EndOfFile signalisiert. Ein Überprüfen auf die Eingabe von END wäre in diesem Beispiel nicht besonders sinnvoll, weil sie einfach zu spät erfolgen würde, nämlich nach der Eingabe der gesammten Liste.

Bei der While-Schleife gibt es noch folgende Kurzschreibweise:

```
while (defined($_ = <FH>)) {
  print $_;
} # while
```

$_ ist eine eingebaute Perl-Variable, die häufig die Standardvariable von Funktionen ist. Sie kann man häufig auch weglassen, z.B

```
while (<FH>) { # hier wird automatisch an $_ zugewiesen und auf defined überprüft
  print; # hier wird automatisch die Standardvariable $_ benützt
} # while
```

Und weil das Motto von Perl TIMTOWDI (=There Is More Than One Way to Do It) lautet, funktionieren auch

```
print while <FH>;
```

```
print <FH>;
```

Man muß halt von Fall zu Fall unterscheiden, welches der Konstrukte die gewählte Absicht am besten ausdrückt.

IO- und File-Handles:

Es gibt in Perl sogenannte Handles, über die man auf Eingaben zugreifen und Ausgaben senden kann. Ein Handle ist ein Variablentyp, der auf eine Datei, auf die Konsole oder auch auf Netzwerksockets usw. zeigen kann. Standardmäßig hat ein Perl-Script (wie fast jedes andere Programm) drei sogenannter IO-Handles verbunden:

- **STDIN**: darüber kann man Eingaben über die Konsole einlesen (oder auch via Pipes), vergleiche **<STDIN>**
- **STDOUT**: darüber kann man Ausgaben in die Konsole machen, z.B. mit **print**
- **STDERR**: darüber kann man Fehler und Warnungen in die Konsole ausgeben, z.B. **die** oder **warn**

Zunächst zeige ich die Verwendung des "klassischen" Filehandles. Dies hat zwar einige Nachteile, aber die fallen bei etwa 90% der Perl-Scripte nicht ins Gewicht. Es ist jedoch wichtig, diesen Weg zu kennen, weil er sehr häufig verwendet wird. Für weitere Informationen siehe das Kapitel zu "Lexikalische Filehandles".

Man kann mit **open (FILEHANDLE, $dateiname)** auch eine Datei mit einem Handle verbinden und dann darüber ansprechen. Diese Handles nennt man Filehandles. Perl stellt identische Funktionen für IO-Handles oder Filehandles zur Verfügung (z.B. **<HANDLE>**, **print HANDLE $text**). Da **STDOUT** der Standardhandle für print usw. ist, kann man den auch weglassen, und braucht nicht immer zu schreiben: **print STDOUT $text;** sondern kann die Kurzschreibweise verwenden, wie wir es bisher getan haben: **print $text;**

Dateien schreiben:

Code: file_write_1.pl

```
01: #! /usr/bin/perl
02:
03: $datei = "file_write_1.txt";
04:
05: unless( open(FH, "> $datei") ) {
06:     die "Error: couldn't write to file '$datei': $!\n";
07: } # if
08:
09: foreach $index (0..10) {
10:     print (FH "Dies ist Zeile $index\n");
11: } # foreach
12:
13: close (FH);
```

- Zeile 05: öffne die Datei **$datei** unter dem Filehandle **FH** zum schreiben. **open (FH, "> $datei")**. Man kann auch schreiben: **open (FH, ">" . $datei)** oder: **open (FH, '>' . $datei)**.
- **Achtung: Wenn da eine schon vorhandene Datei zum Schreiben geöffnet wird, wird die überschrieben.**
- Zeile 10: schreibe einen Text in die Datei, die mit dem Filehandle **FH** zum Schreiben geöffnet ist.
- **Achtung: zwischen FH und der Zeichenkette darf kein Komma angegeben werden!**
- Anmerkung: Man kann Dateien auch folgendermaßen zum Lesen öffnen: **open (FH, "< $datei")**. Wir haben bisher jedoch die abgekürzte Version **open (FH, $datei)** verwendet.

Code: file_write_2.pl

```
01: #! /usr/bin/perl
02:
03: $dateiZumLesen     = "file_write_2.pl";
04: $dateiZumSchreiben = "file_write_2.txt";
05:
06: unless( open (IN, $dateiZumLesen) ) {
07:     die "Error: couldn't read from file '$dateiZumLesen': $!\n";
08: } # unless
09:
10: unless( open(OUT, "> $dateiZumSchreiben") ) {
11:     die "Error: couldn't write to file '$dateiZumSchreiben': $!\n";
12: } # if
13:
14: foreach $line (<IN>) {
15:     chomp($line);
16:     $output = reverse($line);
17:     print (OUT "$output\n");
18: } # foreach
19:
20: close (IN);
21: close (OUT) or
22:     die "Fehler beim Schliessen von '$dateiZumSchreiben': $!\n";
```

- Zeile 6-8: Datei zum Lesen öffnen
- Zeile 10-12: Datei zum Schreiben öffnen
- Zeile 20: Lesedatei wieder schließen
- Zeile 21: Schreibedatei mit Fehlerauswertung wieder schließen (z.B. Festplatte voll)
- Wenn man schon vor dem close() eventuelle Fehler mitbekommen will (z.B: bei Daemons, die monatelang Logdateien schreiben), dann muß man den Rückgabewert von print auswerten.
- Zeile 16: **reverse()** dreht eine Zeichenkette (Zeichen für Zeichen) oder eine Liste (Element für Element) um. **reverse()** reagiert jedoch unterschiedlich, je nachdem, ob man es in skalarem Kontext oder im Listenkontext ausführt. Da wir es durch eine Zuweisung an eine skalare Variable (=**$output**) im skalaren Kontext ausführen, wird die Zeichenkette Zeichen für Zeichen umgedreht und in Zeile 17 ausgegeben. Wenn man Zeile 16 und 17 zusammenziehen will, kann man deshalb nicht schreiben
  print (OUT reverse($line), "\n");
  weil print einen Listenkontext erzwingt, **$line** dann als Liste mit einem Element angesehen wird und deshalb scheinbar nichts passiert. Durch den richtigen Operator kann man jedoch das **reverse()** in einen skalaren Kontext zwingen, z.B.
  print (OUT reverse($line) . "\n");

print (OUT scalar(reverse($line)), "\n");
und die Zeichenkette wird - wie gewünscht - Zeichen für Zeichen umgedreht in die Datei, die mit dem Filehandle OUT verbunden ist, geschrieben..

Ausgabefunktionen:

- print ($zeichenkette);
- print (FH $zeichenkette);
- printf ("Muster", @werte);
- printf (FH "Muster", @werte);

Die Klammern sind hier optional.

Die Funktion **printf()** wurde aus den Sprachen C/C++ übernommen und funktioniert (fast) identisch:

```
z.B. printf("Zahl: %i Zeichenkette: %s        \n", 20, 'text');
```

Mit % werden Platzhalter (eventuell mit Formatanweisungen) angegeben. Dabei sind folgende möglich:

Platzhalter:	Ersetzung:	Beispiel:	Ausgabe:
%%	ein normales %-Zeichen	`printf("%%");`	%
%s	eine Zeichenkette	`printf("%s", 'abc');`	abc
%d oder %i	eine Integerzahl	`printf("%i", 255);`	255
%o	Integer in Oktalschreibweise	`printf("%o", 255);`	377
%x	Integer in Hexadezimalschreibweise	`printf("%x", 255);`	ff
%f	eine Fliesskommazahl	`printf("%f", 30.22);`	30.220000
%e	Fliesskommazahl in wissenschaftlicher Notation	`printf ("%e", 30.22);`	3.022000e+001

Es gibt noch weitere Platzhalter, die jedoch nur selten verwendet werden. Eine vollständige Liste erhält man, indem man `perldoc -f printf` in die Kommandozeile eingibt.

Man kann bei einem Platzhalter noch weitere Optionen angeben, und zwar zwischen dem % und dem Buchstaben:

Zeichen:	Bedeutung:	Beispiel:	Ausgabe:
Zahl	Die minimale Feldlänge	`printf('%10s %4s', 'ab', 'cd');`	ab cd
Leer-zeichen	Bei Rechtsbündiger Ausgabe werden vor die Zahl/Zeichenkette Leerzeichen ausgegeben	`printf('% 20s', 'abcd');`	abcd
Ziffer 0	Bei Rechtsbündiger Ausgabe werden vor die Zahl/Zeichenkette Nullen geschrieben	`printf('%04i %04i', 20, 3);`	0020 0003
+	Stelle vor positiven Zahlen ein Plus dar	`printf('%+4i', 20);`	+20
-	Gib die Zeichenkette/Zahl linksbündig aus (Standard ist: Rechtsbündig)	`printf('%-+4i', 20);`	+20
Zahl.Zahl	Genauigkeit bei Fließkommazahlen (kann man wunderbar zum Runden verwenden).	`printf('%4.2f', 30.2272);`	30.23

Damit kann man die Ausgabe recht schön formatieren

Damit man diese Formate auch bei Zuweisungen verwenden kann, gibt es in Perl zusätzlich noch die Funktion **sprintf()**, die identisch wie **printf** funktioniert, allerdings die Zeichenkette einer Variablen zuweist, z.B.

```
$output = sprintf("%4d %10s\n", $lineNumber, "Zeile" . $lineNumber);
$gerundet = sprintf("%4.3f", 12.1259);
```

(s)printf hat jedoch den Nachteil, daß man nur eine Mindestlänge angeben kann, nicht jedoch eine Maximallänge. Falls da Zeichenketten länger sind als die angegebene Stellenzahl, kann so das Format zerstört werden. Dazu kann man entweder die Zeichenketten vorher mit `$var = substr($var, 0, $maximalLänge);` ausschneiden, oder die Funktion **pack** verwenden. Diese Funktion erwartet ähnlich wie (s)printf als erstes Argument einen Formatstring, und danach eine Liste von Werten. Die Platzhalter im Formatstring funktionieren jedoch anders als bei (s)printf:

```
my $string = pack("A10 A20", 'Langer String', 'Kurzer String');
print $string, "\n";
```

unpack wird häufig zum Schreiben von Dateien mit fixen Satzlängen benützt. Es kann jedoch noch viel mehr, so z.B. Konvertierungen zwischen verschiedenen "Formaten". Für nähere Informationen siehe `perldoc -f pack`. Die Umkehrfunktion von **pack** ist **unpack**, was als ersten Parameter einen Formatstring erwartet und als zweiten einen String. Dieser String wird anhand des Formates aufgeteilt und/oder konvertiert und als Liste zurückgegeben. Die Formatoptionen sind dieselben wie bei **pack**, nur haben sie die umgekehrte Wirkung. **unpack** wird auch häufig zum Einlesen von Dateien mit fixen Satzlängen benützt. Für nähere Informationen siehe `perldoc -f unpack`

Globale und lokale Variablen:

Bisher haben wir immer globale Variablen verwendet. Es gibt jedoch in Perl wie auch in den meisten anderen Sprachen die Möglichkeit, lokale Variablen mit eingeschränktem Gültigkeitsbereich zu verwenden. Diese haben den Vorteil, daß man sich keine Gedanken zu machen braucht, was außerhalb eines Bereiches mit einer anderen Variable desselben Namens passiert.

In Perl gibt es das Konzept der Blöcke, z.B. **foreach (...) { BLOCK }** oder **while (...) { BLOCK }** oder auch, wie wir bald bei Subroutinen sehen werden, **sub Name { BLOCK }**. Man kann in so einen Block Variablen deklarieren, die nur innerhalb dieses Blockes gültig sind. Das Schlüsselwort, um eine solche Variable zu deklarieren, heißt **my**, z.B. **my $text;** Während der Deklaration kann man auch gleich einen Wert zuweisen und die Variable so initialisieren. Ohne diese Intitalisierung hat eine Variable den Wert **undef**.

Code: my_1.pl

```
01: #! /usr/bin/perl
02:
03: my $var = 5;
04:
05: print "Vor dem Block: $var\n";
06:
07: if ($var == 5) {
08:     my $var = 20; # ist andere Variable als oben
09:     print "Im Block: $var\n";
10: } # if
11:
12: print "Nach dem Block: $var\n";
```

```
F:\apacheweb\test_8085\html\codes>perl my_1.pl
Vor dem Block: 5
Im Block: 20
Nach dem Block: 5

F:\apacheweb\test_8085\html\codes>
```

Variablen (in Perl mit **my**) zu deklarieren ist fast jeder Programmiersprache ein sehr guter Stil, so auch in Perl, weil man die Programme in kleine logisch-vollständige Blöcke unterteilen kann, die von einander unabhängig sind. Man kann Perl sagen, es solle einen dazu zwingen, alle Variablen vor der (oder bei der ersten) Verwendung zu deklarieren, um so

schwierig zu findende Tippfehler einfacher unter Kontrolle bekommen.

Die Zähmung des Kamels:

wo liegt der Fehler bei folgendem Code?

