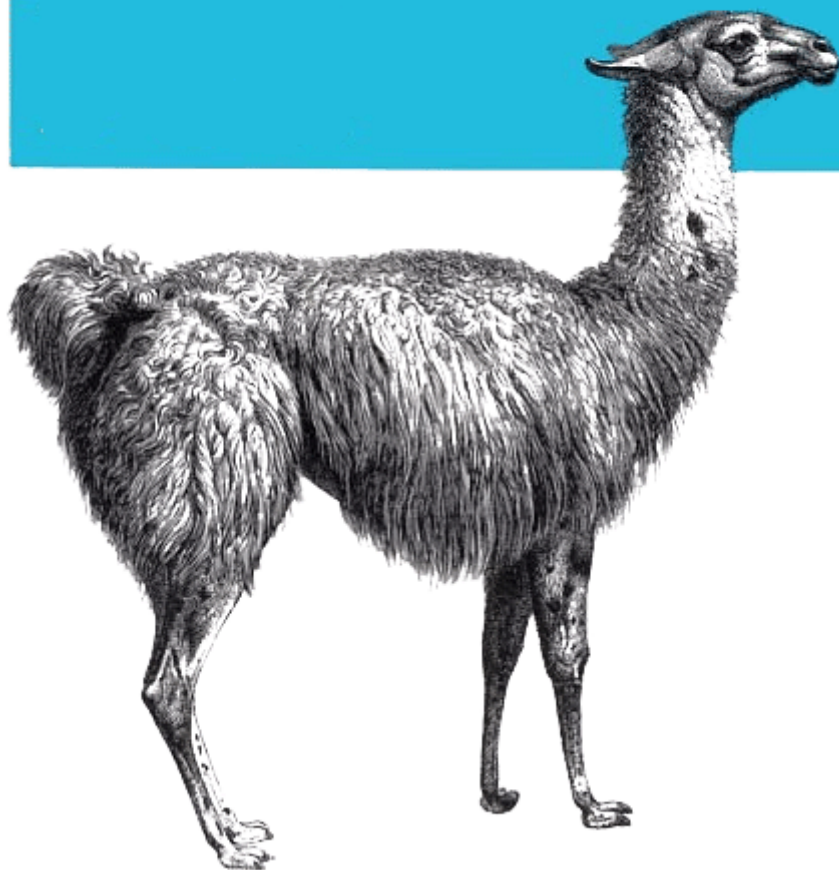


S P E Z I A L G E B I E T
Perl Programmieren



SPEZIALGEBIET AUS INFORMATIK:

PERL Programmieren:

1. Einführung

- a. Was ist Perl?
- b. Warum Perl?

2. Perl Programmieren

- a. das erste Perl Programm
- b. skalare Variablen
- c. Array Variablen
- d. Hashes (assoziative Arrays)
- e. Dateienverarbeitung
- f. Schleifen + Kontrollstrukturen
- g. Mustererkennung und Substitution
- h. Mehr String Funktionen
- i. Subroutinen
- j. Essentielles

3. Nachwort

- a. Referenz
 - b. Zusammenfassung
-

1.a Was ist Perl?

Perl steht für **P**ractical **E**xtraction and **R**eport **L**anguage, auf Deutsch übersetzt heißt es „Praktische Extraktions- und Report-Sprache“.

De facto ist dies eine ziemlich treffende Beschreibung von Perls besonderen Stärken: **Extraktion** für das Anschauen von Dateien und das **Herausziehen** wichtiger Teile; **Report** für das Generieren von Ausgaben und **Reports** über die gefundenen Informationen. Die Sprache ist praktisch, weil es viel leichter ist, diese Art von Programmen in Perl zu schreiben als in einer Sprache wie C.

Perl ist eine vollständige Programmiersprache, denn sie enthält die drei Grundstrukturen, die dafür benötigt werden:

1. Folgen
2. Maschen (Verzweigungen)
3. Schleifen

Fehlt eine dieser Strukturen, handelt es sich nicht um eine Programmiersprache. Beispiele sind HTML (hier fehlen alle drei Strukturen).

Entwickelt wurde Perl 1987 von Larry Wall. Es geht die Geschichte um, dass Larry gerade an einer Aufgabe arbeitete für die das Unix- Programm *awk* (das zu dieser Zeit populäre Extraktions- und Report-Programm) nicht ausreichte. ihm wurde aber klar, dass die Lösung seiner Aufgabe in einer Sprache wie C eine Menge Arbeit bedeuten würde. So erfand er eine eigene Skriptsprache - Perl.

Perl wurde frei an die Unix-Gemeinschaft gegeben, welche Perl dankbar aufgriff. Jahrelang war Perl die bevorzugte Sprache für Unix-Systemadministratoren und andere Unix-Programmierer. Sie brauchten eine flexible, schnell zu programmierende Sprache zur Lösung von Aufgaben, die für Shell-Skripting zu komplex waren und deren Lösung in Sprachen wie C einen deutlichen Mehraufwand bedeutet hätte. Es lag an seiner Popularität als Unix-Sprache, dass Perl sich auch als Web-Sprache zum Erstellen von CGI-Skripts durchsetzte - so gut wie alle Webserver liefen früher auf Unixsystemen. Mit CGI (*Common Gateway Interface*) konnte - und kann - man mit Formularen oder anderen „webseitigen“ Eingaben interaktiv Programme und Skripts auf dem Webserver ablaufen lassen. Perl

passte also wunderbar in diese Nische, und als in den letzten Jahren das WWW und CGI immer populärer wurden, wuchs auch das Interesse an Perl enorm.

