# Erzeugen dynamischer Webseiten mit Perl

Marc Nause

Stand: 24.01.2007

# Inhaltsverzeichnis


Vorwort....4
1. Grundsätzliches....5
  1.1 Was ist Perl?....5
  1.2 Ausstattung....5
  1.3 Warum nicht PHP?....6
2. HTML....7
  2.1 Das Gerüst....7
  2.2 Einfache HTML-Tags....8
  2.3 Bilder und Links....9
3. Jetzt aber los!....11
  3.1 Aufbau eines Perl-Programms....11
  3.2 Hochladen und alles was dazu gehört....12
  3.3 Zugriff auf den Server per SSH....14
4. Perl genauer betrachtet....16
  4.1 Variablen....16
    4.1.1 Skalare....16
    4.1.2 Arrays....18
    4.1.3 Hashes....20
    4.1.4 use strict....21
  4.2 Kontrollstrukturen....22
  4.3 Schleifen....24
  4.4 99 Bottles of Beer....26
  4.5 Unterroutinen....26
  4.6 Variablen in Blöcken....27
5. Übergabe und Verarbeitung von Daten....29
  5.1 Formulare....29
  5.2 Übernahme der Daten durch ein Skript....31
    5.2.1 Übernahme aus GET....32
    5.2.2 Übernahme aus POST....34
  5.3 CGI.pm....35
  5.4 Pattern Matching....38
6. Datenquellen....43
  6.1 Dateien....43
  6.2 LWP::Simple....47
  6.3 Datenbanken....47
7. Beispielprogramme....53
  7.1 Ein einfacher Zugriffszähler....53
Anhang....55

A. Literaturempfehlungen..........55
B. Internetresourcen..........56
C. Software..........57

## Vorwort

Dieses Skript ist während der Vorbereitung des Kurses "Erzeugen dynamischer Webseiten mit Perl" am Rechenzentrum der Technischen Universität Braunschweig im März 2005 entstanden, wurde während der Vorbereitung auf die Kurse im September 2005 und März 2006 ergänzt, verbessert und erweitert und wird auch weiter verbessert, wenn Fehler oder Unzulänglichkeiten auffallen.

Ziel dieses Skriptes ist es, das nötige Handwerkszeug zu vermitteln, um dynamische Webseiten mit Perl zu erzeugen und einen Überblick zu erhalten, was Perl leisten kann. Im Idealfall soll das Skript es dem Leser bzw. der Leserin ermöglichen, sich lediglich anhand des Skripts ein Grundwissen zu erarbeiten, das es möglich macht, sich darauf aufbauend weitere Fähigkeiten anhand der im Anhang aufgeführten Literatur anzueignen. Es handelt sich nicht um eine umfassende Referenz oder eine Arbeit mit dem Anspruch auf Vollständigkeit. Viele Themen können nur angerissen werden. Der Anhang enthält Literaturempfehlungen, die es ermöglichen, den Stoff weiter zu vertiefen.

Fragen? Anregungen? Fehler? Kritik? Kontakt: **marc.nause@audioattack.de**
Die jeweils neuste Version des Skripts ist unter **http://low.audioattack.de** erhältlich.

# 1. Grundsätzliches

In diesem Kapitel werden zunächst eine Grundlagen erläutert, ohne die man zwar auch in Perl programmieren kann, die aber teilweise gut zu wissen sind.

## 1.1 Was ist Perl?

Perl wurde von Larry Wall[1] entwickelt und ist frei erhältlich. Seit der Veröffentlichung im Jahre 1993 wurde die Sprache unter Mithilfe vieler freiwilliger Helfer ständig weiterentwickelt und verbessert. Die zur Zeit aktuelle Version ist 5.8.8 Version 6 ist schon seit einiger Zeit angekündigt, lässt aber noch auf sich warten.[2]

Perl ist eine Skriptsprache[3], die für zahlreiche Systeme erhältlich ist.[4] daher ist es sehr einfach, auf einem System (zum Beispiel Windows) für ein anderes (zum Beispiel Linux) zu entwickeln und die Programme dann ohne Änderungen laufen zu lassen. Auch zur schnellen Entwicklung und zum Testen eigener Ideen ist Perl sehr gut geeignet. Ganz besonders gut kann man Perl dazu gebrauchen, große Textmengen zu bearbeiten. Zwar gibt es Menschen, die behaupten, es sein nicht möglich, in Perl lesbare Programme zu erzeugen, doch mit etwas gutem Willen ist es durchaus möglich, die Übersicht beim Programmieren zu behalten und Programme zu erzeugen, die auch von anderen gelesen und verstanden werden können.

Perl ist durch Module erweiterbar. Einige werden standardmäßig mitgeliefert, andere kann man zusätzlich installieren. Es gibt Module, die einem die Arbeit bei Routinejobs erleichtern (wie zum Beispiel CGI.pm), den Zugriff auf Datenbanken ermöglichen (DBI) oder eine grafische Benutzeroberfläche zur Verfügung stellen (TK). Einige Module werden wir im Folgenden kennen lernen, andere sind für spezielle Aufgaben geschrieben worden, die uns hier nicht weiter interessieren. Allein auf der Seite des CPAN (Comprehensive Perl Archive Network) sind über 7600 Module aufgelistet.[5]

## 1.2 Ausstattung

Zum Programmieren in Perl benötigt man nicht viel. Ein beliebiger Texteditor ist ausreichend, solange er in der Lage ist, Texte in einfachem ASCII auf die Festplatte zu speichern. Es sind auch spezielle Editoren erhältlich, die einem die Arbeit durch automatische Ergänzungen oder farbliche Hervorhebungen erleichtern, notwendig ist das jedoch nicht.

Außerdem benötigt man einen Rechner, auf dem man seine Programme laufen lassen kann. Dies wird in den meisten Fällen eine angemieteter oder sonst zur Verfügung gestellter Webserver sein. Viele extrem billige Angebote erlauben nicht die Ausführung eigener Programme oder haben erst

1 http://www.wall.org/~larry/
2 http://dev.perl.org/perl6/faq.html
3 Ja, es gibt Compiler...
4 http://www.perl.com/perl
5 http://www.cpan.org/modules/01modules.index.html (Die Datei ist über 1MB groß!)

gar kein Perl installiert. Im Zweifelsfall hilft ein Blick in den mit dem Hoster geschlossenen Vertrag oder ein Anruf beim Administrator.

Da niemand fortlaufend völlig korrekten Code produziert, sollte man seine Programme, bevor man sie auf einem öffentlich zugänglichen Server laufen lässt, vorher auf einem eigenen System testen, um Leistungseinbrüche oder Sicherheitsgefährdungen des Servers zu vermeiden. Zum Testen reicht die Installation von Perl auf dem Rechner, auf dem das Programm entwickelt wird, aus. Auf Linuxsystemen ist Perl meist schon installiert, entwickelt man unter Windows, muss man dies nachholen.[6] Optimal ist es, wenn man die Möglichkeit hat, ein Serversystem aufzusetzen, das dem System, auf dem das Programm laufen soll, ähnlich ist, aber nicht öffentlich zugänglich ist. Zu Testzwecken sind auch ältere Computer gut zu gebrauchen.

Eine sehr einfache Möglichkeit zur Einrichtung einer Testumgebung bietet das Projekt XAMPP der Apache Friends.[7] Das Paket enthält den Apache Webserver, die Datenbank MySQL, sowie die Sprachen PHP und Perl und ist für die Betriebssysteme Linux, Microsoft Windows 98, NT, 2000, 2003 und XP, Solaris und Mac OS X erhältlich.

## 1.3 Warum nicht PHP?

Natürlich kann man zur dynamischen Erzeugung von Webseiten u.a. PHP nutzen. Oft wird sie Entscheidung für eine Sprache getroffen, ohne vorher lange über Alternativen, die es durchaus gibt, nachzudenken. Es gibt aber auch gute Gründe, die für Perl und gegen PHP sprechen. Für mich sehr wichtig ist, dass Perl nicht nur für die Erstellung dynamischer Webseiten geschaffen wurde, sondern auch viele alltägliche Aufgaben schnell und einfach mit einem Perl-Programm erledigt werden können. Außerdem empfinde ich Perl insgesamt als konsistenter.

Im Internet findet man viele Seiten, die sich mit dem Vergleich von Perl und PHP (und gelegentlich noch weiteren Sprachen) befassen[8]. Eine Suche nach `perl vs php` mit einer beliebigen Suchmaschine sollte zahlreiche Seiten zum Vorschein bringen, die dieses Thema in großer Ausführlichkeit behandeln.

6 http://www.activestate.com/Perl.plex

7 http://www.apachefriends.org

8 Zum Beispiel: http://verplant.org/perl.vs.php.shtml

## 2. HTML

Zwar befassen wir uns mit der Erzeugung von Webseiten mit Perl, allerdings verstehen Browser nur HTML (Hypertext Markup Language).[9] Daher folgt nun eine kurze Einführung in die Grundlagen von HTML, wobei ich mich auf die wichtigsten Elemente konzentrieren werde. Auf die später benötigten Formulare werde ich eingehen, wenn sie gebraucht werden.

### 2.1 Das Gerüst

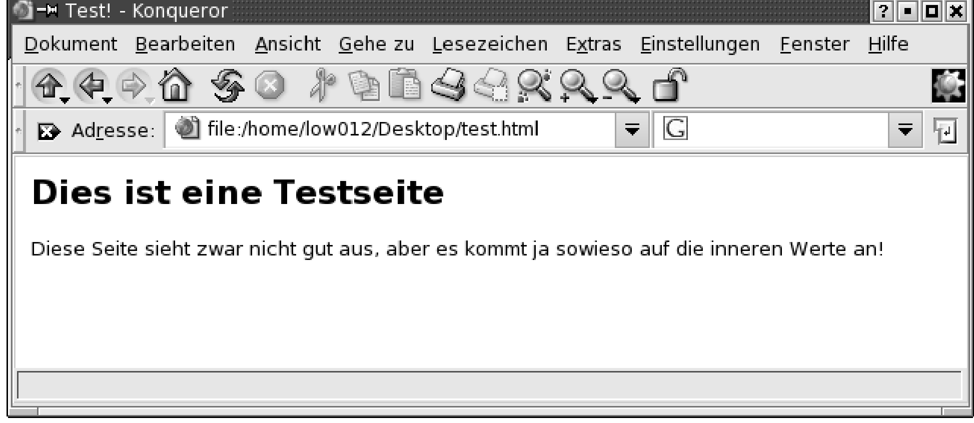
Jede HTML-Seite ist im Grunde gleich aufgebaut: Sie besteht aus einem Kopf, der Informationen enthält, die (bis auf eine Ausnahme) dem gewöhnlichen Betrachter verborgen bleiben. Diese Informationen sind interessant für den Browser oder auch für Suchmaschinen. Dann folgt der Körper der Seite, der die Informationen enthält, die der Browser dem Betrachter anzeigt.

Hier ein Beispiel für eine (sehr) einfache Webseite:

```
<html>
<head>
<title>Test!</title>
</head>
<body>
<h1>Dies ist eine Testseite</h1>
<p>Diese Seite sieht zwar nicht gut aus, aber es kommt ja sowieso auf die
inneren Werte an!</p>
</body>
</html>
```

Diesen Text speichert man nun unter einem beliebigen Namen (z.B. index), gefolgt von der Endung htm oder html als `index.htm` oder `index.html` ab.

Im Browser sieht das dann so aus:




Vergleicht man Quelltext und Anzeige im Browser, so fällt auf, dass einige Teile des Quelltextes angezeigt werden, andere nicht. Die Teile, die im Browserfenster angezeigt werden, sind die die im

9 OK, fast alle Browser verstehen Javascript, einige besonders kranke auch ActiveX, aber auch damit allein kann man keine Webseiten erstellen.

Quelltext zwischen `<body>` und `</body>` stehen. Aus dem Teil zwischen `<head>` und `</head>` ist auch etwas zu sehen, aber nicht ganz so offensichtlich. In der Titelleiste des Browsers sieht man "Test!". Das ist genau die Zeichenkette, die auch im HTML-Dokument im Kopf als Titel angegeben ist.

## 2.2 Einfache HTML-Tags

Wie man bei genauer Betrachtung des Quelltextes erkennt, treten HTML-Tags scheinbar immer paarweise auf. Dies stimmt im Allgemeinen auch, es gibt jedoch auch Ausnahmen, wie wir später sehen werden. Als erstes wird ein öffnendes Tag benutzt, das den Browser eine bestimmte Anweisung gibt (zum Beispiel "Ab hier handelt es sich um eine Überschrift!") und dann eine schließendes Tag (zu erkennen am Schrägstrich), das dem Browser mitteilt, dass die Darstellungsvorschrift wieder aufgehoben wird.

Doch zunächst hier zu den wichtigsten HTML-Tags, die benötigt werden, um Texte in einem Browser darzustellen:

```
<h1>Text</h1>
...
<h6>Text</h6>
```

Dieses Tag umschließt einen Text, der eine Überschrift enthält. Die Zahl gibt die Hierarchieebene der Überschrift an. Es empfiehlt sich, Überschriften mit diesem Tag zu versehen, um Suchmaschinen und Menschen mit eingeschränkter optischer Wahrnehmung, die eventuell die Seite nicht über das Auge wahrnehmen, das erfassen des Textes und seiner Struktur zu vereinfachen. Je kleiner die Zahl, desto größer wird die Überschrift normalerweise in Browsern dargestellt.

```
<p>Text</p>
```

Diese Tags umschließen einen Absatz, der Text enthält.

```
<b>Text</b>
```

Der Text zwischen diesen Tags erscheint **fett** (bold).

```
<i>Text</i>
```

Der Text zwischen diesen Tags erscheint *kursiv* (italic).

```
<u>Text</u>
```

Der Text zwischen diesen Tags erscheint unterstrichen (underlined).

Diese Tags können natürlich auch miteinander kombiniert werden. Dabei sollte man allerdings darauf achten, die Tags in der gleichen Reihenfolge zu schließen, wie sie auch geöffnet wurden.

FALSCH: `<p>Dies ist ein <b>Text!</p></b>`
RICHTIG: `<p>Dies ist ein <b>Text!</b></p>`

Um einen Zeilenumbruch im Browser zu erzwingen, muss an der entsprechenden Stelle im Text das `<br>`-Tag eingefügt werden.

## 2.3 Bilder und Links

Die HTML-Tag zum Einfügen von Bildern und Links benötigen etwas Wissen über die Pfade auf der Festplatte, die zu den Bilddateien bzw. Seiten, auf die Verweise zeigen sollen, führen.