Code: my_2.pl

```
1: #! /usr/bin/perl
2:
3: foreach my $variableOhneNamen (1..3) {
4:     print "$variable0hneNamen\n";
5: } # foreach
```

```
F:\apacheweb\test_8085\html\codes>perl my_2.pl

F:\apacheweb\test_8085\html\codes>
```

Wenn man die Compileroption strict verwendet, wird der Fehler schnell klarer:

Code: my_3.pl

```
1: #! /usr/bin/perl
2: use strict;
3:
4: foreach my $variableOhneNamen (1..3) {
5:     print "$variable0hneNamen\n";
6: } # foreach
```

Output:

```
F:\apacheweb\test_8085\html\codes>my_2.pl
Global symbol "$variable0hneNamen" requires explicit package name at F:\apachewe
b\test_8085\html\codes\my_2.pl line 5.
Execution of F:\apacheweb\test_8085\html\codes\my_2.pl aborted due to compilatio
n errors.
F:\apacheweb\test_8085\html\codes>85\html\codes>
```

Diese Meldung sagt, daß die Variable **$variable0hneNamen** in der Zeile 5 nicht mit my deklariert worden ist. Wir haben sie jedoch in der Zeile 4 deklariert und initialisiert. Also muß da ein Rechtschreibfehler vorliegen. Und wenn man genau hinsieht, bemerkt man, daß ich anstelle von dem Buchstaben O die Ziffer 0 geschrieben habe. Bei größeren Programmen kann die Suche nach einem solchen Fehler oft Stunden dauern, wenn man **use strict;** nicht verwendet.

Code: my_4.pl

```
1: #! /usr/bin/perl
2:
3: my $var = "abcdefg\n";
4: print "abcdefg", var, "\n";
```

Output:

```
F:\apacheweb\test_8085\html\codes>my_4.pl
abcdefgvar

F:\apacheweb\test_8085\html\codes>
```

Code: my_5.pl

```
1: #! /usr/bin/perl
2: use strict;
3:
4: my $var = "abcdefg\n";
5: print "abcdefg", var, "\n";
```

Output:

```
F:\apacheweb\test_8085\html\codes>my_5.pl
Bareword "var" not allowed while "strict subs" in use at F:\apacheweb\test_8085\
html\codes\my_5.pl line 5.
Execution of F:\apacheweb\test_8085\html\codes\my_5.pl aborted due to compilatio
n errors.

F:\apacheweb\test_8085\html\codes>
```

Dies sagt, daß in der Zeile 5 ein **$** vor **var** vergessen wurde.

use strict; achtet auch noch auf weitere Themen, die jedoch erst im Fortgeschrittenenkurs behandelt werden.

Es gibt noch eine Hilfe, die versucht, einen auf verdächtige Konstrukte hinzuweisen, z.B. **use warnings;** (ab Perl5.6):

Code: my_6.pl

```
1: #! /usr/bin/perl
2: use strict;
3: use warnings;
4:
5: my $var;
6: unless ($var) {
7:     my $var = 20;
8: } # unless
9: print "Variable: $var\n";
```

Output:

```
F:\apacheweb\test_8085\html\codes>my_6.pl
Use of uninitialized value in concatenation (.) or string at F:\apacheweb\test_8
085\html\codes\my_6.pl line 9.
Variable:

F:\apacheweb\test_8085\html\codes>
```

Perl ist, wie wir nach und nach kennenlernen werden, eine äußerst mächtige Programmiersprache. Sie versucht zu ahnen, was der Benutzer will, und versucht das dann zu tun. Diese Verhaltensweise ist jedoch oft gefährlich, weil man viele Fehler oft nur sehr schwer entdeckt.

Die beiden Ausdrücke **use strict;** und **use warnings;** helfen einem dabei, Perl so unter Kontrolle zu bekommen, daß es einem bei der Fehlersuche hilft und Perl zu einer vernünftigen, vollwertigen Programmiersprache macht. In verschiedenen Webforen wie z.B. http://www.perl-community.de/ oder http://www.perlmonks.org/ wird häufig Fragen zu Problemen gestellt, die der Programmierer mit der Verwendung von **strict** und **warnings** viel schneller selbst hätte lösen können. Und auch guten Programmierern passiert es gelegentlich, daß sie vor einem unerklärlichen Fehler stehen und dann um Hilfe suchen, nur weil sie mal schnell was ohne strict und warnings geschrieben haben. Aus diesem Grund verwende ich für jedes Programm, das länger als so 2-3 Zeilen ist, automatisch **strict** und **warnings**, und irgendwie habe ich fast nie unerklärliche Fehler in meinen Codes. Um die Verwendung von strict und warnings zu demonstrieren, werde ich ab sofort alle Codebeispiele mit **strict** und **warnings** angeben.

Die Variable $_ kann man allerdings nicht mit my deklarieren, weil es sich um eine eingebaute Perl-(Package-)Variable handelt. Wenn man Gültigkeit des Wertes von $_ auf einen Block beschränken will, muß man das altbackene local verwenden, was eine globale Variable mit demselben Namen durch eine lexikalische (= nur in diesem Bereich gültige) überdeckt. Die globale Variable wird dadurch nicht verändert; sie ist lediglich im aktuellen Bereich dann nicht mehr sichtbar.

```
while (defined( local($_) = <FH>)) {
  print $_;
} # while
```

Eine **while**-Schleife lokalisiert das $_ nicht automatisch, während eine **for**-Schleife dies schon macht.

Bei neueren Perl-Versionen reicht es auch zu schreiben:

```
while (local $_ = <FH>) {
  print;
} # while
```

Lexikalische Filehandles

Wo liegt das Problem beim folgenden Codeausschnitt?

Code: file_lex_01.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $file1 = "file_lex_01.pl";
06:
07: unless (open (FH, "<", $file1)) {
08:     die "Error: couldn't open file '$file1': $!\n";
09: } # unless
10:
11: while (my $line1 = <FH>) {
12:     chomp($line1);
13:     print "$file1: $.: $line1\n";
14:
15:     my $file2 = "file_write_2.pl";
16:     open (FH, "<", $file2) { or
17:         die "Error: couldn't open file '$file2': $!\n";
18:
19:     my $count = 0;
20:     while (my $line2 = <FH>) { $count++ }
21:
22:     close (FH);
23:     print "$file2 has $count lines\n";
24:
25: } # while
26:
27: close (FH);
```

- Was geschieht, wenn man diesen Code ausführt?
- Warum?
- Kann man das Problem beheben, indem man die Zeile 22: close(FH) auskommentiert?

- Wenn man einen Filehandle mit Namen FH öffnet, wird ein eventuell offener mit demselben Namen geschlossen.
- Bevor man einen Filehandle öffnet, muß man überprüfen, ob dieser Name überhaupt noch frei ist, und wenn nicht, einen anderen Namen wählen ("Klassische" Filehandles sind immer global.
- Es ist schwierig, solche Filehandles an Subroutinen zu übergeben, oder sie in Arrays oder Hashes zu speichern.

Perl bietet ab Version 5.6 die Möglichkeit, anstelle der "klassischen" Filehandles normale lexikalische Variablen für Filehandles zu verwenden, z.B.

Code: file_lex_02.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $file1 = "file_lex_01.pl";
06:
07: my $FH;
08: unless (open ($FH, "<", $file1)) {
09:     die "Error: couldn't open file '$file1': $!\n";
10: } # unless
11:
12: while (my $line1 = <$FH>) {
13:     chomp($line1);
14:     print "$file1: $.: $line1\n";
15:
16:     my $file2 = "file_write_2.pl";
17:     open (my $FH, "<", $file2) or
18:       die "Error: couldn't open file '$file2': $!\n";
19:
20:     my $count = 0;
21:     while (my $line2 = <$FH>) { $count++ }
22:
23:     close ($FH);
24:     print "$file2 has $count lines\n";
25:
26: } # while
27:
28: close ($FH);
```

Man beachte die my's in den Zeilen 07 und 17

Wichtige Punkte:

- bei einem open - or - die kann man das my direkt beim open schreiben (Zeile 17)
- bei einem if (open ...) {...} else { die ... } kann man ebenfalls das my direkt beim open schreiben, dann bleibt das $FH auf den if-Block beschränkt.
- bei einem unless funktioniert dies nicht, weil es auf den unless-Block beschränkt bleibt und im else-Zweig nicht vorhanden ist (Zeile 07)
- Lexikalische Filehandles werden am Ende ihres Gültigkeitsbereiches automatisch geschlossen. Ich empfehle aber, sie immer explizit mit close($FH) zu schließen, weil das den Hinweis gibt, daß der Filehandle bewusst und nicht unabsichtlich geschlossen wurde.
- Lexikalische Filehandles immer mit Grossbuchstaben verwenden, z.B. $FH, $MY_FILE, $FILEHANDLE, ...

Code: file_lex_03.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $file = "file_lex_01.pl";
06:
07: my @content1 = do {
08:     open (my $FH, "<", $file)
09:       or die "Error in opening file '$file': $!\n";
10:     my @lines = <$FH>;
11:     close ($FH);
12:     @lines;
13: };
14:
15: # oder kuerzer
16: my @content2 = do {
17:     open (my $FH, "<", $file)
18:       or die "Error in opening file '$file': $!\n";
```

```
19:     <$FH>;
20: };
21:
```

- Zeilen 07-13: hier wird das Öffnen und Auslesen der Datei in einen do-Block gekapselt. Ein do { .... }; - Block bietet genauso wie die meisten anderen Blöcke die Möglichkeit, darin lokale Variablen zu verwenden.
- Zeilen 12/19: die letzte Anweisung wird zurückgegeben
- Zeilen 16-20: da die lexikalische Variable $FH am Ende des Blockes ungültig wird (=out of Scope), kann man sich sogar das Schließen der Datei sparen, weil dies am Blockende automatisch geschieht. Ich empfehle jedoch, (gerade bei größeren Blöcken) immer die Datei explizit zu schließen, damit sich derjenige, der irgendwann mal diesen Code liest oder erweitert (vielleicht auch man selbst) , ob die Datei hier absichtlich geschlossen wird, oder ob es sich vielleicht einen Fehler handelt.
- Zeilen 13/20: Am Ende eines do-Blockes ist das Semikolon Pflicht!

Fazit: lexikalische Filehandles haben den Vorteil, daß sie nicht-global sind. Man muß sich also nicht darum kümmern, ob derselbe Handle noch woanders verwendet wird. Also (gerade bei größeren Programmen und Modulen) stehts lexikalische Filehandles verwenden, genauso wie strict und warnings..

Ausführen von externen Programmen:

In Perl gibt es mehrere Möglichkeiten, externe Kommandos auszuführen. Dabei muß man sich entscheiden, was man vom externen Programm wissen will.

Wenn man herausfinden will, ob ein externes Programm korrekt gelaufen ist (mit exitcode 0), aber der Output interessiert einen nicht, dann ist der Befehl **system()** angebracht:

Code: command_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: unless (system("ls -la") == 0) {
06:   die "Error: $?\n";
07: } # unless
08: else {
09:   print "All right\n";
10: } # else
```

Wenn man **system()** eine Zeichenkette mitgibt, wird (meistens) eine Shell geöffnet, die das Kommando interpretiert und dann ausführt. Wenn man dies vermeiden will, übergibt man system einfach eine Liste:

Code: command_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: unless (system("ls", "-la") == 0) {
06:   die "Error: $?\n";
07: } # unless
08: else {
09:   print "All right\n";
10: } # else
```

So wird die Shell nicht verwendet, und man kann überdies auch ein klein wenig Laufzeit sparen.

Wenn man an der Ausgabe eines Programmes interessiert ist, kann man sich zwischen den folgenden Möglichkeiten entscheiden:

1. **Backticks**: (sind auf der deutschen Tastatur rechts neben dem ?)

Code: command_3.pl

```
1: #! /usr/bin/perl
2: use warnings;
3: use strict;
4:
5: my @result = `dir`; # Rueckgabe als Liste, eine Zeile ist ein Element
6: print @result, "\n\n";
7:
8: my $result = `dir`; # der gesammte Output steht in $result
9: print $result;
```

Anstelle von Backticks kann man auch **my $result = qx(ls -l);** schreiben

2. **Pipe-open:**

Dies funktioniert im Grunde genauso wie das lesen von einer Datei:

Code: command_4.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $command = 'ls -la';
06: unless(open (CMD, "$command |")) {
07:   die "Error in executing '$command': $!\n";
08: } # unless
09: while (<CMD>) {
10:   chomp($_);
11:   print "Zeile: $_\n";
12: } # while
13: close (CMD) or die "Error in closing '$command': $!\n";;
```

Ebenso kann man Daten an ein anderes Programm pipen (vorausgesetzt, das Programm versteht die Daten per Pipe):