Mit dem plötzlichen Anstieg von Perls Beliebtheit kümmert sich jetzt eine engverzahnte Gruppe von freiwilligen Programmierern um die Weiterentwicklung. Diese Programmierer, darunter auch Larry Wall selbst, arbeiten weiter am Quellcode von Perl, portieren ihn auf Nicht-Unix- Plattformen, koordinieren Bugfixes und geben die Perl Releases heraus. Perl gehört keiner einzelnen Organisation.

Die aktuelle Perl-Version ist Perl 5.8.2, veröffentlicht 1999. An einer v.6 wird bereits seit längerer Zeit fieberhaft gearbeitet. Seit Perl 5.005 unterstützt der Kern auch Windows.

1.b Warum Perl?

PHP oder Perl

Die Zahl der heute auf dem Markt erhältlichen Programmiersprachen ist enorm. Jede von ihnen hat ihre Vor- und Nachteile. Der wohl größte Konkurrent von Perl im Web ist allerdings PHP, besonders in Verbindung mit einer MySQL Datenbank. Der erste Unterschied zwischen diesen Programmiersprachen ist offensichtlich. PHP ist eine Programmiersprache, die praktisch ausschließlich in der Webprogrammierung eingesetzt wird und für diesen Zweck optimiert wurde. Das heißt, dass typische Probleme der Webprogrammierung mit PHP viel einfacher lösbar sind als mit Perl. Das Verschicken von Mail, der Upload von Dateien, der Zugriff auf Datenbanken gestaltet sich sehr viel einfacher. Viele Probleme, die in Perl nur über Referenzen gelöst werden können (z.B. Zugriff auf die Tabellenstruktur in einer Datenbank) können mit PHP einfacher gelöst werden, da hierfür eigene Funktionen zur Verfügung gestellt werden. Viele Funktionalitäten, die in Perl nur über das Einbinden von Modulen zur Verfügung stehen, sind Bestandteil von PHP selbst.

Allerdings verfügt Perl über etwa 3500 Module, welche es zu einer enorm mächtigen Sprache, die nicht auf die Programmierung im Internetumfeld beschränkt ist, machen. Es scheint im Moment so zu sein, dass im wissenschaftlichen Umfeld, z.B. in der Bioinformatik, eindeutig Perl dominiert, bei Websites wohl PHP.

Verwandtschaft

Die Schreibweise von Perl wurde nicht völlig neu erfunden, sondern aus bereits bewährten Sprachen abgeleitet. Dies bedeutet, dass sich sehr viele Konstrukte und Schreibweisen von Perl in anderen Sprachen wie C, C++, Java und C# wieder finden. Dies erleichtert einen Wechsel erheblich, da sehr viel Vorwissen gleich mitgenommen werden kann.

Anwendungsbereich

Perl hat im Gegensatz zu PHP ein enormes Spektrum an Anwendungsmöglichkeiten, während PHP allein auf das Umfeld im Web beschränkt ist. So lässt es sich als Allzweck- Skriptsprache für Systemadministratoren, als Horizonterweiterung für UNIX-Anwender, oder als Websprache perfekt einsetzen. Mit Perl lässt sich sinnvolle Arbeit sehr schnell erledigen.

einfaches Programmieren

Um ein Perl Script zu schreiben braucht man weder einen Perl-Compiler, der den Code wie bei einer Sprache wie C oder Java in ein anderes Format, etwa eine ausführbare Datei, verwandelt, noch eine integrierte Entwicklungsumgebung. Man braucht keinen Browser, der Perl unterstützt, noch ein eigenes Betriebssystem. Einzig und allein ein Perl- Interpreter ist notwendig, der kostenlos aus dem Internet herunter geladen werden kann.

Perl ist schnell zu programmieren

Perl ist eine Skriptsprache. Das bedeutet, dass Perl-Skripts reine Textdateien sind, die sofort ausgeführt werden, wenn Perl sie ablaufen lässt. Mit Perl kann man daher schneller und einfacher als in C erste Programme laufen lassen.

Perl ist leistungsfähig

In Perl sind viele komplexe Unix-Tools eingeflossen. Damit geht fast alles, was in einer anspruchsvollen Sprache wie C möglich ist, auch in Perl, obwohl es natürlich Aufgaben gibt, für die C besser geeignet ist als Perl und umgekehrt.

Wenn Perl alleine nicht gut genug ist, gibt es außerdem umfassende Archive mit vielerlei Tools und Modulen für diverse häufige Aufgaben (Module für Datenbanken, Interaktivität, Netzwerke, Verschlüsselung, Verbindungen zu anderen Sprachen etc.). Damit ergibt sich eine enorme Menge an Ressourcen, die nutzbar werden.

Perl ist flexibel

Unterschiedliche Programmierer haben unterschiedliche Methoden, sich Problemen zu nähern und sie zu lösen. Deshalb stellt Perl, anstatt zu verlangen, die Denkweise an einen kleinen Satz von Befehlen und Strukturen anzupassen, eine enorme Anzahl von Konstruktionen und Abkürzungen zur Verfügung - von denen viele letztlich genau das gleiche machen wie andere, nur auf eine etwas andere Art und Weise. Der Umfang von Perl und die Vielzahl seiner Möglichkeiten machen es extrem flexibel und angenehm im Gebrauch.