Zunächst die einfache Möglichkeit, bei der HTML-Dokument und Bild im gleichen Verzeichnis liegen:

```
<img src="bild.jpg">
```

Hier wird das Bild mit dem Namen `bild.jpg` angezeigt. Liegt nun das Bild nicht im gleichen Verzeichnis, wie die HTML-Datei, sondern in einem Unterverzeichnis, so muss der Pfad angegeben werden:

```
<img src="unterverzeichnis/bild.jpg">
```

Liegt die Datei in dem Verzeichnis über dem, in dem sich die HTML-Datei befindet, so muss der Pfad folgendermaßen verändert werden:

```
<img src="../bild.jpg">
```

Links auf andere Seiten lassen sich ganz ähnlich anlegen:

```
<a href="seite.html">Text</a>
```

Hier wird ein Link auf eine Seite namens `seite.html` erzeugt, die im gleichen Verzeichnis wie das HTML-Dokument liegt. `Text` wird als Linktext angezeigt.

Es können auch Bilder als Links benutzt werden. Dazu muss der Text lediglich durch eine Verweis auf ein Bild ersetzt werden:

```
<a href="seite.html"><img src="bild.jpg"></a>
```

Die Pfade, die bis jetzt benutzt wurden, waren relative Pfade, da stets ein Ort auf dem Server ausgehend von der zu bearbeitenden Datei angegeben wurde. Man kann aber auch absolute Pfade angeben. Dies ist auch die einzige Möglichkeit, auf Dateien zu Verweisen, die auf anderen Servern liegen.

```
<a href="http://www.google.com/technology/pigeonrank.html">Aha!</a>
```

Absolute Pfade auf dem eigenen Server bringen eigentlich nur Nachteile mit sich, sind aber trotzdem möglich.

## 3. Jetzt aber los!

Genug der Vorrede, jetzt kommen wir zum ersten Programm in Perl, das zwar sehr einfach ist, aber doch genug Substanz bietet, um alle typischen Schritte zu erläutern, die notwendig sind, um in Perl eine HTML-Seite zu erzeugen.

```
#!/usr/bin/perl
#Ein erster Versuch!
print"Content-type: text/html\n\n";
print"<html><head><title>Hallo Welt!</title></head>\n";
print"<body><h1>Hallo Welt!</h1></body></html>\n";
```

### 3.1 Aufbau eines Perl-Programms

Zunächst wollen wir uns kurz den Aufbau des Programms ansehen. Auffällig ist schon die erste Zeile. Hier wird dem Server gesagt, wo er den Perl-Interpreter finden kann, der das Programm ausführen soll. Schreibt man das erste Mal ein Programm für einen Server und weiß nicht, wo Perl installiert ist, so kann man (sofern es sich um einen Unix/Linux-Server handelt) sich per SSH einloggen und mit `which perl` herausfinden, welcher Perl-Interpreter standardmäßig auf diesem Server genutzt wird. Wichtig ist, dass diese Zeile wirklich die allererste Zeile des Skripts ist. Auch vorhergehende Leerzeilen führen zu einem Fehler!

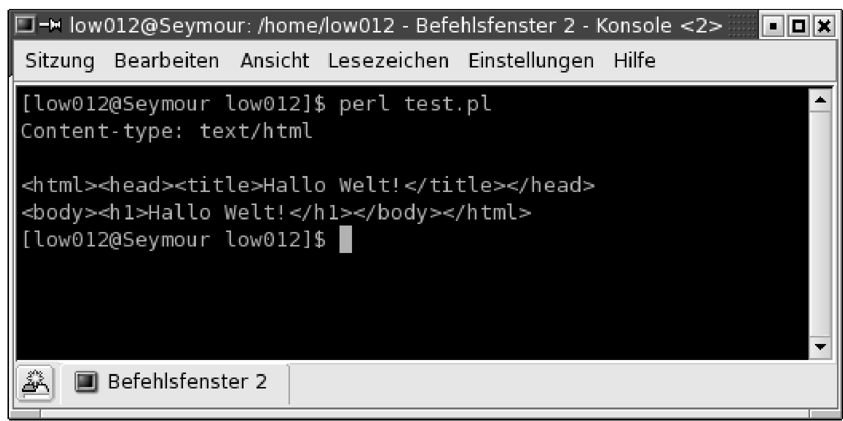
Bei der zweiten Zeile handelt es sich um einen Kommentar. Zeilen, die mit einer Raute beginnen, werden nicht ausgeführt. Tritt eine Raute innerhalb einer Zeile (und nicht innerhalb von Anführungszeichen einer Anweisung) auf, so wird der nach dieser Raute folgende Teil der Zeile nicht ausgeführt. An Kommentaren sollte man nicht sparen, denn sie erleichtern die Wartung eines Programms ungemein.

In den nun folgenden 3 Zeilen wird eine HTML-Seite ausgegeben. Nach jedem Befehl folgt ein Semikolon. Es können mehrere Anweisungen durch jeweils ein Semikolon getrennt in eine Zeile geschrieben werde. Davon ist jedoch in den meisten Fällen abzuraten, da die Lesbarkeit darunter meist erheblich leidet.

Zeile 3 enthält den MIME-Type des zu erzeugenden Dokuments. Kurz gesagt ist dies eine Anweisung an den Browser, die folgenden Daten als HTML-Seite aufzufassen und anzuzeigen. Bei statischen Seiten erledigt der Server das Erstellen und Senden dieser Information, aber da wir die Daten selbst mittels des Perl-Programms erstellen, weiß der Server ja nicht, worum es sich handelt.

Die Zeilen 4 und 5 enthalten das eigentliche HTML-Dokument, wie es auch ähnlich schon weiter oben in Kapitel 2.1 aufgetaucht ist. Die Zeichenfolge `\n` am Ende der in den Anführungszeichen eingeschlossenen Zeichenfolgen gibt an, dass in der erzeugten HTML-Seite an dieser Stelle ein Zeilenumbruch erfolgen soll. (Wohlgemerkt geschieht dies nur im Quelltext, um in der Browseransicht einen Zeilenumbruch zu erzeugen, muss `<br>` verwendet werden.)

Man sollte, wie schon angesprochen, ein selbst geschriebenes Programm zunächst offline Testen. Zum Testen kann man das Programm auf dem eigenen Rechner aufrufen (hier unter Linux):




Das sieht nicht besonders aufregend aus, man erkennt aber, dass der Text ausgegeben wurde, der auch erscheinen sollte. Hätte das Programm einen Fehler enthalten, so hätte die Ausgabe einen Hinweis auf den Fehler gegeben. Möchte man zusätzlich noch Warnungen erhalten, wenn man fehleranfällige Konstruktionen benutzt, sollte man die erste Zeile wie folgt erweitern:

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

## 3.2 Hochladen und alles was dazu gehört...

Damit nun die so erzeugte Seite im Webbrowser angezeigt werden kann, muss das Skript auf einen Server geladen werden, der es dann ausführen kann und die HTML-Seite an einen Browser ausliefern kann:

Zunächst soll hier noch kurz etwas nachgeholt werden. Ich habe unser Perl-Programm `test.pl` genannt, ohne weiter darauf einzugehen. Es ist jedoch so, dass die meisten Server die Endungen `.pl` und `.cgi` für Perl-Programme akzeptieren. Man kann einen Server aber auch so konfigurieren, dass völlig andere Endungen erwartet werden. Im Zweifelsfall ist der Administrator des Servers hier der richtige Ansprechpartner.

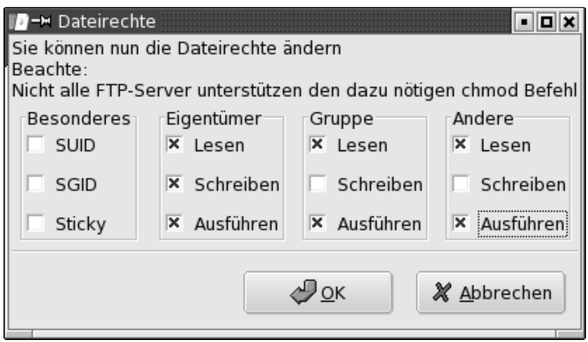
Weiterhin kann es wichtig sein, in welchem Verzeichnis auf dem Server das Perl-Programm abgelegt wird. Hier gibt es einmal die Möglichkeit, dass Perl-Programme in allen Verzeichnissen ausgeführt werden können. Aus Sicherheitsgründen werden Perl-Programme auf manchen Servern

nur in bestimmten Verzeichnissen ausgeführt, die meist `cgi-bin` heißen.

Das Hochladen des Perl-Programms sollte mit einem FTP-Client geschehen. Zwar bieten auch Webbrowser oft ein rudimentäre Unterstützung des FTP-Protokolls, allerdings fehlen meist einige für uns wichtige Funktionen. Im Anhang findet sich eine Liste beliebter FTP-Clients.

Zunächst muss sichergestellt sein, dass der FTP-Client sich entweder im ASCII-Modus befindet oder er eine zuverlässige Automatik besitzt, die auch aktiviert ist. Das Hochladen eines Perl-Programms im Binary-Modus (in dem sich FTP-Clients oft standardmäßig befinden), führt dazu, dass das Programm später auf dem Server nicht funktioniert.

Nach dem erfolgreichen Hochladen des Perl-Programms, müssen jetzt noch die Dateirechte angepasst werden. Dies kann auch mittels FTP-Client geschehen. Meist findet sich ein entsprechender Menüpunkt, wenn man mit der rechten Maustaste auf den Namen des Programms auf dem Server klickt. Der Menüpunkt heißt "CHMOD" oder ähnlich wie "Rechte ändern". "CHMOD" weist bereits darauf hin, dass es sich hier um das Äquivalent des Unix-Befehls `chmod` handelt. Die Dateirechte müssen nun so gesetzt werden, wie man es auch mit dem Unix-Befehl `chmod 755 test.pl` machen würde. Dies entspricht dem im folgenden Bild Dargestellten.




Zwischen der etwas abstrakten Zahl und den Dateirechten besteht folgender Zusammenhang:

| | **User** | **Group** | **Other** |
|---|---|---|---|
| **Lesen** | 4 | 4 | 4 |
| **Schreiben** | 2 | 2 | 2 |
| **Ausführen** | 1 | 1 | 1 |
| Summe | 7 | 5 | 5 |

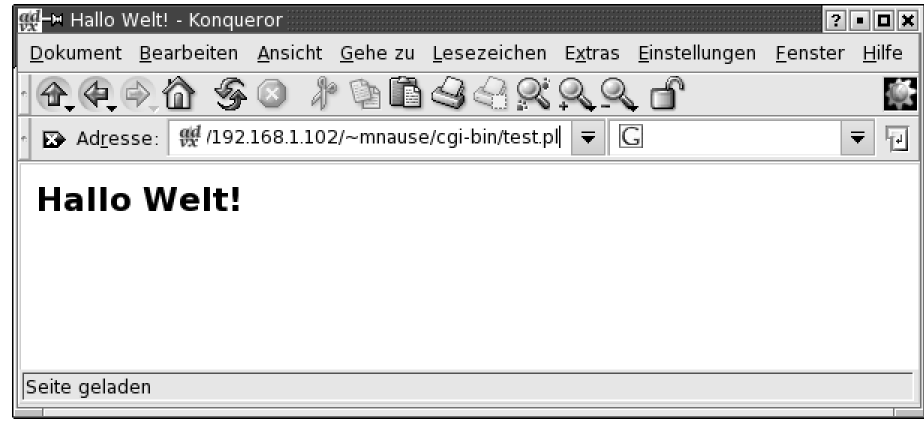
Es darf also der User (Eigentümer der Datei) die Datei lesen, schreiben und ausführen, alle anderen, die der gleichen Benutzergruppe angehören, dürfen lesen und ausführen und alle Benutzer außerhalb der Benutzergruppe des Eigentümers der Datei dürfen ebenfalls lesen und ausführen.

Die folgenden Dateirechte sollten für den Großteil aller Anwendungen funktionieren:

755 - Verzeichnisse, die Perl-Programme enthalten
777 - Verzeichnisse, die keine Perl-Programme enthalten
755 - Perl-Programme
666 – Text-Dateien (zum Beispiel Log-Dateien)
777 - HTML-Dateien

Sollte ein Programm nicht funktionieren, obwohl es im ASCII-Modus hochgeladen wurde und die Dateirechte richtig gesetzt sind, lohnt es sich, einen Blick auf die Rechte des Verzeichnisses zu riskieren, in dem sich das Programm befindet. Gelegentlich sind diese Rechte falsch gesetzt und treiben einen so an den Rand eines Nervenzusammenbruchs.

Angenommen wir haben die Datei `test.pl` im Ordner `/public_html/cgi-bin` abgelegt, sollte sich beim Aufruf der Datei mit einem Webbrowser folgender Anblick bieten:




## 3.3 Zugriff auf den Server per SSH

Für viele Arbeiten wäre es sehr bequem, wenn man direkt auf dem Server arbeiten könnte, als sitze man direkt davor. Viele Server bieten die Möglichkeit, sie per SSH zu bedienen. Dazu gibt man in der Kommandozeile

`ssh severname -l benutzername` oder `ssh benutzername@servername`

ein. Sind Benutzername auf dem Server und dem lokalen Rechner identisch, kann auf die Angabe des Benutzernamens verzichtet werden. Nachdem man die Eingabe mit `ENTER` bestätigt hat, wird versucht mit dem Server eine Verbindung aufzubauen. Es kann sein, dass man zunächst das Zertifikat des Servers akzeptieren muss. Hat dies funktioniert, so fragt der Server nach dem Passwort. Danach kann man auf dem Server arbeiten, als säße man direkt davor.

Microsoft Windows bringt keinen SSH-Client mit. Hier bietet es sich an, Putty zu benutzen, das

kostenlos erhältlich ist.[10]

Einige Server bieten auch die Möglichkeit, sich per Telnet einzuloggen. Dies sollte man, wenn möglich, nicht tun, da bei Telnet im Gegensatz zu SSH die übertragenen Daten nicht verschlüsselt werden.

10 http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

# 4. Perl genauer betrachtet

Nun haben wir also mittels Perl eine Webseite erzeugt. Dies ging vorher mit statischem HTML allerdings viel einfacher und man musste auch weniger beim Hochladen beachten. Aber natürlich bietet Perl viel mehr Möglichkeiten, als wir bisher kennen gelernt haben (und auch viel mehr, als wir im Rahmen dieses Kurses noch kennen lernen werden). Um richtige Programme schreiben zu können, müssen zunächst allerdings einige Grundbegriffe und Konzepte geklärt werden.

## 4.1 Variablen

Eins der wichtigsten Hilfsmittel sind Variablen. In Variablen können Zahlen, Wahrheitswerte, Buchstaben und Zeichenketten gespeichert werden, später wieder ausgelesen und bei Bedarf geändert werden. Sie sind somit das Gedächtnis eines Programms.