Code: command_5.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $email = 'irgendwas@local';
06: my $sendmail = '/usr/bin/sendmail -t';
07:
08: my $SENDMAIL;
09: unless (open ($SENDMAIL, "| $sendmail")) {
10:     die "Error: couldn't open '$sendmail': $!\n";
11: } # unless
12:
13: print $SENDMAIL "To: irgendwem <$email>\n"  or die "Error: ... $!";
14: print $SENDMAIL "From: sonstwem <$email>\n" or die "Error: ... $!";
15: print $SENDMAIL "Subject: Testmail\n"       or die "Error: ... $!";
16:
17: print $SENDMAIL "Email-Text\n"              or die "Error: ... $!";
18:
19: close ($SENDMAIL) or die "Error in closing '$sendmail': $!\n";
20:
```

- Gerade beim Schreiben ist es wichtig , immer auf eventuell aufgetretene Fehler zu überprüfen, und zwar sowohl beim open als auch beim close. Aber auch beim print, damit man auftretende Fehler so schnell wie möglich mitbekommt.
- Häufig löst man die Fehlerkontrolle beim print (oder ähnlichem) durch den Einsatz von Subroutinen, die wir später behandeln werden.

Es gibt noch weitere Möglichkeiten, externe Programme auszuführen und fernzusteuern; da diese jedoch alle Module benötigen, behandle ich sie hier nicht.

Bei der Ausführung externer Kommandos lege ich sehr viel Wert die Fehlerabfrage, weil sie mir schon Stunden von Fehlersuche erspart hat.

Datenstrukturen:

Nochmal zurück zum Beispiel hello_5.pl:

Code: hello_5.pl

```
01: #! /usr/bin/perl
02:
03: # Konfiguration
04: $prompt = "Sag was (mit <Enter> absenden): ";
05: $answer = "Du sagtest:";
06:
07: print $prompt;
08: chomp( $input = <STDIN> );
09:
10: while( $input ne "ende" ) {
11:     print "$answer $input\n";
12:
13:     print $prompt;
14:     chomp( $input = <STDIN> );
15:
16: } # while
17:
18: print "Ciao!\n";
```

Wie kann man diesen Zutrittsmechanismus so verfeinern, daß man für jede Person einen eigenen Benutzernamen und ein eigenes Passwort vergeben kann?

1. Die Namen und Benutzernamen in eine Liste packen, und zwar jeweils mit einem Trennzeichen getrennt (z.B. #)

```
my @accounts = ("dumbledore#geheim", "ron#weasly", "harry#potter", "neville#");
```

und dann $name und $wort eingeben lassen, und dann über die Liste @accounts iterieren:

```
for my $combination (@accounts) {
         my ($sName, $sPassword) = split(/#/, $combination, 2);
         if ($name eq $sName and $wort eq $sPassword) {
        print "ok\n";
         } # if
} # for
```

oder den umgekehrten Weg wählen:

```
for my $combination (@accounts) {
    my $test = join("#", $name, $wort);
          if ($combination eq $test) {
        print "ok\n";
         } # if
} # for
```

Diese Vorgehensweisen sind gut, aber es gibt noch bessere, indem man eine andere Datenstuktur verwendet. Wir arbeiten mit einer Liste, und auf ein Element einer Liste kann man zwar mit einem Index zugreifen, jedoch muss man dafür den Index kennen. Sonst muss man, wie in unserem Beispiel, die komplette Liste durchsuchen, was gerade bei größeren Listen manchmal doch unnötig lange braucht. Ein Passwort "gehört" zu einer Person. Also wäre die Idealvorstellung, wenn wir ein Array verwenden könnten, dessen Index der Name und dessen Wert das Passwort ist. In Perl kann der Index eines Arrays jedoch lediglich eine Integerzahl sein. Larry-sei-Dank gibt es für solche recht häufig vorkommenden Fälle eine eigene Datenstruktur:

Hashes:

Hashes (oft auch als assoziative Arrays bezeichnet, oder in anderen Sprachen wie Tcl als Array oder in VB Dictionary) sind eine Datenstruktur wie ein Array, nur daß auf einzelne Elemente über eine Zeichenkette als Index zugegriffen werden kann anstelle von einer Zahl. Den Index nennt man Schlüssel (oder englisch: key) und den Wert einfach Wert (oder englisch: value). Man kann einfach von einem Key auf einen Value zugreifen, umgekehrt wird es aber komplizierter. Genau wie bei einer Liste kann ein Hashkey auf den Inhalt einer skalaren Variable zeigen (derzeit Zeichenketten oder Zahlen; im Fortgeschrittenenkurs werden auch Referenzen auf andere Daten(strukturen) behandelt).

Beispiel Array:	Array	Hash	Beispiel Hash:
$array[3] = 20;	Index ist Zahl	Index ist String	$hash{'zwanzig'} = 20;
[ ]	Eckige Klammern	Geschwungene Klammern	{ }
@liste	@ für ganze Liste	% für ganzen Hash	%hash

@array

Index:	Wert:
0	'abcde'
1	42
2	3.1415926354
3	$skalareVariable
4	$otherArray[2]
5	$hash{'Antwort auf alle Fragen'}

%hash

Index:	Wert:
'code'	'abcde'
'Antwort auf alle Fragen'	42
'PI'	3.1415926354
'irgendwas'	$skalareVariable
'arrayWert'	$array[2]
'hashwert'	$otherHash{'code'}

Wenn man einem Element eines Arrays oder Hashes ein Element eines anderen Arrays oder Hashes zuweist, wird der Wert kopiert. Die beiden Container werden dadurch nicht in irgendeiner Weise verbunden. Dies könnte man mit Referenzen machen, wie wir später sehen werden.

Bei einem Hash kann man auf die Anführungszeichen beim Hashkey verzichten, wenn er keine Leerzeichen enthält, nicht mit einem mathematischen (+, -) oder anderswertig reserviertem Zeichen ($, @) beginnt, z.B. $hash{abcd}, aber $hash{'mit Leerzeichen'}

Code: hash_01.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %passwords = (
06:                  "dumbledore" => 'geheim',
07:                  'ron'        => "weasly",
08:                  "harry"      => "potter",
09:                  "neville"    => '',
10:                  );
11:
12: print "Usernamen eingeben: ";
13: my $username = <STDIN>; chomp($username);
14:
15: print "Passwort eingeben: ";
16: my $password = <STDIN>; chomp($password);
```

```
17:
18: if (exists $passwords{$username} ) {
19:
20:     if ( $password eq $passwords{$username} ) {
21:         print "Passwort korrekt. Herzlich willkommen, $username\n";
22:     } # if
23:
24:     else {
25:         print "Passwort $passwords{$username} ist nicht korrekt\n";
26:     } # else
27:
28: } # if
29:
30: else {
31:     print "Kenne niemanden mit dem Usernamen $username\n";
32: } # else
33:
```

- Zeile 5-9: der Hash wird deklariert und gleich mit Keys und Values versehen
- Zeile 6-9: der => Operator ist eigentlich ein normales Komma, er wird jedoch häufig verwendet, weil er die Zusammengehörigkeit von Value zu Key besser veranschaulicht und den Wert davor "stringifiziert", das heißt, man kann die Anführungszeichen beim Key weglassen.
- Zeile 17: **exists $hash{$key}** gibt wahr zurück, wenn der Schlüssel existiert, sonst falsch. Damit überprüfen wir, ob ein Benutzername überhaupt existiert.

Initialisierung eines Hashes:

Code: hash_02.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %hash1 = (
06:               "dumbledore" => 'geheim',
07:               'ron'        => "weasly",
08:               "harry"      => "potter",
09:               "neville"    => '',
10:               );
11:
12: my %hash2 = (
13:               "dumbledore", 'geheim',
14:               'ron'       , "weasly",
15:               "harry"     , "potter",
16:               "neville"   , '',
17:               );
18:
19: my %hash3 = qw(
20:               dumbledore geheim
21:               ron        weasly
22:               harry      potter
23:               neville    vergessen
24:               );
25:
26: my %hash4 = ();
27: $hash4{dumbledore} = 'geheim';
28: $hash4{ron} = 'weasly';
29: $hash4{harry} = 'potter';
30: $hash4{neville} = '';
```

Über einen Hash iterieren:

1. for(each)-Schleife

```
foreach my $key (keys %hash) {
```

```
   print "$key: $hash{$key}\n";
} # foreach
```

keys(%hash) ist eine Funktion, die die ganzen Schlüssel eines Hashes in einer Liste zurückgibt. Analog dazu gibt es noch **values(%hash)**, das die ganzen Werte zurückgibt. values wird aber sehr selten verwendet.

2. while-Schleife

```
while (my ($key, $value) = each %hash) {
   print "$key: $value\n";
} # while
```

Wobei da meistens die foreach-Schleife verwendet wird, weil man da noch einfach weitere Angaben machen kann, wie z.B. Sortierungen:

```
foreach my $key (sort( keys %hash)) {
    print "$key => $hash{$key}\n";
} # foreach
```

sort nimmt eine Liste entgegen (von keys), sortiert die dann Asciibetisch und gibt die sortierte Liste wieder zurück. Dazu später genaueres.

- Bei Hashes ist der Key stehts eindeutig und kann nur einen Wert aufnehmen. Wenn einem Hash ein Key/Value-Paar zugewiesen wird, dessen Key schon existiert, wird der vorherige Value einfach überschrieben.
- Mit **exists($hash{$element})** kann man überprüfen, ob ein Key schon existiert.
- Mit **defined($hash{$element})** kann man überprüfen, ob für einen Key schon ein Value existiert. Ein Key in einem Hash kann ohne Probleme den Wert **undef** haben, genauso wie ein Array (dies würde man dann "Array mit Löchern" nennen).
- Mit **keys(%hash)** bekommt man eine Liste von Hashkeys zurück.
- Mit **values(%hash)** bekommt man eine Liste von Werten zurück.
- Die Reihenfolge, in der die Keys und Values zurückgeliefert werden, ist nicht vorhersagbar; festgelegt ist lediglich, dass bei keys und bei values dieselbe Reihenfolge der Elemente zurückgegeben wird. Diese Reihenfolge kann sich jedoch von Perl-Version zu Perl-Version ändern, also sollte man sich nicht auf eine fixe Reihenfolge verlassen.
- Ein Hash ist in der Regel langsamer als ein Array, was jedoch häufig durch die einfachere Zugriffsweise auf einzelne Werte wieder wettgemacht wird.
- Der Hash ist eine der wichtigsten Datenstrukturen in Perl und häufig die Basis von komplexeren Datenstrukturen und Objekten.
- Hashes sind häufig etwas komplizierter zu verstehen, weil sie eine andere algorithmische Denkweise als Arrays erfordern.
- Ein Element eines Hashes kann man mit dem Befehl **delete** löschen, z.B. **delete $hash{$element};**

Viele Perl-Neulinge versuchen, alle möglichen Probleme mit Arrays zu lösen, weil ihnen diese Vorgehensweise aus vielen anderen Programmiersprachen bekannt ist. Oft ist jedoch die Verwendung von Hashes einfacher und sorgt für besser lesbaren (und somit wartbaren) Code. Das folgende Beispiel würde ich z.B. nur ungern mit Arrays coden:

Wörter zählen mit Hashes:

Wir haben in einer Textdatei (wordlist.txt) alle Wörter dieses Vortrages, die länger als 8 Zeichen sind, und zwar ein Wort pro Zeile. Wie können wir herausfinden, welche Wörter öfter als 9 Mal vorkommen? Mit Listen wäre dies ein recht aufwendiges Unterfangen, mit einem Hash ist dies jedoch recht einfach lösbar:

Code: hash_04.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $textfile = 'wordlist.txt';
06:
07: unless (open (WORDS, $textfile)) {
08:     die "Error: can't open '$textfile': $!\n";
09: } # unless
```

```
10:
11: my %wordsCount = ();
12:
13: while (defined (my $word = <WORDS>)) {
14:     chomp($word);
15:
16:     if (exists $wordsCount{$word}) {
17:         $wordsCount{$word} ++;
18:     } # if
19:     else {
20:         $wordsCount{$word} = 1;
21:     } # else
22:
23: } # while
24:
25: close(WORDS);
26:
27: foreach my $key (sort keys %wordsCount) {
28:     if ( $wordsCount{$key} >= 9 ) {
29:         printf "%-30s => %4d\n", $key => $wordsCount{$key};
30:     } # if
31: } # foreach
```

- Zeile 7-9: Datei öffnen mit Fehlerabfrage
- Zeile 13: Zeile einlesen und $word zuweisen, $word auch gleich deklarieren, und überprüfen, ob der Wert auch definiert ist (solche zusammengezogenen Schreibweisen kommen in Perl häufig vor und sind meistens von rechts nach links zu lesen, wenn man da die Klammern weglässt.
- Zeile 16: Wenn ein Wort schon als Hashkey eingetragen ist, dann erhöhe den Value dieses Keys um eins (Zeile 17), sonst erzeuge diesen Key und initialisiere dessen Wert mit **1** (Zeile 20).
- Zeile 27: iteriere über die Keys des Hashes, überprüfe, ob dieses Wort (=Key) öfter als 9 Mal vorgekommen ist (Zeile 28), und wenn ja, dann gib das Word und die Anzahl formatiert aus (Zeile 29).