2.a das erste Perl Programm

Um in Perl ein Programm zu schreiben benötigt man nichts als einen einfachen Texteditor.

die erste Zeile

Im Gegensatz zu PHP wird in Perl der html Code in die Perl Datei geschrieben. Daher muss in der ersten Zeile deklariert sein, dass es sich um ein Perl Dokument handelt und wo der entsprechende Interpreter zu finden ist. Meist beginnt ein Perl Programm daher mit dieser Zeile:

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

Diese kann von System zu System unterschiedlich aussehen. [/usr/local/bin/perl] ist der hier gültige Pfad zum Perl Verzeichnis.

Kommentare und Anweisungen

Um sich später in einem Programm besser zu Recht finden zu können kann man im Programm Kommentare setzen. In Perl öffnet das Symbol # einen Kommentar. Alles zwischen # und dem Zeilenende wird vom Interpreter ignoriert (Ausnahme: erste Zeile). Kommentare können überall im Programm verwendet werden. Der einzige Weg um Kommentare über mehrere Zeilen ausdehnen zu können, ist die Verwendung von # in jeder Zeile. Alles Übrige sind Perl-Anweisungen, welche mit einem Strichpunkt beendet werden müssen.

einfache Ausgabe

Die `print` Funktion gibt Information aus. Um hiermit ein html Dokument ausgeben zu können muss vorher deklariert werden, dass es sich um ein solches handelt. Dies passiert mittels:

```
print „Content-Type: text/html\n\n“;
```

Diese Befehlszeile sagt dem Perl Interpreter, dass es sich um ein html bzw. Text- Dokument handelt und dass er es entsprechend ausgeben soll. Das Zeichen [n] stellt einen Zeilenumbruch dar. Perl ist fähig verschiedenste Dokumententypen auszugeben (z.B.: Bilder, Word Dokumente, Audiodokumente).

Mit diesen Informationen lässt sich bereits ein kleines Programm basteln. Da ein ungeschriebenes Gesetz in der Informatik lautet: Das erste Programm einer neuen Programmiersprache muss „Hallo Welt“ am Bildschirm ausgeben, will ich dieses hier demonstrieren:

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

print „Content-Type: text/html\n\n“;      # HTML Dokument
print „Hallo Welt“;                       # Ausgabe von Hallo Welt
```

2.b skalare Variablen

Die elementarsten Variablen in Perl sind die skalaren Variablen. Skalare Variablen können Zahlen und Strings beinhalten und, je nach Kontext, werden sie als Zahlen oder Strings interpretiert. Als Indikator einer skalaren Variable dient das Dollar Zeichen (\$). Hier wird der Variable „main“ die Zahl 5 bzw. der Text „Hallo“ zugeordnet. Es gibt eine Vielzahl an Möglichkeiten Text in Variablen zu speichern.

```
$main = 5;           # Zahl 5
$main = '5';        # Zahl 5 als String
$main = "Hallo";    # String Hallo
$main = qq~Hallo~;  # String Hallo anders zugeordnet
```

Das Semikolon (;) muss hier am Zeilenende stehen.

Variablenamen können aus Buchstaben, Zahlen und Unterstrich zusammengesetzt werden, sollten jedoch nicht mit einer Zahl beginnen. Die Variable \$_ hat eine spezielle Bedeutung. Sie ist die so genannte Standardvariable, die verwendet wird wenn keine anderen Parameter angegeben werden (weitere Spezialvariablen: \$! – Systemfehler, \$] – Perl Version, \$^O – Betriebssystem). Variablen müssen in Perl nicht extra deklariert werden. Sie werden, wie zum Beispiel unter Java, dann eingerichtet, wenn sie das erste Mal erwähnt werden.

Operationen und Zuweisungen

Perl verwendet alle gebräuchlichen C-Operatoren.

```
$a = 1 + 2;          # Addition: Addiere 1 und 2 = $a
$a = 3 - 4;         # Subtraktion: Subtrahiere 4 von 3 ...
$a = 5 * 6;        # Multiplikation: Multipliziere 5 und 6
$a = 7 / 8;        # Division: Dividiere 7 mit 8 ($a = 0.875)
$a = 9 ** 10;      # Neun hoch zehn
++$a; bzw. $a++;   # entspricht $a + 1
--$a; bzw. $a--;   # entspricht $a - 1
int($a);          # abrunden
sqrt($a);         # Wurzel
$a = $b;          # $a wird $b
$a .= $b;         # $a = „$a$b“
```

speziell für Strings

```
$a = $b . $c;      # Verkettung: $b und $c -> „$b$c“
$a = $b x 3;       # entsprich „$b$b$b“ -> Multiplikator
```

Ausgeben der Variablen

Um die Variablen nun nicht nur gespeichert zu haben, sondern auch auszugeben, müssen diese unter einem `print` Befehl stehen. Allerdings nicht mit einfachem Anführungszeichen, da sonst das Dollar Zeichen nicht mehr als Variable sondern als normales Zeichen interpretiert würde. Angenommen in der Variable `a` wäre die Zahl 5 gespeichert so würden diese Befehle die im Kommentar beschriebenen Ergebnisse erbringen.