Als Perl-Programmierer muss man sich im Vergleich zu anderen Programmiersprachen um nicht besonders viel kümmern, wenn man mit Variablen arbeitet. Der Inhalt einer Variablen kann im Laufe eines Programms wechseln[11], allerdings – und das ist in vielen anderen Programmiersprachen nicht so – kann eine Variable erst eine ganze Zahl, dann eine Zeichenkette und später dann eine gebrochene Zahl beinhalten. Das trägt zwar auch nicht besonders zur Übersichtlichkeit bei, befreit den Programmierer aber andererseits von der Last, ständig auf den Datentyp der Variablen achten zu müssen.

In Perl gibt es die folgenden Variablentypen:

| Typ | Zeichen | Beispiel | Beschreibung |
|---|---|---|---|
| Skalar | $ | $wort | Eine Zahl oder eine Zeichenkette |
| Array | @ | @werte | Eine Liste von Werten mit einem ganzzahligen Index |
| Hash | % | %tage | Eine Gruppe von Werten mit einer Zeichenkette als Index |
| Unterroutine | & | &sub | Ein aufrufbares und ausführbares Stück Perl-Code |
| Typeglob | * | *kram | Alle Variablen namens kram |

Im den folgenden drei Kapiteln werde ich kurz auf Skalare, Arrays und Hashes eingehen. Unterroutinen werde ich am Ende des Kapitels besprechen. Typeglobs werden wir im Rahmen dieses Skripts nicht benötigen.

### 4.1.1 Skalare

Skalaren können sowohl Zahlen, als auch Zeichenketten zugewiesen werden. Hier einige Beispiele:

```
$zahl = 23
```
Integer (Ganzzahl)

11 Deshalb heißen sie ja auch Variablen.

| | |
|---|---|
| `$pi = 3.1415` | Realzahl |
| `$viel = 4.23e12` | wissenschaftliche Notation (4.23e12 = $4{,}23 * 10^{12}$) |
| `$ort = "Schwimmbad"` | String (Zeichenfolge) |
| `$woist  = "Ich bin im $ort"` | String mit Interpolation (wird zu "Ich bin im Schwimmbad") |
| `$teuer  = 'Es kostet $10'` | String ohne Interpolation |

Perl bietet außerdem die folgenden arithmetischen Operationen für numerische Werte:

| Beispiel | Name | Ergebnis |
|---|---|---|
| $a = $a + $b | Addition | $a wird die Summe von $a und $b zugewiesen |
| $a = $a - $b | Subtraktion | $a wird der Wert von $a abzüglich $b zugewiesen |
| $a = $b * $c | Multiplikation | $a wird das Produkt von $b und $c zugewiesen |
| $a = $x / $y | Division | $a wird das Ergebnis von $x dividiert durch $y zugewiesen |
| $a = $y % $z | Modulo | $a wird der Rest von $y dividiert durch $z zugewiesen |
| $a = $a ** $b | Potenzierung | $a wird der Wert $a hoch $b zugewiesen |
| ++$a, $a++ | Autoinkrement | Entspricht $a = $a + 1 |
| --$a, $a-- | Autodekrement | Entspricht $a = $a -1 |

In einem Programm würde das beispielsweise so aussehen:

```
$a = 423;
$b = 235;
print $a - $b;
```

Dieser Codeschnipsel würde den Wert 188 ausgeben.

Soll eine Variable um einem festen Wert erhöht (oder erniedrigt oder multipliziert...) werden, so ist auch die folgende Kurzform zulässig:

| | | |
|---|---|---|
| `$a += 5;` | entspricht | `$a = $a + 5;` |
| `$a *= 3;` | entspricht | `$a = $a * 3;` |

Mit den anderen oben genannten Operationen wird entsprechend verfahren.

Für Zeichenfolgen gibt es ebenfalls Additions- und Multiplikationsoperatoren. Die Addition entspricht der Verkettung zweier Zeichenketten:

```
$a = "The quick brown fox ";
$b = "jumps over the lazy dog.";
```

```
print $a . $b;
```

Dieses Codefragment gibt den Satz "The quick brown fox jumps over the lazy dog." aus.

Die Multiplikation entspricht der mehrmaligen Ausgabe einer Zeichenfolge.

```
$a = "Wiederholung! ";
$b = 3;
print $a x $b;
```

Hier wird "Wiederholung! Wiederholung! Wiederholung! " ausgeben.

Um die Länge eines Skalars zu erhalten, kann man den Befehl `length` folgendermaßen nutzen:

```
$Laenge = length($skalar);
```

Wenn man lediglich einen bestimmten Teil eines Strings benötigt, ist der Befehl `substr` sehr hilfreich. Er ermöglicht es, Aus einem gegebenen String einen Teil zu extrahieren. Man muss dazu lediglich angeben (und natürlich vorher wissen), welchen Teil man benötigt:

```
$a = substr $string, 0, 5;
```

Hier werden der Variablen `$a` die ersten 5 Zeichen des Strings `$string` zugewiesen. Bei der ersten Zahl (Offset) handelt es sich um die Anfangsposition, bei der zweiten Zahl um die Länge. Wählt man für den Offset eine negative Zahl, so wird ab dem Ende des Strings gezählt. Bei einer negativen Länge werden Zeichen vom Ende des Strings weggelassen. Wird die Länge weggelassen, bedeutet dies, dass die Zeichen bis zum Ende des Strings gewählt werden:

```
$a = substr $string, 6;
```

Hier werden der Variablen `$a` die Zeichen ab dem siebten Zeichen bis zum Ende zugewiesen.

Es ist auch möglich, einen bestimmten Bereich eines Strings durch andere Zeichen zu ersetzen:

```
$string = substr $string, 0, 5, "Sonne";
```

Hier werden die ersten fünf Zeichen des Strings `$string` durch das Wort „Sonne“ ersetzt.

Möchte man Bool'sche Werte durch Skalare darstellen, so geht dies ohne Probleme. Jeder Skalar mit einem Wert "0", 0, "" oder ein undefinierter Zustand liefert in Perl FALSE. Alle anderen Werte bedeuten TRUE.

### 4.1.2 Arrays

Arrays sind Variablen, die mehr als einen Wert enthalten, es sind geordnete Listen von Skalaren.

Der Zugriff erfolgt über einen Index, der bei 0 beginnt.

Hier ein Beispiel für ein Array:

```
@gieseler3 = ("Merz", "Vibe", "Soleil", "Schwanensee");
```

Auf die einzelnen Werte kann nun mittels Index zugegriffen werden:

```
print $gieseler3[0].$gieseler3[1].$gieseler3[3];
```

Es wird "MerzVibeSchwanensee" ausgegeben. Achtung: Es wird beim Zugriff auf die einzelnen Elemente des Arrays `$` verwendet, da die einzelnen Elemente ja Skalare sind. Weiter zu beachten ist, dass `$gieseler3` und `$gieseler3[1]` völlig unterschiedliche Dinge sind, obwohl beide ähnlich aussehen und zufällig fast gleich heißen. Bei dem einen handelt es sich um einen Skalar, das andere ist Teil eines Arrays und die Werte, die diese Variablen haben, sind völlig unabhängig voneinander.

Eine Besonderheit der Arrays in Perl ist, dass sie im Gegensatz zu zahlreichen anderen Sprachen nach der Initialisierung noch wachsen können.  Man definiert einfach ein neues Element.

```
$gieseler3[4] = "Wohnungen";
```

Schon ist das Array um ein Element größer, ohne dass man sich als Programmierer um irgendwas kümmern muss.

Ein Array muss nicht wie oben geschehen mittels `@gieseler3` definiert werden. Man kann auch die Elemente einzeln hinzufügen:

```
$gieseler3[0] = "Merz";
$gieseler3[1] = "Vibe";
$gieseler3[2] = "Soleil";
$gieseler3[3] = "Schwanensee";
```

Will man das gesamte Array Skalaren zuweisen, so kann das kurz mit

```
($null, $eins, $zwei, $drei) = @gieseler3
```

geschehen. Dies hat den gleichen Effekt wie

```
$null = $gieseler3[0];
$eins = $gieseler3[1];
$zwei = $gieseler3[2];
$drei = $gieseler3[3];
```

ist aber einfacher und schneller zu schreiben.[12]

12 Außerdem ist es etwas verwirrender für jemanden, der vielleicht irgendwann mal den Code warten muss.

Wo wir schonmal bei verwirrenden Anweisungen sind, noch ein kleiner Hinweis auf eine effiziente Methode, den Inhalt zweier Variablen zu tauschen:

```
($abc, $xyz) = ($xyz, $abc);
```

Viele andere Programmiersprachen benötigen für so eine Operation noch eine temporäre Variable, die man sich auf diese Art und Weise sparen kann.

Noch verwirrender wird es, wenn man negative Werte benutzt, wenn man auf Werte des Arrays zugreift:

```
$dashinterletzte = $gieseler3[-2];
```

Hier wird auf den vorletzten Wert des Arrays zugegriffen.

Um die Anzahl der Elemente eines Arrays zu erhalten, kann man den Befehl `scalar` folgendermaßen nutzen:

```
$elements = scalar(@array);
```

Ein Array kann komplett gelöscht werden, indem man ihm eine leere Liste zuweist:

```
$elements = ();
```

### 4.1.3 Hashes

Hashes werden auch "Assoziative Arrays" genannt, was etwas klarer werden lässt, um was es sich dabei handelt. Ein Hash ist eine ungeordnete Listen, bei der der Zugriff auf die einzelnen Elemente durch Skalare erfolgt. Das hört sich jetzt vielleicht etwas seltsam und abstrakt an, kann aber sehr nützlich sein.

Schreibt man zum Beispiel ein Programm, in dem die Wochentage eine Rolle spielen, so steht man vor dem Problem, dass sich mit Zahlen (1 für Montag, 2 für Dienstag usw.) zwar gut rechnen lässt, bei der Ausgabe aber möglicherweise die ausgeschriebenen Wochentage besser aussehen.

Dieses Problem lässt sich elegant mit dem folgenden Hash lösen:

```
%tage = ("1", "Montag", "2", "Dienstag", "3", "Mittwoch", "4", "Donnerstag",
"5", "Freitag", "6", "Sonnabend", "7", "Sonntag");
```

Es wird eine Liste erzeugt, in der der erste Wert mit dem zweiten, der dritte Wert mit dem vierten usw. assoziiert ist. In einem Programm kann nun mit Zahlen gerechnet werden. Bei der Ausgabe wird einfach wie folgt verfahren:

```
$a = 4;                         #In einem echten Programm wäre das natürlich ein
                                #errechneter Wert!
$b = $tage{$a};
print "Heute ist $b!\n";
```

Es würde "Heute ist Donnerstag!" ausgegeben. Sehr vorteilhaft ist hier, dass Perl Skalare bei Bedarf automatisch in den gebrauchten Datentyp umwandelt. Obwohl die Schlüssel im Hash als Strings abgelegt sind, kann das Programm mit Zahlen rechnen. Vor der Auflösung des Namen des Wochentages muss keine explizite Typumwandlung erfolgen, `$tage{$a}` funktioniert trotzdem.

Da die Erzeugung eines Hashes als lange Liste mit wachsender Größe unübersichtlich werden kann, gibt es eine Schreibweise, die einem hilft, sich in seinem eigenen Code nicht völlig zu verheddern.

```
%acronyms = (
      "BOFH" => "bastard operator from hell",
      "DLRG" => "Deutsche Lebens-Rettungs-Gesellschaft",
      "TSV"  => "Turn- und Sportverein",
      "TINC" => "The (International) Noise Conspiracy"
);
```

Hier ist auf den ersten Blick klar, welcher Schlüssel (der linke String) zu welchem Element (dem rechten String) gehört.

Dass in den Beispielen in diesem Kapitel nur Zeichenfolgen als Schlüssel verwendet wurden, bedeutet nicht, dass nicht auch Zahlen als Schlüssel verwendet werden können. Im ersten Beispiel wäre es tatsächlich einfacher, Integerwerte statt Strings zu verwenden (also die Anführungszeichen wegzulassen), aber ich wollte nochmal auf die Nützlichkeit der automatischen Typumwandlung hinweisen.

Um dem Hash einen weitern Wert zuzuweisen, muss lediglich eine neue Assoziation definiert werden:

```
$acronyms{"RATM"} = "Rage Against The Machine“;
```

Um ein Hash zu löschen, weist man ihm einen leere Menge zu:

```
%acronyms = ();
```

### 4.1.4 use strict

Bisher wurden Variablen einfach benutzt, ohne sie vorher zu deklarieren. Das ist sehr bequem, kann jedoch auch zu Problemen führen. Wenn in einem Programm die Variable `$test` benutzt wird und später wieder darauf zugegriffen wird, man sich aber vertippt und stattdessen die Zeile

```
print $tset;
```

in das Programm einfügt, wird immer ein leerer Text ausgegeben, auch wenn `$test` zuvor ein Wert zugewiesen wurde, da ja gar nicht auf diese Variable, sondern auf eine andere, leere (ich gehe davon aus, dass `$tset` zuvor kein Wert zugewiesen wurde) zugegriffen wird. Hier nochmal ein kurzes Programm, das diesen Fehler enthält:

```
#!/usr/bin/perl
$test = "Beispieltext";
print $tset;
```

Wird das Programm ausgeführt, wird kein Text ausgegeben, aber es kommt auch zu keiner Fehlermeldung, da ja Variablen ohne vorherige Definition benutzt werden können. Solche Fehler zu finden ist sehr schwer und kann sehr viel Zeit in Anspruch nehmen. Man sollte daher auf etwas Bequemlichkeit verzichten und dafür darauf bestehen, dass Variablen stets deklariert werden müssen. Dies kann man Perl mitteilen, indem man `use strict` verwendet. Nun müssen alle Variablen, die in einem Programm benutzt werden, deklariert werden. Dies geschieht mittels `my` vor dem ersten Auftreten einer neuen Variablen. Das vorherige Beispiel würde dann also so aussehen:

```
#!/usr/bin/perl
use strict;
my $test = "Beispieltext";
print $tset;
```

Führt man diesen Code aus, erhält man eine Fehlermeldung:

```
Global symbol "$tset" requires explicit package name at ./programm.pl line 4.
```

Man erhält also einen Hinweis darauf, dass man eine Variable benutzt hat, die man vorher nicht deklariert hat und wo man diese Variable benutzt hat. Dies erleichtert die Fehlersuche enorm und hilft, viele unnütze Fehler zu vermeiden.