Code: hash_05.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $textfile = 'wordlist.txt';
06:
07: unless (open (WORDS, $textfile)) {
08:     die "Error: can't open '$textfile': $!\n";
09: } # unless
10:
11: my %wordsCount = ();
12:
13: while (defined (my $word = <WORDS>)) {
14:     chomp($word);
15:     $wordsCount{$word} ++;
16: } # while
17:
18: close (WORDS);
19:
20: foreach my $key (sort keys %wordsCount) {
21:     printf("%-30s => %4d\n", $key => $wordsCount{$key})
22:         if $wordsCount{$key} >= 9;
23: } # foreach
```

- Das zweite Beispiel ist eine Kurzschreibweise des ersten; die Zeilen 16-21 wurden durch die Zeile 15 ersetzt, da der ++-Operator, der ja den aktuellen Wert um eins erhöht, auch aus **undef 1** macht.
- In den Zeilen 21/22 wurde das **if** an das Ende gestellt: **printf (....) if Bedingung**; So kann man gut gewichten, was wichtiger ist: das printf oder die Überprüfung der Anzahl. Bei dem nachgestellten **if** kann man auf die Klammern um die Bedingung herum verzichten. Bei dem nachgestellten **if** kann man jedoch kein **elsif** oder **else** verwenden.

Sortierungen:

Bisher haben wir die Funktion **sort()** immer benützt, wenn wir was asciibetisch sortiert haben wollten. Manchmal will man jedoch, daß unabhängig von der Groß- und Kleinschreibung sortiert wird, oder numerisch (10 ist numerisch gesehen größer als 2). Dafür kann man der Funktion sort einen Block mitgeben, der einen Vergleichsalgorithmus implementiert, der dann von **sort** verwendet wird. Die beiden Elemente, die da verglichen werden sollen, stehen in den globalen Variablen $a und $b (deshalb sollte man die auch sonst im Programm nicht verwenden, weil sie nämlich von der Überprüfung von **use strict;** ausgenommen sind).

Asciibetisch bedeutet, nach den ASCII-Werten von Zeichen sortieren, z.B. A => 65, Z => 90, a => 97, z => 122

Bisher haben wir sort immer ohne diesen Block benützt, was der folgenden Schreibweise entsprechen würde:

```
@listeSortiert = sort { $a cmp $b } @liste;
```

cmp vergleicht zwei Zeichenketten (**$a** und **$b**) nach ASCII-Zeichen, und liefert **-1** zurück, wenn **$a** kleiner ist; **0** wenn beide gleich groß sind und **+1**, wenn **$b** kleiner ist. Sort wertet diese Ergebnisse (negativ, 0, positiv) aus und gruppiert diese Elemente in **@listeSortiert** dementsprechend neu.

Wenn wir jetzt so sortieren wollen, daß Gross-/Kleinschreibung keine Rolle spielt, könnten wir folgenderweise schreiben:

```
@listeSortiert = sort { lc($a) cmp lc($b) } @liste;
```

Dadurch würden für den Vergleich beide Kriterien in Kleinschreibung umgewandelt. Natürlich kann man auch in Großschreibung umwandeln, was das identische Ergebnis liefert:

```
@listeSortiert = sort { uc($a) cmp uc($b) } @liste;
```

Wenn man nun Zahlen numerisch sortieren will (1,2,10,20,40), verwendet man anstelle des Zeichenkettenvergleichs **cmp** einfach einen numerischen Vergleich <=> :

```
@listeSortiert = sort { $a <=> $b } @liste;
```

Diese Sortierungen waren bisher immer aufsteigend. Wenn man jedoch absteigend sortieren will, kann man entweder noch ein **reverse** einbauen:

```
@listeAbsteigendSortiert = reverse sort { $a <=> $b } @liste;
```

oder besser noch $a und $b vertauschen (ist schneller):

```
@listeAbsteigendSortiert = sort { $b <=> $a } @liste;
```

Wenn man z.B. die Keys eines Hash in einer for-Schleife nach den Werten asciibetisch sortiert haben will, könnte man schreiben:

```
foreach my $key (sort { $hash{$a} cmp $hash{$b} } keys %hash) { .... }
```

Wie könnten wir im letzten Beispiel das häufigste Wort als erstes, das zweithäufigste als zweites usw. ausgeben?

Subroutinen:

In Perl kann man ebenso wie in den meisten anderen Programmiersprachen Codeteile zusammenfassen und in Subroutinen (=Prozeduren oder Funktionen) verlagern, die man dann von außerhalb und innerhalb einer Funktion aufrufen kann. Eine Subroutine ist ein Block, in dem man auch lokale Variablen deklarieren kann. So kann man eine Subroutine bis auf die Parameterübergabe völlig von außen abschotten. Die Verwendung von Subroutinen erspart oft eine Menge Code, weil man die ja an mehreren Stellen eines Programmes aufrufen kann, und verbessert in der Regel die Strukturierung und somit die Lesbarkeit eines Programmes. Und es erleichtert auch die Wiederverwendbarkeit von Code in anderen Programmen.

Eine Subroutine beginnt immer mit dem Schlüsselwort sub, gefolgt von dem Namen der Subroutine und danach folgt der Block, der den Code enthält:

```
sub NameDerSubroutine {
  # code
}
```

Folgende Möglichkeiten gibt es, eine Subroutine aufzurufen:

- &NameDerRoutine()
- NameDerRoutine()
- NameDerRoutine
- &NameDerRoutine

Ich empfehle, die erste oder zweite Möglichkeit zu verwenden. Ich selbst verwende für eigene Subroutinen immer die erste, weil durch das & vor dem Namen gleich klar wird, daß es sich um eine nicht in Perl eingebaute Subroutine handelt, und weil man auch vor Variablen immer $ @ oder % schreiben muß. Die erste Möglichkeit umgeht nebenbei auch Prototypen. Die dritte Möglichkeit funktioniert nur dann, wenn die Subroutine vorher schon deklariert worden ist. Die vierte Möglichkeit übergibt eine eventuell vorhandene Parameterliste der Funktion an NameDerRoutine weiter, wodurch man sich schwierig zu findende Fehler einhandeln kann.

Code: sub_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: &PrintTrennstrich();
06: print "Hallo, Welt\n";
07: &PrintTrennstrich();
08:
09: sub PrintTrennstrich {
10:     print "-" x 60, "\n";
11: } # PrintTrennstrich
```

- Zeile 9-11: Subroutine **PrintTrennstrich** wird erzeugt
- Zeile 10: der **x**-Operator "multipliziert" die Zeichenkette, die vorher steht, durch den Wert, der dahinter steht; in diesem Fall wird also ein Strich sechzig Mal aufgegeben
- Zeile 5 und 7: Die Subroutine **PrintTrennstrich** wird aufgerufen

Output:

```
F:\apacheweb\test_8085\html\codes>sub_1.pl
------------------------------------------------------------
Hallo, Welt
------------------------------------------------------------

F:\apacheweb\test_8085\html\codes>
```

Mit der Funktion return(....) kann man Werte aus der Subroutine als Liste ans Hauptprogramm zurückgeben, z.B.

Code: sub_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: print &GetTrennstrich();
06: print "Hallo, Welt\n";
07: print &GetTrennstrich();
08:
```

```
09: sub GetTrennstrich {
10:     my $string = "-" x 60;
11:
12:     return ($string . "\n");
13: } # GetTrennstrich
```

Bei der Rückgabe von Listen und Hashes muß man darauf achten, daß die bei return in eine Liste umgewandelt werden und man somit eventuell die Listen nicht mehr eindeutig aus der Übergabeliste extrahieren kann:

```
sub Subroutine {
    my @list1 = (A..F);
    my @list2 = (G..M);
    return (@liste1, @liste2);
} # Subroutine
my (@liste1, @liste2) = &Subroutine();
```

Hier stehen in @liste1 alle Werte A-M und @liste2 ist leer. Genauso beim Hash, der da on-the-Fly in ein Array umgewandelt wird.

Referenzen:

Um dieses Problem lösen zu können, muß man sich entweder selbst um die Verwaltung der Rückgabeparameterliste kümmern, oder man verwendet Referenzen. Eine Referenz ist ein neuer Variablentyp. Es handelt sich dabei um einen Verweis auf eine andere Variable (d.h. eigentlich auf den Speicherbereich einer Variable), genauso wie in Java und ähnlich wie ein Pointer in C/C++. Eine Referenz ist auch eine skalare Variable, d.h. alle Namensregeln einer skalaren Variablen treten dort in Kraft:

- der Name einer Referenz beginnt stets mit einem $
- danach muß ein Buchstabe oder Unterstrich folgen
- danach null oder mehr Buchstaben, Unterstriche oder Ziffern
- Umlaute wie ä, ö, Ü sind nicht erlaubt

Array-Referenzen:

- Eine Referenz auf eine Variable wird folgendermaßen erzeugt: \@array
- Der Inhalt einer Arrayreferenz kann folgendermaßen dereferenziert werden: @{ $arrayRef }.
- In Perl kann man die geschwungenen Klammern häufig weglassen: @$arrayRef
- Auf ein Element einer Arrayreferenz kann man folgendermaßen zugreifen: $arrayRef->[$index] (oder die alte Schreibweise: ${$arrayRef}[$index] oder $$arrayRef[$index]; das $ ersetzt in diesem Fall das @, weil in Perl immer das Zeichen für den Typ gesetzt wird, der herauskommt. Und da auch ein Element eines Arrays ein Skalar ist, muß man da ${ }[$index] schreiben).

Code: ref_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my @array = (0..9);
06: print "Liste: ", join(" ", @array), "\n";
07:
08: my $arrayRef = \@array;
09: print "ListenReferenz: ", join(" ", @{ $arrayRef } ), "\n\n";
10:
11: $array[1] = 10; # das Array aendern
12: print "Aenderung1: ", join(" ", @array), "\n";
13: print "Aenderung1: ", join(" ", @$arrayRef), "\n\n";
14:
15: $arrayRef->[3] = 29; # einen Wert der Daten hinter der Referenz aendern
16: print "Aenderung2: ", join(" ", @array), "\n";
17: print "Aenderung2: ", join(" ", @$arrayRef), "\n";
18:
```

Output:

```
F:\apacheweb\test_8085\html\codes>ref_1.pl
Liste: 0 1 2 3 4 5 6 7 8 9
ListenReferenz: 0 1 2 3 4 5 6 7 8 9

Aenderung1: 0 10 2 3 4 5 6 7 8 9
Aenderung1: 0 10 2 3 4 5 6 7 8 9

Aenderung2: 0 10 2 29 4 5 6 7 8 9
Aenderung2: 0 10 2 29 4 5 6 7 8 9

F:\apacheweb\test_8085\html\codes>
```

Hash-Referenzen:

- Eine Referenz auf eine Variable wird folgendermaßen erzeugt: \%hash
- Der Inhalt einer Hashreferenz kann folgendermaßen dereferenziert werden: %{ $hashRef }.
- In Perl kann man die geschwungenen Klammern häufig weglassen: %$hashRef
- Auf ein Element einer Hashreferenz kann man folgendermaßen zugreifen: $hashRef->{$key} (oder die alte Schreibweise: ${$hashRef}{$key} oder $$hashRef{$index}; das $ ersetzt in diesem Fall das %, weil in Perl immer das Zeichen für den Typ gesetzt wird, der herauskommt. Und da auch ein Wert eines Hashes ein Skalar ist, muß man da ${ }{$key} schreiben).

Code: ref_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %hash = ( key1 => 'value1',
06:              key2 => 'value2',
07:              key3 => 'value3',
08:              );
09:
10: print "Hash: ";
11: foreach my $key (keys %hash) {    print "$key => $hash{$key}  "; }
12: print "\n";
13:
14: my $hashRef = \%hash;
15: print "HashRef: ";
16: foreach my $key (keys %$hashRef) {    print "$key => $hashRef->{$key}  "; }
17: print "\n";
18:
19: $hashRef->{key4} = 'value4'; # neues key/value-Paar anlegen
20: print "Aenderung1: ";
21: foreach my $key (keys %hash) {    print "$key => $hash{$key}  "; }
22: print "\n";
23: print "Aenderung1: ";
24: foreach my $key (keys %$hashRef) {    print "$key => $hashRef->{$key}  "; }
25: print "\n";
26:
27: $hash{key1} = 'perl'; # Wert aendern
28: print "Aenderung2: ";
29: foreach my $key (keys %hash) {    print "$key => $hash{$key}  "; }
30: print "\n";
31: print "Aenderung2: ";
32: foreach my $key (keys %$hashRef) {    print "$key => $hashRef->{$key}  "; }
33: print "\n";
```

Output:

```
F:\apacheweb\test_8085\html\codes>ref_2.pl
Hash: key2 => value2  key1 => value1  key3 => value3
HashRef: key2 => value2  key1 => value1  key3 => value3
```

```
Aenderung1: key2 => value2  key4 => value4  key1 => value1  key3 => value3
Aenderung1: key2 => value2  key4 => value4  key1 => value1  key3 => value3
Aenderung2: key2 => value2  key4 => value4  key1 => perl  key3 => value3
Aenderung2: key2 => value2  key4 => value4  key1 => perl  key3 => value3

F:\apacheweb\test_8085\html\codes>
```

Es gibt noch weitere Referenzen:

- Referenzen auf Zahlen oder Zeichenketten (da ist das Dereferenzierungszeichen $): treten fast nie auf
- Referenzen auf Subroutinen (Dereferenzierungszeichen &): werden im Fortgeschrittenenkurs behandelt
- Referenzen auf Reguläre Ausdrücke: kommen sehr selten in der objektorientierten Programmierung vor
- GLOBs: (Erkennungszeichen: *): sind ein Überbleibsel aus dem alten Perl4, als es noch keine Referenzen gab und kommen heutzutage höchstens noch bei sehr fortgeschrittenen Handles (z.B. tief verborgen in der Netzwerksocketprogrammierung) vor

Um zu erkennen, von welchem Typ eine Referenz ist, kann man die Funktion ref($xRef) verwenden.