```
print '$a';           # Ausgabe von -> $a
print "$a";          # Ausgabe von -> 5
print qq~$a~;        # Ausgabe von -> 5
```

Um unter einem `print` Befehl mit doppelten Anführungszeichen ein Dollarzeichen oder ein anderes besetztes Zeichen wie ein weiteres Anführungszeichen ausgeben zu können, muss vor diesem ein Backslash (\) stehen. Bsp.:

```
print "eine \"komische\" Sprache";      # eine "komische" Sprache
```

2.c Array Variablen

Eine weitere Art von Variablen sind die Array Variablen. Sie stellen eine Liste von skalaren Variablen dar (z.B.: Zahlen und Strings). Die Namen von Array-Variablen haben dasselbe Format, wie die skalaren Variablen, mit dem Unterschied, dass sie anstelle eines `$`-Symbols, durch ein `@`-Symbol definiert werden. Diese gewinnen besonders in `foreach`- Schleifen an Bedeutung, um mit jedem der zugeordneten Variablen einen Befehl auszuführen (mehr unter Schleifen). Beispiel:

```
@essen = ('pizza', 'apfel', 'birne');    # Liste von Elementen
```

Damit wird der Variable `@essen` eine Liste von drei Elementen zugewiesen.

Um auf ein bestimmtes Element des Arrays zugreifen zu können wird eine Indexnummer unter eckigen Klammern einer skalaren Variable mit gleichem Namen nachgestellt. Der Index beginnt bei 0 zu zählen. Der Wert von `$essen[2]` ist somit „birne“.

Array Zuweisung

```
@mehressen = ('ei', @essen, 'kaese'); # ei, pizza, apfel, birne, kaese
```

Hiermit können Elemente zu einem Array hinzugefügt werden. Ein anderer Weg um Elemente hinzuzufügen sieht folgendermaßen aus:

```
push(@essen, 'kaese');           # kease wird an @essen angefügt
push(@essen, @mehressen);        # @mehressen wird an @essen angefügt
```

Damit wird der String `kaese` als Element an das Ende des Arrays `@essen` angehängt, oder die Elemente vom Array `@mehressen` an `@essen` angehängt.

Mit `pop` kann das letzte Element von einer Liste entfernt werden (mit `shift` das erste Element). `pop` entfernt von der ursprünglichen Liste `@essen` das letzte Element (hier: `birne`). Der Rückgabewert ist das entfernte Element. `@essen` hat danach nur noch zwei Elemente:

```
$letzteselement = pop(@essen);    # Jetzt ist $letzteselement = 'birne'
```

Um die Anzahl der Elemente zu erfahren muss `@essen` einer skalaren Variable zugewiesen werden (ohne Anführungszeichen, ansonsten werden die Elemente selbst ausgegeben). Die Nummer des letzten Array Elements kann mit `$#essen` herausgefunden werden.

```
$count = @essen;                 # $count = 3;
$count = "@essen";               # $count = pizza apfel birne
$count = $#essen;                # $count = 2;
```

weitere Befehle:

```
sort @namen;           # sortiert alphabetisch
sort {$a <=> $b} @werte; # sortiert numerisch
@verkehrt = reverse @namen; # dreht die Reihenfolge um
@namen = qw(Christine Karl Doris); # erleichtert die Eingabe
```

2.d Hashes (assoziative Arrays)

Normale Arrays oder Listen erlauben den Zugriff auf bestimmte Elemente nur über die Elementnummern. Mit Perl können aber auch Arrays erzeugt werden, deren Elemente durch Strings referenziert werden. Diese Arrays nennt man *hashes*.

Um einen Hash zu definieren wird eine normale Klammer- Notation verwendet und der Arrayname hat ein vorangestelltes %-Zeichen. Im folgenden Beispiel wird jedem Namen ein Alter zugeordnet im Hash %alter.

```
%alter = ('Person1', 10, # Person1 ist 10 Jahre alt
          'Person2', 20, # Person2 ist 20 Jahre alt
          'Person3', 30, # Person3 ist 30 Jahre alt
          'Hund', '21 Hundejahre'); # ein Hund ist 21 HJ alt

$alter{'Person1'} = '10'; # ordnet Person1 das Alter 10 zu
```

Das Alter dieser Personen kann mit folgenden Variablen aufgerufen werden:

```
$alter{'Person1'}; # Ergibt 10
$alter{'Person2'}; # Ergibt 20
$alter{'Person3'}; # Ergibt 30
$alter{'Hund'}; # Ergibt "21 Hundejahre"
```

Ein Hash kann in einen normalen Array umgewandelt werden, indem er einfach einem normalen Array zugewiesen wird. Umgekehrt wird ein normaler Array in einen Hash umgewandelt.

Operatoren

Die Elemente von Hashes sind ungeordnet, aber man kann auf die Elemente der Reihe nach mit den Funktionen *keys* und *values* zugreifen. So können beispielsweise sämtliche Namen in ein anderes Array gespeichert werden.

```
@namen = keys %alter; # Person1, Person2, Person3, Hund
```

Die Funktion *keys* erzeugt eine Liste der Schlüsselwörter des Hashes, die Funktion *values* erzeugt eine Liste der Werte. Die Reihenfolge der beiden Listen ist gleich. Daneben gibt es noch eine Funktion *each*, welche einen Array mit zwei Elementen erzeugt, dem Schlüsselwort und dem Wert.

Umgebungs-Variablen

Programme, die unter Windows oder UNIX ausgeführt werden, haben Zugriff auf so genannte Umgebungsvariablen. In Perl sind diese im Hash %ENV definiert. Diese Fix-Variablen geben Auskunft über Server- Informationen (Server-Adresse, Domain-Namen, vollständiger Pfad...) sowie über Besucherinformationen (letzte- besuchte Seite, Programmname des Webbrowsers...).