`use strict` hat noch weitere Vorteile[13]. Wichtig ist an dieser Stelle anzumerken, dass mit `my` deklarierte Variablen lediglich in dem Block sichtbar sind, in dem sie deklariert wurden. Da ich bisher Blöcke noch nicht angesprochen habe, werde ich mich hier mit dieser kurzen Bemerkung begnügen und an geeigneter Stelle näher auf diese Tatsache eingehen.

## 4.2 Kontrollstrukturen

Bereits in Kapitel 4.1.1 bin ich kurz auf Bool'sche Wahrheitswerte eingegangen. Diese können durch so genannte Kontrollstrukturen ausgewertet werden. Umgangssprachlich handelt es sich dabei schlicht und einfach um Wenn-Dann-Beziehungen:

13 http://www.cs.cf.ac.uk/Dave/PERL/node139.html

Wenn der Kaffee fertig ist, schalte die Kaffeemaschine aus.
Wenn der Kaffee fertig ist, schalte die Kaffeemaschine aus, sonst schaue dem blauen Balken zu.
Klaue dem Nachbarn den Gartenzwerg, es sei denn jemand sieht dich dabei.

Ganz so einfach ist es in Perl nicht, aber auch nicht viel schwerer. Eine Kontrollstruktur, die in jeder Programmiersprache vorhanden ist, ist das `if`. Es überprüft eine Bedingung und führt, wenn diese erfüllt (TRUE) ist, eine Anweisung aus.

```
if(<BEDINGUNG>) {<ANWEISUNG>;}
```

In den geschweiften Klammern kann mehr als eine Anweisung stehen, wenn dazwischen immer ein Semikolon gesetzt wird. Damit haben wir auch gleich das Konzept der Befehlsblöcke (das ist nämlich das, was zwischen den geschweiften Klammern steht) erschlagen. Eine Anweisung kann jeder beliebige Perl-Befehl sein. Zwar kennen wir bis jetzt nur `print` und ein paar mathematische Operatoren, aber das reicht bereits, um sich etwas unter einer Anweisung vorstellen zu können.

Bedingungen sollten intuitiv zwar klar sein, aber wie formuliert man eine Bedingung in Perl? Wichtig dafür sind die Vergleichsoperatoren. Mit ihnen kann man Dinge formulieren wie "Ist $a größer als 5?". Bei den Vergleichsoperatoren ist wichtig zu bemerken, dass für es für Zahlen andere Operatoren gibt, als für Zeichenfolgen. Eine Übersicht bietet die folgende Tabelle.

| Vergleich | Numerisch | String | Rückgabe |
|---|---|---|---|
| Gleich | == | eq | TRUE, wenn $a gleich $b ist |
| Ungleich | != | ne | TRUE, wenn $a und $b nicht gleich sind |
| Kleiner als | < | lt | TRUE, wenn $a kleiner als $b ist |
| Größer als | > | gt | TRUE, wenn $a größer als $b ist |
| Kleiner gleich | <= | le | TRUE, wenn $a kleiner oder gleich $b ist |
| Größer gleich | >= | ge | TRUE, wenn $a größer oder gleich $b ist |

$a ist dabei stets das linke, $b das rechte Argument, also z.B. `$a < $b`.

Umgangssprachlich sind Dinge wie "Ist der Pudel geföhnt und das Kamel gestriegelt?" formulierbar, wo es darauf ankommt, dass mehre Bedingungen zu einer zusammengefasst werden können. In Perl lassen sich mehrere Bedingungen durch logische Operatoren verknüpfen.

| Beispiel | Name | Ergebnis |
|---|---|---|
| $a && $b | Und | TRUE, wenn $a und $b TRUE sind |
| $a || $b | Oder | TRUE, wenn $a oder $b oder beide TRUE sind |
| ! $a | Nicht | TRUE, wenn $a FALSE ist |

| Beispiel | Name | Ergebnis |
|---|---|---|
| $a and $b | Und | TRUE, wenn $a und $b TRUE sind |
| $a or $b | Oder | TRUE, wenn $a oder $b oder beide TRUE sind |
| not $a | Nicht | TRUE, wenn $a FALSE ist |

Mit diesen Operatoren kann man beliebige Bedingungen formulieren. So soll im folgenden Beispiel ein Text ausgegeben werden, wenn $a nicht kleiner als 53, aber auch nicht größer als 56 ist. Dies lässt sich auf mehrere Arten ausdrücken. Im Zweifelsfall sollte man diejenige wählen, die andere Personen, die den Code lesen, am wenigsten verwirrt.

Hier ein paar Beispiele:

```
if(!($a<53) && !($a>56)){print "Hurra!";}
if(($a>=53) && ($a<=56)){print "Hurra!";}
if(!(($a<53) || ($a>56))){print "Hurra!";}
if(($a==53) || ($a==54) || ($a==55) || ($a==56)){print "Hurra!";}
```

Obwohl die letzte Lösung hier die übersichtlichste ist, wird sie doch sehr unpraktisch, wenn die Grenzen weiter auseinander liegen und sie versagt völlig, falls es sich bei $a nicht um eine ganze Zahl handelt.

Wenn nicht nur im Erfolgsfall ($a liegt zwischen 53 und 56) eine Ausgabe erscheinen soll, könnte man nun die gleiche Abfrage nur negiert nochmal stellen. Eins von beidem muss ja schließlich stimmen. Perl hat aber (wie die meisten Programmiersprachen) eine elegantere Lösung zu bieten und die sieht so aus:

```
if(($a>=53) && ($a<=56)){print "Hurra!";}
else {print "Das war wohl nichts!";}
```

Wenn es noch mehr Möglichkeiten gibt, hilft `elsif`:

```
if ($username eq "root") {print "Hallo, mein Gebieter!";}
elsif ($username eq "nina") {print "Herzlich Willkommen!";}
else {print "Tach...";}
```

Neben dem `if` gibt es noch noch das `unless` , das wie ein `if` mit invertierter Bedingung funktioniert.

```
unless ($benutzer eq "root") {print "Dazu hast du kein Recht!";}
```

Den Befehl `elsunless` gibt es jedoch nicht.

## 4.3 Schleifen

Schleifen sind Konstruktionen, die einen bestimmten Programmteil normalerweise mehrmals abarbeiten.

Eine sehr einfache Schleife ist die `while`-Schleife. Allerdings muss man bei ihr auch aufpassen, dass sie nicht zu einer Endlosschleife wird. Die `while`-Schleife wird so lange wiederholt, wie eine bestimmte Bedingung erfüllt ist. ("Wenn der Topf nicht voll ist, schäle noch eine Kartoffel.")

```
$a = 10;
while ($a < 100) {print "$a ist kleiner als 100!";
                  $a++;}
```

Dieses Beispiel lässt sich weiter vereinfachen, indem man die Inkrementierung der Variablen $a gleich in der Bedingung vornimmt:

```
$a = 10;
while ($a++ < 100) {print "($a-1) ist kleiner als 100!";}
```

Dies hat den Vorteil, dass man gleich zu Beginn der Schleife die Voraussetzung schafft, dass die Bedingung auch erfüllt werden kann. Es muss allerdings die Schleife etwas angepasst werden, da $a sofort nach der Auswertung der Bedingung inkrementiert wird und somit zu dem Zeitpunkt, zu dem der Block in den geschweiften Klammer ausgeführt wird, um den Wert eins größer ist, als in der Variante vorher.

Wie im Fall von `if` und `unless` gibt es auch hier eine ähnliche Funktion, die jedoch die Bedingung negiert. Sie heißt `until`.

```
$a = 10
until ($a >= 100)
{
      print "$a ist kleiner als 100!";
      $a++;
}
```

Ein anderes Beispiel, diesmal mit einer Zeichenfolge:

```
$a = "";
until ($a eq "aaa") {$a .="a";}        #$a.="a" entspricht $a=$a."a"
```

Eine andere Art der Schleife ist die `for`-Schleife. Man kann sich ihre Syntax zwar ein wenig schlechter merken, dafür hat sie aber den Vorteil, den Startwert der Schleife, eine Bedingung und eine Inkrementierung oder Dekrementierung bereits zwingend zu erwarten. Eine Schleife, die die Zahlen von 0 bis 100 in Zehnerschritten ausgibt, sieht so aus:

```
for ($i = 0; $i <= 100; $i += 10) {print $i."\n";}
```

Möchte man die Werte absteigend ausgegeben bekommen, kann man die Schleife folgendermaßen benutzen:

```
for ($i = 100; $i >= 0; $i -= 10) {print $i."\n";}
```

Die vierte häufig benutzte Schleife in Perl ist die `foreach`-Schleife. Sie dient dazu, eine Menge von Skalaren, die z.B. in einem Array gespeichert sind, abzuarbeiten.

Möchte man jedes Element eines Arrays ausgeben, so geht man folgendermaßen vor:

```
foreach $element (@array) {print $element;}
```

Es wird also zuerst `$array[0]` in `$element` geschrieben und ausgegeben, dann `$array[1]`usw., bis die Schleife am Ende des Arrays `@array` ankommt.

## 4.4 99 Bottles of Beer

Mit dem Wissen aus den letzten Kapiteln sind wir nun in der Lage, ein Programm zu schreiben, das den Text des Liedes "99 Bottles of Beer on the Wall" als HTML-Seite ausgibt.

```
#!/usr/bin/perl
use strict;

print"Content-type: text/html\n\n";
print"<html>\n</head><title>99 bottles of beer</title></head>\n";
print"<body>\n<h1>99 Bottles of Beer on the Wall</h1>\n<p>";

for(my $bottles = 99;$bottles > 0;$bottles--)  #Von 99 herunterzählen nach 1
{
  if($bottles == 99)                            #Sonderfall 99
      {print"99 bottles of beer on the wall, 99 bottles of ".
      "beer.<br>\nTake one down, pass it around, ";}

  elsif($bottles > 1)                           #98 bis 2
      {print"$bottles bottles of beer on the wall. $bottles ".
      "bottles of beer on the wall, $bottles bottles of ".
      "beer.<br>\nTake one down, pass it around, ";}

  else                                          #Sonderfall 1
      {print"1 bottle of beer on the wall. 1 bottle of beer on ".
      "the wall, 1 bottle of beer.<br>\nTake one down, pass it ".
      "around,  no bottles of beer on the wall.";}
}
print"</p>\n</body>\n</html>\n";
```

Alle in diesem Programm vorkommenden Elemente sollten aus früheren Kapiteln bekannt sein. Neu ist lediglich die Verwendung der Punkte zur Verkettung über mehrere Zeilen, um den Code übersichtlich zu halten. Aber auch sie Verkettung selbst ist bereits bekannt.

## 4.5 Unterroutinen

Wie bereits früher angekündigt, kommen wir nun zu den Unterroutinen. Da es sehr ineffizient ist, den gleichen Programmblock immer wieder in das Programm zu schreiben, wenn man ihn braucht,

gibt es Unterroutinen. Außerdem erlauben es Unterroutinen, Programme übersichtlicher zu gestalten.

Eine Unterroutine muss vor der Benutzung natürlich deklariert werden. Wo im Programm das geschieht, ist völlig egal, man sollte nur einem klar erkennbaren Schema folgen, damit der Code lesbar bleibt.

```
sub fehler {print"Es ist ein Fehler aufgetreten!\n";}

sub addition
{
      $summe = $_[0] + $_[1];
      return $summe;
}
```

Die Erste Unterroutine heißt `fehler` und gibt lediglich eine Text aus. Sie wird aufgerufen durch den Befehl `&fehler;` oder `fehler();` (Ja, ohne vorgestelltes Kaufmannsund!).

Die zweite Routine wird mit `&addition($a,$b);` oder `addition($a,$b);` aufgerufen. Die Variablen in der Klammer werden als Array mit dem etwas seltsamen Namen `@_` übergeben, das dann in der Routine zerlegt wird, damit die einzelnen Werte bearbeitet werden können.

Damit der Rückgabewert der Unterroutine nicht verloren geht, sollte man ihn entweder direkt ausgeben (wenn das alles ist, was mit ihm passieren soll) oder in einer Variablen speichern.

```
print "Die Summe von $a und $b beträgt ".&addition($a, $b)."!";
$c = &addition($a, $b);
```

## 4.6 Variablen in Blöcken

In den vorangegangenen Beispielen wurden stets Codeteile in geschweifte Klammern eingefasst benutzt, die nach einer bestimmten Bedingung oder einem Sprung ausgeführt wurden. Diese in geschweiften Klammern eingefassten Codeteile nennt man Blöcke.

Wie ich in 4.1.4 schon angemerkt habe, sind in Blöcken definierte Variablen auch nur dort sichtbar. Das kann von Vorteil sein, wenn man in mehreren Unterroutinen Variablen mit gleichem Namen verwenden möchte. Dies kann z.B. die Übersichtlichkeit erhöhen, wenn diese Variable stetes in ähnlichem Zusammenhang benutzt wird.

Ein Beispiel soll dies verdeutlichen:

```
#!/usr/bin/perl
use strict;
my $ergebnis;

$ergebnis = &addition(3,2);
```

```
print $ergebnis."\n";

$ergebnis = &subtraktion(3,2);
print $ergebnis."\n";

sub addition
{
	my $summe = $_[0] + $_[1];
	return $summe;
}

sub subtraktion
{
	my $summe = $_[0] - $_[1];
	return $summe;
}
```

Dieses kleine Programm beinhaltet zwei Unterprogramme und in beiden Unterprogrammen wird die Variable `$summe` benutzt. Die Besonderheit ist hier, dass die Variable erst im Unterprogramm deklariert wird und daher auch nur dort sichtbar ist. Wir der Variablen also in einem Unterprogramm eine Wert zugewiesen, so ist dies für das andere Unterprogramm nicht relevant. Wäre die Variable außerhalb der Unterprogramme, z.B. gleich am Anfang des Programms, unter `$ergebnis` deklariert worden, könnte sie von einem Unterprogramm geändert werden und in einem anderen Unterprogramm könnte darauf zugegriffen werden. Dies kann je nach Anwendungszweck erwünscht sein oder auch nicht. Wo man eine Variable deklariert, hängt also von ihrer Funktion im Programm ab.

## 5. Übergabe und Verarbeitung von Daten

Jetzt können wir zwar HTML-Seiten mit Perl erzeugen, das Ganze ist aber noch nicht besonders dynamisch, denn wir können keine Werte an Programme übermitteln. Es gibt zwei Möglichkeiten, Daten an den Server zu übermitteln. Die eine Möglichkeit heißt GET, die andere POST.