```
print ref( $referenz );
```

Die Ergebnisse können sein: ARRAY, HASH, SCALAR, CODE, REF, ...

Komplexere Rückgabewerte aus Subroutinen:

Code: sub_3.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my ($value1, $arrayRef, $hashRef) = &GetData();
06: my @array = @$arrayRef;
07: my %hash = %$hashRef;
08:
09: # ------------------------------------------------------------
10: sub GetData {
11:     my $string = "abcde";
12:     my @array = (1..10);
13:     my %hash = ( a => 1, b => 2, c => 3 );
14:
15:     return ($string, \@array, \%hash);
16: } # GetData
```

- Zeile 15: der String wird 1:1 der Rückgabeliste hinzugefügt, das Array und der Hash jeweils als Referenz
- Zeile 5: der String, die Arrayreferenz und die Hashreferenz werden aus der Rückgabeliste ausgelesen
- Zeile 6: Dereferenzierung der Arrayreferenz
- Zeile 7: Dereferenzierung der Hashreferenz

Parameterübergabe an Subroutine:

Code: sub_4.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: # Ueberschrift ausgeben
06: printf ("%4s | %-8s\n", 'Zahl', 'Wurzel');
07: print '-' x 15, "\n";
08:
09: # Werte ausgeben
10: foreach my $i (1..10) {
```

```
11:     my $wurzel = &BerechneWurzel($i);
12:     printf "%4i | %2.6f\n", $i, $wurzel;
13: } # foreach
14:
15: # -------------------------------------------------------------
16: sub BerechneWurzel {
17:     my ($zahl) = @_;
18:
19:     my $wurzel = sqrt($zahl);
20:
21:     return $wurzel;
22: } # GetData
```

- Zeile 11: Die Subroutine BerechneWurzel wird aufgerufen und ihr der Parameter $i übergeben
- Zeile 17: Der Parameter $zahl (=$i) wird aus der Parameterliste @_ ausgelesen.
- Zeile 19: Die Quadratwurzel von $zahl wird berechnet
- Zeile 20: und wieder zurückgegeben und in Zeile 11 an die Variable $wurzel übergeben
- Zeile 12: $i und $wurzel werden formatiert ausgegeben

Hier nochmal das Beispiel mit den Hashreferenzen, diesmal aber mit Subroutinen.

Code: sub_5.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %hash = ( key1 => 'value1',
06:              key2 => 'value2',
07:              key3 => 'value3',
08:              );
09: &PrintHash('Anfang', %hash);
10:
11: my $hashRef = \%hash;
12: &PrintHashRef('Anfang', $hashRef);
13:
14: $hashRef->{key4} = 'value4'; # neues key/value-Paar anlegen
15: &PrintHash('Aenderung1', %hash);
16: &PrintHashRef('Aenderung1', $hashRef);
17:
18: $hash{key1} = 'perl'; # Wert aendern
19: &PrintHash('Aenderung2', %hash);
20: &PrintHashRef('Aenderung2', $hashRef);
21:
22: # -------------------------------------------------------------
23: sub PrintHash {
24:     my ($prefix, %data) = @_;
25:
26:     print "HASH: $prefix: ";
27:     foreach my $key (keys %data) {
28:         print "$key = $data{$key}  ";
29:     } # foreach
30:     print "\n";
31: } # PrintHash
32: # -------------------------------------------------------------
33: sub PrintHashRef {
34:     my ($prefix, $data) = @_;
35:
36:     print "HRef: $prefix: ";
37:     foreach my $key (keys %$data) {
38:         print "$key = $data->{$key}  ";
39:     } # foreach
40:     print "\n";
41: } # PrintHashRef
```

Wenn man nur das erste Element aus der Übergabeparameterliste braucht:

```
my $param1 = shift(@_); # lange Schreibweise

my $param1 = shift(); # Abkürzung

my $param1 = shift; # ganz kurz
```

sonst müßte man schreiben: my ($param1) = @_; weil ein Array im skalaren Kontext sonst die Anzahl seiner Elemente zurückgibt (vergleiche $anzahl = scalar(@liste);

Reguläre Ausdrücke:

Die Suche von Zeichenketten mit den Funktionen (r)index oder eq ist nicht besonders mächtig, dafür aber sehr umständlich. Und Perl wurde ja dazu entwickelt, damit man einfache Probleme einfach lösen kann, und auch komplexe Probleme lösbar zu machen. Und deshalb gibt es da viel mächtigere Mittel, die man reguläre Ausdrücke nennt. Diese Bezeichnung kommt aus der Mathematik und haben in viele Programmiersprachen, Editoren und Kommandozeilenwerkzeuge Einzug gehalten, z.B. Elisp, Python, Tcl, emacs, sed, grep, find, usw. Leider gibt es da auch verschiedene Dialekte, die zwar alle auf einer Basis aufbauen, aber doch manchmal unterschiedliche Prägungen haben. Die in Perl verwendeten regulären Ausdrücke sind wahrscheinlich die mächtigsten.

Reguläre Ausdrücke sind äußerst mächtige Hilfsmittel, die jedoch leider auch ziemlich kompliziert sind.

Bei einem regulären Ausdruck handelt es sich um eine Art Grammatik, die auf eine Zeichenkette passt (man sagt auch: matcht) oder nicht. Sie sind so was ähnliches wie die Platzhalter * und ? in der Dos-Kommandozeile, z.B. werden mit **dir *.txt** im aktuellen Verzeichnis alle sichtbaren Dateien mit der Endung .txt angezeigt. Oder mit **dir *.do?** werden alle sichtbaren Dateien mit den Endungen doa, .dob, .doc, ..., .dot, ...do5 angezeigt. Mit * oder ? wird eine kleine Grammatik erzeugt, die da bestimmte Dateinamen erfasst und andere nicht.

Zurück zum Beispiel der Passwortabfrage (Hash):

Code: hash_01.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %passwords = (
06:                  "dumbledore" => 'geheim',
07:                  'ron'        => "weasly",
08:                  "harry"      => "potter",
09:                  "neville"    => '',
10:                  );
11:
12: print "Usernamen eingeben: ";
13: my $username = <STDIN>; chomp($username);
14:
15: print "Passwort eingeben: ";
16: my $password = <STDIN>; chomp($password);
17:
18: if (exists $passwords{$username} ) {
19:
20:     if ( $password eq $passwords{$username} ) {
21:         print "Passwort korrekt. Herzlich willkommen, $username\n";
22:     } # if
23:
24:     else {
25:         print "Passwort $passwords{$username} ist nicht korrekt\n";
26:     } # else
27:
28: } # if
29:
30: else {
31:     print "Kenne niemanden mit dem Usernamen $username\n";
```

```
32: } # else
33:
```

Bei diesem Beispiel mußte man harry eingeben; Harry wurde z.B. nicht erkannt. Damit da Groß- und Kleinschreibung ignoriert wird, könnte man mit einem regulären Ausdruck folgendermaßen schreiben:

Code: regex_01.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %passwords = (
06:                  "dumbledore" => 'geheim',
07:                  'ron'        => "weasly",
08:                  "harry"      => "potter",
09:                  "neville"    => '',
10:                  );
11:
12: print "Usernamen eingeben: ";
13: chomp(my $username = <STDIN>);
14:
15: foreach my $key (keys %passwords) {
16:
17:     if ($username =~ m/$key/i) {
18:         print "Der Username '$key' wurde gefunden\n";
19:
20:     } # if
21: } # foreach
22:
```

- Zeile 13 ist wieder eine Zusammenziehung von mehreren Kommandos

Zeile 17: ein regulärer Ausdruck hat immer die folgende Form:

$zeichenkette =~ m/Muster/Optionen;

$zeichenkette ist der String, in dem gesucht wird

=~ ist der Operator; auf Deutsch sagt man dafür: passt auf, matcht auf; in Englisch: matches, is like

m/.../ umrahmt den Ausdruck. Anstelle von / können auch andere Trennzeichen verwendet werden, z.B.

m(...), m|...|, m#...#, m~...~, m{...}

Wenn das Trennzeichen jedoch / ist, kann man das m auch weglassen: /.../

Am Ende kann man noch Optionen angeben, die das Suchverhalten beeinflussen. In diesem Beispiel habe ich m/.../i verwendet, was die Suche von Gross- und Kleinschreibung unabhängig durchführt.

Unser Vergleich in dem letzten Beispiel hat jedoch eine Schwäche: was passiert, wenn da als Username z.B. **ron**ald oder cha**ron** angegeben wird?

Ankerwerfen:

Dieses Problem kann man durch die Verwendung von Ankern lösen:

^ ist ein Anker, der am Anfang der Zeichenkette passt

$ ist ein Anker, der am Ende der Zeichenkette passt.

Mit diesen Hilfsmitteln können wir die Regex (=Kurzausdruck für regulären Ausdruck) sicherer machen:

```
if ($username =~ m/^$key$/i ) {
```

Sonderzeichen in der Regex:

In einer Regex werden genauso wie in einer Zeichenkette mit doppelten Anführungszeichen bestimmte Zeichen(-kombinationen) (z.B.\, \n, \t, $) zu Sonderzeichen. Wenn man in einer Zeichenkette ein $ ausgeben will, muß man entweder einfache Anführungszeichen verwenden, oder vor dieses Zeichen einen Backslash stellen (\$ steht für das Zeichen $). Genauso bei den Formatanweisungen von (s)printf: da kann man ein %-Zeichen angeben, indem man %% schreibt.

Wenn man will, daß $key nicht auf den Inhalt der Variablen $key passt, sondern auf die Zeichen von $key, dann muß man das $ escapen, d.h. einen Backslash davor schreiben: m/^\$key$/. Oder wenn man will, daß ein $ am Ende des Strings als $ erkannt wird, muß man genauso schreiben: m/\$/; Weiters haben noch die Zeichen ? + * [{ - / \ (Sonderbedeutungen

Grammatik:

Verwenden wir mal folgendes Beispiel als Basis, bei dem sich nur das Muster ändert:

Code: regex_02.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my @sprachen = qw(Perl Pascal VisualBasic Quickbasic
06:                   Fortran C C++ Java Python Smalltalk
07:                   Cobol Oberon Basic Delphi Ada);
08:
09: foreach my $sprache (@sprachen) {
10:     if( $sprache =~ m/r/ ) {
11:         print "$sprache\n";
12:     } # if
13: } # foreach
```

In diesem Beispiel ändern wir derzeit nur das Muster in Zeile 10:

1. Im aktuellen Beispiel geben wir alle Programmiersprachen aus, die ein kleines **r** beinhalten (Perl und Fortran)
2. Wir wollen alle Sprachen wissen, die mit **P** beginnen und mindestens 5 Zeichen lang sind:
   Das Sonderzeichen . (der Punkt) steht für genau ein beliebiges Zeichen (außer \n). Also könnten wir schreiben:
   `$sprache =~ m/^P..../;` (Pascal, Python)
   (Falls man wirklich einen Punkt haben will, muß man den escapen und schreiben: \.)
3. alle Sprachen, in denen zu Beginn ein P steht und am Ende ein l und mindestens ein Zeichen dazwischen ist: Mit dem Sonderzeichen + kann man sagen, daß der letzte Ausdruck einmal oder öfter kommen muß:
   `$sprache =~ m/^P.+l$/;` (Perl, Pascal)
4. alle Sprachen, die am Ende ein ++ haben: Da es sich bei dem + um ein Sonderzeichen handelt, muß man schreiben
   `$sprache =~ m/\+\+$/i;`
   Mit {Anzahl} kann man auch genau angeben, wie oft das letzte Zeichen kommen soll:
   `$sprache =~ m/\+{2}$/i;`
5. alle Sprachen, deren Länge mindestens 6 und höchstens 11 Zeichen beträgt: Mit {start,ende} kann man auch Bereiche angeben
   `$sprache =~ m/^.{6,11}$/i;`
6. alle Sprachen, deren Länge höchstens 5 Buchstaben beträgt:
   `$sprache =~ m/^.{0,5}$/i;`
   + könnte man auch als {1,} schreiben (von 1 bis unendlich)
   .* steht für {0,}, also nullmal oder öfter
7. alle Sprachen, die auf i oder ic enden:
   `$sprache =~ m/ic{0,1}$/i;`

Für den Ausdruck {0,1} gibt es auch eine Abkürzung: ?
```
$sprache =~ m/ic?$/i;
```

8. alle Sprachen, in denen ein großes oder kleines B vorkommt, gefolgt von einem kleinen a: Mit [Bb] kann man Zeichenklassen angeben, von denen genau eins vorkommen muß:
```
$sprache =~ m/[Bb]a/;
```
9. alle Sprachen, die nur aus den Buchstaben von A bis O bestehen:
```
$sprache =~ m/^[abcdefghijklmno]+$/i;
```
Mit - kann man auch Bereiche angeben, z.B.
```
$sprache =~ m/^[a-o]+$/i;
```
Und wenn man auf das /i auch noch verzichten will, kann man auch schreiben:
```
$sprache =~ m/^[a-oA-O]+$/;
```

Ausdruck:	passt auf:
^	den Anfang einer Zeichenkette
$	das Ende einer Zeichenkette
.	ein beliebiges Zeichen (außer \n)
\.	einen Punkt
a{5}	fünf a hintereinander
a{4,}	mindestens vier a hintereinander
a{0,6}	höchstens sechs a hintereinander
a{3,5}	mindestens drei a, aber maximal 5 a
a+	mindestens ein a
a*	kein, ein oder mehrere a
[a-z]	einen Kleinbuchstaben (=Zeichenklasse)
[^a-z]	ein Zeichen, das kein Kleinbuchstabe ist
[0-9]	eine Ziffer
\d	eine Ziffer
[^0-9]	ein Zeichen, das keine Ziffer ist
\D	ein Zeichen, das keine Ziffer ist
\s	whitespace (Leerzeichen, Tabulator, Zeilenumbruch)
\S	ein Zeichen, das kein Whitespace ist
\w	ein Zeichen der Zeichenklasse [A-Za-z0-9_]
\W	ein Zeichen, das nicht in der Zeichenklasse [A-Za-z_] ist
\n	ein Zeilenumbruch
\t	ein Tabulator
(abc|def)	abc oder def
\b	Wortgrenze (am Anfang oder Ende eines Wortes)
\B	keine Wortgrenze (also in einem Wort)
[\b]	Backspace (lösche das letzte Zeichen; nur in Zeichenklassen, sonst Wortgrenze)
\@	@-Zeichen
\/	/ (wenn das Trennzeichen des Musters / ist, muß es escaped werden)
\?	?-Zeichen

Speicherung mit regulären Ausdrücken:

Mit ( ) kann man den Text, der durch einen oder mehrere Ausdrücke gefunden wurde, abspeichern. Im Muster selbst kann man mit \1 \2 ... \9 darauf zugreifen, nach dem Muster durch die Spezialvariablen $1 $2 ... $9

Code: regex_03.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $string = "192.168.1.14";
06:
07: if ( $string =~ m/(\d{1,3})\.(\d+)\.(\d+)\.(\d+)/ ) {
08:
09:     print "\$1: $1\n";
10:     print "\$2: $2\n";
11:     print "\$3: $3\n";
12:     print "\$4: $4\n";
13:
14: } # if
```