```
$ENV{'SCRIPT_FILENAME'}; # vollständiger Pfad zu einer Datei + Dateiname
```

2.e Dateiverarbeitung

Perl macht es möglich Dateien entweder zum Bearbeiten oder als Erweiterung zum Script Code einzufügen. Für letzteren Gebrauch dient die Funktion *require*. Hiermit lassen sich Dateien einfügen, die weiteren Script Code enthalten (z.B.: Konfigurationsdateien mit gespeicherten Variablen).

```
require "infos.pl";      # fügt Datei infos.pl als weiteren Script Code an
```

Um eine Datei zum Weiterbearbeiten zu öffnen wird der Befehl *open* genutzt. Vorab muss ein Bearbeitungsmodus gewählt werden (Lesen, Schreiben, Anhängen). Geschlossen wird eine Datei mit dem Befehl *close*.

Im folgenden Beispiel enthält die Variable *\$file* den Pfad zur zu bearbeitenden Datei, *INFO* dient als Referenz (so genannte „filehandler“) um die Datei später bearbeiten oder schließen zu können. Zur besseren Übersichtlichkeit werden Großbuchstaben verwendet.

```
open(INFO, $file);      # Öffnet Datei zum Lesen
open(INFO, ">$file");    # Öffnet Datei zum Schreiben (überschreiben)
open(INFO, ">>$file");   # Öffnet Datei zum anhängen am Ende
open(INFO, "<$file");    # Öffnet Datei zum Lesen
```

Das Schreiben in der Datei funktioniert mittels *print* Befehl und entsprechender Referenz (Im Beispiel: *INFO*). *\n* steht für einen Umbruch, der in der Datei dann eingefügt wird.

```
print INFO "Schreibe Text in Datei\n";      # schreibt in die geöffnete
print INFO "Schreibe noch mehr\n";         # Datei mit Referenz INFO
```

Um eine Datei zwischenspeichern, kann diese in ein Array ausgelesen werden, getrennt durch Zeilenumbrüche. Würden einzelne skalare Variablen zum einlesen verwendet, würde Zeile für Zeile eingelesen werden (realisierbar mit *while* Schleife).

```
@zeilen = <INFO>;      # Einlesen einer Datei in den Array @zeilen
```

weitere Befehle:

```
close(INFO);          # File schließen
opendir($ordner);     # Ordner wird geöffnet
closedir($ordner);    # Ordner wird geschlossen
mkdir "neuerordner";  # erstellt einen neuen Ordner
chdir "$ordner";      # wechselt den aktuellen Ordner
@dateien = glob "*";  # sucht im aktuellen Ordner nach Dateien
rename "x.htm", "x.htm.alt"; # umbenennen
unlink "x.htm.alt";   # Datei löschen
if (-e $datei) {}     # prüft ob Datei vorhanden ist
if (-z $datei) {}     # prüft ob Datei leer ist
$size = -s $datei;    # gibt Größe der Datei aus
if (-f $datei) {}     # prüft ob es eine Datei ist
if (-d $datei) {}     # prüft ob es ein Ordner ist
$alter = -M $datei;   # Tage seit letzter Änderung
$zugriff = -A $datei; # Tage seit letztem Zugriff
```

2.f Schleifen + Kontrollstrukturen

Das eigentliche Arbeiten mit Programmen wird erst durch Schleifen und Kontrollstrukturen ermöglicht. Perl unterstützt viele verschiedene Arten von Kontrollstrukturen, welche zum Teil C ähnlich sehen, aber auch an Pascal erinnern.

Schleifen

Wenn ein Block von Anweisungen mehrfach abgearbeitet werden soll (z.B.: für verschiedene Werte), werden Schleifen benötigt. Schleifen durchlaufen einen Block Anweisungen und werden beendet, indem eine *Abbruchbedingung* bei jedem Schleifendurchlauf prüft. Fehlt eine Abbruchbedingung,

handelt es sich um eine *Endlosschleife*. Es gibt mehrere Arten von Schleifen. Man unterscheidet zwei Varianten:

Zählschleifen: Es ist schon vor dem Eintritt in die Schleife bekannt, wie häufig der Schleifenblock durchlaufen werden soll. Deswegen wird die Abbruchbedingung der Schleife durch einen Zähler realisiert, oder durch die Anzahl der Elemente eines Arrays gegeben.

Bedingte Schleifen: Vor Antritt des Schleifendurchlaufs wird die Abbruchbedingung getestet. Die Anzahl der Schleifendurchläufe ist vorerst unbekannt.

for (Zählschleifen)

In der for- Schleife wird anfangs eine Bedingung überprüft, dann der Schleifenblock ausgeführt, dann Bedingung neu geprüft. Dies läuft so lange bis die Bedingung nicht mehr wahr ist. Mit der folgenden for- Schleife werden die Zahlen 1 bis 9 ausgegeben. Man startet mit dem Wert $\$i = 0$, dann wird die Bedingung festgelegt, dass $\$i$ kleiner als 10 sein muss und solange die Bedingung wahr ist wird $\$i$ bei jedem Durchlauf um eins größer.