GET wird vom Browser an den Server gesendet, wenn dieser Daten anfordert. Gut zu erkennen ist das aus dieser Zeile, die aus der Logdatei eines Servers in einem lokalen Netzwerk entnommen ist:

```
192.168.1.3 - - [22/Feb/2005:18:50:00 +0100] "GET /~mnause/index.html HTTP/1.1"
200 73 "-" "Opera/8.0 (X11; Linux i686; U; en)"
```

Der Browser auf dem Client mit der IP-Nummer 192.168.1.3 hat vom Server die Datei index.html im für die Veröffentlichung im Web vorgesehenen Ordner des Benutzers `mnause` angefordert. Die 200 nach der Anfrage zeigt an, dass die Abfrage funktioniert hat und die Daten an den Client gesendet wurden.

Die GET-Anfrage kann nun auch dafür benutzt werden, Daten an ein Skript zu senden. Dazu wird einfach eine Anfrage der Form http://www.server.com/skript?arg1=blah&arg2=blub gesendet. Der Vorteil dieser Lösung ist, dass man solche Links sehr einfach in seine Webseiten einbauen kann und Besucher der Seite auch Bookmarks auf solche Links setzen können. Ein Nachteil ist, dass in der Adresszeile des Browsers natürlich alle Daten zu sehen sind, die an den Server gesendet werden. Unter Umständen ist das nicht wünschenswert, wenn es sich z.B. um vertrauliche Daten handelt.

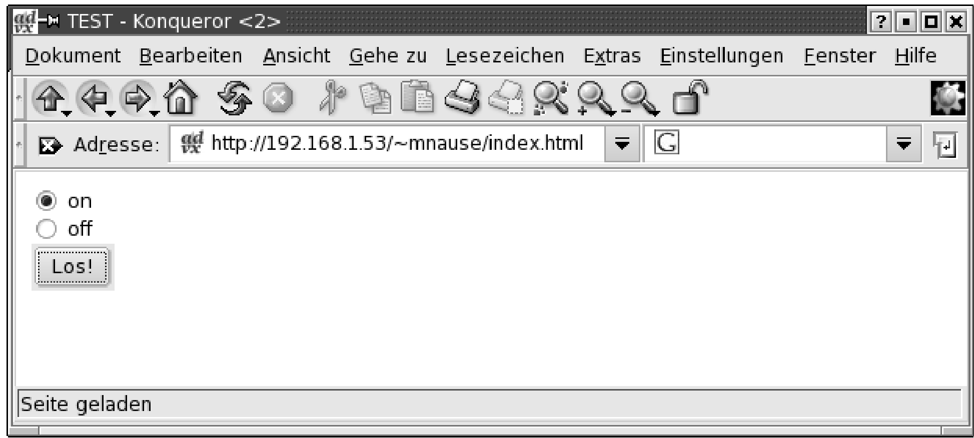
Die andere Möglichkeit ist, die Daten per POST zu versenden. Diese Methode ist geeignet für größere Datenmengen, die nicht in der Adresszeile des Browsers auftauchen sollen. POST kann man mit Formularen benutzen, die man per HTML erzeugt. Formulare können aber auch GET nutzen.

### 5.1 Formulare

Formulare werden in HTML stets nach dem folgenden Schema beschrieben:

```
<form action="cgi-bin/skript.pl" method="post">
<input type="radio" checked name="arg1" value="on"> on
<br>
<input type="radio" name="arg1" value="off"> off
<input type="submit" value="Los!">
<br>
</form>
```

Eingebettet in ein HTML-Dokument sieht das dann so aus:




Zu Beginn des Formulars wird angegeben, wohin die Daten später geschickt werde sollen. Hier wird auch festgelegt, ob die Daten per POST oder per GET verschickt werden sollen. In diesem Beispiel ist es POST. GET müsste nicht angegeben werden, da das dem Standard entspricht. Es dient allerdings der Übersichtlichkeit, dies trotzdem zu tun.

Dann folgen in den `<input>`-Tags Möglichkeiten Daten einzugeben. Hier kann zwischen den Möglichkeiten on und off gewählt werden.

Danach folgt ein Button, der das Formular absendet. Dieser Button muss sich nicht am Ende des Formulars befinden, er kann sich überall zwischen `<form>` und `</form>` befinden.

Statt einer Auswahl gibt es noch einige andere Eingabemöglichkeiten.

```
<input size="x" maxlength="y" name="Elementname" value="blahblah">
```

Hier handelt es sich um ein einzeiliges Eingabefeld, das x Zeichen breit ist und Eingaben bis zu einer Maximallänge von y annimmt. Es hat den Namen Elementname. Der Name ist wichtig, damit das Skript später die eingegebenen Daten auch einem bestimmten Eingabefeld zuordnen kann. Mit `value` kann man einen bestimmten Text in das Feld schreiben, der vom Benutzer des Formulars dann geändert oder übernommen werden kann. Lässt man `value` weg, bleibt das Eingabefeld leer.

```
<input type="password" size="x" maxlength="y" name="Elementname">
```

Dieses Eingabefeld ist dem darüber sehr ähnlich, allerdings werden eingegebene Werte nur als Sternchen angezeigt.

```
<textarea cols="x" rows="y" name="Elementname">
Vorbelegung (optional)
</textarea>
```

Hier handelt es sich um eine Eingabebox, die x Zeichen breit und y Zeichen hoch ist.

```
<select name="Elementname">
<option value="opt1" selected> Option1
<option value="opt2"> Option2
<option value="opt3"> Option3
</select>
```

Dies ist eine Auswahlliste, in der man aus 3 Möglichkeiten auswählen kann. Option1 ist bereits ausgewählt. `<select multiple name="Elementname">` erzeugt eine Auswahlliste, in der mehrere Möglichkeiten ausgewählt werden können. Der Wert in `value` wird an das Skript gesendet.

```
<input type="radio" name="arg1" value="on" checked> on
<input type="radio" name="arg1" value="off"> off
```

Wurde bereits im Beispiel weiter oben benutzt. Hier ist zu beachten, dass, wenn sie gleiche Namen haben, sie sich gegenseitig ausschließen, also nur eine Option angewählt werden kann.

```
<input type="checkbox" name="Name" value="val1"> Text
```

Eine Checkbox ist ein kleines Kästchen, das man ankreuzen kann. Mehrere Checkboxes können den gleichen Namen haben und gehören dann zu einer Gruppe. Checkboxes schließen sich nicht gegenseitig aus.

```
<input type="submit" value="Beschriftung">
<input type="reset" value="Beschriftung">
```

Der Submitbutton ist schon im kurzen Beispiel weiter oben vorgekommen. Der Resetbutton setzt alle Werte des Formulars auf ihren Ursprungswert zurück. Wird `value` weggelassen, wird der Button mit einem durch den jeweiligen Browser vorgegebenen Standardtext beschriftet.

## 5.2 Übernahme der Daten durch ein Skript

Nun wissen wir, wie man Daten an einen Server schickt, dort müssen sie dann durch ein Programm weiterverarbeitet werden. Bevor ich näher darauf eingehe, hier ein wichtiger

**Hinweis zum Thema Sicherheit:**
Daten die von außen an ein Perl-Programm übergeben werden, sollte man immer als gefährlich ansehen. Je nachdem, was in einem Programm mit den Daten geschieht, kann es möglich sein, dass ein kreativer Benutzer Daten ausliest, die nicht für ihn bestimmt sind, Daten durcheinander bringt oder sogar Daten löscht. Dies muss gar nicht vorsätzlich geschehen. Bei ungeschickter Programmierung kann es geschehen, dass falsche Eingaben nicht abgefangen werden und zu unvorhergesehenen Ereignissen führen. Man sollte also stets überlegen, welche Daten eingegeben werden könnten und welche Daten gebraucht werden. Grundsätzlich sollte man bei Programmen, die Daten von außen erhalten, den so genannten Taintmodus aktivieren. Dies erfolgt, indem die erste Zeile des Perl-Programms folgendermaßen abgewandelt wird:

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

Allein der das Anhängen von `-T` hilft, einige für Perl offensichtlich gefährliche Eingaben abzufangen. Darauf sollte man sich jedoch nicht verlassen.[14]

### 5.2.1 Übernahme aus GET

Bei der Übergabe von Daten durch GET (durch einen Link oder ein Formular) werden die Daten in einer so genannten Umgebungsvariable gespeichert. Umgebungsvariablen enthalten alle möglichen Informationen, die Perl nutzen kann, wie beispielsweise den Ort des Programms auf der Festplatte, den Namen des Servers, die IP-Adresse des Servers. In der Variable `QUERY_STRING` finden sich die Daten, die übergeben werden, wenn sie per GET übertragen wurden.[15] Man kann diese Umgebungsvariable leicht abfragen und sie einer Variablen zuweisen:

```
$input = $ENV{'QUERY_STRING'};
```

Dummerweise ist die Eingabe codiert, damit sie richtig verschickt werden kann. Die Anfrage

```
input.pl?argument1=Tony der Koch&argument2=Motörhead
```

Liefert dem Skript `input.pl` die Eingabe

```
argument1=Tony%20der%20Koch&argument2=Mot%F6rhead
```

Bevor man diese Eingabe vernünftig bearbeiten kann, muss sie also decodiert werden. Dies macht das folgende Programm:

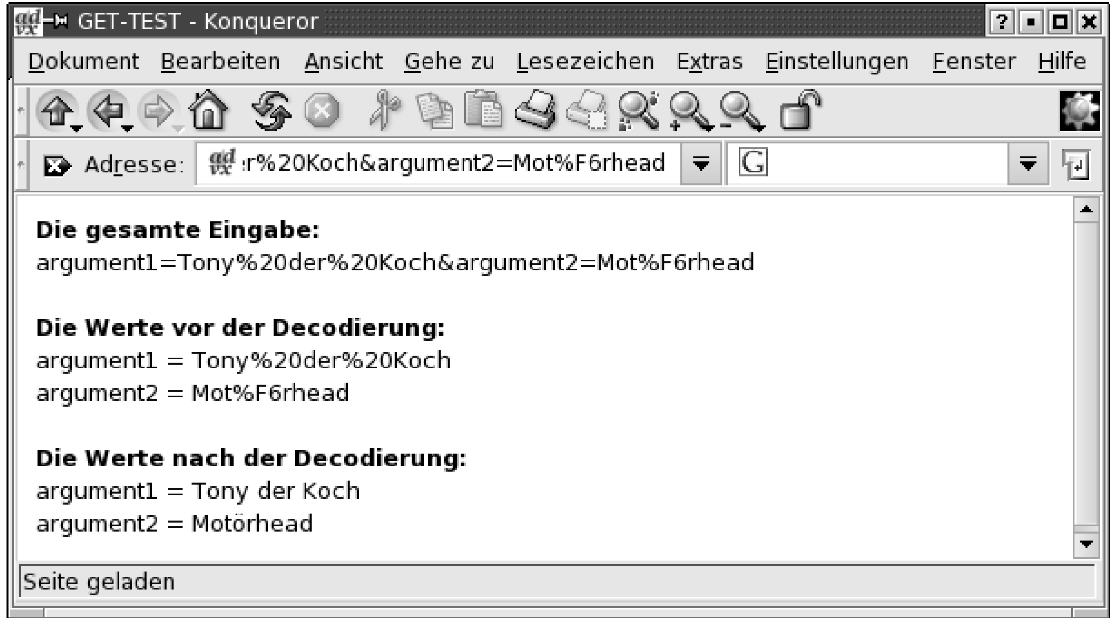
```
#!/usr/bin/perl -T
#Programm zum Testen von Eingaben über GET

use strict;
use CGI::Carp qw(fatalsToBrowser);

#Hier werden erstmal alle Variablen deklariert
my $input;
my @data;
my $i;
my @pair;
my $pair;
my @key;
my $key;
my @value;
```

14 Weiterführende Informationen:
http://gunther.web66.com/FAQS/taintmode.html
http://www.xwolf.de/artikel/cgisec.shtml
http://ds.ccc.de/081/insertion-attacks

15 Übersicht bekannter CGI-Umgebungsvariablen:
http://de.selfhtml.org/cgiperl/intro/umgebungsvariablen.htm#uebersicht

```
$input = $ENV{'QUERY_STRING'};

#Eingabe bei & trennen und in @data schreiben
@data = split /&/, $input;

#Anfang der HTML-Seite und Ausgabe von $input
print"Content-type: text/html\n\n";
print"<html><head><title>GET-TEST</title></head>\n";
print"<body><p>";
print"<b>Die gesamte Eingabe:</b><br>$input<br><br>\n";

#Trennen bei = und in @key und @value schreiben
$i = 0;
foreach $pair (@data)
{
      ($key[$i], $value[$i]) = split /=/, $pair;
      $i++;
}

#Ausgabe vor dem Dekodieren
print"<b>Die Werte vor der Decodierung:</b><br>";
$i = 0;
foreach $key (@key)
{
      print"$key[$i] = $value[$i++]<br>\n";
}

#Decodieren von @key und @value
$i = 0;
foreach $key (@key)
{
      $key[$i] =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
      $value[$i] =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
      $i++;
}

#Ausgabe nach dem Dekodieren
print"<br><b>Die Werte nach der Decodierung:</b><br>";
$i = 0;
foreach $key (@key)
{
      print"$key[$i]  = $value[$i++]<br>\n";
}

#Ende des HTML-Dokuments
print"</p></body></html>";
```

Wird mit `input2.pl?argument1=Tony der Koch&argument2=Motörhead` das obige Skript aufgerufen, dann sieht das im Browser so aus:




Das Programm sollte nachvollziehbar sein, bis auf den Ausdruck

```
$key =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
```

Hier treten gleich zwei neue Dinge auf. Das eine ist das =~ Zeichen, das die Variable `$key` an den rechten Ausdruck bindet. Was damit gemeint ist, wird klarer, wenn man weiß, dass der rechte Ausdruck eine Ersetzung ist.

Eine Ersetzung sieht folgendermaßen aus:

```
s/zu_ersetzen/ersetzt_durch/
```

Der Wert in `zu_ersetzen` wird dabei durch `ersetzt_durch` ersetzt. Zur Verdeutlichung hier ein Beispiel:

```
$key = "Mario";
$key =~ s/M/W/;
print $key;
```

Als Ausgabe erhalten wir das Wort "Wario".

Auf Konstruktionen wie diese werden wir in Kapitel 5.4 noch ausführlicher zu sprechen kommen, denn sie sind äußerst mächtig und nützlich.

### 5.2.2 Übernahme aus POST

Werden Dateien mit POST übergeben, geht man ganz ähnlich vor, jedoch erhält man die Daten nun nicht aus der Umgebungsvariablen `QUERY_STRING`, sondern aus der Standardeingabe.

```
read(STDIN, $input, $ENV{'CONTENT_LENGTH'});
```

Es werden aus der Standardeingabe so viele Zeichen gelesen, wie die Eingabe lang ist (die Länge erhält man aus der Umgebungsvariablen `CONTENT_LENGTH`) und diese werden dann in `$input` gespeichert.