Es gibt (parallel zu /i) die Option /x, die es erlaubt, Muster so zu schreiben, daß man sie auch wieder lesen kann. Dort kann man beliebig viele Leerzeichen einfügen und sogar Kommentare. Wenn man allerdings in einem Muster nach einem Leerzeichen suchen will, muß man es entweder escapen (also \ ) oder allgemeiner \s verwenden:

Code: regex_04.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $string = "192.168.1.14";
06:
07: if ( $string =~ m/
08:       ( \d+ )      # speichere mindestens eine Zahl
09:       \.           # ein Punkt (nicht speichern)
10:       ( \d+ )      # speichere die naechsten Zahlen
11:       \.           # ein Punkt (nicht speichern)
12:       ( \d{1,3} ) # speichere ein bis drei Zahlen
13:       \.           # ein Punkt (nicht speichern)
14:       ( \d{1,3} ) # speichere die naechsten 1-3 Zahlen
15:       /x ) {
16:
17:     print "\$1: $1\n";
18:     print "\$2: $2\n";
19:     print "\$3: $3\n";
20:     print "\$4: $4\n";
21:
22: } # if
```

Bei komplexeren Suchmustern empfehle ich, diese Möglichkeit zu verwenden, denn dann hat man auch nach Jahren noch die Chance, ein Suchmuster zu verstehen.

Ersetzungen mit regulären Ausdrücken:

Eine Ersetzung mit regulären Ausdrücken sieht prinzipiell folgendermaßen aus:

```
$string =~ s/Muster/Ersetzung/;
```

z.B. `$string =~ s/\b[Bb]asic/Perl/;`

Code: regex_05.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $string = "Seife ist Basich. Heute lernen wir Basic.";
06:
07: $string =~ s/
08:     \b        # wortgrenze
09:     [bB]asic  # basic oder Basic
10:     \b        # wortgrenze, damit Basich nicht gefunden wird
11:     /Perl/gx; # ersetze das gefundene durch Perl
12:
13: print "$string\n";
```

- Zeile 11: hier wird die Option /g verwendet, die sagt, daß nach dem ersten Vorkommen noch weiter gesucht werden soll, also alle gefundenen Muster ersetzt werden sollen.

Output:

```
F:\apacheweb\test_8085\html\codes>regex_05.pl
Seife ist Basich. Heute lernen wir Perl.

F:\apacheweb\test_8085\html\codes>
```

Wenn man will, daß die Ersetzung als Perl-Code ausgeführt wird, kann man als Option /e mitgeben:

```
$string =~ s/(\d\d\d)/ my $i = $1; $i++; "$i$i" /ge;
```

Reguläre Ausdrücke zu gierig?

Grundsätzlich versucht die Regex-Engine bei Ausdrücken wie .+ immer, auf möglichst viele Zeichen zu matchen.

Code: regex_06.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $string = "Dies ist ein <b>fetter Ausdruck</b>. Und dies ist ein " .
06:     "<b>zweiter fetter Ausdruck</b>. Und dies ein <b>dritter</b>";
07:
08: while ($string =~ m/<b>(.+)<\/b>/gs) {
09:     print "$1\n";
10: } # while
```

- Zeile 08: die Option /s sagt Perl, daß . auch auf Zeilenumbrüche passen soll

Output:

```
F:\apacheweb\test_8085\html\codes>regex_06.pl
fetter Ausdruck</b>. Und dies ist ein <b>zweiter fetter Ausdruck</b>. Und dies e
in <b>dritter

F:\apacheweb\test_8085\html\codes>
```

Manchmal will man jedoch so wenig Zeichen wie möglich haben. Da kann man nach Quantoren (z.B. +, *) das Zeichen ? schreiben:

Code: regex_07.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $string = "Dies ist ein <b>fetter Ausdruck</b>. Und dies ist ein " .
06:     "<b>zweiter fetter Ausdruck</b>. Und dies ein <b>dritter</b>";
07:
08: while ($string =~ m/<b>(.+?)<\/b>/gs) {
09:     print "$1\n";
10: } # while
```

Output:

```
F:\apacheweb\test_8085\html\codes>regex_07.pl
fetter Ausdruck
zweiter fetter Ausdruck
dritter

F:\apacheweb\test_8085\html\codes>
```

Nützliche Konstrukte:

- Überprüfung, ob was matcht:

```
if ($string =~ /pattern/) {
  print "passt\n";
}
```

- Filterung auf erlaubte Zeichen (Buchstaben und Ziffern):

```
$string =~ s/[^A-Za-z0-9]//g;
```

- Direkte Zuweisung ohne $1, $2,... (Achtung: danach sollten $value1, $value2, usw. auf Fehler überprüft werden)

```
my ($value1, $value2) = $string =~ /(\d+)\s+(\d+)/
```

- Wie oft kommt der Ausdruck "perl" in einem $string vor? (Array in skalarem Kontext!)

```
my $anzahl = $string =~ /perl/ig;
```

- Für jeden Match eine Schleife durchlaufen:

```
while( $string =~ /(\d+)/g ) {
  print "Match: $1\n";
}
```

- Integerzahlen Tausenderpunkte verpassen:

```
1 while $string =~ s/(\d+)(\d\d\d)/$1.$2/;
```

Nützlich sind die folgenden perldoc's: perlrequick, perlretut und perlre

Directoryhandling:

Das Öffnen eines Verzeichnisses funktioniert fast genauso wie das Öffnen einer Datei:

Code: dir_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $dir = $ENV{'WINDIR'};
06:
07: unless (opendir(DIR, $dir)) {
08:     die "Error in opening '$dir': $!\n";
09: } # unless
10:
11: foreach my $entry (readdir(DIR)) {
12:     if ($entry =~ m/\.exe$/) {
13:         print "$entry\n";
14:     } # if
15: } # foreach
16: closedir(DIR);
```

- Zeile 5: Über die Umgebungsvariable **%ENV** wird das Windows-Verzeichnis ermittelt und in **$dir** gespeichert
- Zeile 7-9: Verzeichnis öffnen und dem Handle **DIR** zuweisen, Fehlerabfrage
- Mit **readdir(DIR)** werden die Inhalte (d.h. Dateien, Verzeichnisse, ...) ausgelesen und bei jeder Iteration **$entry** mit einem neuen Wert versorgt. **readdir()** funktioniert auch kontextabhängig; im skalaren Kontext gibt es einen Eintrag zurück, im Listenkontext alle Einträge.
- Zeile 16: Das Verzeichnis wird wieder geschlossen und der Handle freigegeben
- Zeile 12: Wenn eine Datei die Endung .exe hat, gib den Namen aus

Verzeichnis oder Datei?

Code: dir_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $dir = $ENV{'WINDIR'};
06:
07: opendir(my $DIR, $dir) or
08:     die "Error in opening '$dir': $!\n";
09:
10: foreach my $entry (readdir($DIR)) {
11:
12:     if (-d "$dir/$entry") {
13:         printf "%-10s $entry\n", 'Verzeichnis';
14:     } # if
15:     elsif (-f "$dir/$entry") {
16:         printf "%-10s $entry\n", 'Datei';
17:     } # elsif
18:     else {
19:         printf "%-10s $entry\n", 'Unbekannt';
20:     } # else
21:
22: } # foreach
23: closedir($DIR);
```

Achtung: bei opendir(DIR) muß man den Pfad mit angeben!

Kommando:	Gibt wahr zurück, wenn der Eintrag
-e	existiert
-f	eine Datei ist
-d	ein Verzeichnis ist
-s	größer als 0 Bytes ist (gibt die Größe zurück)

siehe auch: perldoc -f -e

Inhalte filtern:

Entweder verwendet man opendir/readdir/closedir und testet die Dateinamen gegen eine Regex, oder man verwendet glob(Muster):

Code: dir_3.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $dir = $ENV{'WINDIR'};
06:
07: my @files = glob("$dir/*.exe");
08:
09: if ($!) {
10:     die "Fehler: konnte '$dir/*.exe' nicht auslesen: $!\n";
11: } # if
12:
13: foreach my $file (@files) {
14:     print "$file\n";
15: } # foreach
```

- Zeile 7: die Funktion glob kann man mit Dos-Platzhaltern * ? ansprechen und gibt eine Liste der Dateien zurück
- Zeilen 9-11: Fehler abfangen
- Die Funktion glob() hat allerdings Probleme mit Leerzeichen in Dateinamen. Von daher empfehle ich opendir/readdir/closedir

Programmparameter:

Wenn einem Perl-Script Parameter übergeben werden, findet man die in der Liste @ARGV

Code: params_1.pl

```
1: #! /usr/bin/perl
2: use warnings;
3: use strict;
4:
5: print "Programmname: $0\n";
6: print "Parameter:\n";
7: foreach my $argv (@ARGV) {
8:     print "\t$argv\n";
9: }
```

Output:

```
F:\apacheweb\test_8085\html\codes>params_1.pl -c=2 "ausdruck mit leerzeichen"
Programmname: F:\apacheweb\test_8085\html\codes\params_1.pl
Parameter:
        -c=2
        ausdruck mit leerzeichen

F:\apacheweb\test_8085\html\codes>
```

Hier ein Beispiel, wie man Parameter der Form x=y a=b in einen Hash umwandeln kann:

Code: params_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my %argv = &ExtractParameters(@ARGV);
06:
07: foreach my $key (sort keys %argv) {
08:     print "$key => $argv{$key}\n";
09: } # foreach
10:
11:
12: # ------------------------------------------------------------
13: sub ExtractParameters {
14:     my @argv = @_;
15:     my %argv = ();
16:
17:     foreach my $argv (@argv) {
18:         my ($key, $value) = split(/=/, $argv, 2);
19:
20:         if (exists $argv{$key}) {
21:             die "Fehler: der Parameter '$key' wurde doppelt angegeben\n";
22:         } # if
23:         else {
24:             $argv{$key} = $value;
25:         } # else
26:
27:     } # foreach
28:
29:     return (%argv);
30: } # ExtractParameters
```

Output:

```
F:\apacheweb\test_8085\html\codes>params_2.pl -c=2 -b=3
-b => 3
-c => 2

F:\apacheweb\test_8085\html\codes>
```

Sowas in der Art - nur um einiges besser - gibt es aber schon fertig als Perl-Modul. GetOpt::Long verwende ich gerne, z.B.