```
for ($i = 0; $i < 10; ++$i)    # Starte mit $i = 0
{                               # falls $i < 10 werden Aktionen ausgeführt
    print "$i\n";             # Aktion wird ausgeführt, und ^ $i + 1
}
```

foreach (Zählschleife)

Um jedes Element eines Arrays oder einer anderen Listenstruktur (wie zum Beispiel die Zeilen einer Datei) zu durchlaufen, wird bevorzugt die *foreach* Schleife verwendet. Im folgenden Beispiel wird jedes Element des Arrays *@namen* bei jedem Durchlauf der Schleife der Variable *\$name* zugeordnet, diese wird in der Schleife weiterverwendet (in diesem Fall mittels Funktion *print*). Die Schleife läuft so lange bis alle Elemente im Array durchlaufen wurden.

```
foreach $name (@namen)        # Gehe der Reihe nach durch
{                               # @namen und ordne das Element $name
    print "$name\n";           # Ausgabe des Elementes
}
```

Die Anweisungen (hier *print*), welche jedes Mal durchgeführt werden sollen, müssen in den geschweiften Klammern (Schleifenblock) angegeben werden.

Bool'sche Operatoren (Bedingung oder Abbruchbedingung)

Um zu testen ob eine Aussage wahr oder falsch ist wird ein Gleichnis benötigt, welches überprüft wird. Hierzu dienen die so genannten „Bool'schen Operatoren“. Falls die Bedingung erfüllt (oder nicht erfüllt) wird, wird der Teil in den geschweiften Klammern (Schleifenblock) ausgeführt.

```
$a == $b                       # ist die Zahl $a gleich der Zahl $b?
$a != $b                       # ist die Zahl $a ungleich der Zahl $b?
$a eq $b                       # ist $a gleicher String wie $b?
$a ne $b                       # ist $a und $b nicht der gleiche String?
```

Es können mehrere Bedingungen gleichzeitig geprüft werden, sie können einander ausschließen.

```
(BEDINGUNG1) && (BEDINGUNG2)    # BED1 und BED2 müssen eintreten
(BEDINGUNG1) || (BEDINGUNG2)    # BED1 oder BED2 müssen eintreten
```

Diese Bedingungen werden dann in *if*, *elsif*, *while*, *for* oder *until* Kontrollstrukturen verwendet.

while und until

Eine typische Schleife ist "while". Vor dem Eintreten in die Schleife wird eine Bedingung getestet. Solange die Bedingung *wahr* ergibt, wird der Schleifenblock ausgeführt. Ergibt die Bedingung *falsch*, bricht die Schleife ab. Der einzige Unterschied zur *for*- Schleife ist, dass hier keine Vor- bzw. Durchlaufaktion angegeben werden. In diesem Beispiel wird der Variable `$a` der Wert 0 zugeordnet. Die Schleife läuft so lange `$a` kleiner als 10 ist. Um eine Endlosschleife zu vermeiden muss `$a` daher vergrößert werden, dies geschieht im Schleifenblock.

```
$a = 0;                # $a bekommt den Wert 0
while ($a < 10)       # Während $a kleiner als 10 ist
{
    print $a;         # führe Aktionen aus
    $a++;             # drucke $a
                    # vergrößere $a um 1
}

while (<DATA>) { .. } # Inhalt einer Datei wird Zeilenweise gelesen
```

Zusätzlich bietet Perl eine Negation dieser Schleifenkonstruktion an, die *until*- Schleife. Die *until*-Schleife wird in genau gleicher Weise verwendet, im Gegensatz zu *while* wird der Block solange ausgeführt, *bis* der Test wahr ist und nicht solange er wahr ist. Eine weitere Schleifenkonstruktion wäre *do...until*, welche ähnlich wie *until* arbeitet.

if, elsif, else, unless

Um bei Eintreten einer bestimmten Bedingung eine Aktion auszuführen werden *if*, *elsif*, *else*, *unless* verwendet. Hier kommen die Bool'schen Operatoren zum Einsatz. Nach Kontrollstrukturenbeginn sowie nach öffnen und schließen der geschweiften Klammern darf kein Semikolon am Schluss stehen (gleiches gilt für alle Schleifen).

```
if ($a == $b)        # Falls Zahl $a gleich Zahl $b ist
{
    print $a;        # führe folgende Aktionen aus
                    # drucke $a
                    #
}
```

Falls eine *if*- Schleife nicht eintritt können noch unendlich viele Nachsätze daran gehängt werden, die eine andere Bedingung überprüfen mit der Voraussetzung, dass die erste Bedingung nicht gegeben war.

```
elsif ($a == $c)     # möglicher Nachsatz zur ersten Kontrollbedingung
{
    print $a;        # Falls Zahl $a gleich Zahl $c ist
                    # drucke $a
                    #
}
```

Falls nun keine Vorlauf- Bedingung erfüllt wird, kann eine Kontrollfunktion ohne Bedingung, *else*, angehängt werden.