Die restliche Verarbeitung erfolgt wie bei der Übergabe mit GET.

## 5.3 CGI.pm

Da die obige Art und Weise, die gewünschten Daten aus der Eingabe zu ziehen, sehr umständlich ist, gibt es das Modul CGI.pm, das einem die Arbeit sehr erleichtert. CGI.pm ist seit Perl 5.004 ein Standardmodul, muss also nicht extra installiert werden, sondern steht zur Verfügung, sobald Perl installiert ist. Sehr schön ist auch, dass man sich keine Gedanken machen muss, ob die Daten per GET oder per POST übergeben werden, es funktioniert beides ohne Änderungen.

Das Auslesen der Daten mit CGI.pm erfolgt durch:

```
#!/usr/bin/perl
#ggt.pl

use CGI;        #Zeigt an, dass CGI.pm benutzt wird.

$input= CGI::new();
```

Danach kann durch

```
$a = $query->param("Argument");
```

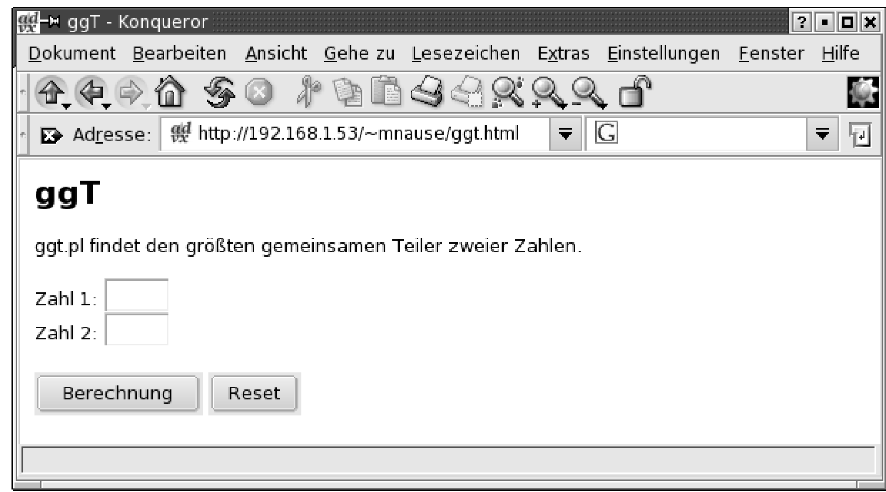
jeder gewünschte Wert einfach abgefragt werden. Dieser muss nicht weiter bearbeitet werden, denn er ist bereits fertig decodiert. Das folgende Beispiel soll die Arbeitsweise eines Programms, das CGI.pm benutzt, in Zusammenarbeit mit einem Formular zur Dateneingabe verdeutlichen. Das Programm errechnet mit dem euklidischen Algorithmus den größten gemeinsamen Teiler (ggT) zweier Zahlen.[16]

Zuerst das Formular zur Eingabe:

```
<html>
<head><title>ggT</title></head>
<body>
<h1>ggT</h1>
<p>ggt.pl findet den gr&ouml;&szlig;ten gemeinsamen Teiler zweier Zahlen.</p>
<form action="cgi-bin/ggt.pl" method="post">
<p>Zahl 1: <input size="4" maxlength="3" name="zahl1">
<br>
Zahl 2: <input size="4" maxlength="3" name="zahl2">
```

16 ggT: http://de.wikipedia.org/wiki/GGT
Euklidischer Algorithmus: http://de.wikipedia.org/wiki/Euklidischer_Algorithmus

```
<br><br>
<input type="submit" value="Berechnung">
<input type="reset" value="Reset">
</p>
</body>
</html>
```




Wie am `<form>`-Tag zu erkennen ist, wird POST zur Datenübertragung gewählt.

Hier das Skript das die Eingabe bearbeitet (nun auch mit `use strict`):

```
#!/usr/bin/perl -T
#ggt.pl

use strict;
use CGI::Carp qw(fatalsToBrowser);
use CGI;        #Zeigt an, dass CGI.pm benutzt wird.

#Variablen deklarieren!
my $query;
my $m;
my $n;
my $r;

$query = CGI::new();
$a = $query->param("zahl1");
$b = $query->param("zahl2");

print CGI::header();
print CGI::start_html("ggT Ergebnis");
print CGI::h1("ggT");
print CGI::p("Der ggt von $a und $b ist ".&ggt($a,$b)."!");
print "<br><a href=\"".$ENV{'HTTP_REFERER'}."\">Nochmal!</a>!";
print CGI::end_html();

sub ggt
{
        #Argumente auslesen
        $m = $_[0];
        $n = $_[1];
```

```
	#Sicherstellen, dass $m>0, $n>0
	if($m < 0){$m *= -1};
	if($n < 0){$n *= -1};

	#Sicherstellen, dass $m>=$n
	if($n>$m){($m, $n)=($n, $m);}

	if(($m != 0)&&($n != 0))
	{
		#ggT errechnen mit Euklid
		$r=1;
		while($r!=0)
		{
			$r = $m % $n;
			$m = $n;
			$n = $r;
		}
	}
	return $m;
}
```

Zusätzlich zu CGI.pm wurde hier noch mit `use CGI::Carp qw(fatalsToBrowser);` bewirkt, dass eventuelle Fehlermeldungen an den Browser des Benutzers gesendet werden. Dies ist sehr sinnvoll, da es eine eventuelle Fehlersuche sehr erleichtert.

Als Erstes wird dann im eigentlichen Programm mit `$query = CGI::new();` ein Objekt angelegt, das die übertragenen Daten enthält. Aus `$query` können nun mit `$a = $query->param("zahl1");` die gewünschten Werte ausgelesen werden. Dabei muss in die Anführungszeichen in der Klammer der Name des Wertes eingegeben werden, der vom Formular übergeben wurde. In `$a` steht dann der Wert, der durch den Benutzer in das Formular eingetragen wurde.

CGI.pm kann aber noch mehr. All die lästigen Dinge, wie das Schreiben des Headers (`"Content-type: text/html"`), sowie der ersten Zeilen (bis es endlich richtig los geht) und den Abschluss des HTML-Dokuments kann CGI.pm für einen übernehmen.

Dies geschieht durch

```
print CGI::header();
print CGI::start_html("Titel der Seite");
#Hier kommt der Rest der Seite hin!
print CGI::end_html();
```

Hier sollte schon erkennbar sein, nach welchem Schema die Aufrufe der Methoden erfolgen: Zuerst wird mitgeteilt, dass ein Befehl aus dem Modul CGI.pm benutzt wird (`CGI::`), dann folgt der Befehl und in den Klammern können noch Argumente übergeben werden, wie z.B. der Titel der HTML-Seite, der dann automatisch in den Header geschrieben wird, aber auch so genannte META-Tags können auf einfache Art und Weise hinzugefügt werden. Da wir dies in dieser kurzen Einführung nicht benötigen, werde ich diese Möglichkeiten hier aber nicht weiter ausführen.

Gut zu wissen ist auch, dass sich HTML-Tags dank CGI.pm leicht abkürzen lassen und man sich so keine Sorgen um das korrekte Schließen und das Vermeiden unkorrekter Verschachtelungen machen muss. Dies wurde zum Beispiel hier benutzt:

```
print CGI::p("Der ggt von $a und $b ist ".&ggt($a,$b)."!");
```

Nach einem String mit Interpolation wird das Ergebnis der Berechnung der Subroutine &ggt angehängt und daran nochmal ein Ausrufungszeichen angefügt.

Man kann aber auch wie gewohnt normales HTML per `print` ausgeben lassen:

```
print "<br><a href=\"".$ENV{'HTTP_REFERER'}."\">Nochmal!</a>!";
```

Hier wurde nochmals Gebrauch von einer Umgebungsvariablen gemacht welche hilft, einen Link auf die Seite zu setzen, von der aus `ggt.pl` aufgerufen wurde.[17]

Da das <a href>-Tag Anführungszeichen benutzt, in die die Adresse der verlinkten Seite eingesetzt wird, diese aber mit den Anführungszeichen des Perl-Befehls `print` in Konflikt geraten würden, wird vor jedes Anführungszeichen, das durch den Perl-Befehl ausgegeben werden soll, ein Backslash (\) gesetzt. Dies muss u.a. auch bei Zeichen geschehen, die Sonst für Verwirrung sorgen würden, wie das `@` und das `$`, die ja Variablennamen vorangestellt sind.

CGI.pm ist sehr hilfreich und kann die Programmierung sehr erleichtern. Die für einfache Programmierung wichtigsten Befehle sind mit diesem Kapitel abgedeckt, allerdings lohnt es sich, sich weiter mit den Möglichkeiten dieses Moduls vertraut zu machen. Dazu empfehle ich:

http://de.selfhtml.org/cgiperl/module/cgi.htm und das Buch "CGI – kurz & gut" (Einzelheiten dazu sind Anhang A. zu entnehmen).

## 5.4 Pattern Matching

Ein großes Problem, dass das oben angesprochene Programm zur Berechnung des größten gemeinsamen Teilers hat, ist die Tatsache, dass es sämtliche dreistelligen Werte als Argumente annimmt und an die Subroutine zur Berechnung weiterreicht. Das können neben Zahle auch Buchstaben oder Sonderzeichen sein. In diesem Fall mag das nicht besonders problematisch erscheinen, allerdings sind leicht Situationen denkbar, in denen unerwartete Eingaben zu ernsthaften Problemen führen können.

Wir könnten zwar durch hässliche Javascriptkonstruktionen im Formular dafür sorgen, dass nur Zahlen eingegeben werden könne, allerdings könnte man dies leicht umgehen, indem man entweder Javascript deaktiviert oder gar nicht erst das Formular nutzt, sondern die Daten auf andere Arten an

17 Und das völlig ohne hässliche Javascript-Krücken oder ähnliche unsichere Lösungen! :-)

das Programm schickt. Man muss also im Programm dafür sorgen, dass nur solche Werte verarbeitet werden, die auch zulässig sind.

Sehr hilfreich hierbei ist das so genannte Pattern Matching. Weiter oben haben wir gesehen, wie an ein Programm geschickte Daten dekodiert wurden. Es handelte sich um eine äußerst hässliche Codezeile mit scheinbar zusammenhangslosen Zeichen. Wie wir aber auch schon gesehen haben, kann man bei genauerer Betrachtung überraschenderweise doch verstehen, was dort passiert. Dieses Kapitel kann das Thema Pattern Matching und reguläre Ausdrücke nur anreißen, jedoch entsteht hoffentlich ein Eindruck davon, wie mächtig diese Werkzeuge sind.[18]

Die Befehle zum Pattern Matching in Perl sind

```
m/Suchausdruck/
```

```
s/Suchen/Ersetzen/
```

```
tr/a-z/A-Z/
```

Im ersten Fall (Suchen) wird ein bestimmter Suchausdruck gesucht und im zweiten Fall (Suchen und Ersetzen) wird ein Wort durch ein anderes ersetzt. Im dritten Fall (Transliteration) werden in einer Liste aufgezählte Zeichen (hier a bis z) durch Zeichen in einer anderen Liste (hier A bis Z) ersetzt. Im oben angegebenen Beispiel würden alle Kleinbuchstaben durch Großbuchstaben ersetzt.

Um diese Befehle auch anzuwenden, müssen sie, wie wir auch schon gesehen haben, durch `=~` (oder die Verneinung `!~`) an Variablen gebunden werden. Den Suchbefehl kann man leicht nutzen, um zu überprüfen, ob ein bestimmter Suchausdruck in einer Variablen enthalten ist:

```
if ($satz =~ m/Leberwurst/) {print"Knackwurst";}
```

Wird im Skalar `$satz` das Wort Leberwurst gefunden, wird das Wort Knackwurst ausgegeben. Das `m` kann übrigens auch weggelassen werden, was ich im weiteren Verlauf auch tun werden.

Ersetzungen sind genauso einfach:

```
$Liebesbrief =~ s/Gertrude/Conzuela/;
```

Hier werden in `$Liebesbrief` alle Gertrudes durch Conzuelas ersetzt.[19]

```
$Verwirrung = "abc";
$Verwirrung =~ tr/ac/xz/;
```

---

18 An dieser Stelle sei auf das Kapitel "Mustererkennung" im Perl Kochbuch und "Pattern-Matching" in "Programmieren mit Perl" verwiesen, die diesem Thema erheblich mehr Raum widmen können. Wer ganz Tief in das Thema einsteigen möchte, dem sei das Buch "Reguläre Ausdrücke" von Jeffrey E. F. Friedl ans Herz gelegt.

19 Ob das Schreiben von Liebesbriefen mit dem Computer ratsam ist, soll an dieser Stelle außer Acht gelassen werden.

In `$Verwirrung` würde nun die Zeichenkette `"xbz"` stehen.

Diese Möglichkeiten sind schon ganz nett, aber sehr unflexibel. In unserem Beispiel mit dem ggT wollen wir ja wissen, ob eine Eingabe nur aus Zahlen besteht. Um nicht völlig umständlich allerhand Fälle durchprüfen zu müssen, wenden wir für unser Problem lieber Werkzeuge an, die Perl uns bietet. In unserem Fall muss geprüft werden, ob die Eingabe andere Zeichen als 0 bis 9 enthält. In anderen Fällen müsste mit der Ausgabe einer Fehlermeldung und dem Abbruch des Programms reagiert werden.

Die von uns benötigte Abfrage kann leicht mit Hilfe der Zeichenklasse `\D` (alle Nicht-Ziffern) realisiert werden:

```
if ($zahl =~ /\D/){print"Fehler! Bitte nur positive Zahlen eingeben!\n";}
else {#Hier passiert der Rest!}
```

Perl bietet einige dieser Zeichenklassen, die extrem nützlich sein können. Hier ein paar wichtige:

| Code | Zeichen |
|---|---|
| \n | Zeilenvorschub (newline) |
| \r | Wagenrücklauf (carriage return) |
| \t | Tabulator |
| \f | Seitenvorschub (formfeed) |
| \e | Escape |
| \d | Ziffer (digit, die Zahlen von 0 bis 9) |
| \D | Nicht-Ziffer (alles außer 0 bis 9) |
| \w | Wortzeichen (enthält nur a bis z, A bis Z, _ und 0 bis 9) |
| \W | Nicht-Wortzeichen |
| \s | Whitespaces (\t \n \r oder \f) |
| \S | Nicht-Whitespaces |

Es ist auch möglich, eigene reguläre Ausdrücke zu definieren. Gleich zu Beginn soll noch darauf hingewiesen werden, dass einige Zeichen in regulären Ausdrücken nur mit vorangestelltem Backslash verwendet werden können, da es sich bei ihnen um so genannte Metazeichen handelt, die den regulären Ausdruck beschreiben. Diese Zeichen sind: `\ | ( ) [ ] { } ^ $ * + ? .`

Möchte man überprüfen, ob ein Wort aus einer Gruppe von Wörtern in einem Text vorkommt, so kann man das so tun:

```
/wort1|wort2|wort3/
```

Taucht eins dieser Wörter auf, so ist die Bedingung erfüllt. Dies kann man auch ausnutzen, um herauszufinden, ob in einem Text die Zeichenfolge "Karl Schmidt" oder "Gustav Schmidt" vorkommt:

```
/(Karl|Gustav) Schmidt/
```

Durch die Klammern wird eine Gruppe von veroderten Ausdrücken gebildet, an die ein Leerzeichen und der Name Schmidt angehängt wird.