Code: params_3.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: use Getopt::Long;
06:
07: my $infile  = "input.ldif";
08: my $server  = "127.0.0.1";
09: my $port    = 389;
10: my $verbose = 0;
11:
12: my $result = GetOptions("infile=s" => \$infile,
13:                         "server=s" => \$server,
14:                         "port=i"   => \$port,
15:                         "verbose"  => \$verbose);
16:
17: print "Result:  $result\n";
18: print "Infile:  $infile\n";
19: print "Port:    $port\n";
20: print "Verbose: $verbose\n";
```

- Zeile 5: Laden des Moduls Getopt::Long; hier wird eine Funktion namens GetOptions zur Verfügung gestellt
- Zeilen 7-10: Deklarieren der Variablen und Zuweisung von Standardwerten
- Zeilen 12-15: Die Funktion GetOptions wird mit einigen Parametern aufgerufen. Als Parameter dient ein Hash, der als Key eine Kombination von Parameternamen und Format enthält und als Value die Referenz auf eine Variable, die den Wert des Parameters erhalten soll.
- Zeile 12: "infile=s": der Parameter infile ist ein beliebiger **S**tring
- Zeile 14: "port=i": als Port soll nur eine **I**ntegerzahl zulässig sein; wenn man hier alse Fehlermeldung Zeichenkette übergibt, kommt eine Fehlermeldung.
- Zeile 15: "verbose": hier wird kein Format angegeben; das bedeutet, daß $verbose bei Angabe des Parameters --verbose auf einen wahren Wert gesetzt wird (1), und falls --verbose nicht als Parameter angegeben wird, dann auf einen falschen Wert (0)

Dann kann man das Script auf viele verschiedene Arten aufrufen und so die Parameter auf vielfältige Arten übergeben, z.B.

```
params_3.pl --server=hostname --port 689 --verbose --infile abcde.txt

params_3.pl -server=hostname -port 689 -verbose -infile=abcde.txt

params_3.pl -server hostname -infile abcde.txt
```

Wenn ein Parameter nicht angegeben wird, wird der Standardwert der "verlinkten" Variablen verwendet. Für weitere Informationen siehe **perldoc Getopt::Long**

Eine Suche auf http://search.cpan.org/ nach Getopt liefert eine Liste vieler weiterer Module, die Funktionalität zum Scannen von Programmparametern bietet.

Datum und Zeit:

Die Funktion **time()** liefert die Epochsekunden zurück. Epochsekunden ist die Anzahl der verstrichenen Sekunden seit dem 1.1.1970. Mit **localtime(time)** kann man die Epochsekunden in ein lesbareres Format umwandeln, wobei sich localtime je nach Kontext unterschiedlich verhält.

- Skalarer Kontext: gibt eine Zeichenkette zurück der Form: Wed Sep 3 11:06:10 2002
- Listenkontext: gibt eine Liste zurück, deren Elemente folgende Bedeutung haben:

Position:	Wert:	Bereich:	Anmerkung:
0	Sekunden	0-60	
1	Minuten	0-59	
2	Stunden	0-23	
3	Tag des Monats	1-31	
4	Monat	0-11	Achtung: Januar ist 0
5	Jahre, die seit 1900 vergangen sind	0-138 (oder mehr)	Achtung: immer 1900 addieren
6	Wochentag	0-6	0 ist Sonntag, 1 Montag, ...
7	Tag des Jahres	1-366	
8	Ist Sommerzeit aktiv?	0-1	Wahr, wenn Sommerzeit

Code: date_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $epochSeconds = time;
06: my $printableDate = &ConvertEpochSecondsToDate($epochSeconds);
```

```
07:
08: print "EPOCH:  $epochSeconds\n";
09: print "String: ", scalar(localtime(time)), "\n";
10: print "Datum:  $printableDate\n";
11:
12: # ------------------------------------------------------------
13: sub ConvertEpochSecondsToDate {
14:     my $epochSeconds = shift; # Kurzschreibweise
15:
16:     my @date = ( localtime($epochSeconds) )[0..5];
17:     $date[4]++; $date[5] += 1900;
18:
19:     foreach my $d (@date[0..4]) {
20:         $d = sprintf("%02d", $d);
21:     } # foreach
22:
23:     my $date = join(".", @date[3,4,5]);
24:     my $time = join(":", @date[2,1,0]);
25:
26:     return "$date $time";
27: } # ConvertEpochSecondsToDate
```

Output:

```
F:\apacheweb\test_8085\html\codes>perl date_1.pl
EPOCH:  1062580938
String: Wed Sep  3 11:22:18 2003
Datum:  03.09.2003 11:22:18

F:\apacheweb\test_8085\html\codes>
```

Datum in Epochsekunden umwandeln:

Code: date_2.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $date = "12.10.2003 16:33:19";
06:
07: if ($date =~ /^
08:     (\d\d?)\.(\d\d?)\.(\d\d\d\d) # dd.mm.yyyy
09:     \s+
10:     (\d\d?)\:(\d\d?)\:(\d\d?)    # hh:mm:ss
11:     $/x) {
12:
13:     # sek, min, stunde, tag, monat jahr
14:     my @date = ($6, $5, $4, $1, $2, $3);
15:     $date[4]--; # januar ist 0
16:     $date[5] -= 1900; # Jahr 0 ist 1900
17:
18:     use Time::Local; # Modul laden
19:     my $epochSeconds = timelocal(@date); # umwandeln
20:
21:     print "EPOCH: $epochSeconds\n";
22: } # if
23:
24: else {
25:     warn "Kein Datum in $date gefunden\n";
26: } # else
```

Es gibt eine Menge an Perl-Modulen, die die Arbeit mit Zeit und Datum vereinfachen, z.B. Time::Local, Date::Calc, Date::Manip, Date::Parse, ...

Formate:

Eine der Erklärungen des Namens Perl ist "**P**ractical **E**xtraction and **R**eport **L**anguage". Und Perl unterstützt auch dabei, Reporte und ähnliches zu erstellen. Bisher haben wir die Funktionen print und printf kennen gelernt.

Code: format_1.pl

```
01: #! /usr/bin/perl
02: use warnings;
03: use strict;
04:
05: my $inputFile = "rangliste.txt";
06: my @lines = ();
07:
08: unless (open (INPUT, $inputFile)) {
09:     die "Error: couldn't read '$inputFile': $!\n";
10: } # unless
11:
12: # lese die Zeilen ein
13: foreach my $line (<INPUT>) {
14:     chomp($line);
15:     next if $line =~ m/^\s+$/; # ueberspringe Leerzeilen
16:     push (@lines, $line); # fuege sie zu @lines hinzu
17: } # foreach
18:
19: close(INPUT);
20:
21: my @sortedLines = sort {
22:     my $timeA = ( split(/\t/, $a))[2]; # extrahiere die Zeit
23:     my $timeB = ( split(/\t/, $b))[2]; # extrahiere die Zeit
24:     &ConvertTimeToSeconds($timeA) <=> &ConvertTimeToSeconds($timeB)
25: } @lines;
26:
27: foreach my $i (0..$#sortedLines) {
28:     my ($name, $land, $zeit) = split(/\t/, $sortedLines[$i]);
29:     printf("%2d %-20s %-10s %10s\n", $i+1, $name, $land, $zeit);
30: } # foreach
31: # ------------------------------------------------------------
32: sub ConvertTimeToSeconds {
33:     my $time = shift;
34:
35:     if ($time =~ m/^(\d+)\:(\d+)\:(\d+)$/) {
36:         return 3600 * $1 + 60 * $2 + $3;
37:     } # if
38:     return;
39: } # ConvertTimeToSeconds
```

Es gibt jedoch mächtigere Funktionen wie Templates. Ein Template ist eine Art von Text, in dem Platzhalter vorkommen, die dann bei der Ausgabe durch die entsprechenden Werte ersetzt werden (vergleiche **printf()**) Eines dieser Templatingsysteme nennt sich Formate. Es ist zwar schon etwas angegraut, weil es aus finsterster Perl-Frühzeit stammt, aber man kann sie doch recht gut verwenden. Der Grund, wieso sie recht selten verwendet werden ist, daß sie globale Variablen benötigen:

Code: format_2_short.pl

```
01: #...
02:
03: use vars qw($pos $name $land $zeit); # globale Variablen deklarieren
04:
05: format STDOUT =
06:   @>> @<<<<<<<<<<<<<<<<<<< @<<<<<<< @>>>>>>>>
07: $pos, $name,                $land,    $zeit
08: .
09:
10: foreach my $i (0..$#sortedLines) {
```

```
11:     ($name, $land, $zeit) = split(/\t/, $sortedLines[$i]);
12:     $pos = $i + 1;
13:
14:     write;
15: } # foreach
16:
17: # ...
```

- Zeilen 1: entspricht den Zeilen 1-25 des letzten Beispieles:
- Zeile 3: hier werden die globalen Variablen deklariert (macht man normalerweise ganz oben im Programm)
- Zeile 5-8: ein Format für den Filehandle **STDOUT** wird definiert. Es beinhaltet eine Zeile der Ausgabe. @<<< usw. sind Platzhalter für Variablen. Welche Variablen das sind, wurde in Zeile 31 angegeben. Die Reihenfolge muß dieselbe sein wie die Reihenfolge der Platzhalter.
- Zeile 6: @>> bedeutet, daß zwei Stellen rechtsbündig für den Inhalt einer Variable reserviert sind, @<<<<< bedeutet, daß fünf Stellen linksbündig für eine Variable reserviert werden, und @|||| würde bedeutet, daß vier Stellen zentriert für eine Variable reserviert sind.
- Zeile 8: Ein Format wird durch einen Punkt in einer Zeile beendet
- Zeile 14: hier wird das Format ausgeführt. Es geht immer auf den gerade aktuellen Filehandle. Wenn es in eine Datei gehen soll, die dem Programm unter dem Filehandle **FH** bekannt ist, muß man das Format **FH** nennen und kann dann schreiben: **write (FH);**

Code: format_3_short.pl

```
01: #...
02: use vars qw($pos $name $land $zeit); # globale Variablen deklarieren
03:
04: $= = 8; # wie viele Zeilen hat eine Seite (Standard: 60)
05:
06: format STDOUT =
07:   @>> @<<<<<<<<<<<<<<<<<<< @<<<<<<< @>>>>>>>>
08: $pos, $name,                $land,    $zeit
09: .
10:
11: format STDOUT_TOP =
12: ------------------------------------------------------------
13: |               Rangliste irgendeines Rennens               |
14: |                       Seite @<<                           |
15: $%
16: ------------------------------------------------------------
17:
18: .
19:
20: foreach my $i (0..$#sortedLines) {
21:     ($name, $land, $zeit) = split(/\t/, $sortedLines[$i]);
22:     $pos = $i + 1;
23:
24:     write;
25: } # foreach
26: #...
```

- Zeile 1: entspricht den Zeilen 1-25 des vorletzten Beispiels:
- Zeile 4: durch das Zuweisen einer Zahl an die Variable $= kann gesagt werden, wie viele Zeilen eine Seite haben soll. Für unser Beispiel habe ich mal 8 Zeilen festgelegt. Der Standardwert ist 60 Zeilen. Nach so vielen Zeilen wird jedesmal ein Seitenumbruch gesendet.
- Zeilen 11-18: hier ist der Kopf für das Format STDOUT deklariert. Es muß den gleichen Namen wie das dazugehörige Format haben, nur mit der Endung _TOP
- Zeile 15: die Variable $% beinhaltet die aktuelle Seitennummer.

Wenn man mehrere verschiedene Formate für ein Filehandle haben will, kann man durch Zuweisen des Formatnamen an die Variable $~ = "NEUESFORMAT"; das aktuelle Format ändern.

Für nähere Informationen verweise ich auf ein Perl-Buch (z.B. Einführung in Perl, Programmieren mit Perl) oder perldoc perlform.

Eine sauberere Alternative ist das CPAN-Modul Perl6::Form

Kleine Helfer:

Die folgenden Module sind oft sehr hilfreich:

- diagnostics: wenn eingebunden, erklärt es Fehlermeldungen und Warnungen
- Benchmark: ermittelt Laufzeiten oder vergleicht mehrere Subroutinen
- Data::Dumper: zeigt Datenstrukturen an und erleichtert deren Analyse
- Devel::DProf: Profiler, ermittelt, wo im Code viel Zeit verbraten wird
- Encode: einfache Konvertierung von Zeichenketten in verschiedene Codetabellen
- File::Copy: kopiert und verschiebt Dateien
- File::Find: Verzeichnisse rekursiv durchlaufen
- FindBin: die Variable $FindBin::Bin enthält den Pfad zum aufgerufenen Perl-Script
- IO::Handle: lesbare Alternativen zu $|
- IO::Prompt: einfaches Einlesen von Userinput von STDIN
- List::Util, List::Utils, List::MoreUtil: einige Listenfunktionen, die in Perl selbst noch fehlen, z.B. first, max, min, sum, shuffle, ...
- PAR: das Programm pp konvertiert ein Perl-Script in eine exe-Datei (bzw. in ein Binary unter Linux/Unix)
- Readonly: endlich vernünftige Konstanten für Perl
- Regexp::Common : viele "fertige" reguläre Ausdrücke

Weitere nützliche Funktionen:

- chdir($verzeichnis): Wechselt in das aktuelle Verzeichnis
- rename($aktuellerName, $neuerName): benennt eine Datei um
- unlink( ): löscht eine Datei oder eine Liste von Dateien (Achtung: die Datei landet da nicht im Papierkorb)
- use FindBin qw($Bin); die Variable $Bin enthält den absoluten Pfad zum ausgeführten Script
- binmode(HANDLE); setzt einen Handle so, daß die Dateien binär eingelesen werden. Am besten direkt nach dem open ausführen
- sleep(20); läßt das Script für 20 Sekunden pausieren

Installation von Perl-Modulen:

Perl-Module findet man unter folgenden URLs:

1. Für ActivestatePerl (http://www.activestate.com/) gibt es einige schon vorkompilierte Module, die sehr einfach zu installieren sind. Unter Windows sollten diese verwendet werden (wenn vorhanden):
   - Perl5.8: http://ppm.activestate.com/PPMPackages/zips/8xx-builds-only/
   - Perl5.6: http://ppm.activestate.com/PPMPackages/zips/6xx-builds-only/
   - Perl5.5: http://ppm.activestate.com/PPMPackages/zips/5xx-builds-only/

   Diese Module können als Zip-Dateien downgeloaded und z.B. mit Winzip entpackt werden. Danach wechsle man in das Verzeichnis, in dem eine Datei mit der Endung *Modulname***.ppd** (*Modulname* verwende ich als Platzhalter) liegt und gibt dort ein: **ppm install** *Modulname***.ppd**. Dieser Weg ist ideal, wenn ein Rechner keine Internetverbindung hat, oder wenn diese über einen kompexeren Proxy abgewickelt wird.
   Bei aktiver Internetverbindung kann man diese Module auch mit ppm direkt aus dem Internet installieren:

```
E:\>ppm
PPM interactive shell (2.1.6) - type 'help' for available commands.
PPM> install Acme-Code-Police
Install package 'Acme-Code-Police?' (y/N): y
Installing package 'Acme-Code-Police'...
Bytes transferred: 1226
PPM> quit
Quit!