```
else                 # alle vorigen Bedingungen treten nicht ein
{
    print $b;        #
                    # drucke $b
                    #
}
```

Der gleich angewandte Gegensatz zu *if* ist *unless* oder *if not*. *unless* drückt das Gegenteil aus, das heißt, dass die genannte Bedingung nicht eintreten darf um die nachfolgenden Aktionen ausführen zu können.

2.g Mustererkennung + Substitution

Die Mustererkennung ist eine berühmte und berüchtigte Fähigkeit von Perl. Berühmt, weil damit viel Programmierarbeit erspart bleibt, berüchtigt, weil viele Sonderzeichen verwendet werden müssen. Der

Motor der Mustererkennung sind die regulären Ausdrücke (RA), welche von vielen UNIX Hilfsprogrammen verwendet werden.

Reguläre Ausdrücke

Ein *regulärer Ausdruck* steht zwischen zwei Schrägstrichen '/' und der Mustererkennungs-Operator ist =~. Der folgende Ausdruck ist wahr, falls der String (oder das Muster, oder der reguläre Ausdruck) „das“ in der Variable \$satz vorkommt.

```
$satz =~ /das/;
```

Effektiv ist der "generische Musteroperator" m//. Die Schrägstriche können mit der m-Notation durch ein beliebiges anderes Zeichen ersetzt werden. Ohne m-Notation braucht es die Schrägstriche.

Mittels Mustererkennung lässt sich folgende Schleife verwenden:

```
if ($satz =~ /das/) {  
    print "Das Muster 'das' kommt vor.\n";  
}
```

welche eine Nachricht ausgibt, falls wir einen der folgenden Strings hätten:

```
$satz = "das Muster ohne Wert";  
$satz = "sodass";
```

Modifiers

Am Ende des match- Befehls kann noch ein Buchstabe stehen, der diesen Befehl modifiziert. Sie können kombiniert und sowohl für Erkennung als auch für Substitution angewandt werden.

```
$satz =~ /das/i; # Groß und Kleinschreibung werden nicht unterschieden  
$satz =~ /das/m; # Zeichenkette wird mehrzeilig betrachtet  
$satz =~ /das/g; # mehrmals anwenden, falls das Muster mehrmals vorkommt  
$satz =~ /das/s; # \n wie ein normales Zeichen behandeln
```

Klassen

Zwischen eckigen Klammern [] kann eine Auswahl von Zeichen angegeben werden.

```
$satz =~ /[das]/i # die Buchstaben d a und s matchen  
$satz =~ /^[^das]/i # die Buchstaben d a und s matchen nicht
```

Mehr über RAs

RAs können sehr viel mehr als einen reinen Vergleich von Zeichenketten. Es gibt viele Spezialzeichen, die eine bestimmte Bedeutung haben. Mit diesen Spezialzeichen wird erst die volle Funktionalität (und auch die Komplexität) der RA's erreicht.

Als Beispiel möchte man wissen, ob in einem String die Zeichenfolge *ac* vorkommt. Der RA dazu ist */ac/*. Anstelle von *ac* möchte man noch *bc* zulassen, dann heißt der RA */[ab]c/*. Falls nun mehrere *a*'s und *b*'s vor dem *c* erlaubt sein sollen, wird verwendet: */[ab]+c/*. Folgende Strings wären damit erlaubt:

```
ac; bc; aac; bc; abc; bac; aaac
```

Wäre anstelle des *+*-Zeichens ein *** wäre ein *c* alleine auch erlaubt. Wenn man am Anfang sicher ein *a* will, muss man */a[ab]*c/* schreiben.

Um anstelle der Zeichenfolge wird nach Zahlen zu suchen wird ein Spezialzeichen `\d` (Gegensatz `\D` - > keine Zahlen), was nichts anderes bedeutet als `[0123456789]` oder in Kurzform `[0-9]` verwendet. Diese drei Darstellungen sind äquivalent. Angenommen, am Anfang des Strings wäre eine Zahl anschließend ein Leerzeichen. Der RA hierzu heißt `/\^d+ /`. Es könnte aber auch sein, dass anstelle eines Leerzeichens auch ein Tabulator oder ein Umbruch steht, dazu gibt es wiederum ein Spezialzeichen, das `\s`. Also heißt der RA `/\^d+\s/`.

weitere Spezialzeichen:

```
.      # steht für genau ein Zeichen
^      # Zeilen- oder Stringanfang
$      # Zeilen- oder Stringende
*      # Null oder mehrere Male den letzten Buchstaben!
*?     # dito, aber minimal
+      # Ein oder mehrere Male den letzten Buchstaben
?      # Null oder ein Mal den letzten Buchstaben
```

Ein senkrechter Strich `|` bedeutet „entweder oder“ und Klammern `(...)` werden verwendet, um Dinge zu gruppieren:

```
Bus|Auto      # Entweder Bus oder Auto
(Ha|Ma)us    # Entweder Haus oder Maus
(da)+        # Entweder da oder dada oder dadada oder...
```

Spezialzeichen:

```
\n          # Zeilenumbruch
\t          # Tabulator
\w          # irgendwelche Buchstaben
\W          # alles nur keine Buchstaben
\d          # eine Zahl
\D          # keine Zahl
\s          # Leerzeichen oder Tabulator
\S          # kein Leerzeichen, keine Tabulator
```

Zeichen, wie `$`, `|`, `[`, `]`, `\`, `/` sind Spezialfälle in RA's. Falls sie als normale Zeichen verwendet werden sollen, muss ihnen ein Backslash vorstehen. z.B.: `^\$+`

2.h Mehr String Funktionen

Substitution

Sowie Perl Muster in Strings erkennen kann, können Muster durch Strings ersetzt werden. Dazu wird die *s-Funktion* verwendet. Es gelten dieselben Möglichkeiten und Regeln. Um den Substring `london` durch `London` in der Variablen `$satz` zu ersetzen, verwenden wir die Anweisung