Soll ein Wort mehrmals nacheinander gefunden werden, kann man Quantoren nutzen. So findet

```
/(la){3}/
```

das Wort "lalala", während

```
/la{3}/
```

Das Wort "laaa" findet. Dies lässt sich auch noch weiter verfeinern. Die notwendigen Quantoren sind in der folgenden Tabelle zusammengefasst.

| Quantor | Wertebereich |
|---|---|
| {m,n} | Mindestens m Mal, höchstens n Mal |
| {m,} | Mindestens m Mal |
| {m} | Genau m Mal |
| * | Null Mal oder öfter {0,} |
| + | Mindestens einmal {1,} |
| ? | Keinmal oder einmal {0,1} |

Will man einen String nach einer Buchstabenkombination durchsuchen und dabei soll es nicht auf Groß/Kleinschreibung ankommen, kann man einen Modifier einsetzen:

```
$wort =~ /$kombination/i
```

Möchte man in einem String jedes Vorkommen eines Wortes ersetzen, ist ein einfaches

```
$satz =~ s/ALT/NEU/
```

nicht geeignet, da dies nur das erste Vorkommen des Wortes `ALT` durch `NEU` ersetzt. Um jedes Vorkommen zu ersetzen, benutzt man ebenfalls einen Modifier:

```
$satz =~ s/ALT/NEU/g
```

# 6. Datenquellen

Bisher wurden Daten übergeben und verarbeitet, aber es handelte sich dabei stets nur um Daten, die vom Browser an den Server übergeben wurden. Um den Zugriff oder die Speicherung bestehender Daten geht es in diesem Kapitel.

## 6.1 Dateien

Oft möchte man Daten speichern, sie durch Benutzer eingegeben wurden oder auf schon gespeicherte Daten zugreifen. Dies geht relativ einfach, wenn diese Daten auf der Festplatte des Rechners gespeichert sind, auf dem sich auch das Skript befindet.

Um Dateien zu lesen oder zu schreiben, öffnet man diese zuerst, schreibt oder liest dann und schließt die Datei danach wieder.

```
open(DATEI, "Dateiname");     #Öffnet zum Lesen
open(DATEI, "<Dateiname");    #Öffnet ebenfalls zum Lesen
open(DATEI, ">Dateiname");    #Erzeugt Datei und schreibt hinein
open(DATEI, ">>Dateiname");   #Hängt an existierende Datei an
```

`DATEI` ist der so genannte Dateihandle, ein frei wählbarer Name, durch den die Datei im Programm eindeutig benannt wird. `Dateiname` enthält den Dateinamen und gegebenenfalls den Pfad zur zu öffnenden Datei. Dadurch, dass eine Datei immer nur zu einem bestimmten Zweck (lesen, schreiben oder anhängen) geöffnet wird, verringert sich die Gefahr, dass man versehentlich Aktionen durchführt, die so nicht gewollt sind. Beim Öffnen zum Schreiben ist allerdings zu beachten, dass eine eventuell schon vorhandene Datei gleichen Namens überschrieben wird.

Es ist aber auch möglich eine Datei zum Schreiben und Lesen zu öffnen.

```
open(DATEI, "+<Dateiname");   #Öffnet zum Lesen und Schreiben
open(DATEI, "+>Dateiname");   #Ebenso, allerdings ist eine Leseaktion erst
                              #nach dem Schreiben möglich.
open(DATEI, "+>>Dateiname");  #Lesen und Schreiben, hängt an exist. Datei an
```

Da es beim Öffnen einer Datei zu Problemen kommen kann, sollte man stets den Fall abfangen, dass eine Datei nicht geöffnet werden kann. Man kann sich dann eine Fehlermeldung und den Grund des Fehlers ausgeben lassen.

```
open(DATEI, "$name") or die "Fehler beim Öffnen von $name: $!\n";
open(DATEI, "$name") || die "Fehler beim Öffnen von $name: $!\n";
```

Beide Zeilen sind äquivalent. Die Variable `$!` ist eine Spezialvariable von Perl und enthält einen kurzen Text, der den Grund des Fehlers beschreibt.

Ein weiteres Problem, das umgangen werden muss, ist der gleichzeitige Zugriff von verschiedenen

Programmen oder verschiedenen Instanzen des gleichen Programms auf eine Datei. Dies kann zu Problemen führen, wenn eine Datei von einem Programm geschrieben oder gelesen wird, während sie bereits durch ein anderes Programm geschrieben wird. Es ist also sinnvoll, vor Schreibaktionen die entsprechende Datei zu sperren. Hierzu gibt es die Möglichkeit, eine Datei für den Zugriff für andere Programme zu sperren. Zu beachten ist dabei, dass das Betriebssystem, auf dem Perl läuft, dies auch unterstützen muss. Bei Windows 95/98 und älteren Versionen von MacOS ist dies nicht der Fall, MS WindowsNT/2k/XP, neuere Versionen von MacOS und unixoide Betriebssysteme dürften aber keine Probleme bereiten.

```
#!/usr/bin/perl

use strict;

my $name = "datei.txt";

open(DATEI, ">$name") or die "Fehler beim Öffnen von $name: $!\n";
print"Verlange exklusiven Lock...\n";
flock(DATEI, 2) or die "Kein Lock bekommen: $name $!\n";
print DATEI "DATENDATENDATEN\n";
close DATEI;
```

Hier wurde die Datei `$name` zum Schreiben geöffnet. Daraufhin wurde die Datei zum schreiben durch andere Programme gesperrt. Nach dem Schreiben wurde die Datei ordnungsgemäß geschlossen und damit auch die Zugriffssperre aufgehoben. Soll die Sperre explizit aufgehoben werden, kann dies mit `flock(DATEI, 8)` geschehen.

Möchte man eine Datei nicht überschreiben, sondern Daten anhängen, kann es passieren, dass in dem Zeitraum zwischen Öffnen der Datei und Anfordern der Sperre, ein anderes Programm etwas an die Datei angehängt hat. Dies könnte zu einem Fehler führen, daher sollte in diesem Fall das Beispiel wie folgt erweitert werden:

```
#!/usr/bin/perl

use strict;

my $name = "datei.txt";

open(DATEI, ">>$name") or die "Fehler beim Öffnen von $name: $!\n";
print"Verlange exklusiven Lock...\n";
flock(DATEI, 2) or die "Kein Lock bekommen: $name $!\n";
seek DATEI, 0, 2;
print DATEI "DATENDATENDATEN\n";
close DATEI;
```

Der `seek`-Befehl sorgt dafür, dass auch wirklich am Ende der Datei eingefügt wird. (Es wird an das nullte Zeichen von hinten gesprungen, also wirklich am Ende eingefügt.)

Für den Fall, dass eine Datei nur gelesen werden können soll, aber nicht von einem anderen

Programm geschrieben werden darf, kann `flock(DATEI, 4)` eingesetzt werden.

Das Wichtigste, das Lesen aus Dateien haben wir bisher nicht beachtet. Um eine Datei lesen zu können, muss sie ebenfalls geöffnet werden. Danach wird sie zeilenweise in ein Array geschrieben.

```
#!/usr/bin/perl

use strict;

my $name = "datei.txt";
my @Daten;

open(DATEI, $name) or die "Fehler: $!\n";
while(<DATEI>)
 {
  push(@Daten,$_);
 }
close(DATEI);
```

Das Array `@Daten` kann nun wie gewohnt verarbeitet werden.

Mit diesem Wissen können wir nun ein Programm erstellen, mit dessen Hilfe sich Studenten zu einer Klausur anmelden können.  Das Skript soll sicher stellen, dass Daten nicht doppelt eingegeben werden und es soll die Daten in eine Datei schreiben, die später die einfache Weiterverarbeitung ermöglicht. Ein solches Format ist CSV. Dabei werden Daten durch Komma getrennt in eine Zeile geschrieben. Dann erfolgt ein Zeilenumbruch und der nächste Datensatz wird eingefügt. Solche Dateien lassen sich in viele Programme, zum Beispiel in MS Excel oder OpenOffice importieren.

Unsere Datei soll so aussehen:

```
Müller,Michael,Informatik,1111111
Schrader,Sebastian,Wirtschaftsinformatik,2222222
Becker,Bastian,Informatik,3333333
```

Das Eingabeformular könnte so aussehen:

```
<html><head><title>Anmeldung</title></head>
<body><h1>Anmeldung</h1>
<form action="cgi-bin/anmeldung.pl">
Name: <input type="text" size="20" name="name"><br>
Vorname: <input type="text" size="20" name="vorname"><br>
<input type="radio" name="studiengang" value="Info">Informatik<br>
<input type="radio" name="studiengang" value="Winfo">Wirschaftsinformatik<br>
Matrikelnummer: <input type="text" size="9" name="matrikelnummer"><br>
<input type="submit" value="OK!">
</form>
</body></html>
```

Das Skript, das die Daten entgegen nimmt, muss nun testen, ob identische Daten bereits gespeichert sind und wenn nicht, die neuen Daten an die schon gespeicherten Daten anhängen.

```perl
#!/usr/bin/perl -T
use strict;
use CGI::Carp qw(fatalsToBrowser);
use CGI;

my $datei = "dateiname.csv";
my $bool = 1;
my $query = CGI::new();
my $n = $query->param("name");
my $vn = $query->param("vorname");
my $sg = $query->param("studiengang");
my $mn = $query->param("matrikelnummer");
my $a;
my $b;
my $c;
my $d;

open(DATEI, ">>$datei") or die "Argl!";  #Falls Datei nicht existiert, wird
close DATEI;                             #sie hier erzeugt.

open(DATEI, "<$datei") or die "Kann $datei nicht öffnen: $!\n";

#Überprüfung, ob Daten schon in Datei. Wenn ja -> $bool=0 (FALSE)
while(<DATEI>)
{
        chop;           #entfernt Zeilenende
        ($a,$b,$c,$d) = split(/,/, $_);
        if(($a eq $n)&&($b eq $vn)&&($c eq $sg)&&($d eq $mn)){$bool=0;}
}
close DATEI;

#Wenn Daten noch nicht in Datei sind, werden sie hineingeschrieben.
if($bool)
{
        open(DATEI, ">>$datei") or die "Kann $datei nicht öffnen: $!\n";
        print DATEI $n.",".$vn.",".$sg.",".$mn."\n";
        close DATEI;
}
close(DATEI);

#Ausgabe der HTML-Seite
print CGI::header();
print CGI::start_html("Anmeldung $vn $n");

#Ausgabe, falls Daten gespeichert wurden
if($bool)
{
        print CGI::h1("Anmeldung erfolgreich!");
        print CGI::p("Folgende Daten wurden gespeichert:<br>");
        print CGI::p("$vn $n, $sg, $mn");
}

#Ausgabe, falls Daten nicht gespeichert wurden
else
{
        print CGI::h1("Keine Anmeldung erfolgt:");
        print CGI::p("Sie sind schon angemeldet!");
}
print CGI::end_html();
```

Dieses Programm funktioniert, hat aber den Nachteil, dass Eingaben, die ein Komma enthalten die Datei durcheinander bringen können. Man müsste sich also entweder für ein anderes Trennzeichen entscheiden, wenn das Komma in der Eingabe zwingend erforderlich ist oder die zu speichernden Daten vor dem Speichern auf das Vorkommen eines Kommas überprüften und das Komma durch ein anderes Zeichen oder eine Kombination von Zeichen ersetzen.

## 6.2 LWP::Simple

Die Erweiterung LWP::Simple[20] erlaubt es, per URL auf entfernte Daten zuzugreifen. Die Benutzung ist sehr einfach, der Nutzen aber sehr hoch.

```
#!/usr/bin/perl -T
use strict;
use LWP::Simple;

my $content;

$content = get("http://www.heise.de/newsticker/heise.rdf")
or die "Couldn't get it!" unless defined $content;

print $content;
```

Dieser mögliche Anfang eines Perl-Programms holt eine Datei von einem entfernten Server und speichert sie in der Variablen `$content`. Die Daten könnten nun gespeichert oder weiterverarbeitet werden.

## 6.3 Datenbanken

Daten in Datenbanken zu speichern ist oft bequemer, sicherer und schneller, als sich selbst um das Ablegen und Wiederfinden zu kümmern, da man viele Dinge, die man sonst selbst programmieren müsste, der Datenbanksoftware überlassen kann. Perl stellt für die Arbeit mit Datenbanken die Erweiterung DBI zur Verfügung.

Der Vorteil von DBI ist, dass es eine einheitliche Schnittstelle zu einer Vielzahl von verschiedenen Datenbanken bietet. Es gibt zum Beispiel die Möglichkeit, auf die freie und für kleinere Webanwendungen gern genutzte Datenbank MySQL, aber auf auch Systeme wie Postgres, Oracle, Informix oder Ingres zuzugreifen.

Das folgende Programm stellt eine Verbindung zur Datenbank namens `test` her und schließt sie danach wieder.

```
#!/usr/bin/perl
use strict;
use DBI;
```

20 http://search.cpan.org/dist/libwww-perl/lib/LWP/Simple.pm

```
my $database="test:localhost";
my $username="";
my $password="";
my $dbh;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password)
       or die "Database connection not made: $DBI::errstr";
$dbh->disconnect();
```

Nach dem Einbinden des Moduls DBI durch `use DBI` werden mehrere Variablen definiert. In der Ersten befindet sich der Name der Datenbank, zu der verbunden werden soll und nach dem Doppelpunkt der Rechner, auf dem sich die Datenbank befindet. Befinden sich Skript und Datenbank auf dem gleichen Rechner, wie dies meistens der Fall ist, setzt man als Rechnernamen `localhost` ein. Danach folgen Benutzername und Passwort, mit denen sich zur Datenbank verbunden werden kann.