E:\>
```

   Mit **search Modulname** bekommt man eine Liste aller Module, deren Namen **Modulname** enthalten. Bei der Online-Installation mit ppm braucht man sich um die Perl-Version keine Gedanken machen, weil ppm automatisch nach den richtigen Modulen sucht.
   Activestate-Perl und vorcompilierte ppm-Module gibt es derzeit für Win32, Linux und Solaris.

2. Wenn diese Vorgehensweise nicht zum Ziel führt oder man kein Activestate-Perl benützt, muß man die Module selbst compilieren. Dazu gibt es einfachere und schwierigere, aber sicherere Wege:
   - Manuelle Installation: Modul von CPAN (=Central Perl Architecture Network, **die** zentrale Modulanlaufstelle für Perl auf http://www.cpan.org/) downloaden (als .tar.gz oder .zip), das Archiv entpacken und sich die Datei namens README (und, wenn vorhanden, INSTALL) durchlesen. Da steht drinnen, wie man das Modul installieren muß.
     Im Laufe der Zeit hat sich jedoch ein Standardweg herausgebildet, dem mittlerweile fast alle CPAN-Module folgen. Dafür braucht man ein **make** für das entsprechende Betriebssystem, und für manche Module auch einen C-Compiler. (Für Windows bekommt man ein Standalone-make z.B. von ftp://ftp.microsoft.com/Softlib/MSLFILES/nmake15.exe . Downloaden und in ein Verzeichnis entpacken, das in PATH eingetragen ist, z.B. in das Verzeichnis Perl/bin. Danach lautet der Standardweg:

     ```
     ... in das Verzeichnis mit der Makefile.PL wechseln
     perl Makefile.PL
     make
     make test
     make install
     ```

     Bei Windows muß anstelle von **make nmake** geschrieben werden.Und nur wenn ein Befehl fehlerfrei durchläuft, kann der nächste ausgeführt werden (es macht wohl ziemlich wenig Sinn, ein fehlerhaftes Modul zu installieren, oder?) Wenn da eine Fehlermeldung wie **cl.exe not found** ausgegeben wird, benötigt man MsVisualC++6 (bzw. den C-Compiler, mit dem man Perl übersetzt hat).
   - Da dieser Standardweg mittlerweile bei fast allen Modulen eingehalten wird, wurde dieser Weg zu einem Modul zusammengefasst, das einem die Installation erleichtert. Dieses Modul nennt sich CPAN, und wird in der Shell folgendermaßen ausgeführt:

     ```
     E:\>perl -MCPAN -e shell
     cpan shell -- CPAN exploration and modules installation (v1.61)

     cpan> install HTML::Template
     # ..... gekuerzt ...
     cpan> quit
     E:\>
     ```

     Wenn dieser Weg nicht funktioniert, kann man noch den manuellen Weg (wie oben beschrieben) wählen. Damit der Weg mit CPAN auch unter Windows gut funktioniert, braucht man neben **(n)make** noch einige weitere Programme, die unter Linux/Unix schon standardmäßig installiert sind (bzw. sein sollten). Von http://www.cygwin.com/ kann eine komplette Kommandozeilenumgebung (bash) für Windows downgeloaded und installiert werden, die fast alle nötigen Programme für Windows enthält. Dies sollte man vor der ersten Verwendung von CPAN installieren, weil dieses Modul beim ersten Aufruf eine ganze Menge Fragen stellt, von denen man fast immer die Default-Antworten verwenden kann (nur darauf achten, dass nmake anstelle des cygwin-make's verwendet wird). Die cygwin-Tools sind auch für den nächsten Weg eine Voraussetzung:

   - CPANPLUS: Dieses Modul ist leider noch kein Standardmodul, wird es aber in einer der nächsten Perl-Versionen (vermutlich 5.10) werden. Es versucht recht erfolgreich, die Probleme mit CPAN zu lösen und die Modulsuche und -installation komfortabler zu gestalten.

     ```
     E:\>perl -MCPANPLUS -e shell
     CPANPLUS::Shell::Default -- CPAN exploration and modules installation
     *** Please report bugs to <cpanplus-bugs@lists.sourceforge.net>.
     *** Using CPANPLUS::Backend v0.042.  ReadLine support disabled.
     CPAN Terminal> i TestCPAN Terminal>
     Installing: Test
     Checking if your kit is complete...
     Looks good
     Writing Makefile for Test
             E:\apps\gnu\Perl58\bin\perl.exe "-MExtUtils::Command::MM" "-e" \
     "test_harness(0, 'blib\lib', 'blib\arch')" t\fail.t t\mix.t t\onfail.t t\
     qr.t t\skip.t t\
     success.t t\todo.t
     t\fail.......ok
     t\mix........ok
     t\onfail.....ok
     t\qr.........ok
     t\skip.......ok
     t\success....ok
     ```

```
        1/11 skipped: just testing skip()
t\todo.......ok
All tests successful, 1 subtest skipped.
Files=7, Tests=40,  0 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)

Microsoft (R) Program Maintenance Utility Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

Successfully installed Test

All modules installed successfully
CPAN Terminal> help
# .... gekuerzt ...
CPAN Terminal> q
Exiting CPANPLUS shell
E:\>
```

Wenn der Weg über ppm nicht möglich oder erfolgreich ist, empfehle ich den CPANPLUS-Weg.
Die Shell kann meistens auch über den Befehl **cpanp** aufgerufen werden... Die Hilfe zur Shell kann man sich mit dem Befehl help ausgeben lassen. Die sollte man sich unbedingt mal durchlesen, weil darin steht, wie man sich viel Tipperei ersparen kann.

Für Perl geeignete Editoren:

Um möglichst effektiv Perl programmieren zu können, sollte ein Editor die meisten der folgenden Anforderungen erfüllen:

- Syntaxhighlightning (markiert Variablen, Schlüsselwörter, Strings, Kommentare usw. durch eine bestimmte Formatierung aus, sodaß der Code besser lesbar wird)
- (Halb-)Automatisches Indenting (damit kann man Codeblöcke einfacher einrücken, und braucht nicht mehrmals die Leertaste oder den Tabulator zu drücken, bis der Cursor in einer neuen Zeile an der richtigen Position steht)
- Zeilennummern
- Bearbeitung mehrerer Dateien gleichzeitig (z.B. MDI)
- Einfaches Copy&Paste
- Mächtige Such- und Ersetzungswerkzeuge (z.B. inkrementelle Suche, reguläre Ausdrücke als Suchmuster, ...)
- Für Code gut geeignete Schriftarten mit einer fixen Zeichenbreite (z.B. Courier, Courier New, ...)
- Zusammenpassende Klammern markieren
- eventuelles Debugging von Perl-Scripten

Der Autor dieser Einführung in Perl verwendet am liebsten Emacs, weil diese OpenSource-Software sowohl mit als auch ohne GUI für sehr viele Betriebssysteme verfügbar ist, nach einer kurzen Eingewöhnungszeit sehr intuitiv bedienbar und fast beliebig konfigurierbar ist. Es gibt aber viele weitere gut geeignete Editoren. Eine Liste findet man unter http://links.perl-community.de/.

Dokumentation zu Perl-Modulen:

Fast alle Module haben die Dokumentation eingebaut (im POD-Format). Diese Dokumentation kann man sich, wenn das Modul installiert ist, mit dem Kommando **perldoc Modulname** ansehen, z.B.

```
E:\>perldoc CGI
NAME
    CGI - Simple Common Gateway Interface Class
SYNOPSIS
            # CGI script that creates a fill-out form
            # and echoes back its values.
# ... gekuerzt ...
E:\>
```

Wenn das Modul noch nicht installiert ist, kann man bei http://search.cpan.org/ danach suchen.

Gute Perl-Bücher für Anfänger und Fortgeschrittene:

Titel:	Autor(en):	Verlag:	Preis:	Anmerkung:
Einführung in Perl	Randal L. Schwartz, Tom Phoenix	O'Reilly	~35€	Gute Einführung in Perl für Unix/Linux-Betriebssysteme.
Einführung in Perl für Win32-Systeme	Randal L. Schwartz, Erik Olson, Tom Christiansen	O'Reilly	~20€	Gute Einführung in Perl für Windows-Betriebssysteme
Programmieren mit Perl	Larry Wall, Tom Christiansen, Jon Orwant, Randal Schwartz	O'Reilly	~56€	Das Kamel-Buch ist eine vollständige Referenz zu Perl. Wenn man all das kann, was darin vorkommt, kann man Perl sehr gut. Man sollte jedoch mindestens die dritte Auflage kaufen, die zweite ist nur teilweise noch Up-To-Date
Perl-Kochbuch	Tom Christiansen, Nathan Torkington	O'Reilly	~46€	Kleine und große Rezepte zu Perl und eine sehr gute Ergänzung zu Programmieren mit Perl oder Einführung in Perl. Sollte jeder Perl-Programmierer haben.
Programmierung mit Perl DBI	Alligator Descartes, Tim Bunce	O'Reilly	~38€	Datenbankprogrammierung mit Perl (SQL über ODBC, DBI, ...)
Effektiv Perl programmieren	Joseph N. Hall, Randal L. Schwartz	Addison-Wesley	~20€	Guter Leitfaden, wie man besser und effektiver Perl Programmieren kann
Perl Best Practices	Damian Conway	O'Reilly	?	Sehr guter Styleguide mit super Tips; ist aber eher was für Fortgeschrittene

Hilfe bei Perl-Problemen:

Es gibt mehrere Perl-Webforen, in denen man gute Hilfe findet, wenn man sein Problem genau beschreibt und auch Informationen wie Fehlermeldungen, welches Betriebssystem verwendet wird usw. angibt.

- www.perlmonks.org : Das größte Forum weltweit, wo man sogar Antworten von Perl-Größen wie Randal Schwartz (=merlyn) oder Damian Conway (=TheDamian) bekommen kann. Sprache: Englisch. Dort bin ich unter dem Pseudonym strat vertreten, allerdings nicht besonders aktiv.
- www.perl-community.de : Das größte deutschsprachige Perl-Forum, das nach seinem Umzug von http://www.perl.de/ wieder neu aufgebaut wurde. Dort ist das Niveau auch recht gut. Dieses Forum ist eins meiner größten Hobbies. Dort bin ich unter dem Pseudonym Strat zu finden.

Perl-Stammtische:

In fast jeder größeren Stadt gibt es Perl-Mongers-Gruppen, die sich meistens 1-2 Mal pro Monat treffen, Vorträge über Perl oder Perl-verwandte Themen halten, gemeinsam Projekte machen oder einfach gemütlich ein Bier trinken. Auf http://www.pm.org/groups/europe.html sind die europäischen Perlmongers-Gruppen aufgelistet. Ich bin Mitglied bei den Frankfurter Perl-Mongers (http://frankfurt-pm.org/) und Röderbergweg.pm.

Veranstaltungen:

Jedes Jahr von Aschermittwoch bis zum darauffolgenden Freitag findet der Deutsche Perlworkshop mit Tutorials und Vorträgen statt: http://www.perlworkshop.de/ (so 2006 in Bochum)

Jedes Jahr findet irgendwo in Europa die YAPC::Europe statt, z.B. 2004 in Belfast, oder 2005 in Braha/Portugal: http://www.yapceurope.org/