```
$satz =~ s/london/London/;          # london -> London
$satz =~ s~\[i\](.+?)\[\/i\]~<i>$1</i>~isg;  # (.+?) = irgendwas -> $1
```

Die Funktion `tr` ermöglicht die Übersetzung von einzelnen Buchstaben und wird gleich wie `s/` oder `m/` angewendet.

split-Funktion

Eine sehr nützliche Funktion in Perl ist *split*. Sie unterteilt einen String an definierten Stellen und kopiert die Teile in einen Array. Unterteilt wird durch ein angegebenes Teilzeichen, wenn dieses im String vorhanden ist, wird an dieser Stelle ein neues Element in den Array gespeichert, das

Teilzeichen wird gelöscht. Ein String kann weiters in mehrere Einzelstrings unterteilt werden, mit gleichen RAs wie oben. Im folgenden Beispiel ist der Doppelpunkt das Teilzeichen.

```
$info = "Schokolade:Gurke:Schinken ";           # Vorgabestring
@teile = split(/:/, $info);                     # Bsp.: $teile[0] = Schokolade
($sueses,$gemuese,$fleisch) = split(/:/, $info);# Bsp.: $fleisch = Schinken
```

substr- Funktion

Die *substr*- Funktion dient dazu Text zu kürzen, einen Abschnitt anhand von bestimmten Grenzen rauszupicken, oder einen Abschnitt anhand von bestimmten Grenzen zu ersetzen.

```
substr("Hallo Welt", 3, 4); # Ergebnis: "lo W" -> Begin 3, 4 Zeichen lang
substr("Hallo Welt", 7);   # Ergebnis: "elt" -> Begin 7, bis zum Ende
substr("Hallo Welt", -6, 4);# Ergebnis: " Wel" -> Begin -6, 4 Zeichen lang
$str = "Hallo Welt";
substr($str, 7, 4) = "Stadt"; # Ergebnis: "Hallo Stadt"
substr($str, 6, 0) = "Perl "; # Ergebnis: "Hallo Perl Stadt"
```

2.i Subroutinen

Wie jede gute Programmiersprache können in Perl eigene Funktionen programmiert werden. Man nennt sie in Perl *Subroutinen*. Sie können irgendwo im Programm platziert werden und werden bei Bedarf aufgerufen.

```
sub namederroutine           # sub + Name der Subroutine
{                             # Inhalt der Funktion
    print "Dies ist eine Subroutine. Ist $_[0] und $_[1] da?\n";
}
```

Folgenden Anweisungen rufen diese Subroutine auf.

```
&namederroutine;           # Aufruf ohne Parameter
&namederroutine ("Peter", "Franz"); # Aufruf mit Parameter
```

2.j Essentielles

Hier werden noch einige unverzichtbare Funktionen vorgestellt.

$\$r$: numerischer Ausdruck, ganzzahlig oder mit Kommaanteil

$\$s$: Zeichenkettenausdruck

```
abs($r); # Absolutwert von $r
cos($r); # Cosinus von $r
exp($r); # Exponentialfunktionen
log($r); # natürlicher Logarithmus von $r
sin($r); # Sinus von $r
chop($s); # das letzte Zeichen von $s wird entfernt
chomp($s); # Abschneiden des Eingabetrennzeichens in $s (z.B.: Umbruch)
length($s); # Länge von $s
```

```
die "Fehler"; # bricht das Programm mit nachfolgender Meldung ab
warn "Warnung"; # gibt eine Warnung mit nachfolgender Meldung aus
exit; # kompletter Ausstieg aus dem Programm
last; # Ausstieg aus einer Schleife
```

und viele andere...

3.Nachwort

Referenz

Als Referenz, die alle oben genannten und noch einige Funktionen mehr verwendet, soll ein von mir programmiertes und betreutes Script verwendet werden. Das i-dreams.net Gästebuch. Es ist mein erstes eigenes Perl Script und wird bereits über ein Jahr lang entwickelt. Es findet im Internet großen Anklang und wird auf tausenden Internetseiten verwendet.

Zusammenfassung

Um mit Perl annähernd gut arbeiten zu können ist genügend Praxis eine zwingende Voraussetzung. Während einer Praxiszeit eignet man sich Routine an mit seinen bevorzugten Funktionen zu arbeiten und diese nach bestem Wissen einzusetzen, da es nicht nur einen Weg zur Problemlösung gibt.

Diese Arbeit soll nur eine kleine Einführung in die Programmiersprache Perl darstellen. In diesem Text wurden nur die absoluten Grundlagen erörtert, die benötigt werden, um erste kleine Experimente durchzuführen. Leider gibt es viel zu viele Funktionen um sie alle hier auflisten zu können, die nach meinem Gebrauch Wichtigsten wurden jedoch kurz erklärt. Dennoch wurde selbst der Funktionsumfang dieser Funktionen noch nicht ausreichend geschildert. Hier wurde weder auf die Konfiguration, noch auf die Handhabung von Perl Scripten eingegangen, auch fehlen viele wichtige Themen, wie das arbeiten mit Mails, mit Zeit sowie andere Notwendigkeiten. Eine Vielzahl an Modulen erweitert die Funktionalität noch um ein großes Stück um alle erdenklichen Aufgaben zu lösen.