Mit der dann folgenden Zeile wird die Verbindung zur Datenbank hergestellt. Hier wird unter anderem angegeben, dass es sich um eine MySQL-Datenbank handelt. Möchte man ein Skript, das man ursprünglich für den Zugriff auf eine MySQL-Datenbank geschrieben hat, dahingehend ändern, dass das Skript auch mit einer anderen Datenbank zusammenarbeitet, muss auf jeden Fall hier der Name des so genannten Treibers für die entsprechende Datenbank eingetragen werden.[21] Der Verbindung zur Datenbank wird hier `$dbh` zugewiesen. Die folgende Zeile illustriert, wie alle Aktionen ausgeführt werden, an denen eine Datenbank beteiligt ist. Hier wird in der letzten Zeile die Verbindung zur Datenbank wieder ordnungsgemäß geschlossen.

Relationale Datenbanken, wie sie heute überwiegend eingesetzt werden, speichern Daten in Tabellen ab. Hier ein Beispiel für eine Tabelle:

| ***ID*** | ***Name*** | ***Vorname*** | ***Spitzname*** | ***Kinder*** | ***Einkommen*** |
|---:|---|---|---|---:|---:|
| 1 | Schmidt | Sebastian | Sebi | 2 | 1000 |
| 2 | Müller | Michael | Michi | 0 | 1321 |
| 3 | Schulze | Sebastian | Schu | 1 | 1500 |
| 4 | Meier | Maren | Mari | 4 | 2000 |

Die ***ID*** soll eine fortlaufende Nummer sein, die jede Person eindeutig identifiziert, als Datentyp bietet sich Integer (Ganzzahl) an. ***Name***, ***Vorname*** und ***Spitzname*** sind Strings. MySQL benötigt eine Angabe, wie lang die Strings sind, die gespeichert werden. Hier sollte ein Wert von 20 ausreichend sein. Es soll keine Person den gleichen Spitznamen haben, wie eine andere. ***Kinder*** und ***Einkommen*** sind wieder Ganzzahlen und sollen als Integer in die Datenbank übernommen werden. Der Name der Tabelle soll **Namen** sein.

21 Mehr Informationen zu den verfügbaren Treibern und deren Eigenschaften findet sich im Anhang B des Buches "Programmierung mit Perl DBI" von Alligator Descartes und Tim Bunce.

Um nun diese Tabelle in der Datenbank anzulegen, muss die Datenbank geöffnet werden, die Tabelle erzeugt werden und die Datenbank dann wieder geschlossen werden. Dies geschieht durch das folgende Skript:

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password)
        or die "Database connection not made: $DBI::errstr";

$sth = $dbh->prepare("CREATE TABLE Namen
                (ID             int auto_increment not null,
                Name            varchar(20) not null,
                Vorname         varchar(20) not null,
                Spitzname       varchar(20),
                Kinder          int,
                Einkommen       int,
                primary key     (ID),
                unique          (Spitzname)
                );") or die "Fehler bei der Vorbereitung: ".$dbh->errstr()."\n";


$sth->execute() or die "Fehler bei der Ausführung: ".$sth->errstr()."\n";

$dbh->disconnect();
```

Durch den Befehl `prepare` wird eine Anfrage an die Datenbank vorbereitet. Sollte bei der Ausführung ein Fehler gefunden werden, so wird der Befehl nicht ausgeführt. Mittels `errstr()` kann auf eine Fehlerbeschreibung zugegriffen werden.

Die nun leere Datenbank kann nun mit Werten gefüllt werden. Das Einfügen der Daten ist vom Perl-Code her identisch mit dem Erzeugen einer Tabelle, lediglich der SQL-Befehl ändert sich. (Auf die Abfrage möglicher Fehler wurde hier nur der höheren Übersichtlichkeit wegen verzichtet, grundsätzlich sollte man mögliche Fehler immer abfangen.)

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);
```

```
$sth = $dbh->prepare("INSERT INTO Namen
                      (Name, Vorname, Spitzname, Kinder, Einkommen)
                      values ('Schmidt', 'Sebastian', 'Sebi', 2, 1000)
                     ;");

$sth->execute();

$dbh->disconnect();
```

Für die Spalte ***ID*** wird kein Wert angegeben, da für diesen Wert festgelegt ist, dass er automatisch hochgezählt wird.

Nun soll die Datenbank komplett ausgegeben werden. Hierzu wird der SQL-Befehl `select * from Namen` benutzt, der alle Spalten der Tabelle `Namen` auswählt.

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;
my $ID;
my $Name;
my $Vorname;
my $Spitzname;
my $Kinder;
my $Einkommen;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);

$sth = $dbh->prepare("SELECT * FROM Namen;");
$sth->execute();
while(($ID,$Name,$Vorname,$Spitzname,$Kinder,$Einkommen)=$sth->fetchrow_array())
{
     print "$ID,$Name,$Vorname,$Spitzname,$Kinder,$Einkommen\n";
}
$sth->finish();

$dbh->disconnect();
```

Nachdem die Anfrage gestellt wurde, wird eine Schleife so lange durchlaufen, bis keine Daten mehr aus `$sth` geliefert werden. Statt alle Werte sofort per `print` auszugeben, kann man sie zum Beispiel auch in einem Array speichern, um sie später zu verarbeiten.

Möchte man nicht alle Zeilen, sondern nur Name und Vorname, ausgeben lassen und auch nur diejenigen, die eine bestimmte Bedingung erfüllen (hier: mindestens ein Kind), so muss das Programm folgendermaßen abgewandelt werden:

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);

$sth = $dbh->prepare("SELECT Name,Vorname FROM Namen WHERE Kinder > 0;");
$sth->execute();
while(($Name,$Vorname)=$sth->fetchrow_array())
{
     print "$Name,$Vorname\n";
}
$sth->finish();

$dbh->disconnect();
```

Es gibt noch weit ausgefeiltere Möglichkeiten, mittels SQL auf Datenbanken zuzugreifen, die aber den Rahmen dieses Skripts sprengen würden.[22]

Um Datensätze zu ändern, benutzt man den SQL-Befehl `update`.

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);

$sth = $dbh->prepare("UPDATE Namen SET Kinder = 3 WHERE Spitzname = 'Sebi';");

$sth->execute();

$dbh->disconnect();
```

Hier wurde nun für die Person mit dem Spitznamen **Sebi** die Kinderzahl auf drei gesetzt. Da in der Tabelle `Namen` der Spitzname per Definition beim Anlegen der Tabelle eindeutig ist, wird nur die Zeile mit der ***ID*** 1 geändert. Hätten mehrere Personen den Spitznamen **Sebi**, würde bei allen die Kinderzahl auf 3 gesetzt.

Als letztes sollen hier noch Möglichkeiten vorgestellt werden, einzelne Datensätze und komplette

22 Weitere Informationen zu SQL speziell mit MySQL: http://dev.mysql.com/doc/mysql/de/reference.html

Tabellen zu löschen.

Zuerst sollen alle Zeilen aus der Tabelle `Namen` gelöscht werden, bei denen ***Kinder*** größer als 1 ist:

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);

$sth = $dbh->prepare("DELETE FROM Namen WHERE Kinder > 1;");

$sth->execute();

$dbh->disconnect();
```

Um komplette Tabellen zu löschen, benutzt man den `drop`-Befehl:

```
#!/usr/bin/perl
use strict;
use DBI;

my $database="test:localhost";
my $username="";
my $password="";
my $dbh;
my $sth;

$dbh = DBI->connect("DBI:mysql:$database", $username, $password);

$sth = $dbh->prepare("DROP TABLE Namen;");

$sth->execute();

$dbh->disconnect();
```

Nach Ausführung dieses Skripts sind alle Daten der Tabelle `Namen` und die Tabelle selbst beseitigt.

## 7. Beispielprogramme

An dieser Stelle folgen einige Beispielprogramme, die als Anregung für eigene Entwicklungen dienen sollen. (Bisher gibt es hier nur ein Beispielprogramm, aber es sollen noch weitere folgen.)

### 7.1 Ein einfacher Zugriffszähler

Eine Anwendung für Perl, die recht beliebt ist, ist ein Counter für Webseiten. Wenn alle Seiten durch ein Skript erzeugt werden, ist es natürlich am einfachsten, das erzeugende Skript entsprechend zu verändern, dass ein Zähler hochgezählt wird.

Hat man aber statische HTML-Seiten, so ist eine einfache Methode, die Zahl der Seitenzugriffe zu zählen, eine kleine durchsichtige Grafik im GIF-Format in der Größe 1 mal 1 Pixel in das HTML-Dokument einzubetten, wobei die Grafik durch ein Skript geliefert wird.

Die HTML-Seite könnte in etwa so aussehen:

```
<html>
<head><title>Counted</title></head>
<body>
<p>Hier steht ein Text!</p>
<img src="cgi-bin/counter.pl">
</body>
</html>
```

Wird nun die Seite aufgerufen, wird das Skript `counter.pl` aufgerufen, wobei erwartet wird, dass dieses ein Bild zurückgibt.

Das entsprechende Skript sieht folgendermaßen aus:

```
#!/usr/bin/perl
use strict;

my @count;

#Hier wird das Bild ausgegeben
print "Content-Type: image/gif\n\n";
open(DATEI,"bild.gif");
print while (<DATEI>);
close(DATEI);

#Auslesen des alten Counterstandes in die Variable $count[0]
open(DATEI,"count.txt");
while(<DATEI>){push(@count,$_);}
close(DATEI);

#Hier wird der Zähler um 1 erhöht und in count.txt geschrieben
open(DATEI,">count.txt") or die "Kann Datei nicht schreiben!";
flock(DATEI,2) or die "Kein Lock erhalten!";
print DATEI $count[0]+1;
close(DATEI);
```

Es muss natürlich sichergestellt werden, dass im Verzeichnis, in dem sich `counter.pl` befindet auch die Datei `bild.gif` liegt, die durch das Skript zurück gegeben werden soll. Die Datei `count.txt` wird nach dem ersten Durchlauf erzeugt, wenn sie noch nicht existiert und kann nun von einem anderen Programm ausgewertet werden.

Das Skript bietet viele Möglichkeiten zur Erweiterung. Statt lediglich die Anzahl der Zugriffe zu speichern, könnte man die IP-Adresse des aufrufenden Rechners speichern oder auch die Kennung des aufrufenden Browsers. Diese Informationen lassen sich aus Umgebungsvariablen extrahieren. Eine sehr gute Übersicht über die verschiedenen Umgebungsvariablen bietet SelfHTML: http://de.selfhtml.org/servercgi/cgi/umgebungsvariablen.htm#uebersicht

# Anhang

## A. Literaturempfehlungen

**Programmieren mit Perl** von Larry Wall, Tom Christiansen, Jon Orwant, Randal Schwartz
(2. Auflage, 2001, O'Reilly)
Das Standardwerk zum Thema Perl!

**Programmierung mit Perl DBI** von Alligator Descartes, Tim Bunce
(1. Auflage, 2000, O'Reilly)
DBI ist die Standard-Datenbankschnittstelle für Perl. Programmierung mit Perl DBI beschreibt die Architektur von DBI und erklärt, wie DBI-basierte Programme geschrieben werden.

**Perl Kochbuch** von Tom Christiansen, Nathan Torkington
(2. Auflage, 2004, O'Reilly)
Eine Umfassende Sammlung von Beispiele und Lösungen für Perl-Programmierer.

**Reguläre Ausdrücke** von Jeffrey E. F. Friedl
(2. Auflage, 2003, O'Reilly)
Eine ausführliche Einführung in die Welt der regulären Ausdrücke. Es wird besonders auf Perl, Java und .NET eingegangen.

**CGI – kurz & gut** von Martin Vorländer
(2. Auflage, 2003, O'Reilly)
Dieses Buch fasst die Umgebungsvariablen, CGI.pm, mod_perl, sowie SSI kurz und knapp zusammen.

**HTML - kurz & gut** von Jennifer Niederst
(2. Auflage, 2002, O'Reilly)
HTML-Referenz für Leute, die wissen was sie tun, aber stets eine Beschreibung sämtlicher HTML-Tags griffbereit haben möchten.

## B. Internetresourcen

### Webseiten

http://www.perl.com/ .................................................. Informationsportal zum Thema Perl

http://www.cpan.org/ .................................................. Quelle für Perl-Module

http://www.perl.org/ .................................................. Linksammlung zum Thema Perl

http://www.perl.com/perl/ .................................................. Perl-Homepage mit Downloadquellen für Apple, Windows und Linux

http://perl-seiten.privat.t-online.de/ .................................. deutschsprachige Online-Dokumentation

http://www.netzmafia.de/skripten/perl/index.html ............. Grundlagen CGI-Programmierung mit Perl – Tutorial von Prof. Jürgen Plate

http://www.cbkihong.com/index.pl?op=perltut&lang=e .... Perl 5 Tutorial - Tutorial von Bernard Chan (englisch)

http://de.selfhtml.org/ .................................................. HTML-Dateien selbst erstellen

http://de.selfhtml.org/servercgi/cgi/index.htm ................ Informationen zum Thema CGI (u.a. Umgebungsvariablen)

http://www.worldmusic.de/perl/ ....................................... FAQ der deutschsprachigen Perl-Newsgoups

### Newsgroups

de.comp.lang.perl.misc .................................................. deutschsprachige Gruppe zum Thema Perl

de.comp.lang.perl.cgi .................................................. deutschsprachige Gruppe zum Thema CGI

comp.lang.perl.misc .................................................. englischsprachige Gruppe zum Thema Perl

### Foren

http://www.perl.de/ .................................................. deutschsprachiges Forum zum Thema Perl

http://board.perl-community.de/ ....................................... deutschsprachiges Forum zum Thema Perl

## C. Software

http://filezilla.sourceforge.net/ ........................................... Sehr guter, freier FTP-Client (Windows)

http://www.gftp.org/ .......................................................... Freier FTP-Client (*NIX)

http://www.chiark.greenend.org.uk/~sgtatham/putty/ ....... Putty, ein kosteloser SSH-Client für Microsoft Windows

http://www.ipswitch.com/products/WS_FTP/ .................. Beliebter kommerzieller FTP-Clienet (Windows)

http://www.arachnoid.com/arachnophilia/ ........................ Editor, mit Syntaxhighlighting für zahlreiche wichtige Sprachen

http://httpd.apache.org/ ...................................................... Webserver für Linux und Windows

http://de.selfhtml.org/servercgi/server/allgemein.htm ..... Hinweise zur Installation von Apache unter Windows

http://www.apachefriends.org/ ......................................... Anbieter von XAMPP, einer sehr guten Zusammenstellung von Serversoftware